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

Ludwig
Maximilians
Universität
München
LMU

Efficient Model Generation through Compilation

Heribert Schütz, Tim Geisler

appeared in *Proc. 13th Int. Conf. on Automated Deduction (CADE)*,
Springer LNCS, 1996
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1996-2, Januar 1996

To appear in the Proc. of CADE-13, 1996

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 model-generation theorem provers 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 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 significantly more efficient strategy.
We have compared various combinations of 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".

1 Introduction

Refutation-oriented theorem provers search for proofs for the unsatisfiability of a theory, whereas model-generation theorem provers search for proofs for the satisfiability of a theory, i.e., for models. Application areas for model-generation theorem proving include certain planning and configuration problems, finding counterexamples for conjectures and testing integrity constraints in databases. Model generation can also be used to test the unsatisfiability of a theory if there is no need for a more direct proof.

The work presented in this paper is based on the model-generation theorem prover Satchmo [8]. We describe techniques to enhance the efficiency of model generation, most importantly compilation into a procedural Prolog program and avoidance of redundant computations.

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-generation theorem provers. Compiling Horn clauses for forward-chaining is well understood in the area of deductive databases. The ideas described in this paper are heavily influenced by techniques for the evaluation of Horn clause programs with tuple granularity [11].

Using logic programming technology in theorem proving has become usual (most prominently by Stickel [12]). As opposed to 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.

The problem of redundant computations that the original implementation of Satchmo suffers from, has also shown up in other rule-processing AI systems and in deductive databases. Solutions like the Rete algorithm [4] and "seminaïve" fixpoint iteration [2, 1] have been developed. For the Satchmo-like theorem prover MGTP solutions to the problem have also been developed [5, 7], which are, however, more complex than ours and less suited for compilation.

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" (Section 4.1).

2 Preliminaries

2.1 PUHR Tableaux

Satchmo has been formalized with positive unit hyperresolution (PUHR) tableaux for clausal theories [3]. For the rest of this paper let $\mathcal{S}$ be a set of 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. We write empty bodies and heads as $\top$ and $\bot$, respectively.

PUHR tableaux are trees with nodes labeled by ground atoms (except for the root). Starting from the tree consisting of a single unlabeled node, PUHR tableaux for the clause set $\mathcal{S}$ are generated by repeated application of a single expansion rule: If $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$ is a ground instance of a clause in $\mathcal{S}$ and there is some branch containing the atoms $A_1, \ldots, A_n$ but none of the atoms $B_1, \ldots, B_m$, then append m children with labels $B_1, \ldots, B_m$ to the final node of this branch.

A branch of a PUHR tableau for $\mathcal{S}$ is *closed* if it contains atoms $A_1, \ldots, A_n$ such that some clause in $\mathcal{S}$ has a ground instance $A_1 \wedge \ldots \wedge A_n \rightarrow \bot$. A PUHR tableau is *closed* if all its branches are. A branch or a PUHR tableau is *open* if it is not closed. 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 [3]:

1. A saturated open branch of a PUHR tableau for $\mathcal{S}$ is a Herbrand model[1]

[1] As usual, we identify a tableau branch with the set of node labels along the branch and a Herbrand interpretation with the set of ground atoms it satisfies.

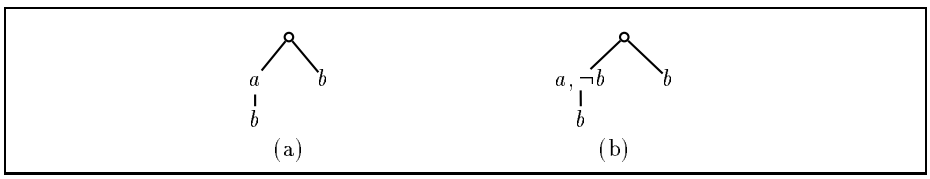


Fig. 1. PUHR tableaux (a) without and (b) with complement splitting.

of $\mathcal{S}$. As a consequence, if $\mathcal{S}$ is unsatisfiable, then every saturated PUHR tableau for $\mathcal{S}$ is closed.

2. A model of $\mathcal{S}$ satisfies some open branch of every PUHR tableau for $\mathcal{S}$. As a consequence, if $\mathcal{S}$ is satisfiable, then there is no closed PUHR tableau for $\mathcal{S}$.
3. Every minimal Herbrand model of $\mathcal{S}$ occurs as an open branch in every saturated PUHR tableau for $\mathcal{S}$.

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 Fig. 1(a) has a non-minimal Herbrand model as its saturated open left branch.

We can avoid this and many similar cases by allowing tableau nodes to be labeled by several negative ground literals in addition to the positive ground literal. The tableau expansion rule is modified in such a way that a newly appended 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 [8, 3]. A branch is now considered to be closed also if it contains 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 Fig. 1(b) is closed because it contains b and $\neg b$.

Properties 1 to 3 of the PUHR-tableaux proof procedure given in the previous section also hold for PUHR-CS tableaux if we ignore the negative node labels. In addition, 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. 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 [8, 3]. However,

```
saturate :- expandable_with(Head) -> expand(Head) ; true.

expandable_with(Head) :- (Body ---> Head), Body, \+ Head.

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

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

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

Fig. 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.
```

Fig. 3. An example set of clauses.

this approach is generally inefficient in the presence of several function symbols with arity > 0, but we are able to handle clause sets without such functions and also some other clause sets, especially ones with a single unary function.

2.4 Basic Satchmo

A simple Prolog implementation of PUHR tableaux for range-restricted clauses is given in Fig. 2. We call this program *Basic Satchmo*. It is basically the original implementation of Satchmo [8], 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`,` $\ldots$`,` A_n `--->` B_1`;` $\ldots$`;` B_m`.`", where "`--->`" is an infix operator with lower precedence than "`,`" and "`;`". $\top$ and $\bot$ are represented by `true` and `false`. See Fig. 3 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. The node labels for the current branch are stored in Prolog's dynamic database, which should initially be empty.

When the top-level procedure[2] `saturate` is called, `expandable_with` looks for a clause instance whose body atoms do occur in the current branch (i.e.,

[2] 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.

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

fire_list([]).
fire_list([Head | Heads]) :- fire(Head), fire_list(Heads).

fire(Head) :- Head -> true ; expand(Head).

update(Atom) :-  assume(Atom),
                 findall(Head, delta(Atom, Head), Heads),
                 fire_list(Heads).

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).
```

Fig. 4. Incremental Satchmo.

the dynamic database) and whose head atoms do not (making use of the Prolog built-ins `true`, `false`, ",", and ";").[3] If no such clause instance exists, a model has been found and `saturate` returns successfully. Otherwise `expand` is called to expand the current branch. With a clause head `false` the branch is closed. With a disjunctive head the branch is expanded by generating branches for all disjuncts. A new leaf node 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 `expandable_with` repeats in later expansion steps work done in earlier steps, because typically many clauses are examined whose body is not affected by atoms added to the dynamic database recently. We can avoid this repeated work if we specifically look for clause body instances that have just become true due to a new atom, and collect all the corresponding head instances. This is done by the program *Incremental Satchmo* (Fig. 4; together with unchanged procedures `assume` and `expand` from Fig. 2).

In the beginning we find all clauses with empty bodies, collect their heads, and "fire" them one after the other. "Firing" a head (instance) means to test

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

whether it already holds, i.e., if one of its atoms already appears in the current branch, and using it in an expansion step if not. Whenever (in `update`) we add a new atom `Atom` to a branch, we look (in `delta` and `delta_body`) for (the heads of) those clause instances whose bodies contain `Atom` and whose remaining body atoms also appear in the branch. We again fire all found heads rather than only the first one. If the expansion of some head leads to branch splitting, later heads in the accumulated list will have to be processed in all subbranches. On the other hand, if some head is empty (`false`) and a branch is closed, later heads will be ignored. The test of whether some head atom is already in the branch has been moved to a place immediately before the call of `expand` in the procedure `fire`. This is necessary because such an atom might have been added to the branch after the head has been determined by `delta`.

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

Wunderwald [14] describes a similar bottom-up evaluation technique for *Horn* clauses in Prolog. Since in his approach choicepoints are not needed to separate tableau branches, they can be used to separate atoms and clause instances, instead. Failure-driven loops are used instead of recursion over lists accumulated by `findall`.

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 interpreted/compiled. 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, `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 `expand` in `fire`, and the calls to `--->` and `delta_body` in `delta`. With our example clause set (Fig. 3) 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(Head, delta(edge(Arg1, Arg2), Head), Heads),
        fire_list(Heads).

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 clause heads in `saturate`, `delta`, and `fire`. These identifiers consist of a clause identifier (given as comment in Fig. 3) and a tuple of the variables occurring in the clause 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 identifiers for clause heads, 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(Head, delta_edge(Arg1, Arg2, Head), Heads),
        fire_list(Heads).

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.

Some of these program transformation steps can be performed by a partial evaluator. We have experimented with "Mixtus" [10], 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.

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 (Fig. 4).

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 clause instance to use. 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.[4] 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.[5] We call such problems *non-nesting*.

We will first transform the central group of mutually recursive procedures of Incremental Satchmo (`fire_list`, `fire`, `expand`, 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 `Heads`, we pass this list on to the procedures `fire`, `expand`, and `update`. In `update` we call `fire_list` anyway for a newly generated list of heads. Here we can simply append the two lists and call `fire_list` only once. We also have to call `fire_list` in procedure `fire` if `Head` holds. The modified procedures are given in Fig. 5.

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

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

```
fire_list([]).
fire_list([Head | Heads]) :- fire(Head, Heads).

fire(Head, Heads) :- Head -> fire_list(Heads) ; expand(Head, Heads).

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

update(Atom, OldHeads) :- assume(Atom),
                          findall(Head, delta(Atom, Head), NewHeads),
                          append(NewHeads, OldHeads, Heads),
                          fire_list(Heads).
```

Fig. 5. Tail recursion for Incremental Satchmo.

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

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

fire(Head, CL, NL) :- Head -> fire_list(CL, NL) ; expand(Head, CL, NL).

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

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

Fig. 6. Layers for Incremental Satchmo.

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

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

Layers: To overcome this problem we split the list of heads into a "current layer" and a "next layer" (Fig. 6). We can now efficiently *pre*pend the newly generated heads to the next layer (`NL`) without sacrificing fairness, since heads in the current layer (`CL`) are expanded 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(Head) | CL], NL) :- !, fire_list(CL, [Head | NL]).
fire_list([Head | CL], NL)        :-    fire(Head, CL, NL).

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

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

Fig. 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 Fig. 9(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 heads into the next layer, we do so only for nesting clauses[6] and insert other heads into the current layer. For technical reasons, we actually first insert all heads into the current layer, but some of them tagged by a functor `delay`. Later the newly added third rule for `fire_list` (Fig. 7) moves these heads 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.[7] First, we parse disjunctions in clause heads as left-associative operators. Then in a disjunction $X \vee Y$ the right alternative Y will always be an atom. Before expanding X we add $\neg Y$ to the corresponding tableau branch(es) by asserting `neg(`Y`)`. The second rule of the procedure `expand` becomes:

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

Note that the recursive call `expand(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), ...
```

[6] We could be even more restrictive here (cf. Footnote 5).

[7] 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 head containing A is expanded, 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 Fig. 3 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 [7], Otter [9], and SETHEO [6]). 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 [13], 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

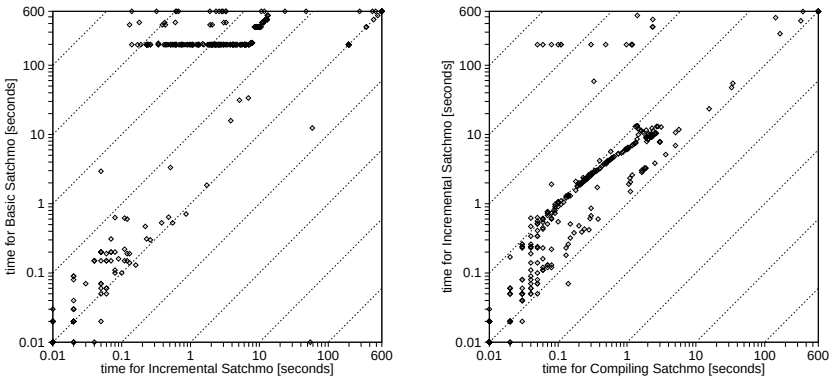
We have used TPTP release 1.2.0 without additional instances of the parametric problems. Clauses have not been rearranged. Literals have been rearranged only to achieve implication form for Satchmo, MGTP/G and SETHEO. For Satchmo and MGTP/G domain enumeration clauses and literals have been added as mentioned 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" [7] (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



(a) (b)

Fig. 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.

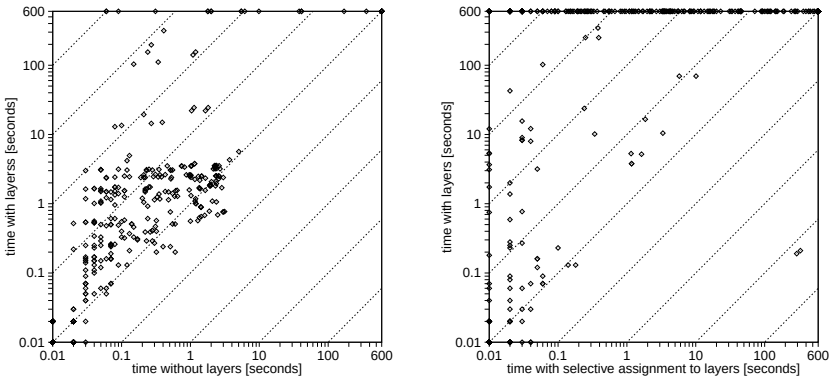
Incremental Evaluation: As expected, incremental evaluation is a powerful optimization in general (Fig. 8(a)[8]). There is, however, no simple correlation between the times needed by Basic and Incremental Satchmo, because the two programs happen to generate different tableaux. While Basic Satchmo prefers to expand instances of clauses from the beginning of the enumeration of the clauses as Prolog facts, Incremental Satchmo prefers clause instances with recently asserted atoms in the body. The long horizontal cluster in Fig. 8(a), which appears—as a diagonal line—also in Fig. 8(b), results from nearly 200 very similar problems (SYN103-1 to SYN301-1).

Compilation: Compilation yields, compared to Incremental Satchmo, a speedup of up to 10 for the majority of problems, but there are also problems with more dramatic speedups (Fig. 8(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 (Fig. 9(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 (Fig. 9(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.

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



(a) (b)

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

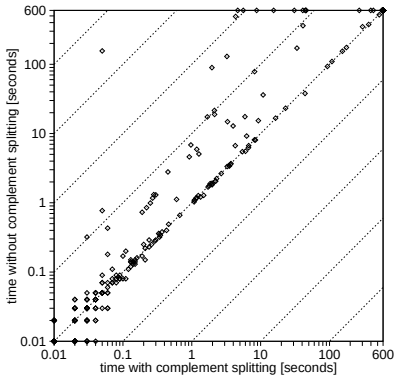
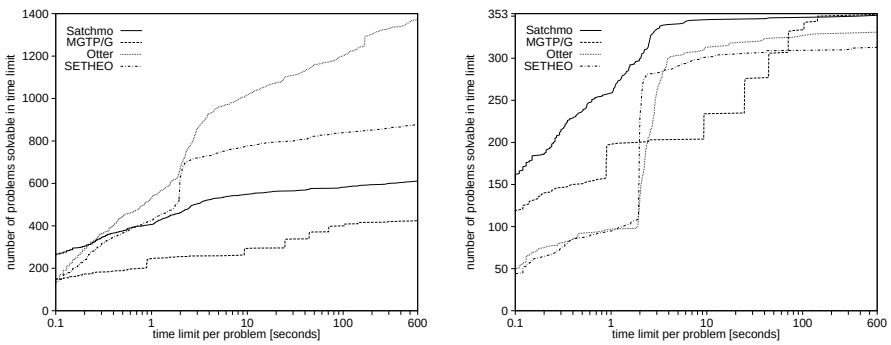


Fig. 10. Effect of complement splitting for non-Horn problems (Compiling Satchmo with selective assignment to layers).

Complement Splitting: Complement splitting is relevant only for non-Horn problems. While it requires minor overhead, the possibility of closing branches earlier enables Satchmo to solve many unsatisfiable problems faster (Fig. 10).

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. Fig. 1). For the few satisfiable TPTP problems, however, this was not observed.



(a) (b)

Fig. 11. Comparison of theorem provers (a) for all TPTP problems and (b) for range-restricted or non-nesting TPTP problems.

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 (Fig. 11(a)) 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. 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 (Fig. 11(b)), the model generators Satchmo and MGTP/G are able to handle nearly all problems[9]. Satchmo is more than an order of magnitude faster than MGTP/G is for many problems.

6 Conclusion

We have presented a simple yet efficient model-generation theorem prover, Incremental 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 clause head) and transformations (e.g., prefering clauses with empty head) to compile time thus not sacrificing runtime performance.

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

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.

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. R. Bayer. Query evaluation and recursion in deductive database systems. Technical Report TUM-I8503, Technische Universität München, 1985.
3. 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.
4. C. Forgy. Rete: A fast algorithm for the many patterns/many objects pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
5. 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.
6. 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.
7. Institute for New Generation Computer Technology. *Model Generation Theorem Prover: MGTP*, 1995. http://www.icot.or.jp/ICOT/IFS/IFS-abst/082.html.
8. 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.
9. W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL 94/6, Argonne National Laboratory, 1994.
10. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* SICS Dissertation Series 04, The Royal Institute of Technology (KTH), 1991.
11. H. Schütz. *Tuple-oriented Bottom-up Evaluation of Logic Programs.* PhD thesis, Technische Universität München, 1993. in German.
12. M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.
13. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In *Automated Deduction — CADE-12*, Springer LNAI 814, pages 252–266, 1994.
14. J. E. Wunderwald. Memoing evaluation by source-to-source transformation. In *Fifth Int. Workshop on Logic Program Synthesis and Transformation*, 1995.

This article was processed using the LaTeX macro package with LLNCS style