Algorithms and Data Structures for First-Order Equational Deduction (extended abstract)

Stephan Schulz
Technische Universität München
schulz@eprover.org

1 Introduction

First-order logic with equality is one of the most widely used logics. While there is a large number of different approaches to theorem proving in this logic, the field has been dominated by saturation-based systems using some variant of the superposition calculus [BG90, BG94, BG98, NR01], i.e. systems that employ paramodulation, restricted by ordering constraints and possibly literal selection, as the main inference mechanism, and rewriting and subsumption as the main redundancy elimination techniques. Many systems complement equational reasoning with explicit resolution for non-equational literals. Examples of provers based on the combination of paramodulation, rewriting and (possibly) resolution include SPASS [WBH+02], Vampire [RV01], Otter [MW97], its successor Prover9, and E [Sch02, Sch04b].

The power of a saturating prover depends on four different, but interrelated aspects:

- The calculus (What inferences are necessary and possible?)
- The inference engine (How are they implemented?)
- The search organization (How is the the proof state organized and which invariants are maintained?)
- The heuristic control of the search (Which subset of inferences is performed and in what order?)

In this talk I will discuss the basic concepts of the *given clause* saturation algorithm and its implementation. In particular, I will describe the behaviour of the *DISCOUNT loop* version of this algorithm on practical examples, and discuss how this affected the choice of algorithms and data structures for E. I will try point out some low-hanging fruit, where a lot of performance can be gained for relatively modest investment in code complexity, as well as some more advanced techniques that have a significant pay-off.

2 Rewrite-Based Theorem Proving

Superposition is a refutational calculus. It attempts to make the potential unsatisfiability of a formula (consisting of axioms and negated conjecture) explicit using saturation. The search state is represented by a set of first-order *clauses* (disjunctions of *literals* over *terms*). The proof search employs two different mechanisms. First, new clauses are added to the proof state by *generating inference rules*, using existing clauses as premises. Secondly, *simplifying* or *contracting* inference rules are used to remove clauses or to replace them by simpler ones. The proof is successful if this process eventually produces the *empty clause*, i.e. an explicit contradiction.

While the generating inferences are crucial to establish the theoretical completeness of the calculus, extensive use of contracting inferences has turned out to be indispensable for actually finding proofs.

The practical difficulty of implementing a high-performance theorem prover results mostly from the fact that the proof state grows extremely fast with the depth of the proof search. A successful implementation has to be able to handle large data sets, efficiently find potential inference partners, and in particular, be able to quickly identify clauses that can be simplified or removed.

While the number of inference and simplification rules can be much larger, in nearly all cases only three rules are critical from a performance point of view:

- *Superposition* (including resolution as a special case) or a similar restricted form of paramodulation is by far the most prolific generating inference rule. Typically, between 95% and 99% of clauses in a non-trivial proof search are generated by a paramodulation inference. Paramodulation can essentially be described as a combination of instantiation (guided by unification) and lazy conditional rewriting. For superposition, this inference is further restricted by ordering constraints (a smaller term cannot be replaced by a larger term) and literal selection (only certain literals need to be considered as applicable or as targets of the application).

- Unconditional *rewriting* allows the simplification of a clause by replacing a term with an equivalent, but simpler term. In contrast to superposition, no instantiation of the rewritten clause takes place, and a term is always replaced by a smaller one. As a consequence, this is a *simplifying* inference, and the original of the rewritten clause can be discarded. Practical experience has shown that in most cases rewriting drastically improves the search behaviour of a prover.

- *Subsumption* allows the elimination of clauses if a more general clause already is known. As such, it helps to reduce the search space explosion typical for saturating provers. However, finding subsumption relations between clauses can be very expensive.

For completeness, it is necessary to eventually consider all combinations of non-redundant clauses as premises for generating inferences. To avoid the overhead of keeping track of each possible combination, the *given-clause* algorithms splits the proof state into two distinct subsets, the set P of *processed* (or *active)* clauses, and the set U of *unprocessed* (or *passive*) clauses, and maintains the invariant that all necessary

inferences between clauses in P have been performed. On the most abstract level, the algorithm picks a clause from U, performs all inferences with this clause and clauses from P (adding the resulting newly deduced clauses to U), and puts it into P. This process is repeated until either the empty clause is deduced, the set U runs empty (in which case there is not proof), or some time or resource limit has been reached (in which case the prover terminates without a useful result). The major heuristic decision in this algorithm is in which order clauses from U are picked for processing.

The two main variants of the given clause algorithm differ in how they integrate simplification into this basic loop. The *Otter loop* maintains the whole proof state in a fully simplified (or *interreduced)* state. It uses all unit equations for rewriting and all clauses for subsumption attempts. The *DISCOUNT loop*, on which E is built, only maintains this invariant for the set P of processed clauses. It simplifies newly generated clauses once (to weed out obviously redundant clauses and to aid heuristic evaluation, but it does not use them for rewriting or subsumption until they are actually selected for processing. It thus trades reduced simplification for a higher rate of iterations of the main loop. Opinions differ on which of the two designs is more efficient (see e.g. [RV03]).

3 Representing the Proof State

Analysis of the behavior of provers over a large set of examples has shown that the size of U typically grows about quadratically with the size of P. Therefore for non-trivial proof problems, the vast majority of clauses is in U, and unprocessed clauses are responsible for most of the resources used by the prover. The overall proof state can easily reach millions clauses, with a corresponding number of literals and terms as their constituents.

Surprisingly, with naive implementations much of the CPU time can be taken up with seemingly trivial operations. In the first version of DISCOUNT [ADF95, DKS97] we found to our surprise that assumedly complex operations like first-order unification were negligible, while linear time operations like the naive insertion of clauses into U (organized as a linear list sorted by heuristic evaluation) took up around half of the total search time.

The first and most basic data type for a first-order prover is the *term*. Simple, direct implementations realize terms as ordered trees, with each node labeled by a function or variable symbol, and with the subterms of a term as successor nodes in the tree. Alternatives to this basic design are either increasingly optimized for size, as flat-terms, string terms, and eventually implicit terms (reconstructed on demand) as in the Waldmeister prover [GHLS03], or they try to store more and more precomputed information at the term nodes to speed up repetitive operations. This is particularly successful if sets of terms are not represented as sets of trees, but as a shared directed acyclic graph, with repeated occurrences of any given term or subterm only represented once. This approach has been followed in E. We found sharing factors varying from 5-15 in the unit equational case, up to several 1000 in the general non-Horn case.

4 Rewriting

Shared terms do not only allow a reduction in memory consumption, they also allow the sharing of (some) term-related inferences. In particular, they allow the sharing of rewrite operations and, even more importantly, normal form information among terms. For any rewritten term, we can add a link pointing to the result of the rewrite operation. If we encounter the same term again in the future, we can just follow this link instead of performing a real search for matching and applicable rewrite rules. Less glorious, but not less effective, is the sharing of information about non-rewritability. In either version of the given clause algorithm, the set of potential rewrite rules changes over time, and the strength of the rewrite relation grows monotonically. If a term is in normal form with respect to all rules at a given time, no older rule has to be ever considered again for rewriting this term. This criterion can be integrated into indexing techniques to further speed up normal form computation.

5 Subsumption

A single rewriting step is usually cheap to perform. The high cost of rewriting comes from the large number of term positions and rules to consider. For subsumption, on the other hand, even a single clause-clause check can be very expensive, as the problem is known to be NP-complete [KN86]. Unless reasonable care is taken, the exponential worst case can indeed be encountered, and with clauses that are large enough that this hurts performance significantly[1].

To overcome this problem, a number of strategies can be implemented. First, presorting of literals with a suitable ordering stable under substitutions can greatly reduce the number of permutations that need to be considered. Secondly, a number of required conditions for subsumption can be tested rather cheaply. If any of these tests fail, the full subsumption test becomes superfluous.

Feature vector indexing [Sch04a] arranges several of these tests in a way that they can be performed not only for single clauses, but for sets of clauses at a time.

6 Conclusion

Engineering a modern first-order theorem prover is part science, part craft, and part art. Unfortunately, there is no single exposition of the necessary knowledge available at the moment - something that the community should aim to rectify.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In J. Hsiang, editor, *Proc. of the 6th RTA, Kaiserslautern*, volume 914 of *LNCS*, pages 397–402. Springer, 1995.

[1] This is an understated version of "The prover stops cold".

[BG90] L. Bachmair and H. Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, volume 449 of *LNAI*, pages 427—441. Springer, 1990.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[BG98] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 2(18):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[GHLS03] J.M. Gaillourdet, Th. Hillenbrand, B. Löchner, and H. Spies. The New Waldmeister Loop At Work. In F. Bader, editor, *Proc. of the 19th CADE, Miami*, volume 2741 of *LNAI*, pages 317–321. Springer, 2003.

[KN86] D. Kapur and P. Narendran. NP-Completeness of the Set Unification and Matching Problems. In J.H. Siekmann, editor, *Proc. of the 8th CADE, Oxford*, volume 230 of *LNCS*, pages 489–495. Springer, 1986.

[MW97] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science and MIT Press, 2001.

[RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (System Description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 376–380. Springer, 2001.

[RV03] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1–2):101–115, 2003.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[Sch04a] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland*, ENTCS. Elsevier Science, 2004. (to appear).

[Sch04b] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topić. SPASS version 2.0. In A. Voronkov, editor, *Proc. of the 18th CADE, Copenhagen*, volume 2392 of *LNAI*, pages 275–279. Springer, 2002.