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

Ludwig
Maximilians-
Universität
München

# Efficient Model Generation through Compilation

**Heribert Schütz, Tim Geisler**

accepted for publication in *Journal of Information and Computation*
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-21, Juni 1997

# Efficient Model Generation through Compilation

Heribert Schütz and Tim Geisler

Universität München, Institut für Informatik, Oettingenstr. 67, D-80538 München
{Heribert.Schuetz|Tim.Geisler}@informatik.uni-muenchen.de

**Abstract.** We present a collection of simple but powerful techniques for enhancing the efficiency of tableau-based model generators such as Satchmo. The central ideas are to compile a clausal first order theory into a procedural Prolog program and to avoid redundant work of a naïve implementation.
We have compared various combinations of our techniques among each other and with theorem provers based on various calculi, using the TPTP Problem Library as a benchmark. Our implementation has turned out to be the most efficient for range-restricted problems and for a class of problems we call "non-nesting".

## 1 Introduction

Refutation-oriented theorem provers search for proofs for the unsatisfiability of a theory, whereas model generators search for proofs for the satisfiability of a theory, i.e., for models. Application areas for model generation include certain planning and configuration problems, finding counterexamples for conjectures and testing integrity constraints in databases. Finite failure of an exhaustive model generation procedure can also be used as a proof for the unsatisfiability of a theory if there is no need for a more direct proof. Recall that there are complete procedures for proving the unsatisfiability of first-order theories but there is no complete procedure for proving satisfiability.

The work presented in this paper is based on the model generator Satchmo [16]. We describe techniques to enhance the efficiency of model generation, most importantly compilation into a procedural Prolog program and avoidance of redundant computations.

We also give an efficient implementation for complement splitting, a method for minimizing the first generated model and for pruning the search space. Furthermore we show that it is not efficient to ensure fair selection of clauses by a purely breadth-first search strategy and present a far more efficient strategy.

We have evaluated our techniques among each other and with the theorem provers MGTP/G, Otter, and SETHEO, using the TPTP Problem Library as a benchmark. Our implementation has turned out to be the most efficient for range-restricted problems and for a class of problems we call "non-nesting" (see Section 4.1).

The paper is organized as follows: In Section 2 we briefly describe the basis from which our work started, i.e., Satchmo and its underlying calculus. Then we explain our central techniques for improving Satchmo's efficiency (incremental evaluation and compilation) in Section 3 and further refinements in Section 4. The performance evaluation is given in Section 5. After a discussion of related work in Section 6 we conclude the paper in Section 7, suggesting possible directions for future research.

## 2 Preliminaries

### 2.1 PUHR Tableaux

Satchmo has been formalized with positive unit hyperresolution (PUHR) tableaux for clausal theories [5]. For the rest of this paper let $\mathcal{S}$ be a set of clauses. By $\mathrm{gnd}(\mathcal{S})$ we denote the set of all ground instances of clauses in $\mathcal{S}$. 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. We write empty bodies and heads as $\top$ and $\bot$, respectively.

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 clause set $\mathcal{S}$ are inductively defined as follows:

1. The tree consisting of a single unlabeled node is a PUHR tableau for $\mathcal{S}$.
2. Let $T$ be a PUHR tableau for $\mathcal{S}$.

   **(PUHR rule)** If $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$ is a ground instance of a clause (the *nucleus*) in $\mathcal{S}$ and there is some branch containing $u$-nodes with labels $A_1, \ldots, A_n$ (the *electrons*), then the tree obtained from $T$ by adding 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 is a PUHR tableau for $\mathcal{S}$. This rule can be visualized as follows:

$$\frac{\begin{array}{c} u : A_1 \\ \vdots \\ u : A_n \end{array}}{p : B_1 \vee \ldots \vee B_m} \quad A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m \ \in \ \mathrm{gnd}(\mathcal{S})$$

   **(Splitting rule)** If there is some branch containing a $p$-node with label $B_1 \vee \ldots \vee B_m$, but no $u$-nodes labeled with $B_1$ or ... or $B_m$, then the tree obtained from $T$ by adding $m$ $u$-nodes labeled with labels $B_1, \ldots, B_m$ as children to the final node of this branch is a PUHR tableau for $\mathcal{S}$.
   This rule, which is a variant of the usual $\beta$-rule for tableau calculi with a regularity condition, can be visualized as follows:

$$\frac{\begin{array}{c} p : B_1 \vee \ldots \vee B_m \\ \cancel{u : B_1} \\ \vdots \\ \cancel{u : B_m} \end{array}}{u : B_1 \ | \ \cdots \ | \ u : B_m}$$

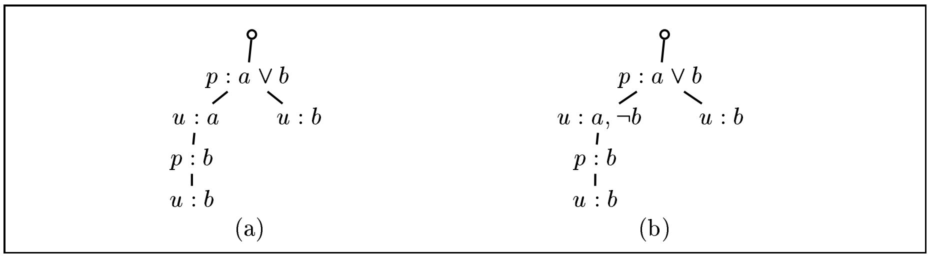


**Figure 1.** A PUHR tableau (a) and a PUHR-CS tableau (b).

3. There are no other PUHR tableaux for $\mathcal{S}$.

A branch of a PUHR tableau for $\mathcal{S}$ is *closed* if it contains a $p$-node labeled with the empty clause $\perp$. A PUHR tableau is *closed* if all its branches are closed. A branch or a PUHR tableau is *open* if it is not closed.

We will now identify a branch of a PUHR tableau with the set of $u$-node labels along the branch. An open branch is *saturated* if for every ground instance of a clause in $\mathcal{S}$ some body atom does not occur in the branch or some head atom does. A PUHR tableau is *saturated* if all of its open branches are saturated.

The PUHR-tableaux proof procedure has the following properties [5]:

1. A saturated open branch of a PUHR tableau for $\mathcal{S}$ is a Herbrand model[1] of $\mathcal{S}$ ("model soundness"). As a consequence, if $\mathcal{S}$ is unsatisfiable, then every saturated PUHR tableau for $\mathcal{S}$ is closed ("refutation completeness").
2. A model of $\mathcal{S}$ satisfies some open branch of every PUHR tableau for $\mathcal{S}$ ("model completeness"). As a consequence, if $\mathcal{S}$ is satisfiable, then there is no closed PUHR tableau for $\mathcal{S}$ ("refutation soundness").
3. Every minimal Herbrand model of $\mathcal{S}$ occurs as an open branch in every saturated PUHR tableau for $\mathcal{S}$ ("minimal model completeness").

Because of these properties, PUHR tableau procedures should try to construct a saturated PUHR tableau, but they need not care which one they construct. The PUHR-tableau calculus is confluent in the sense that applications of tableau rules never need to be undone in an attempt to construct some saturated PUHR tableau.

### 2.2 Complement Splitting

PUHR tableaux can be used to find the minimal Herbrand models of a clausal theory, but they will in general also produce non-minimal Herbrand models. For example, for the clauses $\{\top \rightarrow a \vee b, a \rightarrow b\}$ the saturated PUHR tableau in Figure1(a) has a non-minimal Herbrand model as its saturated open left branch.

[1] As usual, we identify a Herbrand interpretation with the set of ground atoms it satisfies.

We can avoid this and many similar cases by allowing $u$-nodes of the tableau to be labeled by several negative ground literals in addition to the positive ground literal. The tableau splitting rule is modified in such a way that a newly appended $u$-node is not only labeled by an atom $B_i$, but also by the complements $\neg B_{i+1}, \ldots, \neg B_m$ of the positive labels of all its right siblings [16, 5].

The modified splitting rule can be visualized as follows:

$$
\begin{array}{l|l|c|l|l}
\multicolumn{5}{c}{p : B_1 \vee \ldots \vee B_m} \\
\multicolumn{5}{c}{\cancel{u : B_1}} \\
\multicolumn{5}{c}{\vdots} \\
\multicolumn{5}{c}{\cancel{u : B_m}} \\
\hline
u : \quad B_1 & u : \quad B_2 & & u : B_{m-1} & u : B_m \\
\qquad \neg B_2 & \qquad \neg B_3 & & \qquad \neg B_m & \\
\qquad \vdots & \qquad \vdots & \ldots & & \\
\qquad \neg B_{m-1} & \qquad \neg B_m & & & \\
\qquad \neg B_m & & & &
\end{array}
$$

A branch is now considered to be closed not only if it contains a $p$-node labeled with the empty clause $\perp$, but also if the union of the labels of its $u$-nodes contain complementary literals $A$ and $\neg A$. We call tableaux generated in this way *PUHR-CS tableaux.*

The left branch of the saturated PUHR-CS tableau for $\{\top \rightarrow a \vee b, a \rightarrow b\}$ in Figure1(b) is closed because the union of the labels of its $u$-nodes contains $b$ and $\neg b$.

We identify a branch of a PUHR-CS tableau with the set of *positive* labels of its $u$-nodes. Then properties 1 to 3 of the PUHR-tableaux proof procedure given in the previous section also hold for PUHR-CS tableaux. Furthermore, PUHR-CS tableaux have the property that a saturated open branch cannot be a subset of any saturated open branch to the left of it. Therefore, the *leftmost* open branch of a saturated PUHR-CS tableau for $\mathcal{S}$ is always a minimal Herbrand model of $\mathcal{S}$.

### 2.3 Range Restriction

All the implementations of PUHR tableaux given in this paper will require that the clauses in $\mathcal{S}$ are *range-restricted*, i.e., that the variables occurring in the head of a clause also occur in its body. The main reason for this is that otherwise a resolvent of a hyperresolution step contains variables. Such a resolvent, e.g., $p(X) \vee q(X)$, can be read as $\forall X(p(X) \vee q(X))$, but splitting it essentially means to transform it into $\forall X p(X) \vee \forall X q(X)$, which is not equivalent to the previous formula.

We expect (and have experienced) that many real world problems (e.g., planning and configuration problems, integrity checking in databases) can be naturally formalized by range-restricted clauses. Other clause sets are transformed into range-restricted clause sets by introducing an auxiliary predicate enumerating the Herbrand base [16, 5].

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

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

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

update(Atom) :- assume(Atom), saturate.

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

**Figure 2.** Basic Satchmo.

```
/* c1 */ true ---> edge(a, b).        /* c2 */ true ---> edge(b, c).
/* c3 */ true ---> edge(c, d).        /* c4 */ true ---> edge(d, e).

/* c5 */ true ---> traverse(a).
/* c6 */ traverse(X), edge(X, Y) ---> traverse(Y); stop(Y).

/* c7 */ traverse(d) ---> false.
```

**Figure 3.** An example set of clauses.

## 2.4 Basic Satchmo

A simple Prolog implementation of PUHR tableaux for range-restricted clauses is given in Figure2. We call this program *Basic Satchmo*. It is basically the original implementation of Satchmo [16], but has been adapted to the terminology of PUHR tableaux and modified to be more similar to the other Satchmo variants we will develop in this paper.

The clause set $\mathcal{S}$ should be given as Prolog facts of the form "$A_1$`,` ..., $A_n$ `--->` $B_1$`;` ...`;` $B_m$`.`", where "`--->`" is an infix operator with lower precedence than "`,`" and "`;`". $\top$ and $\bot$ are represented by `true` and `false`. See Figure3 for an example clause set: `traverse` traverses a graph given by `edge`, starting from a node `a`. We may stop after every edge, and we should not continue traversing at node `d`.

The partial search tree traversed by this Prolog program up to any point of time roughly corresponds to a PUHR tableau. Successful and failing branches of the Prolog search space correspond to saturated and closed tableau branches, respectively.[2] The $u$-node labels for the current branch are stored in Prolog's dynamic database, which should initially be empty.

When the top-level procedure[3] `saturate` is called, `puhr` looks for a clause instance whose body atoms occur in the current branch (i.e., the dynamic

[2] Because of the confluence of the PUHR-tableau calculus, Prolog's backtracking mechanism need not be used for traversing a search space induced by the posibility to expand tableaux in several different ways. Thus it can be used to traverse the different branches of a tableau.

[3] We will speak of *predicates* and *clauses* when refering to a logical theory, and of *procedures*, *rules*, and *facts* when refering to a Prolog program.

database)[4] and `saturate` tries to split the resolvent immediately. According to the splitting rule it is first checked whether the resolvent is already subsumed by an atom. If so, we backtrack to find a resolvent which is not subsumed. If no such resolvent exists, a model has been found and `saturate` returns successfully. Otherwise `split` is called to split the branch according to the resolvent. With the empty resolvent `false` the branch is closed. With a disjunctive resolvent the branch is split by generating branches for all disjuncts. A new leaf $u$-node (in `update`) is labeled by asserting the respective atom in the dynamic database and `saturate` is called again to handle further clause instances. Asserted atoms will be retracted on backtracking. In general `saturate` is resatisfiable, yielding more models on backtracking.

## 3 Efficient Implementation

In this section we describe the central techniques for an efficient implementation of model generation: incremental evaluation and compilation.

### 3.1 Incremental Evaluation

A major drawback of Basic Satchmo is that in every iteration of `saturate`, i.e., whenever a resolvent is needed for a splitting step, `puhr` is called in such a way that it typically repeats work done in earlier iterations, because many clauses are examined whose body is not affected by atoms added to the dynamic database recently. This is due to the fact that a PUHR step that has once been applicable in a branch will always remain applicable.

We can avoid this repeated work if we specifically look for PUHR steps that have just become possible due to a newly derived atom, and collect all the corresponding resolvents. This is done by the program *Incremental Satchmo* (Figure4; together with unchanged procedures `assume` and `split` from Figure2).

In the beginning we find all clauses with empty bodies, collect their heads, which are the trivial hyperresolvents, and "fire" them one after the other. "Firing" a resolvent means to test whether it already holds, i.e., if one of its atoms already appears in the current branch, and using it in a splitting step if not. Whenever (in `update`) we add a new atom `Atom` to a branch, we look (in `delta` and `delta_body`) for new nuclei and hyperresolvents that can be derived using `Atom` as one electron and with the remaining electrons also appearing in the branch. We again fire all found resolvents rather than only the first one.

If a disjunctive resolvent leads to branch splitting, later resolvents in the accumulated list will have to be processed in all subbranches. On the other hand, if some resolvent is empty (`false`) and a branch is closed, later resolvents will be ignored.

Note the different nature of choicepoints and resatisfiability in `delta` and `delta_body` on the one hand and in `split`, `fire`, `saturate`, and `update` on the

[4] Note that in the procedure `puhr` the argument `Head` is fully instantiated by the call to `Body` only for range-restricted clauses.

```
saturate :- findall(Head, (true ---> Head), Rsvs),
            fire_list(Rsvs).

fire_list([]).
fire_list([Rsv | Rsvs]) :- fire(Rsv), fire_list(Rsvs).

fire(Rsv) :- Rsv -> true ; split(Rsv).

update(Atom) :-  assume(Atom),
                 findall(Rsv, delta(Atom, Rsv), Rsvs),
                 fire_list(Rsvs).

delta(Atom, Head) :- (Body ---> Head), delta_body(Atom, Body).

delta_body(Atom, (X, Y)) :- !, ( delta_body(Atom, X), Y
                               ; X, delta_body(Atom, Y) ).
delta_body(Atom, Atom).
```

**Figure 4.** Incremental Satchmo.

other hand. In the latter case resatisfiability corresponds to different branches of a tableau, whereas in the former case it corresponds to different "activated" clause instances in a single branch. We use the higher-order Prolog procedure `findall` to convert the choicepoints (i.e., disjunctions) from `delta` into lists and the recursive procedure `fire_list` to convert the lists into conjunctions.

### 3.2 Compilation

A set of clauses can be seen as a program interpreted by Satchmo. We avoid this interpretation level by compiling the clause set into a Prolog program that can be executed directly. Compilation is done by specializing an interpreter (i.e., Incremental Satchmo) for the program (i.e., the set of clauses) to be executed. We call this approach *Compiling Satchmo*. The compilation can be described by the following steps:

1. We specialize `saturate` for the clauses with empty body, `fire` for the clause heads (of which the resolvents are instances), `update` for the predicates, and `delta` for the body atoms occurring in $\mathcal{S}$. To do so, we unfold the calls to `--->`, `findall`, and `fire_list` in `saturate`, the call to `split` in `fire`, and the calls to `--->` and `delta_body` in `delta`. With our example clause set (Figure3) this leads to the Prolog rules

```
saturate :- fire(edge(a, b)), ..., fire(traverse(a)).

fire(edge(a, b)) :- edge(a, b) -> true ; update(edge(a, b)).
...
fire(traverse(Y) ; stop(Y)) :-
        (traverse(Y) ; stop(Y)) ->
            true
        ;   (update(traverse(Y)) ; update(stop(Y))).
fire(false) :- false -> true ; fail.

update(edge(Arg1, Arg2)) :-
```

```
        assume(edge(Arg1, Arg2)),
        findall(Rsv, delta(edge(Arg1, Arg2), Rsv), Rsvs),
        fire_list(Rsvs).

delta(traverse(X), (traverse(Y); stop(Y))) :- edge(X, Y).
delta(edge(X, Y) , (traverse(Y); stop(Y))) :- traverse(X).
delta(traverse(d), false).
```

and a similar rule for `update(traverse(...))`. For predicates that do not occur in clause bodies (`stop` in our example) the call to `delta` will always fail. We can therefore simply omit the calls to `findall` and `fire_list` in the respective `update` rule.

2. To simplify the matching in `fire` we use *identifiers* for resolvents in `saturate`, `delta`, and `fire`. These identifiers consist of a clause identifier (given as comment in Figure3) for the nucleus and a tuple of the variables occurring in the nucleus' head. For the example we get rules like:

```
saturate :- fire(c1), ..., fire(c5).

fire(c1) :- edge(a, b) -> true ; update(edge(a, b)).

delta(traverse(X), c6(Y)) :- edge(X, Y).
```

3. We can now replace the procedures `update` and `delta` by specialized procedures `update_`*p* and `delta_`*p* for each predicate *p*. *p*'s arguments become top-level arguments of the specialized procedures. In a similar way we replace `fire` by a specialized procedure for every clause. `delta_`*p* must now return a complete call to these specialized procedures rather than resolvent identifiers, and `fire_list` must be modified to handle complete procedure calls. For the example we get:

```
saturate :- fire_c1, ..., fire_c5.

fire_c1 :- edge(a, b) -> true ; update_edge(a, b).
...
fire_c6(Y) :- (traverse(Y); stop(Y)) ->
                  true
              ;   (update_traverse(Y) ; update_stop(Y)).
fire_c7 :- false -> true ; fail.

fire_list([]).
fire_list([Call | Calls]) :- Call, fire_list(Calls).

update_edge(Arg1, Arg2) :-
        assume(edge(Arg1, Arg2)),
        findall(Rsv, delta_edge(Arg1, Arg2, Rsv), Rsvs),
        fire_list(Rsvs).

delta_traverse(X, fire_c6(Y)) :- edge(X, Y).
delta_edge(X, Y, fire_c6(Y)) :- traverse(X).
delta_traverse(d, fire_c7).
```

4. Since procedures like `fire_c7` for clauses with head `false` always fail, we eliminate them and replace them by `fail` in the respective procedures `delta_`*p*. In the example we get for clause `c7`:

```
delta_traverse(d, fail).
```

In order to close tableau branches earlier, we move a Prolog rule like this one towards the beginning of the definition of the respective procedure `delta_`$p$. Furthermore, we may add a "cut" to the rule.

We utilize symmetries in the domain enumeration clauses (if present) to generate more efficient code than for other clauses.

## 4 Further Refinements

In this section we describe extensions and further optimizations for the Satchmo implementations given above. We have implemented and tested these techniques for both Incremental and Compiling Satchmo. For the sake of simplicity, we will describe them in depth only for Incremental Satchmo (Figure4).

### 4.1 Tail Recursion and Fairness

When expanding a PUHR tableau, we have two degrees of freedom: We have to choose (1) which branch to expand and (2) which PUHR or splitting step to perform for this branch. To achieve saturated tableaux we need to be fair with respect to either choice. The variants of Satchmo described in previous sections use depth-first search strategies, which are not fair in general. In this section we develop an efficient method ensuring fairness w.r.t. choice 2.[5] It suffices to ensure fair application of the splitting step if after every splitting step all (finitely many) newly possible PUHR steps are performed immediately.

Note that in the absence of *nesting* clauses, i.e., clauses in which some variable occurs in the head at a deeper term nesting level than in the body, already depth-first search is fair.[6] We call such problems *non-nesting.*[7]

We will first transform the central group of mutually recursive procedures of Incremental Satchmo (`fire_list`, `fire`, `split`, and `update`) to make it tail-recursive. The tail-recursive variant can then easily be made fair.

**Tail Recursion:** Tail recursion is violated by the consecutive calls to `fire` and `fire_list` in the second rule defining `fire_list`. We eliminate the second of these calls: Instead of calling `fire_list` for the list of resolvents `Rsvs`, we pass this list on to the procedures `fire`, `split`, and `update`. In `update` we call `fire_list` anyway for a newly generated list of resolvents. Here we can simply concatenate the two lists and call `fire_list` only once. We also have to call `fire_list` in procedure `fire` if `Rsv` holds. The modified procedures are given in Figure5.

[5] We have found no similarly simple and efficient method ensuring fairness w.r.t. choice 1. But when Satchmo is used as a refutation procedure, we anyway have to close *all* branches.

[6] There are even weaker necessary conditions for the fairness of depth-first search, but testing these requires global analysis of the clause set.

[7] The class of range-restricted, non-nesting problems is called $\mathrm{PVD}_+$ in [8].

```
fire_list([]).
fire_list([Rsv | Rsvs]) :- fire(Rsv, Rsvs).

fire(Rsv, Rsvs) :- Rsv -> fire_list(Rsvs) ; split(Rsv, Rsvs).

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

update(Atom, OldRsvs) :- assume(Atom),
                         findall(Rsv, delta(Atom, Rsv), NewRsvs),
                         append(NewRsvs, OldRsvs, Rsvs),
                         fire_list(Rsvs).
```

**Figure 5.** Tail recursion for Incremental Satchmo.

```
saturate :- findall(Head, (true ---> Head), Rsvs),
            fire_list(Rsvs, []).

fire_list([], [])          :- !.
fire_list([], NL)          :- fire_list(NL, []).
fire_list([Rsv | CL], NL)  :- fire(Rsv, CL, NL).

fire(Rsv, CL, NL) :- Rsv -> fire_list(CL, NL) ; split(Rsv, CL, NL).

split(false,   CL, NL) :- !, fail.
split((X ; Y), CL, NL) :- !, (split(X, CL, NL) ; split(Y, CL, NL)).
split(Atom,    CL, NL) :-    update(Atom, CL, NL).

update(Atom, CL, NL) :- assume(Atom),
                        findall(Rsv, delta(Atom, Rsv), NewRsvs),
                        append(NewRsvs, NL, NewNG),
                        fire_list(CL, NewNG).
```

**Figure 6.** Layers for Incremental Satchmo.

**Fairness:** In the tail-recursive variant the set of resolvents yet to be split is organized as a stack, since we *pre*pend newly generated resolvents to older ones. We make the selection of clause instances fair by *ap*pending the newer resolvents (`append(OldRsvs, NewRsvs, Rsvs)`), turning the stack into a queue.

Unfortunately the complexity of `append` is linear in the length of its first argument. This is acceptable for `NewRsvs`, since the accumulated costs in a Satchmo run will then be proportional to the number of all generated resolvents. It is not acceptable for `OldRsvs`, i.e., in the fair case.

**Layers:** To overcome this problem we split the list of resolvents into a "current layer" and a "next layer" (Figure6). We can now efficiently *pre*pend the newly generated resolvents to the next layer (`NL`) without sacrificing fairness, since resolvents in the current layer (`CL`) are split first. Whenever the current layer is exhausted, the next layer is made the current one and a new next layer is initialized to the empty list. This is done by the second rule defining `fire_list`.

**Selective Assignment to Layers:** The fair variants of Satchmo described so far implement a breadth-first selection strategy. This strategy has turned out to

```
fire_list([], [])                :- !.
fire_list([], NL)                :-    fire_list(NL, []).
fire_list([delay(Rsv) | CL], NL) :- !, fire_list(CL, [Rsv | NL]).
fire_list([Rsv | CL], NL)        :-    fire(Rsv, CL, NL).

update(Atom, CL, NL) :- assume(Atom),
                        findall(Rsv, delta(Atom, Rsv), NewRsvs),
                        append(NewRsvs, CL, NewCL),
                        fire_list(NewCL, NL).

delta(Atom, Output) :- (Body ---> Head),
                       (nesting(Body, Head) -> Output = delay(Head)
                       ;                       Output = Head       ),
                       delta_body(Atom, Body).
```

**Figure 7.** Selective assignment to layers for Incremental Satchmo.

be far less efficient than a depth-first strategy for many clause sets not requiring fairness (as we will see in Figure9(a)). Therefore we also expected a speedup for clause sets requiring fairness by using breadth-first searching selectively for nesting clauses. That is, rather than inserting all newly generated resolvents into the next layer, we do so only for resolvents derived from PUHR steps with nesting clauses as nuclei[8] and insert other resolvents into the current layer. For technical reasons, we actually first insert all resolvents into the current layer, but some of them tagged by a functor `delay`. Later the newly added third rule for `fire_list` (Figure7) moves these resolvents from the current to the next layer, stripping off the `delay` tag. Of course, the compiler evaluates the test `nesting(Body, Head)` at compile time and chooses the respective case, thereby reducing considerable runtime overhead.

### 4.2 Complement Splitting

Inclusion of complement splitting into Basic or Incremental Satchmo is straightforward.[9] First, we parse disjunctions in clause heads as left-associative operators. Then in a disjunction $X \vee Y$ occurring in a resolvent the right alternative $Y$ will always be an atom. Before splitting $X$ we add $\neg Y$ to the corresponding tableau branch(es) by asserting neg($Y$). The second rule of the procedure `split` becomes:

```
split(X ; Y) :- !, (assume(neg(Y)), split(X) ; update(Y)).
```

Note that the recursive call `split(Y)` has been unfolded.

Whenever we assume a positive literal, we first check whether its complement has been assumed and close the current branch (i.e., fail) if necessary:

```
update(Atom) :- \+ neg(Atom), assert(Atom), ...
```

[8] We could be even more restrictive here (cf. Footnote 6).

[9] Note that complement splitting is orthogonal to the implementations of fairness given in Section 4.1 and can easily be combined with these.

`neg(`$A$`)` is never asserted after $A$, because `neg(`$A$`)` is only assumed when a resolvent containing $A$ is split, which is avoided if $A$ already occurs in the current branch. Thus we need not test for `Y` when assuming `neg(Y)` above.

As a small optimization in the compilation, we omit the test for `neg(`$p$`(...))` in `update` in those cases where every occurrence of $p$ in a clause head is in the leftmost atom (in the example of Figure3 this is true for `edge` and `traverse`), because in these cases no negative literals with predicate $p$ are ever assumed. To make better use of indexing in the dynamic database of certain Prolog systems, we replace terms of the form `neg(`$p$`(...))` by `neg_`$p$`(...)`.

## 5 Performance Evaluation

We have performed extensive benchmarks for different variants of Satchmo and also for other well-known theorem provers (MGTP/G [14], Otter [17], and SETHEO [11]). We report the most interesting results here.

There is no standard benchmark for model generation, but we also did not want to construct a set of benchmark problems ourselves to avoid a bias in favor of our techniques. Therefore we have chosen to use problems from the widely accepted TPTP Problem Library [23], although most of these problems have been constructed with refutation-oriented theorem provers in mind, therefore favoring such systems. This can be seen from the fact that only 44 of the 2729 TPTP problems are known to be satisfiable. Furthermore TPTP contains many problems involving equality, which Satchmo cannot handle efficiently.

### 5.1 Settings

We have used TPTP release 1.2.0 without additional instances of the parametric problems. Clauses have not been rearranged. Literals have been rearranged only to achieve implication form for Satchmo, MGTP/G and SETHEO. For Satchmo and MGTP/G domain enumeration clauses and literals have been added as mentioned in Section 2.3. For Otter, equality axioms have been omitted.

We have used the following software and hardware: MGTP/G (Prolog version) with "strategy III" [14] (which we found to be most efficient); Otter 3.0.4 in autonomous mode; SETHEO V3.2 with options `-cons -dr`; SICStus Prolog v3, compiling to WAM code; HP 9000/710-50 workstations with 32MB RAM.

Runs have been stopped as soon as the first model or refutation was found, when the system ran out of memory, or after a time limit of 600 seconds. Times given for compiling variants of Satchmo comprise compilation and run times.

### 5.2 Comparison of Satchmo Variants

In the graphical comparisons of two Satchmo variants (Fig. 8 to 10) each plotted point corresponds to one TPTP problem with the two coordinates representing the respective times needed by the two variants. A value of 600 seconds means that either the problem could not be solved by the respective variant within the

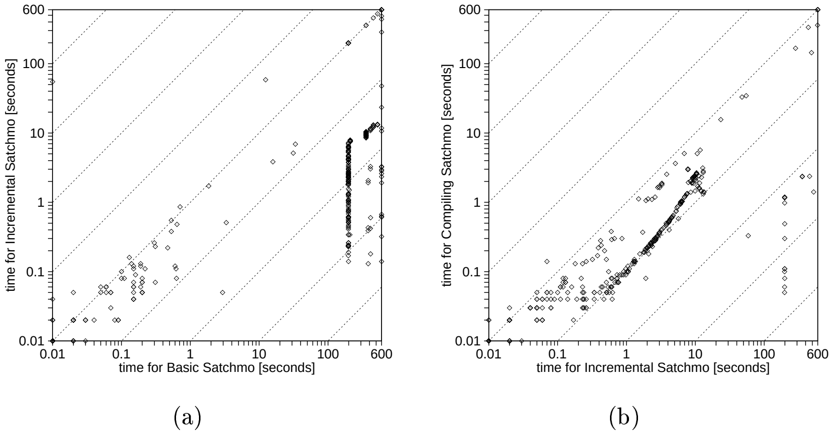


**Figure 8.** (a) Basic vs. Incremental and (b) Incremental vs. Compiling Satchmo.

time limit or a memory overflow occurred. The diagonal lines mark time ratios equal to a power of ten. Note that due to the logarithmic scales small errors in measurement and the clock granularity become visible for short measured times.

**Incremental Evaluation:** As expected, incremental evaluation is a powerful optimization in general (Figure8(a)[10]). There is, however, no simple correlation between the times needed by Basic and Incremental Satchmo, because the two programs happen to generate different tableaux. While Basic Satchmo prefers PUHR steps with nuclei from the beginning of the input clause list, Incremental Satchmo prefers PUHR steps with recently derived electrons. The long vertical cluster in Figure8(a), which appears—as a diagonal line—also in Figure8(b), results from nearly 200 very similar problems (SYN103-1 to SYN301-1).

**Compilation:** Compilation yields, compared to Incremental Satchmo, a speed-up of up to 10 for the majority of problems, but there are also problems with more dramatic speedups (Figure8(b)).

**Fairness and Layers:** We observed that, as expected, the layered approach to fairness is more efficient than the non-layered one (no evidence given here). However, the former still turned out to be less efficient than a depth-first strategy for most non-nesting problems (Figure9(a)).

Applied to nesting problems, selective assignment to layers leads for most problems to significant speedups compared to the plain layered approach and allows to solve many problems within the time limit (Figure9(b)). We conjecture that in problems converted to range-restricted form using domain enumeration, selective assignment to layers prevents from applying the domain enumeration clauses too early and can thus control domain enumeration to some degree.

**Complement Splitting:** Complement splitting is relevant only for non-Horn

[10] Basic and Incremental Satchmo have depth-first search strategies. Therefore Figure8(a) and 8b give times only for problems without nesting clauses.

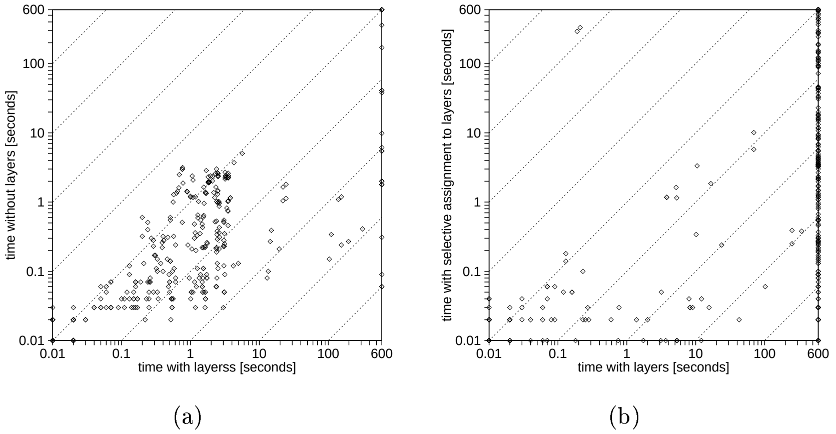


**Figure 9.** Effect of (a) layers and (b) selective assignment to layers (Compiling Satchmo with complement splitting).

problems. While it requires minor overhead, the possibility of closing branches earlier enables Satchmo to solve many unsatisfiable problems faster (Figure10).

When we search for the first model of a satisfiable problem, complement splitting might lead to longer search times, because the leftmost saturated branch(es) of a PUHR tableau might be closed in the corresponding PUHR-CS tableau (cf. Figure1). For the few satisfiable TPTP problems, however, this was not observed.

### 5.3 Comparison with MGTP/G, Otter and SETHEO

We have compared the fastest variant of Satchmo (i.e., Compiling Satchmo with complement splitting and selective assignment to layers) with MGTP/G, a model-generation theorem prover, as well as with Otter and SETHEO, which are refutation-oriented theorem provers.

In a comparison for the 2729 problems of the TPTP Problem Library (Figure11) Satchmo beats MGTP/G, but cannot compete with Otter and SETHEO, which is probably in part due to TPTP's bias towards refutation-oriented theorem provers mentioned above.[11] Each curve gives for one of the theorem provers the number of problems solvable in some time limit per problem, by the time limit.

In a similar comparison for the 353 range-restricted or non-nesting TPTP problems (Figure12), the model generators Satchmo and MGTP/G are able to handle nearly all problems[12]. Satchmo is more than an order of magnitude faster

[11] The fact that Satchmo cannot compete with genuine theorem provers on typical (unsatisfiable) problems for theorem provers has ben reconfirmed at the CADE-13 ATP System competition [24].

[12] In the TPTP Problem Library the range-restricted problems are nearly a subset of the non-nesting problems.

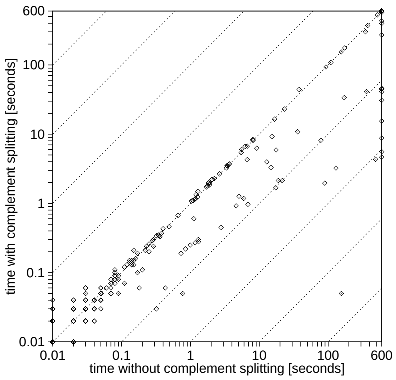


**Figure 10.** Effect of complement splitting for non-Horn problems (Compiling Satchmo with selective assignment to layers).

than MGTP/G is for many problems. One reason for this is certainly that the developers of MGTP have put an emphasis on parallel processing whereas we have used a sequential version of MGTP.

## 6 Related Work

Using logic programming technology in theorem proving has become usual (most prominently by Stickel [22]). As opposed to some other approaches we do not modify a Prolog system or even reimplement parts of it. The programs we describe in this paper run with standard Prolog systems. Posegga's [18] and Wunderwald's [26] approaches, on which we comment in some more detail below, are other examples for compiling deduction systems that work on top of Prolog.

The problem of redundant computations that the original Basic Satchmo suffers from, has also shown up in other rule-processing AI systems and in deductive databases. Solutions like the Rete algorithm [9] and "semi-naïve" fixpoint iteration [3, 1] have been developed. For the Satchmo-like theorem prover MGTP solutions to the problem have also been developed [10, 14], which are, however, more complex than ours and less suited for compilation.

Many theorem provers perform some preprocessing on their input formulas before the central inference mechanism is started. However, to our knowledge there is no preceding work on similarly far-reaching compilation for model generators.

In the rest of this section we briefly investigate the following questions: How are our techniques related to compilation techniques in automated theorem proving and logic programming? Can they be transferred to other approaches for model generation or to other variants of Satchmo?

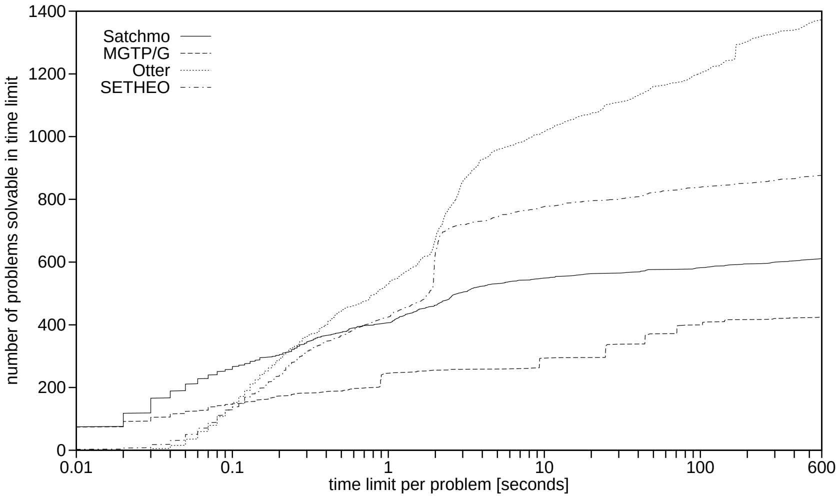


**Figure 11.** Comparison of theorem provers for all TPTP problems.

### 6.1 Automated Theorem Proving and Logic Programming

We cannot cover here all compilation approaches in this area but discuss some which are more closely related to our work.

**Partial Evaluation:** Some of our compilation steps can also be performed by a partial evaluator. We have experimented with "Mixtus" [19], a powerful partial evaluator for SICStus Prolog. Due to the generality of this partial evaluator the compilation times have been orders of magnitude longer than compilation times with our specialized compiler.

**Hyperresolution:** Our compilation covers both the hyperresolution step and the splitting step of the PUHR-tableau calculus. For calculi based on hyperresolution, it is usual that all possible nuclei are known before the deduction process begins. Therefore it is a well-known fact that the process of finding electrons for a given nucleus can be compiled. Since in our case all the electrons of a hyperresolution step are unit clauses, the resolvent is an instance of the head of the nucleus. Therefore the structure of the resolvent is known to a large extent at compile time and the splitting step, which is applied to the resolvent, can be compiled easily as well. In a hyperresolution calculus without restriction to unit electrons the structure of generated resolvents is not known at compile time and so their generation and further use cannot be compiled to the same extent as we do.

Compiling Horn clauses for forward-chaining is well understood in the area of deductive databases. Their implementation is usually set-oriented, i.e., all possible hyperresolution steps with some given nucleus are performed in one step. The nuclei are frequently compiled into relational-algebra expressions, which are then evaluated using relational database techniques. The ideas described in this

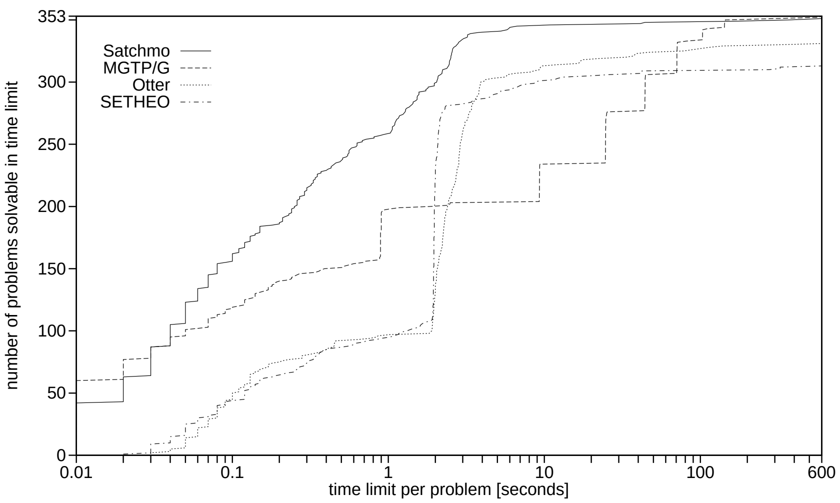


**Figure 12.** Comparison of theorem provers for range-restricted or non-nesting TPTP problems.

paper are heavily influenced by techniques for the evaluation of Horn clause programs with tuple granularity [20].

Wunderwald [26] describes an evaluation technique (including compilation) for *Horn* clauses in Prolog, which is based on hyperresolution like Satchmo. Because of the restriction to Horn clauses, splitting is not needed, which also allows for some simpler solutions on the technical level: Since backtracking is not needed to iterate over tableau branches, it can be used to iterate over derived atoms, instead. Failure-driven loops are therefore used instead of our recursion (in `fire_list`) over lists accumulated by `findall`.

We also want to note that our compilation approach can be seen as a counterpart to Magic Sets in the sense that a forward-chaining evaluation of a Magic-Sets-transformed program simulates a backward-chaining evaluation of the original program, whereas a backward-chaining evaluation of our compiler output simulates a forward-chaining evaluation of the original program.

**Free Variable Tableaux:** Posegga [18] has described a compilation technique for theorem proving based on free-variable tableaux. Although the underlying tableau calculus is quite different to PUHR tableaux, there are some interesting similarities between Posegga's approach and ours. First, lean$T^4\!P$ [4], the interpretive approach corresponding to Posegga's compiler, is based on the same "lean deduction" philosophy as Satchmo. That is, both Satchmo and lean$T^4\!P$ are very clear small and efficient programs. It is this clarity of the interpreter which makes compilation tractable.

An interesting technical similarity between Posegga's approach and ours is the introduction of names for (sub)formulae to avoid passing them around at runtime. The speedup from compilation by a factor of 10, which Posegga has found, is comparable to our results.

### 6.2 Other Approaches to Model Generation

The model generation procedure for clausal theories described by Fermüller and Leitsch [8] is—like ours—based on hyperresolution, subsumption, and some sort of splitting, but in a different way: Hyperresolution (also with non-unit electrons) and subsumption are performed until saturation first. Then, while there is a non-unit resolvent, one such resolvent is replaced by one of its atoms[13] and again hyperresolution and subsumption are performed until saturation. The procedure terminates if all remaining resolvents are unit clauses.

According to our previous considerations, it would be straight-forward to compile the hyperresolution part of the procedure. But since the structure of the generated resolvents is not known at compile time, neither subsumption nor splitting can be compiled in a similar way as we did it for PUHR tableaux.

The model generation techniques by Caferra and Zabel [6] and by Tammet [25] are based on non-linear binary resolution. For these we do not see how to compile input clauses in a way as simple as it can be done for Satchmo.

There is also a body of work concerned with generation of finite models of a fixed size, e.g. [21]. The techniques used by these systems are fundamentally different to the techniques used for the generation of Herbrand models. They are typically techniques for propositional rather than first-order logic.

### 6.3 Variants of Satchmo

**Relaxing Range Restriction:** The versions of Satchmo described in this paper can only deal with range-restricted clauses (Section 2.3), since in this case all resolvents are ground and splitting is correct. Splitting is, however, also correct if the resolvent does contain variables, but all variables occur in only one literal. We may avoid ground instantiation of these variables.

Baumgartner, Furbach, and Niemelä [2] suggest to check for variables occurring in more than one literal after every hyperresolution step and to instantiate them by ground terms. From our compilation point of view this is a runtime test which probably cannot be compiled but can be included as part of the "runtime system".

There are syntactic conditions for clause sets that ensure that no resolvent contains variables occurring in more than one literal [8]. For such clause sets the aforementioned runtime checks are not needed, which simplifies compilation a lot.

Furthermore, without range restriction the "one-way" matching between nuclei and electrons has to be replaced by a more general unification method. The compilation of this remains especially simple with the target language Prolog.

[13] This can be considered as a "committed" splitting step, because it is not necessary to try all the atoms.

**Generation of Minimal Models:** Bry and Yahya [5] describe an extension to PUHR-CS tableaux to entirely avoid the generation of non-minimal Herbrand models. This extension works by generation of new negative clauses during the deduction process. Since these clauses are not known at compile time, integration of this extension into compiling Satchmo is not straight-forward.

## 7 Conclusion

We have presented Incremental Satchmo, a still simple yet significantly more efficient variant of the model generator Satchmo, and developed from it a technique for compiling clausal first order theories into Prolog programs. Furthermore we have developed an efficient technique for achieving fairness. The advantages of compilation are a speedup for the individual deduction steps and the possibility to move some decisions (e.g., whether to delay a resolvent) and transformations (e.g., prefering clauses with empty head) to compile time thus not sacrificing runtime performance.

We have evaluated our techniques by benchmarks based on the TPTP Problem Library. It has turned out that for range-restricted problems and for non-nesting problems Compiling Satchmo is more efficient than the other theorem provers used in the comparison, even ones implemented in C and using more sophisticated data structures for indexing. For many other problems model generation either is not appropriate or needs to be extended.

We have compared our techniques with other compiling approaches for deduction and with other model generation approaches. These comparisons have indicated several possible directions for further developments, but other directions are possible as well:

From a theorem proving and model generation perspective, an extension to be investigated and implemented is reasoning with equality [7, 13]. In addition, bidirectional search as in Satchmore [15] can accelerate refutation 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.

From an efficiency perspective, better data structures such as term indexes [12] have to be used. Further optimizations (e.g., techniques making use of heuristics, symmetries, or functional dependencies) should be investigated. We currently consider compilation to a language different from Prolog, e.g., to an abstract machine language tailored to the specific needs of model generation.

### Acknowledgments

We would like to thank François Bry, Norbert Eisinger, Sven Panne, and Adnan Yahya for helpful comments and discussions. The support for the first author by the Bayerischer Habilitations-Förderpreis is appreciated.

## References


1. F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD 1986*, pages 16–52. ACM, 1986.
2. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper tableaux. In *Logics in Artificial Intelligence: European Workshop, JELIA '96*, Springer LNAI 1126, 1996.
3. R. Bayer. Query evaluation and recursion in deductive database systems. Technical Report TUM-I8503, Technische Universität München, 1985.
4. B. Beckert and J. Posegga. lean$T^AP$: lean, tableau-based theorem proving. In A. Bundy, editor, *13th Int. Conf. on Automated Deduction (CADE)*, Springer LNAI 814, Nancy, France, June/July 1994.
5. F. Bry and A. Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In *5th Workshop on Theorem Proving with Tableaux and Related Methods*, Springer LNAI, 1996.
6. R. Caferra and N. Zabel. Extending resolution for model construction. In *Logics in AI: European Workshop JELIA '90*, Springer LNAI 478, pages 153–169, 1991.
7. M. Denecker and D. De Schreye. On the duality of abduction and model generation. In I. Staff, editor, *Proceedings, International Conference on Fifth Generation Computer Systems '92*, pages 650–657. IOS Press, 1992.
8. C. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2):173–203, 1996.
9. C. Forgy. Rete: A fast algorithm for the many patterns/many objects pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
10. H. Fujita and R. 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.
11. C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent devleopments. In *Automated Deduction — CADE-12*, Springer LNAI 814, pages 778–782, 1994.
12. P. Graf. *Term Indexing*. PhD thesis, Universität des Saarlandes, 1995. Published by Springer-Verlag, LNAI 1053.
13. L. Herr. E-SATCHMO: introduction de l'égalité dans le démonstrateur automatique SATCHMO. Rapport de stage de D.E.A., Ecole Normale Supérieure, 1993. In French.
14. Institute for New Generation Computer Technology. *Model Generation Theorem Prover: MGTP*, 1995. http://www.icot.or.jp/ICOT/IFS/IFS-abst/082.html.
15. D. W. Loveland, D. W. Reed, and D. S. Wilson. SATCHMORE: SATCHMO with RElevancy. *Journal of Automated Reasoning*, 14:325–351, 1995.
16. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *9th Int. Conf. on Automated Deduction (CADE)*, Springer LNCS 310, pages 415–434, 1988.
17. W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL 94/6, Argonne National Laboratory, 1994.
18. J. Posegga. Compiling proof search in semantic tableaux. In *Proc. 7th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, Springer LNCS 671, pages 67–77, 1993.
19. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. SICS Dissertation Series 04, The Royal Institute of Technology (KTH), 1991.
20. H. Schütz. *Tuple-oriented Bottom-up Evaluation of Logic Programs*. PhD thesis, Technische Universität München, 1993. in German.

21. J. Slaney. FINDER (Finite Domain Enumerator): Notes and guide. Technical Report TR-ARP-1/92, Australian National University Automated Reasoning Project, Canberra, 1992.
22. M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.
23. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In *Automated Deduction — CADE-12*, Springer LNAI 814, pages 252–266, 1994.
24. C. B. Suttner and G. Sutcliffe. The design of the cade-13 atp system competition. In M. McRobbie and J. Slaney, editors, *13th Int. Conf. on Automated Deduction (CADE)*, Springer LNAI 1104, pages 146–160, New Brunswick, NJ, USA, 30 July–3 August 1996.
25. T. Tammet. Using resolution for deciding solvable classes and building finite models. In *Baltic Computer Science: selected papers*, Springer LNCS 502, pages 33–64, 1991.
26. J. E. Wunderwald. Memoing evaluation by source-to-source transformation. In *Fifth Int. Workshop on Logic Program Synthesis and Transformation*, 1995.