System Description: KRHyper

Christoph Wernhard

Institut für Informatik, Universität Koblenz-Landau, D-56070 Koblenz, Germany,
wernhard@uni-koblenz.de

Abstract. KRHyper is a first order logic theorem proving and model generation system based on the hyper tableau calculus. It is targeted for use as an embedded system within knowledge based applications. In contrast to most first order theorem provers, it supports features important for those applications, for example queries with predicate extensions as answers, handling of large sets of uniformly structured input facts, arithmetic evaluation and stratified negation as failure.

1 Introduction

KRHyper is a first order logic theorem proving and model generation system based on the hyper tableau calculus [2]. It is targeted for use as an embedded system within knowledge based applications such as those described in [1, 3].

While KRHyper is based on techniques used in first order theorem provers — the hyper tableau calculus for first order logic proving, unification, term indexing and fair search control by iterative deepening — it supports a number of features important for knowledge based applications that are not usually found in first order provers. These features are discussed in the following section.

Compared to state-of-the-art first order provers, as represented by the participants of the annual *CADE ATP System Competition*,[1] the performance of KRHyper is in the mid range for unsatisfiable Horn problems without equality and for satisfiable problems with finite Herbrand universe. It is at the low edge for unsatisfiable non-Horn problems without equality and Horn problems with equality. For other problem classes it is poor. KRHyper is not equipped with special equality handling. Also, since the behavior as competitive universal first order prover was not our goal, there are still possible optimizations to explore.

Our system shall be a basis for future experiments with new techniques in real-world application contexts. So the implementation has to be both modular and accessible in the depth. After working with prototype systems written in Prolog, where one gets operations such as unification, indexing and search control for free — but only in the limited sense supported by Prolog — we choose the functional-imperative compiler language *OCaml*[2] as implementation language.

[1] See `http://www.cs.miami.edu/~tptp/CASC/`.
[2] See `http://www.ocaml.org/`.

2 Features

With KRHyper application tasks are specified using first order logic — plus possibly non-monotonic constructs — in clausal form. While the system can be used straightforwardly to prove theorems, it also allows the following features which go beyond the common capabilities of first order theorem provers:

1. Queries which have the listing of predicate extensions as answer are supported.
2. Queries may also have the different extensions of predicates in alternative models as answer.
3. Large sets of uniformly structured input facts are handled efficiently.
4. Arithmetic evaluation is supported.
5. Non-monotonic negation is supported.
6. The reasoning system can output proofs of derived facts.
7. The system can be used as reasoner for a description logic, enhanced with rules and ABox reasoning capability.

Hyper tableau is a "bottom-up" method, which means that it generates instances of rule[3] heads from facts that have been input or previously derived. Derived facts are stored as lemmas. This has the heuristic effect of avoiding redundant re-computations and supports the use of the calculus for model generation, since it makes it possible to detect when a fixed point of rule application is reached.

If a hyper tableau derivation terminates without having found a proof, the derived facts form a representation of a model of the input clauses.[4]

A rule head may be a disjunction. In hyper tableau, disjunctions are handled by exploring the alternative branches in a systematic way. KRHyper uses backtracking to generate one model after the other.

Of the features listed above, items 1 and and 2, the generation of answer sets, are made possible through model generation.

Large sets of uniformly structured input facts play a role comparable to base relations of databases in conventional applications, however nested and incomplete data structures can be represented by terms. So item 3 benefits from implementation techniques used in database systems, which can be smoothly integrated with the hyper tableau method.

The handling of special language constructs, items 4 and 5, is facilitated by two aspects of the controlled way in which the hyper tableau calculus builds up data structures: First, it does not generate new clauses with negative literals,

[3] A clause is viewed as rule "*Head* `:-` *Body*.", where *Head* consists of its positive literals, combined by "`;`" (*or*), and *Body* consists of its negative literals, combined by "," (*and*). If a clause contains only positive literals, i.e. is a fact or a disjunction, it is notated as "*Head*.", if it contains only negative literals, as "`false :-` *Body*.".

[4] The Herbrand model output consists of all ground instances of the derived facts. Since the derived facts must not necessarily be ground, in some cases they can characterize an infinite model.

which means that only input clauses have negative literals, and so information about the context in which special predicates will be evaluated is statically available at preprocessing. Second, the implementation of nonmonotonic operations such as negation as failure is facilitated by the possibility to detect when a fixed point of inferencing is reached. Intuitively speaking, we then know that it is not possible to infer certain information, and can use this knowledge positively.

Regarding item 6, many first order theorem provers can output the proofs of refutations, albeit often in an idiosyncratic syntax that makes it difficult to process them further. For model generation, it is additionally desirable that derivations of the facts belonging to a model are available.

Item 7, the practical suitability as a processor for description logic extended by rules and ABox reasoning, is a consequence of the other features. KRHyper has been used as target platform for a higher level knowledge representation language based on description logic, which is outlined in [1].

3 Model Generation and Theorem Proving Method

KRHyper implements the hyper tableau calculus by a combination of semi-naive rule evaluation [15], a technique from the field of deductive databases, with backtracking over alternative disjuncts and iterative deepening over a term weight bound.

Iterative deepening over the term weight bound ensures refutational completeness also for problems which are non-Horn. Furthermore it provides a heuristics that restricts rule evaluation for refutation tasks which involve the generation of terms of ever increasing size.

Hyper Tableau. In its basic variant, the hyper tableau calculus has a single inference rule, *hyper tableau extension*, which is similar to positive hyperresolution, except that non-Horn clauses are processed differently: a derived disjunction leads to a case-split. The disjuncts may not share variables, which means that, as in the γ rule of tableau, some arbitrary instance of the disjunction is inferred.

Processing in the hyper tableau framework is characterized by maintenance of three clause sets:

1. *Facts* — positive unit clauses, which are either directly from the input or have been derived. For non-Horn problems, they are arranged in a tree that branches at case-splits, the *hyper tableau.* With a depth first control regime, at any point of time the calculus works on a single such branch. So it suffices to maintain a single *facts* set. If no proof can be found, *facts* provides a straightforward representation of a model: the Herbrand model consisting of all ground instances (with respect to the input signature) of the elements of *facts.*
2. *Disjunctions* — positive non-unit clauses. With the strategy of our implementation this set is used to retain disjunctions that have been input or derived. They are waiting to be moved to the hyper tableau as case-split nodes.

3. *Rules* — clauses which contain negative literals. They come directly from the input and remain fixed during the reasoning process. This suggests the application of preprocessing techniques to them.

Outline of the Operation of KRHyper. *Facts* and *disjunctions* are initialized by the positive input clauses. The term weight bound is initialized to some positive integer value.

KRHyper proceeds by extending *facts* by the fixed point of the application of *rules* to it. Derived disjunctions are not used within the fixed point computation, but retained in *disjunctions*. Derived facts and disjunctions that exceed the term weight bound are thrown away.

After the fixed point of rule application has been reached, a disjunction is picked from *disjunctions* and used as branching node for backtracking over the alternatives. *Facts* is extended by the respective disjunct and the algorithm proceeds recursively by generating the fixed point again.

If, after a fixed point is reached, *disjunctions* is empty, then, if the term weight bound had not been exceeded in the branch, a model is found. Computation halts with *facts* as a representation of the model and a continuation object to compute further models as output. If the term weight had been exceeded, computation is restarted with an increased weight bound at the topmost ancestor node where it had been exceeded.

If during computation of the fixed point a solution for a rule with `false` head is obtained, that branch is abandoned immediately, since it is *closed*, and the computation backtracks to the next branch. If there is no next branch, computation halts with the result that there is no [more] model.

4 Computation of the Fixed Points of Rule Application

Evaluation of a Single Rule. From the hyper tableau perspective, the *evaluation* [15] of a rule with respect to a set of facts is the computation of all possible hyper tableau extension steps for the given rule and selected branch. Rule evaluation can be viewed as a relational join over the rule body, followed by a projection on the variables in the rule head.

We presently have a single implementation of rule evaluation, which is a form of index nested loop join, that proceeds like the enumeration of solution substitutions for a Prolog body. In contrast to Prolog, however, solutions for a literal are only obtained from the fact base and not by entering further rules.

Solutions for literals should be obtained as far as possible through index accesses which have as keys either non-variable terms already present in the rule or variables that are bound by solutions from preceeding literals. So the "join order", the ordering of body literals, is important for performance.

Our current implementation determines this ordering heuristically when rules are preprocessed by a greedy algorithm. The most significant property used is the applicability of indexes, which is determined by the portion of indexed arguments that get non-variable arguments. Since discrimination tree (see e.g.

[10]) index lookup is faster if the first arguments are not variables (at least in our implementation), also the positions of the non-variable arguments are considered.

Semi-Naive Evaluation of a Rule Set. The core algorithm for evaluating rule sets is an adaption of *semi-naive evaluation* [15], a standard technique used in deductive database systems to compute the extensions of recursively defined relations. After an initial step in which each of the rules is evaluated straightforwardly, the algorithm proceeds in rounds, where information which facts had been newly derived in the previous round is available. One of these fact must then participate in each rule evaluation.

With respect to Horn clauses, semi-naive evaluation may be regarded as a modification of the *Otter* inference loop [11]: all clauses in the *sos* are transferred to *usable* at the end of a round. The whole set of clauses derived in the previous round plays the role of *given clause*.

5 Refinements and Implementation Details

User Configurable Indexing per Predicate. A separate fact store for each predicate is maintained. For each predicate, indexing methods and argument positions to be used as keys can be declared by the user. Several different indexes can be declared for a single predicate.

Currently, methods that index on nested terms with variables and use discrimination trees are implemented. For the symbol dispatch within the discrimination trees association lists or hash tables can be used. The latter allow to use the general term indexing mechanism also for fact bases that are more like conventional databases, with large signatures and terms that are not deeply nested.

Backtrackable Index Implementation. Backtracking over alternative disjuncts requires to modify and restore the state of certain data structures, including the *facts* set, which is represented by term indexes. A backtrackable term index is implemented simply as a list of ordinary term indexes: New entries are put into the head element, lookups are performed on all elements. On backtracking, the head element is popped from the list.

Level Cut and Complement Splitting. KRHyper optionally employs two refinements of model generation theorem proving calculi: *complement splitting* (*folding down*) [9, 8] and *level cut* [2].

Maintenance of Negative Unit Clauses. Negative unit clauses are often used as input to express the goal of a theorem proving task, and they can be derived when complement splitting is used. KRHyper maintains a special store for negative literals. If a positive literal that unifies with a negative on the same branch is derived, the branch is abandoned immediately.

Rule Ordering. Rules with heads containing fewer literals evaluated first. So if a rule with an empty (i.e. `false`) head has a solution, the branch is abandoned immediately before any rule with a nonempty head is evaluated. After a disjunction has been derived, a subsumption check against unit literals in the branch is performed.[5] By the ordering of rules, unit literals that have been newly derived in the same round are also considered in this subsumption checks.

Handling of Term Weight Bounds. As noted above, if a fixed point of rule application is reached, but the term weight had been exceeded, computation is restarted with an increased weight bound at the topmost ancestor node where it had been exceeded. Facts already inferred at this node are retained, and semi-naive evaluation starts with a non-incremental "initial step" at this node.

The number of symbols in the atom is used as weight of a literal. The weight of a clause is that of its heaviest literal. Parameters used for iterative deepening, such as the initial bound, can be configured by the user. For most problems of the *TPTP* library [14] a small start value is good.

6 Special Logic Operators and Builtins

At the preprocessing of rules, the system ensures that variables in input arguments of builtins are instantiated when they are evaluated, under the assumption of range restriction.[6] Specific predicate argument positions can be declared as unrestricted by the user.

Stratified Negation as Failure. The system supports stratified negation as failure [15]. Rules are grouped according to their strata. If a model of some stratum has been computed, the next higher rule set is evaluated with the model as a basis. Backtracking then can effect that further models of the lower stratum are found. Computation of the higher strata is started anew on each of these models.

Stratified Set Abstraction. A stratified set abstraction construct and a group of builtins for operations on sets (of ground terms) represented as ordered lists, similar to the Prolog `ordset` library, is included.

Arithmetic Evaluation. The arithmetic constant types, evaluable arithmetic functors and arithmetic builtins specified for ISO standard Prolog [4] are provided.

[5] This check has heuristic purposes only, to let the list of retained disjunctions not grow too large. It does not in general make a subsumption check immediately before a disjunction is going to be attached to the hyper tableau redundant.

[6] A clauses is range restricted, if each of its variables occurs in a body literal that is "ordinary", i.e. non built-in and not using a special logic operator such as stratified negation as failure. A set of clauses is range restricted if each of its clauses is so.

7 Further Features of the System

Proofs. Proofs of refutations and derivations of facts in models can be output as terms. They can be visualized as graphs with the *Graphviz*[7] tool.

Concrete Syntax. The input syntax of KRHyper is the *Protein* format which is supported by the TPTP tools. Any outputs which are intended for machine processing are Prolog readable. The syntax of in- and outputs complies with the ISO standard for Prolog [4].

Aside from clauses, input statements can be commands, for example to set configuration values, declare indexes, start the model generation process, continue to generate the next model or output the extension of a predicate in the last generated model.

Input statements can be contained in files or read from the standard input. KRHyper accepts arbitrarily many input files. In this way clause sets can be modularized and separated from configuration and control information.

Interaction and Caching. KRHyper can not only be run on a single proving or model generation task, but also with its in- and output connected to a terminal or a client program in an interactive session.

It maintains a set of "current facts": Before model generation is started, this set consists of the input facts. When a model has been generated, "current facts" represents that model. When no more model can be generated (i.e. after the last model had been generated or when a refutation is found), "current facts" again has the input facts as value.

Optionally the system behavior can be modified, such that facts derived at the root of the hyper tableau tree (i.e. before any disjunction is attached, which implies that they hold in any model) are merged into the input fact base. This feature can be dynamically switched on and off, thus allowing to cache the extensions of derived predicates, which are subsequently used in answering queries.

Implementation Language. We choose *OCaml* as implementation language for KRHyper. It accomplished what we expected: It allows a functional-imperative programming style similar to Lisp. A powerful module system provides coarse structuring of a growing system. Also a documentation generator, similar to the well known *Javadoc*, helps to keep programs accessible. Static type checking supports program modifications with widespread consequences. The OCaml compiler produces standalone executables that do not seem to consume time or space where not necessary. We did not make use of OCaml's object oriented extension, since it seems not fully matured and difficult to hide at the module interface level. We did not really miss it: in most cases code sharing and code dispatch by other means, such as parameterized modules and pattern matching on type constructors, was satisfactory.

[7] See `http://www.research.att.com/sw/tools/graphviz/`.

8 Performance Evaluation

The profile of the input data from applications for which KRHyper is intended, for example the document management application described in [3], is different from the majority of problems in the TPTP: We have large sets of facts with few variables, sometimes with large signatures. The focus is on model generation instead of theorem proving. Derived terms often do not exceed the size of the terms in their parent facts.

An Excerpt from an Application. An extract from the aforementioned document management application includes a fact base of 16470 ground facts with the `references/2` predicate, representing cross references between document fragments. The fragments themselves are denoted by nested terms. An example fact is:

```
references(analysis/0/1/1/11/7, analysis/0/0/4/17).
```
[8]

The transitive closure of `references` has 217439 facts. With the current version of KRHyper, its computation needs 17 seconds.[9] After loading the input facts, 4.5 megabytes of virtual memory are consumed, after computing the transitive closure 63 megabytes. Two other numbers show that our efforts are indeed necessary for such tasks: A seemingly similar system, *Satchmo Compiler*[10] [13], with default settings under SWI-Prolog needs 2:14h and 271 megabytes. *Otter 3.2*, with positive hyperresolution as inference rule and the fact base as *sos* needs 37 minutes and 124 megabytes.[11]

Evaluation on TPTP Problems. We tested KRHyper on three subsets of the TPTP, including the problems of the most recent leading systems competition and all problems without equality.[12]

Problems of the CADE-18 ATP System Competition. This is the problem set of the *CADE-18 ATP System Competition* (*CASC-18*) which was held in 2002.[13]

All experiments with KRHyper were performed with a single refutation complete configuration that starts with a low initial term weight bound. Problems which are non-Horn and not range restricted are transformed by a preprocessor:

[8] The binary functor `'/'` is written here as operator that associates to the right.

[9] All experiments with KRHyper reported here were performed on a 2.4GHz Pentium 4 Linux PC.

[10] `http://www.pms.informatik.uni-muenchen.de/software/satchmo-compiler.tgz`

[11] The example is available for download at `http://www.uni-koblenz.de/~wernhard/provers`.

[12] Detailed tables are available from `http://www.uni-koblenz.de/~wernhard/provers`.

[13] See `http://www.cs.miami.edu/~tptp/CASC/18/`.

The definition of an auxiliary predicate, that explicitely has all ground terms over the signature as extension is added, and for each variable in a disjunctive rule head, a literal that requires this variable to satisfy the auxiliary predicate is put into the rule body. This auxiliary predicate is not handled specially by the prover.

Since KRHyper by itself currently provides no means for goal sensitivity, we also experimented with the non-Horn magic sets transformation by Mark Stickel which is provided with the TPTP.

Table 1 compares for each competition division the number of problems that could be solved by KRHyper with that of the competition systems. For each division there were 35 problems to solve, with exception of `UEQ`, which contained 70 problems. Not all competition systems took part in all divisions. KRHyper provides no special means to handle equality.

Table 1. Results on Problems of the CADE-18 ATP System Competition

Division	*Eql*	*Horn*	*Sat*	*Finite*	*KRH*	*KRH-Mag*	*KRH-All*	*Worst*	*Best*
HNE		•			19	16	24	12	34
NNE					11	8	11	10	33
EPT				•	5	4	5	7	34
SNE			•		5	3	5	10	28
EPS			•	•	15	15	15	0	27
HEQ	•	•			16	16	16	18	33
NEQ	•				2	2	2	6	32
UEQ	•				4	4	4	11	70
PEQ	•				0	0	0	2	29
SEQ	•		•		0	0	0	14	33

The meaning of the table columns is the following: *Division* is the name of the division; *Eql*, *Horn*, *Sat* and *Finite* indicate that the problems in the division include equality, are Horn, are satisfiable, and have a finite Herbrand universe respectively; *KRH*, *KRH-Mag*, *KRH-All*, *Worst* and *Best* are the numbers of problems that could be solved by KRHyper directly, by KRHyper with the non-Horn magic sets transformation, by KRHyper in at least one of these two variants and by the worst and the best competition system for the respective division.[14]

Horn Problems without Equality. KRHyper was tested on the 749 problems of the TPTP 2.5.0 which are Horn and without equality. The configuration used is refutation complete.

The results are summarized in table 2. The solved problems with the highest TPTP ratings are `LCL099-1` and `LCL125-1` with a rating of 0.86. We played with different initial term weight bounds: column *KRH-Std* shows the number

[14] The TPTP problem files for all tests with KRHyper reported here were from version 2.5.0 of the TPTP. The timeout in this test was 600s and the memory limit 500MB.

of solved problems for a value of 3, *KRH-20* for a value of 20 and *KRH-Std/20* for both variants together.[15]

As in the CASC-18 experiments, we also tried the non-Horn magic sets transformation included with the TPTP. Especially in the `PLA` domain, the magic sets transformation contributes solutions.

To check satisfiability, no extra methods, such as propositional abstraction, have been used: for each satisfiable problem that has been solved, a model has been computed. With the magic sets transformation, the models obtained are obviously models of the transformed clauses and not the original ones.

Table 2. Results on Horn Problems without Equality from the TPTP

Status	*KRH-Std*	*KRH-20*	*KRH-Std/20*	*KRH-Mag*	*KRH-All*	*No. of Problems*
satisfiable	22	22	22	31	32	50
unsatisfiable	572	550	582	529	603	676
unknown/open	0	0	0	0	0	23

Of the 594 solutions obtained by *KRH-Std*, 71% took less than 0.1s, 8.4% 0.1-1s, 7.6% 1-10s, 8.6% 10-100s and 4.4% more than 100s. All of the 22 solutions for satisfiable problems were obtained in less than 0.2s.

Among the 676 problems with status unsatisfiable three could not been solved since the memory limit was reached. For another three problems, clauses with more than 64 variables, a limit fixed when compiling the KRHyper executable, were generated. For the remaining 98 unsolved problems, the time limit was reached.

Non-Horn Problems without Equality. We also tried the 1172 problems of the TPTP 2.5.0 which are non-Horn and without equality. The configuration used is refutation complete and has a low initial term weight bound. The results are summarized in table 3.[16]

Table 3. Results on Non-Horn Problems without Equality from the TPTP

Status	*Range Restricted*	*KRH*	*No. of Problems*
satisfiable	yes	52	63
satisfiable	no	67	331
unsatisfiable	yes	164	235
unsatisfiable	no	137	399
unknown	yes	0	137
unknown	no	0	7

[15] The timeout was 650s and the memory limit 500MB.
[16] The timeout was 300s and the memory limit 500MB.

9 Related Systems

There are some systems, dating from the late eighties up to now, which are related to KRHyper, since they are based on the same background methods, implementation techniques, also generate models or address similar applications. A profound comparison seems not easy, since there is no widespread set of benchmark problems for model generation systems.

Various techniques for efficiently implementing *Satchmo* [9], one of the ancestors of hyper tableau, are discussed in [13]. One of the techniques is *incremental evaluation*, which seems similar to semi-naive evaluation. Two of these advanced Satchmo systems participated in the *Mixed Problems* division of CASC-13, 1996, where they could solve 6 of the 50 problems. With a timeout of 60s KRHyper can solve 15 of them. A Satchmo variant also participated in the *SAT* division of CASC-15, 1998, where it solved 11 of the 30 problems. KRHyper, with a timeout of 20s, solved 12.[17]

MGTP [5] is another family of systems descending from Satchmo with advanced implementations. In [6] a variant of MGTP is described, that, like ours, adapts techniques like semi-naive evaluation to model generation theorem proving.

SModels [12] is a model generating system, which computes stable models with the help of a preprocessing step that generates ground instances of the input program. Thus it works essentially on propositional instead of first order logic: terms with function symbols are either evaluated in the manner of built-ins during preprocessing or used as syntactic sugar for constants.

DLV [7] is a system for disjunctive datalog with constraints including nonmonotonic negation with several front ends for special reasoning modes. It addresses applications in the field of knowledge representation, but its does not support full first order logic.

Compared to a deductive database system such as *Aditi* [16], our system lacks database system features: it is not disk based, it does not support concurrent access with transactions and does not include a programming language binding. Also it currently does not include any means for goal sensitivity, such as the magic sets transformation.

References

1. P. Baumgartner, U. Furbach, M. Gross-Hardt, T. Kleemann, and C. Wernhard. KRHyper Inside — model based deduction in applications. In *Proc. of the CADE-19 Workshop on Challenges and Novel Applications for Automated Reasoning*, Forthcoming, 2003.
2. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In J. J. Alferes, L. M. Pereira, and E. Orlowska, editors, *Proc. of the European Workshop on Logics in Artificial Intelligence (JELIA'96)*, LNAI, pages 1–17. Springer, 1996.

[17] KRHyper was run on a 2.4GHz Pentium 4 Linux PC. Problems files were from TPTP 2.5.0, where one of the CASC13 problems of had been removed, and one of CASC-15 SAT problems had been bug-fixed and is now unsatisfiable.

3. P. Baumgartner, M. Gross-Hardt, and A. Sinner. Living books in education. *Submitted*, 2003.
4. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The standard: reference manual.* Springer, Berlin, 1996.
5. H. Fujita and R. Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In K. Furukawa, editor, *Proc. 8th Int. Conf. on Logic Programming*, pages 535–548. MIT Press, 1991.
6. K. Furukawa, T. Ozaki, T. Murakami, K. Ueno, and K. Shimazu. Query evaluation of deductive database by MGTP and its application to data mining. In U. Geske, C. Ruiz, and D. Seipel, editors, *Proc. of the 5th Int. Workshop on Deductive Databases and Logic Progamming*, volume 317 of *GMD-Studien*, 1997.
7. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, C. Koch, C. Mateis, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. Technical Report INFSYS RR-1843-02-14, Technische Universität Wien, 2002.
8. R. Letz, K. Mayr, and C. Goller. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13(3):297–337, Dec. 1994.
9. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E. L. Lusk and R. A. Overbeek, editors, *Proc. of the 9th Conf. on Automated Deduction*, LNCS, pages 415–434. Springer, 1988.
10. W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
11. W. W. McCune. *OTTER 3.0 Reference Manual and Guide.* Argonne National Laboratory/IL, USA, 1994.
12. I. Niemelä and P. Simons. Smodels — an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proc. of the 4th Int. Conf. on Logic Programming and Non-Monotonic Reasoning*, pages 420–429, 1997.
13. H. Schütz and T. Geisler. Efficient model generation through compilation. In M. A. McRobbie and J. K. Slaney, editors, *Proc. of the 13th Int. Conf. on Automated Deduction*, LNCS, pages 433–447. Springer, 1996.
14. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In A. Bundy, editor, *Proc. of the 12th Int. Conf. on Automated Deduction*, LNAI, pages 252–266, Berlin, 1994. Springer.
15. J. D. Ullman. *Principles of Database and Knowledge-Base Bystems.* Computer Science Press, Rockville, Maryland, 1989.
16. J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, P. J. Stuckey, T. S. Leask, and J. Harland. The Aditi deductive database system. *VLDB Journal: Very Large Data Bases*, 3(2):245–288, 1994.