Comprehensive Exam Report 3 - Proof Generation

Arjun Viswanathan

1 Introduction

SMTCoq [6] is a skeptical cooperation between the Coq proof assistant, and (SAT and) SMT solvers [2], implemented as a Coq plugin. This integration is useful for the ITP (Coq) since it allows proof goals in Coq to be automatically discharged by an SMT solver. However, unlike Coq's small, verifiable kernel, SMT solvers have tens of thousands of lines of code and are more prone to bugs. Since an SMT solver's code base is much harder to verify, many SMT solvers produce proofs of their results in addition to a claim of (un)satisfiability of a set of formulas. For satisfiable formulas, a proof is simply a satisfying model which is easy to specify and check (for quantifier-free formulas). For unsatisfiable formulas, however, SMT solvers need to show that the input formulas are inconsistent. They do this by reducing the formulas to a trivial unsatisfiable formula $\bot$, or the empty clause $\bot$. A proof producing SMT solver is integrated with Coq by checking the proof inside Coq for a particular representation of SMT formulas in Coq. The checker is proven correct, lifting up the formulas to Booleans goals in Coq via a process called computational reflection (described in Report 2).

An added complication in this integration of SMT solvers with Coq is the incompatibility of their logics. Propositions in Coq have the `Prop` type which implements a constructive or intuitionistic logic. SMT solvers, on the other hand, implement a classical logic. This incompatibility is overcome in SMTCoq by using SMT solvers to prove Boolean goals in Coq (of type `Bool`) that are classical, and additionally, supporting a fragment of propositions that don't violate the laws of constructive logic.

The first half of this report discusses the proof production systems of SMT solvers. The second half explores the difference between the logics of SMT solvers and Coq, and SMTCoq's efforts to prove formulas in Coq that SMT solvers can help with, while being logically consistent.

2 Preliminaries

Many-sorted first-order logic (MSFOL) extends first-order logic with types (or sorts). We present the syntax of MSFOL in this section and the semantics are presented in [2]. Syntactically, the components of MSFOL are sorts σ, terms t, and formulas ϕ. Sorts are atomic entities that represent types. Function types — $(\sigma_1, \ldots, \sigma_n) \to \sigma$ — and relation types — $(\sigma_1, \ldots, \sigma_m)$ are defined over sorts, and are types of functions and predicates. Terms t and formulas ϕ are specified as:

$$t := x^\sigma \mid f^{(\sigma_1,\ldots,\sigma_n)\to\sigma}(t_1, \ldots, t_n)$$
$$\phi := \bot \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \forall x^\sigma.\phi \mid t_1 = t_2 \mid P^{\sigma_1,\ldots,\sigma_m}(t_1, \ldots, t_m)$$

Terms are either sorted variables, or applications of sorted function symbols to terms. Formulas are constants or logical connectives applied to other formulas, quantified formulas, equality over terms, or predicates symbols applied to terms. Connectives $\top$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and the existential quantifier $\exists$ can be specified using the connectives and quantifiers mentioned above.

An *atomic formula* is one of $\bot$, $t_1 = t_2$, $P^{\sigma_1,\ldots,\sigma_m}$ and a *literal* is an atomic formula or it's negation. A *clause* is a disjunction of literals. All formulas can be converted to conjunction normal form (CNF) which is a conjunction of clauses. For ease of notation, a formula in CNF is also represented as a set of clauses within curly braces. $\bot$, the symbol representing false, is overloaded to also represent the empty clause, which is trivially unsatisfiable (unlike the empty set, or the empty conjunction which is trivially satisfiable).

Informally, a formula is satisfiable if there exists a satisfying model — an assignment of well-sorted values to its variables such that the interpretation of the formula is $\top$; it is unsatisfiable if no such model exists; it is valid if, for all models its interpretation is $\top$. This interpretation is recursively defined in Barrett et al. [2]. We use the notation $var \mapsto val$ to signify the assigment of value val to variable var, and group assignments in curly braces. For example, $a \mapsto \top$ assigns Boolean value $\top$ to a. $\models$ stands for logical entailment and $\models_T$ stands for entailment in theory T. A formula is entailed from nothing, if it is valid. For example, $a \wedge b \models a$, $x > 3 \models_{LIA} x > 0$, and $\models_{LIA} x > 3 \Rightarrow x > 0$ are all logically sound entailments — the first one propositionally, and the last two in the theory of linear integer arithmetic.

Declarative rules of the form

$$\frac{P_1 \quad \ldots \quad P_n}{C}\ \textit{rule\_name}$$

specify that if the premises $P_1, \ \ldots, \ P_n$ hold, then the conclusion C holds. The resolution rule, which is used heavily in DPLL(T) proofs, simplifies two clauses to a new clause as follows:

$$\frac{\phi_1 \vee \cdots \vee \phi_n \vee \chi \qquad \neg\chi \vee \psi_1 \vee \cdots \vee \psi_m}{\phi_1 \vee \cdots \vee \phi_n \vee \psi_1 \vee \cdots \vee \psi_m}\ \textit{Res}$$

The following is an instance of the resolution rule.

$$\frac{a \vee \neg b \quad b \vee c}{a \vee c}$$

Given that two clauses hold, and a *pivot*, which is a literal that occurs with opposite polarities in each clause (b in the above example), the resolution rule concludes that the combination of the two clauses without either occurrence of the pivot holds.

3 DPLL(T) Proofs

The typical SMT solver composes a SAT solver with multiple theory solvers in an abstraction-refinement cycle, where the SAT engine tries to find a satisfying model of the propositional abstraction of a given set of constraints, and the theory solvers find a refutation of the refinement of the model, if one exists. This cycle is guided by the DPLL(T) algorithm, which is an extension of the DPLL [7] (Davis-Putnam-Logemann-Loveland) algorithm that adds theory-level reasoning.

An SMT solver converts its input constraints into conjunction normal form (CNF), which is a conjunction of clauses. The DPLL(T) algorithm then tries to find a satisfying assignment for these clauses, and otherwise, by exhaustion concludes their unsatisfiability. The steps taken to conclude the unsatisfiability can be translated into a chain of resolutions of the input clauses to conclude the empty clause $\bot$ from them, which is the most basic form of unsatisfiability.

A proof of unsatisfiability is a tree that derives, from the input formulas and *theory lemmas* at the leaves, the empty clause $\bot$ at the root. The input formulas are converted to CNF via CNF conversion rules that are part of the proof tree. We omit the CNF conversion phase and start with formulas that are already in CNF. Thus, proof trees as shown in this report have input clauses and theory lemmas at the leaves, and $\bot$ at the root, where clauses are reduced in the tree using the resolution rule. Theory lemmas are the theory solvers' means of refining the SAT solver's model for the input. A theory lemma, produced by the corresponding theory solver, is a clause that is valid in a theory. The theory solvers use theory-specific rules to prove these lemmas. A theory lemma in the proof tree is supported by a satellite proof of the lemma provided by the theory solver.

In what follows, we explain the DPLL(T) algorithm and the extraction of a resolution proof from an execution of it on an unsatisfiable input.

3.1 The DPLL(T) Algorithm

DPLL(T) was originally presented in Nieuwenhuis et al. [8], and reiterated in Katz et al. [5] as a transition system. We briefly present the transition system here. Katz et al. also present the proof-production system of the CVC4 [3] SMT solver for quantifier-free logics, that we follow in this report. While other SMT solvers' proofs might vary in specifics from CVC4's, the general idea of a resolution-based proof is still common across solvers.

As previously mentioned, the DPLL(T) algorithm can be understood as trying to satisfy the input constraints at the propositional level, with refinements of models from theory solvers. Checking whether a formula is unsatisfiable at the propositional level involves incrementally assigning literals deterministically to satisfy clauses (propagation), and when that isn't an option, by guessing an assignment. When a guess goes down a path that results in unsatisfiability, the algorithm backtracks and checks whether the formula is still unsatisfiable when the guess is undone (and prevented from being repeated).

We now formalize the DPLL(T) algorithm as a state transition systems that operates on a set of transition rules. The state of the system is either $fail$ or $\langle M, F, C\rangle$ where M is a sequence that represents the current *context*, that is, the current assignment of literals in the formula; a literal in M is preceded by a $\bullet$ if the literal was a decision, that is, it was guessed. For some variable x, the assignment $x \mapsto \top$ is added to the context by appending x to M, and its complement is added by appending $\neg x$. F is a set of clauses that is equisatisfiable with the input formula. The initial formula F_0 may be transformed by the algorithm but the transformation is always equisatisfiable with the initial input. C represents the conflict clause, a clause from F that is falsified by the current assignment (or an equisatisfiable transformation of a clause from F) — it is either a singleton set with the conflict clause, or the empty set representing no conflict. $\prec_M$ is an ordering on the literals in M — $l_1, \dots, l_n \prec_M l$ if $l_1, \dots, l_n$ occur before l in M. If $M = M_0 \bullet M_1 \bullet \dots \bullet M_n$, such that no $\bullet$ occurs in the M_i's, then each M_i is a *decision level*, $M^{[i]}$ denotes $M_0 \bullet \dots \bullet M_i$, and and `lev` is a function that maps a literal in M to its decision level. Lit_F represents the set of literals occurring in F.

The initial state of the system is $\langle (), F_0, \emptyset\rangle$. The final state is one of:

- $fail$, when F_0 is unsatisfiable.
- $\langle M, F, \emptyset\rangle$ where F_0 is satisfiable, F is a transformation of F_0 by the algorithm that is equisatisfiable with F, and $M \models F$.

$$\frac{l_1 \vee \cdots \vee l_n \vee l \in F \quad \neg l_1, \ldots, \neg l_n \in M \quad l, \neg l \notin M}{M := Ml}\ \mathit{Prop}$$

$$\frac{l \in \mathit{Lit}_F \quad l, \neg l \notin M}{M := M \bullet l}\ \mathit{Dec}$$

$$\frac{C = \phi \quad l_1 \vee \ldots \vee l_n \in F \quad \neg l_1, \ldots, \neg l_n \in M}{C := \{l_1 \vee \cdots \vee l_n\}}\ \mathit{Confl}$$

$$\frac{C = \{\neg l \vee D\} \quad l_1 \vee \cdots \vee l_n \vee l \in F \quad \neg l_1, \ldots, \neg l_n \prec_M l}{C := \{l_1 \vee \cdots \vee l_n \vee D\}}\ \mathit{Expl}$$

$$\frac{C = \{l_1 \vee \cdots \vee l_n \vee l\} \quad \mathtt{lev}\ \neg l_1, \ldots, \mathtt{lev}\ \neg l_n \leq i < \mathtt{lev}\ \neg l}{C := \phi \quad M := M^{[i]} l}\ \mathit{Backj}$$

$$\frac{C \neq \phi}{F := F \cup C}\ \mathit{Learn}$$

$$\frac{C \neq \phi \quad \bullet \notin M}{\mathit{fail}}\ \mathit{Fail}$$

Figure 1: Transition rules at Propositional level

The transition system goes from the initial state to one of the possible final states by a non-deterministic application of the transition rules presented in Figure 1 and Figure 2. In practice, they are applied using a particular strategy for efficiency. A rule is applicable when all its premises hold, and it is applied to make the conclusion true.

The propositional rules from Figure 1 model the behavior of the SAT solver. Propagations (*Prop*) force assignments of Boolean values to literals that are propositionally entailed. For instance, if the input formula is $\neg a \wedge (a \vee b)$, then the assignment $a \mapsto \bot$ is required to satisfy the first conjunct, $\neg a$. Now, $b \mapsto \top$ is the only way to satisfy the second conjunct, $(a \vee b)$. Both of these assignments are logically entailed propagations, given the goal of satisfying the formula. Sometimes a deterministic choice might not be entailed, instead, an assignment must be guessed using the decide rule (*Dec*). For example, to satisfy $(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a)$, a formula with no possible propagations, one of the literals must be non-deterministically assigned. For instance, the guess $a \mapsto \top$ forces propagations $b \mapsto \top$ and $c \mapsto \bot$. Propogations and/or decisions could lead to a model in which the input formula is not satisfied. The conflict rule (*Confl*) recognizes this. In the previous example, the decision $a \mapsto \top$ led via propagations to the assignment $\{a \mapsto \top,\ b \mapsto \top,\ c \mapsto \bot\}$, which falsifies the second conjunct of the formula and satisfies the others. The *Confl* rule recognizes this clause $(\neg b \vee c)$ to be the conflict clause. Once a conflict is encountered, where no previous decisions were made, then the conflict was entailed, and the unsatisfiability of the input formula can be concluded via the *Fail* rule. If, there are previous decisions at the time of conflict, then the backjump rule

$$\frac{l \in \mathit{Lit}_F \quad \models_i l_1 \vee \cdots \vee l_n \vee l \quad \neg l_1, \ldots, \neg l_n \in M \quad l, \neg l \notin M}{M := Ml}\ \mathit{Prop}_i$$

$$\frac{C = \phi \quad \models_i l_1 \vee \cdots \vee l_n \quad \neg l_1, \ldots, \neg l_n \in M}{C := \{l_1 \vee \cdots \vee l_n\}}\ \mathit{Confl}_i$$

$$\frac{C = \{\neg l \vee D\} \quad \models_i l_1 \vee \cdots \vee l_n \vee l \quad \neg\, l_1, \ldots, \neg\, l_n \prec_M l}{C := \{l_1 \vee \cdots \vee l_n \vee D\}}\ \mathit{Expl}_i$$

Figure 2: Transition rules at Theory level

(*Backj*) uses the conflict clause to undo one or more decisions and add a propagated literal to the context. As a special case of this, the newly propagated literal is the complement of a previously assigned (and undone) decision. To do this, it might need the explain rule (*Expl*) to modify the conflict clause using resolution. Since the conflict clause is initially a clause in the input, and it is only modified by resolving it with other input clauses, it is always entailed by the input. At any point, a modified conflict clause may be learned (*Learn*) by adding it to the set of clauses in the input formula. The restriction in the *Backj* rule that requires a literal in the conflict clause to be at least one decision level above the rest of the clause, forces the transition system to apply the *Expl* rule so that the algorithm is able to backtrack further than chronological backtracking (where the most recent decision is undone), to reach the source of the conflict. In our example, since there is only one decision so far, the algorithm backjumps to the assignment of a and flips it to $a \mapsto \bot$. The propagations $b \mapsto \top$ and $c \mapsto \bot$ resulted from the assignment $a \mapsto \top$, and these are undone by the backjump. After flipping our initial decision, we have $a \mapsto \bot$ which satisfies the first and third clauses of the formula $(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a)$. Since there are no further propagations, the decision $b \mapsto \bot$ can be made to satisfy the second clause, and the entire formula. The theory-level rules from Figure 2 model the behavior of an abstract theory solver. The Prop_i, Confl_i, and Expl_i rules work exactly like their propositional variants, except that they can use theory lemmas, that are valid clauses in a theory produced by the theory solver. Additionally, there is a Learn_i theory-level rule that is not presented here for simplicity.

The transition system is *refutationally sound*: if an execution starting with $\langle \phi, F, \phi \rangle$ ends with $fail$, then F is unsatisfiable in T. Under suitable restrictions on the theory T involved, it is also *complete*: for every exhausted execution (an execution in which no rule other than Learn_T applies) starting with $\langle \phi, F, \phi \rangle$, and ending with $\langle M, F_0, \phi \rangle$, M is a satisfying model of F_0 (equisatisfiable with F) in T.

Example 3.1 Consider the following CNF formula:

$$F : \{f(g(z)) = h(y) \vee x \neq y,\;\; f(g(z)) \neq h(y) \vee x \neq y,\;\; x = y \vee f(m) = f(n),\;\; f(m) = x,\;\; f(n) = y\}$$

Each term contains additional information in the theory of equality over uninterpreted functions (EUF) where the only predicate is equality, and all functions are uninterpreted. However, this additional information isn't available to the SAT solver at the propositional level. We consider the following abstraction of the theory literals in F, for simplicity of representation.

M	**F**	**C**	**Rule**	**Step**
	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E$		*Prop* (D)	1
D	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E$		*Prop* (E)	2
DE	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E$		*Dec* (A)	3
$DE \bullet A$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E$		*Prop* $(\neg A \vee \neg B)$	4
$DE \bullet A \neg B$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E$		*Prop* $(B \vee C)$	5
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E$		Confl_{EUF} $(\neg C \vee \neg D \vee \neg E \vee B)$	6
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E$	$\neg C \vee \neg D \vee \neg E \vee B$	*Expl* (D)	7
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E$	$\neg C \vee \neg E \vee B$	*Expl* (E)	8
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E$	$\neg C \vee B$	*Expl* $(B \vee C)$	9
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E$	B	*Expl* $(\neg A \vee \neg B)$	10
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E$	$\neg A$	*Learn* $(\neg A)$	11
$DE \bullet A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$	$\neg A$	*Backj* $(\neg A)$	12
$DE \neg A$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$		*Prop* $(A \vee \neg B)$	13
$DE \neg A \neg B$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E,\ \neg A$		*Prop* $(B \vee C)$	14
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$		Confl_{EUF} $(\neg C \vee \neg D \vee \neg E \vee B)$	15
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E,\ \neg A$	$\neg C \vee \neg D \vee \neg E \vee B$	*Expl* (D)	16
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C, D,\ E,\ \neg A$	$\neg C \vee \neg E \vee B$	*Expl* (E)	17
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$	$\neg C \vee B$	*Expl* $(B \vee C)$	18
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$	B	*Expl* $(A \vee \neg B)$	19
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$	A	*Expl* $(\neg A)$	20
$DE \neg A \neg B\ C$	$A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E,\ \neg A$	$\bot$	*Fail*	21

Figure 3: Execution of the DPLL(T) algorithm on F

$$F : \{A \vee \neg B,\ \neg A \vee \neg B,\ B \vee C,\ D,\ E\}$$

A abstracts $f(g(z)) = h(y)$, $\neg A$ abstracts, $f(g(z)) \neq h(y)$, and so on.

An execution of the DPLL(T) algorithm from the previous section on F, from initial to final state is presented in Figure 3. The first and only decision is made in step 3 when $A \mapsto \top$ is assigned. In step 6, the current assignment is found to be conflicting with theory lemma $\neg C \vee \neg D \vee \neg E \vee B$, but since this conflict came in a context where a decision was made, the decision is reversed in step 12, after the conflict clause is simplified and learned. In step 15, when the same lemma is found to be conflicting with the new context, we can conclude that the input is unsatisfiable using the *Fail* rule, since we tried both choices of assignment for A. This is what the solver would typically do. However, the empty clause $\bot$ is trivially unsatisfiable and in some given context where $x \mapsto \top$ for some x, an acceptable proof of unsatisfiability is to to derive from the context that $x \mapsto \bot$ holds. Such a state where some x and $\neg x$ holds logically reduces to $\bot$ (x and $\neg x$ resolve to $\bot$). Thus, given a set of formulas, a more uniform proof of unsatisfiability of the set is to derive $\bot$ from it. This is why, in proof mode, the SMT solver, only applies the final *Fail* rule after reducing the conflict clause to $\bot$ using explanations. This is done from step 15 to 21. □

Refutation tree of F:

$$\frac{\text{Proof of } \neg A \qquad \text{Proof of } A}{\bot}\;20$$

where Proof of $\neg A$:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}}{\neg C \vee \neg D \vee \neg E \vee B} \qquad D}{\neg C \vee \neg E \vee B}\;7 \qquad E}{\neg C \vee B}\;8 \qquad B \vee C}{B}\;9 \qquad \neg A \vee \neg B}{\neg A}\;10$$

and Proof of A:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}}{\neg C \vee \neg D \vee \neg E \vee B} \qquad D}{\neg C \vee \neg E \vee B}\;16 \qquad E}{\neg C \vee B}\;17 \qquad B \vee C}{B}\;18 \qquad A \vee \neg B}{A}\;19$$

Figure 4: Proof tree for example in Figure 3. $\mathcal{P}$ is the proof of the theory lemma from the theory solver.

3.2 DPLL(T) Proofs

Given an execution of the DPLL(T) algorithm that ends in the $fail$ state, the solver proves that the input formula F is unsatisfiable by building a *refutation tree* of F — a tree with the input clauses and theory lemmas at the leaves, that lead, through the application of the resolution rule, to the empty clause $\bot$ at the root. A resolution proof can be extracted from a DPLL(T) execution by following the applications of the *Confl* and *Expl* rules. Notice, that the *Expl* rule modifies the current conflict clause by resolving it with an input clause, and it is precisely these resolutions that guide the creation of the proof tree.

Example 3.2 Figure 4 shows the proof tree of the unsatisfiability of F from Example 3.1. The nodes of the tree contain the clause being used. Every reduction of (a pair of) nodes to a child node is an application of the resolution rule. The label of the resolution of two nodes in the tree shows the step number of the application of the *Expl* from Figure 3. The left subtree is a proof of $\neg A$, which is learned in steps 1 to 11, and the right subtree is the proof of A which is concluded from steps 12 to 20. Step 21 performs the final resolution of A and $\neg A$ to derive $\bot$. $\mathcal{P}$ is the proof of the theory lemma that is expanded in the next subsection. □

Finally, the theory lemmas used in the proof appear at the leaf of the tree, and need to be justified by proofs of the lemmas from the theory solver. This is explained in the next subsection.

3.3 Theory Lemma Proofs

Theory lemmas at the leaves are justified by satellite proofs from the theory solver that produce them. They prove the lemma by contradiction — they derive the formula $\bot$ from the negation of

$$\dfrac{\boxed{\dfrac{\dfrac{\dfrac{\dfrac{f(m)=x}{x=f(m)}\,\mathit{Symm} \quad f(m)=f(n)}{x=f(n)}\,\mathit{Trans} \quad f(n)=y}{x=y}\,\mathit{Trans} \quad x\neq y}{\bot}\,\mathit{Contra}}}{f(m)\neq f(n)\vee f(m)\neq x\vee f(n)\neq y\vee x=y}\,\mathit{Lemma}$$

Figure 5: EUF Proof from Figure 4

clause C, and the *Lemma* rule derives C from this. A negation of a clause is a conjunction of each of the clause's disjuncts, separately negated. The premise of the *Lemma* rule is a subproof of $\bot$ from the complements of the disjuncts of the theory lemma, and from this it concludes the lemma.

Example 3.3 The only theory lemma used in our example is

$$\begin{aligned} L &: \neg C \vee \neg D \vee \neg E \vee B \\ \text{or, } L &: f(m) \neq f(n) \vee f(m) \neq x \vee f(n) \neq y \vee x = y \\ \text{equivalently, } L &: f(m) = f(n) \Rightarrow f(m) = x \Rightarrow f(n) = y \Rightarrow x = y \end{aligned}$$

Taking its negation gives:

$$\begin{aligned} \neg L &: C \wedge D \wedge E \wedge \neg B \\ \text{or, } \neg L &: f(m) = f(n) \wedge f(m) = x \wedge f(n) = y \wedge x \neq y \end{aligned}$$

Finally, the proof tree from the theory solver of L, or $\mathcal{P}$ from Figure 4 is shown in Figure 5. It derives $\bot$ at the root of a tree that contains the conjuncts of $\neg L$ as leaves — $C, D, E, \neg B$. It applies the *Lemma* to this subproof to derive L. The application of the *Lemma* rule is the unlabeled rule application from $\mathcal{P}$ to L in Figure 4, so $\mathcal{P}$ is technically the proof tree in the box. The proof uses the transitivity and symmetry of equality in addition to the rule that specifies the contradiction between an equality and its negation. □

4 Classical vs Constructive Logic in Coq

Coq is a proof assistant based on the Calculus of Inductive Constructions (CIC) [9], which is an extension of the dependent type theory — the Calculus of Constructions (CoC) [4] — with inductive types. Coq's type systems implements an infinite hierarchy of types, where all objects, including types, are terms that have types. "Higher-order" types (types of types) are distinguished from "first-order" types (types of terms that are not types), by referring to them as *kinds*. At the bottom of the type hierarchy, Coq has (1) regular first-order types such as `Bool` representing Booleans and `Nat` representing natural numbers, that are defined inductively and are themselves instances of the `Set` kind, and (2) propositions or properties in Coq that have kind `Prop`. The rest of the hierarchy looks like this: `Prop` and `Set` have kind $\texttt{Type}_1$ and $\texttt{Type}_i$ has kind $\texttt{Type}_{i+1}$.

The implementation of propositions as a kind in Coq is its means of enforcing the Curry-Howard correspondence between proofs and terms (or programs), and between propositions and types. As a consequence, propositions having the `Prop` kind in Coq, are only provable, if a (proof) term inhabiting the proposition can be constructed. Thus, a `Prop` P can be proved in Coq only if a proof term having type P can be constructed. This term can be constructed within a *proof script* in the Coq user-interface. Given a lemma, Coq provides a *goal* and a *proof context*, which contains the terms available to prove the goal. Initially, the goal is the entire proposition being proven, and the context is empty, and the user can manipulate the goal to break it down and move over its constituents into the context using Coq functions called *tactics*. Coq implements a constructive logic via its `Prop` type. Constructive or intuitionistic logic is a branch of mathematical logic studying arguments concerning constructive objects.

Example 4.1 The following Coq code proves the formula $a \Rightarrow a \vee b$, for any `Props` a and b.

```
Theorem orIntroL : forall (a b: Prop), a -> a \/ b.
Proof.
  intros a b Ha.
  left. apply Ha.
Qed.
```

The formula asserts that given a proof of the proposition a, we have a proof of the disjunction of a and b. The universal quantifier (`forall`) is a binder that introduces arbitrary propositions `a` and `b` into the lemma. The proof script between `Proof` and `Qed` constructs a proof term of type `forall (a b: Prop), a -> a \/ b`. The first line of the script introduces the propositions a and b and the proof Ha of a into the proof context (using the `intros` tactic). The only way to prove a disjunction is to prove one of the disjuncts. This is a crucial detail that distinguishes constructive logic from classical logic, as we will see later in the section. The second line of the proof script proves the left disjunct by using the proof of a from the context. $\square$

Classical logic extends constructive logic with the *law of the excluded middle* that states that for arbitrary proposition a, $a \vee \neg a$ holds. This can also be stated as the double negation elimination law or as proving a proposition by contradiction. As such, some statements that are true in classical logic might not have a proof in constructive logic. The law of the excluded middle itself is a convenient example. `forall (a : Prop), a ∨ ~a` is not provable in Coq, because, as mentioned before, the only way to prove a propositional disjunction is to prove that one of the disjuncts holds and unlike `orIntroL`, there is no way to prove either of the disjuncts here.

An SMT solver can prove classical formulas that include both $a \Rightarrow a \vee b$ and $a \vee \neg a$ for arbitrary Booleans a and b. As such, `Prop` is not an ideal candidate to embed SMT formulas into, since with such an embedding, SMT solvers would be able to prove propositions in Coq that are incompatible with Coq's constructive logic. In addition to the `Prop` type, Coq has a Boolean (`bool`) type in its standard library [1], that defines Booleans as an inductive type with two nullary constructors `true` and `false`. This Boolean type in Coq effectively implements a classical logic. Any Boolean value can be inspected, pattern matched on, and evaluated to either `true` or `false`. Terms of type `bool` cannot, by themselves, be stated as theorems in Coq, since they don't have the `Prop` type (they are not propositions). To state that a Boolean term B holds for arbitrary values of its variables, the term is equated to `true` and its variables are universally quantified.

$$F:\ Q,\ B = \texttt{true}$$

where Q represents the universal quantification of the variables in B. Since equality is a proposition, and quantified propositions are themselves propositions, F represents an embedding of a Boolean term into a proposition, which can now be used as a Coq goal. We call such goals that embed a Boolean term into a proposition *Boolean goals*.

Example 4.2 The law of excluded middle is stated in Coq as a Boolean goal and proved as follows.

```
Theorem LEMBool : forall a : bool, a || (negb a) = true.
Proof.
  intros a. destruct a.
  + simpl. reflexivity.
  + simpl. reflexivity.
Qed.
```

`orb` is the disjunction operator over Booleans and has infix notation `||`, and `negb` is the negation operator over Booleans. `destruct a` performs a case analysis on a, and since a is a Boolean that can only be either `true` or `false`, the two lines beginning with `+` replace the a in the context with exactly those values. When a is `true`, the goal reduces to `true || not true = true` and when it is `false`, it reduces to `false || not false = true`. Both of these are simplified to `true = true` using the tactic `simpl` and then by the reflexivity of `=`, both equalities are provable. Notice that, neither a proof of a, nor a proof of $\neg a$ was constructed here. Arguments were made in terms of every possible value a could take, and the conjecture was proven in each case. Such a proof, as we saw above, is impossible in constructive logic. □

4.1 The Role of SMTCoq

SMTCoq is a plugin that adds automation to Coq by dispatching goals to SMT solvers. Since SMT solvers implement a classical logic, SMTCoq uses SMT solver proofs to prove classical formulas expressed in Coq as Booleans goals.

Example 4.3 The `LEMBool` lemma from above would be a simple goal that could be dispatched to an SMT solver using SMTCoq's `smt` tactic:

```
Theorem LEMBool' : forall a : bool, a || (negb a) = true.
Proof.
  smt.
Qed.
```

□

SMTCoq embeds SMT formulas in Coq as Boolean goals in what is called a *shallow embedding*, which uses types in Coq that correspond to SMT formulas, to embed SMT formulas in Coq. As such, SMT formula semantics are already defined in Coq. A proof of a formula from an SMT solver is translated to the proof of a Boolean goal in Coq via a *deep embedding*. SMTCoq defines

- the deep embedding of SMT formulas in Coq as the `form` type,
- an interpretation of `form`s with respect to `Bool`s,

- a checker for proofs of (unsatisfiability of the negation) the deep embedding of formulas coming from the SMT solver, and,
- a proof of correctness of the checker, also called the *reflection lemma*, that *reflects* a proof of the deep embedding of a formula from the SMT solver (successfully checked by the checker) to the proof of a Boolean goal in Coq (the shallow embedding of formulas).

This process is described in further detail in Report 2. Figure 6 and Figure 7 reproduce parts of the figure from Report 2 illustrating this process, although they do it in the context of explaining SMTCoq's `Prop` to `Bool` conversion, which is discussed in the next subsection.

4.2 Theories and Prop Predicates

SMTCoq integrates both SAT and SMT solvers with Coq. If it were reasoning about purely Boolean goals such as `LEMBool` above, it wouldn't need SMT solvers at all. It could do that with just SAT, or propositional satisfiability, solvers. SMT, or satisfiability modulo theories, solvers provide the ability to reason in theories such as finite- and infinite-precision integers and arrays. Using its SMT solver integration, SMTCoq is able to prove more expressive Boolean goals over the theories of uninterpreted functions, linear integer arithmetic, arrays, and bit-vectors. Recall that atomic formulas are applications of predicate symbols to terms. Earlier in the section, we looked at logical connectives being represented classically (as connectives over the `Bool` type) or constructively (as `Prop` connectives).

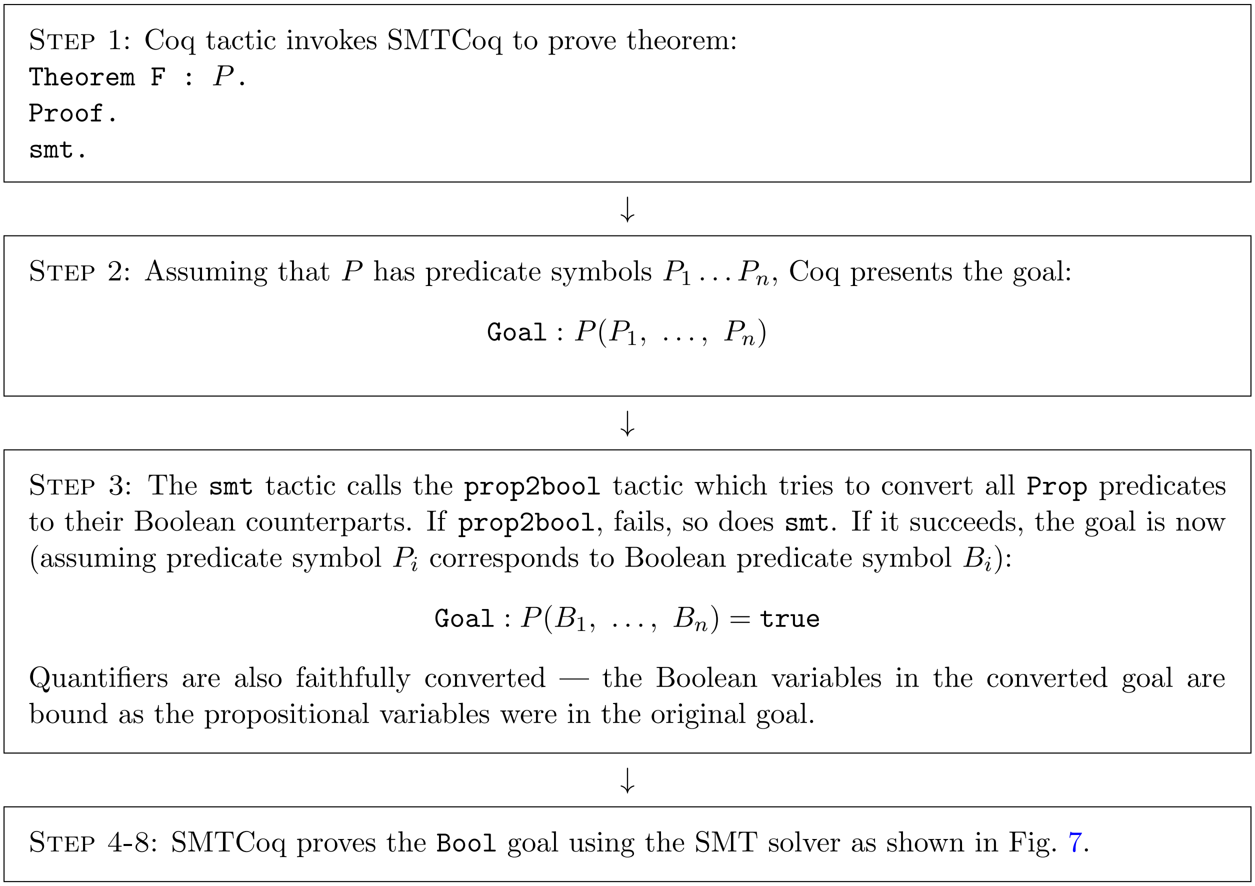
Similarly, in this section, we will refer to predicate symbols that construct `Props` as `Prop` predicate symbols, and those that construct `Bools` as `Bool` (or Boolean) predicate symbols, in Coq. A `Prop` predicate is a proposition in Coq, parameterized by values (of any type, but we will focus on the types in the theories supported by SMTCoq.). A `Bool` predicate is a Boolean term, parameterized by values. To state that a `Bool` term holds, it's free variables must be quantified and the term equated to `true`, as demonstrated in the previous section for connectives.

Example 4.4 Consider the following examples of `Bool` predicates over integers and bit-vectors.

```
Theorem Z_congBool : forall (a b : Z) (f : Z -> Z),
  implb (Z.eqb a b) (Z.eqb (f a) (f b)) = true.
Proof.
  smt.
Qed.

Theorem BV_ulteZeroBool : forall (b : bitvector 4),
  (bv_ult #b|0|0|0|0| b) || (bv_eq b #b|0|0|0|0|) = true.
Proof.
  smt.
Qed.
```

`Z_congBool` proves the congruence rule over integer to integer functions and `BV_ulteZeroBool` proves that all 4-bit integers (bit-vectors of length 4) in their unsigned interpretation are greater than or equal to 0. The operator `implb` is the implication operator over Boolean terms, `Z.eqb` is the Boolean equality predicate symbol over integers in Coq, and `bv_ult` and `bv_eq` are respectively the Boolean less than (over unsigned integers) and equality predicate symbols for bit-vectors. □



Step 1: Coq tactic invokes SMTCoq to prove theorem:

```
Theorem F : P.
Proof.
smt.
```

↓

Step 2: Assuming that P has predicate symbols $P_1 \dots P_n$, Coq presents the goal:

$$\texttt{Goal} : P(P_1,\ \dots,\ P_n)$$

↓

Step 3: The `smt` tactic calls the `prop2bool` tactic which tries to convert all `Prop` predicates to their Boolean counterparts. If `prop2bool`, fails, so does `smt`. If it succeeds, the goal is now (assuming predicate symbol P_i corresponds to Boolean predicate symbol B_i):

$$\texttt{Goal} : P(B_1,\ \dots,\ B_n) = \texttt{true}$$

Quantifiers are also faithfully converted — the Boolean variables in the converted goal are bound as the propositional variables were in the original goal.

↓

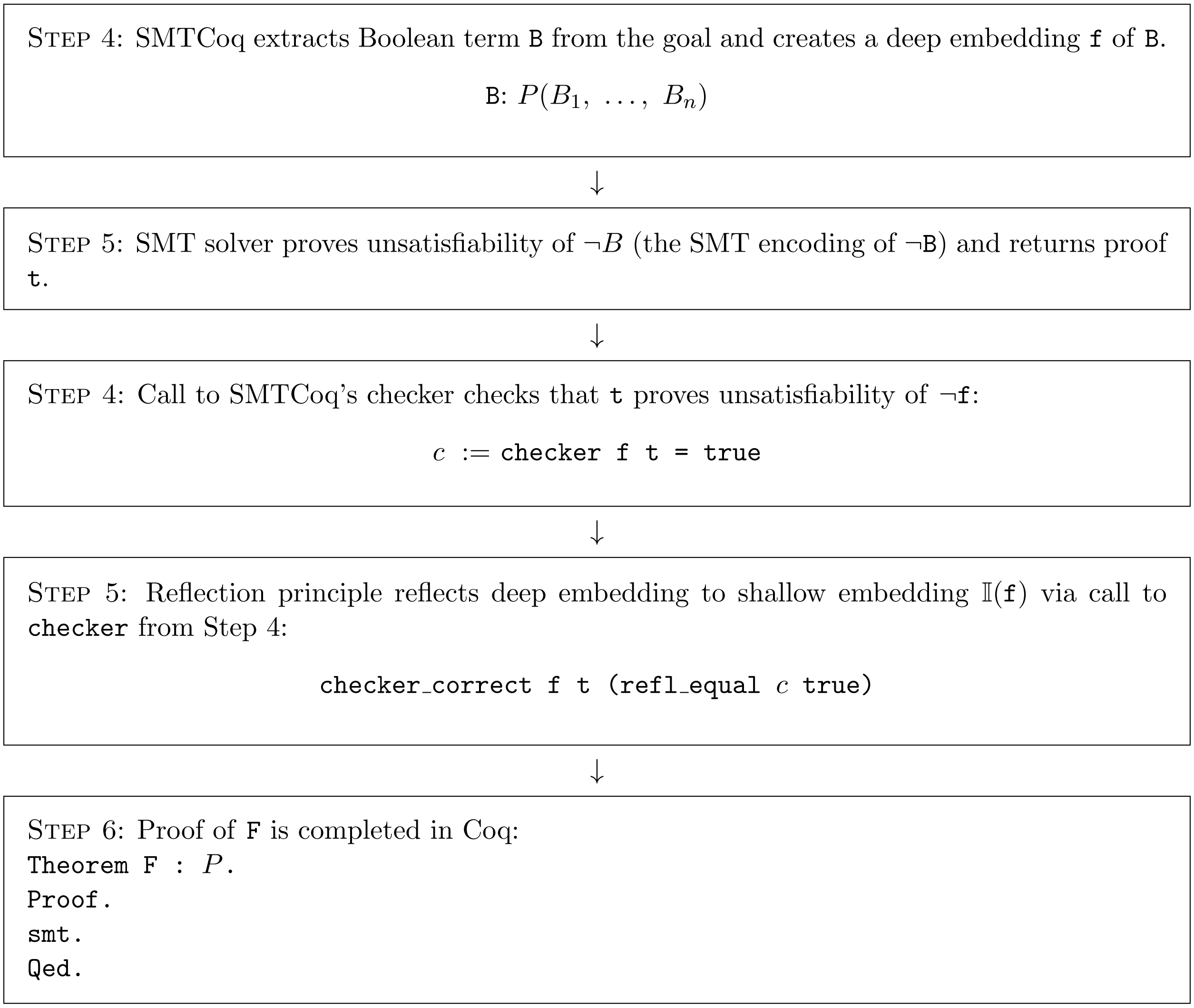
Step 4-8: SMTCoq proves the `Bool` goal using the SMT solver as shown in Fig. 7.

Figure 6: SMTCoq's journey in proving example Boolean formula `F`

SMTCoq is able to prove formulas over some `Prop` predicates if they are reducible to their corresponding `Bool` predicates. Such a reduction is specified using lemmas of equivalence between the two. SMTCoq defines the `prop2bool` tactic that converts a `Prop` predicate symbol to its corresponding `bool` counterpart, by applying the equivalence lemma, if one is available. The `prop2bool` tactic only succeeds on a goal if it is able to recursively convert all the predicate symbols in the goal. When a user asks SMTCoq to prove a goal consisting of `Prop` predicate symbols, the `smt` tactic first calls the `prop2bool` tactic to convert all `Prop` predicate symbols to their corresponding `bool` predicate symbols. The `smt` tactic then uses the SMT solver to prove the resulting Boolean goal (using the checker and the reflection lemma). If it is successful, the theorem can be closed. This process is described by adding the steps in Figure 6 to the ones in Figure 7, already presented in Report 2.

Example 4.5 Conveniently, the `prop2bool` tactic is successful with the predicate symbols used in Example 4.4. So, SMTCoq can prove their corresponding `Prop` theorems:

```
Theorem Z_congProp : forall (a b : Z) (f : Z -> Z), a = b ->
  f a = f b.
Proof.
```



Step 4: SMTCoq extracts Boolean term B from the goal and creates a deep embedding f of B.

B: $P(B_1,\ \ldots,\ B_n)$

↓

Step 5: SMT solver proves unsatisfiability of $\neg B$ (the SMT encoding of ¬B) and returns proof t.

↓

Step 4: Call to SMTCoq's checker checks that t proves unsatisfiability of ¬f:

c := checker f t = true

↓

Step 5: Reflection principle reflects deep embedding to shallow embedding $\mathbb{I}$(f) via call to checker from Step 4:

checker_correct f t (refl_equal c true)

↓

Step 6: Proof of F is completed in Coq:

```
Theorem F : P.
Proof.
smt.
Qed.
```

Figure 7: SMTCoq's journey in proving example Boolean formula F

```
  smt.
Qed.

Theorem BV_ulteZeroProp : forall (b : bitvector 4),
  bv_ultP #b|0|0|0|0| b \/ b = #b|0|0|0|0|.
Proof.
  smt.
Qed.
```

□

Depending on the predicate symbol, some of the equivalence lemmas between the `Prop` and `bool` versions are either in Coq's libraries, or proved in SMTCoq's libraries.

Example 4.6 SMTCoq's bitvector library proves the following lemma of equivalence between the `Bool` and `Prop` version of the unsigned less than predicate over bit-vectors. This equivalence lemma is used by SMTCoq's `prop2bool` tactic when a user uses the `Prop` version of this predicate symbol in a query to SMTCoq, as in `BV_ulteZeroProp` from Example 4.5.

```
Theorem bv_ult_B2P : forall n (a b : bitvector n),
  bv_ult a b = true <-> bv_ultP a b.
```

An example of an equivalence proved in a Coq library is that of the integer less than predicate.

```
Lemma ltb_lt (n m : Z) : (n <? m) = true <-> n < m.
```

□

References

[1] The Coq Bool Library. `https://coq.inria.fr/library/Coq.Bool.Bool.html`.

[2] C. Barrett and C. Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[3] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[4] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2–3):95–120, Feb. 1988.

[5] G. Katz, C. W. Barrett, C. Tinelli, A. Reynolds, and L. Hadarean. Lazy proofs for dpll(t)-based SMT solvers. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 93–100, 2016.

[6] C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers. (Question de confiance : communication sceptique entre Coq et des prouveurs externes)*. PhD thesis, École Polytechnique, Palaiseau, France, 2013.

[7] D. Loveland, A. Sabharwal, and B. Selman. *DPLL: The Core of Modern Satisfiability Solvers*, pages 315–335. Springer International Publishing, Cham, 2016.

[8] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[9] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In B. W. Paleo and D. Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, Jan. 2015.