Modular SMT Proofs for Fast Reflexive Checking inside Coq[⋆]

Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie

INRIA Rennes – Bretagne Atlantique, France

Abstract. We present a new methodology for exchanging unsatisfiability proofs between an untrusted SMT solver and a sceptical proof assistant with computation capabilities like Coq. We advocate modular SMT proofs that separate boolean reasoning and theory reasoning; and structure the communication between theories using Nelson-Oppen combination scheme. We present the design and implementation of a Coq reflexive verifier that is modular and allows for fine-tuned theory-specific verifiers. The current verifier is able to verify proofs for quantifier-free formulae mixing linear arithmetic and uninterpreted functions. Our proof generation scheme benefits from the efficiency of state-of-the-art SMT solvers while being independent from a specific SMT solver proof format. Our only requirement for the SMT solver is the ability to extract unsat cores and generate boolean models. In practice, unsat cores are relatively small and their proof is obtained with a modest overhead by our proof-producing prover. We present experiments assessing the feasibility of the approach for benchmarks obtained from the SMT competition.

1 Introduction

During the past few years, interactive proof assistants have been very successful in the domain of software verification and formal mathematics. In these areas the amount of formal proofs is impressive. For Coq, one of the mainstream proof assistants, it is particularly impressive to see that so many proofs have been done with so little automation. In his POPL'06 paper on verified compilation [19, page 12], Leroy gives the following feedback on his use of Coq:

> Our proofs make good use of the limited proof automation facilities provided by Coq, mostly eauto (Prolog-style resolution), omega (Presburger arithmetic) and congruence (equational reasoning). However, these tactics do not combine automatically and significant manual massaging of the goals is necessary before they apply.

Yet, efficient algorithms exist to combine decision procedures for arithmetic and equational reasoning. During the late '70s, Nelson and Oppen have proposed a cooperation schema for decision procedures [23]. This seminal work, joint with

[⋆] This work was partly funded by the ANR DeCert, FNRAE ASCERT and Région Bretagne CertLogS projects.

the advances in SAT-solving techniques, has greatly influenced the design of modern SMT solvers [11, 4, 8]. Nowadays, these solvers are able to discharge enormous formulae in a few milliseconds. A proof assistant like Coq would gain a lot in usability with only a small fraction of this speed and automation.

Integrating such algorithms in a proof assistant like Coq is difficult. Coq is a *sceptical* proof assistant and therefore every decision procedure must justify its verdict to the proof kernel with an adequate typable proof term. We distinguish between two different methods for integrating a new decision procedure in a system like Coq. First, we can rely on an external tool, written in an other programming language than Coq, that builds a Coq proof term for each formula it can prove. The main limit of this approach is the size of the exchanged proof term, especially when many rewriting steps are required [17]. Second, we can *verify the prover* by directly programming it in Coq and mechanically proving its soundness. Each formula is then proved by running the prover inside Coq. Such a *reflexive* approach [17] leads to short proof terms but the prover has to be written in the constrained environment of Coq. Programming a state-of-the-art SMT solver in a purely functional language is by itself a challenging task; proving it correct is likely to be impractical — with a reasonable amount of time.

Our implementation is a trade-off between the two previous extreme approaches: we program a reflexive verifier that uses hints (or certificates) given by an untrusted prover (programmed in OCaml). Such an approach has the following advantages: 1) The verifier is simpler to program and prove correct in Coq than the prover itself; 2) Termination is obtained for free as the number of validation steps is known beforehand; 3) The hint conveys the minimum amount of information needed to validate the proof and is therefore smaller than a genuine proof term. This last point is especially useful when a reasoning takes more time to explain than the time to directly perform it in the Coq engine. Recall that the Coq reduction engine [16] allows the evaluation of Coq programs with the same efficiency as OCaml programs. This design allows us to find a good trade-off between proof time checking and proof size.

The mainstream approach for validating SMT proofs [15, 20, 6] requires a tight integration with an explanation-producing SMT solver. The drawbacks are that explanations may contain too much or too little details and are solver specific. Despite on-going efforts, there is no standard SMT proof format. In contrast, our methodology for generating unsatisfiability proofs is based on a coarser-grained interaction with the SMT solver. Our current implementation only requires an SMT solver producing unsat cores and boolean models. In practice, unsat cores are relatively small and their proofs are obtained with a modest overhead by our hand-crafted proof-producing prover. Our prover is co-designed with the Coq verifier and therefore has the advantage of generating the exact level of details needed to validate the proof. The contributions of this work can be summarised as follows:

– A new methodology for exchanging unsatisfiability proofs between an untrusted SMT solver and a sceptical proof assistant with computation capabilities like Coq. Our proof format is modular. It separates boolean reasoning

from theory reasoning, and structures the communication between theories using the Nelson-Oppen combination scheme.
- A modular reflexive Coq verifier that allows for fine-tuned theory specific verifiers exploiting as much as possible the efficient Coq reduction engine. The current verifier is able to verify proofs for quantifier-free formula mixing linear arithmetic and uninterpreted functions.
- A proof-generation scheme that uses state-of-the-art SMT solvers in a black-box manner and only requires the SMT solvers to extract unsat-cores and boolean models. (These features are standardised by the SMT-LIB 2 format.)
- A proof-producing multi-theory prover that generate certificates to discharge theory lemmas, *i.e.*, unsat-cores. It is based on a standard Nelson-Oppen combination of a simplex prover for linear arithmetic and a congruence closure engine for uninterpreted functions.

To discharge SAT proofs, we use the reflexive boolean SAT verifier developed by Armand *et. al.* [2, 1]. We only consider ground formula and therefore quantifier instantiation is not in the scope of this paper.

Our Coq development, our proof-producing prover and the benchmarks of Section 6 are available at `http://www.irisa.fr/celtique/ext/chk-no`.

2 Overview

This section gives an overview of some concepts used in state-of-the-art SMT solvers. It presents the SMT solver approach in three layers. Our proof format follows closely this layered presentation. We focus on formulae that must be proved unsatisfiable. We take as running example the following quantifier free *multi-theory* formula, that mixes specifically the theories of equality and Uninterpreted Functions (UF) and Linear Real Arithmetic (LRA).

$$f(f(x)-f(y)) \neq f(z) \ \wedge\ x \leq y \ \wedge\ ((y+z \leq x \wedge z \geq 0) \vee (y-z \leq x \wedge z < 0)) \quad (1)$$

For UF, a literal is an equality between multi-sorted ground terms and a formula is a conjunction of positive and negative literals. The axioms of this theory are reflexivity, symmetry and transitivity, and the congruence axiom $\forall a \forall b, a = b \Rightarrow f(a) = f(b)$ for functions. For LRA, a literal is a linear constraint $c_0 + c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie 0$ where $(c_i)_{i=0..n} \in \mathbb{Q}$ is a sequence of rational coefficients, $(x_i)_{i=1..n}$ is a sequence of real unknowns and $\bowtie \in \{=, >, \geq\}$. Following Simplify [14], disequality is managed on the UF side. Therefore, a formula is a conjunction of positive literals.

From input formula to unsat multi-theory conjunctions. The lazy SMT solver approach [13] abstracts each atom of the unsatisfiable input formula by a distinct propositional variable, uses a SAT solver to find a propositional model of the formula, and then checks that model against the theory. Models that are incompatible with the theories are discarded by adding a proper lemma to the original formula. This process is repeated until all possible propositional models have been explored. For the given running example, the initial boolean abstraction (2) is $A \wedge B \wedge ((C \wedge D) \vee (E \wedge \neg D))$ with the following mapping

A	B	C	D	E
$f(f(x) - f(y)) \neq f(z)$	$x \leq y$	$y + z \leq x$	$z \geq 0$	$y - z \leq x$

(3)

The first boolean model, $A{:}True, B{:}True, C{:}True, D{:}True, E{:}False$, corresponds to the conjunction

$$(f(f(x) - f(y)) \neq f(z)) \wedge (x \leq y) \wedge (y + z \leq x) \wedge (z \geq 0) \wedge \neg(y - z \leq x)$$

and can be proved unsatisfiable by a multi-theory solver. Hence the boolean model is discarded by adding the theory lemma $\neg(A \wedge B \wedge C \wedge D \wedge \neg E)$ to the original boolean formula. The process is repeated until no more boolean model can be found, showing that the current boolean formula is unsatisfiable.

This process can be speed up with several optimisations. First, theory lemmas can by obtained from *unsat cores*, *i.e.*, minimal subsets of a propositional model still unsatisfiable for the theories. Some SMT solvers also check partial models incrementally against the theory in order to detect conflicts earlier. Second, the multi-theory solver may discover propagation lemmas, *i.e.*, theory literals that are consequence of partial models. In a boolean form, such lemmas allow the SAT solver to reduce further its search tree. In all cases, a witness of unsatisfiability of the input formula is given by a proof of unsatisfiability of a boolean formula composed of the boolean abstraction of the input formula, plus boolean lemmas that correspond to negation of unsatisfiable multi-theory conjunctions. This leads to the first proof rule of our proof format:

$$\frac{\begin{array}{c} f^{\mathbb{B}}, \neg C_1^{\mathbb{B}}, \ldots, \neg C_n^{\mathbb{B}} \vdash_{Boolean} cert^{\mathbb{B}} : \text{False} \\ \forall i = 1, \ldots, n,\ \sigma(C_i^{\mathbb{B}}) \vdash_{NO} cert_i : \text{False} \end{array}}{\sigma(f^{\mathbb{B}}) \vdash_{SMT} (\sigma, (cert^{\mathbb{B}} : f^{\mathbb{B}}), [(cert_1 : C_1^{\mathbb{B}}), \ldots, (cert_n : C_n^{\mathbb{B}})]) : \text{False}}$$

In the following, a judgement of the form $\Gamma \vdash cert : F$ means that formula F can be deduced from hypotheses in Γ, using certificate *cert*. In the judgement $\sigma(f^{\mathbb{B}}) \vdash_{SMT} cert$: False, the certificate *cert* is composed of three elements: a mapping σ between propositional variables and theory literals, a boolean abstraction $f^{\mathbb{B}}$ of F and a list $C_1^{\mathbb{B}}, \ldots, C_n^{\mathbb{B}}$ of conjunctions of boolean variables. For this judgement to establish that the ground formula F is unsatisfiable, several premises have to be verified by the reflexive checker. First, $\sigma(f^{\mathbb{B}})$ must be reducible to F. It means that the boolean abstraction is just proposed by the untrusted prover and checked correct by the reflexive verifier. Second, the conjunction of $f^{\mathbb{B}}$ and all the negation $\neg C_1^{\mathbb{B}}, \ldots, \neg C_n^{\mathbb{B}}$ must be checked unsatisfiable with a boolean verifier. This verifier can be helped with a dedicated certificate $cert^{\mathbb{B}}$ (for example taking the form of a refutation tree). As explained before, the current paper does not focus on this specific part. We instead rely on the reflexive tactic proposed by Armand *et al.,* [2, 1]. At last, every multi-theory conjunction $\sigma(C_i^{\mathbb{B}})$ must be proved unsatisfiable with a dedicated certificate $cert_i$. This is done with the judgement $\vdash_{NO}$ which is explained in the next subsection. For our example, the certificate would be composed of the mapping (3), the boolean abstraction (2), and the conjunctions $(A \wedge B \wedge C \wedge D)$ and $(B \wedge \neg D \wedge E)$.

Generation of SMT proofs. To generate our SMT proof format, we implement the simple SMT loop discussed earlier using SMT-LIB 2 scripts to interface with off-the-shelf SMT solvers. The SMT-LIB 2 [3] exposes a rich API for SMT solvers that makes this approach feasible. More precisely, SMT-LIB 2 defines scripts that are sequence of commands to be run by SMT solvers. The `asserts f` command adds the formula `f` to the current context and the `check-sat` command checks the satisfiability of the current context. If the context is satisfiable (`check-sat` returns `sat`), the `get-model` command returns a model. Otherwise, the `get-unsat-core` command returns a so-called unsat core that is a minimised unsatisfiable subset of the current context.

The SMT loop we described is implemented using SMT-LIB 2 compatible off-the-shelf SAT and SMT solvers (we chose Z3 for both). Given an initial unsatisfiable formula, the protocol is the following. To begin with, the boolean abstraction of the input formula is computed and sent to the SAT solver. For each boolean model returned by the SAT solver, the SMT solver is asked for an unsat core, whose negation is sent to the SAT solver. The loop terminates when the SAT solver returns an `unsat` status. Once all the unsat cores have been discovered, our OCaml prover generate certificates for them using the proof system described in Section 3 and Section 4. This untrusted certifying prover implements the Nelson-Oppen algorithm [23] described below. Overall, unsat cores tend to be very small (10 literals on average) and therefore our certifying prover is not the bottleneck. The boolean proof is obtained by running an independent certifying SAT solver. Unlike SMT solvers, DPLL-based SAT solvers have standardised proofs: *resolution proofs*.

Our prototype could be optimised in many ways. For instance, a boolean proof could be obtained directly without re-running a SAT solver. Our scheme would also benefit from a SMT-LIB 2 command returning all the theory lemmas (unsat cores are only a special kind of those) needed to reach a proof of unsatisfiability.

From unsat multi-theory conjunctions to unsat mono-theory conjunctions. In the previous steps, the theory solvers have been fed with conjunctions of multi-theory literals. We now explain the Nelson-Oppen (NO) algorithm that is a sound and complete decision procedure for combining infinitely stable theories with disjoint signatures [23]. Figure 1 presents the deduction steps of this procedure on the previous first theory conflict (corresponding to the boolean conjunction $(A \wedge B \wedge C \wedge D)$). We start from the formula at the top of Figure 1 and first apply a *purification* step that introduces sufficiently many intermediate variables to flatten each terms and dispatch *pure* formulae to each theory. Then each theory exchanges new equalities with the others until a contradiction is found.

Theory exchange is modelled by the Nelson-Oppen proof rule given below.

$$\frac{\begin{array}{c}\Gamma_i \vdash_{T_i} cert_i : (\Gamma_i', eqs) \\ \bigwedge_{x_k = y_k \in eqs} (\Gamma_1[j \mapsto x_k = y_k], \ldots, \Gamma_i', \ldots, \Gamma_n[j \mapsto x_k = y_k] \vdash_{NO} sons[k] : \text{False})\end{array}}{\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (cert_i, sons) : \text{False}}$$

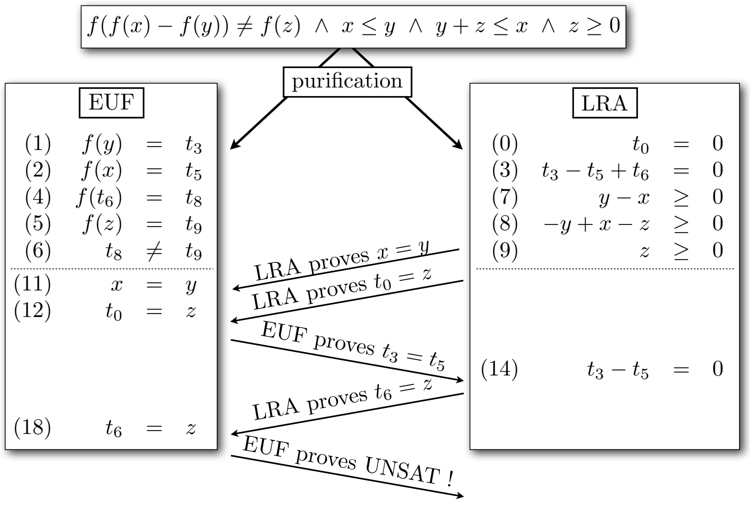

Fig. 1: Example of Nelson-Oppen equality exchange

We assume here a collection of n theories $T_1, \ldots, T_n$. In this judgement Γ_i represents an environment of pure literals of theory T_i. Each theory is equipped with its own deduction judgement $\Gamma_i \vdash_{T_i} cert_i : (\Gamma_i', eqs)$ where Γ_i and Γ_i' are environments of theory T_i, $cert_i$ is a certificate specific to theory T_i and *eqs* is a list of equalities between variables. Such a judgement reads as follows: assuming that all the literals in Γ_i hold, we can prove (using certificate $cert_i$) that all the literals in Γ_i' hold and that the disjunction of equalities between variables in *eqs* can be proved. The judgement $\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (cert_i, sons)$: False holds if given an environment $\Gamma_1, \ldots, \Gamma_n$ of the joint theory $T_1 + \ldots + T_n$, the certificate $(cert_i, sons)$ allows to exhibit a contradiction, *i.e.*, False. Suppose that certificate $cert_i$ establishes a judgement of the form $\Gamma_i \vdash_{T_i} cert_i : (\Gamma_i', eqs)$. If the list *eqs* is empty (*i.e.*, represents an empty disjunction), we have a proof that Γ_i is contradictory and therefore the joint environment $\Gamma_1, \ldots, \Gamma_n$ is contradictory and the judgement holds. An important situation is when the list *eqs* is always a singleton during a proof. This corresponds to the case of convex theories for which the Nelson-Oppen algorithm never needs to perform case-splits [23]. In the general case, we recursively exhibit a contradiction for each equality $(x_k = y_k)$ using the k^{th} certificate of *sons*, *i.e.*, $sons[k]$ for a joint environment $(\Gamma_1[j \mapsto x_k = y_k], \ldots, \Gamma_i', \ldots, \Gamma_n[j \mapsto x_k = y_k])$ enriched with the equality $(x_k = y_k)$. For completeness, the index j used to store the equality $(x_k = y_k)$ should be fresh. The judgement holds if all the branches of the case-split over the equalities in *eqs* lead to a contradiction.

For the example given in Figure 1, we start with the sets Γ_{LRA} and Γ_{UF} of LRA hypotheses (resp. UF hypotheses). A first certificate $cert_1^{\text{LRA}}$ is required to prove the equality $x = y$, then a certificate $cert_1^{\text{UF}}$ to prove $t_3 = t_5$, then a certificate $cert_2^{\text{LRA}}$ to prove the equality $t_6 = z$, and at last a certificate $cert_2^{\text{UF}}$

to find a contradiction. The whole reasoning is hence justified by the following certificate: $(cert_1^{\text{LRA}}, \{(cert_1^{\text{UF}}, \{(cert_2^{\text{LRA}}, \{(cert_2^{\text{UF}}, \{\})\})\})\})$.

Discharging unsat mono-theory conjunctions. Each part of the NO proof is theory-specific: each theory must justify either the equalities exchanged or the contradiction found. A LRA proof of $a = b$ is made of two Farkas proofs [27] of $b - a \geq 0$ and $a - b \geq 0$. Each inequality is obtained by a linear combination of hypotheses that preserves signs. For example, the previous certificate $cert_1^{\text{LRA}}$ explains that hypothesis (7) gives $y - x \geq 0$ and $(8) + (9)$ gives $x - y \geq 0$. A UF proof of $a = b$ is made of a sequence of rewrites that allows to reach b from a. For example, the certificate $cert_1^{\text{UF}}$ explains the equality $t_3 = t_5$ with the following rewritings: $t_3 \xrightarrow{\texttt{trans. with (1)}} f(y) \xrightarrow{\texttt{congr. with (11)}} f(x) \xrightarrow{\texttt{trans. with (2)}} t_5$.

The rest of the paper is organised as follows. Section 3 presents the certificate format for the UF theory. Section 4 presents the certificate format for linear arithmetic. We present the modular Nelson-Oppen verifier in Section 5 and give some experiments in Section 6. We discuss related work in Section 7 and conclude in Section 8 with a discussion on further work.

3 Certificate checking and generation for UF

In this section we introduce the certificate language and checker for UF and present an overview of the certifying prover.

Certificate language. A certificate is a list of commands executed in sequence. Each command operates on the state of the checker which is a pair (Γ, eq). The assumption set Γ is a mapping from indices to assumptions, written $\Gamma(i) \mapsto a = b$, and eq is the *current equality*, *i.e.*, the last one proved. Each command corresponds to an axiom or a combination of axioms of the UF theory.

```
Inductive command :=
 | Refl (t : term) | Trans (i : index) (sym : bool)
 |Congr (i : index) (sym : bool) (pos : index) | Push (i : index).
```

The semantics is given by rules on judgements of the form $(\Gamma, eq) \xrightarrow{\texttt{cmd}} (\Gamma', eq')$ where (Γ', eq') is the state obtained after executing the command `cmd` from the state (Γ, eq). The boolean s in `Trans` and `Congr` commands makes symmetry explicit: if $\Gamma(i) \mapsto t = t'$ then we define $\Gamma(i)^{true} \mapsto t' = t$ and $\Gamma(i)^{false} \mapsto t = t'$.

$$\frac{}{\Gamma, . = . \xrightarrow{\texttt{Refl}(y)} \Gamma, y = y} \qquad \frac{\Gamma(i)^s \mapsto t = t'}{\Gamma, x = t \xrightarrow{\texttt{Trans}(i,s)} \Gamma, x = t'} \qquad \frac{\Gamma' = \Gamma[i \mapsto x = t]}{\Gamma, x = t \xrightarrow{\texttt{Push(i)}} \Gamma', x = t}$$

$$\frac{\Gamma(i)^s \mapsto a_p = a'_p}{\Gamma, x = f(a_0..a_p..a_n) \xrightarrow{\texttt{Congr}(i,p,s)} \Gamma, x = f(a_0..a'_p..a_n)}$$

The command `Refl`(y) corresponds to the reflexivity axiom and initialises the current equality with the tautology $y = y$, whatever the previous equality. Subsequent commands will rewrite the right hand side of this equality. The command `Trans`(i, s) updates the right hand side of the current equality: if we can prove that $x = t$ (current equality) and we know that $t = t'$ (equality indexed by i) then we can deduce $x = t'$. The command `Congr`(i, p, s) rewrites a sub-term of the right hand side: in any given context if we can prove $x = f(y)$ (current equality) and we know that $y = z$ (equality indexed by i) then we can deduce $x = f(z)$ and make it the new current equality. The parameter p is used to determine where to rewrite. The command `Push`(i) is used to update the assumption set Γ with the current equality $x = t$, creating a new context $\Gamma' = \Gamma[i \mapsto x = t]$ to be used to evaluate the next commands. It allows us to factorise sub-proofs and is mandatory to keep invariant the depth of terms.

The relation $\Gamma \vdash_{UF} cert_{UF} : (\Gamma^{'}, eqs)$ implements the theory specific judgement seen in Section 2.

$$\frac{\Gamma, z = z \xrightarrow{cert}_{*} \Gamma', x = y}{\Gamma \vdash_{UF} \texttt{UF\_Eq}(cert) : (\Gamma', [x = y])} \qquad \frac{\Gamma, z = z \xrightarrow{cert}_{*} \Gamma', x = y \quad \Gamma(i) \mapsto x \neq y}{\Gamma \vdash_{UF} \texttt{UF\_False}(i, cert) : (\Gamma', nil)}$$

Suppose that we obtain a state $(\Gamma, x = y)$ after processing a list *cert* of commands. The certificate `UF_False`$(i, cert)$ deduces a contradiction if $\Gamma(i) \mapsto x \neq y$ and the certificate `UF_Eq`$(cert)$ deduces the equality $x = y$.

Certificate generation follows closely [24] where the certifying prover maintains a *proof forest* that keeps track of the reasons why two nodes are merged. Besides the usual *merge* and *find* operations, the data structure has a new operator `explain`$(a, b,$ `forest`$)$ which outputs a *proof* that $a = b$ based on `forest`. In our case the proofs are certificates, while in the original approach they were non-redundant unsatisfiable unordered sets of assumptions.

We show below the proof forest corresponding to the UF part of the example of Figure 1. Trees represent equivalence classes and each edge is labelled by assumptions. The prover updates the forest with each *merge*. Two distinct classes can be merged for two reasons: an equality between variables is added or two terms are equal by congruence.

$$\begin{tikzcd}
\boxed{t_0} & \boxed{x} & \boxed{t_5} & \boxed{t_9} \\
\boxed{z} \arrow[u, "(12)\ t_0 = z"'] & \boxed{y} \arrow[u, "(11)\ x = y"'] & \boxed{t_3} \arrow[u, "\begin{array}{l}(2)\ f(x) = t_5 \\ (1)\ f(y) = t_3\end{array}"'] & \boxed{t_8} \arrow[u, "\begin{array}{l}(5)\ f(z) = t_9 \\ (4)\ f(t_6) = t_8\end{array}"'] \\
\boxed{t_6} \arrow[u, "(18)\ z = t_6"'] & & &
\end{tikzcd}$$

Suppose for example that the problem contains (2) $f(x) = t_5$ and (1) $f(y) = t_3$ and we add the equality (11) $x = y$. First we have to add an edge between x and y, labelled by the reason of this merge, *i.e.*, assumption (11). Then we have to add an edge between t_3 and t_5, and label it with the two assumptions that triggered that merge by congruence, *i.e.*, (1) and (2).

To output a certificate that two variables are equal, we travel the path between the two corresponding nodes, and each edge yields a list of commands. An edge labelled by an equality corresponds to a transitivity: $t_6 \xrightarrow{(18)} z$ yields $[\texttt{Trans}(18, true)]$. An edge labelled by two equalities uses congruence: $t_3 \xrightarrow{(1)(2)} t_5$ yields $[\texttt{Trans}(1, false); \texttt{Congr}(11, 1, true); \texttt{Trans}(2, true)]$. If the equality that triggered the congruence was *discovered* by UF and not an assumption, we have to explain it first, then update the environment accordingly using the `Push` command, and finally use the stored equality with the `Congr` command.

4 Certificate checking and generation for LRA and LIA

In this section we introduce the certificate language and proof system for linear arithmetic and describe its certifying prover. Literals are of the form $e \bowtie 0$ with e a linear expression manipulated in (Horner) normal form and $\bowtie \in \{\geq, >, =\}$.

Certificate language. Since our initial work [5], we are maintaining and enhancing reflexive tactics for real arithmetic (`psatz`) and linear integer arithmetic (`lia`). Those tactics, which are now part of the Coq code-base, are based on the *Positivstellensatz*[28], a rich proof system which is complete for non-linear (real) polynomial arithmetic. Those reflexive verifiers are at the core of our current theory verifiers for linear real arithmetic (LRA) and linear integer arithmetic (LIA). We present here simplified proof systems specialised for linear arithmetic.

For linear real arithmetic Farkas' lemma provides a sound and complete notion of certificate for proving that a conjunction of linear constraints is unsatisfiable [27, Corollary 7.1e]. It consists in exhibiting a positive linear combination of the hypotheses that is obviously unsatisfiable, *i.e.*, deriving $c \bowtie 0$ for $\bowtie \in \{>, \geq, =\}$ and c a constant such that $c \bowtie 0$ does not hold. To construct such a contradiction, we start with a sub-proof system that allows to derive an inequality with a list of commands (a Farkas certificate). Each command is a pair $\texttt{Mul}(c, i)$ where c is a coefficient (in type $\mathbb{Z}$) and i the index of an assumption in the current assumption set. Such a command is used below in a judgement $\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma', e' \bowtie' 0$ with $\bowtie$ and $\bowtie'$ in $\{\geq, >\}$. $\Gamma \cup \{e \bowtie 0\}$ is the current set of assumptions, $e' \bowtie' 0$ is a new deduced inequality and Γ' is an enriched set of assumptions. For LIA, the proof system is augmented with a `Cut` command to generate *cutting planes* [27, chapter 23] and a rule for case-splitting `Enum`. We also need a `Push` and a `Get` command in order to update the environment and retrieve an already derived formula. The semantics of the commands is given in Figure 2. The operators $[*], [+], [-]$ model the standard arithmetic operations but maintain the normalised form of the linear expressions. The rules for the `Mul` command follow the standard sign rules in arithmetic: for example, if e' is positive we can add it c times to the right part of the inequality $e \bowtie 0$, assuming c is strictly positive. To implement the `Cut` rule, the constant g is obtained by

$$\frac{c > 0 \quad \Gamma(i) \mapsto e' \geq 0}{\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma, (c[*]e'[+]e) \bowtie 0} \qquad \frac{\Gamma(i) \mapsto e' = 0}{\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma, (c[*]e'[+]e) \bowtie 0}$$

$$\frac{c > 0 \quad \Gamma(i) \mapsto e' > 0}{\Gamma, e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} \Gamma, (c[*]e'[+]e) > 0} \qquad \frac{\Gamma(i) = e' \bowtie 0}{\Gamma, e \bowtie' 0 \xrightarrow{Get(i)} \Gamma, e' \bowtie 0}$$

$$\frac{\Gamma' = \Gamma[i \mapsto e \bowtie 0]}{\Gamma, e \bowtie 0 \xrightarrow{Push(i)} \Gamma', e \bowtie 0} \qquad \frac{g > 0}{\Gamma, (g[*]e[-]d) \geq 0 \xrightarrow{\texttt{Cut}} \Gamma, (e[-]\lceil d/g \rceil) \geq 0}$$

$$\frac{g \mid d}{\Gamma, (g[*]e[-]d) = 0 \xrightarrow{\texttt{Cut}} \Gamma, (e[-](d/g)) = 0} \qquad \frac{\neg(g \mid d)}{\Gamma, (g[*]e[-]d) = 0 \xrightarrow{\texttt{Cut}} \Gamma, 0 > 0}$$

$$\frac{\begin{array}{c}\Gamma(i_1) \mapsto e[-]l \geq 0 \quad \Gamma(i_2) \mapsto h[-]e \geq 0 \\ \forall v \in [l, h], \Gamma, e = v \xrightarrow{c_{v-l}}_* \Gamma'_v, e' \bowtie' 0\end{array}}{\Gamma, \cdot \bowtie 0 \xrightarrow{\texttt{Enum}(i_1, i_2, [c_0; \ldots; c_{h-l}])} \Gamma, e' \bowtie' 0}$$

Fig. 2: LRA and LIA proof rules

computing the greatest common divisor of the coefficient of the linear expression. For inequalities, the rule allows to *cut* the constant. For equalities, it allows to detect a contradiction if g does not divide d ($\neg(g \mid d)$).

A LRA certificate is then either a proof of $0 > 0$ given by a list of commands or a proof of $x = y$ given by two lists of commands (one for $x - y \geq 0$ and one other for $y - x \geq 0$.

```
Inductive LRA_certificate :=
|LRA_False (l : list command) |LRA_Eq (l1 l2 : list command)
```

$$\frac{\Gamma \vdash l : 0 > 0}{\Gamma \vdash_{LRA} (\texttt{LRA\_False}(l)) : (\Gamma, nil)} \qquad \frac{\Gamma \vdash l_1 : e \geq 0 \quad e = x[-]y \quad \Gamma \vdash l_2 : [-]e \geq 0}{\Gamma \vdash_{LRA} (\texttt{LRA\_Eq}(l_1, l_2)) : (\Gamma, [x = y])}$$

Because the theory LIA is non-convex, it is necessary to deduce contradictions but also disjunction of equalities.

```
Inductive LIA_certificate :=
| LIA_False (l : list command)
| LIA_Eq (eqs : list (var * var)) (l : list (list command))
```

Proving equalities is done by performing a case-split and each list of commands $l \in l$ is used to prove that a case is unsatisfiable.

Certificate generation. In order to produce Farkas certificates efficiently, we have implemented the Simplex algorithm used in Simplify [14]. This variant of the standard linear programming algorithm does not require all the variable to be non-negative, and directly handles (strict and large) inequalities and equalities. Each time a contradiction is found, one line of the Simplex tableau gives us the expected Farkas coefficients. The algorithm is also able to discover new equalities between variables. In this case again, the two expected Farkas certificates are read from the current tableau, up to trivial manipulations.

For LIA, we use a variant of the Omega test [26]. The Omega test lacks a way to derive equalities but the number of shared variables is sufficiently small to allow an exhaustive search. Moreover, an effective heuristics is to pick as potential equalities the dis-equalities present in the unsat core.

5 Design of a modular Nelson-Oppen proof-verifier

This section presents the design of a reflexive Coq verifier for a Nelson-Oppen style combination of theories. Section 5 presents the main features of the theory interface. Section 5 explains the data-structures manipulated by the Nelson-Oppen proof-checker, *i.e.*, its dependently typed environment and its certificates.

Theory interface. A theory `T` defines a type for sorts `sort`, terms `term` and formulae `form`. Sorts, terms and formulae are equipped with interpretation functions `isort`, `iterm` and `iform`. The function `isort:sort→Type` maps a sort to a Coq type. Terms and formulae are interpreted with respect to a typed environment `env∈Env` defined by `Env:=var→∀(s:sort),isort s`. Each theory uses an environment Γ `∈Gamma` to store formulae. Environments expose the following API:

```
Record GammaAPI : Type := {|
 empty : Gamma ; add : form  →  Gamma  →  Gamma;
 ienv : Env  →  Gamma  →  Prop;
 ienv_empty : ∀ env, ienv env empty;
 ienv_add : ∀ (f : form) (s : Gamma) (env : Env), ienv env s  →
                        iform env f  →  ienv env (add f s) |}.
```

Environments are equipped with an interpretation function `ienv`. The `empty` environment represents an empty conjunction of formulae, *i.e.*, the assertion *true* and is such that `ienv env empty` holds for any environment. The operation `add` models the addition of a formula and is compatible with the interpretation `iform` of formulae. Our instantiations exploit the fact that environments are kept abstract: for UF, environments are radix trees allowing a fast look-up of formulae; for LRA, they are simple lists but arithmetic expressions are normalised (put in Horner normal form) by the `add` operation.

The key feature provided by a theory T is a proof-checker `Checker`. It takes as argument an environment Γ and a certificate *cert*. Upon success, the checker returns an updated environment Γ' and a list $eqs = (x_1 =_{s_1} x'_1, \ldots, x_n =_{s_n} x'_n)$ of equalities between sorted variables. In such cases, `Checker_sound` establishes that $\Gamma \vdash_T cert : (\Gamma', eqs)$ is a judgement of the Nelson-Oppen proof system (see Section 2). A representative theory record is given below.

```
Record Thy := {|
 sort : Type; term : Type;  form : Type;
 sort_of_term : term  →  sort;  isort : sort  →  Type;
 Env := var  →  ∀ (s:sort), isort s;
 iterm : Env  →  ∀ (t : term), isort (sort_of_term t);
 iform : Env  →  form  →  Prop
 ...
 Checker : Gamma  →  Cert  →  option(Gamma * (list (Eq.t sort)));
 Checker_sound : ∀ cert Γ Γ' eqs, Checker Γ cert = Some(Γ', eqs)
   →  ∀ (env : Env),  ienv env Γ  →  ienv env Γ'
           /\∃s, ∃x, ∃y, (x =s y) ∈ eqs /\ env x s = env y s |}.
```

Nelson-Oppen proof-checker. Given a list of theories $T_1,\ldots,T_n$ the environment of the Nelson-Oppen proof-checker is a dependently typed list such that the ith element of the list is an environment of type T_i.(Gamma). Dependently typed lists are defined as follows:

```
Inductive dlist (A : Type) (typ : A → Type) : list A  →  Type :=
| dnil : dlist A typ nil
| dcons : ∀ (x : A) (e : typ x) (lx : list A) (le : dlist lx),
                  dlist A typ (x::lx).
```

A term `dcons x e lx le` constructs a list with head `e` and tail `le`. The type of `e` is `typ x` and the type of the elements of `le` is given by `(List.map typ lx)`.

It follows that the environment of the Nelson-Oppen proof-checker has type:

$$\texttt{dlist Thy Gamma } (T_1::\ldots::T_n)$$

A single proof-step consists in checking a certificate `JCert` of the joint theory defined by `JCert :=` T_1`.(Cert) + ... +` T_n`.(Cert)`.

Each certificate triggers the relevant theory proof-checker and derives an eventually empty list of equalities, *i.e.*, a proof of non-satisfiability. Each equality $x =_s y$ is cloned for each sort `s'` such that `isort s = isort s'` and propagated to the relevant theory. Each equality of the list is responsible for a case-split that may be recursively closed by a certificate (see Section 2). A certificate for the Nelson-Oppen proof-checker is therefore a tree of certificates defined by:

```
Inductive Cert := Mk (cert : JCert) (lcert : list Cert).
```

The Nelson-Oppen verifier consumes the certificate and returns `true` if the last deduced list of equalities is empty. In all other cases, the verification aborts and the verifier returns `false`.

6 Experiments

The purpose of our experiments is twofold. They first show that our SMT format is viable and can be generated for a substantial number of benchmarks. The experiments also assess the efficiency of our Coq reflexive verifier. We have evaluated our approach on quantifier-free first-order unsatisfiable formulae over the combinations of the theory of equality and uninterpreted functions (UF), linear real arithmetic (LRA), linear integer arithmetic (LIA) and real difference logic (RDL). All problems are unsatisfiable SMT-LIB 2 benchmarks selected from the SMT-LIB repository that are solved by Z3 in less than 30 seconds.

Table 1 shows our results sorted by logic. For each category, we measure the average running time of Z3 (Solved), the average running time of our certificate generation (Generation). The *Solved* time can be seen as a best-case scenario: the certifying prover uses Z3 and provide proofs that can be checked in Coq, so we do not expect faster results than the standalone state-of-the-art solver. We also measure the time it takes Coq to type-check our proof term (Qed) and have isolated the time spent by our Coq reflexive verifier validating theory lemmas (Thy). The generation phase (Generation) and the checking phases (Checking) have an individual timeout of 150 seconds. These timeouts account for most of the failures, the remaining errors come from shortcomings of the prototype.

Overall, the theory specific checkers account for less then 7% of checking time. However, this average masks big differences. For UFLRA, the checker spends less than 1% of its time in the theories, but for the integer arithmetic fragments it represents 11% of checking time for UFLIA and 32% for LIA. For UFLRA it can be explained by the simplicity of the problems : 80% of these formulae are unsatisfiable even if we only consider their boolean abstractions. For integer arithmetic the success ratio is rather low. It is hard to know whether this is due to the inherent difficulty of the problems or whether it pinpoints an inefficiency of the checker. The fault might also lie on the certifying prover side. In certain circumstances, it performs case-splits that are responsible for long proofs.

Sometimes, our simple SMT loop fails to produce certificates before timeout. For UF and RDL we only generate certificates for a third of the formulae. The generation of certificates could be optimised further. A more clever proof search strategy could improve both certificate generation and checking times: smaller certificates could be generated faster and checked more easily. Yet, the bottleneck is the reflexive verifier, which achieves 100% success ratio for UF only. Currently, we observe that our main limiting factor is not time but the memory consumption of the Coq process. A substantial amount of our timeouts are actually due to memory exhaustion. We are investigating the issue, but the objects we manipulate (formulae, certificates) are orders of magnitude larger than

Logic	Solved (Z3)		Generation		Checking		
	#	Time (s)	Success	Time (s)	Success	Thy (s)	Qed (s)
UF	613	0.96	31.3%	42.55	100%	0.29	16.81
LRA	248	0.65	79.4%	6.79	69.5%	0.28	4.02
UFLRA	407	0.11	100%	0.72	98.8%	0.02	3.56
LIA	401	1.86	74.3%	9.05	46.0%	2.26	7.02
UFLIA	159	0.05	97.5%	8.15	96.1%	0.33	2.91
RDL	79	4.01	38.0%	11.24	53.3%	0.14	3.64
Total	1907	0.87	67.1%	11.02	80.8%	0.45	6.45

Table 1: Experimental results for selected SMT-LIB logics

those manipulated on a day-to-day basis by a proof-assistant. We know we are reaching the limits of the system. Actually, to perform our experiments we already overcome certain inefficiencies of Coq. For instance, to construct formulae and certificates we by-pass Coq front-end, which is not efficient enough for this application, and use homemade optimised versions of a few Coq tactics.

7 Related Work

The area of proof-generating decision procedure has been pioneered by Boulton for the HOL system [7] and Necula for Proof Carrying Code [21]. In the context of the latter, the Touchstone theorem prover [22] generates LF proof terms. In our approach, each decision procedure comes with its own certificate language, and a reflexive checker. It allows us to choose the level of details of the certificates without compromising correctness. Several authors have examined UF proofs [12, 24]. They extend a pre-existing decision procedure with proof-producing mechanism without degrading its complexity and achieving a certain level of irredundancy. However, their notion of proof is reduced to unsatisfiable cores of literals rather than proof trees. Our certificate generation builds on such works to produce detailed explanations. SMT solvers such as CVC3 [4], veriT [8] and Z3 [10] all generate proofs in their own proof language. Many rules reflect the internal reasoning with various levels of precision: certain rules detail each computation step, some others account for complex reasoning with no further details. Such solvers aim at discharging large and/or hard problems, at the price of simplicity. Our approach here differs because our proof rules are specific to the decision procedure we have implemented in our prover. We do not sacrifice soundness since our proof verifier is proved correct (and executable) in Coq.

Several approaches have been proposed to integrate new decision procedures in sceptical proof assistants for various theories. First-order provers have been integrated in Isabelle [25], HOL [18] or Coq [9]. These works rely generally on resolution proof trees. Similar proof formats have been considered to integrate Boolean satisfiability checking in a proof assistant. Armand et al. [2] have extended the Coq programming language with machine integers and persistent array and have used these new features to directly program in Coq a reflexive SAT checker. On a similar topic, Weber and Amjad [29] have integrated a state-of-the-art SAT solver in Isabelle/HOL, HOL4 and HOL Light using translation from SAT resolution proofs to LCF-style proof objects.

Previous work has been devoted to reconstruct SMT solvers proofs in proof assistants. McLaughlin et al. [20] have combined CVC Lite and HOL light for quantifier-free first-order logic with equality, arrays and linear real arithmetic. Ge and Barrett have continued that work with CVC3 and have extended it to quantified formulae and linear integer arithmetic. This approach highlighted the difficulty for proof reconstruction. Independently Fontaine et al. [15] have combined haRVey with Isabelle/HOL for quantifier free first-order formulae with equality and uninterpreted functions. In their scheme, Isabelle solves UF sub-proofs with hints provided by haRVey. Our UF certificate language is more detailed and does not require any decision on the checker side. Böhme and Weber [6] are reconstructing Z3 proofs in the theorem provers Isabelle/HOL and HOL4. Their implementation is particularly efficient but their fine profiling shows that a lot of time is spent re-proving sub-goals for which the Z3 proof does not give sufficient details. Armand et al. [2] have recently extended their previous work [2] to check proofs generated by the SMT solver veriT [8]. Our approaches are similar and rely on proof by reflexion. A difference lies in the certificate generation scheme. Their implementation is tied to a specific SMT solver and its ability to generate proofs. In our approach, we do not require SMT solvers to generate proofs but instead designed our own proof-producing prover to discharge theory lemmas.

8 Conclusion and Perspectives

We have presented a reflexive approach for integrating a SMT solver in a sceptical proof assistant like Coq. It is based on a SMT proof format that is independent from a specific SMT solver. We believe our approach is robust to changes in the SMT solvers but allows nonetheless to benefit from their improvements. For most usages, the overhead incurred by our SMT loop is acceptable. It could even be reduced if SMT solvers gave access to the theory lemmas they use during their proof search. We are confident that such information could be generated by any SMT solver with little overhead. Implementing our approach necessitates proof-producing decision procedures. However, the hard job is left to the SMT solver that extracts unsat cores. A fine-grained control over the produced proof has the advantage of allowing to optimise a reflexive verifier and of ensuring the completeness of the verifier with respect to the prover. Our Nelson-Oppen Coq verifier is both reflexive and parametrised by a list of theories. This design is modular and easy to extend with new theories. Our prototype implementation is perfectible but already validates SMT formulae of industrial size. Such extreme experiments test the limits of the proof-assistant and will eventually help at improving its scalability. In the future, we plan to integrate new theories such as the theory of arrays and bit-vectors. Another theory of interest is the theory of constructors that would be useful to reason about inductive types.

References

1. M. Armand, G. Faure, B. Gregoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP 2011*, LNCS. Springer, 2011.
2. M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In *ITP'10*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010.
3. C. Barret, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0, 2010.
4. C. Barrett and C. Tinelli. CVC3. In *CAV'07*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
5. F. Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *TYPES'06*, volume 4502 of *LNCS*, pages 48–62. Springer, 2007.

6. S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *ITP'10*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.
7. R. J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, 1994. Technical Report 337.
8. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *CADE'09*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.
9. E. Contejean and P. Corbineau. Reflecting proofs in first-order logic with equality. In *CADE'05*, volume 3632 of *LNCS*, pages 7–22. Springer, 2005.
10. L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *LPAR'08 Workshops: KEAPPA*, volume 418. CEUR-WS.org, 2008.
11. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
12. L. M. de Moura, H. Rueß, and N. Shankar. Justifying equality. *ENTCS*, 125(3):69–85, 2005.
13. L. M. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE'02*, volume 2392 of *LNCS*, pages 438–455. Springer, 2002.
14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
15. P. Fontaine, J-Y. Marion, S. Merz, L. P. Nieto, and A. F. Tiu. Expressiveness+automation+soundness: Towards combining SMT solvers and interactive proof assistants. In *TACAS'06*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.
16. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *ICFP'02*, pages 235–246. ACM, 2002.
17. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOLs 2005*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
18. J. Hurd. Integrating Gandalf and HOL. In *TPHOLs 1999*, volume 1690 of *LNCS*, pages 311–322. Springer, 1999.
19. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54. ACM, 2006.
20. S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. *ENTCS*, 144(2):43–51, 2006.
21. G. C. Necula. *Compiling with Proofs*. PhD thesis, CMU, 1998.
22. G. C. Necula and P. Lee. Proof generation in the Touchstone theorem prover. In *CADE'00*, volume 1831 of *LNCS*, pages 25–44. Springer, 2000.
23. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1:245–257, 1979.
24. R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *RTA'05*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.
25. L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In *TPHOLs'07*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.
26. William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, pages 4–13, 1991.
27. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.
28. G. Stengle. A nullstellensatz and a positivstellensatz in semialgebraic geometry. *Mathematische Annalen*, 207(2):87–97, 1973.
29. T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.