TR-145

Making Exhaustive Search Programs Deterministic

by

Kazunori Ueda

November, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Making Exhaustive Search Programs Deterministic

Kazunori Ueda

ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo, 108 Japan

ABSTRACT

This paper presents a technique for compiling a Horn-clause program intended to be used for exhaustive search into a GHC (Guarded Horn Clauses) program. The technique can be viewed also as a transformation technique for Prolog programs which eliminates the 'bagof' primitive and non-determinate bindings. The class of programs to which our technique is applicable is shown with a static checking algorithm; it is nontrivial and could be extended. An experiment on a compiler-based Prolog system showed that our transformation technique improved the efficiency of exhaustive search by 6 times in the case of a permutation generator program. This transformation technique is important also in that it exploits the AND-parallelism of GHC for parallel search.

1. INTRODUCTION

We often use the Horn-clause logic, or more specifically the language Prolog, to obtain all solutions of some problem, that is, to obtain all answer substitutions for the variables in a goal to be solved. Under this framework, however, it is difficult to 'collect' the obtained solutions into a single environment in which to make further processing such as counting the number of the solutions, comparing them, classifying them, and so on. This is because these solutions correspond to different, independent paths of a search tree. For this reason, many of Prolog implementations support as primitives system predicates for creating a list of solutions of a goal given as an argument; examples are 'setof' and 'bagof' of DEC-10 Prolog. [Naish 85] contains a survey of all-solutions predicates in various Prolog systems. These predicates, however, internally use some extralogical features to record the obtained solutions. So it should be an interesting question whether it is impossible to do exhaustive search without such primitives.

Another motivation is that we may sometimes wish to do exhaustive search in GHC [Ueda 85] or other parallel logic programming languages which do not directly support exhaustive search. In this case, parallelism inherent in GHC should be effectively used for the search. (We omit the description of GHC in this paper; it can be found in [Ueda 85] and a very brief explanation can be found also in [Ueda and Chikayama 85].)

One possible way to achieve the above requirements is to directly write a first-order relation which states, for example, that "S is a list of all solutions of the N-queens problem". It is almost evident that such a relation can be described within the framework of the Horn-clause logic. However, in practice, it is much harder to write it manually than to write a program which finds only one solution at a time. A programming tool which automatically generates an exhaustive search program could resolve this situation, and this is the way which we will pursue in this paper.

2. OUTLINES OF THE METHOD

Our method is to compile a Horn-clause program intended to be used for exhaustive search by means of backtracking or OR-parallelism into a GHC program or a deterministic Prolog program which returns the same (multi-)set of solutions in the form of a single list. Here, the word 'deterministic' means that a variable once instantiated is never unbound. Prolog programs in this deterministic subclass are interesting from the viewpoint of implementation, since a trail track need not be prepared to execute them correctly. Furthermore, the determinism in this sense has a similarity with the semantical restriction which GHC imposed to Horn clauses to enable all activities to be done in a single environment. This similarity is reflected by the fact that a transformed program can be interpreted both as a GHC program and as a Prolog program by the slight change between the '¦' (commit) operator and the '!' (cut) operator.

There are two possible views of this transformation technique. One is to regard this as compilation from a Horn-clause program (with no concept of sequentiality) to a guarded-Horn-clause program. By compiling OR-parallelism into AND-parallelism, we eliminate a multiple environment mechanism which is in general necessary for parallel search since each path of a search tree would create its own binding environment. The other view is to regard it as transformation of a Prolog program. This transformation serves as simplification in the sense that the predicate 'bagof' and the unbinding mechanism can be eliminated. Moreover, this transformation may remarkably improve the efficiency of a search program, as we will see later.

Our technique has another important meaning. By making search performed in a single environment, we get the possibility of extending an object program to introduce a mechanism for 'controlling' the search. That is, our technique may provide a staring point for realizing more intelligent search.

A transformed program, viewed as a GHC program, emulates the OR-parallel and AND-sequential execution of the original program. The original OR-parallelism is compiled into AND-parallelism as stated above, and the sequential execution of conjunctive goals is realized by passing a continuation around. The AND-parallelism of GHC we use is a simple one, since two conjunctive goals solving different paths of a search tree have no interaction except when solutions are collected.

A continuation is a data structure which represents remaining tasks to be done before we get a solution. The notion of continuation was effectively used also in a Concurrent Prolog compiler on top of Prolog [Ueda and Chikayama 85] to implement a goal queue. The difference is that our continuation is a stack, not a queue.

3. PREVIOUS RESEARCH

Implementation technique of exhaustive search in parallel logic programming languages can be found in [Hirakawa et al. 84] and [Clark and Gregory 85]. Their approach is to describe an interpreter of Horn-clause programs in Concurrent Prolog [Shapiro 83] or PARLOG [Clark and Gregory 84], but the following problems could be addressed to this approach:

(1) The interpreter approach loses efficiency.
(2) Multiple environment mechanism is implemented as a run-time creation of variants of terms.

where a variant is a term created by systematically replacing all the occurrences of the variables in an original term by new variables.

Problem (1) will not be serious, since it could be resolved by a partial evaluation technique. Alternatively, we could directly write a compiler which corresponds to those interpreters without much difficulty. On the other hand, Problem (2) seems serious.

The reason why we need multiple environments is that different sets of unifiers can be generated when we rewrite a goal in two or more ways by using different candidate clauses at the same time. Therefore, when we interpret an exhaustive search program, we make a necessary number of variants of the current set of goals and the partially determined solution prior to that simultaneous resolution. The above interpreters made some optimization to reduce the amount of variants to be created, but they do not avoid run-time creation of them.

However, run-time creation of variants is a time-sensitive operation. That is, the goal for creating a variant, say 'copy(T1,T2)', cannot be rewritten to the conjunction of two goals 'T1=T3, copy(T3,T2)' although it seems that both must have the same declarative meaning. This rewriting is the reverse operation of usual substitution, and GHC is designed so that its semantics is invariant with respect to this rewriting. Therefore, in the framework of GHC, the semantics of the above 'copy' predicate cannot be defined rationally. In the framework of sequential Prolog also, the predicate 'copy' should be considered extralogical, because it cannot be defined without the extralogical predicate 'var' which checks if its argument is currently an uninstantiated variable. The use of extralogical predicates should of course be discouraged, since it introduces semantical complexity and it hinders the description and the support of programming systems.

4. A SIMPLE EXAMPLE

To illustrate the difference between the previous method and ours, let us consider the example of decomposing a list into two using the usual 'append' predicate:

```
:- append(U,     V, [1,2,3]).
   append([],    Z, Z      ).                          (1)
   append([A|X], Y, [A|Z]  ) :- append(X, Y, Z).       (2)
```

From the head of Clause (2), we get a partial solution U=[1|X]. Then we get three instances for X, namely [], [2], and [2,3], by recursive calls. However, these three solutions cannot share the common prefix '[1|', as long as the value of a variable is represented by a reference pointer rather than by an association list, and this is why we have to make variants of the partial solution [1|X].

Our method, on the other hand, first rewrites Clause (2) as follows:

```
    append(X2     Y, [A|Z]  ) :- append(X, Y, Z), X2 = [A|X].    (2')
```

The predicate '=' unifies its two arguments. This can be defined by a single unit clause:

```
    X = X.
```

We assume that body goals are executed from left to right, following head unification. Then, while Clause (2) generates answer substitutions in a top-down manner, Clause (2') generates them in a bottom-up manner, that is, by combining ground terms only. The first output argument X2 remains uninstantiated until the first recursive goal, which may fork because of the two candidate clauses, succeeds. Therefore, we need not make variants of the partial solution upon that recursive call. Clause (2') is no more tail-recursive, so we must instead push the remaining task, the task of consing A with X to obtain X2, onto the stack representing a continuation. However, since the variable A must have a ground value, the information to be stacked can be represented as a ground term, and hence the continuation need not be copied even when the 'append' goal forks.

Now we are prepared for nondeterminism elimination. Figure 1 shows a GHC program which returns the result equivalent (up to the permutation of solutions) to the following DEC-10 Prolog goal:

```
  :- ..., bagof((X,Y), append(X,Y,Z), S), ...
```

The search corresponding to the two clauses of the original 'append' is performed by the conjunctive goals 'ap1' and 'ap2'. Their arguments are as follows:

(i) the input (third) argument of the original program,
(ii) the continuation,
(iii) the head of the difference list of solutions, and
(iv) its tail.

Since Clause (1) is a unit clause, the corresponding predicate 'ap1' activates the 'remaining tasks' by calling the predicate 'cont' for

```
:- ..., ap(Z, 'L0', S, []), ...

ap(Z, Cont, S0, S2) :- true ¦ ap1(Z, Cont, S0, S1), ap2(Z, Cont, S1, S2).

ap1(Z, Cont, S0, S1) :- true ¦ cont(Cont, [], Z, S0, S1).

ap2([A¦Z], Cont, S0, S1) :- true ¦ ap(Z, 'L1'(A,Cont), S0, S1).
ap2(Z,     _,    S0, S1) :- otherwise ¦ S0=S1.

cont('L1'(A,Cont), X, Y, S0, S1) :- true ¦ cont(Cont, [A¦X], Y, S0, S1).
cont('L0',         X, Y, S0, S1) :- true ¦ S0=[(X,Y)¦S1].
```

Fig.1 List Decomposition Program

continuation processing. At that time, two output results, [] and the input argument itself, are passed to the continuation processing goal. The predicate 'ap2' activates the first recursive goal with the information used for the second goal attached to the continuation in case the input argument has the form [A¦Z]. Otherwise, the unification of the input argument fails and the empty difference list is returned immediately.

The predicate 'cont' is for continuation processing. If the continuation has the form 'L1'(A,Cont), it pushes A in front of the output X and calls 'cont' to process the rest of the continuation, 'Cont'. If the continuation has the form 'L0', it inserts the two outputs it received into the difference list. The function symbols which construct the continuation can be regarded as indicating the locations of the original program: 'L0' indicates the end of the top-level goal, and 'L1' indicates the end of the recursive call of Clause (2'). Interestingly, the predicate 'cont' is very similar to an efficient (non-naive) list reversal program, and the continuation in this example is essentially a list which represents the first parts of all the solutions (each of which is a pair of lists) in a reversed form. Different solutions to be collected are created by different calls of 'cont' which reverse different substructures of the shared continuation.

The program in Figure 1 collects the solutions from 'ap1' and 'ap2' by the concatenation of difference lists, but this is not a fair way of collection. For example, if the first clause of some predicate produced infinite number of solutions, we could not see any solutions from the second clause. When we need a fair collection, we must collect solutions by using a 'merge' predicate implemented fair.

We can interpret Figure 1 also as a Prolog program, provided that the '¦' operators are replaced by the '!' operators, that the 'otherwise' goal in the second clause of 'ap2' is deleted, and that this clause is guaranteed to be the last clause of 'ap2'.

5. GENERAL TRANSFORMATION PROCEDURE

This chapter first presents the class of Horn-clause programs to which the technique as illustrated in Chapter 4 can be easily and mechanically applied, and then briefly show the transformation procedure. We use the permutation program (Figure 2) as an example.

First of all, we show the class of Horn-clause programs to which our transformation technique is applicable. A program in this class must have the following property when the body goals in each clause are executed from left to right, following head unification:

o The arguments of every goal appearing in a program can be classified into input arguments and output arguments. When the goal is called, its input arguments must have been instantiated to ground terms, and then the goal must instantiate its output arguments to ground terms when it succeeds.

Although the above property may look restrictive at a glance, most programs which do not use the notion of 'multiple writers' (see Chapter 6) or the notion of a difference list (which is an incomplete data structure) will enjoy this property. Programs which do use 'multiple writers' require pre-transformation as described in Chapter 6. On the other hand, programs which make use of difference lists could be handled by extending the above notion of input and output, as long as they allow static dataflow analysis. This conjecture is based on the observation that when we write a Prolog program which handles difference lists, we usually fully recognize how uninstantiated variables appear in the data structures.

One way to give input/output modes to a program would be to let the programmer declare them for every goal arguments appearing in the program. However, a more realistic way will be to let the programmer declare the mode of (the arguments of) the top-level goal only and to 'infer' the modes of other goals according to the following rules:

(a) Arguments which have been instantiated to ground terms upon call are regarded as input arguments (though they could be classified otherwise).
(b) All the other arguments are regarded as output arguments.

The mode inference and the check whether the program belongs to the above transformable class can be done in a simple static analysis. We must perform the following analysis for each clause and for each mode in which the predicate containing that clause may be called:

(1) Mark all the variables appearing in the input head arguments as 'ground'.
(2) While there is a body goal yet to be analyzed, do the following repeatedly:
   (i) Determine the mode of the next body goal according to the above inference rules (a) and (b). Here, those terms which are composed

```
perm([],    []).
perm([H|T], [A|P]) :- del([H|T], A, L), perm(L, P).

del([H|T], H, T).
del([H|T], L, [H|T2]) :- del(T, L, T2).
```

Fig.2 Permutation Program

```
Given Declaration:  perm(+, -).
('+': input, '-': output)


       +      -
perm( [],    []).
       +      -              +     -  -        +  -
perm([H|T], [A|P]) :- del([H|T], A, L), perm(L, P).
      +    -    -
del([H|T], H,   T).
      +    -    -               +   -  -
del([H|T], L, [H|T2]) :- del(T, L, T2).
```

Fig.3 Mode Analysis of the Permutation Program

    only of variables marked as 'ground' and function symbols, and only those, are regarded as ground terms.

  (ii) Then mark all the variables appearing in the output arguments of that goal as 'ground'.

(3) Check if the variables appearing the output head arguments are all marked as 'ground'. If the check succeeds, terminate the analysis of this clause with success; otherwise report failure.

Initially, only the modes of top-level goals are known; possible modes of other predicate calls are incrementally obtained during the above analysis. Therefore, the whole algorithm of mode analysis should be as follows. In the following, S denotes a set of 'moded' predicates. A moded predicate is a predicate with a mode in which it is called; different modes of some predicate correspond to different moded predicates.

(A) Let S be a set composed of the 'moded' predicates whose calls appear in the (declared) top-level goal. Mark those predicates as 'unanalyzed'.

(B) Repeatedly do the following until no 'unanalyzed' predicate remains in S. That is, take an 'unanalyzed' predicate from S, unmark it, and analyze all its clauses using the above algorithm, adding to S with the mark 'unanalyzed' all moded predicates whose calls are newly found during the execution of Step (2) .

Figure 3 shows the analyzed permutation program. It is easy to prove, by induction on the number of steps of resolution, that a successfully analyzed program instantiates the output arguments of each goal to ground terms upon successful termination, provided ground terms are given to the input arguments.

A successfully analyzed program is then transformed according to the following steps:

(1) If there is any predicate to be called in two or more different modes, give a unique predicate name for each mode.

(2) Rewrite each clause into the normal form.

(3) Transform each predicate in the program.

Step (1) removes multi-mode predicates. This transformation attaches the concept of mode to each 'predicate', not to each predicate 'call'.

Step (2) is made up of the following steps:

(2-a) For each clause other than unit clauses, replace output head arguments T1, ..., Tn by distinct new variables V1, .., Vn, and place the goals V1=T1, ..., Vn=Tn at the end of the clause.

(2-b) For each goal in the body of each clause, replace its output arguments T1, ..., Tn by distinct new variables V1, ..., Vn and place the goals V1=T1, ..., Vn=Tn immediately after that goal unless T1, ..., Tn are already distinct variables not appearing in the previous goals or the clause head.

The purpose of Step (2-b) is to simplify output arguments. It is clear that a program which has passed the mode analysis and then has been rewritten according to Steps (2-a) and (2-b) is still in the transformable class. Figure 4 shows the normal form of the permutation program.

Now we will show the outline of Step (3), the main part of our transformation method. Figure 5 shows the result applied to the permutation program of Figure 4. In the following, we indicate in braces what in the example of the permutation program are mentioned by each term appearing in the explanation.

(a) The arguments of a transformed predicate are made up of
  o the input arguments of the original predicate,
  o the continuation, and
  o the head and the tail of the difference list for returning solutions.

  Each transformed predicate is responsible for doing the task of the original predicate, followed by the task represented by the continuation.

(b) For a predicate {'perm'} of which at most one clause can be used for resolution of each goal, the transformed predicate is composed of the transformed clauses {<1>, <2>} of the original ones (See (i)). For a predicate {'del'} of which more than one clause may be applicable for resolution, we give a separate subpredicate name {'d1', 'd2'} to each transformed clause {<5>, <7>}, and let the transformed predicate {'d'} call all these subpredicates and collect solutions.

(c) The body of the clause {<1>, <5>} transformed from a unit clause calls a goal for continuation processing {'contp', 'contd'}. This goal is given as arguments the output values {[], (H, T)} returned by the original unit clause.

(d) The body of the clause {<2>, <7>} transformed form a non-unit clause calls the predicate {'d'} corresponding to the first body goal {'del'} of the original clause (See (e) and (j)).

```
perm([],    []).
perm([H|T], X) :- del([H|T], A, L), /*L1*/ perm(L, P), /*L2*/ X=[A|P].

del([H|T], H, T).
del([H|T], L, X) :- del(T, L, T2), /*L3*/ X=[H|T2].
```

Fig.4 Normal Form of the Permutation Program

```
<1>  p([],    Cont, S0,S1) :- true | contp(Cont, [], S0,S1).
<2>  p([H|T], Cont, S0,S1) :- true | d([H|T], 'L1'(Cont), S0,S1).
<3>  p(L,     _,    S0,S1) :- otherwise | S0=S1.

<4>  d(L, Cont, S0,S2) :- true | d1(L, Cont, S0,S1), d2(L, Cont, S1,S2).

<5>  d1([H|T], Cont, S0,S1) :- true | contd(Cont, H, T, S0,S1).
<6>  d1(L,     _,    S0,S1) :- otherwise | S0=S1.

<7>  d2([H|T], Cont, S0,S1) :- true | d(T, 'L3'(H,Cont), S0,S1).
<8>  d2(L,     _,    S0,S1) :- otherwise | S0=S1.

<9>  contp('L2'(A,Cont), P, S0,S1) :- true | contp(Cont, [A|P], S0,S1).
<10> contp('L0',         P, S0,S1) :- true | S0=[P|S1].

<11> contd('L3'(H,Cont), L, T2, S0,S1) :- true |
         contd(Cont, L, [H|T2], S0,S1).
<12> contd('L1'(Cont),   A,  L, S0,S1) :- true |
         p(L, 'L2'(A,Cont), S0, S1).
```

Fig.5 Transformed Permutation Program

(e) When calling a (transformed) predicate {e.g., 'd' in <7>} corresponding to the i-th body goal Gi {the recursive call of 'del'} of some clause, we push the label {'L3'} indicating the next goal Gi+1 together with the input data {H} used by the subsequent goals Gi+1, ..., Gn {X=[H¦T2]}. When calling a predicate {'p'} corresponding to the top-level goal {say 'perm(L,X)' where L is some ground term}, we give as the initial value of the continuation the label {'L0'} indicating the termination of refutation together with the data {none} for creating a term to be collected {X}.

(f) Predicates for continuation processing are composed of clauses {<9>, <10>, <11>, <12>} each corresponding to the label pushed in Step (e). These clauses are classified according to the predicates immediately before those labels and are given separate predicate names {'contp', 'contd'}.

(g) Each clause {e.g. <12>} of a predicate for continuation processing makes input data {L} for the goal {perm(L,P)} indicated by the received label {'L1'}, by using the information {none} stacked with the label and the output {A, L} of the last goal. Then it calls a predicate {'p'} which corresponds to the above goal (See (e) and (j)).

(h) The clause {<10>} to process the label {'L0'} indicating termination generates a term to be collected {P} from the output {P} of the top-level goal and the information {none} stacked with the label, and returns a difference list having that term as a sole element.

(i) For those transformed predicates {'p', 'd1', 'd2'} which may fail in the unification of the input arguments, backup clauses {<3>, <6>, <8>} are generated which return empty difference lists in case the unification fails.

(j) In spite of the above rules, no transformed predicates are generated for '=' and other system predicates, but they are processed immediately 'on the spot', followed by the next task {<9>, <11>}.

It is worth noting that in spite of our restriction, a transformed program can handle some non-ground data structure correctly. That is, the portion of an input data structure which is only passed around and never examined by unification need not be a ground term. For example, when we execute the following goal,

```
:- p([A,B,C], 'L0', S, []).
```

S will be correctly instantiated to a list of six permutations:

```
[[A,B,C],[A,C,B],[B,A,C],[B,C,A],[C,A,B],[C,B,A]]
```

6. ON THE CLASS OF TRANSFORMABLE PROGRAMS

For the technique described above to be effective from the practical point of view, the transformable class of Horn-clause programs defined in Chapter 5 must be large enough to express our problems naturally. The problem in this regard is that we often make use of the notion of 'multiple writers'. By 'multiple writers' we mean two or more goals sharing some data structure and trying to instantiate it cooperatively and/or competitively. In Prolog programming, such a data structure is usually represented directly by a Prolog term and it is operated by the direct use of Prolog unification; a typical example is the construction of the output data of a parser program.

However, this programming technique has problems from the viewpoint of the applicability of our transformation:

(1) It is generally impossible to statically analyze which part of the shared data structure is instantiated by which goal.
(2) The shared data structure may not be instantiated fully to a ground term.

Item (2) is considered a problem also from a semantical point of view. When extracting some information from the shared data structure generated by a search program, we have to use the extralogical predicate 'var' to see whether some portion of the data structure is left undetermined. One may argue that we need not use the predicate 'var' if we analyze the data structure after making it ground, that is, after instantiating its undetermined portions to some ground terms such as new constant symbols. He may further argue that making a term ground never calls for the predicate 'var' since we can accomplish this by trying to unify every subterm of it with a new constant. However, then, the search program which generates a non-ground result and the program to make it ground will be in the relationship of multiple writers, and the latter program should never start before the former program has finished because the latter program must have a lower priority with respect to instantiation of the shared data structure. This means we have to use the concept of sequentiality or priority between conjunctive goals, both of which are concepts outside the pure Horn-clause logic.

Anyway, we must make some pre-transformation to such a Horn-clause program in order to apply our transformation technique. That is, we must change the representation of the shared data structure to a ground-term representation--a list of binding information generated by each writer. Each writer must receive the current list of binding information and return a new one as a separate argument. When a writer is to add some binding information, it must check the consistency of the current and the new information to be added. This checking could be done by trying to construct the original representation from the scratch each time, but it could be done more efficiently by adopting appropriate data structure (possibly other than a list of bindings) to represent the binding information.

Comparing the original and the proposed implementation schemes of multiple writers from a practical point of view, the proposed scheme is apparently disadvantageous in the ease of programming. However, the difference does not lie in the specification of the abstract data but only in the ease of its implementation, which should not be so essential a problem in the sense that accumulation of programming techniques and program libraries should alleviate the difficulty.

Efficiency is another point on which comparison should be made. The original representation is advantageous for the consistency check in sequential Prolog. However, although the original representation is suitable for the execution using backtracking, it requires a multiple environment mechanism for OR-parallel execution instead of the backtracking mechanism, causing additional complexity and overhead. The proposed pre-transformation may make the consistency checking somewhat expensive, but will make parallel execution much easier since no multiple environment mechanism is necessary.

7. PERFORMANCE EVALUATION

Table 1 shows the performance of the original and the transformed programs. The programs measured are those described above, and an N-queens program with N being 5, 6, 7 and 8. The N-queens program we used is the one described in the transformable class shown in Chapter 5.

All programs were measured using DEC-10 Prolog on DEC2065. For each original program, the execution time of exhaustive search (by forced backtracking) without any collection of solutions was measured as well as

Table 1 Performance of Exhaustive Search Programs (in msec.)

PROGRAM	ORIGINAL (bagof)	ORIGINAL (Search Only)	TRANSFORMED	NUMBER OF SOLUTIONS
List Decompositon (50 elements)	836	4	27	51
Permutation Generation (5 elements)	354	34	57	120
5-Queens	45	20	28	10
6-Queens	90	75	106	4
7-Queens	441	325	446	40
8-Queens	1796	1484	1964	92

the execution time by the 'bagof' primitive. The 'setof' primitive was not considered because the sorting of solutions is inessential for us. Each program was measured after possible simplification which took advantage of the fact that Prolog checks candidate clauses sequentially.

As Table 1 shows, the proposed program transformation improved the efficiency of exhaustive search by 6 times for the permutation program and by more than 30 times for the list decomposition program ('append'). This remarkable speedup was brought about by specializing the task of collecting solutions to fit within the framework of the Horn-clause logic, while the 'bagof' primitive uses a extralogical feature similar to 'assert' which an optimizing compiler cannot help. A program such as N-queens, which has only a small number of solutions compared with its search space, cannot therefore expect remarkable speedup; the transformed N-queens program got slightly slower except for the case of 5-queens. After some manual optimization, however, the transformed 8-queens program surpassed the original 'bagof' version.

Another important point to note is that in the case of 8-queens, the transformed program was only by 25% slower than the original program which does not collect solutions and which makes use of the dedicated mechanism for search problems: automatic backtracking. This suggests that the transformed program could not be improved very much without changing the search algorithm.

8. CONCLUSIONS AND FUTURE WORKS

We have described a method of transforming a Horn-clause program for exhaustive search into a GHC program or a deterministic Prolog program. Although not stated above, the method which uses the concept of continuation can be applied also to the case where only one solution is required. Our method also provides the possibility of introducing a control mechanism of search, since all activities are made to be performed in a single environment.

We restricted the class of Horn-clause programs to which our method is applicable. However, this class is never trivial and it is expected that we should not have so much difficulty in writing a program within this class or its natural extension. Rather, we believe that it is practically important to show the class of Horn-clause programs which can be transformed without loss of efficiency and without any use of extralogical predicates.

The loss of performance by not using such dedicated mechanisms as automatic backtracking was small. Conversely, we found that our technique may greatly improve the efficiency of exhaustive search which has been done by using the 'bagof' primitive.

The proposed transformation eases parallel search in that it eliminates the need of multiple environments, but it never eliminates other problems on resource management. Resource management is still an important problem for realizing parallel search. Therefore, our results need not and should not be interpreted as reducing the significance of OR-parallel Prolog machines: Specialized hardware can always perform better for a special class of programs. While our purpose was primarily to extend the possibility of efficient search on a general-purpose parallel machine, we do expect also that our technique will be utilized for improving the efficiency of OR-parallel Prolog machines. Comparison of these two approaches should be an interesting research in the near future.

ACKNOWLEDGMENTS

The author would like to thank all the members of the ICOT Research Center who attended the discussions on the topic and made useful suggestions.

REFERENCES

[Clark and Gregory 84] Clark K.L. and Gregory S., PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology (1984).

[Clark and Gregory 85] Clark K.L. and Gregory S., Notes on the Implementation of PARLOG, J. of Logic Programming, Vol.2, No.1, pp.17-42 (1985).

[Hirakawa et al. 84] Hirakawa H., Chikayama T., and Furukawa K., Eager and Lazy Enumerations in Concurrent Prolog, Proc. 2nd Int. Logic Programming conference, Uppsala, pp.89-100 (1984).

[Naish 85] Naish, L., All Solutions Predicates in Prolog, Proc. 1985 Symposium on Logic Programming, IEEE Computer Society, pp.73-77 (1985).

[Shapiro 83] Shapiro, E.Y., A Subset of Concurrent Prolog and Its Interpreter, ICOT Tech. Report TR-003, Institute for New Generation Computer Technology (1983).

[Ueda 85] Ueda, K., Guarded Horn Clauses, ICOT Tech. Report TR-103, Institute for New Generation Computer Technology (1985). Also to appear in Lecture Notes in Computer Science, Springer-Verlag (1986).

[Ueda and Chikayama 85] Ueda K. and Chikayama, T., Concurrent Prolog Compiler on Top of Prolog, Proc. 1985 Symp. on Logic Programming, IEEE Computer Society, pp.119-126 (1985).