Proofs in Satisfiability Modulo Theories

Clark Barrett[1], Leonardo de Moura[2], and Pascal Fontaine[3]

[1] New York University
barrett@cs.nyu.edu
[2] Microsoft Research
leonardo@microsoft.com
[3] University of Lorraine and INRIA
pascal.fontaine@inria.fr

1 Introduction

Satisfiability Modulo Theories (SMT) solvers[4] check the satisfiability of first-order formulas written in a language containing interpreted predicates and functions. These interpreted symbols are defined either by first-order axioms (e.g. the axioms of equality, or array axioms for operators `read` and `write`,...) or by a structure (e.g. the integer numbers equipped with constants, addition, equality, and inequalities). Theories frequently implemented within SMT solvers include the empty theory (a.k.a. the theory of uninterpreted symbols with equality), linear arithmetic on integers and/or reals, bit-vectors, and the theory of arrays. A very small example of an input formula for an SMT solver is

$$a \leq b \wedge b \leq a + x \wedge x = 0 \wedge \big[f(a) \neq f(b) \vee (q(a) \wedge \neg q(b + x))\big]. \qquad (1)$$

The above formula uses atoms over a language of equality, linear arithmetic, and uninterpreted symbols (q and f) within some Boolean combination. The SMT-LIB language (currently in version 2.0 [4]) is a standard concrete input language for SMT solvers. Figure 1 presents the above example formula in this format.

SMT solvers were originally designed as decision procedures for decidable quantifier-free fragments, but many SMT solvers additionally tackle quantifiers, and some contain decision procedures for certain decidable quantified fragments (see e.g. [26]). For these solvers, refutational completeness for first-order logic (with equality but without further interpreted symbols) is an explicit goal. Also, some SMT solvers now deal with theories that are undecidable even in the quantifier-free case, for instance non-linear arithmetic on integers [13].

In some aspects, and also in their implementation, SMT solvers can be seen as extensions of propositional satisfiability (SAT) solvers to more expressive languages. They lift the efficiency of SAT solvers to richer logics: state-of-the-art SMT solvers are able to deal with very large formulas, containing thousands of atoms. Very schematically, an SMT solver abstracts its input to propositional logic by replacing every atom with a fresh proposition, e.g., for the above example,

$$p_{a \leq b} \wedge p_{b \leq a+x} \wedge p_{x=0} \wedge \big[\neg p_{f(a)=f(b)} \vee (p_{q(a)} \wedge \neg p_{q(b+x)})\big].$$

[4] We refer to [3] for a survey on SMT.

```
(set-logic QF_UFLRA)
(set-info :source | Example formula in SMT-LIB 2.0 |)
(set-info :smt-lib-version 2.0)
(declare-fun f (Real) Real)
(declare-fun q (Real) Bool)
(declare-fun a () Real)
(declare-fun b () Real)
(declare-fun x () Real)
(assert (and (<= a b) (<= b (+ a x)) (= x 0)
             (or (not (= (f a) (f b))) (and (q a) (not (q (+ b x)))))))
(check-sat)
(exit)
```

Fig. 1. Formula 1, presented in the SMT-LIB 2.0 language

The underlying SAT solver is used to provide Boolean models for this abstraction, e.g.

$$\{p_{a\leq b}, p_{b\leq a+x}, p_{x=0}, \neg p_{f(a)=f(b)}\}$$

and the theory reasoner repeatedly refutes these models and refines the Boolean abstraction by adding new clauses (in this case $\neg p_{a\leq b} \vee \neg p_{b\leq a+x} \vee \neg p_{x=0} \vee p_{f(a)=f(b)}$) until either the theory reasoner agrees with the model found by the SAT solver, or the propositional abstraction is refined to an unsatisfiable formula. As a consequence, propositional reasoning and theory reasoning are quite well distinguished. Naturally, the interaction between the theory reasoning and the SAT reasoning is in practice much more subtle than the above naive description, but even when advanced techniques (e.g. online decision procedures and theory propagation, see again [3]) are used, propositional and theory reasoning are not very strongly mixed. SMT proofs are thus also an interleaving of SAT proofs and theory reasoning proofs. The SAT proofs are typically based on some form of resolution, whereas the theory reasoning is used to justify theory lemmas, i.e. the new clauses added from the theory reasoner. The entire proof ends up having the shape of a Boolean resolution tree, each of whose leaves is a clause. These clauses are either derived from the input formula or are theory lemmas, each of which must be justified by theory-specific reasoning. The main challenge of proof production is to collect and keep enough information to produce proofs, without hurting efficiency too much.

The paper is organized as follows. Section 2 discusses the specifics of what is required to implement proof-production in SMT along with several challenges. Next, Section 3 gives a historical overview of the work on producing proofs in SMT. This is followed by a discussion of proof formats in Section 4. We then look at (in Sections 5, 6, and 7) the particular approaches taken by CVC4, veriT, and Z3. Section 8 includes a discussion of the new *Lean* prover, which attempts to bridge the gap between SMT reasoning and proof assistants more directly by building a proof assistant with efficient and sophisticated built-in

SMT capabilities. Finally, Section 9 discusses current applications of SMT proofs, and then Section 10 concludes.

2 Implementing Proof-Production in SMT

Since the core of SMT solvers is mainly a SAT solver adapted for the needs of SMT, the core of the proof is also a SAT proof. We will refer to the other article by Marijn Heule, in the same volume, for more details on SAT solving. Let us just recall that SAT solvers work on a conjunctive normal form, and build a (partial) assignment using Boolean propagations and decisions. If building this assignment fails (i.e. it falsifies a clause) because of some decisions, the algorithm analyses the conflict, learns a new clause that will serve as a guide to avoid repeating the same wrong decisions again, and backtracks. Only this learning phase is important for proofs, and it is easy to generate a resolution trace corresponding to this conflict analysis and production of the learned clause. Proof-producing SMT solvers rely on the underlying SAT solver to produce this resolution proof.

In the context of SAT solving, a conflict is always due to the assignment falsifying a clause. Another kind of conflict can occur in SMT: assume the SAT solver assigned $p_{a=b}$ (i.e. the propositional abstraction of the atom $a = b$) to true, and later $p_{f(a)=f(b)}$ to false. There may be no clause conflicting with such an assignment, but it is obviously inconsistent on the theory level. The theory reasoner embedded in the SMT solver checks assignments for theory consistency. In the present case, it would notify a conflict to the underlying SAT engine, and would produce the clause $\neg p_{a=b} \vee p_{f(a)=f(b)}$. This *theory lemma* refines the propositional abstraction with some knowledge from the theory. This knowledge can then be used like any other clause. If theory reasoners provide detailed proofs for theory lemmas, the SMT proof is just the interleaving of these theory proofs and the resolution proofs generated by the SAT solver.

2.1 Proofs of Theory Lemmas

Simple examples of theory reasoners include decision procedures for uninterpreted symbols and linear arithmetic. The satisifiability of sets of ground literals with equalities and uninterpreted symbols can be checked using congruence closure [37–39]. Efficient algorithms are non-trivial, but the idea of congruence-closure algorithms is simple. They build a partition of all relevant terms, according to the equalities, and check for inconsistencies with negated equalities. For instance, if the relevant terms are a, b, $f(a)$ and $f(b)$, the initial partition would be $\big\{\{a\},\{b\},\{f(a)\},\{f(b)\}\big\}$. If an equality $a = b$ is asserted, the algorithm would merge the classes for a and b, because of this equality, and would also merge the classes for $f(a)$ and $f(b)$ because it merged the arguments of f and because equality is a congruence. The resulting partition would be $\big\{\{a,b\},\{f(a),f(b)\}\big\}$. If the algorithm is aware that $f(a) \neq f(b)$ should hold, it detects a conflict as soon as $f(a)$ and $f(b)$ are put in the same congruence class. Producing proofs for congruence closure is thus simply a matter of keeping a trace of which (two)

terms are merged and why (i.e. either because of a congruence, or an asserted equality). This information can be stored in an acyclic undirected[5] graph, where terms are nodes, and edges represent asserted equalities or congruences. If the equality of two terms can be deduced by transitivity using asserted equalities and congruences, then there is a (unique) path between the two terms in the graph. To prove a congruence (e.g. $g(a, c) = g(b, d)$), it suffices to first prove the equality of the corresponding arguments ($a = b$, $c = d$) and use an instance of the congruence axiom schema, e.g. $(a = b \wedge c = d) \Rightarrow g(a, c) = g(b, d)$. Proof steps (i.e. an equality being the consequence of transitivity, and a congruence instance) are again simply connected using Boolean resolution.

Another very important theory supported by many SMT solvers is linear arithmetic on reals. Decision procedures for this theory are often based on a specialized simplex algorithm [20, 32]. The algorithm maintains (a) a set of linear equalities, (b) upper and lower bounds on the variables, and (c) an assignment. The bounds and the linear equalities come directly from asserted literals; new variables are introduced so that bounds only apply on variables and not on more complicated linear combinations. Starting from an assignment satisfying all equalities, the procedure tries to progressively satisfy all bounds on the variables. During the process, the linear equalities are combined linearly to produce new sets of equalities, always equivalent to the original set. If the right strategies are used, the procedure is guaranteed to terminate either with an assignment satisfying all equalities and bounds, or an assignment that causes a specific linear equality to be in contradiction with the bounds on its variables. It then suffices to collect the constraints corresponding to the equality and the bounds to derive a contradiction. In fact, the coefficients of the variables in the linear equality correspond to the coefficients of the linear combination related to Farkas' lemma (roughly, this lemma states that a set of linear inequalities is inconsistent if and only if there is a linear combination of those inequalities that reduces to $1 \leq 0$; see e.g. [10]).

The above procedure only applies to reals. Various techniques are used to adapt it for mixed real and integer linear arithmetic. One is branching: if, in the real algorithm, an integer variable x has a rational value (e.g. 1.5), the procedure may issue a new lemma stating $x \leq 1 \vee x \geq 2$. Notice that this lemma is a tautology of arithmetic. Another technique is to introduce cuts; cuts basically remove from the set of real-feasible solutions a subset with no integer solution. Very schematically, cuts are linear combinations of already available constraints, strengthened using the fact that variables are integer, e.g. $x + y \leq 1.5$ can be strengthened to $x + y \leq 1$ if x and y are integer variables.

Other theory reasoning engines found in modern solvers include modules for reasoning about arrays, inductive data types, bit-vectors, strings, and non-linear arithmetic. Their internals may vary a lot from one solver to another, and we will not consider them in more detail here. For some non-linear arithmetic techniques

[5] For efficiency reasons, modern congruence closure alogrithms actually use a directed graph [39].

(see e.g. [10]) like cylindrical algebraic decomposition or virtual substitution, it is not even clear how to produce useful certificates.

The theory reasoner may be a decision procedure for a single theory, but it is often a combination of decision procedures for multiple theories. Combination frameworks [36, 51] decide sets of literals mixing interpreted symbols from several decidable languages (like the aforementioned linear arithmetic and uninterpreted symbols) by combining the decision procedures for the individual theories into one decision procedure for the union of languages. Combining the decision procedures is possible when the theories in the combination satisfy certain properties. A sufficient condition, for instance, is that the theories be *stably-infinite*[6] and disjoint; this is notably the case for the combination of linear arithmetic with uninterpreted symbols. Combining theories also involves guessing which shared terms are equal and which are not, or, equivalently, communicating disjunctions of equalities between decision procedures. In order for a formula to be *unsatisfiable*, each possible equivalence relation over the shared terms must be ruled out. This typically is done by producing many individual theory lemmas, which, taken together, rule out all of the equivalence relations. Proofs for combinations of theories are thus built from proofs of the theory lemmas in the individual theories, and the component proofs are simply combined using Boolean reasoning rules.

2.2 Challenging Aspects of Proof Production

Quantifier reasoning in SMT is typically done through Skolemization when possible, and heuristic instantiation otherwise [25, 18]. Proofs involving instantiations are easily obtained by augmenting ground proofs with applications of a simple instantiation rule. Formulas like $\forall \mathbf{x}\, \varphi(\mathbf{x}) \Rightarrow \varphi(\mathbf{t})$ — where $\mathbf{x}$ is a vector of variables and $\mathbf{t}$ is a vector of terms with the same length — are first-order tautologies and can be provided as lemmas for building proofs.

Skolemization, however, is not trivial. This is because Skolemization is typically implemented by introducing a new uninterpreted constant or function, but such a step is not locally sound. It is only when looking at a global view of the proof, noting that the conclusion does not involve any introduced symbols, that soundness can be shown. Proofs involving Skolemization must therefore track the introduced constants and ensure they are not used in the conclusion of the proof. Furthermore, for efficiency, the solver may resort to advanced techniques for Skolemization that are even less amenable to proofs.

Another challenge is *preprocessing*. SMT solvers typically include a module for preprocessing formulas to simplify them before running the main algorithm to search for a satisfying assignment. This module may perform anything from simple rewrites (such as rewriting $x = y$ to $y = x$) to complex global simplifications that significantly change the structure of the formula. At the end of preprocessing, the formula is also typically converted into conjunctive normal

[6] A theory is stably-infinite if every satisfiable set of literals in the theory has an infinite model.

form (CNF). The CNF clauses form leaves in the ultimate resolution proof. Thus, in order to justify one of these clauses, a proof must capture each preprocessing step that was used to derive it from the original formula. Such bookkeeping is tedious and error-prone. One strategy is to disable complicated preprocessing techniques during proof-production. Still, SMT solvers rely on some minimal preprocessing for correct functioning, so in order to generate complete proofs, at least this preprocessing has to be recorded as part of the proof.

3 A Short History of Proofs in SMT

The Cooperating Validity Checker (CVC) [46] was the first SMT solver to tackle the problem of proof-production. CVC was built by Aaron Stump and Clark Barrett, advised by David Dill, at Stanford University. CVC was designed to improve upon and replace the Stanford Validity Checker (SVC) [6] for use in the group's verification applications and also to serve as a research platform for studying SMT ideas and algorithms. CVC uses a proof format based on the Edinburgh Logical Framework (LF) [27] and its proofs can be checked using the flea proof-checker [47, 48], which was developed concurrently with CVC.

The motivation for producing proofs was primarliy that it would provide a means of independently certifying a correct result, so that users would not have to rely on the correctness of a large and frequently-changing code base. Another motivation was the hope that proof-production could aid in finding and correcting nasty bugs in CVC. In retrospect, however, the most important and lasting contribution of CVC's proof infrastructure was that it enabled an efficient integration of a SAT solver for Boolean reasoning. Initial attempts to use a SAT solver within CVC suffered from poor performance because the theory-reasoning part of CVC was being used as a black box, simply flagging a particular branch of the SAT search as unsatisfiable. Without additional information (i.e. a conflict clause specifying a small reason for the unsatisfiability), the SAT solver could not benefit from clause learning or non-chronlogical backjumping and frequently failed to terminate, even on relatively easy problems. During a conversation with Cormaq Flanagan, the idea of using the proof infrastructure to compute conflict clauses emerged, and this was the key to finally making the integration with SAT efficient (see [7] for more details). The technique was implemented in CVC, as well as in Flanagan's solver called Verifun [22] and this laid the groundwork for the so-called DPLL(T) architecture [40] used in nearly every modern SMT solver.

The next important development in proof-producing SMT solvers was the early exploration of using proofs to communicate with skeptical proof assistants. The first work in this area [33] was a translator designed to import proofs from CVC Lite [2] (the successor to CVC) into HOL Lite [28], a proof assistant for higher-order logic with a small trusted core set of rules. The goals of this work were two-fold: to provide access to efficient decision procedures within HOL Lite and to enable the use of HOL Lite as a proof-checker for CVC Lite. Shortly thereafter [23, 30], a similar effort was made to integrate the haRVey SMT solver [19]

with the Isabelle/HOL prove assistant [41]. These early efforts demonstrated the feasibility of such integrations.

In 2008, an effort was made to leverage this work to certify that the benchmarks in the SMT-LIB library were correctly labeled (benchmarks are labeled with a *status* that can be "sat", "unsat", or "unknown", indicating the expected result when solving the benchmark). The certification was done using CVC3 [5] (the successor to CVC Lite) and HOL Lite [24]. Many of the benchmarks in the library were certified and additionally, a bug in the library was found as two benchmarks that had been labeled satisfiable were certified as unsatisfiable. The same work reports briefly on an anecdote that further validates the value of proof-production. A latent bug in CVC3 was revealed during the 2007 SMT-COMP competition. Using HOL Light as a proof-checker, the cause of the bug was quickly detected as a faulty proof rule. The bug in the proof rule had persisted for years and would have been very difficult to detect without the aid of the proof-checker.

The same year also marked the appearance of proof-production capabilities in several additional SMT solvers. Fx7 was a solver written by Michał Moskal which emphasized quantified reasoning and fast proof-checking [35]. MathSAT4 used an internal proof engine to enable the generation of unsatisfiable cores and interpolants [15]. Z3 began supporting proof-production as well [16], though its use of a single rule for theory lemmas meant that checking or reconstructing Z3 proofs requires the external checker to have some automated reasoning capability (see Section 7). Finally, development on veriT, the successor to haRVey, began in earnest [14], with proof-production as a primary goal (see Section 6). Another important community development around this time was an attempt to converge on a standard proof format for SMT-LIB. We discuss this effort in Section 4 below.

At the time of this writing, the state of proofs in SMT are in flux. A few groups are making a serious effort to develop solvers (e.g. CVC4 and veriT) capable of producing self-contained, independently-checkable proofs, though these are in progress and no standard format has been agreed upon. Other solvers (like Z3) produce a trace of deduction steps, but reconstructing a full proof from these steps requires additional search in some cases. Still other solvers (e.g. MathSAT, SMTInterpol), use proof technology primarily to drive additional solver functionality, like computing explanations, interpolants, or unsatisfiable cores. In this paper, we focus on tools that produce proofs with the goal of having them be checked or translated by an external tool.

4 Proof Formats for SMT

The SMT-LIB initiative has produced a series of documents standardizing formats for theories, logics, models, and benchmarks. An early goal of the initiative was to produce a standard format for proofs as well. However, this proved to be challenging, primarily because each solver implements its decision procedures in a different way, meaning the proof rules for each solver could vary significantly.

Another challenge is that it is difficult to formally express the wide variety of rules needed to cover the decision procedures used in SMT solvers using existing proof languages (such as the Edinburgh Logical Framework, LF [27]).

Probably the most ambitious effort to develop a standard proof format was the one led by Aaron Stump [42, 49] (at the time, a leader of the SMT-LIB initiative), who developed a language called LFSC, modeled after LF, but able to express more complicated rules by expressing side-conditions using pieces of trusted code in a small custom programming language. LFSC allows a user to define a set of arbitrary inference rules and then use these to construct a proof. LFSC was used as the proof language for the prototype proof-producing SMT solver CLSAT and was also integrated as an alternative proof-language in CVC3 [50]. LFSC is also the proof language used in CVC4 (see Section 5 for an example).

In 2011, the first Workshop on Proof Exchange for Theorem Proving (PxTP) was held, collocated with the 23rd Conference on Automated Deduction (CADE) in Wrocław, Poland. The topic of proof formats featured heavily, with several papers on the subject. Besson, Fontaine, and Théry proposed a proof format for SMT based on the syntax of SMT-LIB [8]. This later formed the basis for the veriT proof format (see Section 6). Böhme and Weber made a plea from a user's perspective for proof formats to be clear, well-documented, and complete (i.e. not requiring any search to replay or check) [12].

Unfortunately, none of the proposed formats have caught on as a general proof language for SMT-LIB. One reason for this is that there are not nearly as many consumers of SMT proofs as there are of SMT technology generally, so the proof format has not been prioritized. Another is that there are differences of opinion on what features should be included in such a standard. And another is that the few tools that produce proofs have considerable momentum behind their own formats and shifting to a new format would require a lot of work. However, there are clear benefits to a standard format, especially for proof exchange between SMT solvers and other tools, so we expect this will continue to be a subject of debate and research.

5 The CVC Family of SMT Solvers

The original CVC tool used a proof system based on LF. Each deductive step in the tool had a parallel proof-production step. CVC had its own internal Boolean engine which recorded proofs. Proofs were not available when using a SAT solver for Boolean reasoning. In CVC Lite and CVC3, a different design decision was made: a special *Theorem* class was created to hold all derived facts. Theorem classes could only be constructed using special proof-rule functions. This allowed us to isolate all of the trusted code in the system in a few proof-rule files. This approach, while elegant, turned out to not be very efficient. In particular, many deductive steps were not needed in the final proof, but their proofs were recorded anyway. More significantly, there was no way to turn proof production off completely.

In CVC4, we opted for a new model in which proof production is done much more efficiently. First, all proof code is protected under a compile-time flag. By disabling this flag, CVC4 can be built with no proof code at all (for maximum efficiency). When proofs are compiled in, they are generated lazily: only a minimal amount of bookkeeping is done in the SAT solver and no bookkeeping at all is done in the theory solvers.[7] If the user asks for a proof, the SAT solver bookkeeping is leveraged to produce the resolution part of the proof. Proofs of the needed theory lemmas used in the resolution proof are then generated by *re-running* the theory decision procedures in proof-producing mode. In proof-producing mode, theory decision procedures log their deductive steps in a speical *proof manager* object which then stitches all of the pieces together into a full proof.

As mentioned above, the proof format used by CVC4 is LFSC. A full description of LFSC is well beyond the scope of this paper, but a good introduction can be found in [50]. Figure 2 shows an LFSC proof for the example from Section 1. The first section states what is being proved: given real numbers a, b, and x, function f and predicate q, together with four proofs each justifying one of the four conjuncts of Formula 1, it produces a proof of the empty clause (represented by `cln`). The next section assigns names to different formulas appearing in the proof (each formula gets both a propositional name, starting with v and a logical atom name, starting with a). Next, rules are given for converting Formula 1 into clauses in CNF, each of which is assigned a label starting with C. Next, a number of theory derivations are given, again each generating a clause starting with C. Finally, a propositional resolution tree is given (R and Q are resolution rules) showing how to derive the empty clause from the clauses introduced through CNF and through theory lemmas. The proof shown depends on several additional proof files (not shown) which define the rules for resolution, theory reasoning, CNF conversion etc.

At the time this article is being written, CVC4 supports proofs for uninterpreted functions and arrays. Support for the theory of bit-vectors is a current project, and support for arithmetic proofs (already present in CVC3) is expected to follow soon thereafter.

6 The veriT SMT Solver

The veriT SMT solver is developed jointly at Loria, Nancy (France) and UFRN, Natal (Brazil). It is open-source, under the BSD license. veriT is first a testing platform for techniques developed around SMT, but it is sufficienctly stable to be used by third parties. Proofs in veriT are mainly resolution proofs interleaved with theory reasoning lemmas.

The proof trace language of veriT is inspired by the SMT-LIB 2.0 standard. A sample proof for our running example 1 is given in Figure 3. The language

[7] This is not quite true for the bit-vector theory solver which includes a second internal SAT solver as part of its machinery. The same minimal bookkeeping is done in this internal SAT solver as is done in the main SAT solver.

```
(check
(% a var_real
(% b var_real
(% x var_real
(% f (term (arrow Real Real))
(% q (term (arrow Real Bool))
(% @F1 (th_holds (<=_Real (a_var_real a) (a_var_real b)))
(% @F2 (th_holds (<=_Real (a_var_real b) (+_Real (a_var_real a) (a_var_real x))))
(% @F3 (th_holds (= Real (a_var_real x) (a_real 0/1)))
(% @F4 (th_holds (or (not (= Real (apply _ _ f (a_var_real a)) (apply _ _ f (a_var_real b))))
                     (and (= Bool (apply _ _ q (a_var_real a)) btrue)
                          (= Bool (apply _ _ q (+_Real (a_var_real b) (a_var_real x))) bfalse))))
(: (holds cln)

(decl_atom (<=_Real (a_var_real a) (a_var_real b)) (\ v1 (\ a1
(decl_atom (<=_Real (a_var_real b) (+_Real (a_var_real a) (a_var_real x))) (\ v2 (\ a2
(decl_atom (= Real (a_var_real x) (a_real 0/1)) (\ v3 (\ a3
(decl_atom (= Real (a_var_real a) (a_var_real b)) (\ v4 (\ a4
(decl_atom (= Real (apply _ _ f (a_var_real a)) (apply _ _ f (a_var_real b))) (\ v5 (\ a5
(decl_atom (= Bool (apply _ _ q (a_var_real a)) btrue) (\ v6 (\ a6
(decl_atom (= Bool (apply _ _ q (+_Real (a_var_real b) (a_var_real x))) bfalse) (\ v7 (\ a7
(decl_atom (<=_Real (a_var_real b) (a_var_real a)) (\ v8 (\ a8
(decl_atom (= Real (a_var_real a) (+_Real (a_var_real b) (a_var_real x))) (\ v9 (\ a9
(decl_atom (and (= Bool (apply _ _ q (a_var_real a)) btrue)
                (= Bool (apply _ _ q (+_Real (a_var_real b) (a_var_real x))) bfalse)) (\ v10 (\ a10

; CNFication
(satlem _ _ (asf _ _ _ a1 (\ l1 (clausify_false (contra _ @F1 l1)))) (\ C1
(satlem _ _ (asf _ _ _ a2 (\ l2 (clausify_false (contra _ @F2 l2)))) (\ C2
(satlem _ _ (asf _ _ _ a3 (\ l3 (clausify_false (contra _ @F3 l3)))) (\ C3
(satlem _ _ (ast _ _ _ a5 (\ l5 (asf _ _ _ a6 (\ l6 (clausify_false (contra _
  (and_elim_1 _ _ (or_elim_1 _ _ (not_not_intro _ l5) @F4)) l6)))))) (\ C4
(satlem _ _ (ast _ _ _ a5 (\ l5 (asf _ _ _ a7 (\ l7 (clausify_false (contra _
  (and_elim_2 _ _ (or_elim_1 _ _ (not_not_intro _ l5) @F4)) l7)))))) (\ C5

; Theory lemmas
; ~a4 ^ a1 ^ a8 => false
(satlem _ _ (asf _ _ _ a4 (\ l4 (ast _ _ _ a1 (\ l1 (ast _ _ _ a8 (\ l8 (clausify_false (contra _ l1
 (or_elim_1 _ _ (not_not_intro _ (<=_to_>=_Real _ _ l8)) (not_=_to_>=_=<_Real _ _ l4)))))))))) (\ C6
; a2 ^ a3 ^ ~a8 => false
(satlem _ _ (ast _ _ _ a2 (\ l2 (ast _ _ _ a3 (\ l3 (asf _ _ _ a8 (\ l8 (clausify_false
 (poly_norm_>= _ _ _ (<=_to_>=_Real _ _ l2) (pn_- _ _ _ _ _ (pn_+ _ _ _ _ _
  (pn_var a) (pn_var x)) (pn_var b)) (\ pn2
 (poly_norm_= _ _ _ (symm _ _ _ l3) (pn_- _ _ _ _ _ (pn_const 0/1) (pn_var x)) (\ pn3
 (poly_norm_> _ _ _ (not_<=_to_>_Real _ _ l8) (pn_- _ _ _ _ _ (pn_var b) (pn_var a)) (\ pn8
 (lra_contra_> _ (lra_add_>_>= _ _ _ pn8 (lra_add_=_>= _ _ _ pn3 pn2)))))))))))))))) (\ C7
; a4 ^ ~a5 => false
(satlem _ _ (ast _ _ _ a4 (\ l4 (asf _ _ _ a5 (\ l5 (clausify_false
 (contra _ (cong _ _ _ _ _ _ (refl _ f) l4) l5)))))) (\ C8
; a3 ^ a4 ^ ~a9 => false
(satlem _ _ (ast _ _ _ a3 (\ l3 (ast _ _ _ a4 (\ l4 (asf _ _ _ a9 (\ l9 (clausify_false
 (poly_norm_= _ _ _ (symm _ _ _ l3) (pn_- _ _ _ _ _ (pn_const 0/1) (pn_var x)) (\ pn3
 (poly_norm_= _ _ _ l4 (pn_- _ _ _ _ _ (pn_var a) (pn_var b)) (\ pn4
 (poly_norm_distinct _ _ _ l9 (pn_- _ _ _ _ _ (pn_+ _ _ _ _ _
  (pn_var b) (pn_var x)) (pn_var a)) (\ pn9
 (lra_contra_distinct _ (lra_add_=_distinct _ _ _
  (lra_add_=_= _ _ _ pn3 pn4) pn9))))))))))))))) (\ C9
; a9 ^ a6 ^ a7 => false
(satlem _ _ (ast _ _ _ a9 (\ l9 (ast _ _ _ a6 (\ l6 (ast _ _ _ a7 (\ l7 (clausify_false
 (contra _ (trans _ _ _ _ (trans _ _ _ _ (symm _ _ _ l6) (cong _ _ _ _ _ _
  (refl _ q) l9)) l7) b_true_not_false)))))))) (\ C10

; Resolution proof
(satlem_simplify _ _ _ (R _ _ (Q _ _ (Q _ _ C6 C1 v1) (Q _ _ (Q _ _ C7 C2 v2) C3 v3) v8)
(Q _ _ (Q _ _ (Q _ _ (Q _ _ (R _ _ C9 C10 v9) C3 v3) C4 v6) C5 v7) C8 v5) v4)
(\ x x))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Fig. 2. An LFSC proof for example Formula 1.

is quite coarse-grained and rather falls into the "proof trace" category rather than the category of full detailed proofs. It provides, however, a full account of the resolution proof, and equality reasoning is broken down into applications of the congruence and transitivity instances of the axiom schemas for equality. Symmetry of equality is silently used uniformly. Special proof rules are assigned to theory lemmas (from arithmetic), but veriT does not break down arithmetic reasoning to instances of e.g., Presburger axioms. The example here does not feature quantifier reasoning; proofs with quantifiers use quantifier instantiation rules.

```
(set .c1 (input :conclusion ((and (<= a b) (<= b (+ a x)) (= x 0)
                              (or (not (= (f b) (f a))) (and (q a) (not (q (+ b x))))))))))
(set .c2 (and :clauses (.c1) :conclusion ((<= a b))))
(set .c3 (and :clauses (.c1) :conclusion ((<= b (+ a x)))))
(set .c4 (and :clauses (.c1) :conclusion ((= x 0))))
(set .c5 (and :clauses (.c1) :conclusion
           ((or (not (= (f b) (f a))) (and (q a) (not (q (+ b x))))))))
(set .c6 (and_pos :conclusion ((not (and (q a) (not (q (+ b x))))) (q a))))
(set .c7 (and_pos :conclusion ((not (and (q a) (not (q (+ b x))))) (not (q (+ b x))))))
(set .c8 (or :clauses (.c5) :conclusion
           ((not (= (f b) (f a))) (and (q a) (not (q (+ b x)))))))
(set .c9 (eq_congruent :conclusion ((not (= a b)) (= (f b) (f a)))))
(set .c10 (la_disequality :conclusion ((or (= a b) (not (<= a b)) (not (<= b a))))))
(set .c11 (or :clauses (.c10) :conclusion ((= a b) (not (<= a b)) (not (<= b a)))))
(set .c12 (resolution :clauses (.c11 .c2) :conclusion ((= a b) (not (<= b a)))))
(set .c13 (la_generic :conclusion ((not (<= b (+ a x))) (<= b a) (not (= x 0)))))
(set .c14 (resolution :clauses (.c13 .c3 .c4) :conclusion ((<= b a))))
(set .c15 (resolution :clauses (.c12 .c14) :conclusion ((= a b))))
(set .c16 (resolution :clauses (.c9 .c15) :conclusion ((= (f b) (f a)))))
(set .c17 (resolution :clauses (.c8 .c16) :conclusion ((and (q a) (not (q (+ b x)))))))
(set .c18 (resolution :clauses (.c6 .c17) :conclusion ((q a))))
(set .c19 (resolution :clauses (.c7 .c17) :conclusion ((not (q (+ b x))))))
(set .c20 (eq_congruent_pred :conclusion ((not (= a (+ b x))) (not (q a)) (q (+ b x)))))
(set .c21 (resolution :clauses (.c20 .c18 .c19) :conclusion ((not (= a (+ b x))))))
(set .c22 (la_disequality :conclusion ((or (= a (+ b x)) (not (<= a (+ b x))) (not (<= (+ b x) a))))))
(set .c23 (or :clauses (.c22) :conclusion ((= a (+ b x)) (not (<= a (+ b x))) (not (<= (+ b x) a)))))
(set .c24 (resolution :clauses (.c23 .c21) :conclusion ((not (<= a (+ b x))) (not (<= (+ b x) a)))))
(set .c25 (eq_congruent_pred :conclusion
            ((not (= a b)) (not (= (+ a x) (+ b x))) (<= a (+ b x)) (not (<= b (+ a x))))))
(set .c26 (eq_congruent :conclusion ((not (= a b)) (not (= x x)) (= (+ a x) (+ b x)))))
(set .c27 (eq_reflexive :conclusion ((= x x))))
(set .c28 (resolution :clauses (.c26 .c27) :conclusion ((not (= a b)) (= (+ a x) (+ b x)))))
(set .c29 (resolution :clauses (.c25 .c28) :conclusion ((not (= a b)) (<= a (+ b x)) (not (<= b (+ a x))))))
(set .c30 (resolution :clauses (.c29 .c3 .c15) :conclusion ((<= a (+ b x)))))
(set .c31 (resolution :clauses (.c24 .c30) :conclusion ((not (<= (+ b x) a)))))
(set .c32 (la_generic :conclusion ((<= (+ b x) a) (not (= a b)) (not (= x 0)))))
(set .c33 (resolution :clauses (.c32 .c4 .c15 .c31) :conclusion ()))
```

Fig. 3. The proof output by veriT for example Formula 1. The output has been slightly edited to cut and indent long lines.

The first line gives the input. Clauses `c2` to c8 explain clausification of the input. Equality reasoning produces clauses labeled with `eq_congruent(_pred)` and `eq_reflexive`, whereas the linear arithmetic module produces all the clauses labeled `la_disequality` and `la_generic`. The rest of the proof is mainly resolutions, and ends with the empty clause.

In the future, the proof format of veriT will be further improved to even better stick to the SMT-LIB standard. In particular, when using shared terms (for simplicity, the given proof example does not use shared terms), veriT uses a notation to label repeated formulas which was inspired by the SVC custom input language, but which does not fit well with the spirit of the SMT-LIB language.

7 The Z3 SMT Solver

Z3 is a Satisfiability Modulo Theories (SMT) solver developed at Microsoft Research. Z3 source code is available online, and it is free for non-commercial purposes. Z3 is used in various software analysis and test-case generation projects at Microsoft Research and elsewhere. Proof generation is based on two simple ideas: (1) a notion of implicit quotation to avoid introducing auxiliary variables (this simplifies the creation of proof objects considerably); and (2) natural deduction style proofs to facilitate modular proof re-construction.

In Z3, proof objects are represented as terms. So a proof-tree is just a term where each inference rule is represented by a function symbol. For example, consider the proof-rule $mp(p, q, \varphi)$, where p is a proof for $\psi \rightarrow \varphi$ and q is a proof for ψ. Each proof-rule has *consequent*, the consequent of $mp(p, q, \varphi)$ is φ.

A basic underlying principle for composing and building proofs in Z3 has been to support a modular architecture that works well with theory solvers that receive literal assignments from other solvers and produce contradictions or new literal assignments. The theory solvers should be able to produce independent and opaque explanations for their decisions. Conceptually, each solver acts upon a set of hypotheses and produces a consequent. The basic proof-rules that support such an architecture can be summarized as: *hypothesis*, which introduces assumptions, *lemma*, which eliminates hypotheses, and *unit resolution*, which handles basic propagation. We say that a proof-term is *closed* when every path that ends with a hypothesis contains an application of rule lemma. If a proof-term is not closed, it is open.

The main propositional inference engine in Z3 is based on a DPLL(T) architecture. The DPLL(T) proof search method lends itself naturally to producing resolution style proofs. Systems, such as zChaff, and a version of MiniSAT, produce proof logs based on logging the unit propagation steps as well as the conflict resolution steps. The resulting log suffices to produce a propositional resolution proof.

The approach taken in Z3 bypasses logging, and instead builds proof objects during conflict resolution. With each clause we attach a proof. Clauses that were produced as part of the input have proofs that were produced from the previous steps. This approach does not require logging resolution steps for every unit-propagation, but delays the analysis of which unit propagation steps are useful until conflict resolution. The approach also does not produce a resolution proof directly. It produces a natural deduction style proof with hypotheses.

The theory of equality can be captured by axioms for reflexivity, symmetry, transitivity, and substitutivity of equality. In Z3, these axioms are inference

rules, and these inference rules apply for any binary relation that is reflexive, symmetric, transitive, and/or reflexive-monotone.

In the DPLL(T) architecture, decision procedures for a theory T identify sets of asserted T-inconsistent literals. Dually, the disjunction of the negated literals are T-tautologies. Consequently, proof terms created by theories can be summarized using a single form, here called *theory lemmas*. Some theory lemmas are annotated with hints to help proof checkers and reconstruction. For example, the theory of linear arithmetic produces theory lemmas based on Farkas' lemma. For example, suppose p is a proof for $x > 0$, and q is a proof for $2x + 1 < 0$, then $\mathit{farkas}(1, p, -1/2, q, \neg(x > 0) \vee \neg(2x + 1 < 0))$ is a theory lemma where the coefficients 1 and $-1/2$ are hints.

The Z3 simplifier applies standard simplification rules for the supported theories. For example, terms using the arithmetical operations, whether for integer, real, or bit-vector arithmetic, are normalized into sums of monomials. A single proof rule called *rewrite* is used to record the simplification steps. For example, $\mathit{rewrite}(x + x = 2x)$ is a proof for $x + x = 2x$.

Notice that Z3 does not axiomatize the legal rewrites. Instead, to check the rewrite steps, one must rely on a proof checker to be able to apply similar inferences for the set of built-in theories. Thus, like veriT, Z3 proofs are coarse-grained and also fall into the "proof trace" category.

Since Z3 proofs are terms, they can be traversed using the Z3 API. Proofs can also be output in the SMT-LIB 2.0 language. A sample proof for our running example 1 is given in Figure 4.

8 The Lean Prover

Lean is a new open source theorem prover being developed by Leonardo de Moura and Soonho Kong [17]. Lean is not a standard SMT solver; it can be used as an automatic prover like SMT solvers, but it can also be used as a proof assistant. The Lean kernel is based on dependent type theory, and is implemented in two layers. The first layer contains the type checker and APIs for creating and manipulating the terms, the declarations, and the environment. The first layer has several configuration options. For example, developers may instantiate the kernel with or without an impredicative Prop sort. They may also select whether Prop is proof-irrelevant or not. The first layer consists of 5k lines of C++ code. The second layer provides additional components such as inductive families (500 additional lines of code). When the kernel is instantiated, one selects which of these components should be used. The current components are already sufficient for producing an implementation of the Calculus of Inductive Constructions (CIC). The main difference is that Lean does not have universe cumulativity; it instead provides universe polymorphism. The Lean CIC-based kernel is treated as the standard kernel. Another design goal is to support the new Homotopy Type System (HTS) proposed by Vladimir Voevodsky. HTS is going to be implemented as another kernel instantiation.

```
(let (($x82 (q b)) (?x49 (* (- 1.0) b)) (?x50 (+ a ?x49))
      ($x51 (<= ?x50 0.0)) (?x35 (f b)) (?x34 (f a))
      ($x36 (= ?x34 ?x35)) ($x37 (not $x36))
      ($x43 (or $x37 (and (q a) (not (q (+ b x))))))
      ($x33 (= x 0.0)) (?x57 (+ a ?x49 x)) ($x56 (>= ?x57 0.0))
      ($x44 (and (<= a b) (<= b (+ a x)) $x33 $x43))
      (@x60 (monotonicity (rewrite (= (<= a b) $x51))
                          (rewrite (= (<= b (+ a x)) $x56))
                          (= $x44 (and $x51 $x56 $x33 $x43))))
      (@x61 (mp (asserted $x44) @x60 (and $x51 $x56 $x33 $x43)))
      (@x62 (and-elim @x61 $x51)) ($x71 (>= ?x50 0.0)))
(let ((@x70 (trans (monotonicity (and-elim @x61 $x33) (= ?x57 (+ a ?x49 0.0)))
                   (rewrite (= (+ a ?x49 0.0) ?x50)) (= ?x57 ?x50))))
(let ((@x74 (mp (and-elim @x61 $x56) (monotonicity @x70 (= $x56 $x71)) $x71)))
(let ((@x121 (monotonicity (symm ((_ th-lemma arith eq-propagate 1 1) @x74 @x62 (= a b)) (= b a))
                           (= $x82 (q a)))))
(let (($x38 (q a)) ($x96 (or (not $x38) $x82)) ($x97 (not $x96)))
(let ((@x115 (monotonicity (symm ((_ th-lemma arith eq-propagate 1 1) @x74 @x62 (= a b)) (= b a))
                           (= ?x35 ?x34))))
(let (($x100 (or $x37 $x97)))
(let ((@x102 (monotonicity (rewrite (= (and $x38 (not $x82)) $x97))
                           (= (or $x37 (and $x38 (not $x82))) $x100))))
(let (($x85 (not $x82)))
(let (($x88 (and $x38 $x85)))
(let (($x91 (or $x37 $x88)))
(let ((@x81 (trans (monotonicity (and-elim @x61 $x33) (= (+ b x) (+ b 0.0)))
                   (rewrite (= (+ b 0.0) b)) (= (+ b x) b))))
(let ((@x87 (monotonicity (monotonicity @x81 (= (q (+ b x)) $x82)) (= (not (q (+ b x))) $x85))))
(let ((@x93 (monotonicity (monotonicity @x87 (= (and $x38 (not (q (+ b x)))) $x88)) (= $x43 $x91))))
(let ((@x103 (mp (mp (and-elim @x61 $x43) @x93 $x91) @x102 $x100)))
(let ((@x119 (unit-resolution (def-axiom (or $x96 $x38))
                              (unit-resolution @x103 (symm @x115 $x36) $x97) $x38)))
(let ((@x118 (unit-resolution (def-axiom (or $x96 $x85))
                              (unit-resolution @x103 (symm @x115 $x36) $x97) $x85)))
(unit-resolution @x118 (mp @x119 (symm @x121 (= $x38 $x82)) $x82) false))))))))))))))))
```

Fig. 4. The proof output by Z3 for example Formula 1. The output has been slightly edited to cut and indent long lines.

Lean is meant to be used as a standalone system and as a software library. It provides an extensive API and can be easily embedded in other systems. SMT solvers can use the Lean API to create proof terms that can be independently checked. The API can also be used to export Lean proofs to other systems based on CIC (e.g., Coq and Matita).

Having a more expressive language for encoding proofs provides several advantages. First, we can easily add new "proof rules" without modifying the proof checker (i.e., type checker). Proof rules such as *mp* and *monotonicity* used in Z3 are just theorems in Lean. When a new decision procedure (or any other form of automation) is implemented, the developer must first prove the theorems that are needed to justify the results produced by the automatic procedure. For example, suppose a developer is implementing a procedure for Presburger arithmetic, she will probably use a theorem such as:

```
theorem add_comm (n m:nat) : n + m = m + n
:=
  induction_on m
    (trans (add_zero_right _) (symm (add_zero_left _)))
    (take k IH,
      calc
        n + succ k = succ (n+k) : add_succ_right _ _
          ... = succ (k + n) : {IH}
          ... = succ k + n : symm (add_succ_left _ _))
```

Pre-processing steps such as Skolemization can be supported in a similar way. Whenever a preprocessing procedure applies a Skolemization step, it uses the following theorem to justify it.

```
theorem skolem_th {A : Type} {B : A -> Type} {P : forall x : A, B x -> Bool} :
        (forall x, exists y, P x y) = (exists f, (forall x, P x (f x)))
:= iff_intro
      (assume H : (forall x, exists y, P x y), @axiom_of_choice _ _ P H)
      (assume H : (exists f, (forall x, P x (f x))),
            take x, obtain (fw : forall x, B x) (Hw : forall x, P x (fw x)), from H,
                 exists_intro (fw x) (Hw x))
```

Most SMT solvers make extensive use of preprocessing steps, and as pointed out before, it is not easy to be proof-producing for every single rewriting step that can occur in the solver. In Lean, this issue is addressed by providing a generic rewriting engine that can use any previously proved theorems. The engine accepts two kinds of theorems: congruence theorems and (conditional) equations. For example, the following two theorems can be used to distribute universal quantifiers over disjuctions when the left (right) hand side does not reference the bound variable.

```
theorem forall_or_distributel {A : Type} (p : Bool) (q : A -> Bool)
          : (forall x, q x \/ p) = ((forall x, q x) \/ p)
theorem forall_or_distributer {A : Type} (p : Bool) (q : A -> Bool)
          : (forall x, p \/ q x) = (p  forall x, q x)
```

9 Applications

While the ability to use a proof to independently check the result of an SMT solver is generally seen as a valuable objective, other applications have thus far

been more effective in driving the development of proof-producing SMT solvers, particularly the desire to use SMT solvers in skeptical proof assistants and the need for interpolation. Below, we discuss some of the specific applications of the solvers discussed in this paper.

9.1 CVC

Early applications of proofs in CVC and CVC Lite were covered in Section 3. Later, CVC3 was instrumented to produce LFSC-proofs (instead proofs in its *ad hoc* native format) both as a platform for experimenting with proof systems [43] and as a way to produce certified interpolants [45].

Proofs in CVC4 are still at an early stage, but a current project[8] has as its objective the completion of proof-production for the theories of uninterpreted functions, arrays, and bit-vectors in CVC4 by August 2014 and a facility for translating those proofs to Coq by August 2015.

9.2 veriT

Proofs were first implemented in veriT to allow proof reconstruction of SMT proofs within proof assistants. This feature has been successfully used with Isabelle [23], and later with Coq within the SMTCoq tool [1, 31].

Georg Hofferek et al. [29] used the proof producing capability of veriT in the context of controller synthesis for pipelined processors. The method basically builds several interpolants from a unique proof. In the Rodin plugin to SMT [21], proofs from veriT are used as a means to quickly extract unsatisfiable cores.

9.3 Z3

The two main applications for Z3 proof certificates are: proof reconstruction in interactive proof assistants such as Isabelle [11]; and interpolation generation [34].

In Isabelle/HOL, Z3 proofs are reconstructed in a completely different system based on a secure proof kernel. Proof reconstruction is quite an involved process, because proof rules such as *theory lemmas* require several steps of reasoning in Isabelle/HOL.

The interpolation prover iZ3 [34] is implemented on top of Z3. It uses Z3 proofs to guide the construction of proofs by a secondary, less efficient, interpolating prover. It translates Z3 proofs into a proof calculus that does admit feasible interpolation, with "gaps", or lemmas, that must be discharged by the secondary prover.

[8] funded by the DARPA HACMS program

10 Conclusions

Building a proof-producing SMT solver appears easy on the surface: the underlying SAT solver generates Boolean resolution proofs, and, as long as the theories in the combination also produce certificates, these can also be integrated using Boolean resolution. There are, however, challenges both small and large related to SMT proofs. First, it is necessary to collect and store all the necessary information to produce the final proof; this is mostly a technical problem, but, it can be a bottleneck during proof search. Indeed, this may involve an enormous amount of bookkeeping, and, if using main memory, may quickly exhaust the available memory. Second, SMT often relies on many preprocessing techniques, some being necessary for the soundness of tool, some being useful for efficiency. The attitude followed by most SMT developers is to provide some kind of high level trace for essential preprocessing. Non-essential preprocessing is turned off, with consequences for efficiency. It may also be non-trivial to generate good proofs for some preprocessing techniques, e.g. for Skolemization, or symmetry breaking. Finally, SMT solvers may use external tools as specialized reasoners for some theories. Producing a proof in SMT may thus, in some cases, require certificates from those external reasoners as well.

An important challenge for the SMT community is to provide a proof format which is sufficiently flexible to accommodate all needs, sufficiently compact to be practical, and sufficiently elegant (which is very subjective) to be accepted by most. Some formats have been proposed [9, 16, 44], but it seems it is still too early for a consensus on how to represent SMT proofs.

Acknowledgements: Clark Barrett would like to thank Andrew Reynolds for providing the LFSC proof example in Figure 2. Pascal Fontaine would like to thank David Déharbe, who jointly designed veriT with him. Proof-production in CVC4 is funded in part by DARPA award FA8750-13-2-0241.

References

1. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP - Certified Programs and Proofs - First International Conference - 2011*, volume 7086 of *Lecture notes in computer science - LNCS*, pages 135–150, Kenting, Taïwan, Province De Chine, December 2011. Springer.
2. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.
3. Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

4. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard : Version 2.0, December 2010.
5. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
6. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the* 1^{st} *International Conference on Formal Methods In Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California.
7. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002. Copenhagen, Denmark.
8. Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: a proposal. In *First International Workshop on Proof eXchange for Theorem Proving - PxTP 2011*, Wroclaw, Pologne, August 2011.
9. Frdric Besson, Pascal Fontaine, and Laurent Thry. A flexible proof format for smt: a proposal. In *Workshop on Proof eXchange for Theorem Proving*, 2011.
10. Alexander Bockmayr and V. Weispfenning. Solving numerical constraints. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 12, pages 751–842. Elsevier Science B.V., 2001.
11. Sascha Böhme. Proof reconstruction for Z3 in Isabelle/HOL. In *7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, 2009.
12. Sascha Böhme and Tjark Weber. Designing proof formats: A user's perspective — experience report. In *First International Workshop on Proof eXchange for Theorem Proving - PxTP 2011*, Wroclaw, Pologne, August 2011.
13. Cristina Borralleras, Salvador Lucas, Rafael Navarro-Marset, Enric Rodríguez-Carbonell, and Albert Rubio. Solving non-linear polynomial arithmetic via sat modulo linear arithmetic. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2009.
14. Thomas Bouton, Diego B. Caminha de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Renate A Schmidt, editor, *Automated Deduction CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer Berlin Heidelberg, 2009.
15. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, chapter 28, pages 299–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
16. Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
17. Leonardo de Moura and Soonho Kong. Lean theorem prover: `http://github.com/leanprover`, 2014.

18. Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.
19. D. Deharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code. In *First International Conference on Software Engineering and Formal Methods, 2003.*, pages 220–228. IEEE, 2003.
20. Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 2006.
21. David Dharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Smt solvers for rodin. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer Berlin Heidelberg, 2012.
22. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer Berlin Heidelberg, 2003.
23. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin Heidelberg, 2006.
24. Yeting Ge and Clark Barrett. Proof translation and SMT-LIB benchmark certification: A preliminary report. In *Proceedings of the* 6^{th} *International Workshop on Satisfiability Modulo Theories (SMT '08)*, 2008.
25. Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122, February 2009.
26. Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.
27. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.
28. John Harrison. Hol light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer Berlin Heidelberg, 1996.
29. Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Paul Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *FMCAD 2013 - Formal Methods in Computer-Aided Design*, pages 77 – 84. IEEE, 2013.
30. Clément Hurlin, Amine Chaib, Pascal Fontaine, Stephan Merz, and Tjark Weber. Practical proof reconstruction for first-order logic and set-theoretical constructions. In Moa Johansson Lucas Dixon, editor, *The Isabelle Workshop 2007 - Isabelle'07*, pages 2–13, Bremen, Germany, 2007.
31. C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers*. PhD thesis, cole Polytechnique, June 2013.

32. Timothy King, Clark Barrett, and Bruno Dutertre. Simplex with sum of infeasibilities for SMT. In *Proceedings of the* 13^{th} *International Conference on Formal Methods In Computer-Aided Design (FMCAD '13)*, pages 189–196. FMCAD Inc., October 2013. Portland, Oregon.
33. Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In Alessandro Armando and Alessandro Cimatti, editors, *Proceedings of the* 3^{rd} *Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, January 2006. Edinburgh, Scotland.
34. Kenneth L. McMillan. Interpolants from z3 proofs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 19–27, 2011.
35. Michał Moskal. Rocket-Fast proof checking for SMT solvers. In C R Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 486–500. Springer Berlin Heidelberg, 2008.
36. Greg Nelson and Derek C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
37. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.
38. Robert Nieuwenhuis and Albert Oliveras. Union-find and congruence closure algorithms that produce proofs. In Cesare Tinelli and Silvio Ranise, editors, *Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2004.
39. Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.
40. Robert Nieuwenhuis, Albert Oliveras, and Cesare 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, November 2006.
41. Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.
42. Duckki Oe, Andrew Reynolds, and Aaron Stump. Fast and flexible proof checking for SMT. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories - SMT '09*, pages 6–13, New York, New York, USA, August 2009. ACM Press.
43. Andrew Reynolds, Liana Hadarean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic with LFSC. In *Proceedings of the* 8^{th} *International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010. Edinburgh, Scotland.
44. Andrew Reynolds, Liana Haderean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic with LFSC. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.
45. Andrew Reynolds, Cesare Tinelli, and Liana Hadarean. Certified interpolant generation for EUF. In S. Lahiri and S. Seshia, editors, *Proceedings of the 9th International Workshop on Satisfiability Modulo Theories (Snowbird, USA)*, 2011.
46. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.

47. Aaron Stump, Clark W. Barrett, and David L. Dill. Producing proofs from an arithmetic decision procedure in elliptical LF. In Frank Pfenning, editor, *Proceedings of the* 3^{rd} *International Workshop on Logical Frameworks and Meta-Languages (LFM '02)*, volume 70(2) of *Electronic Notes in Theoretical Computer Science*, pages 29–41. Elsevier, July 2002. Copenhagen, Denmark.
48. Aaron Stump and David L. Dill. Faster proof checking in the edinburgh logical framework. In Andrei Voronkov, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 392–407. Springer Berlin Heidelberg, 2002.
49. Aaron Stump and Duckki Oe. Towards an SMT proof format. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning - SMT '08/BPR '08*, page 27, New York, New York, USA, July 2008. ACM Press.
50. Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, July 2013.
51. Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.