A Saturation-Based Automated Theorem Prover for RISCAL

Viktoria Langenreither

18.11.2025

Goals of this Thesis

- extension of RISCTP/RISCAL by a saturation-based automated theorem prover for first-order logic with equality
- the theoretical basis for such a prover and the support for special theories (integer)
- implementation of the prover
- experiments and tests with the prover

Goals of this Presentation

- review of the design for our prover
- show the implementation of our prover
- short software demonstration

Strategy of our Prover

- variant of the superposition calculus (like the E Prover)
- given clause algorithm
  - proof state represented by sets of processed and unprocessed clauses
  - at each traversal of main loop, a *given clause* c gets picked
  - no unprocessed clauses left means the input set is satisfiable
  - if c is the empty clause, the unsatisfiability has been shown
  - all possible generating inferences between c and processed clauses get computed
- Discount loop
  - unprocessed clauses never participate in simplifications

Design of our Prover

```
U := {initial input clauses}; P := ∅
while U ≠ ∅ begin
    c := select_best(U)
    U := U \ {c}; simplify(c, P)
    if not redundant(c, P) then
        if c is the empty clause then
            success; clause set is unsatisfiable
        else T := ∅
        for each p ∈ P do
            simplify(p, (P \ {p}) ∪ {c})
         done
         T := generate(c, P)
         P := P ∪ {c}
         for each p ∈ T do
             p := cheap_simplify(p, P)
             if not trivial(p, P) then
                 U := U ∪ {p}
             fi
         done
         fi
    fi
end
Failure: Initial U is satisfiable, P describes model
```

select_best(U)

function select_best(U)
    $e :=\min_{>_E} \{\text{eval}(c) | c \in U\}$
    select c arbitrarily from $\{c \in U | eval(c) = e\}$
return c

Fig. 2. A simple *select_best()* function

select_best(U) — Clauseweight

- most common evaluation functions are based on *symbol counting*
- return number of function and variable symbols (possibly weighted in some way) of a clause
- preferring clauses with a small number of symbols

Why is this approach successful?

- small clauses are typically more general than larger clauses
- smaller clauses usually have fewer potential inference positions — processing smaller clauses is more efficient
- clauses with fewer literal are more likely to degenerate into the empty clause by appropriate simplifying inferences

select_best(U) — FIFOweight

- *first-in first-out* strategy
- new clauses are processed in the same order in which they are generated
- evaluation function simply returns the value of a counter that is incremented for each new clause
- pure FIFO performs very badly

Remark

If we ignore contraction rules, this heuristic will always find the shortest possible proofs (by inference depth), since it enumerates clauses in order of increasing depth.

simplify

clause-clause simplifying inference rules:

1. rewriting of positive literals

$$\frac{s = t \qquad u = v \vee R}{s = t,\ u[p \leftarrow \sigma(t)] = v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ for a substitution σ

2. rewriting of negative literals

$$\frac{s = t \qquad u \neq v \vee R}{s = t,\ u[p \leftarrow \sigma(t)] \neq v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ for a substitution σ

3. negative simplify-reflect

$$\frac{s \neq t \qquad \sigma(s = t) \vee R}{s \neq t,\ R}$$

for a substitution σ.

simplify & cheap_simplify

intra-clause simplifying inference rules:

1. deletion of duplicated literals

$$\frac{s = t \vee s = t \vee R}{s = t \vee R}$$

2. deletion of resolved literals

$$\frac{s \neq s \vee R}{R}$$

3. destructive equality resolution

$$\frac{x \neq s \vee R}{\sigma(R)}$$

if $x \in V$ and $\sigma = mgu(x, s)$.

redundant

1. clause subsumption

$$\frac{T \qquad R \vee S}{T}$$

if $\sigma(T) = S$ for a substitution σ.

generate

1. $$\frac{s \neq t \vee R}{\sigma(R)} \text{ (Equality resolution)}$$

where $\sigma = \text{mgu}(s, t)$ and $\sigma(s \neq t)$ is eligible for resolution.

2. $$\frac{s = t \vee S \qquad u \neq v \vee R}{\sigma(u[p \leftarrow t] \neq v \vee S \vee R)} \text{ (Superposition into negative literals)}$$

where $\sigma = \text{mgu}(u|_p, s), \sigma(s) \not\leq \sigma(t), \sigma(u) \not\leq \sigma(v), \sigma(s \neq t)$ is eligible for paramodulation, $\sigma(u \neq v)$ is eligible for resolution and $u|_p \notin V$.

3. $$\frac{s = t \vee S \qquad u = v \vee R}{\sigma(u[p \leftarrow t] = v \vee S \vee R)} \text{ (Superposition into positive literals)}$$

where $\sigma = \text{mgu}(u|_p, s), \sigma(s) \not\leq \sigma(t), \sigma(u) \not\leq \sigma(v), \sigma(s \neq t)$ is eligible for paramodulation, $\sigma(u \neq v)$ is eligible for resolution and $u|_p \notin V$.

4. $$\frac{s = t \vee u = v \vee R}{\sigma(t \neq v \vee u = v \vee R)} \text{ (Equality factoring)}$$

where $\sigma = \text{mgu}(s, u), \sigma(s) \not\leq \sigma(t)$ and $\sigma(s \neq t)$ is eligible for paramodulation.

trivial

1. syntactic tautology deletion 1

$$\frac{s = s \vee R}{}$$

2. syntactic tautology deletion 2

$$\frac{s = t \vee s \neq t \vee R}{}$$

Software-Demo

Main.java | Resolution.java | PureEquatio... | Generatingl... | Simplifyingl... | Pair.java | Main.java | Term.java

```java
/**************************************************************************
 * Solve a problem in clausal form.
 * @param problem the problem.
 * @return true if the solution succeeded.
 *************************************************************************/
public boolean solve(ClauseProblem problem)
{
  out.println("=== proof method 'res' not completely implemented yet");

  ProofProblem prob = problem.getProofProblem();

  //(in)equality symbol
  FunctionSymbol eq = prob.equalities.get(prob.boolSymbol);
  FunctionSymbol neq = prob.inequalities.get(prob.boolSymbol);

  //processed and unprocessed clauses as lists
  // the clauses of the problem
  List<Clause> unprocessed = problem.getClauses();
  List<PureEquation> u = new ArrayList<>();
  List<PureEquation> processed = new ArrayList<>();

  //the input problem in form of PureEquations
  for(Clause c : unprocessed) {
      u.add(new PureEquation(c, problem));
      out.println("transforming clauseproblem to pureEquation");
  }

  // for every clause an evaluation gets calculated and stored with the clause
  // unproc wird entsprechend der evaluation Function sortiert
  List<Pair> unproc = evaluationFunction(u); //this should be done after the first initial simplification
  sort(unproc, 0, unproc.size()-1);

  //Begin Hauptalgorithmus
  while (unproc.size() > 0) {
```

Possible Future Work

- Additional simplifying inference Rules
- Experimenting with the term ordering
- Integrating a literal selection function
- Improve Clause Selection

References

- Stefan Schulz. "E - a brainiac theorem prover". In: vol. 15. AI Communication, 2002, pp. 111–126. doi: 10.5555/1218615.1218621.
- Alexandre Riazanov and Andrei Voronkov. Limited resource strategyy in resolution theorem proving. Journal of Symbolic Computation. Oxford Road, Manchester M13 9PL, UK: Department of Computer Science, University of Manchester, 2003, pp. 101–115. doi: 10.1016/S0747-7171(03)00040-3.
- Stephan Schulz. "Learning Search Control Knowledge for Equational Theorem Proving". In: KI 2001: Advances in Artificial Intelligence. Ed. by Franz Baader, Gerhard Brewka, and Thomas Eiter. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 320–334. isbn: 978-3-540-45422-9. doi: 10.1007/3-540-45422-5_23.
- Laura Kovacs and Andrei Voronkov. "First-Order Theorem Proving and VAMPIRE". In: Computer Aided Verification. Springer, Berlin, Heidelberg, 2013, pp. 1–35. doi: 10.1007/ 978-3-642-39799-8_1