A Saturation-Based Automated Theorem Prover for RISCAL

Viktoria Langenreither

24.06.2025

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 and arrays)
- implementation of the prover
- experiments and tests with the prover

Goals of this Presentation

- review of the design for our prover
- show the implementation work done so far
- short software demonstration

Strategy of our Prover

- variant of the superposition calculus with literal selection (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
  - passive clauses never participate in simplifications

Design of our Prover

```
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 := 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 *symbole counting*
- return number of function symbols and variables (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 contracting 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

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. syntactic tautology deletion

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

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

simplify

1. semantic tautology deletion

$$\frac{s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \vee s = t \vee R}{}$$

if $\sigma(s_1 = t_1), \ldots, \sigma(s_n = t_n) \models \sigma(s = t)$, where the substitution σ maps all variables to distinct new constants.

2. destructive equality resolution

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

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

3. clause subsumption

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

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

redundant

1. clause subsumption
2. semantic tautology deletion

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 $\sigma = \text{mgu}(u|_p, s), \sigma(s) \geq \sigma(t), \sigma(u) \geq \sigma(v), \sigma(s \neq t)$ 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 $\sigma = \text{mgu}(u|_p, s), \sigma(s) \geq \sigma(t), \sigma(u) \geq \sigma(v), \sigma(s \neq t)$ 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 $\sigma = \text{mgu}(s, u), \sigma(s) \geq \sigma(t)$ and $\sigma(s \neq t)$ is eligible for paramodulation.

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) {
```

Further Work

What we have done so far:

- State of the art
- Throughout theoretical representation of the concepts needed for the prover
- Collecting strategies to make those concepts reasonably efficient
- Design of the prover

What we are doing now:

- Implementation of the prover
- Test the prover

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