A Saturation-Based Automated Theorem Prover for RISCAL

Viktoria Langenreither

20.06.2023

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

- explain strategies for resolution (saturation)
- presentation of the plans for our prover
- outlook on the integration of SMT solvers

Saturation Principle

- searching for a contradiction proceeds by saturating the given set of clauses
- can be made very efficient, if a powerful concept for redundancy elimination and good saturation algorithms are used
- at every step such an algorithm should select an inference, apply this inference to the set of clauses S, and add conclusions of the inferences to S
- a good strategy for inference selection is crucial for an efficient behaviour of a saturation algorithm

Given Clause Algorithms

- represents the proof state by a set P of processed clauses and a set U of unprocessed clauses
- initially, all clauses are in U and P is empty
- at each traversal of the main loop, the algorithm picks a clause c (the given clause) from U
- if U is empty, the original clause set is satisfiable (and the clauses in P describe a model)
- if c is the empty clause, the unsatisfiability of the original clause set U has been shown
- otherwise the algorithm performs all possible generating inferences between c and other arbitrary clauses from P
- the generated clauses are added to U and the process repeats
- different variants of the given-clause algorithm differ in their handling of contraction (simplification)

Otter Loop

- used by most provers
- popularized by the theorem prover Otter
- uses aktive and passive clauses for simplifications
- These algorithms have some interesting property: if the initial set of clauses is maximally simplified (with respect to itself), then the set of persistent clauses (all active and passive clauses) is also maximally simplified (with respect to itself) when starting a new iteration.

Remark

Given clause algorithms distinguish between kept clauses previously selected for inferences and those not previously selected. Only the former clauses participate in generating inferences. For this reason they are called active. The kept clauses still waiting to be selected are called passive.

Discount Loop

- modify the OTTER saturation algorithm
- inference pace may become dominated by simplification operations on passive clauses
- passive clauses never participate in simplifications

Some important features are:

- The new clauses are forward simplified by the active clauses only, the passive clauses do not take part in this.
- Neither active nor passive clauses are backward simplified by the retained new clauses.
- After selection of the current clause it is simplified again by the active clauses and then is itself used to simplify the active clauses. Again, the passive clauses are not affected.

Limited Resource Strategy

- variation of the Otter saturation algorithm
- a time limit is needed
- the main idea is trying to identify those passive and unprocessed clauses which have no chance to be selected (regarding the time limit) and discard them
- fundamental difference to redundancy
- measures the time spent to process a clause and occasionally estimates how the proof search pace develops towards the time limit

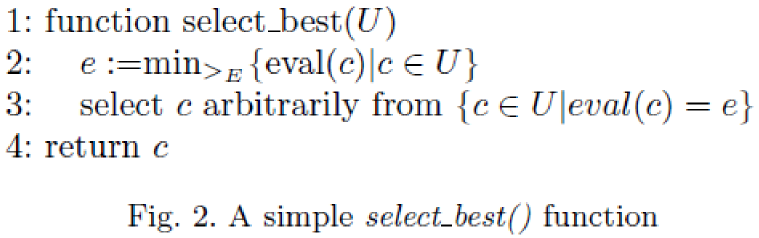
Our Prover

```
while $U \neq \emptyset$ begin
    $c :=$ select best($U$)
    $U := U \setminus \{c\}$; simplify($c, P$)
    if not redundant($c, P$) then
        if $c$ is the empty clause then
            success; clause set is unsatisfiable
        else $T = \emptyset$
        foreach $p \in P$ do
            if $c$ simplifies a maximal literal of $p$ such that the set of
             maximal terms, the set of maximal literals or the
             number of literals in $p$ potentially changes then
                $P := P \setminus \{p\}$; $T := T \cup \{p\}$
                $U := U \setminus \{d \mid d$ is direct descendant of $p\}$
             fi
             simplify($p, (P \setminus \{p\}) \cup \{c\}$)
         done
         $T := T \cup$ generate($c, P$)
         foreach $p \in T$ do
             $p :=$ cheap_simplify($p, P$)
             if not trivial($p, P$) then
                 $U := U \cup \{p\}$
             fi
         done
         fi
    fi
end
Failure: Initial $U$ is satisfiable, $P$ describes model
```

select_best(U)

- The only open choice point is the selection of the next clause.
- controlled by one or more heuristic evaluation functions
- A heuristic evaluation function usually maps a clause to a numerical evaluation.
- the set U is then organized as a priority queue ordered by the evaluations (new clauses are inserted at the proper position)

select_best(U)



```
1: function select_best($U$)
2:     $e$ :=$\min_{>_E}\{\text{eval}(c) | c \in U\}$
3:     select $c$ arbitrarily from $\{c \in U | eval(c) = e\}$
4: 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) — Refinedweight

- variety of modifications that can further improve this heuristic
- weight individual terms and literals in a clause in different ways
- assigning a higher weight to maximal terms and maximal (or selected) literals
- for unit clauses, this will prefer orientable clauses (rewrite rules) to unorientable ones — stronger rewrite relation earlier
- preferring clauses with few (and small) terms eligible for inferences, the explosion of the search space gets delayed

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.

select_best(U)

- one common variation of this general scheme is the introduction of a second priority queue, sorted by age (time at which the clause was generated)
- alternating selection of clauses from either queue with a fixed ratio (the *pick-given ratio*)
- older and lighter clauses are respectively prioritised

Remark

The theorem prover E found out that they get the best results with clause selection functions that combine three or four different priority queues. One of the best ways is to use two different instances of Refinedweight() with the remaining queues using a FIFO scheme.

How to Integrate SMT solvers in our Prover

1. only forward variable-free clauses to the SMT solver
2. forward clauses which contain only 1 literal, possibly with variables
3. the variables in clauses could get replaced by new constants and then the clauses can be forwarded
4. the whole set or a subset gets forwarded
5. break down clauses into variable-disjoint sub-clauses and pass on these sub-clauses

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

What we are doing now:

- Designing the prover (June)
- Implementation of the prover (June till September)

References

- 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
- Stefan Schulz. “E - a brainiac theorem prover”. In: vol. 15. AI Communication, 2002, pp. 111–126. doi: 10.5555/1218615.1218621.