TR-0751

Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers

by

R. Hasegawa, M. Koshimura & H. Fajita

March, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome

(03)3456-3191~5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers

Ryuzo HASEGAWA
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
phone: +81-3-3456-2514 fax: +81-3-3456-1618
internet: hasegawa@icot.or.jp

Miyuki KOSHIMURA
Toshiba Information Systems (Japan)

Hiroshi FUJITA
Mitsubishi Electric Corporation

Abstract

This paper presents several algorithms for implementing efficient theorem provers based on lazy model generation. The tasks of the model generation based prover are the generation and testing of atoms which are to be the elements of a model for the given theorem. A problem with this method is explosion in the number of generated atoms and in the computational cost in time and space consumed by the generation processes. The lazy model generation method avoids generating unnecessary atoms that are irrelevant to obtaining proofs. Indeed, no matter what calculus a prover may take, any theorem which inherently has a large search space cannot be solved with the limited resources available under normal circumstances, unless the generation and retention of information is carefully managed. The lazy model generation method provides flexible control for efficient use of resources in a simple manner, thereby decreasing the time and space complexity of some algorithms by several orders of magnitude.

1 Introduction

The aim of this research is to make high performance theorem provers for first-order logic by using the programming techniques of the parallel logic programming language, KL1.

There are theorem provers using Prolog technology, including PTTP by Stickel [Sti88], SETHEO by Schumann [Sch89], and SATCHMO by Manthey and Bry [MB88]. PTTP and SETHEO are backward-reasoning provers, based on the model elimination method. SATCHMO, on the other hand, is a forward-reasoning prover, based on the model generation method.

As a first step toward developing KL1-technology theorem provers, we adopted the model generation method on which SATCHMO is based. Our reasons were as follows: 1) A useful feature of SATCHMO is that full unification is not necessary, and that matching suffices when dealing with range-restricted problems. This makes it very convenient for us to implement theorem provers in KL1 since KL1, as a committed choice language, provides us with very fast one-way unification. 2) It is easier to incorporate a mechanism for lemmatization, subsumption tests, and other deletion strategies that are indispensable to solving difficult problems such as condensed detachment problems [Wos88][Ove90][MW91].

In implementing model generation based provers, it is important to avoid redundancy in the *conjunctive matching* (defined later) of clauses against atoms in model candidates. For this, we proposed the RAMS [FH91] and MERC [Has91] methods.

A more important issue facing the efficiency of model generation based provers is how to reduce the total computation amount and memory space required for proof processes. This problem becomes more critical when dealing with harder problems that require deeper inferences (longer proofs) such as Lukasiewicz problems[Ove90]. To solve such problems, it is important to recognize that proving processes are viewed as *generation-and-test* processes and that generation should be performed only when required by the testing.

In the case of SATCHMO, model extension (generation) and model rejection (test) are completely synchronized by using assert/retract and sequential control with backtracking of the Prolog system. Hence, it is free from the explosion of assertions caused by the generation process. However, this control makes it impossible to incorporate certain strategies such as weighting heuristics which require items to be generated in aggregates, then sorted in accordance with the weights calculated for each item.

On the other hand, orthodox provers such as OTTER [McC90] are designed to be very general and flexible so as to incorporate many strategies. However, they may suffer from an explosion of generated resolvents, for there is usually no serious distinction between generation and testing in their implementation. Without the generate-and-test way of thinking, a naive implementation would produce redundant computation in the proving processes. Even worse, the computation would cause an explosion of memory space in a parallel environment.

In this paper, we propose a new method called *Lazy Model Generation* in which the idea of demand-driven computation or 'generate-only-at-test' is implemented, and present some techniques for improving the performance of forward reasoning theorem provers.

2 Model Generation Method

Throughout this paper, a clause is represented in an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

where $A_i (1 \leq i \leq n)$ and $C_j (1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, A_2, \ldots, A_n$; and the consequent is a disjunction of $C_1, C_2, \ldots, C_m$. A clause is said to be *positive* if its antecedent is $true (n = 0)$.

and *negative* if its consequent is *false* ($m = 0$), otherwise it is *mixed* ($n \neq 0, m \neq 0$). Positive clauses and mixed clauses are called *generator clauses*. Negative clauses are called *tester clauses*.

The model generation method incorporates the following two rules:

- Model extension rule: If there is a generator clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model candidate M but $C\sigma$ is not satisfied in M, then extend M by adding $C\sigma$ to M.

- Model rejection rule: If a tester clause has an antecedent $A\sigma$ that is satisfied in a model candidate M, then reject M.

We call the process of obtaining $A\sigma$, a *conjunctive matching (CJM)* of the antecedent literals against the elements in a model candidate. Note that the antecedent (*true*) of a positive clause is satisfied by any model.

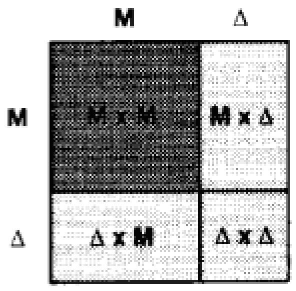
The task of model generation is to try to construct a model for a given set of clauses, starting with a null set as a model candidate. If the clause set is satisfiable, a model should be found for the finite cases. The method can also be used to prove that the clause set is unsatisfiable. This is done by exploring every possible model candidate to check that no model exists for the clause set.

2.1 Avoiding Redundancy in Conjunctive Matching

To improve the performance of the model generation provers, it is essential to avoid, as much as possible, redundant computation in conjunctive matching.

Let us consider a clause, C, having two antecedent literals. To perform conjunctive matching for the clause, we need to pick pairs of atoms out of the current model candidate, M. Imagine also that, as a result of a satisfiability check of the clause, we are to extend the model candidate with Δ, an atom in the consequent of the clause, C, but not in M. Then, in the conjunctive matching for the clause, C, in the next phase, we need to pick pairs of atoms from $M \cup \Delta$. The number of pairs amounts to:

$$(M \cup \Delta)^2 = M \times M \cup M \times \Delta \cup \Delta \times M \cup \Delta \times \Delta.$$



It should be noted here that $M \times M$ pairs were already considered in the previous phase of conjunctive matching. If they were chosen in this phase, the result would contribute nothing since the model candidate need not be extended with the same Δ. Hence, redundant consideration on $M \times M$ pairs should be avoided at this stage. Instead, we have to choose only the pairs which contain at least one Δ. This discussion can be generalized for cases in which we have more than two antecedent literals, any number of clauses, and any number of model candidates.

The approach taken in OTTER to avoid the above sort of redundancy is as follows. For a clause,

$$A_1, A_2, \ldots, A_n \rightarrow C_1 ; C_2 ; \ldots ; C_m ,$$

we first match some A_i against a model-extending atom, Δ. Then, for the rest of the literals,

$$\{A_1, A_2, \ldots, A_n\} - \{A_i\},$$

we match each against an atom picked out of $M \cup \Delta$, where M is the current model candidate.

Note that the above method involves duplicated computation for a clause with more than two antecedent literals. Since a pair, $< A_i, A_j >$ $(i \neq j)$, is matched against both $< \Delta, M \cup \Delta >$ and $< M \cup \Delta, \Delta >$, the conjunctive matching of $\Delta \times \Delta$ is duplicated. This redundancy can be removed by employing the MERC method [Has91].

In the actual implementation adopted here, which we call the Δ-M method, we prepare a pattern like:

$$\{[\Delta, \Delta], [\Delta, M], [M, \Delta]\}$$

for clauses with two antecedent literals,

$$\{[\Delta, \Delta, \Delta], [\Delta, \Delta, M], [\Delta, M, \Delta], [M, \Delta, \Delta],$$
$$[\Delta, M, M], [M, \Delta, M], [M, M, \Delta]\}$$

for clauses with three antecedent literals, and so forth. According to this pattern, we enumerate all possible combinations of atoms for matching the antecedent literals of given clauses.

The Δ-M method is similar to the MERC method. The MERC method, however, requires the preparation of multiple entry clauses or copies of clauses in place of the above pattern.

The RAMS method [FH91] is another approach where every successful result of matching a literal A_i against model elements is memorized to prevent rematching the same literal against the same model element. Both the Δ-M and the MERC method still contain redundant computation. For instance, in the computation for $[M, \Delta, \Delta]$ and $[M, \Delta, M]$ patterns, the common subpattern $[M, \Delta]$ will be recomputed. The RAMS method can remove this sort of redundancy. It tends, however, to require a lot of memory to store the information of partial matching.

```
$M := \phi$;
$D := \{A \mid (true \rightarrow A) \in$ a set of given clauses$\}$;
while $D \neq \phi$ do begin
    $D := D - \Delta$;
    if $CJM_{Tester}(\Delta, M) \ni false$ then return(success);
    $new := CJM_{Generator}(\Delta, M)$;
    $M := M \cup \Delta$;
    $new' := subsumption(new, M \cup D)$;
    $D := D \cup new'$;
end return(fail)
```

Figure 1: Basic algorithm

3 Algorithms for Model Generation

This section and the next present several algorithms for the model generation method and compares them in terms of time and space complexity. To simplify this presentation, we assume that the problem is given only in Horn clauses. However, the principle behind these algorithms can also be applied to non Horn clauses.

3.1 Basic Algorithm

The basic algorithm shown in Figure 1 performs model generation with a search strategy adopting a breadth-first approach. This is essentially the same algorithm as the hyper-resolution algorithm taken by OTTER [McC90][1].

In the algorithm, M represents the model candidate, D represents the model-extending candidate (a set of model-extending atoms that are generated as a result of the application of the model extension rule and that are going to be added to M), and Δ represents a subset of D. Initially, M is set to an empty set, and D is a set of positive (unit) clauses for the given problem.

In each cycle of the algorithm,

1) Δ is selected from D,

2) a rejection test (conjunctive matching for the tester clauses) is performed on Δ and M,

3) if the test succeeds then the algorithm terminates,

4) if the test fails, model extension (conjunctive matching on the generator clauses) is performed on Δ and M, and

[1]OTTER is a slightly optimized version of the basic algorithm, where negative unit clauses are tested on literals in new as soon as they are generated as the full-test algorithm, described in the next section.

```
$M := \phi$;
$D := \{A \mid (true \rightarrow A) \in$ a set of given clauses$\}$;
**while** $D \neq \phi$ **do begin**
        $D := D - \Delta$;
        $new := CJM_{Generator}(\Delta, M)$;
        $M := M \cup \Delta$;
        $new' := subsumption(new, M \cup D)$;
        **if** $CJM_{Tester}(new', M \cup D) \ni false$ **then return**(success);
        $D := D \cup new'$;
**end return**(fail)
```

Figure 2: Full-test algorithm

5) a subsumption test is performed on *new* against $M \cup D$.

If D is empty at the beginning of a cycle, the algorithm terminates as the refutation fails (in other words, a model is found for the given set of clauses).

The conjunctive matching and subsumption test is represented by the following functions on sets of atoms.

$$
\begin{aligned}
&CJM_{Cs}(\Delta, M) = \\
&\quad \{\sigma C \mid \sigma A_1, \ldots, \sigma A_n \rightarrow \sigma C \\
&\qquad \wedge\ A_1, \ldots, A_n \rightarrow C \in Cs \\
&\qquad \wedge\ \sigma A_i = \sigma B (B \in M \cup \Delta)(1 \leq \forall i \leq n) \\
&\qquad \wedge\ \exists i (1 \leq i \leq n) \sigma A_i = \sigma B (B \in \Delta)\}
\end{aligned}
$$

$$subsumption(\Delta, M) = \{C \in \Delta \mid \forall B \in M (B \text{ dosen't subsume } C)\}$$

3.2 Full-Test Algorithm

Figure 2 shows a refined version of the basic algorithm called the full-test algorithm.

The algorithm

1) selects Δ from D,

2) performs model extension using Δ and M generating *new* for the next generation of Δ,

3) performs a subsumption test on *new* against $M \cup D$, and

4) performs a rejection test on *new'*, that has passed the subsumption test, together with $M \cup D$.

Though this refinement seems to be insignificant at the text level, the time and space requirement is reduced significantly, as explained later. The points are as follows. The algorithm performs subsumption and rejection tests

```
process tester;
    repeat forever
        $request(generator, \Delta)$;
        $\Delta' :- subsumption(\Delta, M \cup D)$;
        if $CJM_T(\Delta', M \cup D) \in \bot$ then return(success);
        $D := D \cup \Delta'$.

process generator;
    repeat forever
        while $Buf = \phi$ do begin
            $D := D - \{e\}$; $Buf :=$ delay$CJM_G(e, M)$; $M := M \cup \{e\}$ end;
        $wait(tester)$;
        $\Delta :-$ force$Buf$;
    until $D = \phi$ and $Buf = \phi$.
```

Figure 3: Lazy algorithm

on all elements of new rather than on Δ, a subset of new generated in the past cycles. As a result, if a falsifying atom [2], X, is included in new, the algorithm can terminate as soon as $false$ is derived from X. That is, the algorithm neither overlooks the falsifying atom nor puts it into D as does the basic algorithm. Thus, it never generates atoms that are superfluous after X is found.

3.3 Lazy Algorithm

Figure 3 shows another refinement of the basic algorithm, the lazy algorithm. In this algorithm, it is assumed that two processes, one for generator clauses and the other for tester clauses, run in parallel and communicate with each other.

The tester process

1) requests Δ to the generator process,

2) performs a subsumption test on Δ against $M \cup D$, and

3) performs a rejection test on Δ.

For the generator process,

1) if a buffer, Buf, used for storing a set of atoms that result from the application of the model extension rule, is empty, the generator selects an atom, e, from D and sets a code for model extension (**delay** CJM) for e and M onto Buf.

[2] A falsifying atom, X, is an atom that satisfies the antecedent of a negative clause by itself or in combination with $M \cup D$

2) waits for a request of Δ from the tester process, and

3) forces the buffer, Buf, to generate Δ.

Where **delay** is an operator that delays the execution of its operand (a function call). Hence, the function call, $CJM_G(e, M)$, will not be activated during 1), but instead is stored in Buf as a code. Later, at 3), when the **force** operator is applied to Buf, the delayed function call is activated. This generates the required values. Using this mechanism, it is possible to generate only the Δ demanded by the tester process. After the required amount of Δ is generated, a delayed function call for generating the rest of the atoms is put into Buf as a continuation.

The atoms are stored in M and D in a way that makes the order of generating and testing the atoms exactly the same as in the basic algorithm. The point of refinement in the lazy algorithm is, therefore, to equalize the speed of generation and testing while keeping the order of atoms that are generated and tested the same as that of the basic algorithm. This eliminates any excess consumption of time and space due to the over-generation of redundant atoms.

3.4 Optimization of Unit Tester Clauses

Given the unit tester clauses in the problem, the three algorithms above can be further optimized. There are two ways to do this.

One is a dynamic method called the lookahead method. In this method, atoms are generated excessively in the generation process to apply the rejection rule with unit tester clauses. More precisely, immediately after generating new, the generator process generates new_{next}, that would be regenerated in a succeeding step. Then new_{next} is tested with unit tester clauses. If the test fails, then new_{next} is discarded whereas new is stored.

$$< \Delta, M > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new$$

$$< new, M \cup D > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new_{next}$$

$$new_{next} \Rightarrow test(A \rightarrow false)$$

The reason why new_{next} is not stored is that testing with unit tester clauses does not require M or D, but can be done with only new_{next} itself. On the other hand, for tester clauses with more than one literal, testing cannot be completed, since testing for combinations of atoms from new_{next} would not be performed.

new_{next} will be regenerated as new in the subsequent step. This means that some conjunctive matching will be performed twice for an identical combination of atoms in a model candidate. However, the increase in computational cost due to this redundancy is negligible compared to the order of the total computational cost. Details are given later.

The second method is a static one that uses partial evaluation. This is used to obtain non-unit tester clauses from a unit tester clause and a set of generator clauses by resolving the two.

$$
\begin{gathered}
Generator: \ A_1, A_2 \rightarrow C. \\
Unit\ tester: \ A \rightarrow false. \\
\Downarrow \\
Non\text{-}unit\ tester: \ \sigma A_1, \sigma A_2 \rightarrow false. \\
\qquad\qquad where\ \sigma C \equiv \sigma A
\end{gathered}
$$

The computational complexity for conjunctive matching in the partial evaluation method is exactly the same as that using the lookahead method. The partial evaluation method, however, is simpler than the lookahead method, since the former does not need any modification of the prover itself whereas the latter does. Moreover, the partial evaluation method may be able to reduce the search space significantly, since it can provide propagating goal information to generater clauses. In general, however, partial evaluation results in an increase in the number of clauses. Hence, it may make performance worse.

The two optimization techniques are equally effective, and will optimize the model generation algorithms to the same order of magnitude when they are applied to unit tester clauses.

4 Complexity Analysis

In this section, we discuss the time and space complexity of the algorithms described above.

To prevent the discussion from becoming too complicated, we assume the following.

1) The problem consists of generator clauses having two antecedent literals and one consequent literal, and tester clauses with at most two literals.

2) Δ is a singleton set of an atom selected from D.

3) The rate at which conjunctive matchings succeed for a generator clause, and atoms generated as the result pass a subsumption test, the survival rate, is $\rho (0 \leq \rho \leq 1)$.

4) The order in which Δ is selected and atoms are generated according to Δ is fixed for all of the three algorithms.

In the following, we use the matrix shown in Section 2.1 to discuss the complexity of the algorithms. This matrix depicts conjunctive matching operations on two literals, L_1, L_2, in the antecedent of a clause. The element at the i th row of the j th column in the matrix represents the result of matching L_1 to the i-th element of a model candidate and L_2 to the j-th element of the model candidate. A number is assigned to each result of the conjunctive matching, if it passes a subsumption test and is fixed as an element of a model candidate. For instance, in the following figure, the axiom initially given in the problem is numbered 1, 1×1 is 2, 2×1 is 3, 1×2 is 4, 2×2 is 5, and so forth.

$$
\begin{array}{c|cccc|c|}
 & 1 & 2 & 3 & \ldots & i \\
\cline{2-6}
1 & 2 & 4 & 8 & & 1\times i \\
2 & 3 & 5 & 9 & \ldots & 2\times i \\
3 & 6 & 7 & 10 & & 3\times i \\
 & & & & & \\
\vdots & & \vdots & & & \vdots \\
 & & & & & \\
\cline{2-6}
i & i\times 1 & i\times 2 & i\times 3 & \ldots & i\times i \\
\cline{2-6}
\end{array}
$$

Where a model extension on the i-th model element, $CJM_{Generator}(i, M)$, means performing $i \times (1 \ldots i-1), (1 \ldots i-1) \times i, i \times i$ for every element in M ($1, \ldots, i-1$), thereby obtaining new atoms that will extend the current model candidate, M. After $CJM_{Generator}(i, M)$ completes, i^2 conjunctive matchings have been performed. Assuming the survival rate is ρ, $\lceil \rho i^2 \rceil$ elements will have been stored in $M \cup D$ [3].

4.1 Basic Algorithm

Figure 3 shows the time and space complexity of the basic algorithm. The area in the left matrix represents the number of conjunctive matchings performed in the generator clauses. The area in the right matrix represents the number of conjunctive matchings performed in the tester clauses with two literals. The length of the line below the right matrix represents the number of conjunctive matchings performed in the unit tester clauses. The region denoted by M and D represents the set of atoms resulting from conjunctive matching in the respective region. In the basic algorithm, atoms in M are those for which both model extension and rejection testing have been completed, atoms in D are those for which neither model extension nor rejection testing has yet been performed.

Figure 4 shows a snapshot of the time when *false* is detected (*false* is depicted by $\perp$ in the figures). Where we assume that the model extension on the m-th element, $CJM_{Generator}(m, M)$, generates the falsifying atom, X. Though the falsifying atom must lie between $\lceil \rho(m-1)^2 \rceil + 1$-th and $\lceil \rho m^2 \rceil$-th, we assume it to be precisely $\lceil \rho m^2 \rceil$-th for brevity.

In the basic algorithm, Δ is selected out of the model-extending candidate (D), then a rejection test is performed with $CJM_{Tester}(\Delta, M)$. Contrary to the full-test algorithm, the rejection test on *new*, which is the result of $CJM_{Generator}(\Delta, M)$, is not performed. Hence, even if a falsifying atom, X, is included in *new*, it is overlooked at this point and stored in D. Therefore,

[3]More precisely, i number of elements is stored in M, $\lceil \rho i^2 \rceil - i$ in D.

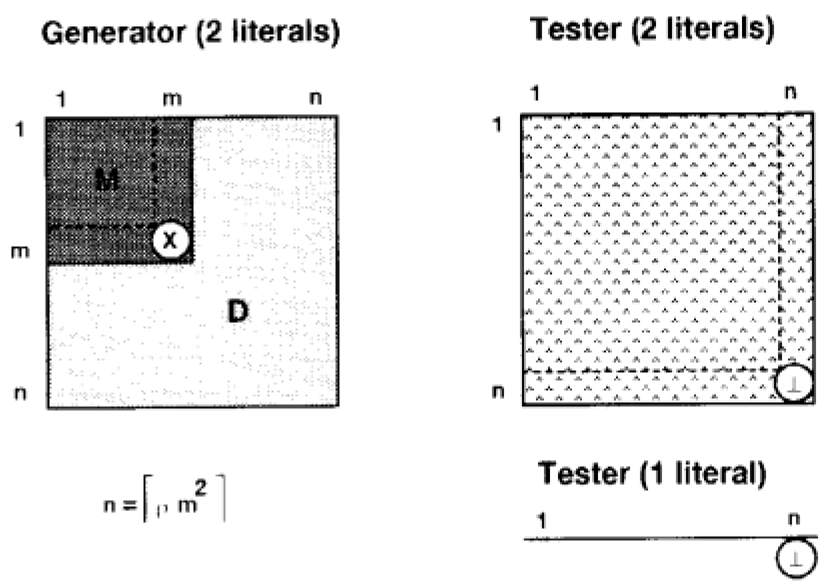


Figure 4: Complexity of basic algorithm

the falsity is not detected until X is selected out of D and the rejection test is performed on it.

As a result, superfluous conjunctive matching will be performed by the generator clauses on the $m+1$-th through $\lceil \rho m^2 \rceil$ th element. When the rejection test is performed on X, $\lceil \rho^2 m^4 \rceil$ conjunctive matchings will have been performed for the generator and tester clauses. Also, the same number of subsumption tests will have been done. In $M \cup D$, $\lceil \rho^3 m^4 \rceil$ atoms are stored. Therefore, the time and space complexity of the basic algorithm is $O(m^4)$.

4.2 Full-Test Algorithm

Figure 5 shows a snapshot of the time when *false* is detected in the full-test algorithm. X and M are the same as in the basic algorithm, but the meaning of D differs in that the model extension rule has not been performed on D and the rejection test has been performed.

In the full-test algorithm, the rejection test is performed not on Δ but on *new*, the result of applying the model extension rule to Δ. Hence, after model extension is performed on the m-th element with $CJM_{Generator}(m, M)$, the falsifying atom, X, is generated and is immediately tested by rejection testing, thereby quickly terminating the algorithm. Since the number of elements in $M \cup D$ is $\lceil \rho(m \times m) \rceil$ [4], the number of conjunctive matchings performed is $(\rho(m \times m))^2 = \rho^2 m^4$.

On the other hand, for the generator clauses, conjunctive matching is performed only for the 1-th through m-th elements, but it is not performed

[4]The number of elements in M is $\lceil (m/\rho)^{1/2} \rceil$, that of D is $\lceil \rho m^2 \rceil - \lceil (m/\rho)^{1/2} \rceil$.

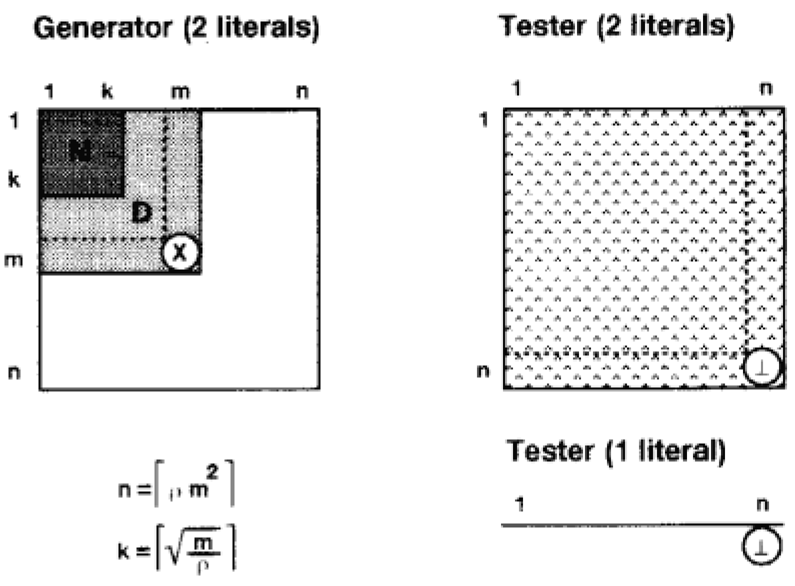


Figure 5: Complexity of the full-test/lazy algorithm

for the $m+1$-th through the $\lceil \rho m^2 \rceil$-th elements [5]. Therefore, the number of conjunctive matchings in the generator process and the size of the memory space for storing generated atoms is reduced to $O(m^2)$ from $O(m^4)$ in the basic algorithm.

4.3 Lazy Algorithm

In the lazy algorithm, Δs are generated one by one as tester clauses demand. The newly generated element is subjected to the rejection and subsumption tests, against the current model elements that have already been tested.

The complexity of the lazy algorithm is exactly the same as that of the full-test algorithm, as shown in Figure 5. This is because the rejection test is done immediately after the new element is generated. Falsity can be detected no later than when the falsifying atom, X, is generated.

Since the lazy algorithm generates atoms on demand, it never over-generates atoms as the basic algorithm would. Also, since the rejection test is done as soon as a new atom is generated, it is as effective in detecting falsity early as the full-test algorithm.

The lazy algorithm is comparable to the full-test algorithm in complexity because the full-test algorithm is assumed to be sequential. Namely, in the full test algorithm, it is assumed that model extension is invoked only after every element in *new*, the result of the current model extension cycle, is tested. However, suppose that the full-test algorithm is run in a parallel environment. Then, the generator process may generate a lot of unnecessary

[5]The white area shaped like a reverse L in the left matrix.

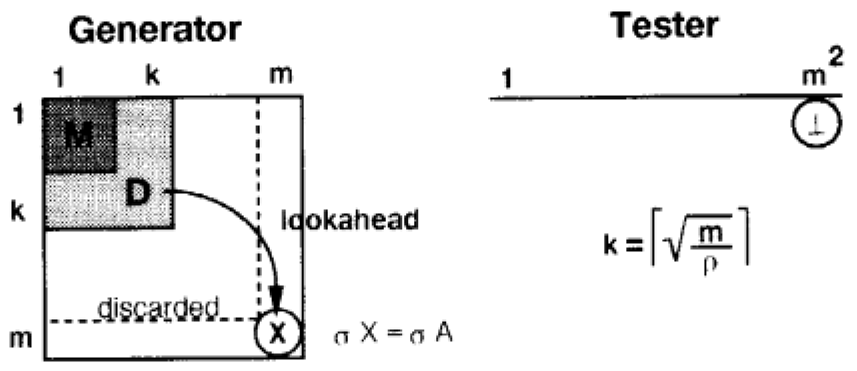


Figure 6: Effect of the lookahead optimization in the full-test/lazy algorithms

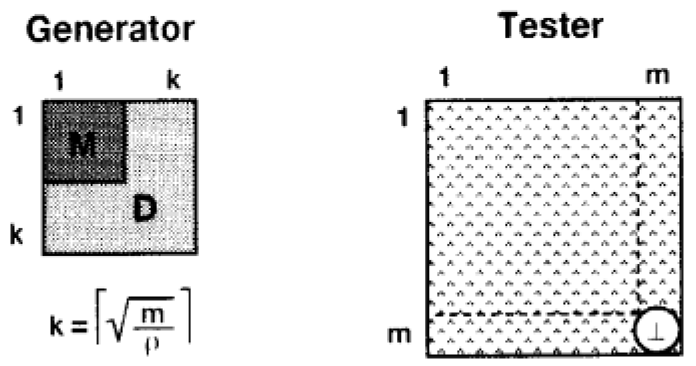


Figure 7: Effect of partial evaluation in the full-test/lazy algorithms

atoms. Hence, a very large number of subsumption tests may be performed before the falsifying atom, X, is generated and tested, unless the process is controlled appropriately. The lazy algorithm, on the other hand, is able to avoid such a generator process overrun without any control other than the laziness even in a parallel environment. Thus, it can always preserve the time and space complexity $O(m^2)$.

4.4 Optimization of Unit Tester Clauses

Both the lookahead and partial evaluation optimizations have the capacity to test one more generation ahead of the current set of atoms being tested.

Under the assumption that the number of literals in the antecedent of a generator clause is two, the number of elements in the model-extending candidate is the square of the number of elements in the model candidate.

In the basic algorithm, the rejection test is not performed on a model-extending candidate but is performed only on model candidates. By applying advanced testing, the rejection test can be applied to model-extending candidates, thereby decreasing the time and space complexity of conjunctive matching processes in the generator by the square root of its order of magnitude. With this optimization, the basic algorithm will be of the same complexity as the full-test and lazy algorithms. This optimization is exactly the same as that in the OTTER system.

On the other hand, in the full-test and lazy algorithms, testing is done in perfect on model-extending candidates. Applying advanced testing to these algorithms, the rejection test can be performed on the next generation ahead of the model-extending atom sets, thereby decreasing the time and space complexity by the order of magnitude of a square root.

In Figure 4, for the basic algorithm, the atoms in the area denoted by D will never be generated after the optimization. Therefore, the time and space complexity of the generator clauses becomes $O(m^2)$.

In Figure 5, for the full test/lazy algorithms, the atoms in the area denoted by D will be tested only after all of the atoms in M are generated. Therefore, the time and space complexity of the generator clauses is reduced to $O(m)$ (Figure 6).

On the other hand, the effect of partial evaluation (Figure 7) due to the resultant two literal tester clauses is equivalent to that of the lookahead optimization in which $new_{\mu,\nu}$ is generated and tested by the original unit tester clauses. Also, in Figures 6 and 7, the atoms in the area denoted by M are computed twice in both optimizations. Thus, the lookahead and partial evaluation optimizations are equivalent in the way they improve time and space complexity.

4.5 Summary of Complexity Analysis

Table 1 summarizes the above discussion on complexity analysis. T/S/G stands for complexity entry of rejection test/subsumption test/model extension. M stands for entry of the required memory space. The value of $\alpha (1 \leq \alpha \leq 2)$ represents the efficiency factor of the subsumption test. $\alpha = 1$ means that a subsumption test is performed in constant order, because the hashing effect is perfect. $\alpha = 2$ means that a subsumption test is performed in a period proportional to the number of elements, perhaps because a linear search was made in the list. For the condensed detachment problem, the hashing effect is very poor and α is very close to two.

The memory space required for the basic, full-test/lazy and lazy lookahead algorithms decreases along this order by a square root for each. This means that the number of atoms generated decreases as the algorithm changes. This, in turn, implies that the number of subsumption tests decreases accordingly. For $\alpha = 2$, the most expensive computation is a subsumption test. A decrease in its complexity means a decrease in the total complexity. On the other hand, for $\alpha = 1$, the most expensive computation is the rejection test with two-literal tester clauses. This situation, however, is the same for all of the algorithms. Adopting lazy computation will result in speedup by a constant factor. In any

Table 1: Summary of complexity analysis

	Unit tester clause			
	T	S	G	M
basic	ρm^2	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
full-test / lazy	ρm^2	$\mu m^{2\alpha}$	m^2	ρm^2
lazy lookahead	m^2	$(\mu/\rho)m^{\alpha}$	m/ρ	m

	2-literal tester clause			
	T	S	G	M
basic	$\rho^2 m^4$	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
full-test / lazy	$\rho^2 m^4$	$\mu m^{2\alpha}$	m^2	ρm^2

† m is the number of elements in a model candidate when *false* is detected in the basic algorithm.
‡ ρ is the survival rate of a generated atom, μ is the rate of successful conjunctive matchings ($\rho \leq \mu$), and α is the efficiency factor of a subsumption test.

case, by adopting lazy computation, the complexity of the total computation is dominated by that of the rejection test.

5 Implementation of Lazy Conjunctive Matching in KL1

This section shows how to implement a lazy conjunctive matching program, the kernel of lazy model generation, in KL1.

5.1 Outline of KL1

KL1 is a parallel logic programming language based on flat guarded Horn clauses[UC90]. A KL1 program consists of clauses of the following form:

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n.$$

where H, G_i, and B_j are called a clause head, a guard goal, and a body goal, respectively. The '|' operator is called the commit operator. The part of a clause before '|' is called a guard, and the part after '|' is called a body. A KL1 clause is executed as follows:

- In contrast to Prolog, all clauses having the same head can be tried in parallel.
- The guard of a clause specifies the conditions for the clause being selected. The guard goals are tested sequentially. If a guard goal cannot succeed without instantiating the variables in the caller goal, then the guard goal suspends until they are instantiated.
- The commit operator selects a clause whose guard has succeeded.

```
clause(M,DM, C, Ui,Uo) :- true |
  ante([{DM,A1},{M,A2}], C, Um1,Uo),  % Δ * M
  ante([{DM,A2},{M,A1}], C, Um2,Um1), % M * Δ
  ante([{DM,A1},{DM,A2}], C, Ui,Um2). % Δ * Δ

ante(R, C, Ui,Uo) :- true |
  new_env(C, Env), % create variable environment
  ante1(R, Env, C, Ui,Uo).

ante1([], Env, C, Ui,Uo) :- true |
  assignValueToVariable(C1, Env), Uo = [C1|Ui].
ante1([{M,Lis}|R], Env, C, Ui,Uo) :-
  literal(M,Lis, R, Env, C, Ui,Uo).

literal(M,Lis, R, Env, C, Ui,Uo) :- true |
  getNext(M, E, M1),
  (/* M is empty */ -> Uo = Ui;
   /* E is a element of M */ -> unify(Lis,E, Env,NEnv),
     (/* unification fail */ ->
       literal(M1,Lis, R, Env, C, Ui,Uo);
      /* unification success */ ->
       ante1(R, NEnv, C, Um,Uo),
       literal(M1,Lis, R, Env, C, Ui,Um))).
```

Figure 8: Eager conjunctive matching program

- The body goals of the selected clause are then executed in parallel.

Using this mechanism, communication between processes and their synchronization is possible.

5.2 Eager Conjunctive Matching

A lazy conjunctive matching program is derived from an eager conjunctive matching program.

To make the program simple, we consider the conjunctive matching of clause C having only two antecedent literals $A_1, A_2 \rightarrow C_1$. In this case, the necessary combinations are $\Delta \times M$, $M \times \Delta$, $\Delta \times \Delta$.

Figure 8 shows an eager conjunctive matching program.

Predicate **clause/5** performs conjunctive matching of an element in M(**M**) and Δ(**DM**) against a literal of the antecedent part. It returns a list of the consequent part of successful combinations in conjunctive matching as d-list **(Ui,Uo)** style.

The predicate **ante/4** unifies M or Δ with a literal **Lis** in accordance with its first argument.

5.3 Lazy Conjunctive Matching

Two lazy conjunctive matching programs are presented. One is the continuation-based program in Figure 9. The other is a process-oriented program suitable for a concurrent environment.

5.3.1 Continuation-Based Implementation

As can be seen from Figures 8 and 9, the difference between an eager program and an continuation-based program is that the lazy program contains a continuation stack `S` for the lazy mechanism and a specified number `L` to indicate the number of elements to be created.

The continuation stack contains goals whose execution is delayed. For example, in the case of `clause/7`, two body goals are pushed onto the stack and their execution is delayed. These delayed goals are popped one by one and forced to execute when the current goal computation is terminated.

This program is defined as a function which returns a continuation stack, so that it is necessary for the caller to manage the continuation stack and call `clause/7` with the stack when it needs elements.

5.3.2 Process-Oriented Implementation

In 5.3.1, `clause` is defined as a function. However, the process-oriented program is suitable in a concurrent environment. For a process-oriented program, we make the generator `clause` and the tester cooperate by using a communication channel.

A process-oriented program is made by adding an extra argument to the continuation based program. The extra argument, **NL**, represents a communication channel between the generator and tester.

When the generator generates the required number of elements, it waits for the next demand from the tester.

```
ante1(L,NL, [], Env, C, S,NS, Uo) :-
  assignValueToVariable(C1, Env), Uo = [C1|Ui], L1 := L - 1,
  (L1 = 0 -> Ui = {Uii},  % mark Lth element
    (NL = {LL,NLL}  ->    % wait for the next L and NL
       clausesCont(LL,NLL, S, Uii));
   ...
```

When the tester needs more elements, it sends the next demand to the generator.

```
...,NL,..., Uo,...) :- Uo = {Uoo} |      % reach Lth element
        NL = {LL,NLL},  % send the next L and NL
        ...
```

```
clause(L, M,DM, C, S,NS, Uo) :- true |
  S1 = [ante([{DM,A2},{M,A1}], C),
        ante([{DM,A1},{DM,A2}], C)|S],
  ante(L, [{DM,A1},{M,A2}], C, S1,NS, Uo).

ante(L, R, C, S,NS, Uo) :- true |
  new_env(C, Env), ante1(L, R, Env, C, S,NS, Uo).

ante1(L, [], Env, C, S,NS, Uo) :- true |
  assignValueToVariable(C1, Env),
  Uo = [C1|Ui], L1 := L - 1,
  (L1 = 0 -> NS = S, Ui = [];
   L1 > 0 -> clauseCont(L1, S,NS, Ui)).
ante1(L, [{M,Lis}|R], Env, C, S,NS, Uo) :- true |
  literal(L, M,Lis, R, Env, C, S,NS, Uo).

literal(L, M,Lis, R, Env, C, S,NS, Uo) :- true |
  getNext(M, E, M1),
  (/* M is empty */ -> clauseCont(L, S,NS, Uo);
   /* E is an element of M */ -> unify(Lis,E, Env,NEnv),
     (/* unification fail */ ->
       literal(L, M1,Lis, R, Env, C, S,NS, Uo);
      /* unification success */ ->
       S1 = [literal(M1,Lis, R, Env, C)|S],
       ante1(L, R, NEnv, C, S1,NS, Uo))).

clauseCont(L, [ante(R,C)|S],NS, Uo) :- true |
  ante(L, R, C, S,NS, Uo).
clauseCont(L, [literal(M,Lis, R, Env, C)|S],NS, Uo) :- true |
  literal(L, M,Lis, R, Env, C, S,NS, Uo).
```

Figure 9: Continuation-based program

6 Results

6.1 Features of Hard Horn Problems

Before presenting the results obtained with the proposed algorithms, we briefly describe the problems we are trying to solve.

All problems are given as a set of Horn clauses only, as follows.

Theorem 4 (XGK [Ove90])

$$\begin{aligned} p(X),\ p(e(X,Y)) &\rightarrow p(Y). \\ true &\rightarrow p(e(X,e(e(Y,e(Z,X)),e(Z,Y)))). \\ p(e(e(e(a,e(b,c)),c),e(b,a))) &\rightarrow false. \end{aligned}$$

Theorem 6 (Lukasiewicz)

$$\begin{aligned} p(X),\ p(i(X,Y)) &\rightarrow p(Y). \\ true &\rightarrow p(i(X,i(Y,X))). \\ true &\rightarrow p(i(i(X,Y),i(i(Y,Z),i(X,Z)))). \\ true &\rightarrow p(i(i(i(X,Y),Y),i(i(Y,X),X))). \\ true &\rightarrow p(i(i(n(X),n(Y)),i(Y,X))). \\ p(i(i(a,b),i(i(c,a),i(c,b)))) &\rightarrow false. \end{aligned}$$

Besides full unification with occurrence check, we need heuristics such as the weighting and deletion of some complex resolvents, and a control mechanism that alternates between breadth-first and depth-first search strategies. Incidentally, Theorem 6 has a very short proof and can be proven easily with only the breadth-first search strategy. Theorem 4, however, is much harder and its proof is longer.

These problems have the following characteristics: 1) The number of conjunctive matchings and subsumption tests is enormous. 2) The size of the model candidates becomes very large.
e.g. When ten thousand atoms are generated, conjunctive matching will amount to one hundred million.

6.2 Performance

Some results are shown in Tables 2 and 3. We did not use heuristics such as weighting and sorting, but only limited term size and eliminated tautologies.

Each algorithm is implemented in KL1 and run on a pseudo Multi-PSI in PSI-II [NN87]. The OTTER entry represents the basic algorithm optimized for unit tester clauses and implemented in KL1. The figures in parentheses are of algorithms for tester clauses with two literals resulting from the application of partial evaluation to unit tester clauses. In unify entries, a figure to the left of + represents the number of conjunctive matchings performed in tester clauses. A figure to the right of + represents the number of conjunctive matchings performed in generator clauses.

These results are a fair reflection of the complexity analysis shown in Table 1. For instance, to solve Theorem 4 without partial evaluation optimization.

Table 2: Results (Theorem 4)

		basic	full-test	lazy	lazy lookahead	OTTER
Time (sec)		>14000	409.17	407.58	210.45	409.16
		(463.86)	(82.40)	(81.82)	(81.69)	(462.13)
Unify			1656+74800	1656+74737	81956+4095	1656+74800
		(43981+74254)	(43981+4158)	(43981+4158)	(43981+4095)	(43981+74254)
Subsumption test			5736	5736	593	5736
		(5674)	(596)	(596)	(593)	(5674)
Memory	M		272	272	63	272
		(272)	(63)	(63)	(63)	(272)
	D		1384	1384	209	1384
		(1375)	(209)	(209)	(209)	(1375)

the basic algorithm did not reach a goal within 14,000 seconds, whereas the full-test and lazy algorithms reached the goal in about 400 seconds. The most time consuming computation in all of the three algorithms (basic, full-test and lazy), is rejection testing. The difference in the time complexity between the basic algorithm and the other two algorithms is $(\mu\rho^2 m^{4\alpha})/(\mu m^{2\alpha}) = \rho^2 m^{2\alpha}$, that results in the time difference mentioned above.

The basic algorithm and the full-test/lazy algorithm do not differ in the number of unifications performed in the tester clauses. However, the number of unifications performed in the generator clauses and the number of subsumption tests decreases as we move from the basic algorithm to the full-test and lazy algorithms. The decrease is about one hundredth when partial evaluation is not applied, and about one tenth when it is applied.

By applying lookahead optimization, the lazy algorithm is further improved. Though the lookahead optimization and the partial evaluation optimization are theoretically comparable in their order of improvement, their actual performance is sometimes very different. For Theorem 4, the lazy algorithm optimized with partial evaluation took 81.82 seconds, whereas the same algorithm optimized with lookahead optimization took 210.45 seconds. This difference is caused by the difference in the number of unifications performed in the tester clauses. This is because in the lazy algorithms with lookahead optimization, the generator clause, $p(X), p(e(X, Y)) \rightarrow p(Y)$, generates an atom before the unit tester clause, $p(A) \rightarrow false$ tests the atom. In the same algorithm, with partial evaluation optimization, the instantiation information of A is propagated to the antecedent of $p(X), p(e(X, A)) \rightarrow false$ and the unification failure can be detected earlier.

Partial evaluation optimization is effective for all algorithms except OTTER. This is because lookahead optimization, in the OTTER algorithm, is already applied to unit tester clauses, and the algorithm remains the basic one for non-unit tester clauses.

Typically in the problems treated here, 1) survival rate (ρ) is almost constant and the complexity of the total computation is dominated by m, and 2) the efficiency factor of the subsumption test (α) is very close to two, and

Table 3: Results (Theorem 6)

		basic	full-test	lazy	lazy lookahead	OTTER
Time (sec)		1647.38 (13.22)	12.82 (2.23)	12.28 (2.29)	3.51 (2.22)	12.73 (12.60)
Unify		220+48618 (358+550)	220+598 (358+70)	220+556 (358+68)	947+68 (358+68)	220+598 (358+550)
Subsumption test		18804 (328)	331 (32)	327 (28)	28 (28)	331 (328)
Memory	M	220 (23)	23 (7)	23 (7)	7 (7)	23 (23)
	D	5369 (195)	197 (16)	197 (16)	16 (16)	197 (195)

the cost of subsumption tests increases as the number of elements in a model candidate increases. This, ultimately, dominates the total computation cost. Thus, the results reported here well reflect the results of the complexity analysis in Section 4.5.

7 Conclusion

It is important to, as much as possible, avoid explosive increases in the amount of time and space consumed, when proving hard theorems that require deep inferences. For this we proposed the lazy model generation method and techniques to improve its performance.

On a sequential machine, it is sufficient to use the full test algorithm since it has a similar effect to the lazy algorithm with respect to the order of computational amount and required memory space. However, in a parallel environment, the full-test algorithm may generate substantial numbers of unnecessary atoms and perform an excess number of subsumption tests. On the other hand, the lazy algorithm has the greatest effect when used on a parallel machine. This mechanism can be incorporated to hyper-resolution based provers thereby achieving a significant performance improvement.

Results show that significant savings in computational amount and memory space can be realized by using the full test or the lazy algorithm optimized for unit tester clauses.

The lazy model generation method can be easily extended from Horn clauses to non-Horn clauses. The idea is also applicable to hyper-resolution or set-of-support strategy based provers in general.

Whereas lazy model generation can achieve 'generate-only-at-testing', it is important to consider 'generate-only-for-testing' that is more general and advanced. By this term we mean incorporating search-pruning strategies such as *magic sets* and *relevancy testing* [WL89]. We believe that these strategies would fit well with the lazy model generation method since its control structure is very simple.

We are now developing a parallel prover based on the lazy model generation method. The result of this will be reported in a forthcoming paper.

Acknowledgments

We would like to thank Kazuhiro Fuchi and Koichi Furukawa for giving us the opportunity to do this research. We also wish to thank Mark E. Stickel and François Bry for their helpful comments on drafts of this paper, as well as Masayuki Fujita, Fumio Mizoguchi and PTP-WG members for useful discussions.

References

[FH91] H. Fujita and R. Hasegawa, A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm. In *Proc. of the Eighth International Conference on Logic Programming*, The MIT Press, 1991.

[Has91] R. Hasegawa, A Parallel Model Generation Theorem Prover: MGTP and Further Research Plan. In Proc. of the Joint American-Japanese Workshop on Theorem Proving, Argonne, Illinois, 1991.

[MB88] R. Manthey and F. Bry, SATCHMO: A Theorem Prover Implemented in Prolog. In *Proc. of CADE 88, Argonne, Illinois*, 1988.

[McC90] W. W. McCune, OTTER 2.0 Users Guide, Argonne National Laboratory, 1990.

[MW91] W. W. McCune and L. Wos, Experiments in Automated Deduction with Condensed Detachment, Argonne National Laboratory, 1991.

[NN87] H. Nakashima and K. Nakajima, Hardware Architecture of the Sequential Inference Machine PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Computer Society Press of the IEEE,1987.

[Ove90] R. Overbeek, Challenge Problems, (private communication) 1990.

[SL91] J. K. Slaney and E. L. Lusk, Parallelizing the Closure Computation in Automated Deduction, In *Proc. of CADE 90*, 1990.

[Sch89] J. Schumann, SETHEO: User's Manual, Technische Universität München, 1989.

[Sti88] M. E. Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. In *Journal of Automated Reasoning*, 4:353-380, 1988.

[UC90] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine, In *Computer Journal*, Dec. 1990.

[Wos88] L. Wos, *Automated Reasoning — 33 Basic Research Problems —*, Prentice-Hall, 1988.

[WL89] D. S. Wilson and D. W. Loveland, Incorporating Relevancy Testing in SATCHMO, CS-1989-24, Department of Computer Science, Duke University, Durham, North Carolina, 1989.