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

Ludwig
Maximilians-
Universität
München

Satchmo
The Compiling and Functional Variants

Tim Geisler, Sven Panne, Heribert Schütz

published in *Journal of Automated Reasoning 18(2), p. 227–236, 1997*
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1996-19, Dezember 1996

Satchmo

The Compiling and Functional Variants

TIM GEISLER, SVEN PANNE and HERIBERT SCHÜTZ
Institut für Informatik, Universität München
München, Germany
{*Tim.Geisler,Sven.Panne,Heribert.Schuetz*}*@informatik.uni-muenchen.de*

Abstract. Compiling Satchmo and Functional Satchmo are two variants of the model generator Satchmo, incorporating enhancements in different directions. Compiling Satchmo is based on the observation that Satchmo (like any model generator or theorem prover) can be seen as an interpreter for a program given as a logical theory, and that this interpretation layer can be avoided by compilation of the theory into a directly executable program. Functional Satchmo is an implementation of Satchmo's calculus in a purely functional language supporting lazy evaluation.

Key words: Automated theorem proving, competition, Satchmo, compilation, incremental evaluation

1. Introduction

1.1. Motivation

The model generator Satchmo [10] was originally developed to test the satisfiability of integrity constraints in deductive databases.

Satchmo does not terminate before it has generated every minimal Herbrand model of a given clausal theory. In particular, if it does return without generating a Herbrand model, we know that there is none. Consequently, by Herbrand's Model Theorem the input theory is unsatisfiable. Therefore it is possible to (ab)use Satchmo as a theorem prover, which allowed us to participate in the ATP competition. It should, however, be noted that theorem proving is not the primary purpose of a model generator. Variants of Satchmo that fulfill a certain fairness condition are *proof-complete*, i. e., they detect the non-existence of (Herbrand) models of a clausal theory in finite—albeit perhaps very long—time.

The original variants of Satchmo are implemented in Prolog. They re-use rather than re-implement several features of the Prolog system, in particular term data structures, unification, and backtracking. Therefore these programs are extremely short: just about ten Prolog rules. Furthermore, they take advantage of much of the development effort that has gone into an efficient implementation of Prolog. An example of such an implementation of Satchmo is given in Section 3 (Figure 1).

Compiling Satchmo and Functional Satchmo,[1] the two variants of Satchmo that participated in the competition, incorporate enhancements in different directions:

Compiling Satchmo: The observation that any model generator or theorem prover can be seen as an interpreter of a program (the given theory) suggests the use of compilation techniques to gain efficiency. The calculus underlying Satchmo turned out to be suitable for this, which lead to the development of Compiling Satchmo [15]. The code generated by the compiler re-uses features of the Prolog system in a similar way as the original variants of Satchmo do.

Functional Satchmo: An implementation tightly embedded in Prolog lacks the flexibility that is needed for several intended modifications, including extensions of functionality and optimizations. Without re-using Prolog features the motivation to use this language diminished. As a reference implementation for a programming course for undergraduate students, we developed a clean and simply structured variant of Satchmo in a purely functional subset of Scheme. From this implementation the variant that participated in the competition has been developed.

1.2. Related Systems

There are several other approaches for generating models of clausal theories.

One such approach, MGTP [5], is based on the calculus of Satchmo. Its developers have, however, put emphasis on other aspects, e. g., parallel execution. MGTP can, like Satchmo, be used as a theorem prover.

Model generators like FINDER [16], SEM [19], and MACE [11] differ from Satchmo and MGTP in that they attempt to generate models with a given finite domain size rather than Herbrand models. If such a system fails to produce a model of a theory, this is not a proof of the theory's unsatisfiability. Therefore, these model generators cannot be used as theorem provers.

Some other approaches (e. g., [4]) for the generation of Herbrand models are based on a theorem prover, which attempts to prove the unsatisfiability of a theory. If this finitely fails, a model can be extracted from the set of clauses generated by the prover.

1.3. Applications

Currently we are investigating the usage of Satchmo for two applications: test pattern generation for digital circuits and analysis of complex dependencies in cell biology.

[1] Available at "http://www.pms.informatik.uni-muenchen.de/software/".

2. Architecture

2.1. Calculus

All variants of Satchmo implement *positive unit hyperresolution (PUHR) tableaux* [1], a calculus for *range-restricted* (see below) clauses. We write clauses in implication form, i.e., as $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$ with atoms A_i and B_j. We call $A_1 \wedge \ldots \wedge A_n$ the *body* and $B_1 \vee \ldots \vee B_m$ the *head* of the clause. A clause is range-restricted if each of its variables occurs in the body. Clauses can be made range-restricted by introducing body literals with an auxiliary predicate enumerating the Herbrand universe [10, 1]. PUHR tableaux are trees in which every node except for the root is either a "u-node" or a "p-node". u-nodes are labeled with ground atoms ($u$nits) and p-nodes are labeled with $p$ositive ground clauses. PUHR tableaux for a given clause set are built starting from the tree consisting of a single unlabeled node, by repeatedly applying the following two rules:

(PUHR rule) If $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$ is a ground instance of an input clause (the *nucleus*) and there is some branch containing u-nodes with labels $A_1, \ldots, A_n$ (the *electrons*), then add a p-node labeled with $B_1 \vee \ldots \vee B_m$ (the *(hyper-)resolvent*) as the single child to the final node of this branch.

(Splitting rule) If there is some branch containing a p-node with label $B_1 \vee \ldots \vee B_m$, but no u-node labeled with B_1 or with $\ldots$ or with B_m, then add m u-nodes labeled with $B_1, \ldots, B_m$ as children to the final node of this branch.[2] As a borderline case, if the label of the chosen p-node is the empty clause ($m = 0$), the branch is marked as *closed*.

A PUHR tableau is *closed* if all its branches are closed. A branch or a PUHR tableau is *open* if it is not closed.

In the sequel we will frequently identify a branch of a PUHR tableau with the set of atoms labeling the u-nodes of the branch. An open branch is *saturated* if for every ground instance of an input clause some body atom does not occur in the branch or some head atom does. A PUHR tableau is *saturated* if all its open branches are saturated.

A saturated open branch is a Herbrand model of the input clauses.[3] All minimal Herbrand models (but possibly some non-minimal ones, too) occur as branches in every saturated PUHR tableau. Therefore, a saturated PUHR tableau is closed iff the given clausal theory is unsatisfiable.

[2] This rule is a variant of the usual β-rule for tableau calculi with a regularity condition.
[3] As usual, a Herbrand interpretation is represented by the set of ground atoms it satisfies.

Complement splitting is a variant of the above splitting rule that retains these properties. A newly appended u-node is not only labeled by an atom but also by the complements of the positive labels of all its right siblings. A branch is now marked as closed not only by a splitting step for the empty clause, but also if the set of the labels of its u-nodes contains two complementary literals. Complement splitting is currently not supported by Functional Satchmo. For Compiling Satchmo it was enabled in the competition.

2.2. Control

To avoid repeated application of the PUHR rule with the same nucleus and electrons, we apply the PUHR tableau calculus *incrementally*:

1. Starting from the initial tableau (which consists of a single unlabeled node) all possible applications of the PUHR rule without electrons (i. e. with positive nuclei) are performed.

2. Whenever a u-node has been appended to a branch by a splitting step, the PUHR rule is applied to this branch in all possible ways where (at least) one of the electrons is the label of this new u-node.

3. After such a set of PUHR steps has been performed, and if the branch is not saturated, some p-node is chosen for splitting.

The PUHR tableau calculus is *proof-confluent,* i. e., if the input clause set is unsatisfiable (and therefore *some* PUHR tableau for a set of clauses is closed), then *every* PUHR tableau for this set can be expanded into a closed tableau. Proof confluence is perserved by complement splitting and by incremental application of the calculus. Therefore, at any time an arbitrary allowed inference step can be applied to a tableau irrevocably.

Nevertheless, to ensure termination for unsatisfiable clausal theories, the selection strategy for inference steps must be fair in the following sense:

- Every PUHR step that is applicable in some branch b must eventually be applied in all branches extending b.

- Every splitting step that is applicable in some branch b must become inapplicable in all branches extending b after a finite number of PUHR steps or splitting steps.

Incremental application of the PUHR tableau calculus obviously fulfills the first condition. The second condition is fulfilled by another refinement of the calculus, which is described in the rest of this section.

Note that the splitting rule can be applied to a p-node at most once in a branch. Therefore it is useful to maintain along every branch a set S of

p-nodes for which splitting has not yet been applied. Functional Satchmo simply implements S as a queue to ensure fairness.

Compiling Satchmo provides an option (which was enabled for the competition) to ensure fairness in a more sophisticated way: If in an input clause some variable occurs in the head at a deeper term nesting level than in the body (as, e. g., the X in $p(X) \rightarrow p(f(X))$), the clause is called *nesting*. Otherwise it is called *non-nesting*. The set S is partitioned into S_n and S_{nn} according to whether the nucleus of the PUHR step that generated the p-node was nesting or non-nesting. Then p-nodes from S_{nn} are preferred over p-nodes from S_n. Within S_{nn} newer p-nodes are preferred over older ones. Within S_n older p-nodes are preferred over newer ones. That is, S_{nn} and S_n are implemented as a stack and a queue, respectively.

This technique for ensuring fairness has empirically proven to be far more efficient than a queue for the entire set S. We have no fully convincing explanation for this behavior.

Finally, note that the preferred selection of p-nodes labeled with the empty clause for splitting (which closes the branch) never sacrifices fairness. Since such a modification of the above strategies is always an optimization, it is applied by both Functional and Compiling Satchmo.

2.3. Tuning for the Competition

Both Compiling Satchmo and Functional Satchmo were submitted to the competition without any special tuning.

3. Implementation

A slightly modified version of the original implementation [10] of Satchmo is given in Figure 1. This program applies the PUHR tableau calculus in a way which is neither incremental nor fair. It expects a set of clauses to be given as Prolog facts of the form A_1`, ..., `A_n `--->` B_1`; ...; `B_m, where the atoms A_i and B_j are directly represented as Prolog terms. The empty body and the empty head are written as `true` and `false`.

The program is started by a call to `saturate`. This procedure will be called again at every node of the generated tableau. The generated PUHR tableau is represented by Prolog's search tree: Successful and failing branches of this search tree correspond to saturated and closed tableau branches, respectively. At any point of time only one branch is represented by data structures: The u-node labels for the current branch are stored in Prolog's dynamic database. Labels of p-nodes (resolvents) are represented by bindings for the variable `Rsv`.

```
saturate :- (puhr(Rsv), \+ Rsv) -> (split(Rsv), saturate) ; true.

puhr(Head) :- (Body ---> Head), Body.

split(false) :- !, fail.
split(X ; Y) :- !, (split(X) ; split(Y)).
split(Atom)  :-    assume(Atom).

assume(Atom) :- asserta(Atom).
assume(Atom) :- retract(Atom), !, fail.
```

Figure 1. Satchmo implemented as an interpreter in Prolog.

3.1. Compiling Satchmo

PUHR tableaux are well suited for compilation: Both the PUHR rule and the splitting rule can be compiled, because the nuclei and the structure of the resolvents are known at compile time.

Compiling Satchmo first compiles the given clausal theory to a set of Prolog rules. These rules are then optionally further compiled by the Prolog system (e. g., to WAM code or to native machine code). The generated code is executed either by a Prolog or WAM code interpreter or by the hardware.[4]

At runtime, the PUHR tableau is mapped to Prolog's search tree in a similar way to the original Satchmo implementation (Figure 1). In particular, the u-node labels of the current tableau branch are again stored in the dynamic database.

Note that all generated positive clauses are ground instances of heads of input clauses. Therefore handling of complex data structures for positive clauses at runtime can be (and is) avoided. A positive clause is represented by a term whose functor uniquely identifies the respective input clause and whose arguments give the instantiations of the variables appearing in the input clause's head.

Compiling Satchmo has some parameters that control the amount of compilation to be done, which kind of control to use, and whether to use complement splitting. In the competition, the highest compilation level, the fairness technique described in Section 2.2, and complement splitting were used.

Compiling Satchmo currently runs on top of the Prolog systems ECLiPSe, SICStus, and SWI Prolog, but can easily be ported to any other reasonable Prolog system. SICStus Prolog v3 was used for the competition. Compilation to WAM rather than to native code was used because typically the compilation times for the latter would have consumed a large share of the time limit imposed for the competition.

[4] The compilation can also be performed by a partial evaluator supplied with a suitable interpretive variant of Satchmo and the input clauses. However, experiments with the partial evaluator Mixtus [14] have shown that this is far too slow even for small examples.

3.2. Functional Satchmo

The kernel of the present implementation of Functional Satchmo in the language Haskell [13] is a direct reimplementation of the Scheme variant used for teaching. Haskell is a modern purely functional language performing lazy evaluation, thereby avoiding unnecessary computations in many cases while keeping the notation of algorithms simple. Furthermore, it provides some convenient features like pattern matching, polymorphic typing and modules.

Functional Satchmo is able to parse first-order formulas as well as clauses in an Otter-like syntax [12]. After transformation to range-restricted clausal form, this theory is interpreted.

A PUHR tableau is represented as a list of its open branches, which are in turn implemented as balanced trees of atoms (the u-node labels). To be able to return a model before the whole tableau is expanded, a *lazy* list is needed. In the original Scheme implementation, a *stream,* Scheme's analogue of a lazy list, was used. The full impact of Haskell's laziness on Functional Satchmo's runtime behavior remains to be investigated.

Incremental evaluation is done in the following way: Suppose a u-node with label l has been appended to a branch by a splitting step. This l is now resolved in all possible ways with the input clauses. Using the resolvents instead of the original set of input clauses ensures that one of the electrons in the next PUHR step is in fact l.

The implementation language for Functional Satchmo is Haskell 1.3 [13], using version 0.29 of the Glasgow Haskell Compiler for the competition.

4. Performance

4.1. Comments on Performance in the Competition

In the ATP competition, Compiling Satchmo and Functional Satchmo brought up the rear in the category "General Hardware, Monolithic Systems, Mixed Problems". All other participating provers, except for two provers from the "Special Hardware" category, had significantly better results. In contrast, both Satchmo variants passed most of the soundness tests with satisfiable problems [18] very fast.

From the results of earlier experiments [15] with Compiling Satchmo, it was expected that both Satchmo variants would perform well only on range-restricted problems and on non-nesting problems.

In the ATP competition, both variants of Satchmo behaved as expected. No range-restricted problem and just three non-nesting problems were among the (unsatisfiable) problems selected for the competition. The three non-nesting problems[5], which were in fact function free, could be solved by either variant of Satchmo in a time comparable to the other competitors. Both variants had great difficulties with the non-range-restricted problems which contain equality clauses or nesting clauses. Only three of these problems could be solved by Compiling Satchmo.

Most of the failures of the Satchmo variants to find a proof result from insufficient memory, which is in part due to too simple data structures. A further reason for the enormous memory usage is the derivation of too many unnecessary ground atoms which is in turn caused by the simple-minded transformation to range-restricted form. With non-range-restricted clause sets, this transformation and the application of the PUHR tableau calculus does not seem to be well suited for theorem proving applications.

Although Compiling Satchmo and Functional Satchmo build different PUHR tableaux and therefore usually need significantly different times to find a proof, Functional Satchmo solved only a proper subset of the problems Compiling Satchmo solved.

4.2. Experimental Results Outside the Competition

In a comparison [15] of Compiling Satchmo with MGTP/G [8], Otter [12] and SETHEO [6], using the TPTP Problem Library [17] as a benchmark, Compiling Satchmo has turned out to be the most efficient for range-restricted problems and for non-nesting problems, i. e., precisely those for which Satchmo had been designed.

Experimental results with Functional Satchmo after the competition show a significant speedup with a representation of tableau branches as *standard discrimination trees* [7]. With this improved data structure, stack-like handling of non-nesting clauses (like Compiling Satchmo), and a technique for fair selection of inference steps preferring splitting steps involving fewer children, Functional Satchmo is able to solve eight of the problems selected for the ATP competition including all problems Compiling Satchmo was able to solve.

5. Conclusion

5.1. Strengths and Weaknesses

We participated in the ATP competition because Satchmo can, in principle, be used for theorem proving and we wanted to get more results on how it compares on this task with specialized theorem provers. It has been

[5] SYN200-1, SYN202-1, and SYN271-1.

confirmed that Satchmo is not suited for classical theorem proving applications from various branches of mathematics, which contain equality and non-range-restricted clauses.

PUHR tableaux as implemented in Compiling Satchmo and Functional Satchmo are, however, well suited for (minimal) model generation for range-restricted clauses. Applications with a database flavor typically can be formalized with range-restricted clauses. Although both variants have considerably more lines of code than the original variant of Satchmo[6], they are nonetheless short enough to be easily modified and included in other applications. Furthermore, the Prolog code generated by Compiling Satchmo remains understandable.

Although the current implementations have no representation of a proof object, it is quite easy to represent an entire PUHR tableau explicitly in memory.

5.2. Future Plans

There are many possible directions for further developments. From a theorem proving perspective, extensions to be investigated and implemented include a more efficient treatment of non-range-restricted clauses and reasoning with equality [3]. In addition, bidirectional search as in SATCHMORE [9] can accelerate proofs of Satchmo.

From a logic programming perspective, we would like to support non-monotonic features such as aggregation and negation as failure, as well as access to predicates defined outside the clausal theory. We would also like to generate *all* minimal Herbrand models of a theory, thereby avoiding the generation of non-minimal Herbrand models [1].

From an efficiency perspective, better data structures such as term indexes [7] have to be used. Further optimizations, e. g., techniques making use of heuristics, symmetries, or functional dependencies, should be investigated. Currently compilation to a language different from Prolog, e. g., to an abstract machine language tailored to the specific needs of model generation, is being considered.

Acknowledgments

We thank Norbert Eisinger, Geoff Sutcliffe, and the anonymous referees for valuable comments. The support for the third author by the Bayerischer Habilitations-Förderpreis is appreciated.

---

[6] Compiling Satchmo has about 800 lines of code. Functional Satchmo has about 1500 lines of code, but only a few hundred of them were really needed for the competition.

References

1. F. Bry and A. Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In *Proceedings of the 5th Workshop on Theorem Proving with Tableaux and Related Methods*, number 1071 in Lecture Notes in Artificial Intelligence, pages 143–159. Springer-Verlag, 1996.
2. A. Bundy, editor. *Proceedings of the 12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994.
3. M. Denecker and D. De Schreye. On the duality of abduction and model generation in a framework for model generation with equality. *Theoretical Computer Science*, 122(1–2):225–262, 1994.
4. C. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2):173–203, 1996.
5. H. Fujita and R. Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 535–548. MIT Press, 1991.
6. C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent devleopments. In [2], pages 778–782.
7. P. Graf. *Term Indexing.* Number 1053 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
8. Institute for New Generation Computer Technology. *Model Generation Theorem Prover: MGTP*, 1995. http://www.icot.or.jp/ICOT/IFS/IFS-abst/081.html.
9. D.W. Loveland, D.W. Reed, and D.S. Wilson. SATCHMORE: SATCHMO with RElevancy. *Journal of Automated Reasoning*, 14:325–351, 1995.
10. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 415–434. Springer-Verlag, 1988.
11. W.W. McCune. A Davis-Putnam program and its application to finite first-order model seach: Quasigroup existence problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, Argonne, IL, USA, 1994.
12. W.W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL 94/6, Argonne National Laboratory, Argonne, IL, USA, 1994.
13. J. Peterson and K. Hammond. Haskell 1.3 – a non-strict, purely functional language. Research Report YALEU/DCS/RR-1106, Yale University, New Haven, CT, USA, 1996.
14. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* SICS Dissertation Series 04, The Royal Institute of Technology (KTH), Stockholm, Sweden, 1991.
15. H. Schütz and T. Geisler. Efficient model generation through compilation. In M. McRobbie and J. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction*, number 1104 in Lecture Notes in Artificial Intelligence, pages 433–447. Springer-Verlag, 1996.
16. J. Slaney. FINDER: Finite domain enumerator – system description. In [2], pages 252–266.
17. G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In [2], pages 252–266.
18. G. Sutcliffe and C.B. Suttner. The Procedures of the CADE-13 ATP System Competition. *Journal of Automated Reasoning*, This issue, 1997.
19. J. Zhang and H. Zhang. SEM: a System for Enumerating Models. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 298–303, 1995.