Making Exhaustive Search Programs Deterministic, Part II

Kazunori Ueda

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

Abstract. This paper complements the previous paper "Making Exhaustive Search Programs Deterministic" which showed a systematic method for compiling a Horn-clause program for exhaustive search into a GHC program or a Prolog program with no backtracking. This time we present a systematic method for deriving a deterministic logic program that simulates coroutining execution of a generate-and-test logic program. The class of compilable programs is sufficiently general, and compiled programs proved to be efficient. The method can also be viewed as suggesting a method of compiling a naive logic program into (very) low-level languages.

1. INTRODUCTION

This paper complements the previous paper [14] by the author and extends the method described in it. The previous paper showed a systematic method for compiling a Horn-clause program into a GHC [13][15][16] program or a deterministic Prolog program that returns a list of all the solutions of the original program. The method compiled away the primitives for collecting solutions such as `bagof`, which had usually been considered as important extension to Prolog. The compilation generally retained the efficiency of original programs on the same compiler-based Prolog system, and a compiled program even outperformed the original one when the number of solutions was large relative to the search space. We showed the class of Horn-clause programs amenable to the method and argued how it is restrictive. These results could be summarized as follows:

(1) A prospect was obtained for efficient (though not optimal) parallel execution of search problems using a general-purpose parallel language like GHC as a base language.

- *Original program*

```
append([],    Z,Z    ).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

- *Calling form and mode information:*

```
                   - - +
bagof((X,Y),append(X,Y,Z),S)
```

  ('+': input (ground upon call); '-': output (ground upon success))

- *Mode analysis using the above mode information*

```
         -    -   +
append( [],  Z,  Z  ).
         -   -   +               - - +
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

- *Pre-transformation (moving output unification)*

```
append([],    Z,Z    ).
append(X2,    Y,[A|Z]) :- append(X,Y,Z), /*L1*/ X2=[A|X].
```

- *Compiled program returning the same result*

```
Calling form:  ..., ap(Z,'L0',S,[]), /*L0*/...
                     % Arg1: input arg; Arg2: continuation;
                     % Arg3,4: d-list (head and tail) of output pairs

ap(Z,Cont,S0,S2) :- true | ap1(Z,Cont,S0,S1), ap2(Z,Cont,S1,S2).
                     % ap1 for unit clause and ap2 for recursive clause

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. Compilation of a list decomposition program.

(2) Horn-clause logic can now be regarded as a user language of GHC for search problems.

Exploiting parallelism in an obtained GHC program is quite easy, because different search paths are examined by independent AND-parallel goals without communication. This means that an obtained program could be processed using restricted AND-parallelism [5] also.

A deterministic program obtained by the previous method simulates 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. Solutions are collected by concatenating (possibly empty) difference lists of solutions returned by the paths of a proof tree.

The crux of the technique is to move output unification (i.e., unification for constructing result values) from a clause head to the end of the clause. In general, OR-parallel search requires a multiple environment mechanism, because incompatible bindings may be generated when a goal is rewritten differently by different program clauses at the same time. However, if output unification is moved, head unification will not return partial results any more, but results will be constructed in a bottom-up manner. Thus a multiple environment mechanism is made inessential. Program 1 shows compilation of an `append` program to be used for decomposing a list into two. We use mode declaration and mode inference for determining whether each argument of a goal is for providing an input value or for receiving a result value.

However, assuming AND-sequentiality often causes incompatibility between clarity and efficiency of a program, as many people have pointed out. For instance, Program 2 for solving the 8-queens problem is said to be naive and clear, because it well separates the generator (a permutation program) and the tester for the generate-and-test-type problem. This program, however, proves to be very inefficient under AND-sequential execution (see Section 5), because the tester starts after the generator has placed all the queens. The situation is even worse with a program for finding acyclic paths on a graph shown in Section 4.2; the generator generates an infinite number of candidate solutions, so although the number of final solutions is finite, the program would not terminate. For this reason, we usually fuse a generator and a tester in Prolog programming and write a program like Program 3 . Program 3 is not so complex, but it obscures the fact that permutation generation is a subproblem of the 8-queens problem.

2. THE PROBLEM

Our goal is to derive systematically (i.e., in a mechanizable manner) an efficient deterministic logic program that returns a list of all the solutions of the original generate-and-test program. We must compile coroutining to achieve this; when the goal `perm([1,...,8],X)` in Program 2 instantiates `X` to `[1,2|X2]`, `nocheck(X)` must be invoked to detect that `[1,2|X2]` cannot be a solution for any instance of `X2`. At the same time, we must compile away the `bagof` primitive that is outside Horn-clause logic.

```
:- bagof(X, eightqueens(X), B).

eightqueens(X) :- perm([1,2,3,4,5,6,7,8],X) // nocheck(X).

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

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

nocheck([H|T]) :- qsafe(H,T,1), nocheck(T).
nocheck( []  ).

qsafe(U,[H|T],N) :- H+N=\=U, H-N=\=U, M is N+1, qsafe(U,T,M).
qsafe(_, [],  _).
```

Program 2. Naive 8-queens program.

```
eightqueens(X) :-  eightq([1,2,3,4,5,6,7,8],[],X).

eightq([H|T],R,P) :-
   del([H|T],A,L), qsafe(A,R,1), eightq(L,[A|R],P).
eightq( [],  R,P) :- rev(R,[],P).

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

qsafe(U,[H|T],N) :- H+N=\=U, H-N=\=U, M is N+1, qsafe(U,T,M).
qsafe(_, [],  _).

rev([A|X],Y,Z) :- rev(X,[A|Y],Z).
rev( [],  Y,Y).
```

Program 3. 8-queens program optimized for sequential execution.

One possible method is to derive an efficient Prolog program like Program 3 first and then to derive an exhaustive search program from it using the previous method. Methods for the first step have been proposed by many researchers [1][3][7][8][11]. Of these, Gallagher [7] and Bruynooghe et al. [1] use meta-level predicates that describe deduction steps of the goals given as their arguments. Obtained programs, however, are not compilable by the previous method. They preserve shared variables between generators and testers instantiated in a top-down manner, which cannot be analyzed by the mode analysis.

Clark [3], Gregory [8], and Seki and Furukawa [11] use methods based on unfold/fold transformation [12]. However, it seems that only the method of [11] can derive Program 3 from Program 2; the derivation requires a new rule in addition to those in [12]. Moreover, it has not been made clear how these methods can be mechanized for what class of Horn-clause programs.

We chose to compile a generate-and-test logic program directly to a deterministic program by extending the analysis and compilation method of the previous paper. Thus our method is not just a direct sum of the known techniques stated above. One reason for taking this approach is that it seemed to be more amenable to mechanization. Another reason is that by using the mode system, the sufficient conditions for the compilation can be explicitly defined in terms of the data-flow of a program. The main problems to be solved are how to compile away the shared variable between a generator and a tester and how to extend the mode system for the analysis of coroutining.

3. METHOD AND EXAMPLE

This section exemplifies how we extend the previous method to handle coroutining, using Program 2.

First of all, we introduce a new syntax for specifying coroutining. We write `g(...X...) // t(...X...)` for the pair of a generator and a tester to be run concurrently, where `X` is the shared variable instantiated by `g` to a list of ground terms in a top-down manner and examined by `t` for its adequacy. This pair of goals can be regarded as a goal `g(...X...)` constrained by `t(...X...)`, and can appear in a sequence of goals executed sequentially from left to right. All the arguments except `X` must be ground, and `X` undefined, when the pair of goals starts.

Our goal is to detect failure as soon as possible by executing `t` incrementally each time `g` instantiates `X`. Let us trace one of the possible execution paths of Program 2.

When `perm([1,...,8],X)` starts and the goal `del` in the first clause of `perm` succeeds, the first element of `X` is instantiated to some integer between `1` and `8`, inclusive. The goal `nocheck(X)` is invoked at this point. In resolution principle, we could of course execute `nocheck(X)` *before* `X` is instantiated, but we have suspended it because we chose to use it for checking the given value of `X`.

Assume `X` is instantiated to `[1|X1]`. Then from `nocheck(X)` we can derive `qsafe(1,X1,1)` and `nocheck(X1)` using the first clause. However, the derived goals must be suspended until `X1` is instantiated, because otherwise `X1` would be instantiated by the tester. So we suspend the execution of nocheck(X) and resume perm([1,...,8], X). The recursive call to perm invokes X1 to, say, [5|X2]. Now from the suspended goal qsafe(1,X1,1) we can derive 5+1=\=1, 5-1=\=1, M is 1+1 and qsafe(1,X2,M), of which the first three goals can be executed immediately in success and qsafe(1,X2,2) remains. From another suspended goal nocheck(X1), we can derive qsafe(5,X2,1) and nocheck(X2), both of which must be suspended.

This procedure will finally result in one of the following situations:

(1) Some call to '=\=' derived from qsafe fails.

(2) X is instantiated to a complete list.

The first case means that the execution path we have been tracing results in failure. In the second case, X has been instantiated to, say, [1,5,8,6,3,7,2,4|X8], and then X8 is instantiated to [] by the second clause of perm. In this case, the nine goals

qsafe(1,X8,8), ..., qsafe(4,X8,1) and nocheck(X8)

have been derived from nocheck(X), and they all succeed when X8 is bound to []. Now both perm([1,...,8],X) and nocheck(X) have succeeded and we have found a solution X=[1,5,8,6,3,7,2,4].

The important point in the above procedure is that the following can be known by static analysis:

(1) The *next element* of X is determined when the call to del in the first clause of perm has succeeded; and the list X is closed with [] when resolution with the second clause of perm has succeeded.

(2) When the *next element* of X is determined, we can derive two goals, a qsafe and a new nocheck, from a suspended nocheck. Moreover, if there exists a suspended qsafe, four goals can be derived from it, and they can be executed immediately except for the recursive qsafe. When X is terminated by [], all suspended goals succeed.

Furthermore, the above properties do not depend on the specific value [1,...,8] of the first argument of perm, but *only on the fact that it is ground.* This suggests that we can statically analyze coroutining as mode analysis. The technique will be described later in Section 4; before that we will derive a deterministic 8-queens program first.

We put labels in the original program and move output unification (Program 4). These labels are used as the constructors of a continuation. Since the predicate del does not directly manipulate the shared

```
      eightqueens(X) :-
        perm([1,2,3,4,5,6,7,8],X) // nocheck(X)
L1:     (construct X).

      perm([H|T],X) :-
        del([H|T],A,L),
L2:     X=[A|P], (send A to nocheck(X) and invoke it)
L3:     perm(L,P).

      perm([],  []) :-
        (X is terminated, so nocheck(X) succeeds).

      del([H|T],H,T).

      del([H|T],X,Y) :-
        del(T,A,T2),
L4:     X=A, Y=[H|T2].
```

Program 4. Labeled 8-queens program *(*`nocheck` *and* `qsafe` *are omitted; see Program 2 for them).*

variable `X` of the generator-tester pair, output unification is moved to the end of the clause so that the output values are constructed in an bottom-up manner (see Section 1). On the other hand, the predicate `perm` determines the value of the shared variable `X`. To realize coroutining, we must ship out a new element `A` at `L2` (i.e., as soon as it is determined by the goal `del([H|T],A,L)`), where we invoke `nocheck(X)` (or goals derived from it) and do possible derivations. If the new element `A` is appropriate, the derivation results in a set of suspended goals and the tasks following `L3` is executed. Otherwise, the derivation fails and `L3` is not reached.

The second clause of `perm` is for closing the shared variable `X`. When it is selected, all the suspended goals that have been derived from `nocheck(X)` are just discarded, because the following are statically known:

(1) When `X` is closed, possible remaining goals derived from `nocheck(X)` are either of the form `qsafe(...Xn...)` or of the form `nocheck(Xn)`, where `Xn` is a sublist of `X` just being bound to `[]`.

(2) All these goals succeed when `Xn` is instantiated to `[]`.

`L1` is the sole return point from the second clause of `perm`. Reaching `L1` means that a solution has been obtained, but since we accumulate the elements of `X` in reverse order, we reverse them again at `L1` to construct the final value of `X`. The reason for accumulating the elements of X in reverse order is that we want to represent the intermediate values of X always as ground terms so that they can be shared when the current search paths split in future.

Program 5 shows a deterministic program derived from Program 4. The control flow of the generator perm described above is implemented in exactly the same continuation management method as the previous one: The predicates *e*, *p* and *d* correspond to eightqueens, perm and del, respectively; *d1* and *d2* correspond to the first and the second clauses of the non-deterministic predicate del; *cont0*, *cont1*, *cont24* and *cont3* are for continuation management (we use a typewriter font for entities of original programs and an *italic* font for compiled programs). The only difference is that the information on the tester nocheck and a partial solution of X are carried around in new forms by *p* (the second and the third arguments, respectively) and by the goals called by *p*. The last two arguments of each predicate represent a difference list of obtained solutions.

On the other hand, the control flow of the tester is implemented in a different manner. Suspended goals derived from nocheck(X) are put in a dedicated continuation (which we call *a subcontinuation* henceforth) managed as a local datum of perm[1]. In Program 5, *Cont* represents the main continuation and *Contn* represents the subcontinuation. The initial value *n* of the subcontinuation, which means there is one goal nocheck to be solved, is set in the sole clause of *e*. The tasks represented by *Contn* are processed by *nresume*. The predicate *nresume* is invoked when the second clause of *cont24* recognizes that the control has reached L2, does possible derivations from the current *Contn* using the value *H* of the next element of the shared variable, and creates a new subcontinuation.

Contn examined by *nresume* has either the form *n* or the form *q(U,N,Contn′)*. The constant *n* represents the goal nocheck(Xn), and the term *q(U,N,Contn′)* represents the goal qsafe(U,Xn,N) and the goals represented by *Contn′*, where Xn means the sublist of X which was instantiated to the form [H|Xn'] just before the current invocation of *nresume*. A subcontinuation is composed only of ground input arguments for qsafe and the constructors *n* and *q*; the shared variable X has been compiled away. The values of the elements of X are given one at a time as the fifth argument of *nresume*.

The first clause of *nresume* executes three of the four goals derived from qsafe(U,[H|Xn'],N) immediately, and stacks the remaining recursive call to the third argument *ContnR* (*R* stands for *re-*

Calling form: `:- e('L0',B,[]).`

```
e(Cont,S0,S1) :- true |
   p([1,2,3,4,5,6,7,8],n,[],'L1'(Cont),S0,S1).

p([],   Contn,SR,Cont,S0,S1) :- true | cont1(Cont,SR,S0,S1).
p([H|T],Contn,SR,Cont,S0,S1) :- true |
   d([H|T],'L2'(Contn,SR,Cont),S0,S1).
p(L,    _,    _, _,   S0,S1) :- otherwise | S0=S1.

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

d1([H|T],Cont,S0,S1) :- true | cont24(Cont,H,T,S0,S1).
d1(L,    _,   S0,S1) :- otherwise | S0=S1.

d2([H|T],Cont,S0,S1) :- true | d(T,'L4'(H,Cont),S0,S1).
d2(L,    _,   S0,S1) :- otherwise | S0=S1.

nresume(q(U,N,Contn),SR,ContnR,Cont,H,S0,S1) :-
   H+N=\=U, H-N=\=U |
   M is N+1, nresume(Contn,SR,q(U,M,ContnR),Cont,H,S0,S1).
nresume(n,           SR,ContnR,Cont,H,S0,S1) :- true |
   rev2(ContnR,q(H,1,n),NewContn),
   cont3(Cont,NewContn,[H|SR],S0,S1).
nresume(_,           _, _,     _,   _,S0,S1) :- otherwise |
   S0=S1.

cont0('L0',S,S0,S1) :- true | S0=[S|S1].

cont1('L1'(Cont),SR,S0,S1) :- true |
   rev(SR,[],S), cont0(Cont,S,S0,S1).

cont24('L4'(H,Cont),       A,T2,S0,S1) :- true |
   cont24(Cont,A,[H|T2],S0,S1).
cont24('L2'(Contn,SR,Cont),A,T2,S0,S1) :- true |
   nresume(Contn,SR,n,'L3'(T2,Cont),A,S0,S1).

cont3('L3'(T2,Cont),Contn,SR,S0,S1) :- true |
   p(T2,Contn,SR,Cont,S0,S1).

rev([A|X],Y,Z) :- true | rev(X,[A|Y],Z).
rev([],   Y,Z) :- true | Y=Z.

rev2(q(A,B,X),Y,Z) :- true | rev2(X,q(A,B,Y),Z).
rev2(n,       Y,Z) :- true | Y=Z.
```

Program 5. Compiled 8-queens program.

versed). The inequalities are executed in a guard since they may fail; if they should fail, the difference list for collecting solutions is short-circuited by the third clause. The second clause recognizes the goal `nocheck([H|Xn'])`, creates a new subcontinuation `NewContn` from

(i) the goals `qsafe(H,Xn',1)` and `nocheck(Xn')` derived from it (which are represented as *q(H,1,n)*) and

(ii) the goals stacked in *ContnR* in reverse order

using *rev2*, and calls *cont3* to process the tasks following `L3`.

The elements of the shared variable `X` are stacked in reverse order (as stated above) in *SR*. *SR* is initialized to *[]* in the sole clause of *e*, and each new element is stacked at the second clause of *nresume* when it passes the control to `L3`. *SR* is reversed by *cont1* as the task following `L1`. The predicate *cont1* is called from the first clause of *p*. This clause discards *Contn*, because the corresponding original clause (the first clause of `perm`) closes the shared variable `X`, which causes all the suspended goals derived from `nocheck(X)` to succeed as we stated above.

4. GENERAL COMPILATION PROCEDURE

The compilation procedure comprises the following:

(1) mode analysis,

(2) pre-transformation to a normal form, and

(3) compilation to a deterministic program.

The primary issue for realizing automatic compilation is that the sufficient conditions for being compilable are given in a statically decidable manner. This decision is done as part of the mode analysis described in Section 4.1. Section 4 describes the mode analysis in detail and the compilation rather briefly.

4.1 Mode Analysis

The purpose of the mode analysis is to obtain information for the compilation by statically analyzing data-flow that will happen at run time. Thus it is a kind of abstract interpretation [10]. While the previous paper used two modes for the analysis, we now use the following four modes and assign one of them to each argument of the body goals of a clause:

‘+’ (*input*) The preceding computation (or the top-level goal clause) guarantees that this argument is instantiated to a ground term when the goal starts.

‘-’ (*output*) The goal guarantees to instantiate this argument to a ground term upon success.

‘?’ (*stream-input*) The goal having the corresponding stream-output argument (say G) guarantees to instantiate this stream-input argument to a list of ground terms. The goal G further guarantees that when this argument is instantiated to the form `[`H`|`T`]`, H has been instantiated to a ground term and that the properties of a stream-input argument can again be assumed for T.

‘^’ (*stream-output*) The goal guarantees to instantiate this argument to a list of ground terms. Moreover, the goal guarantees that when it instantiates this argument to the form `[`H`|`T`]`, H has been instantiated to a ground term and T again has the properties of a stream-output argument.

The modes stream-output and stream-input are given to the shared variable instantiated by a generator-tester pair. If output unification for the stream-output argument of a generator is specified in a clause head (as in the first clause of `perm` in Program 2), the generator may fail to guarantee the requirements for a stream-output argument, that is, it may generate an uninstantiated element. However, the pre-transformation phase (see Section 4.3) tries to guarantee them by moving the output unification to an appropriate place in the clause.

The mode analysis analyzes all the predicates that may be called directly or indirectly from a top-level goal clause, using the mode declaration on that clause. It clarifies the data-flow of the whole program by mode assignment. If the analysis succeeds, the mode assignment is guaranteed to be correct, and the program is amenable to compilation. If it fails, the program cannot be compiled in the current setting. The mode analysis of each clause is done as follows:

(1) Mark all the variables appearing in the input head arguments as *ground*.

(2) If the clause has a stream-input head argument T, do the following:

(2a) If T is `[]`, do nothing.

(2b) If T is an unmarked variable, mark it as *stream*.

(2c) If T is of the form `[`H`|`T`]`, mark all the variables in H as *ground*, and process T according to (2a) to (2d).

(2d) Otherwise, make the analysis fail.

(3) Assign modes to the body goals from left to right, where the mode of each goal is assumed as follows:

(3a) An argument consisting only of function/constant symbols and variables marked as *ground* is assumed to be input.

(3b) An argument that is a single variable marked as *stream* is assumed to be stream-input. If there is a non-variable argument containing variables marked as *stream*, make the analysis fail.

(3c) If the goal in question is a generator-tester pair and the following conditions are satisfied,

(i) The generator has just one argument not assumed to be input, which is an unmarked single variable.

(ii) That variable appears also as a single-variable argument of the tester, and it is the only argument of the tester not assumed to be input.

then assume the argument of the generator to be stream-output and the argument of the tester to be stream-input, and mark the shared variable as *ground*. If the generator-tester pair does not satisfy the above conditions, make the analysis fail.

(3d) Assume all the other arguments (i.e., arguments containing unmarked variables) to be output, and mark all the variables in them as *ground*.

(4) If the clause in question has a stream-input head argument, confirm the following. If not confirmed, make the analysis fail.

(4a) At most one argument is assumed to be stream-input for each body goal.

(4b) All the other arguments of a goal having a stream-input argument are assumed to be input.

(5) Check if all the variables appearing in the output head arguments have been marked as *ground*, and make the analysis fail if the check fails.

(6) Check if a stream-output head argument, if any, is one of the following:

(6a) `[]`

(6b) a variable marked as *ground*

Declared Mode: `eightqueens(-).`

```
            -                  +         ^             ?
eightqueens(X) :- perm([1,2,3,4,5,6,7,8],X) // nocheck(X).
       +     ^             +   - -        + ^
perm([H|T],[A|P]) :- del([H|T],A,L), perm(L,P).
       +     ^
perm( [],    [] ).
      +   -   -
del([H|T],H,  T   ).
      +   -   -             + - -
del([H|T],A,[H|T2]) :- del(T,A,T2).
          ?             + ? +           ?
nocheck([H|T]) :- qsafe(H,T,1), nocheck(T).
          ?
nocheck( []  ).
      +   ?   +     + +   +  + +   +  -    +          + ? +
qsafe(U,[H|T],N) :- H+N=\=U, H-N=\=U, M is N+1, qsafe(U,T,M).
      +   ?   +
qsafe(U, [],  V).
```

Program 6. Mode analysis of the 8-queens program.

(6c) a term of the form `[`H`|`T`]`, where all the variables in H are marked as *ground* and T is again one of (6a) to (6c).

If the check fails, make the analysis fail. Moreover, if there exists a variable classified as (6b) (possibly after recursive checking using (6c)), check if it appears once and only once in the body as a single-variable argument of some goal. If the check succeeds, change the mode of that argument of the body goal to stream-output; otherwise make the analysis fail.

Program 6 shows an analyzed 8-queens program. If the top-level goal clause contains a generator-tester pair, its mode declaration must be compatible with the conditions (i) and (ii) in (3c) and the assumptions made there.

4.2 Purpose of the Restriction and Its Generality

The restrictions imposed by the above mode analysis are for guaranteeing the following properties of a program:

(1) A generator and the corresponding tester communicate using a single shared variable, which is the only output from them.

(2) The generator determines the value of the shared variable in a top-down manner and incrementally. Moreover, each element of

the shared variable is determined by only one goal; that is, we exclude a generator like

```
gen(...X...) :- gen1(...X...), gen2(...X...).
                                    % X is the shared variable.
```

to avoid conflict within the generator.

(3) The tester checks the value of the shared variable in a top-down manner. Two or more goals can check it independently (e.g. the first clause of `nocheck` in Program 6). A goal derived from the tester taking a sublist of the shared variable, as well as the tester itself, concentrates on checking and generates no output.

The mode system is somewhat complex, but it is inevitable for sophisticated control over the execution of goals. Programmers must keep the above properties in mind, but need not remember the detail of the analysis. The "top-down" restriction on a shared variable should be reasonable, because if the generation and the check are not done incrementally, the original program will not be made efficient by coroutining. In such a case, the analyzer would generate an error message like *"The shared variable between* `perm` *and* `nocheck` *is not instantiated incrementally, so coroutining is useless."*

It must be examined whether the class of logic programs that pass the above mode analysis is sufficiently general. This is hard to answer since we do not have a good stock of logic programs written for coroutined execution. However, the class of search problems that generate sequences (i.e., lists) of numbers, actions, etc. is considered quite general. For instance, many textbook examples such as the missionaries-and-cannibals problem, the path-finding problem on graphs, and the block-moving problem share the property of generating sequences of actions. They can be elegantly programmed as generate-and-test programs satisfying the above properties. Program 7 is an analyzed path-finding program in [6] originally written using `freeze` of Prolog-II instead of coroutining.

The mode system in this paper could be extended further (probably at the expense of simplicity); for example, we could allow a generator to have output arguments as well as a stream-output argument.

Our method does not apply to some textbook examples for which coroutining is effective. The cryptarithmetic problem is an example; Program 8 is a program to solve "SEND + MORE = MONEY" [6, p. 150] rewritten in our notation. The data-flow of this program is

```
       +     +    -
path(Start,Goal,Path) :-
            +     +    ^                  ?
    path1(Start,Goal,Path) // good_list(Path).
      + +    ^
path1(X,X, [X]    ).
      + +   ^                    + -         + +   ^
path1(X,Y,[X|Path]) :- neighbor(X,Z), path1(Z,Y,Path).

         + -        + -
neighbor(X,Y) :- nb(X,Y).
         + -        - +
neighbor(X,Y) :- nb(Y,X).

   + -       + -       + -       + -       + -       + -
   - +       - +       - +       - +       - +       - +
nb(a,b).  nb(a,c).  nb(b,d).  nb(b,e).  nb(c,f).  nb(d,g).
   + -       + -       + -       + -       + -
   - +       - +       - +       - +       - +
nb(e,g).  nb(e,h).  nb(f,j).  nb(h,i).  nb(h,j).

            ?
good_list( [] ).
            ?              + ?             ?
good_list([X|L]) :- out_of(X,L), good_list(L).

       +   ?
out_of(X, []  ).
       +   ?           + +          + ?
out_of(X,[Y|L]) :- dif(X,Y), out_of(X,L).
```

Program 7. Path-finding program.

```
test([S,E,N,D,M,O,R,Y]) :-
   add([D,N,E,S],[E,R,O,M],[Y,E,N,O,M]),
   S=\=0, M=\=0, different([S,E,N,D,M,O,R,Y]).

add(Xs,Ys,Zs) :- addc(Xs,Ys,0,Zs).

addc([],[],0,[]).  addc([],[],1,[1]).
addc([],[Y|Ys],C,Zs) :- addc([0],[Y|Ys],C,Zs).
addc([X|Xs],[],C,Zs) :- addc([X|Xs],[0],C,Zs).
addc([X|Xs],[Y|Ys],C,[Z|Zs]) :-
   addc99(X,Y,C,C1,Z), addc(Xs,Ys,C1,Zs).

% The predicate addc99(X,Y,C,C1,Z) is a collection of facts satisfying
%     X + Y + C = 10 × C1 + Z      (0 ≤ X, Y, Z ≤ 9; 0 ≤ C, C1 ≤ 1).

different([X|Xs]) :- out_of(X,Xs), different(Xs).
different([]).

out_of(X,[Y|Ys]) :- X=\=Y, out_of(X,Ys).
out_of(X,[]).
```

Program 8. "SEND + MORE = MONEY".

much more complex than the 8-queens program; however, since the problem is embedded in the program in this very case, we can statically analyze coroutining. We first partially evaluate the body goals of the top-level clause `test` until they are reduced to calls to `addc99` and '`=\=`'. Then we apply the mode analysis of the previous paper and move each call to '`=\=`' to the leftmost place where both its operands are ground$^{2)}$. Now we can derive an exhaustive search program using the previous method. Compilation of the addition table `addc99`, however, would require special consideration, because it is a large collection of facts called in four modes. Note that the complexity of the above procedure is polynomial with respect to the number of unknown digits, which is lower than the complexity of sequential exhaustive search.

4.3 Pre-Transformation and Compilation

A successfully analyzed program is then subject to the following transformation ((1) and (2) are in common with the previous method):

(1) Give a unique predicate name for each mode of an overloaded (multi-mode) predicate.

(2) Move head unification for output arguments to the end of the clause; and if the unification implied by the output arguments of any body goal may cause failure, move it just behind that goal.

(3) If the stream-output head argument, if any, is of the form `[`H_1`,` $\ldots$`,`H_n`|`T_n`]` ($n \geq 1$), replace it by a fresh variable T_0, and put the goals T_0`=[`H_1`|`T_1`]`, T_1`=[`H_2`|`T_2`]`, $\ldots$, T_{n-1}`=[`H_n`|`T_n`]` ($T_1, \ldots,$ T_{n-1} being fresh variables) in appropriate places, where the appropriate place for the goal T_{i-1}`=[`H_i`|`T_i`]` is the leftmost place to the right of T_{i-2}`=[`H_{i-1}`|`T_{i-1}`]` (if any) where H_i is ground.

(4) If the stream-input head argument, if any, is of the form `[`H_1`,` $\ldots$`,`H_n`|`T_n`]` ($n \geq 2$), we prepare $(n-1)$ auxiliary predicates to make sure that only one element is decomposed in each resolution. For example, the clause

```
p([X1,X2|Xs],...) :- test(X1,X2,...), p(Xs,...).
```

is rewritten as follows:

```
p([X1|Xs],...) :- p2(Xs,X1,...).
p2([X2|Xs],X1,...) :- test(X1,X2,...), p(Xs,...).
```

Compilation technique to a deterministic program is the same as that of the previous paper, except for the management of the tester of a generator-tester pair. The tester is managed by the corresponding generator in the form of a subcontinuation. The generator invokes the subcontinuation when it determines a new element of the shared variable, does possible derivations accumulating suspended goals, and returns to its own task letting the suspended goals be the new subcontinuation.

When a subcontinuation is invoked, goals other than those that must be suspended are executed according to the left-to-right rule. In Program 5, such executable goals were all system-defined and deterministic, but in general there can be user-defined goals possibly with more than one solution. The goals may have output arguments also, as long as they are 'normal' goals with no stream-input argument. They can be processed using the usual techniques of continuation handling and process forking.

When a subcontinuation is processed, resolution that would examine the next element of the shared variable, i.e., resolution using a clause having a non-variable stream-input head argument, must be suspended. In principle, suspension in our method should be defined for *resolution* and not for *a goal*, because some candidate clause may examine the shared variable while others do not. However, if the predicate being called by a goal is deterministic with respect to its stream-input argument (like `nocheck` and `qsafe` of Program 6), we need not control suspension of each candidate clause independently but instead we can suspend the goal itself as an optimization. Static mode analysis reveals whether or not resolution using each clause must be suspended and whether or not each predicate is deterministic, and we can generate an appropriate object code using such information.

5. PERFORMANCE

A compiled exhaustive search program can be executed as a deterministic Prolog program [14]. We compared Program 5 with other 8-queens programs using DEC-10 Prolog on DEC2065. The programs were timed after peephole optimization and excluding the time for garbage collection:

Program 2 with `bagof`*:*	*24765msec.*
Program 5 (deterministic program derived from Program 2):	*2045msec.*
Program 3 with `bagof`*:*	*1798msec.*
Deterministic program derived from Program 3:	*1938msec.*

Program 5 was 12 times faster than the original program executed without coroutining. It was 6% slower than the deterministic program derived from Program 3, but the difference seems quite reasonable. The following are the timing results of the path-finding programs (for obtaining four paths from `g` to `j`):

Program 7 with `bagof`*:*	∞
Deterministic program derived from Program 7:	*28msec.*
Program 7 optimized for sequential execution with `bagof`*:*	*36msec.*

We admit that the above comparison is unfavorable to collection of solutions using backtracking and `bagof` [9], since the timing results of backtracking programs include the time for reclamation of a stack area on backtracking, while the timing results of deterministic programs do not include the time for any storage reclamation. However, the situation might be quite different in parallel execution: Backtracking programs are clearly harder to parallelize than deterministic ones. The overheads of various methods of multiple environment management for OR-parallel execution have been reported in [2], but we still need to evaluate how much it pays to compile away multiple environments in parallel execution.

6. CONCLUSION

We have described a mechanizable method for deriving from a naive generate-and-test program an efficient deterministic program that collects all the solutions of the original program. We saw that coroutining of generate-and-test programs is amenable to static analysis for many (though not all) textbook examples. This means that control facilities such as `freeze` [4] can often be compiled away at the cost of static analysis. The analysis of a generate-and-test program whose data-flow is determined only at run time (e.g., a general cryptarithmetic program) is still an open problem, though a meta-programming technique might in principle enable us to use all our techniques at run time.

The results can be viewed as a step towards a user language of GHC (and other parallel logic programming languages) that is higher than AND-sequential pure Prolog. However, the use of GHC as a base language is in fact not essential for our technique. Looking into Program 5 for example, we find that the main mechanisms necessary to run it are composing and decomposing of ground data, recursive call, some means for chaining solutions, and (in case of parallel execution) process forking, all of which could be implemented in procedural (or even assembly) languages easily at least on sequential computers. So we can say that our results also suggest a compilation technique of a class of coroutining logic program into (very) low-level languages.

The mode analysis technique and our representation of subcontinuations apply also to program transformation from a coroutining program to an efficient sequential Prolog program that returns all the solutions by backtracking. The obtained program will have simple data-flow and hence will be easy to optimize compared with programs obtained by existing techniques.

Although we did not pursue full generality on the class of compilable programs, generation of a sequence in a generate-and-test manner should be a quite general framework. The important issue for the further development of the current technique is to accumulate practical and strictly logical generate-and-test programs.

Acknowledgments

The author is indebted to the members of First Research Laboratory, ICOT Research Center, for helpful discussions.

Notes

1) Gallagher [7] also uses more than one stack of goals and does informal static analysis very similar to ours. However, he did not compile away the shared variable and hence the generated program is not amenable to compilation to a deterministic program, while our method performs the analysis and compilation to a deterministic program in our own setting.

2) Seki and Furukawa [11] independently proposed this technique in a different context.

References

[1] Bruynooghe, M., De Schreye, D. and Krekels, B., Compiling Control. In *Proc. 1986 Symp. on Logic Programming*, IEEE Computer Society, 1986, pp. 70–77.

[2] Ciepielewski, A. and Hausman, B., Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs. In *Proc. 1986 Symp. on Logic Programming*, IEEE Computer Society, 1986, pp. 246–257.

[3] Clark, K. L., Predicate Logic as a Computational Formalism. Research Monograph 79/59 TOC, Dept. of Computing, Imperial College of Science and Technology, London, 1979.

[4] Colmerauer, A., Prolog II Reference Manual and Theoretical Model. Internal report, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1982.

[5] DeGroot, D., Restricted AND-Parallelism. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, 1984, pp. 471–478.

[6] Furukawa, K., *Introduction to Prolog.* Ohm-sha, Tokyo, 1986 (in Japanese).

[7] Gallagher, J., Simulating Coroutining for the 8-Queens Problem. *Logic Programming Newsletter*, L. M. Pereira (ed.), Universidade Nova de Lisvoa, No. 3 (1982), pp. 10–11.

[8] Gregory, S., Towards the Compilation of Annotated Logic Programs. Research Report DOC 80/16, Dept. of Computing, Imperial College of Science and Technology, London, 1980.

[9] Helmenegildo, M. V., *Discussion at the Megalips Plus Workshop*, Manchester, 1986.

[10] Mellish, C. S., Abstract Interpretation of Prolog Programs. In Proc. Third Int. Conf. on Logic Programming, Shapiro, E. (ed.), LNCS 225, Springer-Verlag, 1986, pp. 463–474.

[11] Seki, H. and Furukawa, K., Compiling Control by a Program Transformation Approach. ICOT Tech. Memorandum TM-0240, ICOT, Tokyo, 1986.

[12] Tamaki, H. and Sato, T. [1984] Unfold/Fold Transformation of Logic Programs. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ, Sweden, 1984, pp. 127–138.

[13] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. A revised version is in *Proc. Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, 1986, pp. 168–179.

[14] Ueda, K., Making Exhaustive Search Programs Deterministic. In Proc. Third Int. Conf. on Logic Programming, Shapiro, E. (ed.), LNCS 225, Springer-Verlag, 1985, pp. 270–282. A revised version will appear in *New Generation Computing*, Vol. 5, No. 1 (1987).

[15] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. Report TR-208, ICOT, Tokyo, 1986.

[16] Ueda, K., Introduction to Guarded Horn Clauses. ICOT Tech. Report TR-209, ICOT, Tokyo, 1986.