ICOT Technical Report: TR-726

TR-726

Lazy Model Generation for Implementing Efficient Theorem Provers

by
R. Hasegawa, M. Koshimura & H. Fujita

January, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Lazy Model Generation for Implementing Efficient Theorem Provers

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

Miyuki KOSHIMURA
Toshiba Information Systems (Japan)

Hiroshi FUJITA
Mitsubishi Electric Corporation

Abstract

This paper presents a new method called *lazy model generation* for implementing efficient theorem provers. The tasks of theorem proving based on model generation are the generation and testing of atoms that will be members of a model. The generation process tends to fall into combinatorial explosion which makes it difficult to obtain proofs. Lazy model generation is a method aimed at avoiding an over generation of atoms that are irrelevant and unnecessary for obtaining proofs. In fact, without serious consideration of laziness in producing and retaining information of any kind, theorem proving for hard problems would never succeed with a limited computational resources. The lazy model generation method also allows the exploitation of parallelism with fairly good load distribution.

1 Introduction

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

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

As a first step for developing KL1-technology theorem provers, we adopted the model generation method as the basis. The reasons are 1) SATCHMO's property that unification is not necessary is very convenient for us for implementing provers in KL1 since KL1, as a committed choice language, does not support backtrackable 'full' unification, and 2) it is easier to incorporate a mechanism for lemmatization, subsumption test, and other deletion strategies which are indispensable for solving difficult problems such as Lukasiewicz problems [Ove90].

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 RAMS method [FH91] and MERC method [Has91].

A more important issue with regard to 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 which require deeper inferences (longer proofs) such as Lukasiewicz problems. In order to solve this problem, it is important to recognize that proving processes are viewed exactly as *generation-and-test* processes and that generation should be performed only when testing requires it.

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 due to the generation process. However, this control makes it impossible to incorporate useful strategies such as weighting heuristics which require items to be generated in aggregates and then sorted in accordance with the weights calculated for each item.

On the other hand, in the case of orthodox provers such as OTTER [McC90], they 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 usually is no serious distinction between generation and test in their implementation. Without the generation-and-test way of thinking, a naive implementation would produce redundant computation in 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 or 'generate-only-at-test' is implemented.

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* $(n \neq 0, m \neq 0)$.

The model generation method has the following two rules:

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

- Model rejection rule: If there is a negative clause whose antecedent $A\sigma$ is satisfied in a model M, then reject model M.

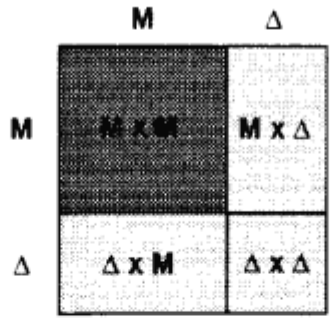
We call the process of obtaining $A\sigma$, a *conjunctive matching (CJM)* of the antecedent literals against elements in a model. Note that the antecedent (*true*) of a positive clause is satisfied by any model.

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. The method can also be used to prove that the clause set is unsatisfiable. This is done by exploring every possible model candidate to see that no model exists for the clause set.

2.1 Avoiding redundancy in conjunctive matching

Imagine that we have a clause, C, of two literals in the antecedent. To perform conjunctive matching for the clause, we need to make a pair of atoms picked 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 which is the 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 make a pair of atoms picked out of $M + \Delta$. The number of pairs amounts to:

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



It should be noted here that $M \times M$ pairs had already been 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 time. Instead, we have to choose just 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 the OTTER to avoid the above sort of redundancy is as follows. For a clause,

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

first, we 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 of them against an atom picked out of $M + \Delta$, 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 $< \Delta, M + \Delta >$ and $< M + \Delta, \Delta >$, the conjunctive matching of $\Delta * \Delta$ are duplicated. This redundancy can be removed with the MERC method.

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

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

for clauses with two antecedent literals, and

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

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 [Has91] method. The MERC method, however, needs to prepare 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 so as not to rematch the same literal against the same model element. Both the Δ-M and the MERC method still contain a 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. However, it tends to require a lot of memory to store the information of the partial matching.

3 Algorithms for Model Generation

In this section and the next, we present several algorithms for model generation and compare them from a viewpoint of computation amount and memory space. In the following sections, we assume that problems are given with Horn clauses only to make the presentation easy to understand. As a matter of fact, most of the challenging problems we are attempting to solve are represented only with Horn clauses, though they require very deep inference. The idea, however, can be generalized for problems with non-Horn clauses as well.

$M := \phi$;
$D := \{A \mid (true \rightarrow A) \in$ a set of given clauses$\}$;
while $D \neq \phi$ **do begin**
    $D := D - \Delta$;
    if $C'JM_{T,stc}(M, \Delta) \ni false$ **then return**(success);
    $new := C'JM_{Generator}(M, \Delta)$;
    $M := M + \Delta$;
    $new' := subsumption(new, M + D)$;
    $D := D + new'$;
end return(fail)

Figure 1: Naive algorithm

3.1 Naive algorithm

Figure 1 shows a very naive algorithm for the model generation method. This is essentially the same algorithm as the one taken by OTTER [McC'90] [1].

In the algorithm, M represents a model candidate (a set of atoms), D represents a set of atoms to be included in model candidates, and Δ represents a subset of D. The initial values for M and D are set to an empty set and a set of consequent atoms of the positive clauses. One cycle of the algorithm comprises 1) choosing a subset, Δ, in D, 2) performing conjunctive matching for the negative clauses, using Δ and M, 3) terminating the algorithm when refutation succeeds, 4) performing conjunctive matching for the mixed clauses, using Δ and M, and 5) performing a subsumption test for newly generated atoms, new against $M + D$. When D is empty at the beginning of the cycle, the algorithm terminates due to refutation failure, or finding a model.

The conjunctive matching and subsumption test are each represented by a function on a set as follows:

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

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

3.2 Basic algorithm

Figure 2, which we call a basic algorithm, shows a modified version of the naive algorithm.

[1] OTTER is a bit more advanced in that tests by unit negative clauses are performed for every atom in new immediately after their generation as in the basic algorithm described in the next subsection.

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

Figure 2: Basic algorithm

In this algorithm, the conjunctive matching for negative clauses is performed after conjunctive matching for mixed clauses and the subsumption test for the newly generated atoms. This modification seems to be quite small in appearance, but contributes, in fact, a large improvement to the total amount of computation and space, as described in the next section. The point is as follows. Tests with negative clauses are performed for all of the newly generated atoms new instead of Δ. As a result, no atom, say X, which can immediately satisfy the antecedent of any negative clause, is added into D. Therefore, we can prevent the generation of irrelevant atoms before X is found.

3.3 Lazy algorithm

The basic algorithm can be improved by further reducing the number of generated atoms that are irrelevant for the test by negative clauses. For this we suppress the generation process so as to make only a small number of atoms at a time, according to the speed of the test process for negative clauses.

A naive implementation of conjunctive matching in imperative languages would be similar to the following:

for $C := 1$ *to* $\mid MixedClauses \mid$ **do begin**
    for $L := 1$ *to* $\mid antecedent(clause(C)) \mid$ **do begin**
        for $A := 1$ *to* $\mid M \mid$ **do begin**
            do matching literal(L) of clause(C) against atom(A) ;
            if a new atom, A_{new}, is generated **then** $output(A_{new})$
end end end

It is difficult to pause the iteration at an arbitrary point in the triple nested for-loop and to resume from that point afterward. To do this we might as well reconstruct the iteration as follows:

```
while goal is not reached do begin
    i := N ; (* No. of atoms to be generated *)
    while i ≠ 0 do begin
        do matching literal(L) of clause(C) against atom(A) :
        if a new atom, A_new, is generated then begin
            output(A_new); i := i - 1 end :
        < C', L', A' >:= advance(< C, L, A >)
end  end
```

where $advance(< C, L, A >)$ updates a triple of counters so as to enumerate all the possible combinations of values for the counters.

In some functional languages, a 'delay and force' mechanism may be available, which would make implementation of the lazy generation process simple and clear.

In parallel logic languages such as KL1, we can represent generator and tester as independently iterating processes and make them cooperate by using a stream for interchanging information. Figure 3 shows an abstract algorithm for implementing this idea. The stream, S, is used for carrying incomplete messages sent and received by the generator and the tester. The generator observes S and upon receiving an incomplete message it generates N atoms(new) and puts the unsubsumed atoms (new') on S. The tester, however, waits for the message being completed by $wait(content(S))$, and upon receiving the Δ_I written by the generator it creates a new incomplete message on S by $newRequest(S)$, after making tests with negative clauses.

This algorithm can be written in KL1 as demand-driven processes very easily as shown in Section 5.

Incidentally, in SATCHMO, the same effect as the lazy algorithm is attained by alternating assertion of a new fact and testing with a negative clause. The enumeration for every possible generation of facts can be performed by using backtracking.

3.4 Optimization

The three algorithms mentioned above can be further improved if there are unit negative clauses in the given clauses. There are two ways to do this.

One is a dynamic way called the lookahead method. The idea, here, is to over-generate atoms for testing with unit negative clauses. Namely, immediately after generating new, we further generate new_{next} which was supposed to be generated in the next phase. Then we test new_{next} with the unit negative clauses. If the test succeeds, we can terminate the algorithm without storing new as well as new_{next}.

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

$$< M, new > \Rightarrow generate(A_1, A_2 \leftarrow C') \Rightarrow new_{next}$$

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

```
M_T := φ;
M := φ;
D := {A | (true — A) ∈ a set of given clauses};
Δ := φ;
S : Stream;
initialize(< C, L, A >) ;
do in parallel
    [ while request comes in S do begin (* generation *)
        i := N ; new = φ ;
        while i ≠ 0 do begin
            do matching literal (L) of clause(C) against atom(A) :
            if a new atom, A_new, is generated then begin
                put(new, A_new); i := i − 1 end ;
            < C', L', A' >:= advance(< C, L, A >)
            if CJM for Δ completed then begin
                D := D − Δ;
                if D = φ then begin
                    if new = φ then return (fail) :
                    i := 0 ;
                end
                else begin
                    M := M + Δ; Δ := subset(D)
                end
            end
        end :
        new' := subsumption(new, M + D);
        put(S, new') ; D := D + new';
    end ] ||
    [ do forever begin (* test *)
        Δ_T := wait(contents(S));
        if CJM_Tester(M_T, Δ_T) ∋ ⊥
        then return(success);
        M_T := M_T + Δ_T;
        newRequest(S)
    end ]
```

Figure 3: Lazy algorithm

Every one of new_{next} which does not match any unit negative clause is discarded. If all of new_{next} fails the test, then they are regenerated in the next phase. This implies that some conjunctive matching may be performed twice for the same information. This increase of computation, however, is negligibly small compared to the significant reduction of total computation. The detail is discussed in the next section.

Another is a static way using partial evaluation. Namely, we unify negative clauses with consequents of mixed clauses to obtain partially evaluated nonunit negative clauses.

$$Generator: \ A_1, A_2 \rightarrow C.$$

$$Unit\ tester: \ A \rightarrow false.$$

$$Partially\ evaluated\ tester: \ \sigma A_1, \sigma A_2 \rightarrow false.$$

$$where\ \sigma C = \sigma A$$

In this method, the computation amount for conjunctive matching is the same as that of the lookahead method. However, this method is simpler than the lookahead method since the prover itself need not be modified in this method. In addition, whereas the lookahead method can reach the goal only one phase earlier, the partial evaluation allows, in principle, the goal to be reached more than one phase earlier by partially deducing between clauses at suitable degree of depth. In doing so, it may also be possible to prune search spaces owing to the propagation of useful information inherent in the negative clauses (goals). There may, however, be some demerits since the number of clauses tends to increase after partial computation.

The two methods are both effective for achieving further improvement in the presence of unit negative clauses. One may choose the most suitable one of the two in accordance with the assessment on the problem and on the merits and demerits of the methods.

4 Complexity Analysis

In this section, we discuss the computational amounts and memory space required by the above proposed algorithms.

To make discussion as simple as possible, we make several assumptions as follows: 1) the problem includes mixed clauses with only two literals in the antecedent, and negative clauses with a maximum of two literals, 2) Δ is assumed to be a single atom, 3) unification never fails in conjunctive matchings, 4) newly generated atoms can never be subsumed by older atoms. These imply that as a result of the conjunctive matching for the i th atom picked out of D and $i-1$ number of atoms in M, $2i-1$ number of new atoms are generated and the total number of atoms generated becomes i^2.

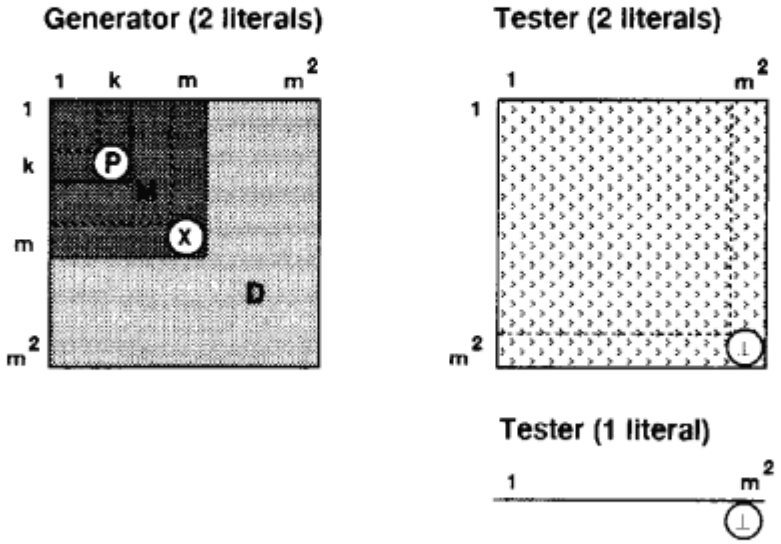


Figure 4: Complexity of the naive algorithm

4.1 Naive Algorithm

Figure 4 depicts the computational amount and memory space of the naive algorithm. The left square area represents the number of conjunctive matchings performed in a generator clause, which is equal to the total number of generated atoms. The right square area represents the number of conjunctive matchings performed in a tester clause with two literals, whereas the length of the line under the square represents that in a tester with one literal.

In the left square, the area indicated by M represents that the conjunctive matchings for generation between the atoms in M have been completed and that false checks have been completed for the atoms in M. The area indicated by D represents that neither conjunctive matching for generation for false checks has been completed yet.

When the generation for the m^2-th element ($m^2 \times m^2$) is completed, m^2 number of the generated atoms are stored in M and the rest ($m^4 - m^2$ number) of the atoms are stored in D. X is a falsifying atom that matches against one of the tester clauses. Let X be generated as the m^2-th atom as a result of the conjunctive matching for the m-th atom, P. In the naive algorithm, tests with negative clauses are performed on Δ, but not on new. Namely, $false(\perp)$ is not derived immediately after X is generated. The falsity is not recognized until all the atoms in D generated before X are picked out of D and conjunctive matchings for them are completed. Therefore, when X is tested, m^4 number of conjunctive matchings and subsumption tests will have been completed both in the generator and the tester clauses. Also the same number (m^4) of atoms are retained in D.

Thus, the computational amount and the memory space required for the

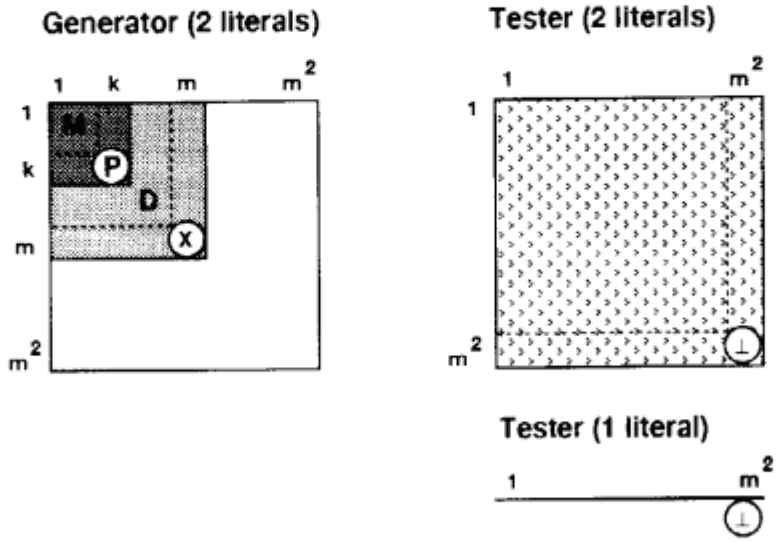


Figure 5: Complexity of the basic and the lazy algorithm

naive algorithm is $O(m^4)$.

4.2 Basic algorithm

Figure 5 is for the basic algorithm. The meanings of P, X and M are the same as in the naive algorithm. The meaning of D, however, differs in that false checks have been completed although the conjunctive matchings have not been completed.

In the basic algorithm, false check is made for new, obtained by the conjunctive matching in the generator instead of Δ. Thus, falsity can be recognized immediately after the falsifying atom, X, is generated. Therefore, although the number of conjunctive matchings in the tester is the same as that of the naive algorithm, the number of conjunctive matchings and subsumption tests in the generator and the memory space for storing generated atoms are reduced from m^4 to m^2.

4.3 Lazy algorithm

The lazy algorithm can be applied both to the naive algorithm and to the basic algorithm.

In the lazy algorithm, atoms are generated one at a time upon requests issued by the tester. The generated atom goes through false tests and subsumption tests using M and D, which are sets of atoms already generated. The atoms that survived both the tests are stored in D.

When we apply the lazy algorithm to the naive algorithm , the complexity of the naive-lazy algorithm is the same as that of the basic algorithm shown in Figure 5, since tests for atoms are performed immediately after their generation and the algorithm can terminate as soon as the falsifying atom, X, is

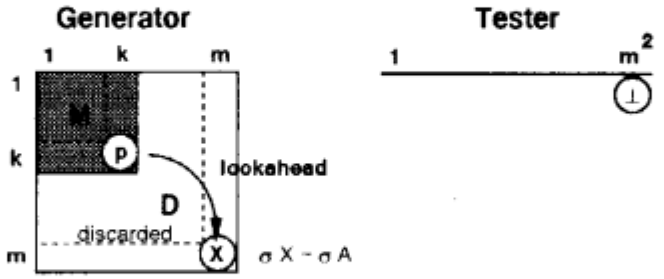


Figure 6: Effect of the lookahead optimization (Basic and Lazy)

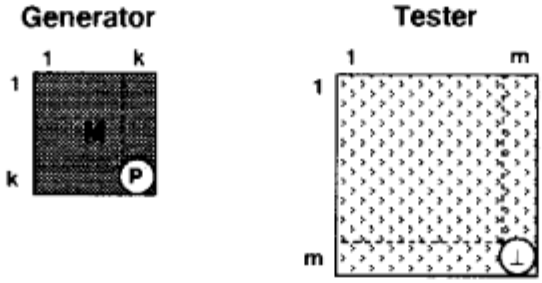


Figure 7: Effect of the partial evaluation (Basic and Lazy)

generated and tested. Hence, the order of the computation amount and the memory space of the naive-lazy algorithm will be improved to the same as that of the basic, namely, from $O(m^4)$ to $O(m^2)$.

On the other hand, the basic algorithm, if used in a sequential execution environment, will not be improved by applying the lazy algorithm. This is because the required computation to reach the falsifying atom, X, cannot be reduced simply by the laziness of its generation. However, when the basic algorithm runs in a parallel execution environment, the generation process might overrun and make a lot of unnecessary atoms before the falsifying atom, X, is recognized unless the process is properly controlled. In such a case, the lazy algorithm naturally makes it possible to avoid having the generator overrun, thereby keeping the order of computation amount and memory space as it should be, namely $O(m^2)$.

4.4 Optimization

In the lookahead optimization, with the currently generated atoms, new, its descendant atoms, new_{next}, are generated in the same phase. These new_{next} are tested by the unit negative clauses, and if the test succeeds, the algorithm can terminate at once. Otherwise, new_{next} are discarded, and the algorithm enters the next phase of generation.

This lookahead effect enables to make the untested set of atoms, D, be tested. For the naive algorithm, D in Figure 4 represents the untested set of atoms. Since this area can be discarded by performing the lookahead, the computational amounts and the memory space are reduced from $O(m^4)$ to $O(m^2)$. For the basic and the lazy algorithms, D in Figure 5 represents the tested set of atoms. However, D should have been untested at the previous phase of generation, and it should have been discarded at the previous phase. Thus, the lookahead has an effect in this case as well, leading to an improvement from $O(m^2)$ to $O(m)$.

On the other hand, in the partial evaluation optimization, when the m-th atom, P, is generated, falsity is detected by the partially evaluated negative clauses having two literals. The task of conjunctive matching with two literal negative clauses is equivalent to the generation of new_{next} and test of them with unit negative clauses in the lookahead optimization. In addition, the area specified by M is recomputed twice for both optimization methods. Therefore, the lookahead and the partial evaluation methods are equivalent in terms of computational amount and memory space.

With these optimization methods, in the case of the basic or the lazy algorithm, we can reduce the complexity from $O(m^4)$ to $O(m^2)$ for the tester and from $O(m^2)$ to $O(m)$ for the generator as shown in Figure 6 and 7.

5 Implementation of Lazy Conjunctive Matching in KL1

This section shows how to implement a lazy conjunctive matching program, which is the kernel part of lazy model generation, in KL1. A lazy conjunctive matching program is derived from an eager conjunctive matching program.

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

5.1 Eager Conjunctive Matching

We first show an eager conjunctive matching program in Figure 8.

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 success combinations in conjunctive patching as d-list (`Ui,Uo`) style.

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

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

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

ante1([], Env, C, Ui,Uo) :-
  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) :- 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 Program

5.2 Lazy Conjunctive Matching

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

5.2.1 Continuation Based Implementation

As seen from Figure 8 and Figure 9, the difference between the eager program and the continuation based program is that the lazy program contains a continuation stack S for the lazy mechanism and a specified number L which indicates the number of elements to be created. The continuation stack corresponds to a triple of counters in section 3.3.

The continuation stack contains goals whose execution is delayed to execute individually. For example, in the case of `clause/7`, two body goals are pushed onto the stack and these executions are delayed. These delayed goals are popped one by one and forced to execute at the terminal condition of the program.

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.

5.2.2 Process Oriented Implementation

In 5.2.1, `clause` is defined as a function. However, the process oriented program is suitable in a concurrent environment. With 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) :-
  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) :-
  new_env(C, Env), ante1(L, R, Env, C, S,NS, Uo).

ante1(L, [], Env, C, S,NS, Uo) :-
  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) :-
  literal(L, M,Lis, R, Env, C, S,NS, Uo).

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

Figure 9: Continuation Based Program

6 Experimental Results

6.1 Features of hard Horn problems

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

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

Theorem 4 (XGK [Ove90])

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

Theorem 6 (Lukasiewicz)

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

Besides full unification with occurs check, we need various 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 just the breadth-first search strategy. Theorem 4, however, is much harder and its proof is longer.

These problems have the following characteristics:

- The number of conjunctive matchings and subsumption tests is enormous.
- The size of model candidates becomes very large.
  e.g. When ten thousand atoms are generated, conjunctive matching will amount to one hundred million.

6.2 Performance measurement

We show some experimental results in Table 1.

In the OTTER algorithm, the basic algorithm is performed for unit false clauses and the naive algorithm is applied for non unit false clauses. A number in parentheses is the result of applying partial evaluation to unit false clauses. As for the unify entries, to the left of the plus sign is the number of conjunctive matchings performed by the tester, to the right is that for the generator. The naive algorithm cannot reach the goal within 4 hours of running without partial evaluation.

Table 1: Performance results (Theorem 4)

	Naive	Lazy	Lazy+LA	Otter
time (sec)	>14000 (463.858)	407.577 (81.822)	210.449 (81.689)	409.161 (462.128)
unify	 (43981+74254)	1656+74737 (43981+4158)	81956+4095 (43981+4095)	1656+74800 (43981+74254)
subsumption test	- (5674)	5736 (596)	593 (593)	5736 (5674)
M	 (272)	1656 (272)	272 (272)	272 (272)
D	 (1375)	-		1384 (1375)

The lazy algorithm, however, can solve it in about 400 seconds. Comparing the naive and lazy, the number of unifications performed by the tester are the same. However, the number of generator unifications and subsumption tests in the lazy is less than one tenth of that in the naive. Also, the lazy algorithm requires less memory than the naive. Partial evaluation is effective for the naive, lazy, and lazy+LA(lookahead), but not for the OTTER. This is because the OTTER has already employed the basic algorithm for unit false clauses, so that it is equivalent to employing lazy, and because, for non-unit clauses, the OTTER is comparable to the naive.

7 Conclusion

It is important to avoid computation and space explosion in proving hard theorems which require deep inferences. For this we proposed the lazy model generation method.

When we use a sequential machine, it is sufficient to use the basic algorithm presented in this paper since it has a similar effect to the lazy algorithm with respect to the order of computational amount and memory space. The lazy algorithm has the most effect when it is used on a parallel machine. We can incorporate this mechanism to OTTER-like provers based on the naive algorithm. This would improve the efficiency to the same degree as in the basic algorithm.

Experiments show that significant computational amounts and memory space can be saved by using the lazy algorithm.

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

Whereas the lazy model generation achieved 'generate-only-at-testing', it is important to consider the more general slogan, 'generate-only-for-testing'. By this term we mean 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 for it has very simple control structure.

We are now developing a parallel prover based on lazy model generation. This result will be reported in a forthcoming paper.

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.

[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 91*, 1991.

[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.

[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.