# SAT Modulo Intuitionistic Implications

Koen Claessen(✉) and Dan Rosén(✉)

Chalmers University of Technology, Gothenburg, Sweden
{koen,danr}@chalmers.se

**Abstract.** We present a new method for solving problems in intuitionistic propositional logic, which involves the use of an incremental SAT-solver. The method scales to very large problems, and fits well into an SMT-based framework for interaction with other theories.

## 1 Introduction

Let us take a look at *intuitionistic propositional logic*. Its syntax looks just like classical propositional logic:

$$\begin{array}{lll} A ::= a \mid b \mid c \mid .. \mid q & & \text{-- atoms} \\ \quad \mid A_1 \wedge A_2 & & \text{-- conjunction} \\ \quad \mid A_1 \vee A_2 & & \text{-- disjunction} \\ \quad \mid A_1 \rightarrow A_2 & & \text{-- implication} \\ \quad \mid \bot \mid \top & & \text{-- false/true} \end{array}$$

However, its definition of truth is considerably weaker than for classical logic. In Fig. 1, we show a Hilbert-style proof system for intuitionistic propositional logic. In the figure, we use $A_1, .., A_n \vdash B$ as a short-hand for $A_1 \rightarrow ..(A_n \rightarrow B)$. Only "computationally valid" derivations can be made in intuitionistic logic. For example, the classical law of the excluded middle $a \vee \neg\, a$ does not hold. Here, we use $\neg\, a$ as a short-hand for $a \rightarrow \bot$.

In this paper, we are interested in building a modern, scalable, automated method for proving formulas in intuitionistic propositional logic. By modern, we mean that we would like to make use of the enormous recent advances in automated theorem proving in the form of SAT and SMT techniques. We do not want to reinvent the wheel, rather we would like to investigate if there exists an SMT-like way of building an intuitionistic theorem prover *on top of* an existing SAT-solver. The hope is that this also results in a scalable method.

It is perhaps surprising that we ask this question, because at first sight it does not seem natural to embed intuitionistic logic into classical logic; after all, we can derive much more in classical logic than intuitionistic logic.

The key insight we make use of comes from Barr [1]. Given a set $S$ of propositional clauses of the shape:

$$(a_1 \wedge .. \wedge a_n) \rightarrow (b_1 \vee .. \vee b_m)$$


M. Davis et al. (Eds.): LPAR-20 2015, LNCS 9450, pp. 622–637, 2015.
DOI: 10.1007/978-3-662-48899-7_43

$$\frac{A \quad A \to B}{B}\ (\text{MP})$$

$$\begin{array}{rll}
A, B & \vdash A & (K) \\
A \to B \to C, A \to B & \vdash A \to C & (S) \\
A \wedge B & \vdash A & (FST) \\
A \wedge B & \vdash B & (SND) \\
A, B & \vdash A \wedge B & (PAIR) \\
A & \vdash A \vee B & (INL) \\
B & \vdash A \vee B & (INR) \\
A \to C, B \to C & \vdash (A \vee B) \to C & (CASE) \\
\bot & \vdash A & (BOT) \\
A & \vdash \top & (TOP)
\end{array}$$

**Fig. 1.** A Hilbert-style proof system for intuitionistic propositional logic

Here, $a_i$ and $b_j$ are propositional atoms. Now, the remarkable insight is this: The question of whether or not a given other clause of that same shape is derivable from the set $S$ is oblivious to which logic we are in: classical or intuitionistic. The derivation power of these two logics on this subset is *equivalent*!

In other words, if we have an intuitionistic logic problem that we are interested in solving, we would like to "distill" this problem into two parts: The first, and hopefully largest, part would be expressible using clauses of the above shape, and the second, hopefully tiny, part would consist of the part of the problem not expressible using clauses of the above shape. We can then use a standard SAT-solver to solve the first part, and an extra theory on the side to deal with the second part.

Indeed, it turns out that clauses of the above shape are not quite enough to represent all intuitionistic formulas. We also need clauses of the following shape:

$$(a \to b) \to c$$

Here, $a, b, c$ are propositional atoms. We call these clauses *implication clauses*, and clauses of the first kind are called *flat clauses*.

Finally, we need one more rule that tells us how flat clauses interact with implication clauses:

$$\frac{(p_1 \wedge .. \wedge p_n \wedge a) \to b \qquad (a \to b) \to c}{(p_1 \wedge .. \wedge p_n) \to c}\ (IMPL)$$

From one implication clause and one flat clause, we can generate a new flat clause. This rule, together with any complete proof system for classical logic applied to flat clauses, turns out to be a complete proof system for intuitionistic

logic. The fact that no new implication clauses are generated during a proof is an extra bonus that alleviates automated proof search even more: Rule ($IMPL$) can be implemented as an SMT-style theory on top of a SAT-solver.

Thus, we ended up constructing a simple, scalable, automated theorem prover for intuitionistic propositional logic based on a SAT-solver.

To our knowledge, this is the first paper to take an SMT-based approach on top of a SAT-solver to intuitionistic logic. As we shall see, the implementation of the theory of intuitionistic implications turns out to be rather unconventional, because it has a recursive structure (calling the theorem prover itself!). But the overall design of the prover is quite simple; only one SAT-solver is needed to do all the reasoning, and no extra quantification is necessary. The result is a robust prover that performs very well on existing as well as new benchmarks.

## 2 The Procedure

In this section, we describe our procedure for proving (and disproving) formulas in intuitionistic propositional logic.

### 2.1 Canonical Form

The first thing we do when trying to prove a formula $A$, is to transform it into canonical form, by a process called *clausification*. A problem in canonical form is characterized by two sets of different kinds of clauses $R$ and $X$, plus an atom $q$:

$$(\bigwedge R \wedge \bigwedge X) \to q$$

Here, the set $R$ only contains so-called *flat clauses* $r$, which are of the following shape:

$$r \ ::= \ (a_1 \wedge .. \wedge a_n) \to (b_1 \vee .. \vee b_m)$$

In the above, $a_i$ and $b_j$ denote atoms. When $n = 0$, the left-hand side is $\top$; when $m$=0, the right-hand side is $\bot$. The set $X$ only contains *implication clauses* $i$, which have the following shape:

$$i \ ::= \ (a \to b) \to c$$

Here, $a, b, c$ are atoms or the constants $\bot$ or $\top$.

Any formula $A$ can be rewritten into a provability-equivalent formula in canonical form. As an example, consider the formula $a \vee \neg\ a$. Its canonical form is:

$$((a \to q) \land ((a \to \bot) \to q)) \to q$$

We can see that the canonical form consists of one flat clause $a \to q$ and one implication clause $(a \to \bot) \to q$, and a final goal $q$.

The procedure we use to rewrite any formula into canonical form is very similar to the Tseitin method for clausification of formulas in classical propositional logic [6], but adapted to be sound for intuitionistic logic.

We start by assuming that $A$ is of the shape $B \to q$, for some atom $q$. If it is not, we can make it so by introducing a new atom $q$ and using $(A \to q) \to q$ (where $B$ would thus be $A \to q$) instead[1].

Next, we transform $B$ into the two sets of clauses $R$ and $X$. We can do this using a number of transformation steps of the shape $A \rightarrowtriangle B_1, .., B_n$, which transform an assumption $A$ into an equivalent set of assumptions $B_i$. Most of these transformations assume that the formulas they work on are implications. However, the first transformation step can be used when a formula is not already an implication:

$$A \quad \rightarrowtriangle \quad \top \to A$$

The next 3 transformations can be used when the left-hand or right-hand side of an implication does not have the right shape, as dictated by the clause being a flat clause or an implication clause:

$$\begin{array}{rcll} (A \lor B) \to a & \rightarrowtriangle & A \to a, & B \to a \\ a \to (A \land B) & \rightarrowtriangle & a \to A, & a \to B \\ A \to (B \to C) & \rightarrowtriangle & (A \land B) \to C & \end{array}$$

In the above, $a$ stands for either a regular atom, or one of the constants $\bot$ or $\top$. In order to have atoms appear at the right places, we can use the following 5 transformation steps:

$$\begin{array}{rcll} A \to (.. \lor B \lor ..) & \rightarrowtriangle & A \to (.. \lor b \lor ..), b \to B & \\ (.. \land A \land ..) \to B & \rightarrowtriangle & (.. \land a \land ..) \to B, A \to a & \\ (A \to B) \to C & \rightarrowtriangle & (a \to B) \to C, & a \to A \\ (A \to B) \to C & \rightarrowtriangle & (A \to b) \to C, & B \to b \\ (A \to B) \to C & \rightarrowtriangle & (A \to B) \to c, & c \to C \end{array}$$

In the above rules, $a$, $b$ and $c$ appearing on the right-hand side of a rule denotes a fresh atom not appearing anywhere else.

The above rules are a complete set of rules to turn any formula $B$ into a set of flat clauses $R$ and implication clauses $X$. The combined size of the clauses in $R$ and $X$ can be kept to at most twice the size of the size of the original formula $B$, because we never copy whole formulas, and we only need to introduce a fresh atom $b$ for any subformula at most once.

[1] provability-equivalent to $A$ because (1) $A$ implies the formula, and (2) if we take $q{:=}A$, the formula implies $A$.

```
  -- flat clauses R
  -- implication clauses X
  -- proof goal q
procedure prove (R, X, q)
  s = newSolver ();
  for r ∈ R :
    addClause (s, r);
  for i ∈ X :
    let (a → b) → c = i
    addClause (s, b → c);
  return intuitProve (s, X, ∅, q);
```

**Fig. 2.** Top-level procedure for intuitionistic proving

### 2.2 The SAT-Solver

The proving procedure makes use of a standard off-the-shelf (classical) SAT-solver $s$, that supports the following operations:

```
procedure newSolver ();
procedure addClause (s, r);
procedure satProve (s, A, q);
```

The procedure *newSolver* creates a new, unconstrained SAT-solver. The procedure *addClause* takes a SAT-solver $s$ and a flat clause $r$, and adds the clause $r$ as a constraint to $s$.

The procedure *satProve* takes a SAT-solver $s$, a set of assumptions $A$, and a goal $q$. The assumptions $A$ as well as the goal $q$ are atoms. The procedure *satProve* tries to prove the goal $q$, from the assumptions $A$ and all flat clauses that have been added so far. It produces one of two results:

- *No* $(M)$, if no proof could be found. Here, $M$ is a model that is found by the SAT-solver represented as a set of true atoms. The model $M$ is guaranteed to satisfy all added clauses, all assumptions, but it makes the goal $q$ false. So, we know $A \subset M$ and $q \notin M$.
- $Yes$ $(A')$, if a proof could be found. Here, $A'$ is the subset of the assumptions that were actually used in the proof of $q$ from the clauses. So, we know that $A' \subset A$.

The set $A'$ in the *Yes* answer can be produced by most modern SAT-solvers. Some solvers (such as MiniSAT [2] which we use in our implementation) support this operation directly in their API. Other solvers support the construction of an unsatisfiable core, which can be used to get the same information.

```
-- SAT-solver s
-- implication clauses X
-- assumptions A
-- proof goal q
procedure intuitProve (s, X, A, q)
  loop
    switch satProve (s, A, q)
      case Yes (A') :
        return Yes (A');
      case No (M) :
        if intuitCheck (s, X, M) then
          return No (M);
```

**Fig. 3.** Standard CEGAR-loop for intuitionistic proving

### 2.3 Proving Procedure

The complete proving procedure (after transformation to canonical form) is pictured in ≈25 lines of code in Figs. 2–4.

The top-level procedure *prove* is shown in Fig. 2. Its arguments are a set of flat clauses $R$, a set of implication clauses $X$, and a goal $q$. It first creates a SAT-solver $s$, and adds all flat clauses $r$ to it. Furthermore, for each implication clause $(a \to b) \to c$ from $X$, it adds the flat clause $b \to c$ (which is implied by the implication clause). Finally, it calls the main proving procedure *intuitProve*.

The main proving procedure *intuitProve* is shown in Fig. 3. Its arguments are a SAT-solver, a set of implication clauses $X$, a set of assumptions $A$, and a goal $q$. The procedure has the standard shape of a CEGAR-loop. First, it tries to find a classical proof, using the SAT-solver only on the flat clauses. If this succeeds, we are done. If not, there is a classical model $M$ which is going to be checked by the procedure *intuitCheck*, explained in the next subsection.

If *intuitCheck* determines that the found model indeed corresponds to an intuitionistic model, it returns $True$, and we return the model as the answer. If *intuitCheck* finds the model inadequate, it will have generated an *extra* flat clause in the SAT-solver, and it returns $False$. In this case, we simply loop and try again.

### 2.4 Checking Procedure

When we find a classical model $M$, which is guaranteed to satisfy all flat clauses, we have to check whether or not it corresponds to an intuitionistic model. This means that, for each implication clause $(a \to b) \to c$ in $X$, we have to check that if $a \to b$ is true under $M$, then $c$ should also be true under $M$.

In order to help us decide which implication clauses should be investigated, let us take a look at the following table, where we consider an implication clause

| | a | b | c | $(a \to b) \to c$ |
|---|---|---|---|---|
| (1) | - | - | 1 | Yes |
| (2) | - | 1 | 0 | No |
| (3) | 1 | 0 | 0 | Yes |
| (4) | 0 | 0 | 0 | ? |

$(a \to b) \to c$, and we have partitioned the $2^3$ possibilities for valuations of $a, b, c$ into 4 separate cases.

In case (1), the implication clause is fulfilled, since $c$ is true, and so the whole implication is also true. Case (2) is definitely something that contradicts the implication clause; $b$ is true and therefore also $a \to b$, but $c$ is not true. Fortunately, for each implication clause $(a \to b) \to c$, we have already added the flat clause $b \to c$ (see Fig. 2) which excludes this case. In case (3), $a \to b$ is definitely not true, and so $c$ does not have to be true either.

The only case that is left that we have to check is case (4). Here, $a \to b$ is classically true, but intuitionistically, we do not know whether or not it is true, and therefore we do not know whether or not $c$ should be true.

Thus, what we have to do is check whether or not we can prove $a \to b$ using the true atoms from the current model $M$. If we can, then surely the current model was wrong since $c$ was not true. If we cannot not, the current model fulfills the implication clause also.

As we can see in Fig. 4, the way we check whether or not $a \to b$ is provable under the current model, is to call *intuitProve* recursively, using $M \cup \{a\}$ as assumptions, and $b$ as the proof goal. If the answer is $No$ (–), everything is fine. If the answer is $Yes$ $(A')$, we generate a new flat clause, using the following proof rule $(IMPL)$:

```
-- SAT-solver s
-- implication clauses X
-- model M
procedure intuitCheck (s, X, M)
  for i ∈ X :
    let (a → b) → c = i
    if a, b, c ∉ M then
      switch intuitProve (s, X − {i}, M ∪ {a}, b)
        case Yes (A') :
          addClause (s, (⋀ A' − {a}) → c)
          return False;
  return True;
```

**Fig. 4.** Checking whether or not a SAT-model is also an intuitionistic model

$$\frac{(p_1 \wedge .. \wedge p_n \wedge a) \rightarrow b \qquad (a \rightarrow b) \rightarrow c}{(p_1 \wedge .. \wedge p_n) \rightarrow c} \quad (IMPL)$$

When the answer is $Yes$ $(A')$, it means that we have proved that $\bigwedge A' \rightarrow b$ (left-hand premise) and using the implication clause (right-hand premise), we can conclude $\bigwedge A' - \{a\} \rightarrow c$ (conclusion), which is the flat clause we then add to the SAT-solver.

We would like to note here that the efficiency of our algorithm is mainly brought by reducing the assumptions $M \cup \{a\}$ from the question to the assumptions $A'$ in the answer. It is the actually needed assumptions $A'$ that are used when constructing the new flat clause, not the originally given assumptions $M \cup \{a\}$!

Rule $(IMPL)$ allows us to create a new flat clause from an existing flat clause and an implication clause. It is the only extra proof rule we use (apart from classical resolution on flat clauses), and thus the total number of implication clauses during proof search remains constant. It is easy to see that $(IMPL)$ is sound once one realizes that the left-hand side premise is equivalent to $(p_1 \wedge .. \wedge p_n) \rightarrow (a \rightarrow b)$; then $(IMPL)$ is simply an instance of the cut rule:

$$\frac{A \rightarrow B \qquad B \rightarrow C}{A \rightarrow C} \quad (CUT)$$

The three procedures $prove, intuitProve, intuitCheck$ together completely make up the proving algorithm.

### 2.5 Correctness

Correctness of the algorithm consists of three parts: Termination (the algorithm terminates on all inputs), soundness (when the algorithm claims to have proven a formula, there indeed is a proof), and completeness (when the algorithm claims the unprovability of a formula, it indeed is not provable).

**Termination.** There are two possible causes of non-termination: The loop in $intuitProve$, and the mutual recursion between $intuitProve$ and $intuitCheck$.

The loop in $intuitProve$ terminates, because in each loop iteration, a clause is added to the SAT-solver that is false in the model $M$ that made all previously added clauses true. Thus, each loop iteration strictly reduces the number of models that satisfy all clauses in the SAT-solver. Eventually, the loop must terminate.

The mutual recursion between $intuitProve$ and $intuitCheck$ terminates because in each recursive call to $intuitProve$, the set $X$ shrinks by one element $i$.

**Soundness.** If the algorithm terminates with a $Yes$ (–) answer, the SAT-solver will have proved the goal from the flat clauses and the assumptions. Soundness is thus argued based on two arguments: (1) classical inference may be used to intuitionistically conclude flat clauses from sets of flat clauses, and (2) all flat clauses in the SAT-solver are implied by the original clauses $R$ and $X$.

As to (1), this observation was already made by Barr [1], but we briefly restate the argument here. Any classical inference of a flat clause from a set of flat clauses can be simulated by the resolution rule:

$$\frac{A \to (C \vee a) \qquad (a \wedge B) \to D}{(A \wedge B) \to (C \vee D)} \quad (RES)$$

The resolution rule $(RES)$ as stated above also holds intuitionistically. Thus, any classical proof deriving a flat clause from a set of flat clauses also admits an intuitionistic proof.

As to (2), all flat clauses in the SAT-solver either directly come from the set $R$, or they are derived using the rule $(IMPL)$, which we have already argued is sound.

**Completeness.** We show that when the algorithm terminates with a $No$ $(M)$ answer, there exists an intuitionistic Kripke model with a "starting world" $w_0$ that satisfies all flat clauses, all implication clauses, and in which the proof goal $q$ is false. We construct this Kripke model below.

Consider the last top-level call to *intuitCheck*, the call that validated the last model that was found. It executed many (recursive) calls to *intuitCheck*, each of them returning $True$. Now, let each of these calls be a unique world $w$ in the set of worlds $W$. The valuation $M$ $(w)$ associated with each world $w$ is the model $M$ with which the corresponding call of *intuitCheck* was given. The world $w_0$ is associated with the top-level call to *intuitCheck*.

Define the call-relation $C$ as follows: $w_1$ $C$ $w_2$ if and only if the call $w_1$ made the call $w_2$. The accessibility relation $\leqslant$ on $W$ is defined to be the *reflexive, transitive* closure of $C$. The relation $\leqslant$ satisfies the *persistency condition*: each call to *intuitCheck* makes the set of true atoms larger by adding the atom $a$ to $M$ in the calls to *intuitProve*.

All flat clauses in $R$ are satisfied by the valuations of all $w$, because all of these are models of the SAT-solver $s$, which guarantees that all flat clauses are made true by all models. This also means that all flat clauses are true in all $w$.

The proof goal $q$ is false in $w_0$, because the top-level call to *intuitProve* generated a counter-model to $q$.

All implication clauses in $X$ are true in $w_0$. To see this, consider an implication clause $i = (a \to b) \to c$ from $X$ and a world $w$ in which $a \to b$ is true, but $c$ is false. If $c$ is false in $w$, then so is $b$, because we have the flat clause $b \to c$ that we added to $s$. If $b$ is false in $w$, then so is $a$, because $a \to b$ is true in $w$ by our assumption. Every world is reachable by $C$-steps from $w_0$, and thus so is $w$. By persistency, $a, b, c$ must be false in *every* world on the $C$-path from $w_0$ to $w$. This means that the implication clause $i$ is still part of the set of implication clauses that belonged to the *intuitCheck*-call represented by $w$. This means that a call to *intuitProve* will be made from $w$ in which $a$ will be added to the assumptions, and with $b$ as the goal, and which furthermore returns with $No$ (–). This contradicts our assumption that $a \to b$ is true in $w$. So, all original implication clauses are true in $w_0$.

## 3 Optimizations

In this section, we discuss a number of possible optimizations that can be made to the basic algorithm, and their perceived effect on efficiency.

**Keep Checking After Finding an Offending Implication Clause.** The procedure *intuitCheck* only generates one new clause when something is wrong with a found model. Typically, a CEGAR-loop may benefit from generating multiple clauses that indicate what is wrong with a proposed model. The reason is that if we find $k$ unrelated reasons for why a found model is wrong, we may save time by not having to find $k$ different models, each triggering that reason.

```
-- SAT-solver s
-- implication clauses X
-- model M
procedure intuitCheck2 (s, X, M)
  okay = True;
  for i ∈ X :
    let (a → b) → c = i
    if a, b, c ∉ M then
      switch intuitProve (s, X − {i}, M ∪ {a}, b)
        case Yes (A′) :
          satAddClause (s, (⋀ A′ − {a}) → c)
          okay = False;
  return okay;
```

**Fig. 5.** Checking whether or not a SAT-model is also an intuitionistic model

In Fig. 5, we present a slightly adapted version of *intuitCheck* that checks for all implication clauses whether or not they are content with the currently found model, instead of aborting after finding the first offending implication clause.

We implemented this adapted method, and compared it experimentally against the simple first version. The change made some running times of the benchmarks worse, others better. All running times remained in the same order of magnitude. There were slightly more cases in which the running time was improved, so we decided to keep this variant.

Conclusion: The balance in favor of this optimization was not completely convincing, and it may change with more different benchmarks.

**Minimize the Number of Offending Implication Clauses.** It is obvious that the running time of the algorithm is affected mostly by the number of implication clauses that need to be checked. So, instead of just finding any model $M$, we considered adding an optimization phase that tried to minimize

the number of implication clauses that needed to be investigated (i.e. the number of implication clauses where $a = b = c = 0$).

The experimental results showed that this method was worse in every case where a difference could be observed. The reason was that the minimization methods we tried were slowing things down very much, and the final effect on the number of implications that had to be checked was not as great as we hoped for. We tried global minimization methods (minimizing the number of offending implication clauses) as well as local optimization methods (using subset-minimization).

Conclusion: We are still interested in minimization, but we need to (1) find better suited minimization methods, and (2) more suitable benchmarks where this would make an actual difference.

**Reuse of Already Found Models.** From the correctness proof, it becomes clear that a Kripke model can be constructed from a run of the algorithm that results in a *No* (–) answer. Currently, this Kripke model always has a tree-shape. Some trees can be compacted into directed acyclic graphs (DAGs), enabling a possible exponential speed-up.

The algorithm can be adapted to keep a table of already found and checked models, which grows during calls of *intuitCheck*. Whenever we call *intuitProve*, we can consult the table of models to see if we can immediately see that the answer is going to be *No* (–).

We have implemented this optimization, but have not thoroughly evaluated it. This remains future work.

## 4 Related Work and Experimental Results

In this section, we compare our method against other, existing methods for automated proving of intuitionistic propositional formulas.

*Competing Tools.* The main competitors for automated proof for intuitionistic propositional logic are IntHistGC [4] and fCube [3]. Both are provers that perform a backtracking search directly on a proof calculus, and are therefore rather different from the approach taken here. IntHistGC implements clever backtracking optimizations that avoid recomputations in many cases. fCube implements several pruning techniques on a tableau-based calculus.

*Benchmarks.* We have used three different benchmark suites to compare the provers experimentally.

– ILTP [5], these are 12 problems parametrized by a size. Being from 2006, this is a quite old set of benchmarks now. We used the extended version that was used in the evaluation of IntHistGC. In this version, two problems were generated up to size 38 and all other problems up to size 100, leading up to a total of 555 problem instances.

– IntHistGC benchmarks, these are 6 problems parametrized by a natural number. These benchmarks are newer. They are carefully constructed sequences of formulas that separate classical and intuitionistic logic. The total number of instances here is 610.
– API solving, these are 10 problems where a rather large API (set of functions with types) is given, and the problem is to construct a new function of a given type. Each problem has variants with API sizes that vary in size from a dozen to a few thousand functions. These problems were constructed by the authors in an attempt to create practically useful problems. The total number of instances here is 35.

The total number of benchmark problems we used is 1200. We did not have access to these benchmarks when we developed our tool.

*Experimental Set-up.* The experiments were run on a 2013 laptop computer with an Intel Xeon E3-1200 v2 processor and the processes were allowed 7 GB of RAM. IntHistGC was run with its best flags (`-b -c -c3`). We used the latest versions of the tools: fCube version 11.1 and IntHistGC 1.29. We used a 300 s timeout.

*Results.* All three tools eventually solve a good portion of the benchmarks: our tool `intuit` solved all but 37 benchmarks, fCube solved all but 38 benchmarks, and IntHistGC all but 39. More interesting is to compare the running times. We compare our tool `intuit` against IntHistGC and fCube for provable problems (`Valid`) and unprovable problems (`CoSat`) in Fig. 6. All time axes are logarithmic.

We can see that `intuit` outperforms both IntHistGC and fCube significantly on virtually all provable problems. The comparison for unprovable problems is in favor of `intuit` as well, although there are a few outliers demanding further scrutiny. We show a table of "interesting" problems (problems that were out of reach for at least one tool but not for all tools) in Fig. 7.

The problems `cross2x` – `mapf3x` are all instances of the API benchmark suite. They contain relatively large sets of axioms, of which typically only a few are needed for a proof, which is representative for automatically generated verification problems in general. Our tool `intuit` does well on these.

The problems `SYJ202` are pigeon-hole problems. In fact, after generating clauses in the fashion described in this paper, there were no implication clauses generated, and thus the problems became purely classical! No surprise that `intuit` does well on these, given that it uses a state-of-the-art SAT-solver internally.

The problems that `intuit` struggled with (and which are the outliers in the scatterplots in Fig. 6) are all instances of `SYJ212`. These are problems that are all counter-satisfiable, and consist of long chains of nested equivalences. Interestingly, on these problems, the running time of `intuit` was not increasing as a function of the size of the problem. Hence there are some gaps in the figure: only the instances where `intuit` times out are shown. The reason for this is not

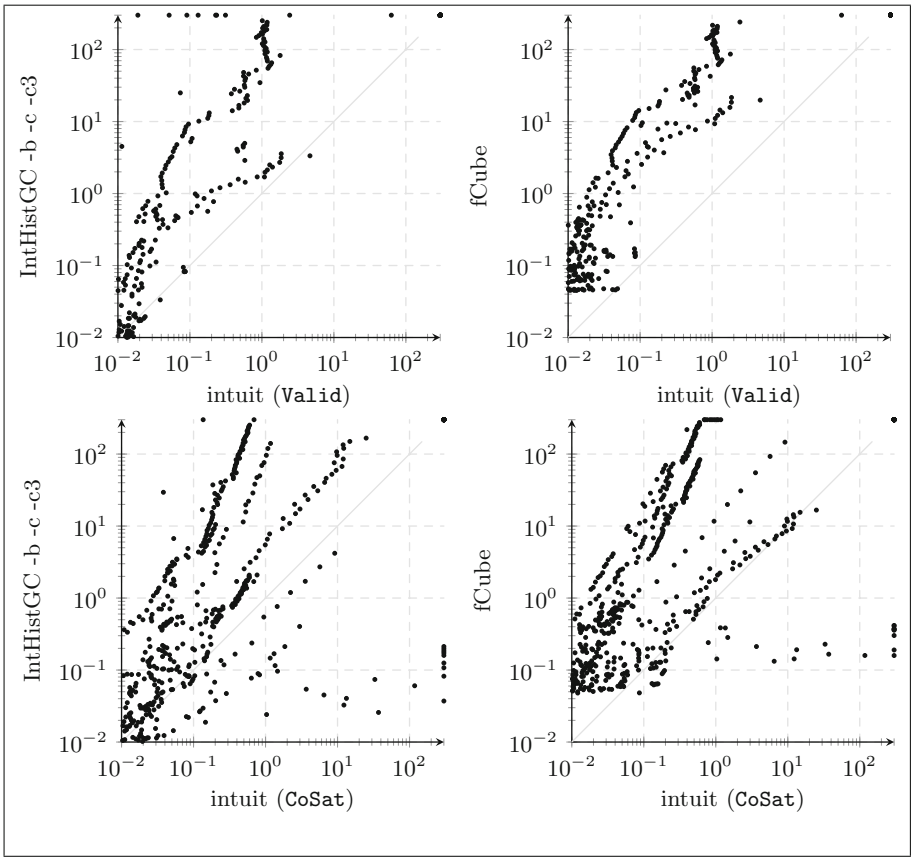


**Fig. 6.** Comparison. Runtimes in seconds, timeout was 300 s.

completely obvious to us; it seems that `intuit` seems particularly sensitive to exactly what models are found by the SAT-solver on these problems. Sometimes, the models indicate just the right implications to check, leading to intuitionistic models quickly, and other times, the models lead the prover astray, and it takes a long time.

## 5 Future Work

The main reason for initiating the work described in this paper was to understand how to build a scalable prover for a logic like intuitionistic propositional logic. However, the number of practical applications of intuitionistic propositional logic is limited. We believe that our insights can benefit two other kinds of logic that have more applications: classical modal logics and intuitionistic first-order logic.

| Benchmark | Our Tool | IntHistGC | fCube | Status |
|---|---|---|---|---|
| cross2x | 0.05 | – | 0.69 | Valid |
| cross3x | 0.13 | – | 4.39 | Valid |
| jm_cross3x | 0.13 | – | 6.84 | CoSat |
| jm_lift2x | 0.31 | – | 9.38 | Valid |
| lift2x | 0.23 | – | 9.50 | Valid |
| mapf3x | 0.69 | – | – | CoSat |
| SYJ202+1.006 | 0.01 | – | 0.67 | Valid |
| SYJ202+1.007 | 0.09 | – | 3.61 | Valid |
| SYJ202+1.008 | 0.22 | – | 27.47 | Valid |
| SYJ202+1.009 | 2.43 | – | 241.06 | Valid |
| SYJ202+1.010 | 62.79 | – | – | Valid |
| SYJ208+1.031 | 0.70 | 38.22 | – | CoSat |
| SYJ208+1.032 | 0.75 | 47.12 | – | CoSat |
| SYJ208+1.033 | 0.83 | 49.82 | – | CoSat |
| SYJ208+1.034 | 0.88 | 69.35 | – | CoSat |
| SYJ208+1.035 | 0.95 | 76.12 | – | CoSat |
| SYJ208+1.036 | 1.02 | 96.03 | – | CoSat |
| SYJ208+1.037 | 1.07 | 119.56 | – | CoSat |
| SYJ208+1.038 | 1.17 | 140.49 | – | CoSat |
| SYJ212+1.030 | – | 0.03 | 0.19 | CoSat |
| SYJ212+1.038 | – | 0.08 | 0.15 | CoSat |
| SYJ212+1.041 | – | 0.10 | 0.37 | CoSat |
| SYJ212+1.043 | – | 0.12 | 0.30 | CoSat |
| SYJ212+1.046 | – | 0.15 | 0.36 | CoSat |
| SYJ212+1.047 | – | 0.17 | 0.41 | CoSat |
| SYJ212+1.048 | – | 0.18 | 0.35 | CoSat |
| SYJ212+1.049 | – | 0.19 | 0.41 | CoSat |
| SYJ212+1.050 | – | 0.21 | 0.37 | CoSat |

**Fig. 7.** Interesting problems (where one tool failed but not all). Runtimes are in seconds. A hyphen indicates that the time limit of 300 s was exceeded.

*Generalization to Modal Logic.* We are currently building a theorem prover for classical modal logic based on the same ideas as presented in this paper. When we want to prove a formula $A$, we generate a fresh literal $q$, and generate constraints that represent the formula $\Box(A \to q)$. The insight is that, in order to do so, it is enough to consider constraints of one of the following three shapes:

1. $\Box p$, for a propositional logic formula $p$,
2. $\Box(a \to \Box b)$, for propositional logic literals $a$ and $b$, and
3. $\Box(a \to \Diamond b)$, for propositional logic literals $a$ and $b$.

Just like the prover we have described in this paper, the theorem prover for modal logic uses one SAT-solver that stores all constraints that hold in all worlds, which are the formulas $p$ above. If $q$ can be proven from these, the proof is done. Otherwise, we get a counter-model which satisfies all $p$ but not $q$, and we have to investigate whether or not constraints of type 2. and 3. are fulfilled. If the answer is yes, we have found a counter model to $A$, otherwise, more constraints of type 1. will be generated and we start over.

*Generalization to Intuitionistic First-Order Logic.* We also have started to look at building an automated prover for intuitionistic first-order logic. There are two strands of work here. The first basically augments a standard SMT-solver with implication clauses. Most SMT-solvers have a heuristic for instantiating universal quantifiers. After that, the implication clauses are dealt with in the same way as in this paper.

However, this can only deal with a fragment of first-order logic. To cope with the full logic, we analyze intuitionistic first-order logic and come to the conclusion that we only have to support "clauses" of the following three shapes:

1. $\forall\, \boldsymbol{x}.(A_1 \wedge .. \wedge A_n) \rightarrow (B_1 \vee .. \vee B_m)$, also called flat clauses,
2. $\forall\, \boldsymbol{x}.(A \rightarrow B) \rightarrow C$, also called implication clauses, and
3. $\forall\, \boldsymbol{x}.(\forall\, \boldsymbol{y}.A) \rightarrow B$, called quantification clauses.

The idea is to let a model-generating first-order prover (SMT-solver) take care of the flat clauses. Every so often, we investigate the (partial) models that are generated, and use the implication clauses and the quantification clauses to generate more flat clauses, and we continue.

The main difficulties here are: (1) first-order logic is only semi-decidable, so we cannot expect to get either a model or a proof from the set of flat clauses, which means that we have to settle for a heuristic method based on partial models, and (2) most representations of partial models use some kind of ground model which makes it hard to deal with the universally quantified variables $\boldsymbol{x}$ in the implication clauses and quantification clauses.

## 6 Conclusions

We presented a new method for automated proving and disproving of formulas in intuitionistic propositional logic. The method makes use of a single instance of an incremental SAT-solver, and implements an SMT-style theory of intuitionistic implication clauses on top of it. The result is a robust theorem prover that can easily tackle most existing benchmarks.

Intuitionistic propositional logic seems to have limited practical applications, as indicated by the (un)availability of standard benchmark sets. Our hope is that the method described in this paper can give rise to scalable and robust methods for related logics with more clear practical applications, such as various modal logics and intuitionistic first-order logic.

**Acknowledgments.** We thank Thierry Coquand and Rajeev Gore for feedback on earlier versions of this work.

## References

1. Barr, M.: Toposes without points. J. Pure Appl. Algebra **5**(3), 265–280 (1974)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

3. Ferrari, M., Fiorentini, C., Fiorino, G.: fCube: an efficient prover for intuitionistic propositional logic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 294–301. Springer, Heidelberg (2010)
4. Goré, R., Thomson, J., Wu, J.: A history-based theorem prover for intuitionistic propositional logic using global caching: IntHistGC system description. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 262–268. Springer, Heidelberg (2014)
5. Raths, T., Otten, J., Kreitz, C.: The ILTP problem library for intuitionistic logic - release v1.1. J. Autom. Reasoning (2006)
6. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning. Symbolic Computation, pp. 466–483. Springer, Heidelberg (1983)