TM-0240

Notes on Transformation Techniques for Generate and Test Logic Programs

by

H. SEKI and K. FURUKAWA

November, 1986

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

Notes on Transformation Techniques for Generate and Test Logic Programs

Hirohisa SEKI and Koichi FURUKAWA[†]

ICOT Research Center
Institute for New Generation Computer Technology
Mita-Kokusai Bldg. 21F, 1-4-28, Mita,
Minato-ku, Tokyo 108, Japan

[†] order not significant

Abstract

This paper presents three transformation techniques for a class of logic programs. Target programs are declaratively written programs based on a generate and test paradigm, which are usually very inefficient under the Prolog default control strategy. All of three transformation techniques are based on a simple idea that a tester should prune incorrect search trees eagerly by interleaving testers into generators immediately after the testers become active. Rules of each transformation technique are given within the framework of Tamaki-Sato's unfold/fold transformation [TS84b], incorporating with a static dataflow analysis and/or some appropriate laws, and the correctness of each transformation is proved. Transformed programs under the default control strategy are assured of being at least as efficient as those under the coroutine mechanism. Interestingly, the above idea naturally leads to a promotion strategy [Bir84] known in functional programming and even for a class of programs to which the coroutine mechanism is not effective, the promotion strategy is proved to be effectively applied.

[Contents]

1 Introduction

Program transformation provides a powerful methodology for the development of software, especially the derivation of efficient programs either from their formal specifications or from declarative but inefficient programs. Programs written in a declarative form are often inefficient under the Prolog's standard left to right control rule. Typical examples are found in programs based on a generate and test paradigm. There seems to exist two alternative approaches to ensure efficiency for such programs. One approach is to introduce a coroutine (or freeze) mechanism implemented in IC-PROLOG [CMG82], PROLOG-II [Col82], MU-PROLOG [Nai85], CIL [Muk85], and so on. The other is a program transformation approach. Although the coroutine mechanism is useful and easy to use, it has often been pointed out that its control mechanism causes substantial runtime overhead. Furthermore, when all solutions of an original program are required, the transformation approach makes it possible to obtain a corresponding GHC [Ued85] program by further applying Ueda's method [Ued86a] to a transformed program. Hence, in this paper, we adopt a program transformation approach to get rid of the coroutine mechanism. Our objective is to provide transformation techniques effective in a class of typical programs of the generate and test type. Program transformation techniques described in this paper ensure those programs at least the same efficiency under the default control strategy as that of the corresponding programs with the coroutine mechanism, and sometimes the efficiency is shown to be substantially improved.

After summarizing preliminary materials in section 2, three transformation techniques based on unfold/fold transformation are proposed. In section 3, an unfold/fold transformation technique incorporating a simple dataflow analysis is described. Section 4 gives another unfold/fold transformation based on structural commutativity. In section 5, an unfold/fold transformation with promotion strategy is explained. The correctness of the above transformations is established in the appendix. In section 6, we give a summary of our transformation techniques and a discussion of related research.

2 Preliminaries

Program transformation rules used in this paper are based on those of Tamaki and Sato [TS84b]. Their basic rules are definition, unfolding and folding (for the strict definition, see [TS84b]) :

(1) Definition : Introduce a new predicate definition.

(2) Unfolding : Replace an atom A occurring in a body of a clause by the body of another clause whose head is unifiable with A.

(3) Folding : Replace goals Gs in a body of a clause by the head of another clause whose body is unifiable with Gs.

The following transformation rules are special cases of goal replacement in [TS84b].

(4) Laws : Transform goals *Gs* in a body by applying available rewriting rules about predicates appearing in *Gs*. (e.g., associativity, commutativity).

(5) Rearrangement : Rearrange order of goals in a body of a clause.

Target programs to which the above transformation rules are applied are typical programs based on the generate and test paradigm. Those programs are usually supposed to be of the form (hereafter, programs are defined following the syntax of DEC-10 Prolog [BBP*83]).

```
:- mode spec(+,-).
spec(X,Y) :- generator(X,Y), tester(Y).
```

The meaning of this program is that, for a given input variable (or possibly a set of input variables) `X`, predicate `generator` generates output `Y` until `Y` satisfies the condition specified by the predicate `tester`. A mode declaration is given for the sake of understandability and it shows how a program is to be used. Variable `Y` is called a *shared variable*, shared by the generator and the tester. In the following, data structures treated here are mainly confined to lists, although we believe our transformation techniques are effective for other recursively defined data structures.

3 Unfold/fold and rearrangement by dataflow analysis

All of our program transformation techniques described below are based on a simple guiding principle that a tester should prune incorrect search trees eagerly by interleaving testers into generators immediately after the testers become active. In order to illustrate this idea, we consider the following class of simple programs in this section, namely, *the tester of a given program can be unfolded into primitive (evaluable) predicates.* For these programs, unfold/fold transformation and rearrangement of goals are sufficient. A simple static dataflow (or mode) analysis of goals is necessary in order to rearrange goals appropriately. Namely, for a given goal, the analysis should give information about which arguments in the goal must be instantiated into ground terms, when it is executed for a certain ground input (if any) and terminates with success. The mode analysis given by [Mel85], for example, is sufficient for the present purpose. Using the mode analysis, the transformation for such class of programs is performed as follows:

i) Unfold the tester into primitive (evaluable) predicates.

ii) Determine by a mode analysis when variables in each derived primitive predicates are instantiated.

iii) Rearrange each primitive predicate just after a generater predicate which instantiates variables in the primitive predicate.

Clearly, the equivalence of the transformed program with an original program is preserved, since only unfolding and rearrangement rule are used. As an example, consider the following program.

Ex. 3.1 *Send More Money [Fur86] (a cryptarithmetic addition program)*[1]

```
send([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]])
     :- sum(R1,0,0,M,O), sum(R2,S,M,O,R1), sum(R3,E,O,N,R2),          (3-1)
        sum(R4,N,R,E,R3), sum(0 ,D,E,Y,R4),                            (3-2)
        different([S,E,N,D,M,O,R,Y]),                                  (3-3)
        M =\= 0, S =\= 0.                                              (3-4)

sum(R,X,Y,Z,R1)
   :-   remainder(R), digit(X), digit(Y),
        T is R+X+Y, R1 is T/10, Z is (T mod 10).

remainder(0).
remainder(1).
digit(0).
...
digit(9).
different([]).
different([X|Y]) :- out_of(X,Y), different(Y).
out_of(X,[]).
out_of(X,[A|L]) :- X =\= A, out_of(X, L).
```

The above program is a well-known cryptarithmetic addition program. For given strings of letters, "SEND", "MORE" and "MONEY", each of which represents different integers among 0,1,...,9, the problem is to find an appropriate assignment of digits for each letter so that adding the numbers represented by "SEND" and "MORE" yields the number represented by "MONEY". Goals (3-1), (3-2) are the generator part and (3-3), (3-4) are the tester part. From the definition, (3-3) can be unfolded into a set of evaluable predicates of the form : `X =\= Y`, where `X` and `Y` are different characters among S,E,N,D,M,O,R,Y. The mode analysis shows which arguments in a goal of the generator (predicate `sum` in this case) are instantiated when it is called and terminates with success. Since we assume that the generator part is executed in default control, it is easily known from the mode analysis when each predicate `X =\= Y` becomes evaluable.

Hence, we can interleave these primitive testers into the generator part immediately after they become evaluable as follows :

```
send([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]])
   :-   sum(R1,0,0,M,O),                          %       M is instantiated
                  M =\= 0,
        sum(R2,S,M,O,R1),                         %       S, O are instantiated
```

[1]Recently, we found that Ueda independently gives a quite similar transformation of this program [Ued86b], although he takes it up in a slightly different context.

```
        S =\= O, S =\= M,  S =\= O, O =\= M,
sum(R3,E,O,N,R2),                        %       E, N are instantiated
        E =\= M, E =\= S, E =\= O, E =\= N, N =\= M, N =\= S, N =\= O,
sum(R4,N,R,E,R3),                        %       R is instantiated
        R =\= M, R =\= S, R =\= O, R =\= E, R =\= N,
sum(O ,D,E,Y,R4),                        %       D, Y are instantiated
        D =\= M, D =\= S, D =\= O, D =\= E, D =\= N, D =\= R, D =\= Y,
        Y =\= M, Y =\= S, Y =\= O, Y =\= E, Y =\= N, R =\= Y.
```

Clearly further transformation is possible if we partially evaluate the generator predicates (i.e., "sum" in the above) and interleave each primitive tester into the appropriate places among those unfolded generator predicates. Then, the behavior of the derived program exactly corresponds to that of the original program under the coroutine mechanism.

4 Unfold/fold based on structural commutativity

In this section we describe another transformation technique which can apply to more complicated programs. They do not enjoy such properties as those given in Section 3. As an example, consider the following program.

Ex. 4.1 *Acyclic Path Finding Program*

```
:- mode good_path(+,-).
good_path(X, Path)
   :-   path(X, Path), good_list(Path).

path(goal, [goal]).
path(N, [N|Path])
   :-   neighbor(N, Next), path(Next, Path).

good_list([]).
good_list([X|L]) :-  out_of(X,L), good_list(L).
out_of(X,[]).
out_of(X,[A|L]) :- X =\= A, out_of(X,L).
```

For a given graph G, predicate good_path(X,Path) finds an acyclic path which starts from node X and ends at a designated node "goal". The relation of the arc from node X to Y in G is specified in terms of the predicate neighbor(X,Y). The predicate path(X, Path) is a generator which generates a path from node X to goal, while good_list(Path) checks whether a list Path contains element repetitions or not. Note that the above program might not terminate if it is executed under the Prolog default left to right control rule. The generator "path" instantiates a shared variable Path successively from its head. When Path is instantiated into [a|L], say, then the tester "good_list" could be unfolded into **out_of(a,L) & good_list(L)**. However, out_of(a,L) cannot be evaluated until the rest of the list L is instantiated. Hence, the unfold/fold transformation and rearrangement are clearly insufficient to interleave testers into generators immediately after the testers become evaluable. However, the following symbolic manipulation would suggest an alleviation of this difficulty. From its definition, the predicate good_list is symbolically transformed as follows:

```
good_list([X1,X2,...,Xn,goal])
   :-   out_of(X1,  [X2,  ...  ,Xn,goal]),
        out_of(X2,  [X3,...,Xn,goal]),
        ...
        out_of(Xn,  [goal]),
        out_of(goal,[]).
                                                % unfolding of each "out_of"
   :-   X1=\=X2,............,X1=\=Xn,X1=\=goal,
                X2=\=X3,...,X2=\=Xn,X2=\=goal,
                ...
                                    Xn=\=goal.
```

Since predicate "=\=" is commutative,

```
   :-   X2=\=X1,X3=\=X1,...,Xn=\=X1,goal=\=X1,
                X3=\=X2,...,Xn=\=X2,goal=\=X2,
                        ...
                                    goal=\=Xn.
```

Rearranging the order of the above goals, the structure of the formula is equivalently transformed as follows:

```
   :-                                   X2=\=X1,
                              X3=\=X2,   X3=\=X1,
                                     ...
                Xn=\=Xn-1,..,Xn=\=X2,    Xn=\=X1,
       goal=\=Xn,  ...      goal=\=X2, goal=\=X1.
```

We call this property "*structural commutativity*" here. Note that this reordering is analogous to the exchange of double summation: $\sum_{i=1}^{N}\sum_{j=i}^{N} x_{ij} = \sum_{j=1}^{N}\sum_{i=j}^{1} x_{ij}$.
Folding goals in the above formula:

```
   :-   out_of(X2,  [X1]),
        ...
        out_of(Xn,  [Xn-1,...,X2,X1]),
        out_of(goal,[Xn,Xn-1,...,X2,X1]).

  =  good_list([goal,Xn,...,X2,X1]).
```

Therefore,

```
   good_list([X1,X2,...,Xn,goal]) = good_list([goal,Xn,...,X2,X1])
```

holds and the predicate good_path is transformed as follows:

```
good_path(X1, [X1,X2,...,Xn,goal])
:-  path(X1, [X1,X2,...,Xn,goal]),       good_list([X1,X2,...,Xn,goal]).

:-  neighbor(X1,X2),                     out_of(X2,  [X1]),              (4-1)
    neighbor(X2,X3),                     out_of(X3,  [X2,X1]),
     ...                                  ...
    neighbor(Xn-1,Xn),                   out_of(Xn,  [Xn-1,...,X2,X1]),
    neighbor(Xn,goal),                   out_of(goal,[Xn,Xn-1,...,X2,X1]).
```

Now, in (4-1), when X2 is produced by neighbor(X1,X2), the tester out_of(X2,[X1]) can immediately check whether X2 is contained in a set of previously generated elements, namely {X1} or not. The same situation holds for other elements X3,...,Xn and goal. The second argument of the tester "out_of" is used as a stack which accumulates elements obtained so far. These considerations lead us to the following improved program good_path1, which includes an extra argument as an *accumulator* of previously generated elements.

```
good_path(X, Path)
   :-   good_path1(X,[],Path).

good_path1(goal, Path, [goal])
   :-   out_of(goal, Path).
good_path1(N, History, [N|Path])
   :-   out_of(N, History), neighbor(N, Next),
        good_path1(Next, [N|History], Path).
```

We can generalize the above discussion and give the following program transformation scheme.

A Program Transformation Scheme based on Structural Commutativity

Suppose we are given a program of the generate and test type described in (3-5). Suppose also that a predicate "tester" satisfies the following condition (we call this property the *structural commutativity* of predicate "tester") :

there exists a predicate tester' s.t.

```
          tester(L) = tester'(L^)  for any list L,                  (C4-1)
```

where

(1) L^ means a reversed list of L , i.e., reverse(L,L^), and

(2) `tester'` is of the following form :

```
                 tester'([]).
                 tester'([X|L]) :- t'(X,L), tester'(L).
```

Then, the above program is equivalently transformed as follows :

```
spec(X,Y) :- spec1(X,[],Y).

spec1(End,Hist,EndValue) :- terminator(End,EndValue), tester1(EndValue,Hist).
spec1(N,Hist,[V|Vs])
   :-   p(N,V,Next), t'(V, Hist),
        spec1(Next, [V|Hist], Vs).
```

Predicate `tester1` is defined in **Appendix-1**.

The detailed process of the program transformation is given in **Appendix-1**. Clark [Cla79] and Gregory [Gre80] show the transformation of eight queens problem (shown below) using accumulators. The above scheme gives a transformation which can deal such programs in a unified way.

Ex. 4.2 *Eight queens problem*

```
:- mode solution(+,-).
solution([1,2,..,8], Perm)
   :-   permutation([1,2,...,8], Perm), safe(Perm).

permutation([],[]).
permutation(L,[Q|M]) :- remove(Q,L,L1), permutation(L1,M).

remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]) :- remove(X,L,M).

safe([]).
safe([Q|List]) :- nodiagonal(Q,List,1), safe(List).

nodiagonal(Q1,[],N).
nodiagonal(Q1,[Q2|L],N)
   :-   noattack(Q1,Q2,N), N1 is N+1, nodiagonal(Q1,L,N1).

noattack(Q1,Q2,N)
   :-   Q1 > Q2, Diff is Q1 - Q2, Diff =\= N.
noattack(Q1,Q2,N)
   :-   Q2 > Q1, Diff is Q2 - Q1, Diff =\= N.
```

A solution of this program is represented as a permutation of `[1,2,...,8]` which gives the column numbers of the queen in each of the eight rows. Predicate `safe(L)` checks whether no two queens in L lie on the same diagonal. It is clear that predicate `safe` satisfies the condition `(C4-1)`, i.e., `safe(L) = safe(L~)`. Hence, the above program is transformed as follows :

```
solution([1,2,...,8], Perm)
     :-  solution1([1,2,...,8],[],Perm).

solution1([], Result, []).
solution1(L, History, [Q|M])
    :-    remove(Q,L,L1),
          nodiagonal(Q,History,1),
          solution1(L1,[Q|History],M).
```

Ex. 4.3 *Slow sort*

```
:- mode sort(+,-).
sort([],[]).
sort(X,Y) :- permutation(X,Y), ordered(Y).

ordered([]).
ordered([X]).
ordered([X,Y|L]) :- X =< Y, ordered([Y|L]).
```

Predicate ordered also satisfies the condition `(C4-1)`, since `ordered(L) = ordered'(L~)`, where predicate `ordered'` is of the form :

```
ordered'([]).
ordered'([X]).
ordered'([X,Y|L]) :- Y =< X, ordered'([Y|L]).
```

Hence, the above sort program is transformed as follows :

```
sort(X,Y)
    :-    sort1(X,[],Y).

sort1([],_,[]).
sort1(X,Hist,[Q|Vs])
    :-    remove(Q,X,X1),
          t'(Q, Hist),
          sort1(X1,[Q|Hist],Vs).
```

```
t'(_, []).
t'(X,[H|T])
   :-   H =< X, t'(X,T).
```

Note that the original program is an $O(n!)$ sort program, while the derived one is an $O(2^n)$ sort program, which achieves the same efficiency as that of the corresponding program under the coroutine mechanism.

Furthermore, the following proposition gives a simple sufficient condition of (C4-1).

A Sufficient Condition of (C4-1)

Suppose that the predicate "tester" is defined as follows :

```
                                                     % cf.
tester([]).                                          % good_list([]).
tester([X|L])                                        % good_list([X|L])
   :-  t(X,L),                                       %     :- out_of(X,L),
       tester(L).                                    %        good_list(L).
t(X,[]).                                             % out_of(X,[]).
t(X,[A|L])                                           % out_of(X,[A|L])
   :-  c(X,A),                                       %     :- X =\= A,
       t(X,L).                                       %        out_of(X,L).
```

Then "tester" satisfies (C4-1), i.e.,

```
            tester(L) = tester'(L~)        for any list L,
```

where

```
        tester'([]).
        tester'([X|L]) :- t'(X,L), tester'(L).
        t'(X,[]).
        t'(X,[A|L]) :- c(A,X), t'(X,L).
```

The correctness of this condition is given in **Appendix-2**. Note that those programs in Ex. 4.1, Ex. 4.3, and Ex. 4.3 satisfy the above sufficient condition.

5 Unfold/fold using promotion strategy

In this section we explain the third program transformation technique. A class of target programs here is quite different from those in previous section in that the shared variable (list) between a generator and a tester is not instantiated incrementally and the coroutine mechanism is no longer effective. Consider the following example which illustrates why the previously mentioned techniques are not appropriate and a new technique is required.

Ex. 5.1 *Another Slow Sort*

```
:- mode sort(+,-).
sort(X,Y) :- perm(X,Y), ordered(Y).                                    (5-1)
                              % the predicate "ordered" is defined in Ex. 4.3.
perm([],[]).
perm([A|X],Y) :- perm(X,Z), insert(A,Z,Y).
insert(A,X,[A|X]).
insert(A,[B|X],[B|Y]) :- insert(A,X,Y).
```

The above sort program differs from the one in Ex. 4.3 in that perm's 2nd clause is defined in different way, which delays the instantiation of the shared variable Y in (5-1). The following symbolic manipulation will illustrate how the computation proceeds.

```
    sort([X1,X2,...,Xn],Y)
:-    perm([X1,X2,...,Xn],Y),                        ordered(Y).
        |                   \
      perm([X2,...,Xn],Y1),   insert(X1,Y1,Y),
        |                 \
      perm([X3,..,Xn],Y2),   insert(X2,Y2,Y1),
       ...
        |              \
      perm([Xn],Yn-1),   insert(Xn-1,Yn-1,Yn-2),
        |          \
      perm([],Yn),    insert(Xn,Yn,Yn-1),
```

For a given list [a1,a2,...,an], the value of Y in a call perm([a1,a2,...,an], Y) is not determined until the value of Y1 in a call perm([a2,...,an], Y1) is completely instantiated and goal insert(X1,Y1,Y) is executed. Hence, neither the coroutine mechanism nor the transformation based on structural commutativity is effective in this case. Again, back to our first principle that a tester should prune incorrect search trees eagerly by interleaving testers into generators immediately after the testers become active, we examine the above computation in more detail. Goals insert(X1,Y1,Y), ordered(Y) in the above can be regarded as (a local) generate and test program. Namely, for ground inputs X1,Y1, a goal insert(X1,Y1,Y) non-deterministically instantiates a shared variable Y and then its value is checked by a goal ordered(Y). This is a reason of the inefficiency of the original program. Hence, instead of checking Y by ordered(Y) at the very end, the ground input Y1 should be checked beforehand whether ordered(Y1) holds. Furthermore, under the condition that ordered(Y1), we can deterministically insert X1 into Y1 obtaining the result Y for which it is guaranteed that ordered(Y), i.e., by employing predicate op_insert defined below.

```
                                                  % order preserving insertion
      op_insert(A,[],[A]).
```

```
op_insert(A,[B|X],[A,B|Y]) :-  A =< B.
op_insert(A,[B|X],[B|Y])
        :-  A > B,  op_insert(A,X,Y).
```

Interestingly, this consideration naturally leads us to the *promotion strategy* known in functional languages [Bir84]. We further apply symbolic manipulation to the above formula according to the promotion strategy. At first, we note that predicate "insert" and "ordered" satisfy the following condition (Bird calls this condition a "*continuity condition*" [Bir84]).

```
insert(A,Y,Z) & ordered(Z) = ordered(Y) & op_insert(A,Y,Z)          (5-2)
```

Then, using this logical equivalence, the original program is successively transformed as follows:
Unfold perm in (5-1) :

```
sort([],[]).                                                           (5-3)
sort([A|X],Y) :- perm(X,Z), insert(A,Z,Y), ordered(Y).                (5-4)
```

Use continuity condition (5-2) :

```
sort([A|X],Y) :- perm(X,Z), ordered(Z), op_insert(A,Z,Y).              (5-5)
```

Fold by sort in (5-5) :

```
sort([A|X],Y) :- sort(X,Z), op_insert(A,Z,Y).                          (5-6)
```

Generalizing the above discussion, we derive a generalized program transformation scheme, the form of which is quite similar to that used in functional languages [Bir84].

A Program Transformation Scheme Using the Promotion Strategy

Suppose that we are given a generate and test type program of the following form :

```
:- mode spec(+,-).
spec2(X,Y) :-  generator2(X,Y),  tester(Y).

generator2(End, EndValue) :- terminator(End, EndValue).
generator2(N, Result)
   :-   p(N,Value,Next), generator2(Next,Vs),
        modifier(Value,Vs,Result).
```

where modifier(Value,Vs,Result) is a certain predicate which returns a value Result from inputs Value and Vs.

If there exists some predicate modifier' s.t.

```
modifier(H,R,Result) & tester(Result)
     = tester(R) & modifier'(H,R,Result) for any H,R,Result,          (C5a)
```

then program spec2 is equivalently transformed into:

```
spec2(End,EndValue) :-  terminator(End,EndValue), tester(EndValue).
spec2(N,Result)
   :-   p(N,Value,Next), spec2(Next,Vs), modifier'(Value,Vs,Result).
```

The correctness of the derivation of the above program based on unfold/fold transformation is omitted because it is quite straightforward. As previously mentioned, the coroutine mechanism is not effective in this case. Hence, as for the coroutine approach, the computational complexity of Ex. 5.1 is $O(n!)$, where n is the length of an input list. On the other hand, program transformation using the promotion strategy produces an $O(n^2)$ insertion sort program.

Note that the promotion strategy is sometimes effective also in the program in (3-5). In this case, the continuity condition requires the existence of a predicate p' satisfying the following condition:

```
   p(N,Value,Next) & tester([Value|Vs])
     =  p'(N,Value,Next) & tester(Vs), where  generator(Next,Vs).        (C5b)
```

or, equivalently:

```
         p(N,Value,Next) & t(Value,Vs) = p'(N,Value,Next),
         where  generator(Next,Vs) & tester(Vs) [i.e., spec(Next,Vs)].   (C5b')
```

It could be considered that the above (C5b') gives an implicit definition of a predicate p'. When the above condition holds, then the original program can be transformed in a similar way as follows :

```
spec(End,EndValue) :- terminator(End,EndValue), tester(EndValue).
spec(N,[Value|Vs]) :- p'(N,Value,Next), spec(Next,Vs).
```

The slow sort program in Ex. 4.3 is the case where the above condition is satisfied. Here, the continuity condition is of the following form :

```
         remove(A,X,X1) & ordered([A|Y])
           =      select(A,X,X1) & ordered(Y), where permutation(X1,Y)
```

where select(A,X,X1) is a predicate which selects an element A from a given list X that is less than or equal to all the other elements X1. Using this condition, the original program is transformed as follows :

```
sort([],[]).
sort(X,[A|Y]) :- select(A,X,X1), sort(X1,Y).
```

The above derived program is precisely a selection sort program.

6 Discussion

We have shown how our program transformation techniques are applied to a class of typical generate and test programs and how they can be equivalently transformed into efficient programs under the Prolog default control rule. These transformation methods achieve the same efficiency as the coroutine mechanism. Indeed, efficiency can sometimes be substantially improved by employing the promotion strategy, even in cases where the coroutine mechanism is not effective. Furthermore, it should be noted that programs derived by the transformation techniques proposed here are shown to fall under the class of Prolog programs specified by Ueda [Ued86a], which are transformable into corresponding GHC (Guarded Horn Clauses) programs [Ued85]. Hence, our transformation techniques described in this paper make it possible to compile programs written in high-level and constraint-based representation languages such as CIL and Prolog-II into corresponding GHC programs.

There exist several studies on the program transformation techniques for generate and test logic programs. Gallagher proposes a transformation method for the eight queens problem [Gal82]. His approach differs from ours in that he has introduced a kind of "metapredicates" to simulate the coroutine mechanism. Another approach is presented by Bruynooghe et al. [BSK86]. From a trace tree obtained by symbolic execution of a given query, they derive a set of clauses which specifies the transition of states in that trace tree. Their approach is quite similar to Gallagher's in that the state transition is described in metapredicates. On the contrary, our transformation method does not introduce such metapredicates. The transformation rules are within the framework of Tamaki and Sato [TS84b], which preserves the equivalence of programs. Hence, the justification of our transformation techniques is quite immediate, as long as the correctness of the laws used is proved.

Gregory's approach is similar to ours [Gre80]. Using the information of IC-PROLOG's annotations which specify whether data is transferred eagerly or lazily, annotated programs are transformed into sequential programs. His method is, however, strictly based on Burstall and Darlington [BD77], and neither transformation techniques based on structural commutativity nor promotion strategy are introduced. Clark [Cla79] also gives a transformation for eight-queens program, which is an instance of our scheme based on structural commutativity. As an application of their theoretical framework, Tamaki and Sato propose a transformation system called "Append Optimizer" [TS84a]. Since they intend to make an efficient and fully automated transformation system, their target programs are confined to programs containing append predicates. However, their aim of reducing redundancy and nondeterminism by transformation is quite similar to ours. Recently, Ueda [Ued86b] proposed a transformation method which simulates the coroutine mechanism using continuation. His approach is quite different from ours in that his method transforms a given program into a deterministic one, and data transfer from the generator to the tester is performed not by a shared variable, but by continuations. Hence, the proof of the equivalence of a transformed program with an original program seems to be left.

Compared with previous transformation approaches, the contributions of this paper could be summarized as follows :

1) Three transformation techniques are presented which are effective for a class of gen-

erate and test logic programs.

2) A sufficient condition is given where a transformation based on structural commutativity is proved to be effective.

3) The promotion strategy is shown to be effective for a class of generate and test logic programs, even if the coroutine mechanism is not effective.

Acknowledgement

The authors wish to express their thanks to Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity to pursue this research. We would also like to thank Yuji Matsumoto, Toshihiko Miyazaki and other members of ICOT First research laboratory who participated in discussion on their research. Special thanks are due to Kazunori Ueda and Akikazu Takeuchi, who gave valuable suggestions on our current work.

References

[BBP*83] D. L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira, and D. H. D. Warren. *DECsystem-10 Prolog User's Manual.* November 1983.

[BD77] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, 1977.

[Bir84] R. S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Trans. on Programming Languages and Systems*, 6:487–504, 1984.

[BSK86] M. Bruynooghe, D.D. Schreye, and B. Krekels. Compiling Control. In *Proc. 1986 Symposium on Logic Programming*, pages 70–77, IEEE Computer Society, 1986.

[Cla79] K.L. Clark. *Predicate Logic as a Computational Formalism.* Research Monograph 79/59, TOC, Dept. of Computing, Imperial College of Science and Technology, 1979.

[CMG82] K.L. Clark, F. McCabe, and S. Gregory. *IC-Prolog Language Features*, in *Logic Programming*, pages 253–266. Academic Press, 1982.

[Col82] A. Colmerauer. *PROLOG II Reference Manual and Theoretical Model.* Technical Report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, 1982.

[Fur86] K. Furukawa. *Introduction to Prolog.* Ohm-sha, Tokyo, 1986. in Japanese.

[Gal82] J. Gallagher. Simulating Corouting for the 8 Queens Program. *Logic Programming Newsletter*, (3):10–11, 1982. Pereira, L.M. (ed.) Universidade Nova de Lisvoa.

[Gre80] S. Gregory. *Towards the Compilation of Annotated Logic Programs.* Research Report DOC 80/16, Dept. of Computing, Imperial College of Science and Technology, 1980.

[Mel85] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *J. Logic Programming*, 2(1):43–66, 1985.

[Muk85] K. Mukai. *Horn Clause Logic with Parameterized Types for Situation Semantics Programming.* ICOT Technical Report TR-101, ICOT, 1985.

[Nai85] L. Naish. Automating Control for Logic Programs. *J. Logic Programming*, 3:167–183, 1985.

[TS84a] H. Tamaki and T. Sato. On Append Optimizer. In *Proceedings of Logic Programming Conference*, Tokyo, 1984. in Japanese.

[TS84b] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, 1984.

[Ued85] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, 1985. A revised version is in Proc. Logic Programming '85, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg, pp.168-179, 1986.

[Ued86a] K. Ueda. Making Exhaustive Search Programs Deterministic. In *Proceedings of the Third International Conference on Logic Programming*, pages 270–282, London, 1986.

[Ued86b] K. Ueda. Making Exhaustive Search Programs Deterministic (II). In *Proceedings of the Third Annual Conference of Japan Society of Software Science and Technology*, Tokyo, 1986. in Japanese.

Appendix 1

Before we prove the correctness of the transformation scheme described in Section 4, we need the following lemma.

Lemma

Suppose that predicate "`tester`" is defined as follows :

```
tester([]).
tester([X|L]) :- t(X,L), tester(L).
```

If "`tester`" satisfies the following condition :

there exists a predicate `tester'` s.t.

```
tester(L) = tester'(L^)  for any list L,                    (C4-1)
```

where

(1) `L^` means a reversed list of `L` , i.e., `reverse(L,L^)` , and

(2) `tester'` is of the following form :

```
                tester'([]).
                tester'([X|L]) :- t'(X,L), tester'(L).
```

then

```
              tester(L) = tester1(L,[]),                                        (Lm)
```

where

```
       tester1([],_).
       tester1([X|L], Y) :- t'(X,Y), tester1(L, [X|Y]).
```

[proof]
(case 1) When L = [ ], the proof is trivial.
(case 2) When length(L) $\geq$ 1, the proof is by induction on the length of list L.

[base case] When length(L) = 1, say L=[a], then the proposition to be proved is that

```
     tester([a])  =  tester1([a],[]).
```

Using (C4-1) and unfolding the right hand side (RHS) of the above,

```
     tester'([a])  =  t'(a,[]), tester1([],[a]).
```

Unfolding the left hand side of the above, we obtain

```
     t'(a,[]) = t'(a,[]),
```

which is trivial.

[induction step] Suppose that (Lm) holds for any list L s.t. length(L) $\leq$ k (k$\geq$1).

Here we introduce the following notation conventions.

For any list L, L^ means a list s.t. reverse(L,L^).

*For any list L,M, L*M means a list N s.t. append(L,M,N).*

Let L be a list s.t. length(L) = k. We must prove that

```
     tester([a|L]) = tester1([a|L], []).
```

Using `(C4-1)` and unfolding RHS,

```
     tester'(L^*[a]) = t'(a,[]), tester1(L,[a]).
```

Let L^ = [l|M^] (i.e., L =M*[l], note that length(M)= k-1). Then the following transformation sequence is allowed :

Unfolding of LHS :

```
     t'(l,M^*[a]), tester'(M^*[a]) = t'(a,[]), tester1(M*[l],[a]).
```

Using (C4-1) :

```
     t'(1,M^*[a]), tester([a|M]) = t'(a,[]), tester1(M*[1],[a]).
```

By the induction assumption :

```
     t'(1,M^*[a]), tester1([a|M],[]) = t'(a,[]), tester1(M*[1],[a]).
```

Unfolding of tester1 in LHS :

```
     t'(1,M^*[a]), t'(a,[]), tester1(M,[a]) = t'(a,[]), tester1(M*[1],[a]),
```

which is equivalent to :

```
     t'(1,M^*[a]), tester1(M,[a]) = tester1(M*[1],[a])                (A1-1)
```

In place of (A1-1), we prove the following generalized formula :

for any list M,N :

```
     t'(1,M^*N), tester1(M,N) = tester1(M*[1],N)                        (A1-1')
```

[proof of (A1-1')] The proof is by induction on M in (A1-1').

[base case] (i.e., M = [ ]) The proposition to be proved is

```
          t'(1,N), tester1([],N) = tester1([1],N).
```

Unfolding RHS, the above is equivalent to

```
          t'(1,N) = t'(1,N), tester1([],[1|N]),
```

which is trivial from the definition.

[induction step] Suppose that, for any N,

```
          t'(1,M^*N), tester1(M,N) = tester1(M*[1],N)
```

holds. Then, we have to prove that

```
          t'(1,M^*[a]*N), tester1([a|M],N) = tester1([a|M]*[1],N).
```

Unfolding tester1 in each side and noting that

```
          [a|M]*[1] = [a]*M*[1], M^*[a]*N = M^[a|N],
```

the above formula is rewritten into :

```
          t'(1,M^*[a|N]), t'(a,N), tester1(M,[a|N])
                    = t'(a,N), tester1(M*[1],[a|N]),
```

which is equivalent to :

```
          t'(1,M^*[a|N]), tester1(M,[a|N])  =  tester1(M*[1],[a|N]).
```

The above formula is immediate from the assumption of induction.

(Q.E.D.)

The correctness of a transformation scheme based on structural commutativity

```
spec(X,Y) :- generator(X,Y), tester(Y).                                (A1-2)
```

By lemma (Lm), the above is equivalent to :

```
spec(X,Y) :- generator(X,Y), tester1(Y,[]).                            (A1-3)
```

Then the following transformation sequence is allowed.
  Introducing a new definition :

```
spec1(X,Hist,Y) :- generator(X,Y), tester1(Y,Hist).                         (A1-4)
```

Folding (A1-3) by (A1-4) :

```
spec(X,Y) :- spec1(X,[],Y).
```

Unfolding generator in (A1-4) :

```
spec1(End,Hist,EndValue) :- terminator(End,EndValue), tester1(EndValue, Hist).

spec1(N,Hist,[Value|Vs])
   :-   /* generator(N, [Value|Vs])           */
              p(N,Value,Next),
              generator(Next,Vs),
        /* tester1([Value|Vs], Hist).         */
              t'(Value,Hist),
              tester1(Vs, [Value|Hist]).                                  (A1-5)
```

Folding (A1-5) by (A1-4) :

```
spec1(N, Hist, [Value|Vs])
   :-   p(N,Value,Next), t'(Value,Hist),
        spec1(Next, [Value|Hist], Vs).
```

Appendix 2

The correctness of the sufficient condition of (C4-1)
  Suppose that the predicate "`tester`" is defined as follows :

```
                                        % cf.
  tester([]).                           % good_list([]).
  tester([X|L])                         % good_list([X|L])
      :-  t(X,L),                       %     :- out_of(X,L),
          tester(L).                    %        good_list(L).
  t(X,[]).                              % out_of(X,[]).
  t(X,[A|L])                            % out_of(X,[A|L])
      :-  c(X,A),                       %     :- X =\= A,
          t(X,L).                       %        out_of(X,L).
```

  Then "`tester`" satisfies (C4-1), i.e.,

```
            tester(L) = tester'(L~)      for any list L,              (C4-1)
```

where

```
        tester'([]).
        tester'([X|L]) :- t'(X,L), tester'(L).
        t'(X,[]).
        t'(X,[A|L]) :- c(A,X), t'(X,L).
```

[proof]
(case 1) When L=[ ], the proof is trivial.
(case 2) When length(L) $\geq$ 1, the proof is by induction on the length of list L.

[base case] When length(L) = 1, say L=[a], then the proposition to be proved is that

```
      tester([a]) =  tester'([a]),
```

both sides of which are unfolded as follows:

```
      t(a,[]), tester([]) = t'(a,[]), tester'([]),
```

which is immediate from the definitions of predicates `t`, `t'`, `tester` and `tester'`.

[induction step] Suppose that `(C4-1)` holds for any list L s.t. length(L) $\leq$ k (k$\geq$1).

Let L be a list s.t. length(L) = k. We must prove that

```
      tester([a|L]) = tester'(L^*[a]).
```

Let L=M*[l] (i.e., L^=[l|M^]). Then the following transformation sequence is allowed:

Unfolding each atom :

```
      t(a,M*[l]), tester(M*[l]) = t'(l,M^*[a]), tester'(M^*[a]).
```

Using the assumption of induction in both sides :

```
      t(a,M*[l]), tester'([l|M^]) = t'(l,M^*[a]), tester([a|M]).
```

Unfolding in both sides :

```
      t(a,M*[l]), t'(l,M^), tester'(M^) = t'(l,M^*[a]), t(a,M), tester(M)
```

Again, using the induction assumption (i.e., `tester'(M^) = tester(M)`),

```
      t(a,M*[l]), t'(l,M^) = t'(l,M^*[a]), t(a,M).                  (A2-1)
```

The correctness of `(A2-1)` is immediate if the following lemmas are proved :

for any a,l, L,

```
      t(a, L*[l]) = t(a,L), c(a,l)                                  (A2-2)
      t'(a, L*[l]) = t'(a,L), c(l,a)                                (A2-2')
```

[proof of (A2-2)] The proof is by induction on L.

[base case] i.e., L=[ ] The proposition to be proved is

```
      t(a,[l]) = t(a,[]), c(a,l),
```

The above is equivalently unfolded into :

```
      c(a,l), t(a,[]) = t(a,[]), c(a,l),
```

which is trivial.

[induction step] We must prove that

```
      t(a,[b|L]*[l]) = t(a,[b|L]), c(a,l).
```

Unfolding predicates "t" in both sides. we obtain :

```
      c(a,b), t(a, L*[l]) = c(a,b), t(a,L), c(a,l),
```

which is equivalent to :

```
      t(a, L*[l]) = t(a,L), c(a,l).
```

The above is clear from the induction assumption. (Q.E.D.)

Note that the proof of `(A2-2')` is similar to the above.

(Q.E.D.)