INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig-Maximilians-Universität München

An Abstract Machine for Model Generation with PUHR Tableaux (Extended Abstract)

Alexander von Drach, Tim Geisler, Sven Panne, David Sacher

In: Proc 13th Workshop Logische Programmierung (WLP '98), TU Wien, Oct. 1998
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1998-13, September 1998

An Abstract Machine for Model Generation with PUHR Tableaux (Extended Abstract)

Alexander von Drach, Tim Geisler, Sven Panne, and David Sacher

Institut für Informatik, Universität München, Oettingenstr. 67, D-80538 München
{vondrach|geisler|panne|sacher}@informatik.uni-muenchen.de

1 Introduction

The model-generation paradigm for automated deduction [MB88] is to demonstrate the *satisfiability* of first-order formulas by constructing satisfying Herbrand interpretations (i. e., models) instead of demonstrating their *unsatisfiability* by constructing a refutation (e. g., a resolution proof). Model generation was originally motivated by database applications (a detailled description of such an application appears in [BEST98]). Other successful applications of model generation include model-based diagnosis [BFFN97], planning [EG98], and even the solution of previously open problems in finite algebra [FSB93].

Among the various approaches for building models of first-order formulas (e. g., [CZ91,FH91,Sla92,ZZ95,FL96,BFN96,Pel98]), we concentrate in this paper on the *positive unit hyper-resolution (PUHR) tableau method* [BY96]. It combines positive hyper-resolution with unit clauses and a beta or splitting tableau rule.

The PUHR tableau method is actually a formalisation of the theorem prover Satchmo [MB88], an early attempt to integrate resolution into the tableau framework for automated deduction in order to yield efficient tableau methods. The Prolog programs given in [MB88] are remarkably short and simple, yet efficient, because they re-use rather than re-implement Prolog's features like backtracking and matching.

Two techniques enhancing the efficiency of PUHR tableaux were introduced with *Compiling Satchmo* [SG96]: *incremental evaluation*, a technique known from deductive databases as *Δ-iteration*, and *compilation*. Compiling Satchmo compiles a set of first-order clauses into a Prolog program, which computes its minimal models more efficiently.

A problem with the approach of using Prolog as a target language for compilation is that the language is still too high-level: Compiling the Prolog rules to native code by a Prolog compiler takes prohibitively long when the initial clause set is large, and using a Prolog system based on an abstract machine (e. g., the WAM [War83,Aït91]) adds another interpretation layer. We therefore developed a target language and corresponding machine at a level which lies more in the "middle" between clause sets and native code. This *abstract machine* directly supports incremental evaluation, backtracking, matching, and an appropriate efficient data structure for the representation of the models under construction.

This extended abstract is based on a longer report [vDGPS98], which describes the systematic developement of the abstract machine, its concrete realization, and the compilation schemes in detail and gives detailed benchmark results.

In the following, we first review the PUHR tableau method, incremental evaluation, and the compilation done by Compiling Satchmo. We then describe the systematic development of a prototype of an abstract machine in Prolog, following the approach developed by Kursawe [Kur86] and Nilsson [Nil92,Nil93] to derive an abstract machine for Prolog. We further present the necessary modifications to the result of this development process in order to obtain an abstract machine that can be implemented in a suitable low-level imperative language.

2 The PUHR Tableau Method

The PUHR tableau method [BY96] is a calculus for range-restricted clauses. We call the set of negative literals of a clause its *body*, and the set of positive literals its *head*. A clause is range-restricted if each of its variables occurs in the body. Many practical problem domains can be naturally formalized with range-restricted clauses [EG98]. If not, clauses can always be transformed into a range-restricted form [MB88,BY96].

PUHR tableaux are trees in which every node is labeled with a set $\mathcal{U}$ of ground atoms (**u**nits) and a (multi-)set $\mathcal{P}$ of **p**ositive ground clauses. PUHR tableaux for a given clause set are built starting from the tree consisting of a node labeled with the empty set of units and the set of positive ground clauses from the input clauses, by repeatedly applying the following two rules to a branch with leaf node $(\mathcal{U}, \mathcal{P})$.

The *PUHR rule* resolves the body literals of an input clause simultaneously with units from $\mathcal{U}$ and extends the branch with a new leaf $(\mathcal{U}, \mathcal{P}')$, where $\mathcal{P}'$ is the result from adding the (hyper-)resolvent to $\mathcal{P}$.

The *splitting rule* extends the branch either with n new leaf nodes $(\mathcal{U} \cup \{p_i\}, \mathcal{P} \backslash \{p_1 \vee \ldots \vee p_n\})$, if none of the p_i is in $\mathcal{U}$, or otherwise with one new leaf node $(\mathcal{U}, \mathcal{P} \backslash \{p_1 \vee \ldots \vee p_n\})$. As a borderline case, if the empty clause is selected from $\mathcal{P}$ (i. e., $n = 0$), the branch is marked as *closed*.

If none of these rules is applicable to the leaf node of a branch, then its set $\mathcal{U}$ represents a Herbrand model of the input clause set. If all branches are closed, then the input clause set is unsatisfiable.

The PUHR tableau method has properties such as refutation completeness and minimal-model completeness which are desirable for model generation [BY96], provided the two rules are applied in a fair way. Note that this *fair control* can be realized by computing all possible hyperresolvents in one step and by implementing $\mathcal{P}$ as a queue.

Incremental evaluation can be used to obtain significantly more efficient implementations of the PUHR tableau method [SG96]. To avoid repeated application of the PUHR rule with the same clauses and to accelerate its applicability test we apply this rule *incrementally*: If a unit is added to $\mathcal{U}$ in a splitting step, then exactly those PUHR steps that use this unit have to be applied as the next steps.

Most implementations of the PUHR tableau method *interpret* a set of clauses. *Compilation techniques* can be used to avoid this level of interpretation, because the PUHR tableau method with incremental evaluation is well suited for compilation [SG97].

The PUHR rule and the splitting rule can be mostly specialized for each input clause. The set of input clauses is fixed. Furthermore, a hyperresolvent generated by applications of the PUHR rule to an input clause is always a (ground) instance of this clause's head. As a consequence, it even suffices to store head identifiers and substitutions in $\mathcal{P}$.

In order to speed up the discovery of potentially resolvable clauses, the incremental application of the PUHR rule can be specialized for each body literal of a clause and for each predicate symbol.

This compilation approach was applied to Prolog implementations of the PUHR tableau method, resulting in Compiling Satchmo. It compiles a set of clauses into a Prolog program, which efficiently computes the set of minimal models of the given clauses [SG96,GPS97].

3 Systematic Development of an Abstract Machine

The purpose of an abstract machine is to bridge the conceptual gap between language and hardware. The design of an abstract machine is a difficult process, which requires a lot of ingenuity. From the existing work towards a methodology for the design of abstract machines, we followed the approach of Kursawe [Kur86] and Nilsson [Nil92,Nil93], who systematically reconstructed from the instruction set of the WAM those parts that deal with unification and control, respectively.

Their approach starts from interpreters for logic languages implemented in Prolog with stepwise enhanced expressivity. Simultaneously to these extensions to the implemented logic language, Prolog prototypes are developed by partial deduction and other program transformation techniques. Each prototype implements the abstract machine instructions and hand-compiled abstract machine code, represented as Prolog rules.

In order to count as a *machine*, the abstract machine instructions have to be implemented in Prolog in such a way that at the machine level they can be interpreted deterministically and neither recursion (apart from tail-recursion, i. e., looping) nor backtracking is used. Unification has to be reduced to simple parameter passing [Nil92].

We follow this methodology, with the extension that global data structures in the interpreter (e. g., the two sets $\mathcal{U}$ and $\mathcal{P}$ in our case) remain global in the abstract machine. Those data structures give rise to abstract machine operations to manipulate them.

We develop the abstract machine for the PUHR tableau method in several steps. In each step, we enhance the expressivity of the logic source language, and simultaneously extend the design of the abstract machine, i. e., introducing or modifying machine instructions and global data structures. During this process, we maintain two invariants: The machine should provide incremental evaluation and a fair control.

3.1 Definite Propositional Clauses

As a starting point, we choose to develop a machine for definite propositional clauses, i. e., propositional clauses with exactly one head atom. This language already requires the PUHR rule with incremental evaluation, but requires neither matching, the application of substitutions, branching in the tableau, nor backtracking.

A definite propositional clause of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \to \beta$ with a non-empty body has $n \geq 1$ *entry points* (also denoted α_k) for incremental evaluation: each of the atoms α_k could have been added to $\mathcal{U}$ by the preceding application of the splitting rule. The following operations are performed by the *body code sequence* for entry point α_k: Lookup all α_i (with $i \neq k$) in $\mathcal{U}$; if all lookups are successful, add to $\mathcal{P}$ the address of the *head code sequence* corresponding to β (PUHR rule) and return to the caller; if any lookup fails, return immediately to the caller.

Head code sequences test if their corresponding β is already in $\mathcal{U}$; if not, β is inserted into $\mathcal{U}$ and all body code sequences with entry point β are called as a subroutine. After this, the next address is fetched from $\mathcal{P}$ and jumped to.

The initial code sequence of the machine adds the addresses of all head code sequences for an entry point β to $\mathcal{P}$, where β is a clause with an empty body, i. e., a fact.

The machine handles two global data structures: the set of atoms $\mathcal{U}$, which are just symbols in the propositional case, and a queue of head code addresses—the *continuation queue* implementing the set of atoms $\mathcal{P}$. The queue ensures fairness.

There are three instructions dealing with the set of atoms $\mathcal{U}$. One instruction inserts an atom into $\mathcal{U}$. The lookup instruction for a body atom returns to the caller of the body code sequence, if the atom is not in $\mathcal{U}$. Otherwise it does nothing. In constrast, the lookup instruction for a head atom does nothing if the atom is not in $\mathcal{U}$. Otherwise it fetches the next address from the continuation queue and jumps to it.

There is an instruction to enqueue a code address and an instruction to proceed with the next element of the queue: if the queue is not empty, it dequeues a code address and jumps to it; otherwise the machine terminates and a model is found.

Two further instructions are required for calling and returning from body code sequences. A single register suffices to store the return address, because there are no subroutine calls in body code sequences, i. e., subroutine calls are not nested.

3.2 Propositional Clauses

In the previous step every clause had exactly one head atom. We now extend the logic language to full propositional clauses, i. e., the head of a clause can now be a (possibly empty) disjunction. With this extension, the splitting rule introduces branching into the tableaux, with the possibility of closed branches. This can be implemented using backtracking. We postpone the introduction of variables—the alternative choice for the current development step—because backtracking is easier to manage without variables.

In the abstract machine backtracking gives rise to a new global data structure—the *splitting choice point stack*. A new instruction is introduced which pushes a new choice point onto the stack, saving the states of $\mathcal{U}$ and the continuation queue and an address for an alternative.

The head code sequences have to be adapted to non-definite clauses. First, for each head atom a lookup instruction is performed. Then, a choice point is generated for all but the rightmost head atom. For each head atom β_i, subroutine calls to all body code sequences with entry point β_i have to be performed.

Empty clause heads are compiled into a single instruction. If possible, this instruction pops a choice point, resets $\mathcal{U}$ and the continuation queue and jumps to the saved alternative address. Otherwise the machine terminates with the result that no model is found.

After a model is found and presented to the user, backtracking can be initiated in order to compute further models.

3.3 Ground Clauses

We now introduce compound ground terms into the logic language. Note that this logic language has the same expressivity as propositional clauses. This step prepares the introduction of variables.

Composite ground terms have to be linearized because the parameters of a machine instruction should have a fixed size. Therefore, every machine instruction having a (possibly compound) atom as argument is replaced by a sequence of instructions with arguments of a fixed size.

3.4 Clauses with Non-Ground Bodies and Ground Heads

We now extend the logic language by introducing variables in clause bodies. We postpone the introduction of variables into the clause heads, because the specialization of the incremental application of the PUHR rule is more complicated with non-ground terms than with ground terms in the clause heads.

We introduce a new data structure—an *environment*—which represents the current bindings of variables to terms. The environment is presently needed only for the body code sequences. If a body code sequence with non-ground entry point α_i is called, the variables in α_i have to be bound to terms by the caller. This parameter passing is implemented by letting the caller allocate the environment and subsequently bind the appropriate variables.

The incremental evaluation technique requires to perform all PUHR steps that use the unit added to $\mathcal{U}$ most recently. Therefore, another backtracking mechanism is needed to find a compatible substitution in the hyper-resolution step. We introduce a second stack—the *hyper-resolution stack*. An entry of this stack consists of a code address to proceed if another variable binding has to be tried, a set of substitutions and an index denoting the substitution to be used in the next backtracking step.

Every code sequence for each body atom containing unbound variables is preceded by an instruction for creating a hyper-resolution-stack entry. Note that due to the range restriction of the input clauses, it can be determined at compilation time for each variable if it is already bound or not. Additional matching instructions are needed in body code sequences, which match and bind unbound variables or match an already bound variable, respectively. If a matching instruction fails, an alternative binding for a variable has to be tried. To accomodate this, the top entry of the hyper-resolution stack is considered: If there is another substitution represented in this entry, the bindings of the environment are updated to that substitution and the control flow continues at the proceed address of this stack entry; otherwise, the entry is popped and the previous stack entry is considered; if there is none, control returns to the caller.

3.5 Non-Ground, Range-restricted Clauses

We finally extend the logic language by introducing variables into the clause heads. This results in the logic language the PUHR tableau method can handle: range-restricted first-order clauses. Note that general first-order clauses can always be transformed into a range-restricted form [MB88,BY96].

The PUHR rule ground instantiates non-ground clause heads. In order to represent an instance of a clause head, the continuation queue entries additionally carry an environment, binding the variables of the corresponding head atom.

The body code sequences are extended, such that in case of a successful match for all body atoms an environment binding the head variables is constructed and enqueued together with the address of the head code sequence. Hyper-resolution backtracking is now enforced in order to find all possible bindings for the head variables.

A head code sequence is now preceded by instructions which allocate an environment for head variables and fetch its bindings from the continuation queue. In the previous development step it was known at compile time whether a body literal of a clause is resolvable with another head atom. Therefore, it was possible to perform the subroutine calls unconditionally. Now, subroutine calls to an entry point which is non-linear, e. g., $p(X, X)$, or a call with non-variable arguments may have to be preceded by a code sequence which tests the bindings of the head variables for compatibility. Some new machine instructions and an additional register were neccessary for this purpose.

4 Concrete Realiziation of the Abstract Machine

Designing an abstract machine is only half of the business: In order to compare its performance with existing implementations of the PUHR calculus and to validate our design, we developed a bytecode [Kin97] compiler/interpreter system. The compiler was implemented in Haskell [PH97], re-using an existing front-end from Functional Satchmo [GPS97], which can transform full first-order formulas into range-restricted clauses. The output of this compilation phase is a compact bytecode, which is then executed by an interpreter written in C++. Although this part could have been done in Haskell, too, we decided not to do so because of the danger of leaving too many things implicit (e. g., garbage collection).

4.1 Extensions due to the Implementation in C++

Going from the Prolog prototype to a realization in an imperative[1] language requires that further parts of the machine are made explicit.

Data Structures. The machine data structures that were developed during the stepwise design were directly translated into the C++ implementation, with two exceptions:

The splitting choice point stack and the hyper-resolution stack can be merged into a single stack, because a hyper-resolution stack is always completely emptied before a previous splitting choice point is accessed.

In the Prolog prototype, the runtime system simply uses a list of ground terms to represent the current interpretation. This list is replaced by a *discrimination tree* [McC88], resulting in two enhancements. Memory usage is reduced by sharing some common prefixes of the terms, but more importantly, matching is more efficient: The machine instructions representing a linearized term implement a depth-first traversal of the tree, aborting an unsuccessful match as soon as possible.

Memory Management. The life time of most dynamically allocated machine data structures is statically known, an exception being those concerned with backtracking, i. e., the continuation queue and the terms referenced by the environment. Backtracking makes it neccessary to return to older versions of the queue. This kind of "persistency" is implemented by saving all versions of the queue, while sharing its common

[1] We mainly use the imperative parts of C++.

parts. To handle deallocation, *reference counting* is used. This simple form of garbage collection suffices, because no cyclic data structures occur.

4.2 Visualization of the Abstract Machine

For demonstration purposes we connected the implementation of the abstract machine with the graph visualization system *daVinci* [FW94]. This GUI to the abstract machine enables the user to single-step through the abstract machine code and to view all data structures of the machine in a graph representation. Additionally, it was extremely useful for debugging the implementation of the machine. Furthermore, we expect to recognize possible inefficiencies and design flaws of the machine and machine code.

5 Preliminary Benchmark Results

We performed some preliminary benchmark experiments with the concrete realization of the compiler and the abstract machine. The details on these and some other experiments are given in [vDGPS98].

5.1 Bytecode Size

Specialization and compilation approaches can suffer from an explosion of the size of the specialized/compiled code. In particular, the specialization of the incremental application of the PUHR rule might lead to a nonlinear increase in code size.

We compared the size of the generated bytecode with the size of the input problem for all problems in clause normal form of the TPTP library [SSY94]. This problem library contains theorem proving problems with a large variety of features like clause size or term nesting depth.

We observed (again, the details are given in [vDGPS98]) that the bytecode size grows non-linearily, but sub-quadratically with the problem size. We further observed that the size of the bytecode grows faster than the size of the compiled code generated by Compiling Satchmo and subsequently compiled to WAM code with a Prolog compiler. We suppose that this is due to the larger amount of compilation and specialization done by our approach.

Note that the compilation times depend on the size of the resulting code as well as on the size and some syntactic features of the input clauses. We do not report the compilation times here because the implementation of our compiler is not optimized for efficiency yet.

5.2 Model-Search Time

The model-search times for implementations of PUHR tableaux depend heavily in the choice of the tableau rule to apply next, because this influences the structure of the search space. Preliminary experiments using some artificial examples showing a unique search space with our system and with Compiling Satchmo show a speedup of about an order of magnitude for our system compared with Compiling Satchmo.

6 Related Work

We are not aware of any abstract machine directly intended and used for model generation. We consider Prolog and KL1, which are used by Compiling Satchmo [SG96] and MGTP [Ins95], respectively, as too high-level for being counted as "abstract machines". Abstract machines are successfully used to implement efficient refutational theorem provers. Most prominently, SETHEO's implementation is based on an abstract machine [LSBB92]. The most important differences between our abstract machine and the WAM [War83,Aït91] are on one hand the (complex) global data structures for the set of positive unit clauses and the set of disjunctions to be split, and on the other hand the absence of full unification. The SLG-WAM is another abstract machine for a Prolog-like language which uses a discrimination tree data structure [RRS$^+$95].

7 Conclusions and Ongoing Work

We designed an abstract machine for model generation with the PUHR tableau method, which ensures fairness and incrementality. During the design process, the expressivity of the logic language was enhanced stepwise and Prolog protypes of the machine were developed in parallel. Finally, we implemented a bytecode compiler/interpreter system in order to validate the design and to be able to compare the performance of the system with other implementations of the PUHR tableau method.

During the whole process, we used the methodology proposed by Kursawe and Nilsson as a guideline. The Prolog prototypes were extremely useful for the design of the instruction set of the abstract machine and for the developement of the compilation schemes. Nevertheless, we did not strictly follow the proposed methodology. Sometimes the creative steps in the design of the abstract machine are very difficult. Our past experiences from implementations of the PUHR tableau method suggested creative steps that are difficult to justify by just following the proposed methodology.

In the future, we plan to verify the design of the abstract machine by further experimentally comparing it with other implementations of the PUHR tableau method and by using statistics gained from execution traces, which will probably have influence on the concrete instruction set. We further plan to extend the abstract machine in order to integrate some variants and refinements of PUHR tableaux. From the implementation point of view, we are planning to avoid the interpretation level completely by directly compiling to a low-level language like C-- [PNO97].

Acknowledgments

We thank François Bry, Norbert Eisinger, and Heribert Schütz for helpful comments on drafts of this paper.

References

[Aït91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[BEST98] François Bry, Norbert Eisinger, Heribert Schütz, and Sunna Torge. SIC: Satisfiability checking for integrity constraints. In Piero Fraternali, Ulrich Geske, Carolina Ruiz, and Dietmar Seipel, editors, *Proc. 6th International Workshop on Deductive Databases and Logic Programming, DDLP '98*, GMD Report 22, Manchester, UK, June 1998.

[BFFN97] Peter Baumgartner, Peter Fröhlich, Ulrich Furbach, and Wolfgang Nejdl. Tableaux for diagnosis applications. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 97*, number 1071 in Springer LNCS, pages 76–90, Pont-a-Mousson, France, May 1997. Springer-Verlag.

[BFK+98] Peter Baumgartner, Ulrich Furbach, Michael Kohlhase, William McCune, Wolfgang Reif, Mark Stickel, and Tomás Uribe, editors. *Problem-solving Methodologies with Automated Deduction, Workshop at CADE-15*, Lindau, Germany, July 1998.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In *Logics in Artificial Intelligence: European Workshop, JELIA '96*, number 1126 in Springer LNAI, 1996.

[BY96] François Bry and Adnan Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proc. TABLEAUX '96*, number 1071 in Springer LNCS, pages 143–159, Terrasini, Palermo, Italy, May 1996. Springer-Verlag.

[CZ91] Ricardo Caferra and Nicolas Zabel. Extending resolution for model construction. In *Logics in AI: European Workshop JELIA '90*, number 478 in Springer LNAI, pages 153–169, 1991.

[EG98] Norbert Eisinger and Tim Geisler. Problem solving with model-generation approaches based on PUHR tableaux. Technical Report PMS-FB-1998-8, Institut für Informatik, LMU München, 1998. In [BFK+98].

[FH91] Hiroshi Fujita and Ryuzo Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In *Logic Programming, Proc. of the 8th Int. Conf.*, pages 535–548, 1991.

[FL96] Christian Fermüller and Alexander Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2):173–203, 1996.

[FSB93] Masayuki Fujita, John Slaney, and Frank Bennett. Automatic generation of some results in finite algebra. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence, IJCAI 93*, pages 52–57. Morgan Kaufmann, 1993.

[FW94] Michael Fröhlich and Mattias Werner. The graph visualization system daVinci – A user interface for applications. Technical Report 5/94, Institut für Informatik, Universiät Bremen, September 1994.

[GPS97] Tim Geisler, Sven Panne, and Heribert Schütz. Satchmo: The compiling and functional variants. *Journal of Automated Reasoning*, 18(2):227–236, 1997.

[Ins95] Institute for New Generation Computer Technology. *Model Generation Theorem Prover: MGTP*, 1995. http://www.icot.or.jp/ICOT/IFS/IFS-abst/082.html.

[Kin97] Andreas Kind. Bytecode-Interpretierung. *Informatik-Spektrum*, 20(2), April 1997. In German.

[Kur86] Peter Kursawe. How to invent a Prolog machine. In Ehud Y. Shapiro, editor, *Third International Conference on Logic Programming*, number 225 in Springer LNCS. Springer-Verlag, July 1986.

[LSBB92] Reinhold Letz, Johann Schumann, S. Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In E. L. Lusk and R. A. Overbeek, editors, *9th Int. Conf. on Automated Deduction (CADE)*, number 310 in Springer LNCS, pages 415–434, Argonne, IL, USA, 1988. Springer-Verlag.

[McC88] William McCune. An indexing method for finding more general formulas. *Association for Automated Reasoning Newletter*, 9(1):7–8, January 1988.

[Nil92] Ulf Nilsson. *Abstract Interpretation & abstract machines: contributions to a methodology for the implementation of logic programs.* PhD thesis, Linköping University, Sweden, 1992.

[Nil93] Ulf Nilsson. Towards a methodology for the design of abstract machines for logic programming languages. *Journal of Logic Programming*, 16(1–2):163–189, 1993.

[Pel98] Nicolas Peltier. A new method for automated finite model building exploiting failures and symmetries. *Journal of Logic and Computation*, 8(4):511–543, 1998.

[PH97] John Peterson and Kevin Hammond. Haskell 1.4: A non-strict, purely functional language. http://www.haskell.org/onlinereport/, April 1997.

[PNO97] Simon Peyton Jones, Thomas Nordin, and Dino Oliva. C--: A portable assembly language. http://www.dcs.gla.ac.uk/~simonpj/pal-ifl.ps.gz, November 1997.

[RRS+95] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient tabling mechanisms for logic programs. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, pages 697–711. MIT Press, 1995.

[SG96] Heribert Schütz and Tim Geisler. Efficient model generation through compilation. In Michael McRobbie and John Slaney, editors, *13th Int. Conf. on Automated Deduction (CADE)*, number 1104 in Springer LNAI, pages 433–447. Springer-Verlag, 1996.

[SG97] Heribert Schütz and Tim Geisler. Efficient model generation through compilation. Technical Report PMS-FB-1997-21, Institut für Informatik, LMU München, 1997. Accepted for publication. Extended version of [SG96].

[Sla92] J. Slaney. FINDER (Finite Domain Enumerator): Notes and guide. Technical Report TR-ARP-1/92, Australian National University Automated Reasoning Project, Canberra, 1992.

[SSY94] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Automated Deduction — CADE-12*, number 814 in Springer LNAI, pages 252–266, Nancy, France, 1994. Springer-Verlag.

[vDGPS98] Alexander von Drach, Tim Geisler, Sven Panne, and David Sacher. SatAM – An abstract machine for model generation with PUHR tableaux. Technical Report PMS-FB-1998-14, Institut für Informatik, LMU München, October 1998.

[War83] D.H.D. Warren. An abstract PROLOG instruction set. Technical Report 309, SRI, 1983.

[ZZ95] Jian Zhang and Hantao Zhang. SEM: A system for enumerating models. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.