A Reflexive Formalization of a SAT Solver in Coq

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

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

Abstract. We present a Coq formalization of an algorithm deciding the satisfiability of propositional formulas (SAT). This *SAT solver* is described as a set of inference rules in a manner that is independent of the actual representation of propositional variables and formulas. We prove soundness and completeness for this system, and instantiate our solver directly on the propositional fragment of Coq's logic in order to obtain a fully reflexive tactic. Such a tactic represents a first and important step towards our ultimate goal of embedding an automated theorem prover inside the Coq system. We also extract a certified OCaml implementation of the algorithm.

1 Introduction

The fact that safety-critical software keeps getting bigger and more numerous has certainly contributed to a lot of effort being spent on certification systems in the recent years. However, improvements of automation techniques have only partially benefited to interactive provers in general, and to the Coq proof assistant [4] in particular. Although specific decision procedures have been and are still being implemented in Coq, they still lack co-operation with each other. For instance, tactics like `omega`, `tauto` and `congruence` respectively address linear arithmetic, propositional logic and congruence closure, but their combination still has to be driven manually by users, whereas tools like SMT solvers can perform this combination automatically.

We are currently developing an SMT solver dedicated to program verification [3] and our goal is to embed it as a decision procedure in Coq. Not only would it validate the algorithms and schemes at work in our solver, but it would also provide the automatic interaction of dedicated decision procedures in Coq. In this paper, we present our first step towards this goal: the formalization of a SAT solver, the decision procedure that lies at the heart of our prover. We also use *reflection* to obtain a tactic that relies on this certified decision procedure.

In Section 2, we quickly recall what the DPLL procedure is and we formalize it by a set of inference rules. In Section 3, we describe a Coq formalization of this procedure and we prove its soundness and completeness. We then use this procedure in Section 4 in order to build a reflexive tactic solving propositional goals. We show examples of applications of this tactic and its limitations in Section 5, before concluding with a discussion of related work in Section 6.

2 The DPLL Procedure

The DPLL procedure [7, 6], named after its inventors Davis, Putnam, Logemann and Loveland, is one of the oldest decision procedure for the problem of checking the satisfiability of a propositional formula. DPLL deals with formulas in *conjunctive normal form* (CNF), i.e. conjunction of clauses, where a clause is a disjunction of propositional literals. A formula in CNF can thus be written $\bigwedge_{i=1}^{n}(l_1 \vee \cdots \vee l_{k_i})$ where each l_j is a propositional variable or its negation.

DPLL essentially tries all possible valuations of the variables of a formula until it finds one that satisfies the formula. Since there is an exponential number of such valuations, DPLL enhances over the basic exploration of all valuations by the eager use of the following two simplifications:

- *Boolean constraint propagation:* once a truth value has been chosen for a given variable, the literals that become false can be removed from their clauses, and if a clause contains a literal that becomes true, the whole clause can be removed from the formula since it is now known to be true;
- *Unit propagation:* whenever a clause is reduced to a single literal, the valuation of this literal's variable must be set so that the literal is true, otherwise the whole formula would be false; an efficient detection of such clauses can dramatically change the overall performance of industrial-strength SAT solvers [17].

When every possible simplification has been applied, the algorithm tries to assign a boolean value to a variable of its choice, which leads to more simplification. Eventually, the procedure reaches one of the following cases:

- the problem is empty, i.e. the formula has been reduced to the empty conjunction, in which case a valuation satisfying the original formula has been found and the algorithm terminates;
- the formula contains an empty clause, which means it is obviously unsatisfiable; in that case, DPLL backtracks to an earlier point, where it tries assigning another value to a variable.

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

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

Fig. 1. The DPLL procedure seen as a proof derivation system

We give a formalization of this DPLL procedure as a set of five inference rules, presented in Fig. 1. 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 set of clauses, i.e. a set of sets of literals.

More precisely, we write $l \vee C$ for a clause that contains the literal l, and l_1, l_2, l_3 for the set of literals $\{l_1, l_2, l_3\}$. We keep the braces around unit clauses so as to avoid confusion, e.g. $\{l\}$ for the clause containing only l. On the right-hand side of the sequent, we denote by Δ, C the conjunction of a clause C and a CNF formula Δ. Finally, the negation of a literal l is denoted $\bar{l}$, with $\bar{\bar{l}} = l$.

We now take a closer look at the rules of Fig. 1, which should be read bottom-up. 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: if the negation of a literal is in Γ, it can be removed from all clauses (Red); if a clause contains a literal that is supposed true, the whole clause can be removed (Elim). 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 formula F starting with an empty context $\emptyset \vdash F$, this means that the whole tree has been explored and no model has been found, and therefore that the formula F is unsatisfiable.

3 Formalizing DPLL in Coq

In this section, we present a Coq formalization of the system presented in Section 2, for which we prove soundness and completeness with respect to a notion of semantics for formulas.

3.1 Preliminary Definitions

We start by defining how literals and formulas shall be represented. To do so, we will make use of Coq's module system [5, 2]. Coq *module types* allow one to pack together types, functions and related axioms by keeping a high level of abstraction. One can then create *functors*, i.e. modules which are parameterized by other modules of a certain signature and which can then be *instantiated* on any modules that match the expected signature.

Therefore, by taking advantage of Coq's module system, it is sufficient to define module types for literals and formulas, and we can then develop our decision procedure in a way that is independent of the actual representation of the input. For instance, the module type `LITERAL` (Fig. 2) provides a type `t` for literals, a function `mk_not` that builds the negation of a literal and some axioms about this function (like the fact that it is involutive). Literals also come with

```
Module Type LITERAL.
  Parameter t : Set.
  (* Stuff literals should provide *)
  Parameter mk_not : t → t.
  Axiom mk_not_invol : ∀l, mk_not (mk_not l) = l.
  ...
  (* t is an ordered type *)
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  ...
End LITERAL.
```

Fig. 2. A module type for literals

a decidable equality and a total order, which are necessary to later define finite sets of literals.

Once literals are defined, we can similarly define a module type `CNF` for formulas, as shown in Fig. 3. Such a module shall of course provide a type `formula` for formulas, and embed a module of type `LITERAL`. It also comes with two modules of finite sets[3]: `LSet`, whose elements are literals, and `CSet`, whose elements are sets of literals, as can be seen in the `with Module E := ...` part of their definitions. Finally, a module for formulas comes with a function `make`, that transforms a formula into a set of sets of literals; in other words, the function `make` performs conversion into conjunctive normal form.

```
Module Type CNF.
  Parameter formula : Set.

  Declare Module L : LITERAL.
  Declare Module LSet : FSetInterface.S with Module E := L.
  Declare Module CSet : FSetInterface.S with Module E := LSet.

  Parameter make : formula → CSet.t.
End CNF.
```

Fig. 3. A module type for formulas

We can now start the definition of a functor `SAT` parameterized by a module `F` of type `CNF` and which will implement our SAT solving algorithm without any knowledge about the actual formulas. The development can only use elements that are defined in `F`'s signature and this ensures modularity as well as reusability. Fig. 4 shows the beginning of the module, as well as the definition of sequents:

[3] `FSetInterface.S` is an interface for finite sets from the standard Coq library; it contains a module `E` which is the module of its elements.

a sequent, noted `G ⊢ D`, is simply a record with a set of literals `G` and a set of clauses `D`, as discussed in Section 2.

```
Module SAT (F : CNF).
  Import F.

  Record sequent : Set := {G : LSet.t; D : CSet.t}.
```

Fig. 4. The `SAT` module and a definition of sequents

The next step is the definition of the rules system presented in Fig. 1. We use an inductive definition shown[4] in Fig. 5 by enumerating all possible ways a derivation can be built from a given sequent. We call this inductive `derivable` and an object of type `derivable (G ⊢ D)` represents a proof derivation of sequent `G ⊢ D`. Note that each constructor faithfully follows from a rule of the original system. For instance, `AUnit` describes unit propagation, and `AElim` and `ARed` together describe the two BCP rules.

```
Inductive derivable : sequent → Set :=
| AConflict : ∀G D, ∅ ∈ D → derivable (G ⊢ D)
| AUnit : ∀G D l, {l} ∈ D → derivable (G, l ⊢ D \ {l}) →
    derivable (G ⊢ D)
| AElim : ∀G D l C, l ∈ G → l ∈ C → C ∈ D →
    derivable (G ⊢ D \ {C}) → derivable (G ⊢ D)
| ARed : ∀G D l C, l ∈ G → l̄ ∈ C → C ∈ D →
    derivable (G ⊢ D \ C, C \ {l̄}) → derivable (G ⊢ D)
| ASplit : ∀G D l, derivable (G, l ⊢ D) → derivable (G, l̄ ⊢ D) →
    derivable (G ⊢ D).
```

Fig. 5. The inductive definition of the proof system

3.2 Semantics

In the previous subsection, we defined literals, formulas and what it means for a sequent to be derivable. To show the correctness of this derivation system, we need a notion of semantics, i.e. what it means for a formula to be "true". We cannot directly (nor do we want to) rely on the prover's notion of truth because we are dealing with abstract formulas and not native Coq propositional formulas.

[4] In this figure and in the following, we use mathematical notations for set-related operations, rather than Coq's concrete syntax, for the sake of readability.

Once again we use Coq's functorization system and define semantics as a functor with respect to a module `F` of type `CNF`. A model is simply defined as a function assigning a propositional value `Prop` to any literal:

Definition `model := {M : L.t → Prop | ∀l, M l ↔ ~(M l̄)}.`

We use a dependent type to ensure that models are only functions which have the reasonable property of not assigning the same truth value to a literal and its negated counterpart. We can then use (via a coercion) a model `M` as a function interpreting a literal to an element of the propositional sort `Prop`. It is straightforward to define what it means for a model to satisfy a clause or a set of clauses, and when a formula is unsatisfiable:

Definition `sat_clause (M : model) (C : LSet.t) := ∃l ∈ C, M l.`
Definition `sat_goal (M : model) (D : CSet.t) := ∀C ∈ D, sat_clause M C.`
Definition `unsatisfiable (D : CSet.t) := ∀(M : model), ~sat_goal M D.`

This gives us a notion of satisfiability for clauses and formulas, but we also need to take the context of a sequent into account. What does it mean for a sequent `G ⊢ D` to be "unsatisfiable"? The context `G` of such a sequent represents a *partial assignment* and instead of considering *any* model when checking the satisfiability of `D`, it means that we have to consider *only* models that entail `G`. This leads to a notion of "submodel" that we define as follows:

Definition `submodel (G : LSet.t) (M : model) := ∀l ∈ G, M l`

Note that this definition of a submodel implies that `G` is a valid partial assignment, in the sense that it does not contain both a literal and its negation. From this notion of submodel naturally follows the correct definition of unsatisfiability for a sequent, which we call *incompatibility*, and states that there is no model of the context that also satisfies the clauses on the right-hand side:

Definition `incompatible (G ⊢ D : sequent) :=`
`∀(M : model), submodel G M → ~sat_goal M D.`

3.3 The Decision Procedure

Using the semantics we just defined, we can now proceed to prove the fundamental theorems about our derivation system. First in line is the soundness of the proof system:

if there exists a derivation of the sequent $\emptyset \vdash$ `D`, `D` *is unsatisfiable*

We actually prove something more general than this statement, using the notion of incompatibility that we just described:

Theorem `soundness : ∀S : sequent, derivable S → incompatible S.`

The special case where the context of sequent `S` is empty yields exactly the above statement. This theorem can be proved by a structural induction on the derivation of `S`: for each case, it is sufficient to show that if the premises are incompatible, then so is `S`. We gave the informal arguments when describing the DPLL procedure in Section 2. The Coq proof is not difficult (about 50 lines of tactics).

Conversely, the completeness of the algorithm could be expressed by the following statement:

Theorem `completeness : ∀S : sequent, incompatible S → derivable S.`

There are at least two reasons why we do not prove completeness in this particular form:

- We do not only want full equivalence between the notions of derivability and incompatibility, but we also want a decision procedure, i.e. a function capable of telling if a given formula is unsatisfiable or not. Proving such a theorem of completeness would certainly give us an equivalence between the derivability of a sequent and its incompatibility, thus bringing the problem of deciding satisfiability down to the one of deciding derivability. However, deciding derivability amounts to try and build a derivation for a given sequent if possible, and it is a proof that actually encompasses the completeness theorem presented above. Thus, we want to avoid doing the same job twice.
- Not only do we need a decision procedure that we could extract, but we want to be able to use that procedure in Coq through the mechanism of reflection, i.e. by actually computing the proof search in the system. It is well known that procedures with propositional contents cannot be executed as efficiently as computational-only functions, because in the first case, proofs need to be replayed along with computations. Thus, we do not want to encode the decision procedure as part of a general completeness theorem.

For these reasons, we will build the decision procedure in two steps: first we will program a function without propositional content to implement the actual decision procedure, and then we will show that its results are correct. This function will not return any "complex" information, but only `Sat G` if it has found a partial model `G`, and `Unsat` otherwise:

```
Inductive Res : Set :=
  Sat : LSet.t → Res
| Unsat.
```

The decision procedure *per se* can now be implemented as a recursive function returning such a result:

```
Fixpoint proof_search (G ⊢ D : sequent) n {struct n} : Res :=
  match n with
    | O ⇒ Sat ∅ (* Absurd case *)
    | S n0 ⇒
      if D = ∅ then Sat G (* Model found! *)
      else
        if ∅ ∈ D then Unsat (* Rule AConflict *)
        else ...
  end.
```

Because the recursion is not structural, we use an extra integer argument n, and we will later make sure that we call the function with an integer large enough so that n never reaches 0 before the proof search is completed. This short excerpt of the function `proof_search` shows that it proceeds by trying to apply some rules one after another, with a given *strategy*. Here, the function first checks if the problem is empty, in which case it returns the current context as a model; otherwise, it checks if the empty clause is in the formula, in which case it returns `Unsat`. We do not explicit the strategy further because it has no real interest in the scope of this paper.

The first theorem about `proof_search` states that when it returns `Unsat`, it indeed constructed a derivation on the way:

```
Theorem proof_unsat : ∀n S, proof_search S n = Unsat → derivable S.
```

The proof follows the flow of the function and shows that each recursive call that was made corresponds to a correct application of the derivation rules. One may wonder why we didn't construct this derivation in `proof_search`, so as to return it with `Unsat`: the reason is that a derivation contains proofs (in side conditions) and had we done so, our function would not have been 100% computational anymore.

The second theorem about `proof_search` is the one that encompasses completeness: it states that if `Sat M` has been returned, it is indeed a model of the formula and of the context[5].

```
Theorem proof_sat :
  ∀n S M, μ(S) < n → wf_context S.G → proof_search S n = Sat M →
    S.G ⊆ M ∧ sat_goal M S.D.
```

A couple of remarks about this theorem are necessary:

- μ is a *measure* of a sequent that we have defined in Coq, and for which we proved that it decreases for every recursive call in the algorithm. We could have defined the function by a well-founded induction on this measure, but it is computationally slightly more efficient to use the extra integer. This is

[5] Technically, the set returned is not a model because it is only partial; it can be completed into a model though, as long as it is a valid partial assignment, and we simplified the actual details here since they seem cumbersome.

a well-known technique to transform non-structural inductions in structural inductions [1]. When calling `proof_search` on a sequent `S`, a suitable integer is $\mu(\texttt{S}) + 1$;
- we need an extra hypothesis that the context remains well-formed (`wf_context S.G`), which means that it doesn't contain a literal and its negation. This is not guaranteed by the derivation rules because the side conditions were purposely very loose in order to allow any kind of strategy. Here, it is our strategy that guarantees this invariant is never broken, and this is part of the completeness proof.

Together with the soundness theorem, these two theorems show that `proof_search` is a decision procedure for unsatisfiability and we can now prove the following theorem:

Theorem `dpll_dec (Δ : CSet.t) :`
`{incompatible (∅ ⊢ Δ)} + {~incompatible (∅ ⊢ Δ)}.`

The definition of `proof_search` and the proofs of its properties require 700 lines of code.

4 Deriving a Reflexive Tactic

The procedure we have developed so far can be extracted to an OCaml functor (cf. Section 5), but we are also greatly interested in directly using this procedure as a tactic to solve goals in our proof assistant.

4.1 Reification

In order to use our SAT solver on Coq propositional formulas, we need to instantiate the `SAT` functor. This raises the question of the actual representation of formulas and literals: we need to build modules of types `LITERAL` and `CNF` that will represent Coq formulas.

A natural choice for the type of literals would be to directly use the type `Prop` of propositions, but this is impossible because we need to build sets of literals, and more generally we need to be able to decide if two given propositions are equal or not. Indeed, consider the formula `A ∧ ~A`: we need to know that the propositional variable `A` is the same on both sides to conclude that this formula is unsatisfiable. Since the only decidable equality on sort `Prop` is the one that is always true, we cannot use `Prop` as the type of literals.

Instead, we resort to Coq's metalanguage Ltac [8]. This language provides pattern-matching facility on Coq terms, and thereby allows us to check the syntactic equality of propositional terms at a metalevel. We will use this language to build, for a given propositional formula `F`, an *abstract representation* of `F` on which we will be able to apply the algorithm. This process, called *reification* or sometimes *metaification*, has already been used and described in [9, 11] .

Using Ltac, we first build a function `get_vars` which traverses a formula `F` and retrieves a list of all the propositional variables of `F`. We define another function `list_to_map` that turns such a list into a balanced map. This map now contains all the propositional variables of `F` and provides an efficient way to search for a particular variable into a map. For instance, if `F` is the following formula:

```
F:  A ∧ (~B ∨ (p A C)) ∧ (∀D, (p D D)).
```

the result of `list_to_map (get_vars F)` will be a map containing the variables `A`, `B`, `(p A C)` and `∀D, (p D D)`. In particular, the last variable is abstracted because our propositional language does not include quantifiers. Given this map, we are able to represent variables by their *path* in the map. It is now straightforward to create the module `LPROP` of literals, where a literal is just a `path` in the map and a boolean saying if it is negated or not, and the `mk_not` function a simple inversion of this boolean:

```
Module LPROP <: LITERAL.
  Inductive path : Set := Lft : path → path | Rgt : path → path | E.
  Definition t := path × bool.
  Definition mk_not (p,b) : t := (p, negb b).
  ...
End LPROP.
```

We can move on to defining the corresponding types for formulas. We will for now assume that our formulas are already in conjunctive normal form, and we address the problem of conversion to CNF later in Section 4.2. In Fig. 6, we show an excerpt of the module `CNFPROP` of type `CNF`, which implements our type of formulas. Its literals are, of course, the literals of the module `LPROP` we just defined. Formulas and clauses are defined in a very natural way by two inductives: a formula is either a clause or a conjunction of formulas; a clause is a literal or a disjunction of clauses. This representation makes the function `make` converting a formula to a set of sets of literals (not represented here) really straightforward.

```
Module CNFPROP <: CNF.
  Module L := LPROP.

  Inductive clause : Set :=
  | COr : clause → clause → clause
  | CLit : L.t → clause.

  Inductive formula : Set :=
  | FAnd : formula → formula → formula
  | FClause : clause → formula.
  ...
End CNFPROP.
```

Fig. 6. A module for propositional formulas

We also define an interpretation function `interp` such that `interp v f` interprets an object `f` of type `formula` to its propositional counterpart in Coq. The extra argument `v` is the map binding `path`s to concrete propositional variables, which is needed to interpret literals. In particular, `interp` uses the following subroutine to interpret literals, where `lookup id v` returns the proposition bound to `id` in the map `v`:

```
Definition linterp (l : L.t) : Prop :=
  match l with | (id, true)  ⇒   lookup id v
               | (id, false) ⇒ ~(lookup id v) end.
```

The last step of the reification process is to build a tactic in Ltac, that, for a given formula `F` in Coq's propositional language, builds an abstract formula `f` of type `formula` and a map `v` such that `interp v f = F`. We have already covered the construction of the map `v`. The construction of the formula `f` is realized by a couple of recursive Ltac tactics which analyze the head symbol of the current formula to construct the corresponding abstract version. For instance, the top-level function matches conjuncts and goes like this:

```
Ltac reify_formula F v :=
  match constr:F with
    | and ?F1 ?F2 ⇒
      let f1 := reify_formula F1 v with f2 := reify_formula F2 v in
        constr:(FAnd f1 f2)
    | ?F ⇒
      let c := reify_clause F v in constr:(FClause c)
  end.
```

Now, if we go back to our previous example, and if we take this formula as our current goal, we can use the tactics we just described to build a suitable map, reify the goal in an abstract formula `f`, and replace the current goal by the interpretation of `f`. The tactic `change` asks Coq to perform the so-called *conversion rule*: it computes the interpretation and checks that it is indeed equal to the original goal.

```
============================
 A ∧ (~ B ∨ (p A C)) ∧ (∀D : Prop, (p D D))

>  match goal with | ⊢ ?F ⇒
     let v := list_to_map (get_vars F) in
     let f := reify_formula F v in
       change (interp v f)
   end.

============================
 interp (...) (FAnd (FClause ...) (FAnd ... ...))
```

4.2 The Generic Tactic

At this point, in order to turn our development into a user-friendly generic tactic, we still need to address a couple of issues.

Conversion to normal form. Before running the actual proof search, a formula should be put in CNF. If it is not in CNF, then some subformulas will be abstracted (like the quantified part in our example above). We could have coded the conversion to normal form as a Coq function running on reified formulas, but we decided to avoid this additional tedious work by again using tacticals. Coq provides a tactic named `autorewrite` which performs automatic rewriting of expressions. When fed with a set of (oriented) equalities describing a normalizing system, `autorewrite` will transform an expression into its normal form with respect to this system. Thus, we encode the conversion into CNF as a set of rewriting rules[6]: linearizing implications, pushing negations to the atomic variables, distributing disjunction over conjunction, etc. Some of these rules only hold in classical logic, and we discuss this further in Section 5.

Lifting the Semantics. In Section 3.2, we defined a notion of semantics and proved the properties of our decision procedure in this respect. Recall that a model is a function associating a propositional variable to every literal. Thus, it turns out that our abstract formulas have a very canonical notion of model: the interpretation of the literals itself. Indeed, if `l` is a literal representing a variable `A` of type `Prop`, the canonical model satisfies `l` if and only if there is a proof of `A`. This result lifts to clauses and formulas, and we can prove this adequation:

Theorem `adequation :`
`∀(f : formula), interp v f → sat_goal (model v) (make f).`

---

where `model v` is the canonical model interpreting literals in the map `v`. This theorem can be read as : "if there is a proof of a formula `F`, then its reified counterpart `f` is satisfiable". Together with the soundness of the decision procedure, this gives us the following fact:

Corollary `validity : ∀(f : formula),`
`proof_search (∅ ⊢ (make f)) = Unsat → ~(interp v f).`

We can now wrap everything up in a high-level tactic `unsat` that builds the conjunction `F` of all the hypotheses in the context, turns it into normal form, constructs the abstract version `f` of `F`, changes `F` in the context to `interp v f`, and finally applies the `validity` theorem to bring the current goal down to a proof of `proof_search (∅ ⊢ (make f)) = Unsat`. Coq is then asked to `compute` the left-hand side of this equation, which triggers the actual proof search. If the procedure returns `Unsat`, the goal is trivial and the proof is completed. Otherwise, it returns a countermodel and we print it out, since it can be very valuable to the user in order to understand why the tactic did not succeed.

The same mechanism can also be used to prove the validity of a current goal `F`, by applying double negation and trying the `unsat` tactic on `~F`. We provide a tactic called `valid` that performs these operations. The definitions and proofs for `unsat` and `valid` represent about 400 lines.

[6] In practice, we use several complementary rewriting systems, because for efficiency reasons, some transformations must be done before others, e.g. rewriting of implications.

5 Results and Examples

We start this section by giving a small example of how the tactic `unsat` can be used in practice. Suppose our goal is the following propositional formula where variables `A` to `D` have type `Prop`:

```
  ============================
  A ∧ (C ∨ ~B ∧ (~D → ~A)) → D ∧ D ∧ ~A

> unsat.
The formula in not valid.
The following countermodel has been found :
D  :  true
B  :  false
A  :  true
```

If we try to apply `unsat` to this goal, the tactic will try to show that the left-hand side of the implication is unsatisfiable. Since it is not, the tactic fails and prints out the countermodel shown above: indeed, one can easily verify that this valuation makes the goal false. We can use this countermodel to add complementary hypotheses to our formula, for instance that `B` is true and `A` is false. By doing so, we see that the `unsat` tactic now succeeds in about one tenth of a second:

```
  ============================
  A ∧ (C ∨ ~B ∧ (~D → ~A)) → B ∧ ~A → D ∧ D ∧ ~A

> Time unsat.
Proof completed.
Finished transaction in 0. secs (0.108007u,0.s)
```

Our experiments with this tactic show that goals that occur in practice during an interactive proof are reasonably small and the biggest part of the time needed to prove a goal is often spent in the conversion into CNF. We also tried our tactic on artificial benchmarks from the SATLIB initiative [14], in comparison to the performance of the extracted SAT solver, and the results are summarized in Fig. 7. The second row gives the number of variables and clauses in each problem. The fourth row gives the time spent by the tactic on the reification alone. The last two rows respectively show the number of nodes in the proof derivation, and the number of Split nodes. These results show that both the tactic and the extracted implementation perform rather badly on these tests. In some cases, the tactic was unable to succeed in reasonable time (less than a couple of hours), even though the input problems were already in CNF. This can be explained by the fact that we used a very unoptimized version of the DPLL procedure and that these benchmarks are especially good at stressing optimizations of DPLL. The high number of branching nodes illustrates the fact that we are exploring a great part of the search tree.

An important characteristic of the DPLL procedure is that its performance heavily depends on how decision literals, i.e. literals that are introduced in the rule Split, are chosen. Considering this, we took advantage of the modularity

	aim-1	aim-2	aim-3	aim-4	phole 5	phole 6	phole 7	phole 8
vars/clauses	50/80	50/80	50/80	50/80	30/81	42/133	56/204	72/297
OCaml time	7m20s	33s	7m	1m50s	0.3s	2.0s	25s	6m
Coq time	-	2h30m	-	-	15s	3m20s	1h9m	-
whereof reification	25s	22s	26s	25s	3.5s	10s	24s	56s
Nodes	13M	1.3M	33M	6.8M	12k	111k	1.2M	15M
Branching	630k	64k	2M	400k	370	3250	32k	378k

Fig. 7. Some results on harder SAT problems

of our development to allow the user to instantiate our SAT solver with a customized function called `pick`, and coming with the module `CNF`. This function `pick` is used by the procedure to pick a literal from the current goal when applying the rule `Unsat`. The correctness of the procedure only relies on a couple of easy properties that the user shall prove for his customized function `pick`. This way, the SAT solver does not depend on particular heuristics and new efficient heuristics can be implemented by the front-end user.

Another advantage of our modular approach comes to light when considering whether our procedure works in classical or intuitionistic logic. A SAT solver for propositional logic is inherently classical, as the rule Split, for instance, relies on the fact that a literal must be either true or false. Nonetheless, some instances of the excluded-middle are provable in intuitionistic logic and there may be cases where the decision literals used in the proof search are actually decidable (e.g. membership of an element in a finite set). For this reason, we made sure that our `SAT` functor did not depend on the excluded-middle, but that we only put the restriction on models instead: a model `M` shall have the property that, for any literal l, `M` $l \vee$ `~M` l is provable. Of course, when creating the tactics `unsat` and `valid`, we were dealing with any kind of propositional atoms, and therefore we had to assume the excluded-middle. Another possible approach, but that we have not implemented yet, would be to generate one subgoal for every decision literal in the proof and let the user deal with them. To ensure that users proving goals with our generic tactics are aware of the fact that they are working in classical logic, we encapsulated the top-level tactic definitions in a functor `LoadTactic` expecting the classical axiom in input:

```
Module Type K.
  Axiom classic : ∀P, P ∨ ~P.
End K.
Module LoadTactic (K : K).
  Import K.
  ...
End LoadTactic.
```

To further illustrate our point, it is possible to instantiate our procedure on boolean formulas: the difference between propositional and boolean formulas in Coq is that booleans are a type in `Set`, whose equality is decidable and which can be extracted to ML booleans. In particular, a canonical model for booleans would be:

Definition `model (b : bool) :=` **if** b **then** `True else False.`

and such a model has the property required by the SAT solver. Therefore, it is possible to use our SAT solver to create a tactic deciding satisfiability of boolean formulas *in intuitionistic logic* by just following the same steps as in Section 4. The whole development presented so far, including these extra tactics on booleans, is available online[7] and can be compiled with Coq v8.1pl3.

6 Related Work and Conclusion

Related Work. Decision procedures for classical propositional logic have already been formalized in proof assistants before. For instance, Harrison [13] on one side, and Letouzey and Théry [15] on the other side, both presented formalizations of Stålmarck's algorithm, respectively in HOL [10] and Coq. In both cases, they extracted certified versions of the algorithm which also produced traces. The traces were then used to reconstruct a proof from inside the prover, yielding a "partially reflexive" tactic. In our work, we focused on developing a totally reflexive tactic in order to avoid the cost of reconstructing a proof from traces. Harrison showed how to implement Binary Decision Diagrams (BDDs) as a HOL derived rule [12], with pretty much the same approach as in [13]. Closer to our work, Verma *et al.* [16] implemented reflexive BDDs in Coq as a first step towards a reflexive model checker in Coq. Our approach is similar in the sense that our SAT solver is the cornerstone of the SMT solver we are planning to formalize.

Conclusion. We have presented the formalization in Coq of a DPLL-like procedure deciding satisfiability of propositional formulas. We extracted a certified OCaml implementation of this SAT solver, which is about 1700 lines long (including 800 lines for the interface). We also instantiated our procedure on Coq's propositional formulas in order to derive a reflexive tactic. Even if its performance remains limited, our tactic can be used in Coq to automatically discharge valid goals or unsatisfiable contexts.

The use of Ltac allowed us to avoid coding parts of the tactic in OCaml, and we hope our development illustrates what we think is a key feature of Coq: the ability to reason and program about terms directly at the top-level, without any knowledge of its inner mechanisms, and without having to compile stubs coded in OCaml. We also showed how modularity can be beneficial, just as in a usual programming language. By using Coq's module system to develop our procedure in a very modular way, we were able to use this procedure on different types of propositional formulas without much pain and to give the user the ability to define his own heuristics. Modularity also ensures that the decision procedure can be used in an intuitionistic setting on suitable data structures.

We plan to work in several directions, mainly by adding common optimizations to the DPLL procedure in order to prune some parts of the proof search.

[7] `http://www.lri.fr/~lescuyer/sat/unsat.tgz`

We are also greatly interested in combining this decision procedure with other, more specific, decision procedures like linear arithmetic or congruence closure.

References

1. Y. Bertot and P. Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, 2004.
2. J. Chrzaszcz. Implementation of modules in the Coq system. In *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.
3. S. Conchon and E. Contejean. The Ergo Theorem Prover. `http://ergo.lri.fr/`.
4. The Coq Proof Assistant. `http://coq.inria.fr/`.
5. J. Courant. A module calculus for pure type systems. In R. Hindley, editor, *Proceedings fo the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, Nancy, France, 1997. Springer-Verlag LNCS.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5(7):394–397, 1962.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
8. D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France)*, volume 1955 of *LNCS/LNAI*, pages 85–95. Springer-Verlag, November 2000.
9. D. Delahaye and M. Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier (France)*. INRIA, Janvier 2001.
10. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.
11. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOLs*, pages 98–113, 2005.
12. J. Harrison. Binary decision diagrams as a HOL derived rule. In T. F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 254–268, Valletta, Malta, 1994. Springer-Verlag.
13. J. Harrison. Stålmarck's algorithm as a HOL derived rule. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234, Turku, Finland, 1996. Springer Verlag.
14. H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. pages 283–292.
15. P. Letouzey and L. Théry. Formalizing Stålmarck's algorithm in Coq. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2000.
16. K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In J. He and M. Sato, editors, *Proceedings of the 6th Asian Computing Science Conference (ASIAN 2000)*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181, Penang, Malaysia, Nov. 2000. Springer.
17. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.