ICOT Technical Report TR-145

Making Exhaustive Search Programs Deterministic

Kazunori Ueda

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

November 1985*
Revised: February 1987**

Abstract. This paper presents a technique for compiling a Horn-clause program intended for exhaustive search into a GHC (Guarded Horn Clauses) program. The technique can be viewed also as a transformation technique for Prolog programs which compiles away 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 technique improved the efficiency of exhaustive search by 6 times for a permutation generator program. This compilation technique is important also in that it exploits the AND-parallelism of GHC for parallel search.

Keywords: Exhaustive search, Multiple binding environments, Compilation, Program transformation, Continuation, Mode analysis, Parallelism, Guarded Horn Clauses

1. Introduction

We often use Horn-clause logic, or more specifically the language Prolog, to obtain all solutions of some problem, that is, to obtain all answer substitutions for a goal to be solved. In this framework, however, it is difficult to *collect* the obtained solutions into a single environment for 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 system predicates for creating a list of all the solutions of a goal given as an argument; examples are '`setof`' and '`bagof`' of DEC-10 Prolog (Bowen et al. [1983]). Naish [1985] made a survey of all-solutions predicates in various Prolog systems. These system predicates, however, internally use some extralogical features to record the obtained

* A slightly modified version appeared in *Proc. Third Int. Conf. on Logic Programming*, Shapiro, E. (ed.), Lecture Notes in Computer Science 225, Springer-Verlag, Berlin Heidelberg, 1986, pp. 270–282.

** This version appeared in *New Generation Computing*, Vol. 5, No. 1, pp. 29–44.

solutions. So it should be an interesting question whether exhaustive search can be done without such primitives.

Another motivation is that we may sometimes wish to do exhaustive search in GHC (Ueda [1985][1986a][1986b]) 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.

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

2. Outlines of the Method

Our method is to compile a Horn-clause program intended for exhaustive search using 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. The word 'deterministic' means that all bindings given to variables are determinate and never undone. Prolog programs in this subclass are interesting from the viewpoint of implementation, since they never call for a trail stack. Furthermore, determinism in this sense has a similarity with the semantical restriction which GHC imposed to a proof procedure for Horn-clause logic in order to make all activities 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 '`|`' (commitment) 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 to a guarded-Horn-clause program. By compiling OR-parallelism into AND-parallelism, we eliminate a multiple environment mechanism for managing different binding environments created simultaneously by the paths of a search tree. The other view is to regard it as transformation of a Prolog program. This transformation serves as simplification in the sense that all-solutions predicates and the unbinding mechanism can be eliminated. Moreover, this transformation may remarkably improve the efficiency of a search program, as we will see in Section 7.

Our technique has another important meaning. By making search performed in a single environment, it becomes possible to introduce a mechanism for controlling the search. That is, our technique may provide a starting point for 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 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 a continuation was effectively used also in Concurrent Prolog and GHC compilers on top of Prolog (Ueda and Chikayama [1985]) to implement a goal queue. The difference is that here we use a stack instead of a queue.

3. Previous Research

Implementation techniques of exhaustive search in parallel logic programming languages were proposed by Hirakawa, Chikayama and Furukawa [1984] and by Clark and Gregory [1984b]. Their approach was to describe an interpreter of Horn-clause programs in Concurrent Prolog (Shapiro [1983]) or PARLOG (Clark and Gregory [1984a]), but the following could be addressed as problems:

(1) The interpreter approach loses efficiency.

(2) The multiple environment mechanism is implemented as a run-time creation of new variants of terms.

Here, a new variant of a term T is a term created by systematically replacing all the occurrences of the variables in T by fresh 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 the original interpreter without much difficulty (Ueda and Chikayama [1985]). On the other hand, Problem (2) seems serious.

The reason why we need multiple environments is that different unifiers can be generated when we rewrite a goal differently by using different program 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 the simultaneous rewriting. The above interpreters made some optimization to reduce the total size of variants to be created, but they did not completely avoid run-time creation of them.

However, run-time creation of variants is a time-sensitive operation. The semantics of GHC is designed so that it is not affected by *anti-substitution* (Ueda [1986]) which replaces an occurrence of some term T in the guard/body of a clause by a fresh variable X and adds the goal $X{=}T$ in that guard/body, respectively.

Anti-substitution serves as an acid test for *new* features among other implications of it. Applying anti-substitution to a variant creation goal, say '`copy(`T_1`,`T_2`)`', we get a conjunction 'T_1`=`T_3`, copy(`T_3`,`T_2`)`'. This rewriting clarifies that the first argument of '`copy`' specifying an original term may be instantiated with potential delay. Thus the predicate '`copy`' is incompatible with anti-substitution, and GHC cannot give any reasonable semantics to it. In 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 uninstantiated. The use of extralogical predicates should be discouraged, since it introduces semantical complexity and it hinders description of programming systems and support from them.

Carlsson [1984] presented implementation of exhaustive search in functional programming. His approach is similar to ours in that both use continuations; however, differences seem more important than the similarity. Firstly, our technique takes parallel execution into account. Secondly, our technique compiles away the environment problem while his approach requires variant creation when collecting solutions. Thirdly, our technique generates logic programs and can therefore be used as a transformation tool within logic programming.

Reddy [1984] also presented a technique for transforming logic programs into functional programs. Although the mode system we use in our technique is very similar to his, our technique is new in the treatment of multiple environments.

4. A Simple Example

To illustrate the difference between the previous methods and ours, let us consider the example of decomposing a list using '`append`':

```
:- append(U,     V,[1,2,3]).                          (4–1)
   append([],    Z,Z      ).                          (4–2)
   append([A|X],Y,[A|Z]   ) :- append(X,Y,Z).         (4–3)
```

From the head of Clause (4–3), 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 instead of 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 (4–3) as follows:

```
   append(X2,    Y,[A|Z]   ) :- append(X,Y,Z), X2=[A|X].     (4–4)
```

The predicate '`=`' unifies its two arguments. It 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 (4–3) generates answer substitutions in a top-down manner, Clause (4–4) generates them in a bottom-up manner by combining ground terms. 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 the recursive call. Clause (4–4) is not tail-recursive, so we must instead push the remaining task of constructing the value of `X2` onto the stack representing a continuation. However, since the variable `A` has a ground value, the remaining task to be stacked can be represented as a ground term and hence the continuation need not be copied when the goal `append(X,Y,Z)` forks.

Now we are prepared for the elimination of nondeterminism. Program 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), ... .                    (4–5)
```

Calling form: `:- ..., 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].
```

Program 1. List decomposition program.

Search corresponding to the two clauses of '`append`' is performed by the conjunctive goals '`ap1`' and '`ap2`' generated by '`ap`'. The arguments of these predicates are as follows:

(i) the input (i.e., the third) argument of the original program,

(ii) the continuation,

(iii) the head of the difference list of solutions, and

(iv) its tail.

The function symbols constructing the continuation can be regarded as indicating the locations of the original program: `'L0'` indicates the end of Clause (4–1) and `'L1'` indicates the end of the recursive call of Clause (4–4). The top-level goal initializes the continuation to `'L0'`.

Since Clause (4–2) is a unit clause, the corresponding predicate '`ap1`' activates the 'remaining tasks' by calling the predicate '`cont`' for continuation processing. At that time, two output results, `[]` and the input argument itself, are passed to the continuation processing goal. The predicate '`ap2`' checks if the input argument has the form `[A|Z]`, and if so, activates the first goal in the original clause with the information on the second goal attached to the continuation. The new continuation `'L1'(A,Cont)` indicates that the control must be returned to `'L1'` and that `A` is the value to be pushed in front of `X`. If the input argument is not of the form `[A|Z]`, the unification of the input argument fails and an empty difference list is returned immediately.

The predicate '`cont`' does 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 has received into the difference list. Interestingly, the predicate '`cont`' is very similar to an efficient (non-naive) list reversal program: The continuation in this example is essentially a list which represents the first part of each solution (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.

Program 1 collects the solutions from '`ap1`' and '`ap2`' by the concatenation of difference lists, but this is not a fair way of collection. 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 fair merging of lists.

We can interpret Program 1 also as a Prolog program, provided that the '`|`' operator is replaced by the '`!`' operator, that the '`otherwise`' goal in the second clause of '`ap2`' is deleted, and that the second clause of '`ap2`' is guaranteed to be the last clause of '`ap2`'.

5. General Transformation Procedure

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

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

```
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).
```

Program 2. Permutation program.

- The arguments of every goal appearing in the program can be classified into input arguments and output arguments. When some 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 Section 6) or the notion of a difference list (which can be an incomplete data structure) will enjoy this property. Programs which use multiple writers require pre-transformation as described in Section 6. 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 make the programmer declare them for every goal argument appearing in the program. However, a more realistic way will be to make the programmer declare the mode of (the arguments of) the top-level goals and to infer the modes of other goals according to the following rules:

- *(Moding Policy for a Single Goal)*

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

(b) All the other arguments are regarded as output.

The mode inference and the check whether the program is transformable 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:

- *(Mode Analysis of a Single Program Clause)*

(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:

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).
```

Program 3. Mode analysis of the permutation program.

(i) Determine the mode of the next body goal according to the above moding policy for a single goal. Here, those terms which are composed 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 in 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 goals are incrementally obtained during the above analysis. Therefore, the whole algorithm of the 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 a predicate correspond to different moded predicates.

- *(Mode Analysis of an Entire Program)*

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

(B) Repeatedly do the following until no *unanalyzed* predicate remains in S or failure is reported. 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 in Step (2).

(C) The program is transformable if and only if *no* failure is reported in Step (3).

Program 3 is 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

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

Program 4. Normal form of the permutation program.

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 as follows:

(2a) For each clause other than unit clauses, replace output head arguments $T_1, \ldots, T_n$ by distinct fresh variables $V_1, \ldots, V_n$, and place the goals V_1=$T_1, \ldots, V_n$=T_n at the end of the clause.

(2b) For each goal G in the body of each clause, replace its output arguments $T_1, \ldots, T_n$ by distinct fresh variables $V_1, \ldots, V_n$ and place the goals V_1=$T_1, \ldots, V_n$=T_n immediately after G unless $T_1, \ldots, T_n$ are distinct variables not appearing in the previous goals or the clause head.

(3) Transform each predicate in the program.

Step (1) removes multi-mode predicates. This transformation attaches the concept of a mode to each *predicate* as well as to each predicate *call*.

The purpose of Step (2b) is to simplify output arguments in a goal. It is clear that a program which has passed the mode analysis and then has been rewritten according to Steps (1) and (2) is still in the transformable class. Program 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. Program 5 shows the result applied to Program 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

- the input arguments of the original predicate,
- a continuation, and

```
 <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).
```

Program 5. Transformed permutation program.

- the head and the tail of a 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 reducing each goal, the transformed predicate consists 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 reduction, 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 a 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 a clause {<2>, <7>} transformed from a non-unit clause calls the predicate {'`d`'} corresponding to the first body goal {'`del`'} of the original clause (See (e) and (j)).

(e) When calling a (transformed) predicate {e.g., '`d`' in <7>} corresponding to the i-th body goal G_i {the recursive call of '`del`'} of some clause, we push the label {`'L3'`} indicating the next goal G_{i+1} together with the input data {`H`} used by the subsequent goals $G_{i+1}, \ldots, G_n$ {`X=L, Y=[H|T2]`}. If G_i is the last goal, then nothing is pushed but the current continuation is passed as it is. 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*} necessary for constructing 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 prepares input data {`L`} for the next 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`'} corresponding to the next goal (See (e), (j) and (k)).

(h) The clause {<10>} for processing 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 the 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 for returning empty difference lists when the unification fails.

(j) Unification goals generated by moving output head unification (Step (2a) of the transformation) are processed 'on the spot' in a transformed program, followed by the next task that must be a call to the continuation processing predicate {<9>, <11>}. The task is to feed the output value(s) {`[A|P]`, `(L,[H|T2])`} to the continuation processing goal.

(k) Unification goals generated by moving output arguments of a goal that may cause failure (Step (2b) of the transformation) are transformed as follows {*this never happens in the permutation program*}: We consider the following sequence of goals

$$\ldots,\ G,\ \texttt{/*La*/}\ V_1\texttt{=}T_1,\ \ldots,\ V_n\texttt{=}T_n,\ \texttt{/*Lb*/}\ \ldots$$

where V_1=$T_1, \ldots, V_n$=T_n are the goals moved from the goal G. Let $W_1, \ldots, W_l$ be the variables in $T_1, \ldots, T_n$ that must be made ground by previous goals, and $X_1, \ldots, X_m$ be the variables in $T_1, \ldots, T_n$ used by subsequent goals that will be made ground by V_1=$T_1, \ldots, V_n$=T_n. Then we define a predicate of the form

```
u(T1,...,Tn,W1,...,Wl,Cont,S0,S1) :- true |
    contx(Cont,X1,...,Xm,S0,S1).
u(V1,...,Vn,W1,...,Wl,Cont,S0,S1) :- otherwise | S0=S1.
```

where '`contx`' is a continuation processing goal for `'Lb'`. This predicate is called as

$$\texttt{u(}V_1,\ldots,V_n\texttt{,}W_1,\ldots,W_l\texttt{,'Lb'(}Y_1,\ldots,Y_p\texttt{,Cont),S0,S1)}$$

when the label `'La'` is recognized, where $Y_1, \ldots, Y_p$ are the input data used by the goals after `'Lb'`.

The above description does not consider system predicates. However, deterministic system predicates that deal with ground data can be handled without essential changes. Predicates for arithmetics and integer comparison fall within this class.

Some peephole optimization may apply to a transformed program. For example, if some predicate is called only once (textually) in an original program, the transformed predicate has only one 'return address'. The operations on a continuation could be optimized in such a case. General unfolding (or partial evaluation) technique may improve efficiency also.

Lastly, it is worth noting that in spite of our restriction, a transformed program can handle some non-ground data structure correctly. That is, the portions of an input data structure which are only passed around and never examined by unification need not be ground terms. 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]].
```

Non-ground data structures could be handled more gracefully by extending the mode system to a generic type system, but we used the mode system for the sake of simplicity.

6. On the Class of Transformable Programs

For the technique described above to be useful from a practical point of view, the transformable class of Horn-clause programs defined in Section 5 must be large enough to express many 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 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 in terms of applicability of our transformation:

(1) It is generally impossible to analyze statically 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.

Problem (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 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 Horn-clause logic.

Let us consider Problem (2) in terms of the declarative semantics of logic programs (Lloyd [1984]). Collection of solutions makes it possible to count the number of solutions (Warren [1982]). Then what must be regarded as the number of solutions when they contain variables? The number of maximally general answer substitutions? The number of elements of the minimal Herbrand model that are instances of the original goal? Both seem unsatisfactory. It seems that the number of solutions can be reasonably defined only by disallowing non-ground solutions, as long as we do not have a notion of types.

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 between the current and the new information to be added. This checking could be done by trying to construct the original representation from scratch each time, but it could be done more efficiently by adopting an appropriate data structure (possibly other than a list of bindings) for 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

Table 1. Performance of exhaustive search programs (in msec.).

Program	*Original* (‘`bagof`’)	*Original* (*search only*)	*Transformed*	*Number of Solutions*
List Decomposition (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 specification of the abstract data but only in the ease of its implementation, which should not be so essential a problem since accumulation of programming techniques and program libraries will alleviate the difficulty.

Efficiency is another point on which comparison should be made. Although the original representation is suitable for the execution using backtracking, it requires a multiple environment mechanism for OR-parallel execution, which may cause additional complexity and overhead (Ciepielewski and Haridi [1983]). The proposed pre-transformation may make the consistency checking somewhat expensive, but will make parallel execution much easier because no multiple environment mechanism is necessary.

Sometimes Problem (2) could be resolved more easily. Consider a parser of English sentences which checks the agreement of number. Neither a noun phrase nor a verb phrase may determine whether the subject is singular or plural, in which case a variable indicating the number may be left uninstantiated. In this case, however, it is easy to rewrite the program so that the analyzer of the noun phrase returns two possible values for that variable in OR-parallel instead of leaving it uninstantiated.

7. Performance Evaluation

Table 1 compares the performance of original and 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 was in the transformable class defined in Section 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* collection of solutions was measured as well as the execution time by the '`bagof`' primitive. The '`setof`' primitive was not considered because the order of solutions was 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 by specializing the task of collecting solutions to fit within the framework of Horn-clause logic, while the '`bagof`' primitive uses a extralogical feature similar to '`assert`' (Bowen et al. [1983]) which an optimizing compiler cannot help. A program such as *N-queens*, which has 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 5-*queens*. After manual optimization, however, the transformed 8-*queens* program surpassed the original '`bagof`' version.

Another 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, namely automatic backtracking. This suggests that the transformed program could not be improved very much without changing the search algorithm.

8. Summary and Future Works

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

We limited the class of Horn-clause programs to which our method is applicable. However, this class is never trivial and it should not be so difficult to write a program within this class or its natural extension. Rather, we believe that it is important from a practical point of view to show the class of Horn-clause programs which can be transformed without loss of efficiency and without resort to extralogical predicates. Programs to which our method is essentially inapplicable seem to require semantical considerations before trying to extend our method.

The loss of performance by not using such dedicated mechanisms as automatic backtracking was small. We found that our technique may even improve the efficiency of exhaustive search that 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 in 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 primary purpose was to examine the possibility of efficient search on a general-purpose parallel machine, our technique could be utilized also 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 is indebted to Hisao Tamaki for pointing out an overlooked case in the transformation procedure of the original version of this paper.

References

Bowen, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D. [1983] *DECsystem-10 Prolog User's Manual*. Dept. of Artificial Intelligence, Univ. of Edinburgh.

Carlsson, M. [1984] On Implementing Prolog in Functional Programming. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 154–159.

Ciepielewski, A. and Haridi, S. [1983] A Formal Model for OR-Parallel Execution of Logic Programs. In *Proc. IFIP '83*, Mason, R. E. A. (ed.), Elsevier Science Publishers B. V., Amsterdam, pp. 299–305.

Clark, K. L. and Gregory, S. [1984a] PARLOG: Parallel Programming in Logic. Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London. Also in *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No.1 (1986), pp. 1–49.

Clark, K. L. and Gregory, S. [1984b] Notes on the Implementation of PARLOG. Research Report DOC 84/16, Dept. of Computing, Imperial College of Science and Technology, London, 1984. Also in *J. of Logic Programming*, Vol. 2, No. 1 (1985), pp. 17–42.

Hirakawa, H., Chikayama, T. and Furukawa, K. [1984] Eager and Lazy Enumerations in Concurrent Prolog. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 89–100.

Lloyd, J. W. [1984] *Foundations of Logic Programming*. Springer-Verlag, Berlin Heidelberg New York Tokyo.

Naish, L. [1985] All Solutions Predicates in Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE Computer Society, pp. 73–77.

Reddy, U. S. [1984] Transformation of Logic Programs into Functional Programs. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 187–196.

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

Ueda, K. [1985] Guarded Horn Clauses. ICOT Tech. Report TR-103, Institute for New Generation Computer Technology, Tokyo. Also in *Proc. Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg (1986), pp. 168–179.

Ueda, K. [1986a] *Guarded Horn Clauses*. Doctoral Thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo.

Ueda, K. [1986b] Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, ICOT Tech. Report TR-208, Institute for New Generation Computer Technology, Tokyo.

Ueda, K. and Chikayama, T. [1985] Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE Computer Society, pp. 119–126.

Warren, D. H. D. [1982] Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, Hayes, J. E., Mitchie, D. and Pao, Y. -H. (eds.), Ellis Horwood, Chichester, England, pp. 441–454.