Fast Quantifier Reasoning With Lazy Proof Explication

Michał Moskal* Jakub Łopuszański*

May 23, 2006

Abstract

Lazy proof explication automated theorem provers do great reasoning about quantifier–free formulas, but when it comes to quantifiers, they struggle. However quantified formulas are of great importance for automated software verification. We present several novel techniques and heuristics used in the lazily proof–explicating Fx7[1] theorem prover to make it perform well (comparably to Simplify [2]) on quantified formulas occurring in the software verification scenarios. This opens a way for further improvements in the area.

1 Introduction

Ble ble ...

This paper is structured as follows. Sections 2 and 3 describe the general method we use in our theorem proving system. Next we describe heuristics we have developed, then discuss their performance impact and finally share some general comments about theorem prover construction.

2 Lazy prof explication

Lazy proof explication [4][1] is a technique of satisfiability checking[2], where a propositional SAT solver is used to drive exploration of the search space, and theory–specific decision procedures are used for discharging solutions, that are propositionally possible, but contradict the theories. In such a case the theory generates a tautology, that is communicated to the SAT solver as a *lemma*. So the proof is constructed lazily and costly theory reasoning is only called when strictly required.

This approach, combined with very fast SAT solvers, that are available today, leads to efficient satisfiability checkers.

The general algorithm used is depicted in Fig. 1. We first pass a formula we need to check to the boolean SAT solver, which should treat atoms in our formula as opaque boolean variables. It returns a *monome* – a conjunction of literals, that entail the boolean formula. Next we call the $CheckMonome$ function that should return a theory–tautology falsifying the monome or accept it.

*Institute of Computer Science, University of Wrocław

[1] http://nemerle.org/ malekith/smt/en.html

[2] We will view our ATP systems as satisfiability checkers instead of validity checkers, which is a perfectly dual problem.

```
CheckFormula(φ)
  loop
    m := BoolSat(φ)
    if m = UNSAT then
      return UNSAT
    else
      ψ := CheckMonome(m)
      if ψ = SAT then
        return SAT(m)
      else
        φ := φ ∧ ψ
```

Figure 1: Lazy proof explication for quantifier–free formulas.

An example (from [4]). Consider we need to prove the following formula:

$$[a = b] \wedge (\neg[f(a) = f(b)] \vee [b = c]) \wedge \neg[f(a) = f(c)]$$

where for clarity atoms are enclosed in square brackets. We throw it into the SAT solver, and it can answer:

$$[a = b] \wedge \neg[f(a) = f(b)] \wedge \neg[b = c] \wedge \neg[f(a) = f(c)]$$

So we throw it into our theory reasoning machinery, and it says that two of the passed literal conflict, and returns a tautology:

$$\neg[a = b] \vee [f(a) = f(b)]$$

We add the tautology to the SAT solver and ask it for another solution:

$$[a = b] \wedge [f(a) = f(b)] \wedge [b = c] \wedge \neg[f(a) = f(c)]$$

which contains another conflict, which is:

$$\neg[a = b] \vee \neg[b = c] \vee [f(a) = f(c)]$$

After adding this clause to the SAT solver, it says, there are no more solutions and thus the entire formula is unsatisfiable.

3 Reasoning with Quantifiers

Now, once we have a procedure for quantifier free formulas, we can proceed with quantified ones. The main expectation here is that the quantified formulas will be relatively small, sitting somewhere inside the main formula. Most often they come from axioms that describe the environment of the verified program.

First we note that we only need to consider the universal quantifier (the existential one is dual and can be encoded with negation). Next we treat any quantified formula as an opaque literal when passing it to the SAT solver in the main loop from Fig. 1, and make the theory reasoning procedures ignore the quantified literals. Now, if theories detect inconsistency we are

```
CheckMonome(n)
  φ := true
  S := ∅; I := ∅
  loop
    m := BoolSat(n ∧ φ)
    if m = UNSAT then return FindUnsatCore(n, φ)
    else
      ψ := CheckTheories(m)
      if ψ = SAT then
        ψ := true
        foreach ¬∀x̄.ξ(x̄) ∈ m and ¬∀x̄.ξ(x̄) ∉ S do
          ψ := ψ ∧ (¬∀x̄.ξ(x̄) ⇒ ¬ξ(c̄))
          S := S ∪ { ∀x̄.ξ(x̄) }
        if ψ = true then
          foreach ∀x̄.ξ(x̄) ∈ m and σ ∈ Matches(m, ∀x̄.ξ(x̄))
                and ⟨σ, ∀x̄.ξ(x̄)⟩ ∉ I do
            ψ := ψ ∧ (∀x̄.ξ(x̄) ⇒ σ(ξ))
            I := I ∪ { ⟨σ, ∀x̄.ξ(x̄)⟩ }
        if ψ = true then return SAT
      φ := φ ∧ ψ
```

Figure 2: Two–tier lazy proof explication for quantified formulas.

fine – axioms were not needed. Otherwise, we try to add some relevant instances of the axioms to the formula and see if we can prove inconsistency then. The algorithm is given in Fig. 2.

We place entire quantifier logic in the *CheckMonome* function from Fig. 1. It will invoke the *CheckTheories* function, which is, as previously *CheckMonome*, responsible for returning a tautology falsifying monome incompatible with the theories.

Because instantiations of quantified formulas can dynamically introduce boolean structure to the formula, we will use a second SAT solver to drive the boolean search. This is called a *two–tier approach* [5].

The formula ϕ will hold a conjunction of tautologies resulting from instantiations and theory reasoning. The S and I sets will hold a list of instantiations generated so far (so we don't repeat ourselves).

Now, when the current solution m of $n \wedge \phi$ is falsified by the theories, we just add the tautology to ϕ. Otherwise we check if there are negative quantified formulas that have not been instantiated yet. If so, we instantiate (skolemize) them all and add the generated tautologies to ϕ. Otherwise we look for all *possibly–relevant* instantiations of positive quantified formulas, and add them to ϕ. If there are no such instantiations or a constant bound[3] of the positive instantiation adding rounds in a single *CheckMonome* function is hit, we return that the monome is satisfiable.

When the SAT solver reports $n \wedge \phi$ to be unsatisfiable, we try to extract a small conflict from ϕ and return it to the main loop. This will be described below.

This procedure is incomplete because of the bound and because we can only guess the possibly–relevant instantiations. It always halts, though.

[3]Currently 5.

3.1 Example

An example, from [5]. Consider the formula:

$$[P] \equiv \forall x.\, [x < 10] \Rightarrow [R(f(x))] \qquad [Q] \equiv \forall y.\, [R(f(y))] \Rightarrow [S(g(y))]$$

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

When we ask the SAT solver in the main loop, we get:

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

This is then passed to $CheckMonome$ where, it is found be consistent with theories. It also doesn't contain any negated quantified formulas, therefore we look for possibly–relevant instantiations and find:

$$[P] \Rightarrow ([b < 10] \Rightarrow [R(f(b))]) \qquad [Q] \Rightarrow ([R(f(0))] \Rightarrow [S(g(0))])$$

We happily add this to the SAT solver and it says:

$$m' = m \wedge \neg[b < 10] \wedge \neg[R(f(0))]$$

which is inconsistent with the theories:

$$\neg[b = 1] \vee [b < 10]$$

now the SAT solver can say:

$$m' = m \wedge [R(f(b))] \wedge \neg[S(g(0))]$$

which is OK, with the theories, but the $[R(f(0))]$ introduced in the previous matching round, triggers a new possibly–relevant instantiation:

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

which forces the SAT solver to return:

$$m' = m \wedge [R(f(b))] \wedge \neg[R(f(0))] \wedge \neg[0 < 10]$$

which is of course inconsistent with the theories:

$$[0 < 10]$$

After adding the last clause, SAT solver says there are no more solutions. Now we need to find the specific conflict in the input literals.

3.2 UnSAT Core

The $FindUnsatCore$ function in the pseudo–code above is used to minimize the size of conflict–tautology returned to the main loop. In [5] it is defined like this:

If $BoolSat(n \wedge \phi) = UNSAT$ and $FindUnsatCore(n, \phi) = m$ then $m \subseteq n$ and $BoolSat(\bigwedge m \wedge \phi) = UNSAT$.

So it finds such a subset of n that it still conflicts with ϕ. We would like this subset to be minimal, but we do not require that. Then we can return $\neg \bigwedge m$ as a tautology to the main loop (this is because ϕ is a tautology).

This way we avoid adding any new literals to the main SAT solver preventing state explosion.

Back to our example, the generated conflict will be:

$$\neg[P] \vee \neg[Q] \vee [S(g(0))]$$

3.3 Matcher

In this section we will describe how to look for possibly–relevant instances of quantified formulas. First we select a set of triggers, which are some subterms of the formula in question (this is described in more detail in Sect. 3.4). Each trigger is a set of terms $\{t_1, \ldots, t_n\}$ and we are interested in such σ, that for all $i = 1 \ldots n$ a term congruent to $\sigma(t_i)$ can be found in the current monome. The congruence relation is induced by equations in the current monome. This can be formalized as:

$$\begin{array}{l} Matches(m, \forall \overline{x}.\xi(\overline{x})) \\ \quad T := Triggers(\forall \overline{x}.\xi(\overline{x})) \\ \quad \{\sigma \mid \{t_1, \ldots, t_n\} \in T,\ \exists s \in Subterms(m).\ \sigma(t_i) \cong s \text{ for } i = 1 \ldots n\} \end{array}$$

where $\cong$ means the congruence induced by m.

Checking if a trigger matches anything in m is NP–hard. However it turns out that most of the time of the matcher is eaten by the linear triggers (that is triggers, where each variable occurs only once). And for linear triggers there is a simple polynomial algorithm for matching. It is described in Sect. 4.3

3.4 Triggers

There are no new techniques involved in trigger selection than described in [2].

A *uni–trigger* of $\forall \overline{x}.\xi(\overline{x})$ is subterm t of one of the literals in $\forall \overline{x}.\xi(\overline{x})$ such that:

- t contains all $\overline{x}$
- t is not in the scope of a nested quantifier
- t passes the loop–test, that is $\forall \overline{x}.\xi(\overline{x})$ doesn't contain a larger instance of t, example:

$$\forall x.\, P(f(x), f(g(x)))$$

- t contains no proper subterm with the above properties

A *multi–trigger* is selected if no uni–trigger can be found. It is a set of subterms of $\forall \overline{x}.\xi(\overline{x})$ such that they collectively contain all the variables, and are not in the scope of a nested quantifier.

Fig. 3 presents a number of queries to the matcher about triggers with given structure. It's interesting to see that vast majority of queries are about very shallow terms.

4 Heuristics

In this section we describe various heuristics we have used to make the prover run fast. It is accompanied by the figures listing performance comparisons. We've run the tests on 1.8GHz Athlon 64 machine, running Linux with mono 1.1.14. The test cases all come from the ESC/Java2 suite – they are the condition generated for verification of the very ESC/Java2 source code.

We have split results in two chunks – the first (in Fig. 5) presents results on all formulas, while the second (in Fig. 6) only of valid formulas. While ESC/Java2 has many (about a half) failing pre/postconditions we feel that the passing ones are going to be far more important in a production environment.

The plots use logarithmic scale. The vertical axis lists results for the regular version and the horizontal one for a version with given optimization. The lines (beside the middle one) mean 2–, 4– and (possibly invisible in print) 8–times–slower/faster.

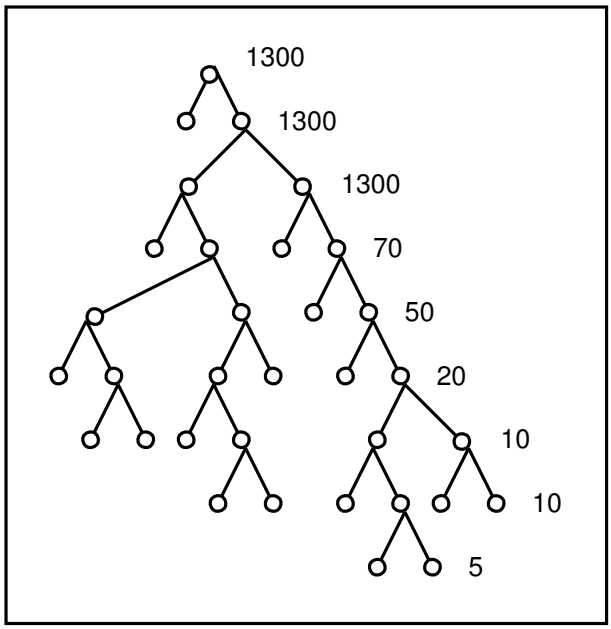

Figure 3: Structure of a typical trigger

4.1 Minimizing Boolean Model

Generally if we give a SAT solver a formula like: $p \wedge (q \vee r) \wedge z$ it will return to us something like $p \wedge q \wedge r \wedge z$ or $p \wedge q \wedge \neg r \wedge z$ even though valuating r is not needed. We would like to avoid such pointless valuations, to reduce work done by theory procedures.

Therefore we employ a simple minimization algorithm right after the SAT solver. It tries to remove variable valuation from model one by one and checks if the formula still evaluates to true.

4.2 Built–in transitive–closure theory

Examination of instance–count statistics lead us to believe that transitive closure axioms could lead to significant slowdowns. We therefore implemented a built–in theory for supporting relation symbols that are implicitly transitive and weakly antisymmetric.

4.3 Polynomial linear matching algorithm

The basic idea of the algorithm is depicted in Fig. 4. It works only for binary terms, but can be trivially extended for any signature.

Another important thing to note here, is the way we match multi–triggers (that is triggers with more than one term). The original Simplify [2] paper described a method where we first try to match the first term, and then for each match we try to extend the substitution with the matching of the second term and so on. Trying to implement it this way killed the performance – it was like 80% of time was eaten by matching multi–triggers. Instead we first match each term in a multi–trigger (hopefully with the polynomial algorithm if they are linear) and only later

```
Match(u)
  case u of
    f(t, s) ⇒
      T := Match(t)
      S := Match(s)
      R := ∅
      foreach f(t', s') ∈ Subterms(m) do
        if t' ∈_E T and s' ∈_E S then
          R := R ∪ {f(t', s')}
      return R
    c ⇒
      return {c}
    x ⇒
      return Subterms(m)
```

Figure 4: Polynomial algorithm for linear triggers.

check if they can be combined. Otherwise for 3–trigger we generally get $O(|m|^3)$ complexity and this is unacceptable.

4.4 Adding relevant instances to the main solver

We said before (in Sect. 3.2) that we add only the conflict clause back to the main SAT solver. This prevents state explosion, but on the other hand, when the *CheckMonome* function is invoked again, with slightly different n we need to start instance generation from the beginning.

It turns out that the typical conflict clause returned from *CheckMonome* contains just a few quantified literals, out of a hundred or so passed in. Therefore it seems reasonable to add the instantiations that led to the conflict to the main SAT solver, as they are not likely to cause state explosion, but are likely to be useful later.

Given a *FindUnsatCore* function we can easily compute set of such instantiations, but we need to look inside ϕ. Fig. 2 implies that it contains tautologies returned by the theory reasoning and implications of the form $\neg\forall\overline{x}.\xi(\overline{x}) \Rightarrow \neg\xi(\overline{c})$ and $\forall\overline{x}.\xi(\overline{x}) \Rightarrow \sigma(\xi)$. So we replace each such implication ψ_i in ϕ with $g_i \Rightarrow \psi_i$, where g_i is a fresh literal. Then we add all such g_i to n and run *FindUnsatCore*. We then extract the conflict to be returned, by just discarding all the guards, and use the guards to identify which instantiations to add to the main solver. This has a strong positive effect on performance.

Another optimization to use here, is based on the observation that the quantified formulas that were useful to prove the conflict in one run of the loop, are more likely to prove it in the subsequent run. Therefore we first try to instantiate only the quantified formulas marked as useful, and only if this fails, we try with all the other. This brings another few percents.

So instead of just the conflict, our version of *FindUnsatCore* returns:

$$(\neg[P] \vee \neg[Q] \vee [S(g(0))]) \ \wedge \ ([P] \Rightarrow ([0 < 10] \Rightarrow [R(f(0))])) \ \wedge \ ([Q] \Rightarrow ([R(f(0))] \Rightarrow [S(g(0))]))$$

4.5 Case–split accounting and explanations

A large part of the prover work was due to case splits generated by instances of axioms. If we have an axiom like $\forall x.\, \phi(x) \vee \psi(x)$ and instantiate it with $x := c$ then at some point we're likely

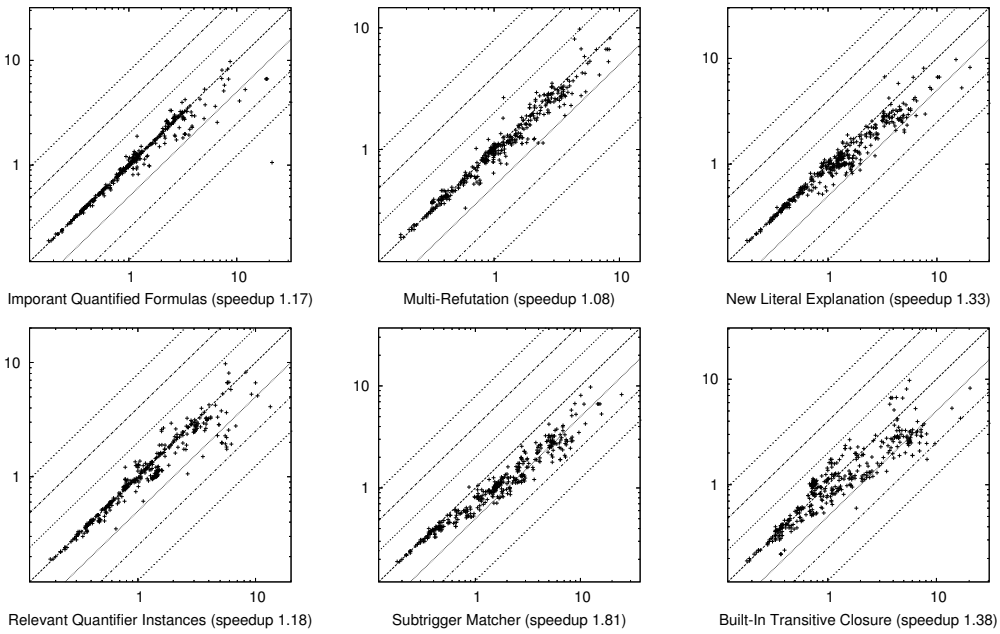

Figure 5: Effects of various optimizations (all formulas).

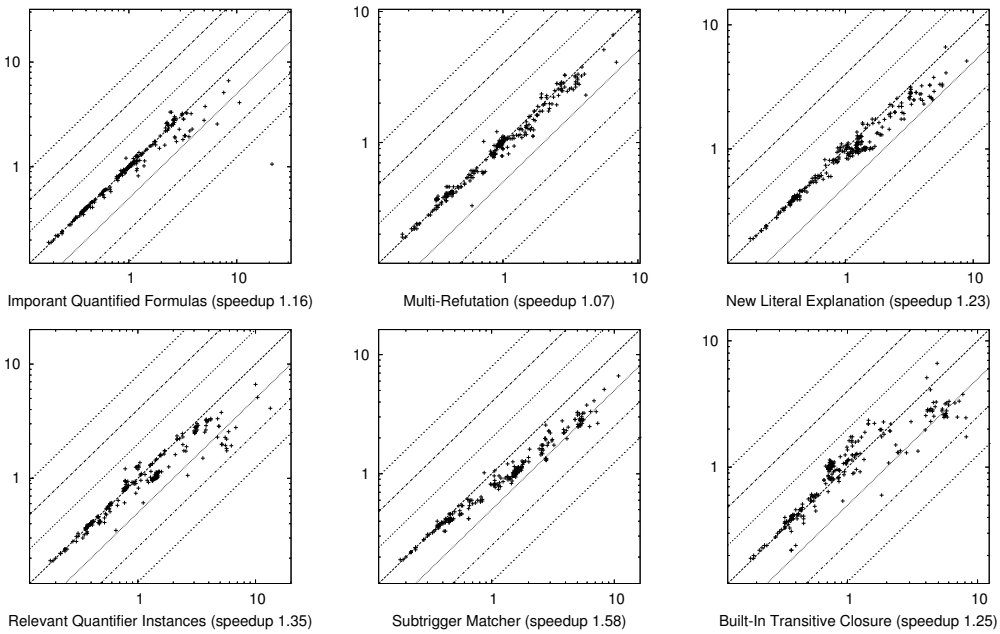

Figure 6: Effects of various optimizations (valid formulas).

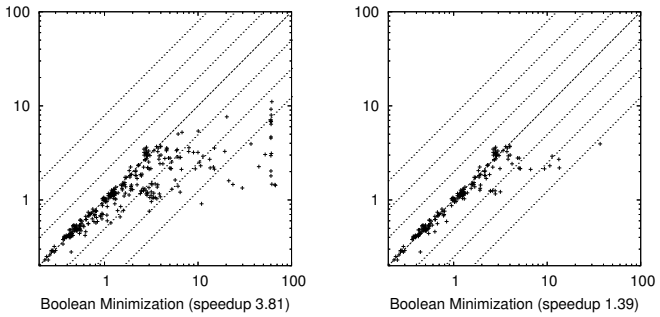

Figure 7: All and valid.

to have to choose if $\phi(c)$ is true, or maybe if $\psi(c)$ is true. However this is not always the case that axioms with alternative inside cause case–splits, because we can know from some other source, that always $\neg\psi(c)$. This is quite common, because often axioms have form of $\forall x.\, \phi(x) \Rightarrow \psi(x)$ and are only instantiated if $\phi(x)$ is found somewhere.

Therefore we tried to measure which axioms do cause case splits.

Let m_1 and m_2 be monomes returned from $BoolSat$ in the $CheckMonome$ function and $\sigma(\xi)$ is an instantiation of some axiom $\forall\overline{x}.\xi(\overline{x})$ and:

- $m_1 \vdash \sigma(\xi)$
- $m_2 \vdash \sigma(\xi)$
- $m_1 \cap Literals(\sigma(\xi)) \neq m_2 \cap Literals(\sigma(\xi))$

then $\sigma(\xi)$ is guilty of case–split.

So if $\sigma(\xi)$ is true in two different ways, we say it's guilty. This doesn't have to be true, but if some axiom is really guilty of case split, it will be marked as guilty.

If there were k theory reasoning invocations in the loop, all axioms have n instantiations guilty of case–split and our axiom $\forall\overline{x}.\xi(\overline{x})$ have m such instantiations than we account it $\frac{mk}{n}$.

It turned out that most case splits were quite stupid. Consider:

$$c = f(d) \wedge G(c) \wedge \forall x.\, G(f(x)) \Rightarrow H(x)$$

where we assume trigger $G(f(x))$. Now our matcher finds and adds the following tautology:

$$G(f(d)) \Rightarrow H(d)$$

and the SAT solver will try a monome where:

$$c = f(d) \wedge \neg G(f(d)) \wedge G(c)$$

which will lead to immediate conflict and second invocation of the SAT solver, where it returns:

$$c = f(d) \wedge G(f(d)) \wedge G(c) \wedge H(d)$$

We would like to avoid the first monome. Therefore for each new literal constructed by substitution generated from matcher, we add an explanation. In this case it would be:

$$(c = f(d) \wedge G(c) \Rightarrow G(f(d))) \ \wedge \ (c = f(d) \wedge G(f(d)) \Rightarrow G(c))$$

where in fact the first part is crucial and the second just helps a bit more. To construct an explanation we look for literal o that is equal to the new literal n and add a formula $proof\_of\_equality(o, n) \wedge o \Rightarrow n$ (and the other way around). The formula is a tautology so we're safe correctness–wise. When adding explanation we generally try to avoid o such that they are equal to n only because both are equated to the special $\$@true$ symbol.

5 What didn't work?

There were several heuristics that we tested, but they didn't work. We list the most interesting of them here.

5.1 Mod–time Optimization

The Simplify paper [2] describes two optimizations they used in the matcher. We tried to implement the mod–time optimization, which is used to prevent the matcher walking into parts of the term–database that hasn't been updated since the last matching.

It is not obvious how to implement this optimization in the lazy proof explication setting. The original implementation was based on the fact that the prover would backtrack upon conflict and undo only part of the changes made to the term database. On the other hand, after each conflict we invoke the SAT solver again and it can return totally different valuation. The SAT solver we use [3], removes all case–split decisions from its decision stack after restart, so it is very likely to return valuations that have little in common.

The implementation of this optimization was never the less possible, but it didn't seem to improve things. While it saved us about 50% of matching checks it didn't improve the running time.

The basic idea is:
After a matching round, store the monome $PREV$.
Before next matching round, with monome CUR:

1. assert $CUR \cap PREV$
2. enable change recording
3. push state
4. assert $PREV \setminus CUR$
5. pop state
6. assert $CUR \setminus PREV$
7. only look for matches in terms, that have some change recorded
8. save CUR as PREV

5.2 Using SAT Solver for Matching

It is possible to reduce the matching problem to SAT. The reduction we have come out with for trigger t and monome m uses $O(|t||m|)$ variables and a similar number of clauses. It is however tens times slower than the regular matching algorithm (which is not surprising given that most of the triggers are linear and hence solved in polynomial time).

6 Performance

Most of the time (about 83%) is spent in the quantifier reasoning module. The rest is 12% spent in theory reasoning not related to quantifiers and 5% in various parsing, CNF–conversion and similar.

Out of 83% spent for quantifier reasoning, about 17% is spent in theory reasoning, 28% in the matcher, 7% for applying substitutions found by matching to formulas, 12% for converting them to CNF, 8% for adding explanations and 3% for deducing conflicts to be returned to the main loop. The remaining 8% are wasted somewhere.

6.1 General implementation notes

If the $CheckMonome$ function doesn't try to minimize the conflict and just returns $\neg m$ as the conflict – the method fails to prove anything.

Trying to use the theories as black boxes (that is trying to deduce the minimal conflict from yes/no answers produced by the theories) causes very poor performance. The theories need to produce proofs, which can be used to extract conflicting literals.

TODO: (distinct ...) implementation.

The theories should be checkpointable – so we can assert the monome n in the $CheckMonome$ loop once, checkpoint it, add the literals resulting from instantiations (which are generally much smaller than n) and restore checkpoint in the loop.

References

[1] L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *SAT 2002*, 2002.

[2] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[3] Niklas En and Niklas Srensson. An extensible sat-solver. In *SAT 2003*, 2003.

[4] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *CAV*, pages 355–367, 2003.

[5] K. Rustan M. Leino, Madan Musuvathi, and Xinming Ou. A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In *TACAS*, pages 334–348, 2005.