A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses

Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, Benjamin Werner

▶ To cite this version:

Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, et al.. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. Jouannaud, Jean-Pierre and Shao, Zhong. CPP - Certified Programs and Proofs - First International Conference - 2011, Dec 2011, Kenting, Taiwan. Springer, 7086, pp.135-150, 2011, Lecture notes in computer science - LNCS. <10.1007/978-3-642-25379-9_12>. <hal-00639130>

HAL Id: hal-00639130
https://hal.inria.fr/hal-00639130

Submitted on 20 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses[*]

M. Armand[1], G. Faure[2], B. Grégoire[1], C. Keller[2], L. Théry[1], and B. Werner[2]

[1] INRIA Sophia-Antipolis
{Michael.Armand,Benjamin.Gregoire,Laurent.Thery}@inria.fr
[2] INRIA - Saclay–Île-de-France at LIX, École Polytechnique
{Germain.Faure,Chantal.Keller,Benjamin.Werner}@inria.fr

Abstract We present a way to enjoy the power of SAT and SMT provers in Coq without compromising soundness. This requires these provers to return not only a yes/no answer, but also a proof witness that can be independently rechecked. We present such a checker, written and fully certified in Coq. It is conceived in a modular way, in order to tame the proofs' complexity and to be extendable. It can currently check witnesses from the SAT solver ZChaff and from the SMT solver veriT. Experiments highlight the efficiency of this checker. On top of it, new reflexive Coq tactics have been built that can decide a subset of Coq's logic by calling external provers and carefully checking their answers.

1 Introduction

When integrating a technology like SAT/SMT solvers in type theoretical provers like Coq, one classically has the choice between two ways, which Barendregt and Barendsen [4] named the *autarkic* and the *skeptical* approach.

In the autarkic approach, all the computation is performed in the proof system. In this case, this means implementing a whole, sufficiently efficient, SAT/SMT solver as a Coq function, and then proving it correct. This approach is followed in the Ergo-Coq effort [10].

In the skeptical approach, the external tool, here the SAT/SMT solver, is instrumented in order to produce not only a yes/no answer but also a *proof witness*, or a *trace* of its computation. It is the approach we follow here. The main contribution of the paper is to propose a modular and effective checker for SAT and SMT proof witnesses, written in Coq and *fully certified*.

In general, the choice between the autarkic and the skeptical approach depends on the considered problem. Typically, when the problem is solved by a greedy algorithm or something similar requiring no backtracking, the autarkic approach is generally to be preferred. In the case of SAT/SMT solvers, where a lot of time is devoted to actually finding the proof path, the skeptical approach can have an edge in terms of efficiency. Another advantage of the skeptical approach may be that it requires much less effort to certify a checker only than

[*] This work was supported in part by the french ANR DECERT initiative.

a whole prover. A difficulty is that the external prover and the internal checker have to speak a common language; indeed, finding the best possible format for the proof witnesses is the crucial step which determines the whole architecture of this prover-checker interaction.

Let us note that we see two benefits of this work. The first one being, as mentioned above, the addition to Coq of powerful and yet sound automation tactics. A second one is that it gives a means to enhance the reliability of automatic provers, by offering the possibility to have their results checked, *a posteriori* in Coq.

One keypoint for success in computational proofs is making the best possible usage of the, somewhat limited, computational power available inside the prover (Coq). This concern underlies the whole work presented here. An important advantage is the new addition to Coq of updatable arrays [3] which we use extensively in this work.

A wholly different approach to the integration of SAT/SMT to provers is to transform the proof witnesses into deductions. This approach has been taken for the Isabelle and HOL Light provers. We provide some comparison in Section 7.

The paper is organized as follows. The next section recalls the basic principles of SAT and SMT solvers. Section 3 describes the modular structure of the checker written in Coq; Sections 4 and 5 detail its different components, dealing respectively with the SAT parts, and two specific theories. Part 6 describes how the different parts are linked together in order to provide a practical tool. Finally, Section 7 is devoted to benchmarks and comparison with other works.

The source code of the checker and information on its usage can be found online [1].

2 The SAT and SMT problems

2.1 SAT solvers

SAT solvers deal with propositional formulas given in Conjunctive Normal Form (CNF); they decide whether or not there exists an assignment of the variables satisfying the formula. We recall basic definitions. A *literal* is a variable or its negation, a *clause* is a disjunction of literals, noted $l_1 \vee \cdots \vee l_n$. Finally, a formula in CNF is given by a *finite set of clauses* $\mathcal{S}$, seen as their conjunction.

A valuation ρ associating a Boolean to each variable straightforwardly induces an interpretation of a set of clauses ($[\![S]\!]_\rho$) as a Boolean. A set of clauses $\mathcal{S}$ is satisfiable if and only if there exists a valuation ρ such that $[\![\mathcal{S}]\!]_\rho = \top$. Conversely, $\mathcal{S}$ is unsatisfiable if and only if for any valuation ρ, $[\![\mathcal{S}]\!]_\rho = \bot$.

Modern SAT solvers rely on variants of the DPLL algorithm which can be customized to generate a proof witness [12]. The witness is:

- either an assignment of the variables to $\top$ and $\bot$ in order to satisfy all the clauses, in the case where the set of clauses is satisfiable;
- or a proof by resolution of the empty clause, in the case where the formula is unsatisfiable.

We recall that the refutationally complete resolution rule is:

$$\frac{v \vee C \qquad \bar{v} \vee D}{C \vee D}$$

where v is called the *resolution variable*. A *comb tree* is a tree where at least one child of every inner node is a leaf. A comb tree of resolution rules is called a *resolution chain*.

From the point of view of result certification, the case of unsatisfiability is the more challenging one. The format used by most SAT solvers for proof witnesses of unsatisfiability is a list of resolution chains. This list should be understood as a shared representation of the resolution tree: each resolution chain derives a new clause (that can be used later by other resolution chains), and the last resolution chain should derive the empty clause. It corresponds exactly to a subset of the learned clauses that the algorithm encountered during its run.

2.2 SMT solvers

SMT solvers decide an extension of the SAT problem in which positive literals are not only Boolean variables but also atomic propositions of some first-order theory (possibly multisorted). Given a signature Σ containing simple types, and function and predicate symbols with their types, a *theory* $\mathcal{T}$ is a set of formulas of type *bool* written using this signature, variables and logical connectives. Those formulas are called *theory lemmas*.

The standard architecture for SMT solvers is an interaction between a SAT solver and decision procedures for the theories [12]. The SAT solver generates models and the theory solvers try to refute them. When a SAT model is consistent with all the theories, the initial problem is found satisfiable. Otherwise, a new clause corresponding to a theory lemma is added to the SAT problem in order to rule out the model. The SAT solver can then be called again to generate another model. Since there are only a finite number of SAT models, this enumeration eventually terminates. If the empty clause is derived by the SAT solver, the initial problem is unsatisfiable.

In this setting, a proof witness for unsatisfiability is a resolution tree of the empty clause where leaves are not only initial clauses but also theory lemmas.

To our knowledge, at least three existing SMT solvers can deliver informative proof witnesses of this kind: Z3, CVC3 and veriT (some other solvers provide less informative witnesses). Although there are some output format differences, each of these three provers does return a resolution tree with theory lemmas in the case of unsatisfiability. They also give witnesses for satisfiability.

3 A modular and efficient Coq checker for SAT and SMT proof witnesses

We have developed a general framework to verify SAT and SMT proof witnesses for unsatisfiability proofs. During the SAT/SMT process, new clauses are generated until reaching the empty clause. These new clauses are either propositional consequences of the initial clauses, theory lemmas, or related to the CNF conversion and to various simplifications. A *small certificate* is generated to explain how each of the new clauses that are useful to obtain unsatisfiability was produced.

Our checker is defined by bringing together small checkers, each dedicated to one aspect of the verification of the resolution tree (the propositional reasoning, the theory lemmas of a given theory, the CNF conversion etc.). This modularity is a key aspect of our work. Small checkers are then independent pieces of code that can be composed in a very flexible way. Section 4.1 is dedicated to checking resolution chains, Section 4.2 to checking CNF computation. For theory lemmas, Section 5.2 describes what has been done for congruence closure and Section 5.3 for linear integer arithmetic. In each section, we present exactly the certificate format, how the checker works and is proved correct. The actual connection between Coq and the SAT and SMT provers is presented only later in Section 6.

The common aim underlying these different parts is preserving efficiency, in time and space. The main difficulty is the very large number of clauses that may need to be handled. We therefore strongly rely on the new *persistent arrays* feature of Coq, described in [3]. Schematically, checkers can be understood as sharing a global *state*, holding the current set of clauses, and implemented by an array. One typical optimization will be to keep this array as small as possible, by re-using a cell as soon as the clause it holds is known to be not used anymore for further computations.

In order to achieve modularity, we restrict ourselves to a very lightweight interface for the small checkers. Our implementation is based on four main data types: $\mathcal{S}$, $\mathcal{C}$, $\mathsf{clauseId}$, and $\mathsf{scertif}$. The first one, $\mathcal{S}$, represents the state. Initially, the only clause in the state is the singleton clause that contains the formula to be proved unsatisfiable. The type for clauses is $\mathcal{C}$. An element of type $\mathsf{clauseId}$ is an identifier that refers to a clause. The get and set functions let us access and modify the state and the main checker only needs to be able to check whether a clause is empty, i.e it represents $\bot$:

$$\mathsf{get} \ : \ \mathcal{S} \to \mathsf{clauseId} \to \mathcal{C} \qquad \mathsf{set} \ : \ \mathcal{S} \to \mathsf{clauseId} \to \mathcal{C} \to \mathcal{S} \qquad \mathsf{isFalse} \ : \ \mathcal{C} \to \mathsf{bool}$$

A *small checker* in our setting is just a Coq program that, given a state and a small certificate c, returns a new clause C. It is *correct* if the new clause that is produced is a consequence of the state S:

$$\text{for any interpretation } \rho, \ [\![S]\!]_\rho \Rightarrow [\![C]\!]_\rho$$

The type $\mathsf{scertif}$ is an enumeration type that collects all the possible small certificates. Associated to this type, there is a dispatching function scheck that, depending on the small certificate it receives, calls the appropriate checker.

Since small checkers just generate clauses, the only information we have to provide when gluing together small certificates is where the new clauses have to be stored. Finally, at the end, the initial formula should be proved unsatisfiable, so the empty clause must have been derived. So, its actual location must be given by the main certificate. The type of such certificate is then

$$\mathsf{certif} := \mathsf{list}\ (\mathsf{clauseId} * \mathsf{scertif}) * \mathsf{clauseId}$$

The main checker is now trivially defined by

```
check $S$ $cert$ =
  let $(l, k) := cert$ in
  let $S' :=$ List.fold_left (fun $S$ $(p, c) \Rightarrow$ set $S$ $p$ (scheck $S$ $c$)) $S$ $l$ in
  isFalse (get $S'$ $k$)
```

It takes an initial state S and a certificate $cert$ and sequentially calls small checkers to compute new clauses extending the state with the generated clauses. Provided all small checkers are correct and the check returns true and since satisfiability is preserved, reaching an absurd state implies that the initial state was indeed unsatisfiable.

As hinted above, the get and set functions are built upon the persistent arrays of Coq and one such array is used in the state to store clauses. clauseIds are thus array indexes, i.e. 31-bit integers. So, access and update to the set of clauses are performed in constant time. Since we use arrays, in the following, (get S n) is written as $S.[n]$.

It is very unlikely that a given SMT solver will output exactly our kind of certificates. A pre-processing phase is required in order to translate proof-witnesses generated by the SMT solver into our format. In particular, the precise clause allocation that is required by our format is usually not provided by the SMT solver. Finding out such an allocation is a post-processing seen from the SMT solver, and a pre-processing seen from Coq. It involves techniques similar to register allocation in compilation. First, the maximal number of clauses that need to be alive at the same time is computed. Then, a cell is explicitly allocated to each clause, in such a way that two clauses that need to be alive at the same time do not share the same cell. In practice, this has a big impact on the memory required to check a certificate.

4 Small checkers for SAT

4.1 A small checker for resolution chains

As explained in Section 2.1, the SAT contribution is represented in the proof witness by chains of resolutions. The constructor (res_certif $[| n_1; \dots; n_i |]$) in scertif represents these chains. Let $R(C_1, C_2)$ be the resolution between the clause C_1 and the clause C_2. Given this certificate and a state S, the corresponding small checker iteratively applies resolution to eventually produce the new clause $R(\dots(R(S.[n_1],\ S.[n_2]), \dots),\ S.[n_i])$.

This efficient treatment of resolution chains requires a careful encoding of clauses and literals. First, we encode propositional variables as 31-bit integers. We follow the usual convention reserving location 0 for the constant true, which means that the interpretation of propositional variables ρ always comes with the side-condition that $\rho(0) = \mathsf{true}$. Literals are also encoded as 31-bit integers, taking advantage of parity. The interpretation for literals is built such that:

$$[\![l]\!]_\rho = \text{if even}\, l \text{ then } \rho(l/2) \text{ else } \neg\rho(l/2).$$

The point being that parity check and division by two are very fast since they are directly performed by machine integer operations as explained in [3].

Clauses are represented by lists of literals. The interpretation $[\![c]\!]_\rho$ of a clause c is then the disjunction of the interpretation of its literals. The interpretation $[\![S]\!]_\rho$ of a state S is defined as the conjunction of the interpretation of its clauses.

To give a concrete example, consider the interpretation of proposition variables: $\rho(0) = \mathsf{true}$, $\rho(1) = x$, $\rho(2) = y$. If $S = [|\ [2; 4];\ [5];\ [3; 4]\ |]$ we have

$$\begin{aligned}[\![S]\!]_\rho &= [\![\,[2;4]\,]\!]_\rho \wedge [\![\,[5]\,]\!]_\rho \wedge [\![\,[3;4]\,]\!]_\rho = ([\![2]\!]_\rho \vee [\![4]\!]_\rho) \wedge [\![5]\!]_\rho \wedge ([\![3]\!]_\rho \vee [\![4]\!]_\rho) \\ &= (\rho(1) \vee \rho(2)) \wedge \neg\rho(2) \wedge (\neg\rho(1) \vee \rho(2)) = (x \vee y) \wedge \neg y \wedge (\neg x \vee y)\end{aligned}$$

In this setting, the interpretation of a set of clauses is always a CNF formula. A proof of unsatisfiability of this formula is the following chain of resolutions

$$\dfrac{\dfrac{\dfrac{x \vee y \qquad \neg y}{x} \qquad \neg x \vee y}{y} \qquad \neg y}{\bot}$$

This corresponds in our format to certificate $([0, \mathsf{res\_certif}\ [0; 1; 2; 1]], 0)$.

4.2 Small checkers for CNF computation

With our previous small checker, proof witnesses for SAT problems in CNF can be checked in Coq. The next step is to be able to verify the transformation of a formula into an equisatisfiable formula in CNF. This is usually done using a technique proposed by Tseitin [14]. This involves generating a new variable for every subterm of the formula; with these new variables, the CNF transformation is linear. It is this idea that we are going to implement in our setting. Naming subterms corresponds to a form of hash-consing. A hashed formula is either an atom, true, false, or a logical connective. Sub-formulas of connectives are literals (i.e., a variable or its negation):

```
Type hform =
  | Fatom (a : atom) | Ftrue | Ffalse
  | Fand (ls : array lit) | For (ls : array lit) | Fxor (l1 l2 : lit)
  | Fimp(l1 l2 : lit) | Fite (l1 l2 l3 : lit) | Fiff (l1 l2 : lit) | Fdneg (l : lit).
```

Note that the connectives Fand and For are n-ary operators which allows a more efficient subsequent computation. Note also that we have no primitive constructor for negation, which has to be pushed to the literals (with little cost, using the odd/even coding described above). However, double negation is explicit and primitive, in order to represent the formula $\neg\neg x$ faithfully.

For computation, the state of the checker is extended with a new array `ftable` containing the table of the hashed formulas. For example, the formula $\neg((x \wedge y) \vee \neg(x \wedge y))$ can be encoded by the literal 9 using the formula table:

$$[|\ \mathsf{Ftrue};\ \mathsf{Fatom}\ 0;\ \mathsf{Fatom}\ 1;\ \mathsf{Fand}\ [|2; 4|];\ \mathsf{For}\ [|6; 7|]\ |]$$

with the interpretation for atoms defined by $\rho_A(0) = x, \rho_A(1) = y$. Three things are worth noticing. First, the sub-formula $x \wedge y$ appears twice in the formula but is shared in the table (at location 3); indeed, this representation allows maximal sharing. Second, we have to ensure that our table does not contain infinite terms, so our state can be interpreted. This is done by preserving some well-formedness on the table: if a literal m appears in the formula stored at location n, we always have $m/2 < n$. It is this condition that actually allows us to define the Boolean interpretation $[\![f]\!]_{\rho_A}$ recursively over the formula f, where ρ is the interpretation of the variables.

Finally, the tables always have Ftrue at location 0. The interpretation of propositional variables of the previous section is simply defined as $\rho(n) = [\![\mathsf{ftable}.[n]]\!]_{\rho_A}$.

Tseitin identifies 40 generic tautology schemes used in the transformation to CNF. In the case of our example $\neg((x \wedge y) \vee \neg(x \wedge y))$, the transformation invokes the following tautology $\neg(A_0 \vee \cdots \vee A_i \vee \cdots \vee A_n) \Rightarrow \bar{A}_i$. For each of these tautologies, we have written a specific test function which verifies that the corresponding tautology can actually be applied. In this case, the certificate is written ($\mathsf{nor\_certif}\ m\ i$) and the corresponding checker verifies that $S.[m]$ is a singleton clause $[k]$ with k being odd and that $\mathsf{ftable}.[k/2]$ is a disjunction ($\mathsf{For}\ [|l_1; \ldots;\ l_n|]$). If these conditions are met, it produces the clause $[\bar{l_i}]$ if $i < n$. If the verification fails, the true clause is returned. This trick of using the true clause as default will be used for all the other small checkers.

The full certificate of unsatisfiability for our example is then:

$$([(1, \mathsf{nor\_certif}\ 0\ 0);\ (0, \mathsf{nor\_certif}\ 0\ 1);\ (0, \mathsf{res\_certif}\ [|1;\ 0|]], 0).$$

The computation of the final set of clauses proceeds like this:

$$[|\,[9];\ [0]\,|] \overset{1,\mathsf{nor\_certif}\ 0\ 0}{\rightsquigarrow} [|\,[9];\ [7]\,|] \overset{0,\mathsf{nor\_certif}\ 0\ 1}{\rightsquigarrow} [|\,[6];\ [7]\,|] \overset{0,\mathsf{res\_certif}\ [|1;\ 0|]}{\rightsquigarrow} [|\,[\,];\ [7]\,|]$$

At the end, we find the empty clause at location 0, which ensures the initial formula is unsatisfiable.

Let us finally remark that our format is compatible with lazy CNF transformation and also that it is possible to delegate the CNF computation to the SMT solver.

5 Small checkers for congruence closure and linear arithmetic

5.1 Refining the term representation

In order to handle theories, we need to provide a proper representation for atoms. Atoms can represent objects of different types, so we also need a proper representation for types. Theories like EUF manipulate uninterpreted functions, so we also need uninterpreted base types. Here is our representation:

$\mathsf{Type\ btype} = \mathsf{Tidx}\ (n : \mathsf{int})\ |\ \mathsf{Tbool}\ |\ \mathsf{TZ}\ |\ \ldots$
$\mathsf{Type\ cst} = \mathsf{Zcst}\ (n : \mathsf{Z})\ |\ \ldots$
$\mathsf{Type\ op} = \mathsf{Oidx}\ (n : \mathsf{int})\ |\ \mathsf{Oeq}\ (t : \mathsf{btype})\ |\ \mathsf{OZle}\ |\ \mathsf{OZlt}\ |\ \mathsf{OZplus}\ |\ \ldots$
$\mathsf{Type\ hatom} = \mathsf{Avar}\ (v : \mathsf{avar})\ |\ \mathsf{Acst}\ (c : \mathsf{cst})\ |\ \mathsf{Aapp}\ (o : \mathsf{op})\ (a_s : \mathsf{list\ atom}).$

As for formulas, our encoding uses a table atable, so the type atom is an abbreviation for int. A hashed atom is either a variable (Avar), a constant of a theory (Acst), or an application of an operator to its arguments (list atom). Operators are uninterpreted functions or predicates (Oidx), or a function (OZplus) or a predicate (OZle) of a given theory. Base types are either uninterpreted (Tidx) or a type of a given theory (TZ).

To illustrate our representation, let us consider the formula $f\,x < 1 \lor g\,(y + 1) < 1$ over the Coq integer Z where the Coq type of x is α and is left uninterpreted. We have the following tables:

$$\begin{aligned}
\mathsf{ftable} &= [|\mathsf{Ftrue}; \mathsf{Fatom}\ 5; \mathsf{Fatom}\ 7; \mathsf{For}\ [|2;4|]|] \\
\mathsf{atable} &= [|\mathsf{Avar}\ 0;\ \mathsf{Avar}\ 1;\ \mathsf{Acst}\ (\mathsf{Z\_cst}\ 1);\ \mathsf{Aapp}\ (\mathsf{Oidx}\ 0)\ [0];\ \mathsf{Aapp}\ \mathsf{OZplus}\ [1;2]; \\
&\qquad \mathsf{Aapp}\ \mathsf{OZlt}\ [3;2];\ \mathsf{Aapp}\ (\mathsf{Oidx}\ 1)\ [4];\ \mathsf{Aapp}\ \mathsf{OZlt}\ [7;2]|]
\end{aligned}$$

Interpreting types is easy. We just need a table ttable associating a Coq type to every type index. We denote by $[\![T]\!]_t$ the interpretation of a base type T with respect to this table. Interpreting atoms is more difficult, since we must build well-typed Coq terms. In particular, different elements of the table may be interpreted into different types. Therefore, our interpretation function returns a dependent pair (T, v) where T has type btype and v has type $[\![T]\!]_t$. The interpretation of atoms $[\![A]\!]$ uses two tables. The first one (vtable) is a valuation associating (T, v) to a variable index. The second one (otable) associates a pair $(([T_1, \ldots, T_n], T), f)$ to an operator index, where T_i, T have type btype and f has type $[\![T_1]\!]_t \rightarrow \ldots [\![T_n]\!]_t \rightarrow [\![T]\!]_t$. With these tables, defining the interpretation $[\![A]\!]$ is straightforward. We simply check that all applications are well-typed, if not we return (Tbool, true). This makes our interpretation a total function. Here are the three tables used by the interpretation for the previous example:

$$\begin{aligned}
\mathsf{ttable} &= [|\alpha|] \\
\mathsf{vtable} &= [|(\mathsf{Tidx}\ 0, x); (\mathsf{TZ}, y)|] \\
\mathsf{otable} &= [|((([\mathsf{Tidx}\ 0], \mathsf{TZ}), f); (([\mathsf{TZ}], \mathsf{TZ}), g)|]
\end{aligned}$$

The interpretation of atoms of the previous section is simply defined as $\rho_A(a) = [\![\mathsf{atable}.[a]]\!]$.

We need some side conditions on the different tables to be able to complete the proof of our small checkers. First, the hashed atom contained at position k of the atable should refer to atoms strictly smaller than k (this ensures that the interpretation terminates). Second, the atable should only contain well-typed hashed atoms with respect to vtable and otable. This last condition allows to reject formulas like $\neg(1 = \mathsf{true}) \lor \neg(\mathsf{true} = 2) \lor (1 = 2)$ which is correct from the transitivity point of view but is interpreted in Coq by $\mathsf{false} \lor \mathsf{false} \lor 1 = 2$

5.2 A small checker to compute congruence closure

The theory of congruence closure is at the heart of all SMT solvers. In our term representation, equality is represented as a binary operator (Oeq) that is parameterised by the representation of the type on which equality operates.

Consider the proof of unsatisfiability of the formula $\neg(f\,a = f\,b) \wedge b = c \wedge a = c$, which belongs to the congruence closure fragment. It creates the following tables:

$$\begin{array}{l}
\mathsf{ftable}\ = [|\,\mathsf{Ftrue};\ \mathsf{Fatom}\ 5;\ \mathsf{Fatom}\ 6;\ \mathsf{Fatom}\ 7;\ \mathsf{Fand}\ [|3;\ 4;\ 6|];\mathsf{Fatom}\ 8\,|];\\
\mathsf{atable} = [|\,\mathsf{Avar}\ 0;\ \mathsf{Avar}\ 1;\ \mathsf{Avar}\ 2;\ \mathsf{Aapp}\ (\mathsf{Oidx}\ 0)\ [0];\ \mathsf{Aapp}\ (\mathsf{Oidx}\ 0)\ [1];\\
\qquad\qquad \mathsf{Aapp}\ (\mathsf{Oeq}\ \mathsf{TZ})\ [3;4];\ \mathsf{Aapp}\ (\mathsf{Oeq}\ \mathsf{TZ})\ [1;2];\\
\qquad\qquad \mathsf{Aapp}\ (\mathsf{Oeq}\ \mathsf{TZ})\ [0;2];\mathsf{Aapp}\ (\mathsf{Oeq}\ \mathsf{TZ})\ [0;1]\,|]\\
\mathsf{vtable} = [|\,(\mathsf{TZ},\ a);(\mathsf{TZ},\ b);(\mathsf{TZ},\ c)\,|]\ \ \mathsf{otable} = [|\,(([\mathsf{TZ}],\ \mathsf{TZ}),\ f)\,|]\ \ \mathsf{ttable} = [|\,|]
\end{array}$$

where the formula is at location 4 of ftable. Note that location 5 is not necessary to encode the formula but for its proof (the same thing happens for the location 8 of atable). This is explained later.

Our checker is only capable of producing clauses obtained by instantiating one of these three theorems:

- transitivity: $x_1 \neq x_2 \vee \cdots \vee x_{n-1} \neq x_n \vee x_1 = x_n$
- function congruence: $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee f\ x_1\ \ldots\ x_n = f\ y_1\ \ldots\ y_n$
- predicate congruence: $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee \neg P\ x_1\ \ldots\ x_n \vee P\ y_1\ \ldots\ y_n$

The small certificates are ($\mathsf{eq\_trans}\ c$), ($\mathsf{eq\_congr}\ c$) and ($\mathsf{eq\_congr\_pred}\ c$) where c represents the candidate clause to be produced. This explains why the tables of our previous example had more than the atoms of the initial formula. They also contain the atoms of the theory lemmas. The small checkers for these certificates only have to verify that c is indeed an instantiation of their corresponding theorem. For instance, the small checker for $\mathsf{eq\_trans}\ [l_1; \ldots; l_n; l]$ verifies that:

- l is even and each l_i is odd;
- $l/2$ refers to $\mathsf{Aapp}\ (\mathsf{Oeq}\ t)\ [a; b]$ and each $l_i/2$ refers to $\mathsf{Aapp}\ (\mathsf{Oeq}\ t_i)\ [a_i; b_i]$;
- a equals a_1, b equals b_n and for $1 \leq i < n$, a_i equals b_{i+1}.

Note that all the equality tests over atoms are just equalities over integers thanks to our maximal sharing of atoms. Furthermore, we do not need to check type equality between the t_i since the small checkers assume that the atom table is always well-typed.

Our modular checker can combine these three simple rules and the resolution checker to derive the empty clause from an unsatisfiable formula. In our example, the checker starts with the initial formula $\neg(f\,a = f\,b) \wedge b = c \wedge a = c$. After evaluating the part of the certificate dedicated to CNF computation, the state contains the clauses $[|[3]; [4]; [6]; [0]; [0]|]$ and what is left to be evaluated is

$$([(3, \mathsf{eq\_trans}\ [5; 7; 10]); (4, \mathsf{eq\_congr}\ [11; 2]); (0, \mathsf{res\_certif}[|3; 1; 2; 4; 0|])], 0).$$

The computation then proceeds like this:

$$[|[3]; [4]; [6]; [0]; [0]|] \overset{3,\mathsf{eq\_trans}\ [5;7;10]}{\rightsquigarrow} [|[3]; [4]; [6]; [5; 7; 10]; [0]|] \overset{4,\mathsf{eq\_congr}\ [11;2]}{\rightsquigarrow}$$

$$[|[3]; [4]; [6]; [5; 7; 10]; [11; 2]|] \overset{0,\mathsf{res\_certif}[|3;1;2;4;0|]}{\rightsquigarrow} [|[]; [4]; [6]; [5; 7; 10]; [11; 2]|]$$

5.3 A small checker for linear arithmetic

The tactic `lia` [5] of Coq proves any valid formula of linear arithmetic. It already contains a checker based on Farkas' certificates:

$$\mathsf{lia\_check} \;:\; \mathsf{lia\_formula} \rightarrow \mathsf{lia\_certif} \rightarrow \mathsf{bool},$$

Note that `lia` uses a different representation (lia_formula). It also provides a proof of correction for this checker. We choose to use lia_check to build a small checker for our modular interface. Thus, the small certificate for linear arithmetic is simply (lia_certif c F) where c is the candidate clause and F of type lia_certif.

In order to validate the clause c, the small checker first calls the function lia_formula_of to translate c into an equisatisfiable formula f of type lia_formula, and then calls (lia_check f F). The correctness of this small checker relies on the correctness of the `lia` checker and on the correctness of our translation.

5.4 The simplifier small checker

For more efficiency, most SMT solvers use on-the-fly term rewriting and simplification and do not give proof witnesses for these simplifications. Furthermore, sometimes the formula needs to be preprocessed before being sent to an external solver, again for efficiency reasons. In consequence, the formula f' proved by the proof witness can be slightly different from the initial formula f one wanted to prove. We have thus developed a dedicated small checker that verifies that a formula f is equivalent to f'. Our checker is able to prove equivalence through associativity of conjunction and disjunction, double negation, symmetry of equality, and simple rewriting of linear inequations (such as: $a \geq b \equiv b \leq a$). It is implemented by a simple simultaneous recursive descent of f and f'. Only the symmetry of equality requires some backtracking.

6 Building a Coq tactic

To build an actual tactic out of our certified checker, we follow the usual steps for reflexive tactics. The first step is reification: given a formula f in Coq on a decidable domain, we have to build 5 tables and a literal l, such that the interpretation of l with respect to these tables is $\neg f$. The second step is to find a certificate that shows that $[l]$ is unsatisfiable. This is done by calling the SAT or SMT solver. We need to translate the problem into the solver input format. Then, the solver returns a proof witness, that we transform into a certificate.

During the first translation, we sometimes need to do some pre-processing. For example, ZChaff only accepts CNF formulas, so the CNF transformation is done before sending it. Also, the CNF transformation of veriT is more efficient if disjunctions and conjunctions are considered as n-ary operators (and not binary like in Coq), so we flatten the formula before sending it. The justification of this pre-processing is the prelude of the certificate.

The transformation of proof witnesses into certificate requires more work. We first need to update our tables so they contain all the formulas of the theory lemmas. Second, we have to transform each step of the proof witness into a sequence of small certificates. In the easiest cases, the solver gives *exactly* what we expect. This is the case, for instance, of the resolution chains produced by SAT solvers and the CNF transformation produced by veriT. In other cases, very lightweight modifications are necessary. For instance, the format for congruence closure of veriT automatically removes duplicates literals. It generates $\neg x = y \vee f\ x\ x = f\ y\ y$ while our certificate expects $\neg x = y \vee \neg x = y \vee f\ x\ x = f\ y\ y$. Finally, in the worst cases, we may have to rebuild completely the certificate. This is the case for veriT where theory lemmas for linear arithmetic come without justification. In this precise example, we use the external solver of `lia` to produce the Farkas' certificate.

Finally, the compactness of the certificate is very important. So, when it is not done by the solver, we prune it by removing all the justifications of unused clauses. It is during this phase that we compute the minimal required size of the array of clauses and perform clause allocation.

This work results into two new Coq tactics called `zchaff` and `verit` .

7 Results and comparison with other works

7.1 Related works

The work presented here extends what is described in [3] in several ways. First, we complemented the SAT checker with checkers for CNF computation, congruence closure, differential logic and linear arithmetic. To do so with a great modularity, the format of certificates has been rethought: the idea of small and main checkers makes it easier to add new theories. Second, we can formally check proof witnesses generated by the veriT solver, which combines all these theories. Finally, we use our checker to enhance the automation of Coq by defining new safe reflexive decision procedures.

Several SAT and SMT solvers have been integrated in LCF style interactive theorem provers including CVC Lite in HOL Light [11], haRVey in Isabelle/HOL [8], Z3 in HOL and Isabelle/HOL [6]. To our knowledge, our work is the first integration relying on proof witnesses in a proof assistant based on Type Theory. In the following, we will focus on the comparison with proof reconstruction in Isabelle/HOL of ZChaff [15] and Z3 [6] (this corresponds to the state-of-the-art). We point out that comparison for theories has to be considered with care since we do not use the same SMT solver.

Another approach is to write the SAT or SMT solver directly inside the proof assistant and formally prove its correctness. This is the approach followed in [10]. It has the advantage to validate the algorithms at work in the prover but is sensitive to any change, any optimization in the proof search. We compare the two approaches.

7.2 Experiments

All the experiments have been conducted on an Intel Quad Core processor with 2.66GHz and 4Gb RAM, running Linux. Our code which served for the experiments is available online [1]. It requires the native version of Coq [7]. It represents around $6,000$ lines of Coq code and $8,000$ lines of Ocaml code. The Coq code for our shared term representation is about $1,000$ lines, the SAT part $1,200$, the CNF part $1,500$, the EUF 600, the LIA part $1,500$ and the simplifier part 500. The complete checker corresponds to $1,000$ lines of Coq code, the other $5,000$ are for specifications and proofs.

SAT verification We first compare our combination of the main checker with the small checker of resolution chains for ZChaff in Coq with proof reconstruction for ZChaff in Isabelle/HOL written by Alwen Tiu and Tjark Weber. We use Isabelle 2009-1 (running with Poly/ML 5.2) and ZChaff 2007.3.12.

We run ZChaff on a database of 151 unsatisfiable industrial benchmarks from SAT Race'06 and '08 with a timeout of 300 seconds. These benchmarks range from 300 to 2.3 million variables and from $1,800$ to 8.9 million clauses. When ZChaff succeeds in the given time, it produces a proof witness whose size range from 41Kb to 205Mb. In that case, we run our checker and the Isabelle/HOL checker on it with a timeout of 300 seconds. Table 1 presents the number of benchmarks solved by ZChaff, and among them, the number of proof witnesses successfully checked by Isabelle/HOL and Coq. The times are the mean of the times for the 57 benchmarks on which ZChaff, Coq and Isabelle/HOL all succeeded, in seconds. Errors in Isabelle/HOL were due to timeouts.

It appears that Coq can check all the proof witnesses given by ZChaff in the given time. This is not surprising since our checker appears to be faster than ZChaff itself. However, the Isabelle/HOL checker is slower than ZChaff, which explains that only 72% of the proof witnesses can be checked without timeout.

The three curves on the left of Figure 1 present the number of benchmarks solved along the time by ZChaff, Isabelle and Coq. It clearly shows that the Coq checker is far faster verifying results than ZChaff is building them; the main time consumed by our combination is taken by ZChaff. However, the limiting factor of the ZChaff and Isabelle/HOL combination is Isabelle/HOL.

SMT verification We now compare our combination of the main checker with the small checkers of resolution chains, CNF computation, congruence closure, differential logic and linear integer arithmetic for veriT in Coq with proof reconstruction for Z3 in Isabelle/HOL written by Sascha Böhme and Tjark Weber. We use Isabelle 2009-1 (running with Poly/ML 5.2), Z3 2.19 and the development version of veriT.

We took a database of unsatisfiable industrial benchmark from the SMT-LIB [2] for theories QF_UF (congruence closure), QF_IDL (differential logic) and QF_LIA (linear integer arithmetic). It is important to notice that veriT is not completely proof producing for QF_LIA, so we selected a subset of the benchmarks where veriT returns either *unknown* or *unsatisfiable* with a proof witness. On the one hand, we run veriT, followed by our Coq checker when veriT succeeds. On the other hand, we run Z3, followed by the Isabelle/HOL checker when Z3 succeeds. Each run has a timeout of 300 seconds. The mean of the

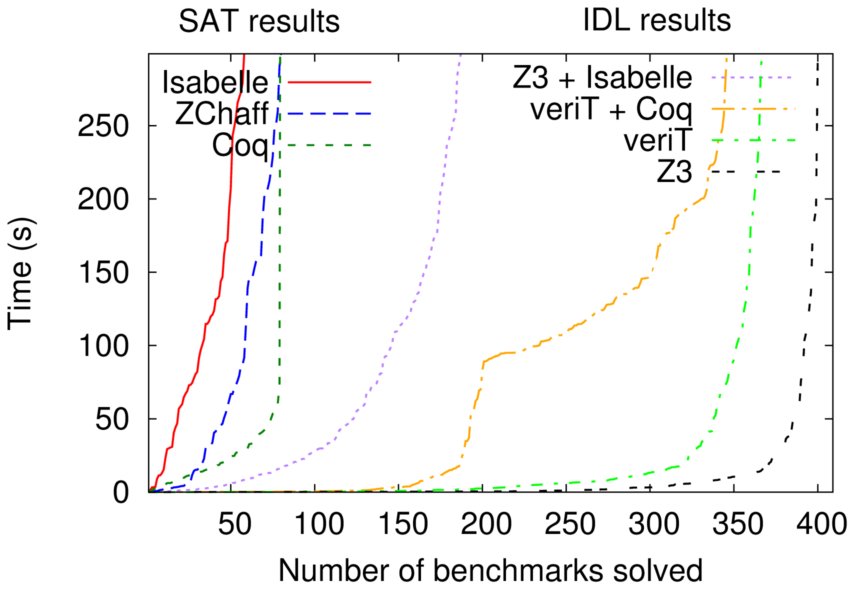

Figure 1: Experiments on industrial benchmarks

Solved ZChaff			Isabelle/HOL checker			Coq checker		
#	%	Time	#	%	Time	#	%	Time
79	52	51.9	57	38	100.	79	52	17.5

Table 1: SAT benchmarks

Benchmarks		Solved Z3			Solved veriT			Isabelle/HOL checker			Coq checker		
Logic	#	#	%	Time	#	%	Time	#	%	Time	#	%	Time
QF_UF	1852	1834	99	2.5	1816	98	6.5	1775	96	25.8	1804	97	1.4
QF_IDL	409	402	98	0.6	368	90	6.3	190	46	55.2	349	85	37.8
QF_LIA	116	107	92	0.7	98	84	11.6	96	83	46.6	98	84	3.1

Table 2: SMT benchmarks

	dplln	zchaff		dplln	zchaff		cc	verit		cc	verit
H_7	28.0	0.2	deb_{700}	111.5	0.8	$F(13,5,8)$	0.5	0.1	D_5	2.3	0.3
H_8	262.7	1.2	deb_{800}	147.9	1.0	$F(25,13,1)$	1.3	0.1	D_8	24.9	1.1
H_9	-	1.6	deb_{900}	201.6	1.2	$F(25,15,5)$	0.5	0.2	D_{10}	118.7	2.2
H_{10}	-	6.7	deb_{1000}	260.4	1.5	$F(25,24,24)$	16.9	0.1	D_{15}	-	45.7

Table 3: Comparison with Ergo in Coq

sizes of Z3 proof witnesses is 12Mb, and the mean of the sizes of veriT proof witnesses is 7.7Mb. Table 2 presents the number of benchmarks solved by Z3 and veriT, and among them, the number of proof witnesses successfully checked by Isabelle/HOL and Coq. The times are the mean of the times for the benchmarks on which Z3, veriT, Isabelle/HOL and Coq all succeeds, in seconds. Errors in Coq were due to timeouts, and in Isabelle/HOL to timeouts and failures.

It appears that Coq can check a large part of the proof witnesses given by veriT (98.6%) whereas Isabelle/HOL can check 88.0% of the proof witnesses given by Z3. As a result, even if Z3 can solve more benchmarks than veriT, the number of benchmarks solved by veriT combined with Coq is greater than the number of benchmarks solved by Z3 combined with Isabelle/HOL. Moreover, our combination is faster than the combination of Z3 with Isabelle/HOL. These results can be explained in great part by the fact that veriT gives much smaller proof witnesses. For instance, for logic QF_IDL, in average, Z3 proof witnesses are 7.9 times bigger than veriT proof witnesses in terms of storing. The quality of veriT proof witnesses strengthens the fact we use it, even if there exists currently more performing SMT solvers. We have been told that the limitation of proof witnesses for LIA should disappear soon.

The four curves on the right of Figure 1 present the number of benchmarks solved along the time by the solvers and their combinations. They clearly indicate that our approach compares well with respect to [6].

Tactics We compare our `zchaff` and `verit` tactics with the reflexive tactics `dplln` and `cc` from Stephane Lescuyer's SMT solver Ergo written in Coq. To do so, we use the same formulas that are presented in Section 11.2 of [10]:

– for SAT:
  - the famous pigeon hole formulas which are unsatisfiable
  - the de Bruijn formulas: $deb_n = \forall x_0, \dots, x_{2n}, (x_{2n} \leftrightarrow x_0) \vee \bigvee_{i=0}^{2n-1} (x_i \leftrightarrow x_{i+1})$

– for EUF:
  - the formulas $FP(n,m,k) = \forall f x, f^n(x) = x \to f^m(x) = x \to f^k(x) = x$ which are true for any n, m, k such that k is a multiple of $\gcd(n,m)$
  - the formulas $D_n =$ $$\forall f, \left(\bigwedge_{i=0}^{n-1} (x_i = y_i \wedge y_i = f(x_{i+1})) \vee (x_i = z_i \wedge z_i = f(x_{i+1})) \right) \to x_0 = f^n(x_n)$$

Results are presented in Table 3. Times are in seconds. We see that our `zchaff` and `verit` tactics here clearly outperform `dplln` and `cc`. This is not surprising since ZChaff and veriT have more efficient algorithms than Ergo. Note it may be difficult to change Ergo's algorithm since it would involve redoing many correctness proofs; the certificate approach is more flexible here. If we store proof witnesses, `zchaff` and `verit` get faster at rather small storage cost: in our examples, the largest proof witness is 41Mb large for D_{15}.

Regarding other existing Coq tactics, `zchaff` is far faster than `tauto`, and `verit` is similar to `congruence`. However, these latter ones do not solve the same goals, since `verit` can solve goals including congruence and propositional reasoning, and `congruence` can deal with inductive data-types.

8 Conclusion and future works

Compared to what the authors call "proof reconstruction" in [6], what we have presented here is much closer to program verification. We have developed inside Coq a program that checks traces generated by SAT and SMT solvers. A particular care has been given to the efficiency of the data representation for clauses and atoms. Even with the limited computing power available inside Coq, the checker is rather efficient: it is able to check in reasonable time huge proof witnesses coming from challenging benchmarks and it compares well with state of the art implementations in Isabelle/HOL.

From the methodology point of view, what we have done is very close to [13]. In this work, the authors have developed a checker for SMT proofs using LFSC. The main difference is that they delegate the verification of the SAT part to an external checker (written in C++). Here, we do everything within the logic.

We also took a special care in being generic: for example the same checker is used for ZChaff and veriT. This relies on the generic format we use after translating proof witnesses into certificates. So we expect the checker to be easily extensible. For the moment, veriT is the only SMT solver that is connected with Coq. We hope that our format of certificate could also be used successfully to connect to other proof producing solvers. Our next step is to integrate Z3.

The checker has also been proved correct. Using the technique of *proof by reflection*, it made it possible to derive a safe and automatic proof procedure within Coq. Formulas usually proved in Coq are rather small so a far less efficient checker could have been sufficient. Still, we believe that our work opens interesting new perspectives of using brute force methods for doing proof with Coq automatically. For example, one could encode the small problem she/he has to prove as a huge Boolean formula that our SAT tactic can solve instantaneously.

Surprisingly the difficult part of this work was more the actual design of the certificate and obtaining a good computational behaviour for the checker than performing the correctness proofs. This is largely due to the fact that we are not proving the full functional correctness of the checker. We are just proving that if the checker replies true, the theorem is valid. This makes a big difference for the proof effort. This reduces drastically the size of the invariants we had to prove and clearly makes the proof of such a large piece of code tractable in Coq.

For future works, our priority is clearly to increase the expressiveness of the formulas we can deal with. In particular, if we want our tool to be widely used by the Coq community, being able to deal with quantified formulas and user-defined functions is a must-have. For quantifiers, it has not been done yet mostly because the current version of veriT does not produce proof witnesses. Though, this should be available in the next version of the system. For definitions, more work has to be done since the type system of Coq is more powerful than the one proposed by the SMT-LIB standard. Other extensions we envision concern non-linear arithmetic, arrays and bit vectors.

Acknowledgments Pascal Fontaine's responsiveness was crucial for this work. We wish to thank Sascha Böhme and Tjark Weber for the details and source code they gave us to reproduce the Isabelle experiments. We finally thank Christine Paulin, Guillaume Melquiond and Sylvain Conchon for their help concerning the comparison with `dpllm` and `cc` in Coq. Anonymous referees provided helful constructive remarks.

References

1. Source code of the development, `http://www.lix.polytechnique.fr/~keller/Recherche/smtcoq.html`
2. SMT-LIB, `http://www.smtlib.org`
3. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with Imperative Features and Its Application to SAT Verification. In: Kaufmann and Paulson [9], pp. 83–98
4. Barendregt, H., Barendsen, E.: Autarkic Computations in Formal Proofs. J. Autom. Reasoning 28(3), 321–336 (2002)
5. Besson, F.: Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In: TYPES. LNCS, vol. 4502, pp. 48–62. Springer (2006)
6. Böhme, S., Weber, T.: Fast LCF-Style Proof Reconstruction for Z3. In: Kaufmann and Paulson [9], pp. 179–194
7. Dénès, M.: Coq with native compilation, `https://github.com/maximedenes/native-coq`
8. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L., Tiu, A.: Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In: TACAS. LNCS, vol. 3920, pp. 167–181. Springer (2006)
9. Kaufmann, M., Paulson, L.C. (eds.): Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings, Lecture Notes in Computer Science, vol. 6172. Springer (2010)
10. Lescuyer, S., Conchon, S.: Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In: FroCos. LNCS, vol. 5749, pp. 287–303. Springer (2009)
11. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. ENTCS 144(2), 43–51 (2006)
12. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). J. ACM 53(6), 937–977 (2006)
13. Oe, D., Stump, A.: Extended Abstract: Combining a Logical Framework with an RUP Checker for SMT Proofs. In: Lahiri, S., Seshia, S. (eds.) Proceedings of the 9th International Workshop on Satisfiability Modulo Theories (Snowbird, USA) (2011)
14. Tseitin, G.S.: On the complexity of proofs in propositional logics. Automation of Reasoning: Classical Papers in Computational Logic (1967–1970) 2 (1983)
15. Weber, T.: SAT-based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Institut für Informatik, Technische Universität München, Germany (Apr 2008), `http://www.cl.cam.ac.uk/~tw333/publications/weber08satbased.html`