Itauto: An Extensible Intuitionistic SAT Solver

Frédéric Besson
Inria, Univ Rennes, Irisa, Rennes, France

Abstract

We present the design and implementation of `itauto`, a Coq reflexive tactic for intuitionistic propositional logic. The tactic inherits features found in modern SAT solvers: definitional conjunctive normal form; lazy unit propagation and conflict driven backjumping. Formulae are hash-consed using native integers thus enabling a fast equality test and a pervasive use of Patricia Trees. We also propose a hybrid proof by reflection scheme whereby the extracted solver calls user-defined tactics on the leaves of the propositional proof search thus enabling theory reasoning and the generation of conflict clauses. The solver has decent efficiency and is more scalable than existing tactics on synthetic benchmarks and preliminary experiments are encouraging for existing developments.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Computing methodologies → Theorem proving algorithms

Keywords and phrases SAT solver, proof by reflection

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.9

Supplementary Material *Software*: `https://gitlab.inria.fr/fbesson/itauto`
archived at `swh:1:dir:986dc28c9844ba2faedc3d134b2acfb31e8c27c4`

Acknowledgements Thanks are due to Alix Trieu for sharing with us his verified Patricia Tree library. Thanks are also due to Samuel Gruetter for stress testing the tactic; his feedbacks have been very valuable.

1 Introduction

Using an ideal proof-assistant, proofs would be written at high-level and mundane proof tasks would be discharged by automated procedures. However, automated reasoning is hard; even more so for *sceptical* [17] proof-assistants: generating and verifying proofs at scale, even for decidable logic fragments, is a challenge requiring sophisticated implementation strategies [7, 4, 2]. For the Coq proof-assistant, the situation is made slightly worse because intuitionistic logic is not mainstream for automated provers. Thus, the Satisfiability Modulo Theory (SMT) approach that is based on a classical SAT solver needs to be revisited.

1.1 Propositional Reasoning and Theory Reasoning in Coq

There are a variety of Coq tactics which perform intuitionistic propositional and theory reasoning. We describe here their main features and explain some limitations, thus motivating the need for a novel (extensible) solver for intuitionistic propositional logic (IPL). We defer to Section 6 the discussion of related approaches rooted in a classical setting and interfacing with external provers.

`tauto` [1] is a complete decision procedure for IPL based on the LJT* calculus [14]. `rtauto` [2] is another decision procedure for IPL verifying proof certificates using proof by reflection. These decision procedures are usually efficient enough for interactive use but do not perform theory reasoning. `lia`[3] is a decision procedure for linear arithmetic. It has a classical

[1] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.tauto`
[2] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.rtauto`
[3] `https://coq.inria.fr/refman/addendum/micromega.html#coq:tacn.lia`

understanding of propositional connectives but abstracts away non-arithmetic propositions. `congruence`[4] [11] does not perform propositional reasoning but decides the theory of equality with constructors. Another tactic that is worth mentioning is `intuition tac`[5]; it first performs propositional reasoning and calls the leaf tactic `tac` when it gets stuck. (`tauto` is actually `intuition fail`.)

In a classical setting, calling a theory solver on the leaves of a propositional proof search is the basis of the $DPLL(T)$ algorithm [16]. Unfortunately, in an intuitionistic setting, completeness is lost. Example 1 illustrates the issue.

▶ **Example 1** (Incomplete Combination)**.** Consider the following goals.

```
Lemma Ex1:∀(p:Prop)(x:Z), x=0 → (x>=0 → p) → p.
Lemma Ex2:∀(A:Type)(p:Prop)(a b c:A),a=b → b=c → (a=c → p) → p.
```

For `Ex1` and `Ex2`, `intuition lia` (resp. `intuition congruence`) fail whereas the goal can be decided using a combination of propositional logic with the theory of linear arithmetic and the theory of equality, respectively. The reason is that the `intuition` part does nothing and therefore calls the leaf tactic on the unmodified goal. For `Ex1`, `lia` abstracts away non-arithmetic propositions[6] and is left with the non-theorem `x=0→(x>=0→True)→False`. For `Ex2`, `congruence` abstracts away propositions and is also left with a non-theorem: `a=b→b=c→False`.

Besides completeness, `intuition tac` may also call the leaf tactic `tac` more often than necessary. This is illustrated by Example 2.

▶ **Example 2** (Spurious Leaf Tactic Call)**.** Consider the following goal.

```
Lemma Ex3:∀(A:Type)(x y t:A)(p q:Prop),x=y→p∨q→(p→y=t)→(q→y=t)→x=t.
```

In this case, `intuition congruence` performs a case-split over `p\/q`, and derives `y=t` by *modus-ponens*. Eventually, it calls the leaf tactic `congruence` twice. However, in both branches, `congruence` solves the same theory problem *i.e.*, `x=y → y=t → x=t`.

1.2 Contribution

A first contribution is the design and implementation of a reflexive intuitionistic SAT solver in Coq. The SAT solver obtains decent performances using features found in state-of-the-art SAT solvers such as hash-consing, lazy unit propagation, backjumping and theory learning. It also makes a pervasive use of native machine integers[7] and Patricia Trees [22].

Another contribution is a variation of the proof by reflection approach whereby the verified extracted SAT solver is first run inside the proof engine. Theory reasoning is then performs by calling user-defined tactics on the leaves of the propositional proof search. Following the $DPLL(T)$ approach, minimal conflict clauses obtained from the tactic proof-terms are passed back to the SAT solver. Eventually, a proof is obtained by asserting the conflict clauses and re-running the reflexive SAT solver with an empty theory module. To our knowledge, this design combining reflexive code with tactics is original.

---

[4] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.congruence`
[5] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.intuition`
[6] The propositional variable `p` is replaced by `True` or `False` depending on its polarity.
[7] `https://coq.inria.fr/refman/language/core/primitive.html#primitive-integers`

The rest of the paper is organised as follows. In Section 2, we present the main features of our SAT solver and its implementation in Coq. The structure of the soundness proof is detailed in Section 3. In Section 4, we explain how to interface the SAT solver with the proof engine and user-defined tactics. We show experimental results in Section 5. Related work is presented in Section 6 and Section 7 concludes.

Note that the code snippets in the paper have been edited and idealised for clarity.

2 Design of an Intuitionistic SAT solver

Our algorithm is reusing several key components of modern SAT solvers. Fortunately, they are only slightly adapted to the intuitionistic setting.

2.1 Syntax and Semantics of Formulae

Our SAT solver takes as input hash-consed [1] formulae defined by the inductive type `LForm`.

```
Inductive LForm : Type :=
| LFF | LAT : int → LForm | LOP : lop → list (HCons.t LForm) → LForm
| LIMPL : list (HCons.t LForm) →  (HCons.t LForm)  → LForm.
```

The syntax is mostly standard and models n-ary propositional operators. `LFF` represents the proposition `False`. `LAT` i represents the propositional variable $\mathtt{p}_i$. `LOP o` $[f_1; \ldots; f_n]$ where `o` $\in\{$`AND`, `OR`$\}$ represents a n-ary conjunction or disjunction. `LIMPL` $[f_1; \ldots; f_n]$ f represents a n-ary implication with f_1, …, f_n being the premisses and f the conclusion. The negation of a formula f is encoded by `LIMPL` $[f]$ `LFF`. All sub-formulae are hash-consed *i.e.* `f:HCons.t LForm` is a pair made of a formula and a unique index. Because it enables a fast equality test of formulae in $O(1)$, hash-consing is essential for efficiency. N-ary operators allow for a sparser conjunctive normal form (see *e.g.* [20]).

The interpretation of a formula `f:LFORM` is given by structural recursion with respect to an environment `e:int` → `Prop` mapping indexes *i.e.*, propositional variables, to propositions.

$$
\begin{array}{lcl}
[\![\mathtt{LFF}]\!]_e & = & \mathtt{False} \\
[\![\mathtt{LAT}\ i]\!]_e & = & e\ i \\
[\![\mathtt{LOP\ AND}\ [f_1; \ldots; f_n]]\!]_e & = & [\![f_1]\!]_e \wedge \cdots \wedge [\![f_n]\!]_e \\
[\![\mathtt{LOP\ OR}\ [f_1; \ldots; f_n]]\!]_e & = & [\![f_1]\!]_e \vee \cdots \vee [\![f_n]\!]_e \\
[\![\mathtt{LIMPL}\ [f_1; \ldots; f_n]\ f]\!]_e & = & [\![f_1]\!]_e \to \cdots \to [\![f_n]\!]_e \to [\![f]\!]_e
\end{array}
$$

In the following, as the environment e is fixed, we drop the subscript and write $[\![f]\!]$ for $[\![f]\!]_e$.

2.2 Intuitionistic Clausal Form

In a classical setting, to prove that a formula f is a tautology, a modern SAT solver proves that the conjunctive normal form (CNF) of the negation of f is unsatisfiable. In other words, a SAT solver exploits the fact that $\neg(\neg f)$ is equivalent to f and that the CNF conversion preserves provability. In an intuitionistic setting, this approach is not feasible because *reductio ad adsurdum* is not logically sound and the usual CNF conversion requires De Morgan laws which are not admissible. Yet, Claessen and Rosén [10] show that it is possible to transform an intuitionistic formula f into an equi-provable formula of the form $\bigwedge F \wedge \bigwedge I \to q$ where q is a variable ; $f \in F$ is a so-called *flat clause* of the form $p_1 \to \ldots p_n \to q_1 \vee \cdots \vee q_m$ where the p_i and q_i are variables and $i \in I$ is a so-called *implication clause* of the form $(a \to b) \to c$ where a, b and c are variables. The transformation is based on Tseitin definitional CNF [26]

with the modification that a clause is written $p_1 \to \cdots \to p_n \to q_1 \vee \cdots \vee q_n$ instead of $\neg p_1 \vee \cdots \vee \neg p_n \vee q_1 \cdots \vee q_n$. The *implication clauses* are reminiscent of the fact that double arrows (*e.g.*, in the LJT proof system) need a treatment that is specific to intuitionistic logic.

Our implementation is along these lines but optimises the transformation in order to reduce both the set of *flat clauses* and *implication clauses*. To reduce the set of *flat clauses*, our definitional CNF is based on Plaisted and Greenbaum CNF [24] which exploits the polarity of formulae and uses memoization thus avoiding recomputing the CNF of identical sub-formulae. In order to reduce the set of *implication clauses*, we exploit the fact that, if the conclusion q is decidable, intuitionistic logic reduces to classical logic. In that case, it is therefore admissible to replace an *implication clause* $(a \to b) \to c$ by the equivalent *flat clauses* $\{a \vee c; b \to c\}$. Moreover, before performing CNF conversion, formulae are flattened using the associativity of $\wedge$ and $\vee$. This has the advantage of augmenting the arity of the operators, reducing the depth of the formulae, and therefore reducing the number of intermediate propositional variables.

2.2.1 Pre-processing

The input formula is only using binary operators. To obtain n-ary operators, we recursively apply the following equivalences (the operator $+\!\!+$ is the concatenation of lists):

$$\begin{array}{lcl}
\texttt{LOP OR}\ (\texttt{LOP OR}\ l_1) :: l_2 & = & \texttt{LOP OR}\ (l_1 +\!\!+ l_2) \\
\texttt{LOP AND}\ (\texttt{LOP AND}\ l_1) :: l_2 & = & \texttt{LOP AND}\ (l_1 +\!\!+ l_2) \\
\texttt{LIMPL}\ (\texttt{LOP AND}\ l_1) :: l_2\ r & = & \texttt{LIMPL}\ (l_1 +\!\!+ l_2)\ r \\
\texttt{LIMPL}\ l_1 (\texttt{LIMPL} l_2\ r) & = & \texttt{LIMPL}\ (l_1 +\!\!+ l_2)\ r
\end{array}$$

2.2.2 Literals

Tseitin style CNF [26] consists in introducing fresh propositional variables. In a proof-assistant, modelling freshness may incur some proof overhead. Fortunately, in our case, the new propositional variables are not arbitrary but correspond to sub-formulae. As a result, we represent a propositional variable by a hash-consed formula (`HFormula` = `HCons.t LForm`) and a literal is a positive or negative hash-consed formula.

```
Inductive literal : Type := | POS (f: HFormula) | NEG (f: HFormula)
```

As usual, a clause is a list of positive and negative literals but with the following interpretation.

$$[\![\texttt{NEG}f :: l]\!]_e = [\![f]\!]_e \to [\![l]\!]_e \qquad [\![\texttt{POS}f :: l]\!]_e = [\![f]\!]_e \vee [\![l]\!]_e \qquad [\![[]]\!]_e = \texttt{False}$$

For readability, we will write $\lfloor f \rfloor$ for $\texttt{POS}f$ and $\lfloor f \rfloor \to$ for $\texttt{NEG}f$. A singleton clause $[\texttt{NEG}f]$ will be written $\lfloor f \rfloor \to \bot$. In the remaining, we have the invariant that the negative literals are always before the positive literals. As a result, a general clause $[\texttt{NEG}f_1; \ldots; \texttt{NEG}f_i; \texttt{POS}g_1; \ldots; \texttt{POS}g_j]$ will be written $\lfloor f_1 \rfloor \to \cdots \to \lfloor f_i \rfloor \to \lfloor g_1 \rfloor \vee \cdots \vee \lfloor g_j \rfloor$ or more compactly $\bigwedge_i \lfloor f_i \rfloor \to \bigvee_j \lfloor g_j \rfloor$. Moreover, we define the negation of a literal l: $\neg l$ is such that $\neg \lfloor f \rfloor = \lfloor f \rfloor \to$ and $\neg \lfloor f \to \rfloor = \lfloor f \rfloor$.

2.2.3 Introduction Rule

Before running the CNF conversion *per se*, we inspect the formula and perform *reductio ad adsurdum* if possible. The function `intro_impl` performs the introduction rule for implication with the twist that the conclusion is double negated if it is decidable. Therefore, `intro_impl: HFormula → (list literal * HFormula)` takes as input a hash-consed formula and returns a pair (l, c) where l are hypotheses and c is the conclusion.

```
intro_impl (LIMPL  l r) = if is_dec r then ((NEG r):: map POS l , HLFF)
                                      else (map POS l, r)
intro_impl f    = if is_dec f then ([NEG f] , HLFF) else ([], f)
```

The constant `HLFF` is the hash-consed formula `LFF` *i.e.*, the syntax for the proposition `False`. The `is_dec` predicate is implemented by a boolean flag stored together with the hash-cons of the formula. It is recursively propagated from atomic propositions that are known to be classical *i.e.*, we have $r \vee \neg r$.

2.2.4 Construction of the CNF

The next step consists in computing the CNF of the literals and of the formula in the conclusion. We recursively compute for a positive literal `CNF-` *i.e.*, a set of clauses modelling the elimination rule; and for a negative literal (or the conclusion) `CNF+` *i.e.*, a set of clauses modelling the introduction rule.

$$\frac{f = (f_1 \wedge \cdots \wedge f_n)}{\begin{array}{l}\texttt{AND-}(f) = \{\lfloor f\rfloor \to \lfloor f_1\rfloor; \ldots; \lfloor f\rfloor \to \lfloor f_n\rfloor\} \\ \texttt{AND+}(f) = \{\lfloor f_1\rfloor \to \cdots \to \lfloor f_n\rfloor \to \lfloor f\rfloor\}\end{array}} \qquad \frac{f = (f_1 \vee \cdots \vee f_n)}{\begin{array}{l}\texttt{OR-}(f) = \{\lfloor f\rfloor \to \lfloor f_1\rfloor \vee \cdots \vee \lfloor f_n\rfloor\} \\ \texttt{OR+}(f) = \{\lfloor f_1\rfloor \to \lfloor f\rfloor; \ldots; \lfloor f_n\rfloor \to \lfloor f\rfloor\}\end{array}}$$

$$\frac{f = (f_1 \to \cdots \to f_n \to r)}{\begin{array}{l}\texttt{IMPL-}(f) = \{\lfloor f\rfloor \to \lfloor f_1\rfloor \to \cdots \to \lfloor f_n\rfloor \to \lfloor r\rfloor\} \\ \texttt{IMPL+}(f) = \{\lfloor r\rfloor \to \lfloor f\rfloor\} \cup \bigcup_{\texttt{is\_dec } f_i} \{\lfloor f_i\rfloor \vee \lfloor f\rfloor\}\end{array}}$$

This clausal encoding is correct and the generated clauses are both classical and intuitionistic tautologies. Except for `IMPL+`, the clausal encoding is also complete *e.g.*, we have

$$\begin{array}{lcl}\texttt{CNF+}(f_1 \wedge \cdots \wedge f_n) \cup \{\lfloor f_1\rfloor \wedge \cdots \wedge \lfloor f_n\rfloor\} & \vdash & \lfloor f_1 \wedge \cdots \wedge f_n\rfloor \\ \texttt{CNF-}(f_1 \wedge \cdots \wedge f_n) \cup \{\lfloor f_1 \wedge \cdots \wedge f_n\rfloor\} & \vdash & \lfloor f_1\rfloor \wedge \cdots \wedge \lfloor f_n\rfloor\end{array}$$

For `IMPL+`, if not all the f_i are classical propositions, the clausal encoding is partial and we also keep the *implication clause* $(\lfloor f_1\rfloor \to \cdots \to \lfloor r\rfloor) \to \lfloor f\rfloor$ for later processing. An exception is when we intend to prove `False`. In that case, the `IMPL+` rule may drop the requirement on the decidability of the f_i thus providing a complete clausal encoding. This is sound because for any $f = (f_1 \to \cdots \to f_n \to r)$, $(\lfloor r\rfloor \to \lfloor f\rfloor) \wedge \bigwedge_{f_i}(\lfloor f_i\rfloor \vee \lfloor f\rfloor) \to$ `False` is an intuitionistic tautology.

▶ **Example 3.** Suppose that we intend to prove the tautology $(a \to (b \wedge c)) \to (b \vee (a \to c))$ where a, b and c are intuitionistic propositional variables. After running the introduction rule, we obtain the following hypothesis and conclusion

$$\{\lfloor a \to (b \wedge c)\rfloor\} \vdash b \vee (a \to c)$$

We compute `CNF-` for the hypothesis *i.e.* `IMPL-`$(\lfloor a \to (b \wedge c)\rfloor)$, and `CNF+` for the conclusion *i.e.*, `OR+`$(\lfloor d \vee (a \to c)\rfloor)$. Recursively, we compute `AND-`$(b \wedge c)$ and `IMPL+`$(a \to c)$ and eventually obtain

$$\left\{\begin{array}{l}\lfloor a \to (b \wedge c)\rfloor, \lfloor a \to (b \wedge c)\rfloor \to \lfloor a\rfloor \to \lfloor b \wedge c\rfloor, \\ \lfloor b \wedge c\rfloor \to \lfloor b\rfloor, \lfloor b \wedge c\rfloor \to \lfloor c\rfloor, \\ \lfloor b\rfloor \to \lfloor b \vee (a \to c)\rfloor, \lfloor a \to c\rfloor \to \lfloor b \vee (a \to c)\rfloor \\ \lfloor c\rfloor \to \lfloor a \to c\rfloor, (\lfloor a\rfloor \to \lfloor c\rfloor) \to \lfloor a \to c\rfloor\end{array}\right\} \vdash b \vee (a \to c)$$

Note that because a is an intuitionistic proposition, we keep the *implication clause* $(\lfloor a\rfloor \to \lfloor c\rfloor) \to \lfloor a \to c\rfloor$.

2.3 Lazy Unit Propagation

Once a clause is reduced to a single literal, say l, *unit propagation* simplifies all the remaining clauses using l. There are three cases to consider. If a clause c does not mention l, it is left unchanged. If the literal l belongs to the clause c, the clause is redundant and it is removed (see `R1` and `R2`). If the negation of the literal l belongs to the clause, we deduce the simplified clause $c \setminus \neg l$ (see `M1` and `M2`). In logic terms, we have the following inferences:

$$\texttt{R1}\,\frac{p}{p \vee r} \qquad \texttt{R2}\,\frac{p \to \bot}{p \to r} \qquad \texttt{M1}\,\frac{p \quad p \to r}{r} \qquad \texttt{M2}\,\frac{p \to \bot \quad p \vee r}{r}$$

In the following, a literal l is said to be *watched* if neither $[l]$ nor $[\neg l]$ is a unit clause. We also say that a literal l is *assigned* if it is not *watched*.

A naive unit propagation algorithm linearly traverses every clause and is therefore inefficient. A key observation is that the purpose of unit propagation is to produce new unit clauses. Said otherwise, it not necessary to traverse clauses where at least 2 literals are watched. As a result, a clause that is neither the empty clause nor the unit clause is represented by the type `watched_clause` given below.

```
Record watched_clause:={
  watch1 : literal;  watch2 : literal; unwatched : list literal }.
```

This is purely functional variant of so called head/tail lists[8] [28, 29] that is simpler than the 2-watched literals optimisation [21]. Watched clauses are indexed on their watched literals but also on whether the watched literals are positive or negative. To get an efficient representation of sets of clauses, each clause is given a unique identifier and is stored in a Patricia Tree [22]. In order to implement so-called *non-chronological backtracking* (see Section 2.4), we also track the set of literals that are needed to deduce a clause. As a result, each clause is annotated with a set of literals `LitSet.t`.

```
  Record Annot (A: Type) := { elt := A ; deps := LitSet.t }
```

Therefore, unit propagation operates on the following `watch_map` data-structure.

```
Definition clause_set := ptrie (Annot watched_clause).
Definition watch_map  := ptrie (clause_set * clause_set).
```

A `watch_map` m maps a propositional variable f *i.e.*, a hash-consed formula, to two sets (n, s) of clauses where clauses in n are watched by $\lfloor f \rfloor \to$ and clauses in p are watched by $\lfloor f \rfloor$. More precisely, we have

$$\begin{array}{lll} c \in n & \textit{iff} & c.watch1 = (\lfloor f \rfloor \to) \vee (c.watch2 = \lfloor f \rfloor \to) \\ c \in p & \textit{iff} & c.watch1 = \lfloor f \rfloor \vee c.watch2 = \lfloor f \rfloor \end{array}$$

Suppose that we perform unit propagation for the literal $\lfloor f \rfloor$ for a `watched_map` m such that $m[f] = (n, p)$. The clauses in p are redundant and can be dropped; the clauses in n need to be processed and reduced. The reduction takes as argument the set of literals that are currently assigned, a watched clause c and returns an annotated clause `Annot clause` where the type `clause` is given below

```
Inductive clause:=|EMPTY |TRUE |UNIT (l:literal)|CLAUSE (wc:watched_clause)
```

[8] Head/tail lists give access to the first and last element of a list in $O(1)$.

`EMPTY` represents the *empty* clause, *i.e.* a contradiction. In that case, unit propagation concludes the proof. `TRUE` represents a redundant clause that will be dropped. `UNIT l` is a unit clause to be propagated and `CLAUSE wc` is a watched clause.

Without loss of generality, suppose that $\lfloor f \rfloor$ belongs to the set of assigned literals `s` and that we have a clause c of the form:

$$c = \{|\texttt{watch1} := \neg\lfloor f \rfloor; \texttt{watch2} := x; \texttt{unwatched} := [y_1; \ldots; y_n;]|\}$$

The `reduce` function takes as input the watched literal x and aims at finding in the list $[y_1; y_n; \ldots; y_n]$ a watched literal y_i (*i.e.*, y_i is not assigned in `s`) and return as clause the rest of the list.

```
reduce s d x [] = {| elt := UNIT x ; deps := d |}
reduce s d x (y::l) = {| elt := TRUE ; deps := d |} when y ∈ s
reduce s d x (y::l) = reduce s ((deps (s y)) ∪ d) l when ¬ y ∈ s
reduce s d x (y::l) =
  let wc := {| watch1 := x ; watch2 := y ; unwatched := l |} in
  {| elt := CLAUSE wc ; deps := d |} when y ∉ s
```

If the list is empty, we produce the unit clause `UNIT` x and unit propagation will be recursively called for x. Let y be the head of the list l. If y is not assigned in s, we return a novel clause where the watched literals are x and y. If y is already assigned with the same polarity in s ($y \in s$), the clause is redundant and can be dropped. If y is already assigned but with opposite polarity ($\neg y \in$ `s`), the reduction is recursively called threading along the dependencies of the literal y.

▶ **Example 4.** Suppose that the set of assigned literals is given by $\texttt{s} = \{f; \neg y_1; \neg y_n\}$ and we perform unit propagation over the clause c. By construction, we know that neither x nor $\neg x$ are assigned in `s`. To get a well-defined watched clause with 2 watched literals, we need to find a replacement for the literal $\neg\lfloor f \rfloor$ in the list $[y_1; y_2; \ldots; y_n]$. As $\{\neg y_1, \neg y_n\} \subseteq \texttt{s}$, a greedy unit propagation would deduce that y_1 and y_n can be removed but as the prohibitive cost of a linear scan of the whole clause. The idea of lazy unit propagation is to stop at the first unassigned literal *i.e.* y_2. Therefore, we generate the watched clause

$$\{\texttt{watche1} := x; \texttt{watche2} := y_2; \texttt{unwatched} := [y_3; \ldots; y_n]\}$$

2.4 Case Splitting and Backjumping

When all the unit clauses are propagated, the solver performs a case-split over a clause. The clause needs to represent a disjunction. If the conclusion is `False`, any clause may be selected. Otherwise, the clause may only contain either positive literals or negative *classical* literals. The soundness of this argument is expressed by the following inferences.

$$\frac{\bigwedge_i \Gamma \cup (p_i \to \bot) \vdash \bot \quad \bigwedge_j \Gamma \cup q_j \vdash \bot}{\Gamma \cup \bigwedge_i p_i \to \bigvee_j q_j \vdash \bot} \qquad \frac{\bigwedge_i (p_i \vee \neg p_i) \quad \bigwedge_i \Gamma \cup (p_i \to \bot) \vdash g \quad \bigwedge_j \Gamma \cup q_j \vdash g}{\Gamma \cup \bigwedge_i p_i \to \bigvee_j q_j \vdash g}$$

A naive algorithm consists in doing a recursive call for each literal, say l_i, of the clause. However, if for one of the cases, the proof does not depend on the literal l_i, it is sound to ignore the other cases and immediately return (*backjump*) to a previous case-split. This is illustrated by Example 5.

▶ **Example 5** (Backjumping). Consider the following goal where each clause H_i is tagged with a set of dependencies d_i where the d_i are disjoint and do not contain the literals $\{l, m, n, o\}$.

$$\{H_1 : (l \vee m)_{d_1}, H_2 : (n \vee o)_{d_2}, H_3 : (n \to \bot)_{d_3}, H_4 : (o \to \bot)_{d_4}\} \vdash \bot$$

Suppose that we first perform a case-split over H_1. For the first case, we introduce the unit clause $H_5 : l_{\{l\}}$ with a singleton dependency *i.e.*, the literal only depends on itself. As no unit propagation is possible, we perform another case split over $H_2 : (n \vee o)_{d_2}$. For the case n, using H_3, we derive the empty clause $\bot_{d_3 \cup \{n\}}$ and for the case o, we derive the empty clause $\bot_{d_4 \cup \{o\}}$. Therefore, gathering both sub-cases, we have $\bot_{d_2 \cup d_3 \cup d_4}$. As $l \notin d_2 \cup d_3 \cup d_4$, the case-split over H_1 is irrelevant for the proof and, therefore, there is no need to explore the second case of the case-split over H_2.

A propositional prover has the idealised type `ProverT` defined below.

```
ProverT := state → HFormula → option LitSet.t
```

It takes as input the prover state `st`, and the formula to prove `g` and returns, upon success, the set of literals that are needed for the proof. Details about the components of `state` are given in Section 2.7. The `case_split` algorithm is parametrised by a prover `Prover:ProverT`. It takes as input a clause `cl` and returns a prover performing a case-analysis over all the literals of the input clause `cl`. In addition to a set of literals, it returns a boolean indicating whether backjumping is possible.

```
Fixpoint case_split cl st g  :=
 match cl with
 | [] => Some (false, LitSet.empty)
 | f::cl => match Prover (st ∪ f) g with
            | None => None
            | Some d  =>
               if f∉ d && st ⊢ d then Some (true,d)
               else match case_split cl st g with
                    | None => None
                    | Some(b, d') => if b then Some(b,d')
                                     else Some(false, d' ∪ (d \ {f}))
                    end end end.
```

If the clause is empty *i.e.*, it denotes `False`, the proof is finished. Suppose that `f` is the first literal of the clause `cl`, the prover is recursively called with the state `st` augmented with the unit clause f. If the proof `d` does not require `f` and all the literals in `d` are assigned in the current state (`st` ⊢ d), the whole case-split is spurious and the prover returns immediately. Otherwise, we recursively call `case_split` and return the updated set of literals needed by the proof. The literal `f` is momentarily removed from the dependencies of `d` but the dependencies are adjusted just after the `case_split` call by adding the dependencies of the clause `cl`.

2.5 Implication clauses

When there is no clause to branch over, the prover considers *implication clauses* and tries to (recursively) prove one of them.

```
Fixpoint prover_arrows (l : list literal) (st: state) (g: HFormula)  :=
 match l with
 | [] => None
 | f :: l => match Prover st f with
             | Some _ => Prover (st ∪ f) g  | None => prover_arrows l st g
             end  end.
```

The function `prover_arrows` takes as argument a list `l` of literals. A literal `f` in the list is of the form `POS(LIMPL` $[a_1;...;a_n]$ $b)$ which encodes an *implication clause* $\lfloor a_1 \rfloor \to \cdots \to \lfloor a_n \rfloor \to (\lfloor a_1 \to \cdots \to a_n \to b \rfloor)$ which was left aside during the CNF conversion. The prover is recursively called with, as goal, the formula $\bigwedge_i a_i \to b$. If the proof succeeds, the literal `f` holds and the proof continues with a state augmented with `f`. Otherwise, the prover tries another literal.

2.6 Theory Reasoning

At leaves of the proof search, the solver has assigned a set of literals that do not lead to a propositional conflict. At this stage, a SAT solver reports that a model is found. Yet, this propositional model may be invalidated by performing theory reasoning. Our solver is parametrised by a theory reasoner. Essentially, it takes as input a list of literals and returns (upon success) a clause. The clause is a tautology of the theory that is added to the clauses of the SAT solver. The interface of a theory reasoner is given below.

```
Record Thy := {
 thy_prover : hmap → list literal → option (hmap * clause);
 thy_prover_sound : ∀ hm hm' cl cl', thy_prover hm cl = Some (hm',cl')
    → ⟦cl'⟧  ∧ hm ⊑ hm' ∧ ∀ l ∈ cl', l ∈ hm'  }
```

A `thy_prover` takes as input a hash-cons map `hm` and a list of literals `cl`. The hash-cons map `hm` contains the currently hash-consed terms. The literals in the list `cl` are obtained from the current state of the SAT solver and are restricted to atomic formulae *i.e.*, `LAT i` for some `i`. The theory prover may either fail to make progress or return an updated hash-cons map `hm'` and a clause `cl'` such that

i) the clause `cl'` holds;
ii) the literals in `cl'` are correctly hash-consed in `hm'`;
iii) the updated hash-cons map `hm'` contains more hash-consed formulae than `hm`.

Typically, the clause `cl'` is a conflict clause and therefore the literals of `cl'` are including in those of `cl`. However, we open the possibility to perform theory propagation and generate a clause made of new literals.

2.7 Solver State and Main Loop

Using the previous components, we are ready to detail the proof state.

```
Record state := {
 fresh_clause_id : int; hconsmap : hmap; arrows : list literal;
 wneg            : iSet; defs : iSet * iSet;
 units           : ptrie (Annot bool); unit_stack : list (Annot literal);
 clauses         : watch_map }.
```

- The field `fresh_clause_id` is a fresh index that is incremented each time a novel clause is created.

- The field `hconsmap` is only updated by the theory prover and is used to ensure well-formedness conditions about hash-consing.
- The field `arrows` contains a list of literals. Each literal is of the form $\texttt{IMPL}[a_1;\ldots;a_n]b$ and represent an *implication clauses i.e.*, $(\lfloor a_1\rfloor \to \cdots \to \lfloor a_i\rfloor \to \lfloor b\rfloor) \to Lit a_1 \to \cdots \to a_i \to b$ which could not be turned into a proper *flat* clause during CNF conversion.
- The field `wneg` contains a set of hash-cons indexes corresponding to watched negative literals. These literals are added to the list of literals sent to the theory prover.
- The field `defs` is a pair of sets of hash-cons indexes. These are used for memoizing the CNF+ and CNF- computations.
- The field `units` encodes using a Patricia Tree the set of assigned literals. The boolean indicates whether the literal is positive or negative and the annotation tells which sets of initial literals were needed for the deduction.
- The field `unit_stack` is the stack of literals for which unit propagation needs to be run.
- The field `clauses` contains the indexed `watched_clauses`.

The type of the implemented prover is slightly more complicated that what we explained in the Section 2.4. In addition to a set of literals, it also threads along a `hmap` and a list of clauses learnt by theory reasoning. Theory reasoning is only running if the boolean `use_prover` is set. Termination is ensured by provided some fuel computed (without proof) from the size of the formula.

```
Fixpoint prover thy use_prover fuel st g :=
 match n with
 | O => Fail OutOfFuel
 | S n => let ProverRec := prover thy use_prover n in
   (prover_unit n ; prover_case_split ProverRec ;
    prover_impl_arrows ProverRec ; prover_thy ProverRec thy use_prover) st g
 end.
```

If the `prover` does not run out of fuel, it calls in sequence the different provers described in the previous sections. After performing *unit propagation*, it performs a case-split and recursively calls the prover. If no case-split is possible, it tries to prove one of the *implication clause* by recursively calling the prover. Eventually, if no *implication clause* can be derived, theory reasoning is called.

▶ **Example 6** (Example 3 continued)**.** Suppose that we have the proof state obtained after constructed the CNF for $a \to (b \wedge c) \to (d \vee (a \to c))$ (see Example 3). The literal $\lfloor a \to (b \wedge c)\rfloor$ triggers unit propagation and generates the clause $\lfloor a\rfloor \to \lfloor b \wedge c\rfloor$. At this stage, neither unit propagation nor case-splitting is possible. Yet, we have a single *implication clause* $(\lfloor a\rfloor \to \lfloor c\rfloor) \to \lfloor a \to c\rfloor$. Hence, we call `prover_arrows` with the singleton list $[\lfloor a \to c\rfloor]$. To prove $\lfloor a \to c\rfloor$, we introduce $\lfloor a\rfloor$ and attempt to prove c. By unit propagation, we derive first $\lfloor b \wedge c\rfloor$ and then $\lfloor b\rfloor$ and $\lfloor c\rfloor$; thus concluding the sub-proof. As a result, we augment the context with $\lfloor a \to c\rfloor$ and conclude the goal by unit propagation.

3 Soundness Proof

In this part, we give some insights about the soundness proof that is based on three main properties: well-formedness, soundness of dependencies and soundness of provers. The only axioms of the development are those of the `Int63` standard library which define native machine integers.

3.1 Well-formedness

Hash-consing has the advantage that the equality of terms can be decided by an equality of native integers without in-depth inspection of terms. However, this requires ensuring that the initial formula is correctly hash-consed and that the prover always operates with hash-consed literals. Fortunately, most of the generated literals are sub-formulae that are obtained by the CNF. The only exception is theory reasoning. The hash-consed formulae are stored in a map `m : hmap := ptrie (bool*LForm)`. The keys of the map are the hash-cons indexes and the boolean indicates whether the formula is a *classical* proposition. The set of hash-consed formulae `has_form m` is inductively defined below.

$$\frac{m(i) = (true, \texttt{LFF})}{\texttt{LFF}_i^{true} \in \texttt{has\_form}\ m} \qquad \frac{m(i) = (b, \texttt{LAT i}) \quad b \leftrightarrow [\![\texttt{LAT}\ i]\!] \vee \neg[\![\texttt{LAT}\ i]\!]}{(\texttt{LAT i})_i^b \in \texttt{has\_form}\ m}$$

$$\frac{\begin{array}{c} m(i) = (b, \texttt{LOP}\ o\ [\_^{b_1}_{i_1}; \ldots; \_^{b_n}_{i_n}]) \\ (f_1)_{i_1}^{b_1} \in \texttt{has\_form}\ m \quad \ldots \quad (f_n)_{i_n}^{b_n} \in \texttt{has\_form}\ m \\ b \leftrightarrow b_1 \wedge \cdots \wedge b_n \end{array}}{(\texttt{LOP}\ o\ [(f_1)_{i_1}^{b_1}; \ldots; (f_n)_{i_n}^{b_n}])_i^b \in \texttt{has\_form}\ m}$$

$$\frac{\begin{array}{c} m(i) = (b, \texttt{LIMPL}[\_^{b_1}_{i_1}; \ldots; \_^{b_n}_{i_n}]\ \_^{b_0}_{i_0}) \\ (f_0)_{i_0}^{b_0} \in \texttt{has\_form}\ m \quad \ldots \quad (f_n)_{i_n}^{b_n} \in \texttt{has\_form}\ m \\ b \leftrightarrow b_0 \wedge \cdots \wedge b_n \end{array}}{(\texttt{LIMP}[(f_1)_{i_1}^{b_1}; \ldots; (f_n)_{i_n}^{b_n}]\ (f_0)_{i_0}^{b_0})_i^b \in \texttt{has\_form}\ m}$$

Essentially, this consists in checking that all the sub-formulae are stored within the hmap `m` when only considering the top constructor and the hash-cons index of the sub-formulae. The advantage of this formulation is that $f_i^b \in \texttt{hash\_form}\ m$ can be checked algorithmically by a linear pass over the formula f. This definition also entails that formulae with the same hash-cons index are the same.

$$f_i^{b_1} \in \texttt{has\_form}\ m \wedge g_i^{b_2} \in \texttt{has\_form}\ m \rightarrow f_i^{b_1} = g_i^{b_2}$$

Our well-formedness conditions state that a solver state `st:state` only contains hash-consed formulae. In particular, all the literals in the clauses are hash-consed formulae. As Patricia Trees come with structural invariants, every Patricia Tree needs to be well-formed. In the prover state, the set of assigned literals (see the `units` field) is represented by a Patricia Tree `ptrie (Annot.t bool)` where the keys are hash-consed indexes. In this case, the well-formedness conditions requires that there exists a hash-consed formula corresponding to this index. As all the formulae with the same hash-cons index are equal this identifies a unique formula.

$$\texttt{wf\_units\_lit}\ u\ m = \forall i, v, u(i) = v \rightarrow \exists f_i^b, \texttt{has\_form}\ m\ f_i^b$$

The proofs of preservation are compositional and do not pose any particular issue.

3.2 Soundness of Dependencies

The logical objects in the proof state are the set of indexed watched clauses (`clauses`) , the set of assigned literals (`units`) and the stack of literals that are yet to be unit-propagated (`unit_stack`). Each clause (or literal) is also annotated by the set of literals needed for the deduction. These sets are represented by Patricia Trees using hash-consed indexes as keys.

We write c_d (resp. l_d) for a (watched) clause (resp. literal) annotated with a set of literals d. For each operation, we prove that the annotation is sound *i.e.* the conjunction of the literals in d entails the clause c (resp. the literal l). The interpretation is indexed by a hash-cons map m linking indexes to hash-consed formulae.

$$[\![c_{\{l_1,\ldots,l_n\}}]\!]_m = [\![l_1]\!]_m \wedge \ldots [\![l_n]\!]_m \to [\![c]\!]$$

A literal l is introduced by either the introduction rule or a case-split. In that case, the set d is the singleton $\{l\}$ and $[\![l_{\{l\}}]\!]$ holds. New clauses are obtained by unit propagation and the soundness of the deduced clauses is obtained from the following deduction rules:

$$\frac{d_1 \to p \qquad d_2 \to (p \to p_1 \to \ldots p_n \to q_1 \vee \ldots q_n)}{d_1 \wedge d_2 \to (p_1 \to \ldots p_n \to q_1 \vee \ldots q_n)} \qquad \frac{d_1 \to \neg q \qquad d_2 \to (q \vee q_1 \vee \ldots q_n)}{d_1 \wedge d_2 \to (q_1 \vee \ldots q_n)}$$

Syntactically, the conjunction $d_1 \wedge d_2$ is modelled by the union $d_1 \cup d_2$.

$$[\![d_1 \cup d_2]\!]_m = [\![d_1]\!]_m \wedge [\![d_1]\!]_m$$

Given a prover state `st`, $[\![\texttt{st}]\!]^{dep}$ holds if all the clauses (resp. literals) in the state have correct dependencies. The proofs that the dependencies are correct are also compositional but require well-formedness conditions to hold.

3.3 Soundness of Provers

Upon success, a prover `p:ProverT` returns a set of valid learnt clause, an updated hash-cons map but also a set of literals that are sufficient to entail a conflict. The soundness of provers is then stated by the following definition.

```
Definition sound_prover (prover: ProverT) (st: state) :=
 ∀ g hm lc d, wf_state st → g_i^b ∈ hconsmap st →
 prover st g = Success (m,lc,d) →
 ([[st]] → [[st]]^dep → [[g]]) ∧ ([[st]]^dep → [[d]]_m → [[g]])  ∧ ⋀_{c∈lc} [[c]]
```

Therefore, soundness requires to prove that the goal formula `g` is entailed by either the clauses in the state and their dependencies or the dependencies alone and the set of literals d. Both properties are needed in order to prove that *backjumping* is correct. We also have to prove that the clauses `lc` obtained by theory reasoning are sound. Sometimes, the fact that we use classical reasoning in an intuitionistic context is a source of complication for the proof because this prevents a direct forward reasoning. For instance, in a pure classical context, the CNF of a formula f generates tautologies and the correctness can be directly stated by

$$\forall cl, cl \in \texttt{cnf}\ f \to [\![cl]\!]$$

The correctness of our CNF depends on the goal formula. For instance, if the conclusion is `False`, De Morgan laws are admissible. Therefore, our formulation is the following.

$$((\forall cl, cl \in \texttt{cnf}\ f \to [\![cl]\!]) \to [\![g]\!]) \to [\![g]\!]$$

Note that if $g = \texttt{False}$, we get a double negation. For most of our proof state transformation, say `T`, a simplified correctness lemma has the form

$$\texttt{wf\_state}\ st\ \to ([\![\texttt{T}\ st]\!] \to [\![g]\!]) \to ([\![st]\!] \to [\![g]\!])$$

4 Proof by Hybrid Reflection

Having a SAT solver for deciding propositional logic, one option is to directly perform proof by reflection [9, Chap. 16]. Using this mode, we get a tactic that is similar to the existing `tauto` with the advantage that our reification process detects decidable propositions declared using the type-class mechanism [25].

In order to perform theory reasoning, the proof by reflection approach demands that we implement the interface of Section 2.6. However, a closer look shows that the interface does not expose a syntax for atoms: they are only represented by indexes. In a nutshell, the interface is purposely an empty shell. A fully reflexive approach would require to enrich the interface with an environment mapping atoms to theory-specific syntactic terms. Here, we follow a different path and implement theory reasoning using classic user-defined tactics. The advantages are that existing decision procedures can be readily reused and that user-defined tactics are more flexible. Yet, user-defined tactics and proof by reflection do not operate at the same level and embedding a powerful reflective tactic language would be a challenge by itself.

To reach our goal, we take the opposite approach and leverage the Coq extraction mechanism, thus allowing to run our SAT solver inside the Coq proof engine, *i.e.*, at the level of tactics. There, the theory reasoning interface may be implemented by calling user-defined tactics. The theory prover `thy_prover` (see Section 2.6) takes a list of literals and attempts to produce a clause. Therefore, to interface with tactics, the followings tasks need to be performed:

1. Construct a goal from the literals provided by the SAT solver.
2. (Optionally) perform theory propagation
3. Run the tactic and obtain a proof-term.
4. Return a reduced clause.

Within the proof engine, we maintain a mapping from literals to actual Coq terms. We interpret the list of literal as a clause to be discharged by the user-provided tactic. We obtain a goal G of the general form: $G{:=}\forall_{i\in I}(x_i : p_i), \bigvee_{j\in J} q_j$. Actually, the p_i and the q_j do not depend on the x_i. Yet, this dependent product notation is convenient to explain our minimisation procedure.

The fact that the conclusion is a disjunction $\bigvee_{j\in J} q_j$ may trigger, for some tactics, a costly case-analysis which would be more efficiently performed at the level of the SAT solver. For this purpose, we leverage the type-class mechanism to perform theory propagation at the level of individual propositions. For instance, if one of the q_j is an equality over `Z`, say `x=y`, we directly return to the SAT solver the tautology clause `x<y` $\vee$ `x=y` $\vee$ `y<x`.

If the tactic fails, the unsuccessful goal G is a potential counter-example and is returned to the user for inspection. If the tactic proves the goal, the clause G holds and is therefore a conflict clause. A sound, but naive, approach is to return this clause containing all the input literals. Yet, to improve the efficiency of the SAT solver, it is desirable to reduce the clause and produce a minimal *unsatisfiable core*. In other words, we search for sets $I' \subseteq I$ and $J' \subseteq J$ such that $G'{:=}\forall_{i\in I'}(x_i : p_i), \bigvee_{j\in J'} q_j$ is still a tautology. In our context, if the tactic succeeds, it also produces a proof-term. Using a syntactic analysis, we track the subset of the x_i that are used in the proof-term and therefore obtain a set $I' \subseteq I$ containing the needed hypotheses. To minimise the set J, a finer grained analysis of the proof term may be possible. However, there is a sweet spot when the positive literals are classical propositions. In that case, the goal G is equivalent to G'

$$G'{:=}\forall_{i\in I}(x_i : p_i), \forall_{j\in J}(y_j : \neg p_j), \texttt{False}$$

Using this formulation, the same syntactic analysis extracts both a subset $I' \subseteq I$ of the negative literals and a subset $J' \subseteq J$ of the positive literals. When the propositions are not classical, we perform some partial iterative proof-search and try to prove each of the q_i, one at a time. Once the minimisation is done, we adjust the proof-term for the minimised conflict clause: this consists in recomputing the correct De Bruijn indexes to accommodate for the removed hypotheses.

If the extracted SAT solver succeeds, we have the guarantee that the goal is provable but the proof elements still need to be pieced together. The SAT solver returns the set of needed literals. We exploit this information to remove from the context the propositions that are irrelevant for the proof. Now, we enrich the context with all the conflict clauses that were generated during the SAT solver run, re-using the cached proof terms. At this stage, we have a goal that is a propositional tautology. It is solved by re-running the SAT solver, without theory reasoning, by a classic proof by reflection.

Using the extracted SAT solver has the advantage that the possible bugs are limited to the interface with user-provided tactic. Another possible design would be to rely on an external untrusted (intuitionistic) SAT solver. This would have some speed advantage for the generation of conflict clauses but would increase the code base. In our case, we reuse the same verified component both in the Coq proof engine and to perform proof by reflection.

5 Experiments

Before showing some larger scale experiments, we come back to the motivating examples (see Section 1.1) and explain how they are solved by our tactic.

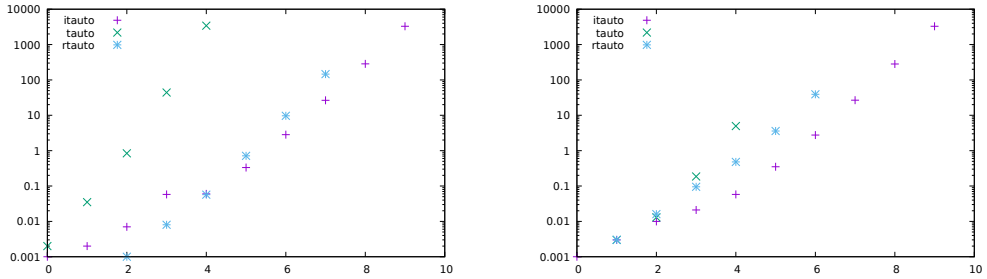
5.1 Back to Motivating Examples

Consider again the goals in Example 1. For `Ex1`, the SAT solver has knowledge that `x>=0` is a classical proposition and therefore performs a case-split over `x>=0→ p`. The first subcase is solved by theory reasoning (`x=0 → ~ x>= 0 → False` holds); and the second by propositional reasoning (`p → p`). For `Ex2`, the SAT solver does not make progress. However, it asks the leaf tactic `congruence` whether a watched negative literal (*i.e.*, `a=c`) may be deduced from the assigned literal (*i.e.*, `a=b` and `b=c`). As `a=b → b=c → a=c` can be proved by `congruence`, the clause is asserted by theory reasoning. The proof follows by propositional reasoning.

For Example 2, the goal is solved by `intuition congruence` using two identical calls to the leaf tactic `congruence`. Because our SAT solver threads learnt clauses along the computation, the same goal is solved by `itauto congruence` by calling `congruence` only once.

5.2 Pigeon Hole

The Pigeon Hole Principle, stating that there is no way to fit n+1 pigeons in n holes, is a hard problem for resolution based SAT solvers. The algorithm presented here is no exception and the running time will be exponential in n. This is however a useful benchmark for assessing scalability. We have benchmarked our `itauto` tactic against the existing `tauto` and `rtauto` tactics using a timeout of 3600s and a memory limit of 15GB on a laptop (Intel Core i7 at 1.8GHz with 32GB of RAM). The results are shown in Fig. 1. For `itauto`, the running time of the tactic and the type-checking time (`Qed` time) are similar. As we perform a pure proof by reflection, this is not surprising. Though `itauto` scales slightly better, the running time



(a) Running time of tactics.

(b) Running time of type-checking.

Figure 1 Pigeon Hole for `itauto`, `tauto` and `rtauto`.

are similar to `rtauto` which only performs proof by reflection to check certificates. `tauto` is the least scalable and the proof search reaches timeout for 5 pigeons. `rtauto` exhausts its memory quota when checking its certificate for 7 pigeons. `itauto` reaches the time limit of 3600s for 10 pigeons.

5.3 On existing developments

We have benchmarked `itauto` against the existing Coq tactics `tauto` and `intuition` for the Bedrock2[9] and CompCert[10] developments using Coq 8.14+alpha. We have replaced calls to `tauto` and `intuition tac` for `tac` $\in$ {`idtac`, `assumption`, `discriminate`, `congruence`, `lia`, `auto`, `eauto` }. We have ruled out calls to `intuition` when they generate sub-goals. In terms of completeness, `itauto` is able to solve the vast majority of the goals. One representative case of failure is given in Example 7.

▶ **Example 7.** The following goal is solved by `intuition congruence`.

```
Goal true = true ↔ (Z → (False ↔ False)).
```

Yet, `itauto congruence` fails because `Z` (*i.e.*, the type of integers) has type `Set` and therefore the whole expression `Z → (False ↔ False)` is reified as an opaque atomic proposition. Our solution is to call `itauto` recursively *i.e.*, `itauto (itauto congruence)` so that an hypothesis `x:Z` is explicitly introduced. The same approach works for goals with inner universal quantifiers.

For a few instances, we also strengthen the leaf tactic replacing *e.g.*, `idtac` by `reflexivity`, and, `discriminate` by `congruence`. This is due to the fact that `intuition` implicitly calls `reflexivity` and that our handling of theory reasoning sometimes introduces negations which fool the `discriminate` tactic.

After modification, Bedrock2 performs 1621 calls to `itauto`. For 40% of the goals, the running time differs by less than 1ms and `itauto` outperforms the historic tactics for 40% of the cases. Overall, `itauto` is faster and there is an slight speedup of 1.07. Yet, all the calls are quickly solved; the slowest goal is solved in 0.18s.

CompCert performs 924 calls to `itauto`. For 76% of the goals, the running time also differs by less than 1ms. `itauto` outperforms the historic tactics for 19% percent of the goals.

[9] `https://github.com/mit-plv/bedrock2`
[10] `https://github.com/AbsInt/CompCert`

Yet, when `itauto` is faster it can be by several orders of magnitudes: the 19% percent of goals are solved more than 20 times faster by `itauto`. For instance, the maximum running time of `tauto` is 5.26s to be compared to 0.81s for `itauto`. Overall, `itauto` performs better with a speedup of 2.8 for solving all the goals.

In summary, the results are rather positive though the advantage may be slim. It seems that `itauto` shows a decisive advantage for goals that are very slow with the existing tactics. As shown by the Pigeon Hole experiment, this would indicate a better scalability. It would not be surprising for `itauto` to be slower on simple goals where the overhead of setting up the proof by reflection cannot be amortised. `itauto` spends time in different proof tasks: SAT solver, theory reasoning and proof by reflection. We are confident that the SAT solver is scalable and reasonably fast. Yet, this is not always the bottleneck, and, the positive results reported here were only made possible by fine tuning other tasks *e.g.*, the reification code.

6 Related Work

For propositional logic, Weber and Amjad [27] for HOL theorem provers and Armand *et al.* [3] for Coq show how to efficiently validate resolution proofs that can be generated by modern SAT solvers using Conflict Driven Clause Learning (*e.g.* zChaff [21]). Satisfiability Modulo Theory (SMT) solvers (*e.g.*, veriT [8], Z3 [12] , CVC4 [13]) produce proof artefacts. Böhme and Weber [7] show how to efficiently perform proof reconstruction for HOL provers. Armand *et al.* [2] and Besson *et al.* [4] extract proof certificates from SMT proofs that are validated in Coq. Sledgehammer [5] interfaces Isabelle/HOL with a variety of provers; Metis [18, 23] is in charge of performing proof reconstruction. We follow a different approach and verify a SAT solver interfaced with the tactics of Coq. What we loose in efficiency, we gain in flexibility because the user might use her own fine-tuned domain specific tactics.

Claessen and Rosén [10] build a prover for intuitionistic logic on top of a black-box SAT solver. Their implementation is more efficient than ours but is not integrated inside a proof-assistant. Lescuyer and Conchon [19, 20] formalise a reflexive SAT solver for classical logic. Their implementation features a lazy definitional CNF that is not fully exploited because of the lack of hash-consing. Compared to ours, their SAT solver performs neither lazy unit propagation nor backjumping. Blanchette *et al.* [6] formalise a Conflict Driven Clause Learning (CDCL) SAT solver and derive a verified implementation. Using this framework, Fleury, Blanchette and Lamich [15] derive an imperative implementation using watched literals. Our implementation has less sophisticated features but is integrating as a reflective proof procedure and allows to perform theory reasoning.

7 Conclusion and Future Work

We have presented `itauto` a reflexive tactic for intuitionistic propositional logic which can be parametrised by user-provided tactics. The SAT solver is optimised to leverage classical reasoning thus limiting, when possible, costly intuitionistic reasoning. Our SAT solver has several features found in modern SAT solver. Yet, the implementation could be improved further. A first improvement would be to generate more sophisticated learnt clauses. This could be done by attaching to each clause, not only the needed literals, but also the performed unit propagations. Another improvement would be to use primitive persistent arrays (available since Coq 8.13) allowing to implement efficiently more imperative algorithms *e.g.*, 2-watched literals.

References

1 John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., USA, 1978.

2 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011. doi:10.1007/978-3-642-25379-9_12.

3 Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In *ITP*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010. doi:10.1007/978-3-642-14052-5_8.

4 Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT Proofs for Fast Reflexive Checking Inside Coq. In *CPP*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011. doi:10.1007/978-3-642-25379-9_13.

5 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. doi:10.1007/s10817-013-9278-5.

6 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. *J. Autom. Reason.*, 61(1-4):333–365, 2018. doi:10.1007/s10817-018-9455-7.

7 Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In *ITP*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010. doi:10.1007/978-3-642-14052-5_14.

8 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In *CADE*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009. doi:10.1007/978-3-642-02959-2_12.

9 Pierre Castéran and Yves Bertot. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004. URL: https://hal.archives-ouvertes.fr/hal-00344237.

10 Koen Claessen and Dan Rosén. SAT Modulo Intuitionistic Implications. In *LPAR-20*, volume 9450 of *LNCS*, pages 622–637. Springer, 2015. doi:10.1007/978-3-662-48899-7_43.

11 Pierre Corbineau. Deciding Equality in the Constructor Theory. In *TYPES*, volume 4502 of *LNCS*, pages 78–92. Springer, 2006. doi:10.1007/978-3-540-74464-1_6.

12 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

13 Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: How it works, and how to use it. In *FMCAD*, page 7. IEEE, 2014. doi:10.1109/FMCAD.2014.6987586.

14 Roy Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *J. Symb. Log.*, 57(3):795–807, 1992. doi:10.2307/2275431.

15 Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In *CPP*, pages 158–171. ACM, 2018. doi:10.1145/3167080.

16 Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004. doi:10.1007/978-3-540-27813-9_14.

17 John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *J. Autom. Reason.*, 21(3):279–294, 1998. doi:10.1023/A:1006023127567.

18 Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

19 Stéphane Lescuyer and Sylvain Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Emerging Trends of TPHOLs*, 2008.

20 Stéphane Lescuyer and Sylvain Conchon. Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In *FroCoS*, volume 5749 of *LNCS*, pages 287–303. Springer, 2009. doi:10.1007/978-3-642-04222-5_18.

21 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

22 Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

23 Lawrence C. Paulson and Kong Woei Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In *TPHOLs*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007. doi:10.1007/978-3-540-74591-4_18.

24 David A. Plaisted and Steven Greenbaum. A Structure-Preserving Clause Form Translation. *J. Symb. Comput.*, 2(3):293–304, 1986. doi:10.1016/S0747-7171(86)80028-1.

25 Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.

26 G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer, 1983. doi:10.1007/978-3-642-81955-1_28.

27 Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Appl. Log.*, 7(1):26–40, 2009. doi:10.1016/j.jal.2007.07.003.

28 Hantao Zhang and Mark E. Stickel. An Efficient Algorithm for Unit Propagation. In *AI-MATH*, pages 166–169, 1996.

29 Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam Method. *J. Autom. Reason.*, 24(1/2):277–296, 2000. doi:10.1023/A:1006351428454.