A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover

K. Rustan M. Leino [0], Madan Musuvathi [0], and Xinming Ou [1]

[0] Microsoft Research, Redmond, WA, USA
{leino,madanm}@microsoft.com
[1] Princeton University, Princeton, NJ, USA
xou@cs.princeton.edu

Manuscript XO 0 / KRML 151, 15 October 2004.

Abstract. Lazy proof explication is a theorem-proving architecture that allows a combination of Nelson-Oppen-style decision procedures to leverage a SAT solver's ability to perform propositional reasoning efficiently. The SAT solver finds ways to satisfy a given formula propositionally, while the various decision procedures perform theory reasoning to block propositionally satisfied instances that are not consistent with the theories. Supporting quantifiers in this architecture poses a challenge as quantifier instantiations can dynamically introduce boolean structure in the formula, requiring tighter interleaving between propositional and theory reasoning.

This paper proposes handling quantifiers by using two SAT solvers, thereby separating the propositional reasoning of the input formula from that of the instantiated formulas. This technique can then reduce the propositional search space, with an eye toward improving performance. The technique can use off-the-shelf SAT solvers and requires only that the theories are checkpointable.

0 Introduction

Automatic verification of hardware and software systems requires a good decision procedure for the conditions to be verified. Verification conditions generated for the static analysis and verification of software reference functions and predicates for many types of values, including those of the source programming language. Designing decision procedures for these individual theories may be easier than designing a decision procedure that handles all of them. Nelson and Oppen [9, 8] developed a famous method for combining decision procedures of a class of first-order theories. Because of its modular architecture, theorem provers based on this method can readily support many interesting theories that are useful in practice. Many theorem provers are based on such combinations, for example Simplify [2], Verifun [4], ICS [1], and CVC Lite [0], and these have been applied to the verification of hardware and software systems.

Software verification conditions also involve quantified formulas. For example, the verifications conditions generated by the program checker ESC/Java [6] use quantified formulas in several ways: (0) to specify a partial set of properties of otherwise uninterpreted functions, (1) to axiomatize properties guaranteed by Java and its type system, (2) to describe a procedure call's effect on the program heap, (3) to state object invariants for all objects of a class, and (4) to support quantifiers, usually over array elements, supplied by the user. Unfortunately, the Nelson-Oppen combination method is applicable only to quantifer-free first-order formulas, so quantifiers in that setting cannot be handled as an ordinary theory, but instead need special support. Another problem is that any theorem prover is necessarily incomplete for quantified formulas, but the prevalence of quantified formulas in important problems demands that theorem provers handle them effectively in practice.

The Simplify theorem prover [2] provides support for quantified formulas that has been shown to be effective for software verification applications, for example in extended static checking [3, 6]. Simplify uses a kind of pattern matching of ground terms to trigger the instantiation of universally quantified formulas. However, Simplify does not handle propositional search very efficiently. A new generation of theorem provers, including Verifun [4], ICS [1], and CVC Lite [0], attempt to speed up the propositional search by leveraging the last decade's advances in SAT solving and using a *lazy proof-explication* architecture. In such an architecture, a Nelson-Oppen combination of decision procedures interact with an off-the-shelf SAT solver: the SAT solver finds ways to satisfy a given formula propositionally, while the combination of other decision procedures performs theory reasoning to block propositionally satisfied instances that are not consistent with the theories.

To use such a new-generation theorem prover in software verification applications, we seek to incorporate support for quantified formulas in the lazy-proof-explication architecture. This poses the following key challenges. First, quantified formulas typically involve propositional connectives. As a result, quantifier instantiations performed during theory reasoning can dynamically introduce boolean structure in the formula. This requires tighter interleaving between propositional and theory reasoning. Second, most quantifier instantiations are not useful in proving the validity of the formula. Blindly exposing such redundant instantiations to the SAT solver could drastically reduce the performance of the propositional search.

The first support for quantified formulas in a lazy-proof-explication prover has been incorporated into Verifun [5]. When the quantifier instantiations result in formulas with propositional structure, Verifun augments the original formula with such instantiations so that the SAT solver can find ways to satisfy these instantiations in the context of the original formula. However, the added disjunctions then persist in the prover's state.

As an alternative approach, we propose a *two-tier* technique in this paper. This technique involves two off-the-shelf SAT solvers, a *main* solver that performs the propositional reasoning of the input formula, and a *little* solver that reasons over the quantifier instantiations. When the main SAT solver produces a propositionally satisfying instance that is consistent with the decision procedures, a pattern matching algorithm, similar to the one in Simplify, generates a set of quantifier instantiations. The little SAT solver, along with the decision procedures, tries to falsify the satisfying instance with the instantiations produced. If successful, the little SAT solver generates a blocking clause that only contains literals from the input formula. By thus separating the propositional reasoning of the input formula from that of the instantiated formulas, this technique reduces the propositional search space, with an eye toward improving performance.

Section 1 introduces some preliminaries and reviews the architecture of theorem provers based on lazy proof explication. Section 2 discusses the main problem in handling quantifiers in lazy-proof-explication theorem provers. The quantifier algorithm is presented in Section 3. We trace through an example in Section 4. The final sections offer a discussion, some related work, and a conclusion.

1 Theorem proving using lazy proof explication

In this section, we review in more detail the architecture and main algorithm of a theorem prover based on lazy proof explication.

1.0 Terminology

A *formula* is constructed by an arbitrary combination of function and predicate symbols, propositional connectives and quantifier bindings. The following is an example formula:

$$\begin{array}{l}(\forall\, a, i, v \bullet\ 0 \leqslant i \wedge i < \mathit{Length}(a) \ \Rightarrow\ \mathit{read}(\mathit{write}(a, i, v), i) = v\ ) \wedge \\ \mathit{Length}(b) > 0 \\ \ \Rightarrow\ \mathit{read}(\mathit{write}(b, 0, 10), 0) = 10\end{array}$$

An *atomic formula* is a subformula that does not start with a propositional connective and is not inside another atomic formula. Propositional connectives include conjunction, ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\Rightarrow$). For example, the following are all atomic formulas:

$$\begin{array}{l}(\forall\, a, i, v \bullet\ 0 \leqslant i \wedge i < \mathit{Length}(a) \ \Rightarrow\ \mathit{read}(\mathit{write}(a, i, v), i) = v\ ), \\ \mathit{Length}(b) > 0, \\ \mathit{read}(\mathit{write}(b, 0, 10), 0) = 10.\end{array}$$

A *literal* is either an atomic formula or its negation. A *monome* is a set of literals. If $\mathcal{P}$ is a set of formulas, we sometimes write just $\mathcal{P}$ when we mean the conjunction of the formulas in $\mathcal{P}$.

A theorem prover can be equivalently viewed either as a validity checker or a satisfiability checker: establishing the validity of a given conjecture P is equivalent to finding a satisfying assignment to $\neg P$. For the theorem provers discussed in this paper, we take the second view, thinking of the input as a formula (the negation of a conjecture) to be satisfied or shown unsatisfiable.

Definition 0. *A* tautology *for formula F is another formula T such that $F \wedge T$ and F are equisatisfiable.*

Proposition 0 *If both T and U are tautologies for F, then $T \wedge U$ is also a tautology for F.*

Proposition 1 *If T is a tautology for F and $T \Rightarrow U$, then U is also a tautology for F.*

1.1 Lazy proof explication

In a lazy-proof-explication theorem prover, an off-the-shelf SAT solver conducts propositional reasoning. Each atomic formula is treated as an opaque propositional variable by the SAT solver and assigned a truth value. When the SAT solver returns a truth value assignment that makes the whole formula propositionally satisfiable, decision procedures are invoked to perform theory reasoning to determine if the truth value assignment (a monome) is consistent with all the underlying theories. If so, the input formula is satisfiable. Otherwise, the theories are responsible for explicating a proof that shows the monome is not satisfiable. The proof must be a tautology and is then conjoined to the original formula, pruning the SAT solver's search space by blocking the inconsistent truth value assignment.

For example, suppose a theorem prover is asked about the satisfiability of the following formula:

$$(\llbracket x \leqslant y \rrbracket \vee \llbracket y = 5 \rrbracket) \;\wedge\; (\llbracket x < 0 \rrbracket \vee \llbracket y \leqslant x \rrbracket) \;\wedge\; \neg\llbracket x = y \rrbracket$$

where for clarity we have enclosed each atomic formula within special brackets. As (the propositional projection of) this formula is passed to the SAT solver, the SAT solver may return a monome containing the following three literals (corresponding to the truth value assignement to three atomic formulas),

$$\llbracket x \leqslant y \rrbracket,\;\; \llbracket y \leqslant x \rrbracket,\;\; \neg\llbracket x = y \rrbracket \tag{0}$$

This monome is then passed to the theories, where the theory of arithmetic would detect an inconsistency and return the following proof:

$$\llbracket x \leqslant y \rrbracket \wedge \llbracket y \leqslant x \rrbracket \;\;\Rightarrow\;\; \llbracket x = y \rrbracket \tag{1}$$

By conjoining this tautology to the original formula, the propositional assignment (0) is explicitly ruled out in the further reasoning performed by the SAT solver. Since (1) is a tautology, it could have been generated and conjoined to the input even before the first invocation of the SAT solver, but the strategy of generating this lemma on demand—that is, lazily—is the reason the architecture is called *lazy* proof explication.

Figure 0 outlines the algorithm of a theorem prover using lazy proof explication. We write $PSat(F)$ to denote the propositional satisfiability of formula F. It is implemented by calling an off-the-shelf SAT solver (after projecting the atomic formulas onto propositional variables). If the result is $True$, a monome m is returned as the satisfying assignment. Then $CheckMonome$ is called to determine if m is consistent with all the underlying theories. The signature-disjoint theories are combined using the Nelson-Oppen method and cooperate by propagating equalities. $CheckMonome(m)$ returns a set of explicated proofs that are sufficient to refute monome m. An empty set indicates that the theories are unable to detect any inconsistency, in which case the original formula really is satisfiable. Otherwise, the explicated proofs are conjoined to F and the loop continues until either the formula becomes propositionally unsatisfiable, or the theories are unable to find inconsistency in the monome returned by $PSat$.

Once the SAT solver returns a monome, the inconsistency can be discovered without any propositional reasoning. This enables a clear interface between the propositional reasoning and theory reasoning.

```
Input: formula $F$
Output: satisfiability of $F$
while $(PSat(\mathrm{F}))$ {
  let monome $m$ be the satisfying assignment ;
  $\mathcal{P} := CheckMonome(m)$ ;
  if $(\mathcal{P} = \emptyset)$ {
    return $True$ ;
  } else {
    $F := F \wedge \mathcal{P}$ ;
  }
}
return $False$ ;
```

Fig. 0. Lazy-proof-explication algorithm without support for quantifiers.

2 Handling Quantifiers

When a formula contains quantifiers, usually the information expressed by the quantifiers must be used in showing a formula is unsatisfiable. This section discusses some basic notations and challenges for handling quantifiers. The main quantifier algorithm is presented in Section 3.

2.0 Terminology

A quantified formula has the form $(\delta\, x \bullet\ F\ )$, where δ is either $\forall$ or $\exists$. Quantifiers can be arbitrarily nested. Provided all the bound variables have been suitably α-renamed, the following three equations hold:

$$\begin{aligned} \neg(\delta\, x \bullet\ F\ ) &\equiv (\bar{\delta}\, x \bullet\ \neg F\ ) \\ (\delta\, x \bullet\ F\ ) \wedge G &\equiv (\delta\, x \bullet\ F \wedge G\ ) \\ (\delta\, x \bullet\ F\ ) \vee G &\equiv (\delta\, x \bullet\ F \vee G\ ) \end{aligned}$$

Here $\bar{\forall} = \exists$ and $\bar{\exists} = \forall$. By repeatedly applying the above three equations, we can move all the quantifiers in a formula to the front and convert it to the prenex form $(\delta_1\, x_1 \bullet\ (\delta_2\, x_2 \bullet\ \ldots\ (\delta_n\, x_n \bullet\ F\ )))$, where F does not contain any quantifier.

The existentially bound variables in the prenex form can be eliminated by skolemization. Skolemization replaces each existential variable x in the quantified body with a term $K(\Psi)$, where K is a fresh function symbol that is unique to the quantified formula and the existential variable x, and Ψ is the list of universally bound variables that appear before x. The skolem term $K(\Psi)$ is interpreted as the “existing term” determined by Ψ. We say the resulting purely universal formula is in *canonical form*. We use $Canon(Q)$ to denote the canonical form of a formula Q.

A *quantifier atomic formula* is an atomic formula that starts with a quantifier. A *quantifier literal* is either a quantifier atomic formula or its negation.

For any quantified formula C in canonical form and any substitution θ that maps each universal variable to a ground term, we write $C[\theta]$ to denote the formula gotten by taking C's body and substituting θ into it.

2.1 Challenges in handling quantifiers

In order to reason about quantifiers, one needs to instantiate the universal variables with some concrete terms. This will introduce new facts that contain boolean structures, which cannot be directly used in the theory reasoning to refute the current monome. Neither can one only rely on propositional reasoning to handle these new facts because some inconsistency has to be determined by the theories. This means that in order to reason about quantifiers, both propositional reasoning and theory reasoning are necessary. This poses a challenge to theorem provers with lazy proof explication, where the two are clearly separated.

One simple approach is to conjoin the original formula with tautologies of instantiating universal quantifiers. Let Q be a quantifier literal in a formula F and let θ be a substitution that maps each universal variable in $Canon(Q)$ to a concrete term. Then, the following is a tautology for F:

$$Q \Rightarrow Canon(Q)[\theta]$$

Intuitively, it is a tautology because both $Q \Rightarrow Canon(Q)$ and $Canon(Q) \Rightarrow Canon(Q)[\theta]$ are tautologies. The latter implication is obviously a tautology. The first implication is a tautology because each existential variable x is assigned a fresh skolemizer K that does not coincide with any other symbol in the formula. This way it expresses exactly the meaning of "some existing term", with respect to the orginal formula.

Conjoining these tautologies puts more constraints on the original formula. If the instantiations are properly chosen, more inconsistencies can be detected and eventually the formula can be shown to be unsatisfiable. Simplify [2] uses a matching heuristic to return a set of instantiations that will likely be useful in refuting the formula. However, there may still be too many (sometimes infinite number of) useless instantiations returned by the matcher. This may blow up the SAT solver because those tautologies often introduce new atomic formulas, adding more case splits.

The quantifier algorithm presented in this paper adopts a different approach. First, the matching heuristic in Simplify is still used to return likely-useful instantiations. But a second SAT solver (the little solver) performs the propositional reasoning for those instantiated formulas. During the reasoning process many new instantiations are generated, but only some of them are relevant in refuting the monome. Once the monome is refuted, we show how to construct a proof that not only involves the theories but also those relevant instantiations. The rationale of using a second SAT solver is that the propositional reasoning for finding a satisfying monome can be separated from the propositional reasoning for refuting a monome. Once a monome is refuted, many of the instantiations are not useful anymore. Without this two-tier approach they would remain in the formula and introduce many unnecssary case splits in the future rounds of reasoning.

3 Quantifier algorithm

The quantifier reasoning is performed in the $CheckMonome$ function in the algorithm shown in Fig 0. We show the quantifier algorithm in two steps. In section 3.0, we present the simple one mentioned in section 2.1. In section 3.1, we show how to use the little SAT solver in *CheckMonome* to perform both propositional and theory reasoning.

3.0 Simple quantifier algorithm

```
Input: Monome $m$
Output: a set of proofs $\mathcal{P}$
Assert $m$ to the theories ;
**if** ($m$ is consistent with all the theories) {
   Generates tautologies $\mathcal{P}$ by instantiating universal variables ;
} **else** {
   Theories output proofs $\mathcal{P}$ ;
}
**return** $\mathcal{P}$ ;
```

Fig. 1. *CheckMonome* algorithm with simple support for quantifiers.

Fig. 1 shows the *CheckMonome* algorithm with the simple quantifier support discussed in section 2.1. The quantifier module is invoked only when the other theories cannot detect any inconsistency in the given monome. As discussed in section 2.1, the tautologies are generated by instantiating universal variables. The instantiations are returned from a matching algorithm similar to that of Simplify. To avoid generating duplicate instantiations, the quantifier module remembers the instantiations it has produced. When no more tautologies can be generated, *CheckMonome* will return an empty set, in which case the theorem prover algorithm in Fig. 0 will terminiate with the result of *True*. Since for some formulas there may be an infinite number of instantiations, we put a upper limit on the number of times the quantifier module can be called for each run of the theorem prover and simply return an empty set when the limit is exceeded.

Unlike the proofs output by the theories after discovering an inconsistency, the tautologies generated by the quantifier module generally are not guaranteed to refute the monome. There are two reasons for this. First, many inconsistencies involve both quantifier reasoning and theory reasoning. Without cooperating with the other theories, the tautologies returned by instantiating quantifiers alone may not be sufficient to propositionally block the monome. Second, the instantiations returned by the matching algorithm depend on the monome. Since instantiating quantifiers may produce more atomic formulas to appear in a monome, it is possible that the matcher can provide the "right" instantiation only after several rounds.

As a result of *CheckMonome* returning a set of tautologies insufficient to refute the monome, the next round may call *CheckMonome* with the same monome, plus some extra literals coming from the quantifier instantiation. This is certainly undesirable, because the SAT solver then repeats its work to find the same monome again. A more serious problem of this simple algorithm is that many of the returned tautologies are not even relevant in refuting the monome. Those useless tautologies remain in the formula during the proving process, and without removing them, the SAT solver will eventually be overwhelmed by many unnecessary case splits.

3.1 The two-tier quantifier algorithm

In order to use quantifier instantiations to refute a monome, propositional reasoning is needed. The key problem of the simple $\mathit{CheckMonome}$ algorithm is, by directly returning the tautologies generated from quantifiers, it essentially relies on the main SAT solver to perform the propositional reasoning of those newly generated formulas. This causes repetitive and unnecessary work in the main SAT solver. To address this problem, we separate the propositional reasoning of the instantiated formula from that of the original formula by using a little SAT solver in our $\mathit{CheckMonome}$ algorithm (Fig. 2).

```
Input: Monome $m$
Output: a set of proofs $\mathcal{P}$
Assert $m$ to theories ;
Checkpoint all theories ;
$\mathcal{P} := \emptyset$ ;
**loop** {
  **if** (Theories say consistent) {
    Quantifier module generates new tautologies $\mathcal{P}_0$ ;
    **if** ($\mathcal{P}_0 = \emptyset$) {
      **return** $\emptyset$ ;
    }
  } **else** {
    Theories explicate proof $\mathcal{P}_0$ ;
  }
  $\mathcal{P} := \mathcal{P} \cup \mathcal{P}_0$ ;
  **if** ($\mathit{PSat}(m \wedge \mathcal{P}) = \mathit{False}$) {
    **return** $\mathit{FindUnSATCore}(m, \mathcal{P})$ ;
  }
  **let** $m \cup m'$ be the satisfying monome ;
  Restore checkpoints in all the theories ;
  Assert $m'$ to theories ;
}
```

Fig. 2. The $\mathit{CheckMonome}$ algorithm using the little SAT solver.

When the quantifier module generates some tautologies in $\mathcal{P}$, the little SAT solver performs propositional reasoning $m \wedge \mathcal{P}$. For every satisfying assignment $m \wedge m'$ produced by the little SAT solver, the algorithm invokes the theories to check if the assignment is consistent. To avoid redundant work, the algorithm checkpoints the theory state before invoking the little solver. As a result, the algorithm only needs to assert m' to the theories in each iteration of the loop in figure 2. If the theories detect an inconsistency, they block m' by adding an explicated proof into $\mathcal{P}$.

The loop continues as long as $\mathcal{P}$ does not propositionally refute m and the quantifier module generates new facts in each iteration. If no more instantiation can be generated but $m \wedge \mathcal{P}$ is still satisfiable, the algorithm terminates and returns an empty set, indicating failure to refute monome m.

Once $\mathcal{P}$ can refute m, the function $\mathit{FindUnSATCore}$ is called to extract a good-quality proof from $\mathcal{P}$. $\mathit{FindUnSATCore}(F, G)$ requires that $F \wedge G$ is propositionally unsatisfiable. It returns a formula H such that $G \Rightarrow H$, $F \wedge H$ is still unsatisfiable, and the atomic formulas in H all occur in F. The proof generated by $\mathit{FindUnSATCore}(m, \mathcal{P})$ has the following characteristics:

0. The proof is sufficient to propositionally refute m.
1. The proof contains only atomic formulas appearing in m.

Returning such a proof always reduces the propositional search space for the original formula. $\mathit{FindUnSATCore}$ can be implemented in various ways. Modern SAT solvers can extract a small unsatisfiable core of a propositional formula [10] and this seems to be useful in $\mathit{FindUnSATCore}$. Alternatively, interpolants [7] may also be used here, because any interpolant of G and F would satisfy the specification of $\mathit{FindUnSATCore}(F, G)$. For our preliminary experiments, we have the following naive (and probably inefficient) implementation of the $\mathit{FindUnSATCore}$ function for the particular kind of arguments that show up in the algorithm:

For a monome m and a formula P, if $\mathit{PSat}(m \wedge P) = \mathit{False}$, then there exists a minimal subset m_0 of m such that $\mathit{PSat}(m_0 \wedge P) = \mathit{False}$. Such a m_0 can be gotten by trying to take out one literal from m at a time and discard the literal if the formula remains propositionally unsatisfiable. It is easy to see that $P \Rightarrow \neg m_0$. We just return $\neg m_0$ as the result of $\mathit{FindUnSATCore}(m, P)$.

Algorithm properties The correctness of the algorithm hinges on the fact that every formula in $\mathcal{P}$ is a tautology for the monome m. The correctness of the algorithm is formalized as the following theorem.

Theorem 1 *Let $\mathcal{P}$ be the set of all tautologies generated during the run of the algorithm. Then, the algorithm refutes the monome m iff $\mathit{Sat}(m \wedge \mathcal{P}) = \mathit{False}$.*

Here, we use $\mathit{Sat}(F)$ to denote the satisfiability of F, taking into account all the domain theories but not quantifiers. We use $\mathit{QSat}(F)$ to denote the satisfiability of F taking into account all the theories and quantifiers. Intuitively, the result of the algorithm is the same as if we had generated all the tautologies $\mathcal{P}$ up front and run a standard Nelson-Oppen theorem prover on the formula $m \wedge \mathcal{P}$. Since conjoining tautologies does not change the satisfiability of a formula, the theorem shows our algorithm to be sound:

Corollary 2 *If the algorithm refutes m, then $\mathit{QSat}(m) = \mathit{False}$.*

This is due to the fact that $\mathit{QSat}(F) \Rightarrow \mathit{Sat}(F)$. On the other hand, the algorithm is not complete, since we cannot always generate all the tautologies relevant to the formula. When the algorithm fails to refute a monome, the best thing we know is $\mathit{Sat}(m \wedge \mathcal{P}) = \mathit{True}$, that is, even with all the information in $\mathcal{P}$, a Nelson-Oppen theorem prover cannot refute the monome either.

4 Example

In this section, we demonstrate how our algorithm works on a small example.

Let P and Q be two quantified formulas:

$$\begin{array}{ll} P: & (\forall x \bullet\ x < 10 \Rightarrow R(f(x))\ ) \\ Q: & (\forall y \bullet\ R(f(y)) \Rightarrow S(g(y))\ ) \end{array}$$

where the match patterns to be used for P and Q are x: $f(x)$ and y: $g(y)$, respectively. These patterns may have been specified by the user or may have been inferred by the theorem prover. We now trace our algorithm through the request of determining whether or not the following formula is satisfiable:

$$[\![P]\!] \wedge [\![Q]\!] \wedge ([\![b = 1]\!] \vee [\![b = 2]\!]) \wedge \neg[\![S(f(b))]\!] \wedge \neg[\![S(g(0)]\!])$$

In the first round, the main SAT solver returns a monome m, say

$$\{\ [\![P]\!],\ [\![Q]\!],\ [\![b = 1]\!],\ \neg[\![S(f(b)]\!]),\ \neg[\![S(g(0)]\!])\ \}$$

Since no theory can detect inconsistency, the quantifier module is invoked to generate tautologies. According to the match pattern, x is instantiated with b in P and y is instantiated with 0 in Q:

$$[\![P]\!] \Rightarrow ([\![b < 10]\!] \Rightarrow [\![R(f(b))]\!]) \quad (2)$$
$$[\![Q]\!] \Rightarrow ([\![R(f(0))]\!] \Rightarrow [\![S(g(0))]\!]) \quad (3)$$

The tautologies (2) and (3) are conjoined to the monome and the little SAT solver is called. The extended monome m' for the newly-introduced atomic formulas might be:

$$\{\ \neg[\![b < 10]\!],\ \neg[\![R(f(0))]\!]\ \}$$

At this point the theories detect an inconsistency between $[\![b = 1]\!]$ and $\neg[\![b < 10]\!]$. So a new tautology is added:

$$[\![b = 1]\!] \wedge \neg[\![b < 10]\!] \Rightarrow \mathit{False} \quad (4)$$

In the next iteration, m' is

$$\{\ [\![R(f(b))]\!],\ \neg[\![R(f(0))]\!]\ \}$$

The theories are unable to detect inconsistency in the monome $m \wedge m'$. The quantifier module is invoked again to generate tautologies. This time the term $f(0)$ in the newly generated formulas matches the pattern, so x in P is instantiated by 0.

$$[\![P]\!] \Rightarrow ([\![0 < 10]\!] \Rightarrow [\![R(f(0))]\!]) \quad (5)$$

The next m' is

$$\{\ [\![R(f(b))]\!],\ \neg[\![R(f(0))]\!],\ \neg(0 < 10)\ \}$$

The theory then detects an inconsistency:

$$\neg(0 < 10) \Rightarrow \mathit{False} \quad (6)$$

After conjoining (6), the original monome m will be propositional refuted. The proof constructed is

$$[\![P]\!] \wedge [\![Q]\!] \wedge \neg [\![S(g(0))]\!] \;\Rightarrow\; \mathit{False}$$

After conjoining this proof to the original formula, it becomes propositionally unsatisfiable.

If we use the simple algorithm, tautology (2) would be conjoined to the input formula, even though it has nothing to do with the contradiction. In the subsequent solving, this unnecessary tautology would introduce a case split on $([\![b = 1]\!] \vee [\![b = 2]\!])$. The theories would have to consider both in order to block the truth value assignment $\neg[\![b < 10]\!]$. By separating the two SAT solvers, our algorithm only needs to consider one of them.

5 Discussion

The quantifier algorithm presented in the last section is in the setting of a theorem prover using lazy proof explication. However, for a theorem prover that calls decision procedures eagerly, such as Simplify and CVC Lite, the problem of useless instantiations also exists. Without properly handling, this will eventually blow up the propositional search. Simplify adopts a simple heuristic that gives atomic formulas generated by instantiating quantifiers a lower priority in case splits. It is not quite clear how effective this approach is. It is possible that the two-tier approach can also be incorporated into a Simplify-like theorem prover, although the close coupling of propositional and theory reasonings would make it more complicated.

The $\mathit{FindUnSATCore}$ function in this paper always returns a formula containing only original atomic formulas. However, this is not always the best thing to do. For example, suppose a theorem prover is asked about the satisfiability of the following formula:

$$[\![(\,\forall x \bullet\; P(x) \;\Rightarrow\; x \leqslant a\,)]\!] \;\wedge\; [\![P(2)]\!] \;\wedge\; ([\![a = 0]\!] \vee [\![a = 1]\!])$$

If the SAT solver first picks a satisfying assignment consisting of the first two conjuncts and the disjunct $[\![a = 0]\!]$, then our algorithm would generate the proof

$$[\![(\,\forall x \bullet\; P(x) \;\Rightarrow\; x \leqslant a\,)]\!] \wedge [\![P(2)]\!] \;\Rightarrow\; \neg[\![a = 0]\!]$$

Then, leading to the next round of theory reasoning, the SAT solver would pick the satisfying assignment that instead selects the other disjunct, $[\![a = 1]\!]$. This will cause the generation of the proof

$$[\![(\,\forall x \bullet\; P(x) \;\Rightarrow\; x \leqslant a\,)]\!] \wedge [\![P(2)]\!] \;\Rightarrow\; \neg[\![a = 1]\!]$$

after which the SAT solver can determine the given formula to be unsatisfiable. But by instead returning the proof

$$[\![(\,\forall x \bullet\; P(x) \;\Rightarrow\; x \leqslant a\,)]\!] \wedge [\![P(2)]\!] \;\Rightarrow\; [\![2 \leqslant a]\!]$$

in response to the first satisfying assignment, the second round of theory reasoning does not need to instantiate any quantifiers.

For efficiency, it is best if theories combined using Nelson-Oppen are *convex*. Informally, a convex theory will never infer a disjunction of equalities without inferring one of them. Thus the decision procedures only need to propagate single equalities. For non-convex theories, sometimes it is necessary for the decision procedure to propagate a disjunction of equalities. For example, the integer arithmetic theory can infer the following fact:

$$0 \leqslant x \wedge x \leqslant 1 \quad \Rightarrow \quad x = 0 \vee x = 1.$$

This fact should be added as a tautology in the proving process. Like the tautologies generated by quantifier instantiation, there is a risk that useless tautologies increase the work required of the propositional search. The same technique discussed in this paper is readily applied to those non-convex theories. In this sense, our algorithm in Fig. 2 actually provides a unified approach to handle both quantifiers and non-convex theories—they can both be viewed as a theory that can generate tautologies of arbitrary forms.

6 Related work

Among decision-procedure based theorem provers, besides our work, Simplify [2], Verifun [4], and CVC Lite [0] all provide some degree of quantifier support.

Simplify's method of using triggering patterns to find instantiations [8, 2] has proved quite successful in practice. Once an instantiation is generated, it remains in the prover until the proof search backtracks from the quantifier atomic formula. We implemented a similar triggering algorithm and use a second SAT solver to reason about the instantiated formulas so that useful instantiations can be identified.

Our handling of quantifiers is based on Verifun's early work [5]. Some attempts have been made in Verifun to identify useful tautologies from instantiations of quantifiers. However, it seems that it is an optimization that works only when the instantiations alone can propositionally refute the current monome. In most scenarios, we believe, the quantifier module needs to cooperate with other theories to find out the instantiations that are useful to refute the monome.

In CVC Lite, each term is given a type and the formula is type checked. Types give hints about which terms can be used to instantiate a universal variable. However, instantiating a variable with every term whose type matches may be unrealistic for large problems.

7 Conclusion

This paper proposes a two-tier approach for handling quantifiers in a lazy-proof-explication theorem prover. The propositional reasoning of the original formula and that of the instantiated formulas are handled by two SAT solvers. The major purpose of this separation is to avoid unnecessary case splits caused by intertwining useless instantiations and the original formula. The $FindUnSATCore$ method can extract, from a set of tautologies generated during quantifier reasoning, a "good tautology" that is both relevant to the problem and sufficient to refute the given monome. We also use checkpointable theories to improve efficiency during the quantifier reasoning.

References

0. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, July 2004.
1. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, May 2002.
2. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
3. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
4. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *15th Computer-Aided Verification conference (CAV)*, July 2003.
5. Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Draft manuscript, May 2004.
6. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
7. Kenneth L. McMillan. Interpolation and SAT-based model checking. In *15th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2003.
8. Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also available as Xerox PARC technical report CSL-81-10, 1981.
9. Greg Nelson and Derek C. Oppen. Simplification by coorperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
10. Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, May 2003.