Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme

Stéphane Lescuyer[1,2] Sylvain Conchon[2,1]

[1] INRIA Saclay-Île de France, ProVal, Orsay F-91893
[2] LRI, Université Paris-Sud, CNRS, Orsay F-91405

Abstract. In an attempt to improve automation capabilities in the Coq proof assistant, we develop a tactic for the propositional fragment based on the DPLL procedure. Although formulas naturally arising in interactive proofs do not require a state-of-the-art SAT solver, the conversion to clausal form required by DPLL strongly damages the performance of the procedure. In this paper, we present a reflexive DPLL algorithm formalized in Coq which outperforms the existing tactics. It is tightly coupled with a lazy CNF conversion scheme which, unlike Tseitin-style approaches, does not disrupt the procedure. This conversion relies on a lazy mechanism which requires slight adaptations of the original DPLL. As far as we know, this is the first formal proof of this mechanism and its Coq implementation raises interesting challenges.

1 Introduction

Interactive provers like the Coq Proof Assistant [8] offer a rich and expressive language that allows one to formalize complex objects and mathematical properties. Unfortunately, using such provers can be really laborious since users are often required to delve into vast amounts of proof details that would just be ignored in a pencil-and-paper proof. Specific decision procedures have been implemented in Coq in an attempt to improve its automation capabilities and assist users in their task. For instance, tactics like `omega`, `tauto` and `congruence` respectively address linear arithmetic, propositional logic and congruence closure, but they still lack co-operation with each other.

Our long-term goal is to design a tactic integrating techniques from the Satisfiability Modulo Theories (SMT) community in Coq in order to automatize the combination of different decision procedures. This would be a drastic improvement over the current situation where combination has to be manually driven by the user. As a first step in that direction, we design a tactic based on a SAT solver, the procedure at the heart of most SMT solvers. This procedure can be used to decide the validity of propositional formulas in Conjunctive Normal Form (CNF). In order for our tactic to be able to deal with the full propositional fragment of Coq's logic and be useful in practice, we must perform a conversion into CNF before applying the procedure. This conversion step can be critical for the efficiency of the whole system since it can transform a rather easy problem into one that is much too hard for our decision procedure. A possible solution

is to rely on a lazy conversion mechanism such as Simplify's [15]. Because this mechanism must be tightly coupled to the decision procedure, this rules out the use of an external tool and leads us to an approach of proof by reflection.

In this paper, we present a reflexive tactic for deciding validity in the propositional fragment of Coq's logic. We show how to adapt a fully certified standard DPLL procedure in order to take a lazy conversion scheme into account. In Sect. 2, we start by some preliminary considerations about reflection and CNF conversion techniques. We describe our abstraction of the lazy CNF conversion method in Sect. 3 as well as the necessary modifications to the DPLL procedure. Section 4 then presents how the lazy CNF conversion can be efficiently implemented in Coq. Finally, we compare our tactic with other methods in Sect. 5 and argue about its advantages and how they could be useful in other settings.

The whole Coq development is browsable online at `http://www.lri.fr/~lescuyer/unsat`. In the electronic version of this paper, statements and objects that have been formalized are marked with ✓, which are hyperlinks to the corresponding documented code.

2 Motivations

2.1 Integrating Decision Procedures in Proof Assistants

Interactive provers in general ensure their correctness by following the so-called LCF-style approach: every proof must be checked by a small, trusted part of the system. Thus, a complex decision procedure for an interactive prover shall not only decide if a formula is provable or not, but it must also generate an actual proof object, which can be checked by the prover's kernel. There have been several different approaches to the problem of integrating decision procedures in interactive provers.

A first possible approach is to use an external state-of-the-art decision procedure. This requires the external tool to be able to return *proof traces* of its proof search. Work must then be done in the interactive prover in order to reconstruct a suitable proof object from the output of the external tool. For instance, Weber and Amjad [28] have successfully integrated two leading SAT solvers, zChaff [23] and MiniSat [18], with Higher Order Logic theorem provers. Integrations of resolution-based provers have also been realized in Coq [1,2] and Isabelle [22]. This approach's main advantage is the ability to use a very efficient external tool.

Another approach is to implement one's own decision procedure in the sources of the interactive prover. It is actually the one being used for most of Coq's automation tactics, including `tauto` by Muñoz [16], `omega` [26] by Crégut and `congruence` by Corbineau [9]. This approach is not as optimized as a mature external tool, but can be specifically designed for the prover in order to have a more efficient proof construction.

The last approach is the so-called *proof by reflection* [3] and is summarized in Fig. 1. It consists in implementing the decision procedure directly as a function in the prover's logic, along with its correctness properties. If a formula Φ can be *reified* into an abstract representation f, proving Φ amounts to applying the

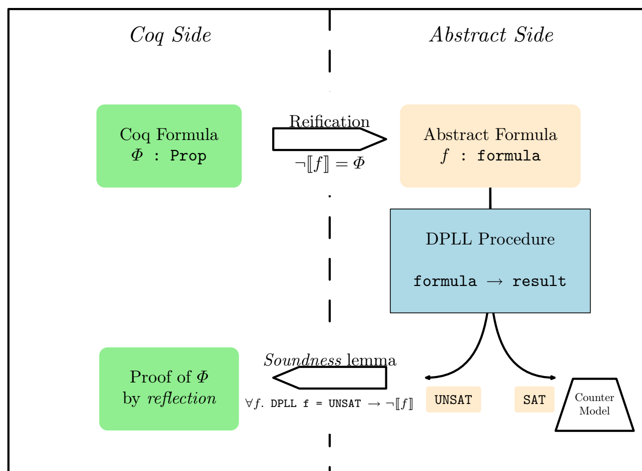

Fig. 1. An overview of our reflexive tactic

soundness theorem and executing the procedure on f. For instance, the tactics `ring` [20] and `field` [14], which respectively solve expressions on ring and field structures, are built along this reflection mechanism. The main advantage of the reflexive approach is the size of the generated proof term, which only consists in one application of the correctness property. The trade-off is that typechecking the proof term includes executing the decision procedure, therefore reflection can be used favourably in cases where the proof traces would not be comparatively simpler than the proof search itself.

2.2 A Mixed Approach Based on Reflection

Although traces-based approaches allow the use of state-of-the-art decision procedures, they raise a couple of practical issues nonetheless. The main difficulty consists in finding the adequate level of detail to describe the reasoning steps performed by the decision procedure. Fine-grained traces make for an easier proof reconstruction but require a substantial amount of work in the decision procedure, including justifying steps that are often implicit in an efficient implementation. On the other hand, coarse-grained traces make proof reconstruction much harder since all implicit steps must be implemented in the proof assistant, for instance using reflection. Looking for an intermediate approach, Corbineau and Contejean [6] and Contejean *et al.* [7] proposed integrations mixing traces and reflection.

Our project of integrating an SMT solver in Coq follows this mixed approach, giving a more prominent role to reflection. Indeed, we are especially interested in proving proof obligations from program verification, similar to AUFLIA and AUFLIRA divisions of the SMT competition. Our experience with our own prover

Alt-Ergo [5] is that these formulas' difficulty lies more in finding the pertinent hypotheses and lemmas' instances than in their propositional structure or the theory reasoning involved in their proofs. Consequently, these problems become rather easy as soon as we know which hypotheses and instances are sufficient for the proof. We thus propose to use the external prover as an oracle to reduce a formula to an easier ground problem, and solve the latter by reflection in the proof assistant. In this way, traces will be simple enough to be easily provided by any SMT solver. Our contribution in this paper is a reflexive, carefully designed, DPLL procedure, which is the core of our reflexive solver.

2.3 The CNF Conversion Issue

In order for a reflexive tactic based on a SAT solver to deal with the full propositional fragment of Coq's logic, it needs to be able to take any arbitrary formula in input and convert it into CNF, which is the only class of formulas that the DPLL procedure can handle. Looking at Fig. 1 once again, there are two possibilities as to where this CNF conversion can occur : on the Coq side or on the abstract side, *i.e.* before or after the formula is reified into an abstract Coq object. When conversion is performed on the Coq side✓, every manipulation of the formula is actually a logical rewriting step and ends up in the proof term. Moreover, it is very slow in practice because the matching mechanism used to rewrite the formula is not very efficient. Altogether, this CNF conversion can yield really big proof terms on average-sized formulas and it even ends up taking much longer than the proof search itself — we experimented it in earlier versions of our system [21]. Performing the CNF conversion on the abstract side, however, can be summarized in the following way:

– we implement a function `conversion : formula → formula` that transforms an abstract formula as wanted✓;
– we show that for all formula `F`, `conversion F` is in CNF✓ and is equivalent✓ (or at least equisatisfiable) to `F` itself.

This method ensures that the CNF conversion takes constant size in the final proof term, and can be performed efficiently since it is executed by Coq's virtual machine.

Once we decide to implement the CNF conversion as a function on abstract formulas, there are different well-known techniques that can be considered and that we implemented.

1. The first possibility is to do a naive, traditional, CNF conversion✓ that uses de Morgan laws in order to push negations through the formula to the atoms' level, and distributes disjunctions over conjunctions until the formula is in CNF. For instance, this method would transform the formula $A \vee (B \wedge C)$ in $(A \vee B) \wedge (A \vee C)$. It is well-known that the resulting formula can be exponentially bigger than the original.
2. Another technique✓ that avoids the exponential blow-up of the naive conversion is to use Tseitin's conversion [27]. It adds intermediate variables for subformulas and *definitional clauses* for these variables such that the size of

the resulting CNF formula is linear in the size of the input. On the $A \vee (B \wedge C)$ formula above, this method returns $(A \vee X) \wedge (\bar{X} \vee B) \wedge (\bar{X} \vee C) \wedge (X \vee \bar{B} \vee \bar{C})$ where X is a new variable.
3. A refinement of the previous technique$^{\checkmark}$ is to first convert the formula to negation normal form and use Plaisted and Greenbaum's CNF conversion [25] to add half as many definitional clauses for the Tseitin variables. In the above example, the resulting formula is $(A \vee X) \wedge (\bar{X} \vee B) \wedge (\bar{X} \vee C)$.

The Need for Another CNF Conversion. The CNF conversion techniques that we have considered so far remain unsatisfactory. The first one can cause an exponential increase in the size of the formula, and the other two add many new variables and clauses to the problem. All of them also fail to preserve the high-level logical structure of the input formula, in the sense that they make the problem more difficult than it was before. There has been lots of work on more advanced CNF conversion techniques but their implementation in Coq raises some issues. For instance, Plaisted and Greenbaum's method was originally intended to preserve the structure of formulas, but in order to do so requires that equal subformulas be shared. Other optimization techniques [24,12] are based on renaming parts of the subformula to increase the potential sharing. However, it is hard to implement such methods efficiently as a Coq function, ie. in a pure applicative setting with structural recursion. Even proving the standard Tseitin conversion proved to be more challenging than one would normally expect.

For the same reason, it is undeniable that a reflexive Coq decision procedure cannot reach the same level of sheer performance and tuning than state-of-the-art tools, which means that we cannot afford a CNF conversion that adds too many variables, disrupts the structure of the formula, in a word that makes a given problem look harder than it actually is. Results presented in Sect. 5 show that this concern is justified. Constraints due to CNF conversion also arise in Isabelle where formulas sent to the Metis prover are limited to 64 clauses. In the description of their Simplify theorem prover [15], Nelson *et al.* describe a lazy CNF conversion method they designed so as to prevent the performance loss due to Tseitin-style CNF conversion. Their experience was that "introducing lazy CNF into Simplify avoided such a host of performance problems that [..] it converted a prover that didn't work in one that did." In the remaining sections of this paper, we describe how we formalized and integrated this lazy CNF conversion mechanism in our DPLL procedure.

3 A DPLL Procedure with Lazy CNF Conversion

In this section, we formally describe how a DPLL procedure can be adapted to deal with literals that represent arbitrary formulas. We start by recalling a formalization of the basic DPLL procedure.

3.1 Basic Modular DPLL

The DPLL procedure [11,10], named after its inventors Davis, Putnam, Logemann and Loveland, is one of the oldest decision procedures for the problem

of checking the satisfiability of a propositional formula. DPLL deals with CNF formulas, ie. conjunctions of disjunctions of literals. A formula in CNF can thus be written $\bigwedge_{i=1}^{n}(l_{i,1} \vee \cdots \vee l_{i,k_i})$ where each $l_{i,j}$ is a propositional variable or its negation.

$$\textsc{Unit}\ \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, l} \qquad \textsc{Red}\ \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C} \qquad \textsc{Elim}\ \frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C}$$

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset} \qquad \textsc{Split}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}$$

Fig. 2. The standard DPLL procedure

We give a formalization of this basic DPLL procedure as a set of five inference rules, presented in Fig. 2. The current state of the algorithm is represented as a sequent $\Gamma \vdash \Delta$ where Γ is the set of literals that are assumed to be true, and Δ is the current formula, seen as a set of clauses, i.e. a set of sets of literals. We use the comma to denote conjunction, $\vee$ for disjunction and $\bar{l}$ for the negation of literal l. The Conflict rule corresponds to the case where a clause has been reduced to the empty clause: this rule terminates a branch of the proof search and forces the algorithm to backtrack in order to find another valuation. Unit implements *unit propagation*: if a clause is reduced to the literal l, this literal can be added to the context before proof search goes on. Elim and Red each perform one kind of boolean constraint propagation by simplifying the current set of clauses. The last rule is the rule that actually performs the branching, and thus the "proof search". Split picks any literal l and adds it to the context Γ. If no instantiation is found on this side (i.e. all the branches end with Conflict), then $\bar{l}$ is supposed true instead and the right branch is explored. If there exists a derivation for a CNF formula F starting with an empty context $\emptyset \vdash F$, this means that the whole tree has been explored and that the formula F is unsatisfiable.

In Coq, we can implement this formalization in a modular way using the module system [4]. The DPLL procedure can be implemented as a functor parameterized by a module for literals. Such a module for literals contains a type equipped with a negation function, comparisons, and various properties:

```
Module Type LITERAL.
  Parameter t : Set.
  Parameter mk_not : t → t.
  Axiom mk_not_invol : ∀l, mk_not (mk_not l) = l.
  ...
  (* Equality, comparisons, ... *)
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  ...
End LITERAL.
```

For more details about a Coq formalization of the basic DPLL procedure, the reader can refer to our previous work [21]. We will now proceed to extend this signature of literals and introduce our abstraction of the lazy CNF conversion.

3.2 Expandable Literals

In a Tseitin-style CNF conversion, new literals are added that represent subformulas of the original formula. To denote this fact, clauses must be added to the problem that link the new literals to the corresponding subformulas. The idea behind lazy CNF conversion is that new literals should not merely *represent* subformulas, but they should *be* the subformulas themselves. This way, there would be no need for additional definitional clauses. Detlefs et al. [15] present things a bit differently, using a separate set of *definitions* for new variables (which they call *proxies*), and make sure the definitions of a given proxy variable are only added to the current context when this variable is assigned a boolean value by the procedure. Our abstraction will require less changes to the DPLL procedure.

In order for literals to be able to stand for arbitrary complex subformulas, we extend the signature of literals given in Sect. 3.1 in the following way✓:

```
Module Type LITERAL.
  Parameter t : Set.
  Parameter expand : t → list (list t).
  (* Negation, Equality... as before *)
  ...
End LITERAL.
```

In other words, literals always come with negation, comparison, and various properties, but they have an additional *expansion* function, named `expand`, which takes a literal and returns a list of lists of literals, in other words a CNF of literals. For a genuine literal which just stands for itself, this list is simply the empty list []. For another literal that stands for a formula F, ie. a proxy F, this function allows one to unfold this literal and reveal the underlying structure of F. This underlying structure must be expressed as a conjunction (list) of disjunctions (lists) of literals, but since these literals can stand for subformulas of F themselves, this CNF does not have to be the full conjunctive normal form of F: it can undress the logical structure of F one layer at a time, using literals to represent the direct subformulas of F. This means that the CNF conversion of formula F can be performed step after step, in a *call-by-need* fashion. In [15], the `expand` function would be a look-up in the set of proxy definitions.

As an example, let us consider the formula $A \vee (B \wedge C)$ once again. A proxy literal for this formula could expand to its full CNF, namely the list of lists $[[A; C]; [B; C]]$. But more interestingly, it may also reveal only one layer at a time and expand to the simpler list $[[A; X]]$, where X itself expands to $[[B]; [C]]$. Note that this variable X is not a new variable in the sense of Tseitin conversion, it is just a way to denote the *unique* literal that expands to $[[B]; [C]]$, and which therefore stands for the formula $B \wedge C$. This unicity will be the key to the structural sharing provided by this method. In Sect. 4, we will describe how these

$$\textsc{Unit}\ \frac{\Gamma, l \vdash \Delta, \texttt{expand}(l)}{\Gamma \vdash \Delta, l} \qquad \textsc{Red}\ \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C} \qquad \textsc{Elim}\ \frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C}$$

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset} \qquad \textsc{Split}\ \frac{\Gamma, l \vdash \Delta, \texttt{expand}(l) \qquad \Gamma, \bar{l} \vdash \Delta, \texttt{expand}(\bar{l})}{\Gamma \vdash \Delta}$$

Fig. 3. The DPLL procedure adapted to expandable literals

expandable literals can be implemented in such a way that common operations are efficient, but for now let us see how the DPLL procedure should be adapted.

3.3 Adaptation of the DPLL Procedure

In order to use expandable literals in the DPLL procedure, we have to adapt the rules presented in Sect. 3.1. Let us consider a proxy literal f for a formula F. If this proxy is assigned a true value at some point during the proof search, this means that the formula F is assumed to be true. Therefore, something should be added to the current problem that reflects this fact in order to preserve the semantic soundness of the procedure. To this end, we use the `expand` function on f in order to unveil the structure of F, and add the resulting list of clauses $\texttt{expand}(f)$ to the current problem.

The revised version of our inference rules system is given in Fig. 3. The only modifications between this system and the one presented in Fig. 2 concern rules which change the current assignment Γ : Unit and Split. When a literal l is assumed in the current context, it is expanded and the resulting clauses are added to the current problem Δ. Intuitively, if l is a proxy for F, $\texttt{expand}(l)$ can be seen as "consequences" of F and must be added in order to reflect the fact that F shall now be satisfied. Now, given an arbitrary formula F, instead of explicitely converting it into a CNF Δ_F and searching a derivation for $\emptyset \vdash \Delta_F$, it is enough to build a proxy literal l_F for F and attempt to find a derivation for $\emptyset \vdash l_F$ instead. This allows us to use a DPLL decision procedure with the lazy conversion mechanism. Note that correctness does not require proxy literals to be added to the current assignment Γ; however, doing so has a dramatic effect on formulas that can benefit from sharing, *e.g.* $l \wedge \neg l$, where l stands for a big formula F. Such formulas are not as anecdotal as they seem, and we discuss this further in Sect. 5.2.

We have implemented this system✓ in Coq as a functor parameterized by a module for literals and have proved its soundness✓ and completeness✓. We have also implemented a correct and complete proof-search function✓ that tries to build a derivation for a given formula, returns `Unsat` if it succeeds and a counter-model satisfying the formula otherwise.

4 Implementing Lazy Literals in Coq

In this section, we show how to design a suitable literal module on which we can instantiate the procedure we described in Sect. 3.3.

4.1 Raw Expandable Literals✓

Expandable literals are either standard propositional atoms, or proxies for a more complex formula. Because a proxy shall be uniquely determined by its expansion (in other words, proxies that expand to the same formula stand for the same formula, and therefore should be equal), we choose to directly represent proxies as their expansion. This leads us to the following definition✓ as a Coq inductive type:

```
Inductive t : Type :=
| Proxy (pos neg : list (list t))
| L (idx : index) (b : bool)
| LT | LF.
```

Standard literals are represented by the `L` constructor which takes a propositional variable `idx` and its sign `b` as argument. The last two constructors `LT` and `LF` are just the true and false literals; they are handled specially for practical reasons and we will mostly ignore them in the following. More interestingly, the `Proxy` constructor expects two arguments: the first one represents the formula that the proxy literal stands for, while the other one corresponds to the expansion of its negation. We proceed this way in order to be able to compute the negation of a literal in constant time, whether it is a proxy or not. Thus, the second parameter of `Proxy` should just be seen as a memoization of the negation function. As a matter of fact, we can easily define✓ the negation function:

```
Definition mk_not (l : t) : t :=
  match l with
    | Proxy pos neg ⇒ Proxy neg pos
    | L idx b ⇒ L idx b̄
    | LT ⇒ LF | LF ⇒ LT
  end.
```

Negating a standard literal is just changing its sign, while negating a proxy amounts to swapping its arguments. This memoization of the negation of a proxy literal is really critical for the efficiency of the method because literals are negated many times over the course of the DPLL procedure. In Sect. 4.3, we show how these proxies are created in linear time.

The implementation of the expansion function✓ is straightforward and requires no further comment:

```
Definition expand (l : t) : list (list t) :=
  match l with
    | Proxy pos _ ⇒ pos
    | L _ _ | LT | LF ⇒ []
  end.
```

We are left with implementing the comparison functions✓ on these literals. For instance, the equality test goes like this :

```
Fixpoint eq_dec (x y : t) : bool :=
  match x, y with
    | LT, LT | LF, LF ⇒ true
    | L idx b, L idx' b' ⇒ index_eq idx idx' && beq b b'
    | Proxy xpos _, Proxy ypos _ ⇒ ll_eq_dec eq_dec xpos ypos
    | _, _ ⇒ false
  end.
```

In this definition `index_eq` and `beq` respectively test equality for indices and booleans, while `ll_eq_dec` recursively applies this equality test point-wise to lists of lists of literals. The part that is worth noticing is that we only compare proxies' first component and we skip the negated part. This of course ensures that the comparison of proxies is linear in the size of the formula they stand for; had we compared the second component as well, it would have been exponential in practice. The issue with such optimizations is that we have to convince Coq that they make sense, and the next section is devoted to that point.

4.2 Adding Invariants to Raw Literals✓

When implementing expandable literals in the previous section, we made a strong implicit assumption about a proxy `Proxy pos neg`, namely that `neg` was indeed containing the "negation" of `pos`. We need to give a formal meaning to this sentence and to ensure this invariant is verified by all literals. It is not only needed for semantical proofs about literals and the DPLL procedure, but for the correctness of the simplest operations on literals, starting with comparisons. Indeed, considering the equality function `eq_dec` presented above, it is basically useless unless we can prove the two following properties:

```
Property eq_dec_true : ∀(x y : t), eq_dec x y = true → x = y.
Property eq_dec_false : ∀(x y : t), eq_dec x y = false → x ≠ y.
```

Proving the first property for standard literals is straightforward, but as far as proxies are concerned, the fact that the equality test returns true only tells us that the first component of the proxies are equal: there is no guarantee whatsoever on the second component. Therefore, this property is not provable as is and we need to add some relation between the two components of a proxy. This relation also ought to be symmetric since the `mk_not` function swaps the first and second component, and should of course preserve the invariant as well.

We are going to link the two components of a proxy literal by ensuring that each one is the image of the other by an adequate function $\mathcal{N}$. Intuitively, this function $\mathcal{N}$ must negate a conjunction of disjunction of literals and return another conjunction of disjunction of literals ; it can be recursively defined in the following way

$$\mathcal{N}((\bigvee_{i=1}^{n} x_i) \wedge C) = \bigwedge_{i=1}^{n} \bigwedge_{D \in \mathcal{N}(C)} (\bar{x}_i \vee D)$$

where the x_i are literals and C is a CNF formula. Once this function is implemented✓, we can define an inductive predicate that specifies *well-formed* literals:✓

```
Inductive wf_lit : t → Prop :=
| wf_lit_lit : ∀idx b, wf_lit (L idx b)
```

```
| wf_lit_true : wf_lit LT
| wf_lit_false : wf_lit LF
| wf_lit_proxy : ∀pos neg, 𝒩 pos = neg → 𝒩 neg = pos →
    (∀l t, l ∈ pos → t ∈ l → wf_lit t) →
    (∀l t, l ∈ neg → t ∈ l → wf_lit t) → wf_lit (Proxy pos neg).
```

The first three constructors express that the true and false literals, as well as all atomic literals, are well-formed. The last one brings up requirements on proxy literals : not only should the two components be each other's image by $\mathcal{N}$ [3], but all literals appearing in these expansions should recursively be well-formed. In particular, if two proxies are well-formed, their second components are equal if and only if their first components are equal, which means that we can establish the needed properties about the comparison function.

Packing everything together. In Coq, one can use *dependent types* in order to define a type of objects that meet certain specifications. We use this feature in our Coq development in order to define a module of well-formed literals✓. In this module, the type of literals is the dependent type of raw literals packed with a proof that they are well-formed:✓

```
Definition t : Type := {l | wf_lit l}.
```

We then have to redefine the required operations on literals. In most cases, it is just a matter of "lifting" to well-formed literals the definition we made for raw literals by showing that the operation preserves well-formedness. For instance, the negation function is (re)defined this way:✓

```
Property wf_mk_not : ∀l, wf_lit l → wf_lit (mk_not l).
Proof. ..... Qed.
Definition mk_not (l:t):t := exist (mk_not π1(l)) (wf_mk_not π1(l) π2(l)).
```

where π_1 and π_2 respectively access to the raw literal and its well-formedness proof in a well-formed literal. We have presented a simplified version here and the real development contains more invariants that are required throughout various proofs about literals and their operations. Altogether, we obtain a module with the signature of literals as expected by the DPLL procedure, and where every operation is totally certified.

4.3 Converting Formulas to Lazy Literals✓

Once we have a module implementing lazy literals as described above, we are left with the task of constructing such literals out of an input formula.

First, note that we should not build arbitrary literals but only literals that are well-formed. Therefore we have to make sure that the proxies we build respect the invariants that we introduced in the last section. Assume we want to build a proxy for a formula $F = F_1 \vee F_2$ and we know how to build proxies l_1 and l_2 for the formulas F_1 and F_2. A suitable proxy for F is the one that expands positively to the list $[[l_1; l_2]]$, and to the list $[[\bar{l}_1]; [\bar{l}_2]]$ negatively. We can check that these two lists are indeed each other's image by $\mathcal{N}$. In practice, we define a function✓ constructing such a proxy and we prove that its result a well-formed:✓

[3] This constraints the form of possible proxies since $\mathcal{N}$ is not involutive in general.

Proxy	*pos*	*neg*
$X \equiv P$	$\{P\}$	$\{\bar{P}\}$
$X \equiv F \vee G$	$\{F \vee G\}$	$\{\bar{F}\}\{\bar{G}\}$
$X \equiv F \wedge G$	$\{F\}\{G\}$	$\{\bar{F} \vee \bar{G}\}$
$X \equiv (F \rightarrow G)$	$\{\bar{F} \vee G\}$	$\{F\}\{\bar{G}\}$
$X \equiv (F_1 \vee F_2 \vee \ldots \vee F_n)$	$\{F_1 \vee F_2 \vee \ldots \vee F_n\}$	$\{\bar{F}_1\}\{\bar{F}_2\}\ldots\{\bar{F}_n\}$
$X \equiv (F_1 \wedge F_2 \wedge \ldots \wedge F_n)$	$\{F_1\}\{F_2\}\ldots\{F_n\}$	$\{\bar{F}_1 \vee \bar{F}_2 \vee \ldots \vee \bar{F}_n\}$

Fig. 4. Proxy construction for each logical connective

```
Definition mk_or_aux f g := Proxy [[f;g]] [[mk_not f];[mk_not g]].
Property wf_mk_or : ∀(l l' : t), wf_lit (mk_or_aux l l').
Proof. ..... Qed.
Definition mk_or f g : t := exist (mk_or_aux f g) (wf_mk_or f g).
```

The last command uses `mk_or_aux` and `wf_mk_or` to define a function that creates a well-formed proxy literal for the disjunction of two well-formed literals. We create such smart constructors for each logical connective : the table in Fig. 4 sums up how proxies are constructed for the usual logical connectives. Creating a proxy for an arbitrary formula is then only a matter of recursively applying these smart constructors by following the structure of the formula. We have implemented✓ such a function named `mk_form` and proved✓ that for every formula `F`, `mk_form F` $\leftrightarrow$ `F`. This theorem is very important since it is the first step that must be done when applying the tactic : it allows us to replace the current formula by a proxy✓ before calling the DPLL procedure. Note that the converted formula is *equivalent* to the original because no new variables have been added, whereas with Tseitin-like methods, the converted formula is only *equisatisfiable*.

Constructing proxies for N-ary operators. Figure 4 also contains proxy definitions for n-ary versions of the $\wedge$ and $\vee$ operators. We have implemented an alternative version✓ of the `mk_form` function above which tries to add as few levels of proxies as possible. When constructing a proxy for a disjunction (resp. conjunction), it tries to regroup all the disjunctive (resp. conjunctive) top-level structure in one single proxy. In this setting, equivalences are interpreted either as conjunctions or as disjunctions[4] in order to minimize the number of proxies.

5 Results and Discussion

5.1 Benchmarks

The whole Coq development is available at `http://www.lri.fr/~lescuyer/unsat/unsat.tgz` and can be compiled with Coq v8.2. It represents about 10000 lines of proofs and definitions and provides two different proof search strategies and all six variants of CNF conversion✓ that were discussed in this paper. No

[4] The equivalence $F \leftrightarrow G$ is logically equivalent to the conjunction $(F \rightarrow G) \wedge (G \rightarrow F)$ and the disjunction $(F \wedge G) \vee (\bar{F} \wedge \bar{G})$.

result has been admitted and no axioms have been assumed, therefore proofs made with our tactics are closed under context.[5] Because our development is highly modular, the procedure can be instantiated to decide boolean formulas as well as propositional formulas.

	tauto	CNF_C	CNF_A	Tseitin	Tseitin2	Lazy	LazyN
hole3	–	0.72	0.06	0.24	0.21	0.06	**0.05**
hole4	–	3.1	0.23	3.5	6.8	0.32	**0.21**
hole5	–	10	2.7	80	–	1.9	**1.8**
deb5	83	–	0.04	0.15	0.10	0.09	**0.03**
deb10	–	–	0.10	0.68	0.43	0.66	**0.09**
deb20	–	–	**0.35**	4.5	2.5	7.5	**0.35**
equiv2	0.03	–	0.06	1.5	1.0	**0.02**	**0.02**
equiv5	61	–	–	–	–	0.44	**0.42**
franzen10	0.25	16	0.05	0.05	0.03	**0.02**	**0.02**
franzen50	–	–	0.40	1.4	0.80	**0.34**	0.35
schwicht20	0.48	–	0.12	0.43	0.23	**0.10**	**0.10**
schwicht50	8.8	–	0.60	4.3	2.2	**0.57**	0.7
partage	–	–	–	13	19	**0.04**	0.06
partage2	–	–	–	–	–	0.12	**0.11**

Fig. 5. Comparison of different tactics and CNF conversion methods. Timings are given in seconds and – denote time-outs (>120s).

We benchmarked our tactic and the different CNF conversion methods on valid and unsatisfiable formulas✓ described by Dyckhoff [17] ; for instance *holen* stands for the pigeon-hole formula with n holes. We used two extra special formulas in order to test sharing of subformulas : *partage* is the formula $hole3 \wedge \neg hole3$, while *partage*2 is *deb*3 where atoms have been replaced by pigeon-hole formulas with varying sizes. Results are summarized in Fig. 5, where CNF_C and CNF_A are naive translations respectively on the Coq side and on the abstract side, Tseitin and Tseitin2 are the two variants of Tseitin conversion described in Sect. 2.3, and LazyN is our lazy conversion with proxies for n-ary operators. These results show that our tactic outperforms `tauto` in every single case (see discussion below for differences between our tactic and `tauto`), solving in less than a second goals that were beyond reach with the existing tactic. About the different CNF conversions, it turns out that the Tseitin conversion is almost always worse than the naive CNF conversion because of the extra clauses and variables. The lazy tactics always perform at least as well as CNF_A and in many cases they perform much better, especially when some sharing is required.

5.2 Discussion and Limitations

Comparison with `tauto/intuition`. In Coq, the tactic `tauto` is actually a customized version of the tactic `intuition`. `intuition` relies on an intuitionistic

[5] We discuss the possible use of the excluded-middle in Sect. 5.2. In any case, users can use the new `Print Asssumptions` command in order to check if their proofs depend on any axioms or not.

decision procedure and, when it can't solve a goal completely, is able to take advantage of the search-tree built by the decision procedure in order to simplify the current goal in a set of (simpler) subgoals ; `tauto` simply calls intuition and fails if any subgoals are generated. Unlike `intuition`, our tactic is unable to return a simplified goal when it cannot solve it completely, and in that sense it can be considered as less powerful. However, `intuition`'s performance often becomes an issue in practice[6], therefore we are convinced that the two tactics can prove really complementary in practice, with `intuition` being used as a simplifier and our tactic as a solver.

Classical reasoning in an intuitionistic setting. The DPLL procedure is used to decide classical propositional logic whereas Coq's logic is intuitionistic. In our development, we took great care in not using the excluded-middle for our proofs so that Coq users who do not want to assume the excluded-middle in their development can still use our tactic. The reason we were able to do so lies in the observation that the formula $\forall A.\neg\neg(A \vee \neg A)$ is intuitionnistically provable: when the current goal is `False`, this lemma can be applied to add an arbitrary number of ground instances of the excluded-middle to the context. In other words, if a ground formula Φ is a classical tautology, $\neg\neg\Phi$ is an intuitionistic tautology[7]. Noticing that $\neg\neg\neg\Phi$ implies $\neg\Phi$ in intuitionistic logic, this means that if $\neg\Phi$ is classically valid, it is also a tautology in intuitionistic logic. Because the DPLL procedure proceeds by refuting the context Φ, ie. proving $\neg\Phi$, we can use it in intuitionsitic reasoning even if it relies on classical reasoning.

In practice, the use of classical reasoning in our development is mainly for the correctness of the Split rule and of the different CNF conversion rules (*e.g.* $F \rightarrow G \equiv \bar{F} \vee G$). This led us to proving many intermediate results and lemmas in double-negation style because they were depending on some classical reasoning steps, but the nice consequence is that our tactic produces intuitionistic refutation proofs and thus can really replace `tauto` when the context becomes inconsistent. Users of classical reasoning can use our tactic for classical validity by simply refuting the negation of the current goal.

Impact of sharing. The results presented above show that the number of proxies has less effect on the performance than the sharing they provide. Depending on the formula, it may not be the best idea to minimize the number of proxies as LazyN does, because this minimizes the number of subformulas that are shared. Once again, we can use our modular development to provide these different alternatives as options to the user. We wrote in Sect. 3.3 that adding proxies to the current assignment made it possible to reduce a whole subformula of a the problem in one single step, and this is why sharing is beneficial. We gave the obvious, rather crafted, example of $l \wedge \bar{l}$ where l is a big formula, but there is a

[6] As Coq users, we often let `tauto` run for 10 seconds to make sure that a goal is provable, but if `tauto` didn't succeed immediately, we then proceed to manually prove it or simplify it in easier subgoals.

[7] This is not true for first-order formulas, because the formula $\neg\neg(\forall A.A \vee \neg A)$, where the quantification lies below the double negation, is not intuitionnistically provable.

less obvious and much more frequent situation where it happens. Formalizations often involve predicate definitions $p(x_1, \ldots, x_n) = \Phi(x_1, \ldots, x_n)$ where Φ can be a big formula, p is then used as a shortcut for Φ throughout the proofs. Now, when calling a DPLL procedure, one has to decide whether occurrences of p should be considered as atoms or should be unfolded to Φ. There is no perfect strategy, since proofs sometimes depend on p being unfolded and sometimes do not, but always unfolding p in the latter case leads to performance losses. Proxies make the DPLL procedure completely oblivious to such intermediate definitions, and this is a great asset when dealing with proof obligations from program verification.

5.3 Application to Other Systems

The advantages of the CNF conversion that we have implemented go beyond the scope of our tactic. It generally allows subformulas to be structurally shared which can give a big performance boost to the procedure. Moreover, in standard programming languages, proxies can be compared in constant time by using *hash-consing* [19], which removes the main cost of using lazy literals.

Lazy literals also provide a solution to a problem that is specific to SMT solvers : definitional clauses due to Tseitin-style variables appearing in contexts where they are not relevant can not only cause the DPLL procedure to perform many useless splits, but they also add ground terms that can be used to generate instances of lemmas. De Moura and Bjorner report on this issue in [13], where they use a notion of *relevancy* in order to only consider definitional clauses at the right time. Lazy CNF conversion is a solution to this issue, and it is the method we currently use in our own prover Alt-Ergo [5].

Finally, one may wonder whether this method can be adapted to state-of-the-art decision procedures, including common optimizations like backjumping and conflict clause learning. Adapting such procedures can be done in the same way that we adapted the basic DPLL and is really straightforward ; an interesting question though is the potential impact that lazy CNF conversion could have on the dependency analysis behind these optimizations. We have not yet thoroughly studied this question but our experience with Alt-Ergo suggests that lazy CNF conversion remains a very good asset even with a more optimized DPLL.

6 Conclusion

We have presented a new Coq tactic for solving propositional formulas which outperforms the existing `tauto`. It is based on a reflexive DPLL procedure that we have entirely formalized in the Coq proof assistant. We have used a lazy conversion scheme in order to bring arbitary formulas into clausal form without deteriorating the performance of the procedure. The results are very encouraging and we are planning on extending this tactic with decision procedures for specific theories in the same spirit as SMT solvers.

References

1. M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *JAR*, 29(3):253–275, 2002.

2. R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In *LPAR*, pages 151–165, 2007.
3. S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997.
4. J. Chrząszcz. Implementation of modules in the Coq system. In *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.
5. S. Conchon and E. Contejean. The Alt-Ergo Prover. `http://alt-ergo.lri.fr/`.
6. E. Contejean and P. Corbineau. Reflecting Proofs in First-Order Logic with Equality. In R. Nieuwenhuis, editor, *CADE-20*, volume 3632 of *LNAI*. Springer, 2005.
7. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *FroCos 07*, volume 4720 of *LNAI*. Springer.
8. The Coq Proof Assistant. `http://coq.inria.fr/`.
9. P. Corbineau. Deciding equality in the constructor theory. In T. Altenkirch and C. McBride, editors, *TYPES*, volume 4502 of *LNCS*, pages 78–92. Springer, 2006.
10. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5(7):394–397, 1962.
11. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
12. T. B. de la Tour. Minimizing the number of clauses by renaming. In *CADE-10*, pages 558–572. Springer-Verlag, 1990.
13. L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, LNCS, pages 183–198, 2007.
14. D. Delahaye and M. Mayero. `Field`: une procédure de décision pour les nombres réels en `Coq`. In *JFLA, Pontarlier (France)*. INRIA, Janvier 2001.
15. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
16. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992.
17. R. Dyckhoff. Some benchmark formulae for intuitionistic propositional logic, 1997.
18. N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. 2004.
19. J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In A. Kennedy and F. Pottier, editors, *ML*, pages 12–19. ACM, 2006.
20. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOLs*, pages 98–113, 2005.
21. S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLS'08 Emerging Trends*, 2008.
22. J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: first prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.
23. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC'01*, pages 530–535. ACM Press, 2001.
24. A. Nonnengart, G. Rock, and C. Weidenbach. On generating small clause normal forms. In *CADE-15*, pages 397–411, London, UK, 1998. Springer-Verlag.
25. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.
26. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992.
27. G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.
28. T. Weber, , and H. Amjad. Efficiently Checking Propositional Refutations in HOL Theorem Provers. In *Journal of Applied Logic*, volume 7, pages 26–40, 2009.