System description: Vampire 1.0

Alexandre Riazanov and Andrei Voronkov

University of Manchester
{riazanov,voronkov}@cs.man.ac.uk

1 General description

Vampire is an automatic theorem prover for first-order classical logic. It implements the calculi of ordered binary resolution, hyperresolution and superposition for handling equality. The splitting rule is simulated by introducing new predicate symbols. The search process is based on a saturation algorithm similar to the one of Otter [4], enhanced with the Limited Resource Strategy briefly described in Section 3. For pruning the search space we exploit a number of standard redundancy criteria and simplification techniques: subsumption, tautology deletion, subsumption resolution and rewriting by ordered unit equalities. The only term ordering used in Vampire at the moment is a special version of the Knuth-Bendix ordering which allows efficient approximation algorithms for solving ordering constraints. One of the main goals of the project is creating an experimental environment for developing and evaluating efficient implementation techniques for first-order theorem proving.

2 Term indexing in Vampire

Even in the presence of good redundancy criteria, the main operations in a typical resolution-based theorem prover — unification, matching, subsumption — involve searching in huge sets of terms and clauses. The standard way of solving this problem is to maintain special datastructures for *term indexing* (see [2], [5]). In Vampire we use basically three kinds of indexes. A version of *discrimination trees* [3] is used for the unification indexes. Code trees [7] are used for forward subsumption and matching. A novel indexing technique based on path indexing [6] and database style joins is used for backward subsumption. Our main approach to designing efficient indexing schemes is based on the notion of *compilation*. To perform retrieval from an index efficiently we specialize the retrieval algorithm for every particular query clause or term in order to utilise specific properties of the query. For example, compilation for indexed unification makes use of information on occurences of variables and ground subterms in the query in order to identify redundant occurence checks. The specialized version is represented as a code for an abstract machine and then interpreted. Moreover, in our indexing for forward subsumption we also compile the indexed objects and combine their codes into a structure called a *code tree*.

3 Other features

In order to solve complex real-life problems in limited time, the completeness of search procedures used in provers is often compromised for the sake of efficiency.

The limited resource strategy implemented in our system is intended to do this in an intelligent way. Vampire tries to estimate what generated clauses can be processed by the given time limit. This estimation makes use of statistics on the speed of processing retained clauses of different complexity and establishes a limit on the complexity of retained clauses. Generated clauses whose complexity exceeds this limit are discarded.

Another distinguishing feature of Vampire is an efficient and flexible implementation of splitting (see for example [1]). When a clause can be split into the components $C_1, \ldots, C_n$ with disjoint sets of variables, we introduce n new propositional variables $p_1, \ldots, p_n$, that can be regarded as names for the components, and replace the clause by $n+1$ new clauses $C_1 \vee p_1 \quad , \ldots , \quad C_n \vee p_n$, $\neg p_1 \vee \ldots \vee \neg p_n$. The literals p_i in the new clauses C_i, p_i are made minimal in order to ensure that they are not selected before all the other literals are cut by resolution. A special index is maintained to avoid giving different names to components that are variants of each other.

4 Work in progress

At the moment we are interested in finding optimisations for our term indexing techniques which could allow us to treat efficiently symbols with special properties such as commutativity of functions and symmetry of predicates. Also, some attempts are being made to integrate checking of ordering constraints into the indexing structures.

5 Implementation and availability

The system is implemented in C++ and can be compiled by 2.91 or newer versions of gcc. Currently it runs on Solaris, Linux and Free BSD. It is available from the authors free of charge, for details check
`http://www.cs.man.ac.uk/~riazanov/Vampire/`.

References

1. Weidenbach C. Spass: combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science and MIT Press, 2000. To appear.
2. P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.
3. William W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
4. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.
5. I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science and MIT Press, 2000. To appear.
6. M. Stickel. The path indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.
7. A. Voronkov. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.