New Generation Computing, 6 (1988) 143-157
OHMSHA, LTD. and Springer-Verlag

Unfolding Rules for GHC Programs

Koichi FURUKAWA, Akira OKUMURA
and Masaki MURAKAMI
ICOT Research Center,
Institute for New Generation Computer Technology,
1-4-28, Mita, Minato-ku, Tokyo 108, Japan.

Received 17 March 1988

Abstract This paper presents a set of rules for the transformation of GHC (Guarded Horn Clauses) programs based on unfolding. The proposed set of rules, called UR-set, is shown to preserve freedom from deadlock and to preserve the set of solutions to be derived. UR-set is expected to give a basis for various program transformations, especially partial evaluation of GHC programs.

Keywords: GHC, Unfold/fold Transformation, Partial Evaluation

§1 Introduction

It is expected that fruitful results will follow program transformation research in parallel logic languages such as GHC,[6] PARLOG[2] and Concurrent Prolog.[4] Several preliminary results have been reported, including the application of partial evaluation to meta-programs in FCP (Flat Concurrent Prolog) to obtain a realistic operating system[5] and program transformation to fuse two concurrent processes to increase efficiency.[3]

However, there are two problems caused by the guard/commit mechanism in the program transformation of parallel logic languages: synchronization and nondeterminacy. In parallel logic languages, causality relations exist between unifications due to the guard/commit mechanism. Therefore, careful handling is necessary for transformation, such as changing not only body parts but also guard patrs of original programs. For example, let us consider the following GHC program:

```
(C0)  p([A|In], O) :- true | q(A, In, O)
(C1)  q(A, In, O) :- true | O=[A|Out], r(In, Out)
(C2)  r([B|In], O) :- true | O=[B]
```

By unfolding the clause, (C1), at the goal, r(In, Out), the following clause is obtained. (The definition of unfolding used in this note is given in the next section.)

```
(C1)′  q(A, [B|In1, O) :- true |
                 O=[A|Out], Out=[B].
```

The problem is that the behavior of program {(C0), (C1), (C2)} differs from that of {(C0), (C1)′}. In the former program, the output variable, O, of p can be instantiated just after the instantiation of the first element of p's first argument, whereas in the latter case, it is delayed until both the first and the second elements of the same argument are instantiated. The delay of output variable instantiation may cause a further problem. Let us add a clause

```
(C3)  s([X|Xs], In) :- true | In=[b|In1],
```

and consider the goal

```
(G)   ?-p([a|In], O), s(O, In).
```

Then, the second element of p's first argument cannot be instantiated before goal s(O, In) is executed and In is instantiated to [b|In1]. However, the instantiation of p's second argument, O, is necessary for the goal, s(O, In), to commit. Therefore, goal (G) will cause a deadlock when it is executed under program {(C0), (C1)′, (C3)}.

Nondeterminacy is another source of difficulties in unfolding. A careless application of unfolding may limit some goals to commit to particular clauses even if there are other alternatives, because determinacy cannot be judged by its textual appearance during program transformation.

Thus, the unfolding based transformation of GHC programs needs much consideration. We have been researching this topic, and have obtained a plausible answer, UR-set. UR-set is a set of transformation rules from one GHC program to another. Each rule preserves a single step of the transformation, and multiple application derives further transformation. These transformations do not change what solutions can be found nor freedom from deadlock of the source program.

Section 2 introduces the rules of UR-set. Section 3 argues the correctness of UR-set. Section 4 gives an example of transformation.

§2 UR-set

UR-set is a set of transformation rules for GHC programs. A program is a set of clauses, and UR-set provides a plausible transformation from one program to another. Each rule makes a single step of the transformation, which is based on replacing a clause of the source program by zero or more new clauses. The new clauses are mainly derived by goal substitution of the source clause by unfolding.

First, several terms which will be used to describe UR-set are defined.

Definition: unfolding
Consider clauses P and Q as

P :: Hp :- Gp | Bp
Q :: Hq :- Gq | Bq

where Hp and Hq are atomic formulas which have all distinct variables for their arguments, and each of Gp, Bp, Gq, and Bq is a sequence of goals. If there is a substitution, θ, which makes Hq the same as a goal, A, in Bp, unfolding clause P at goal A by clause Q is defined as to obtain a merged clause, R, which is

R :: Hp :- Gp, Gqθ | Bp′, Bqθ

where Bp′ is Bp without goal A.

Definition
A clause for a given goal is

satisfied	if its guard is already true without further argument instantiations.
candidate	if the goal is not sufficiently instantiated to judge whether the guard is true or not.
unsatisfiable	if the guard is already known to be unsatisfiable.

Example
For a goal, p(1,A),
(p(X, Y) :- X=1 | ...) is satisfied,
(p(X, Y) :- Y=1 | ...) is candidate, and
(p(X, Y) :- X=2 | ...) is unsatisfiable.

A goal is immediately executable if there is no candidate clause for that goal.

Definition: input related
A variable is input related if it is an input variable appearing in the head of a clause or there exists a guard goal which contains both that variable and an input related variable.

UR-set defined as a set of rules is divided into two groups: the first group (Rule 1 and 2) handles immediately executable goals appearing in the body part, and the second group (Rule 3 and 4) prepares for further unfolding. In UR-set shown below, the differentiation of input and output variables is assumed.

UR-set is for GHC clauses whose head arguments are all distinct variables. It is easy to understand that the same effect as any double occurrences of the same variable or constant patterns in a head can be implemented by its guard goals instead. The clause so implemented is called the normal form of its original clause.

Example
(p(A, B, C) :- A=1, B=C | ...) is the normal form of

(p(1, X, X) :- true | ...).

UR-set

Rule 1 Unification Execution/Elimination
An explicit unification (=/2) appearing in the guard or the body of a clause, C, is symbolically executed within the body part; that is, a further instantiated value substitutes corresponding variable occurrences within the body. If a unification in the guard fails, the clause is eliminated. Furthermore, if neither side of = includes any variables which also appear in any other literal of the clause, the unification goal is eliminated after the substitution. Thus, a new clause, C′, is derived from the original C. A new program is derived by replacing C of the original program by C′.

Example
(p(X) :- X=a | q(X))
is substituted by (p(X) :- X=a | q(a)).
(p :- true | X=a, q(X))
is substituted by (p :- true | q(a)).

Rule 2 Unfolding at an Immediately Executable Gogl
Let a clause, C, be of the form:

A :- G1, G2, ..., Gm | A1, A2, ..., An.

C is unfolded at an immediately executable body goal, Ai, by all its satisfied clauses, Cij ($1 \leq j \leq l$; l is the number of satisfied clauses). The resulting clause, Dij, is obtained from the original clause, C, by replacing goal Ai by the body of Cij. Dij is a guarded resolvent of C and Cij whose guard goals are the same as C, because the guards of Cij must be true. Thus, a new program is derived by replacing clause C of the original program by all of Dij.

Example
{(p :- true | q, a(1), r)}
is substituted by

{(p :- true | q, b, c, r),
(p :- true | q, d, e, r)}

where

{(a(X) :- X=1 | b, c),
(a(X) :- X>0 | d, e),
(a(X) :- X=2 | f, g)}

Rule 3 Predicate Introduction and Folding

Let the clause, C, be defined as

P :- G1, G2, ..., Gm | U1, U2, ..., Up, N1, N2, ..., Nq

where Ui ($0 \leq i \leq p$) are output unifications and Nj ($0 \leq j \leq q$) others. Furthermore, let the intersection of a set of variables appearing in G1, G2, ..., Gm, U1, U2, ..., Up and that appearing in N1, N2, ..., Nq, be X1, X2, ..., Xr. Then, a new clause, C1, of newP is introduced as

newP(X1, X2, ..., Xr) :- true | N1, N2, ..., Nq.

Then, the sequence of Nj of C is folded by C1 and a transformed clause, C′, is obtained as

P :- G1, G2, ..., Gm | U1, U2, ..., Up, newP(X1, X2, ..., Xr).

Thus, a new program is derived by replacing clause C of the original program by C1 and C′. This rule is used to transform clauses into forms where Rule 4 can be applied.

Example
(p(X, Y) :- X>0 | Y=[X|Z], q(Z), r)
is substituted by

{(p(X, Y) :- X>0 | Y=[X|Z], newP(Z)),
(newP(X) :- true | q(X), r)}.

Rule 4 Unfolding across Guard
Let a clause, C, be of the form:

C :: A :- G1, G2, ..., Gm | A1, A2, ..., An.

If Rule 1 cannot be applied to C and no Ai is an output unification, C is unfolded at each Ai simultaneously. Let Cij ($1 \leq j \leq mi$; mi is the number of the clauses) which is satisfied or candidate for Ai be of the form

Aij :- H1, H2, ..., Hp | I1, I2, ..., Iq.

Furthermore, let Dij be the result of unfolding C at a goal, Ai, by Cij and let θ be the substitution used to derive Dij from C and Cij.

If there exists a guard goal, Hiθ, in Dij which is false or contains variables of Ai but not input related variables of A, then discard the Dij.

A new program is derived by replacing clause C by all of Dij ($1 \leq i \leq n$, $1 \leq j \leq mi$) which are not discarded.

Example
(p(X) :- true | q(X), r(X))
is replaced by

{(p(X) :- X>3 | q1, r(X)),

(p(X) :- X≤3 | q2, r(X)),
(p(X) :- X<2 | q(X), r1),
(p(X) :- X≥2 | q(X), r2)}

where

{(q(X) :- X>3 | q1),
(q(X) :- X≤3 | q2),
(r(X) :- X<2 | r1),
(r(X) :- X≥2 | r2)}.

§3 Informal Discussion of the Correctness of UR-set

A rule of transformation must provide some equivalence. The follwing properties are expected between the original program, P, and a transformed program, P′, for any goal, G, in P.

(a1) If G has a solution in P, it has the same solution in P′.
(a2) If G has a solution in P′, it has the same solution in P.
(a3) If G can never lead to a deadlock in P, neithr can it in P′.

The above properties do not allow for cases of deadlock, failure, or infinite loops in the original program. However, we consider those programs as mistakes, and have not considered them. This section gives an informal discussion showing that UR-set provides these properties.

Note that property (a1) is related to soundness and is often called partial correctness of the transformation. Also, properties (a1) and (a2) together are related to completeness and are called total correctness. (a3) is a special property related to concurrent programming languages. These three points are discussed separately. In the following discussion, arguments for Rule 3 are omitted because it is obvious that it will keep the above properties.

3.1 Partial Correctness (a1)

If commit operators are ignored, then every rule in UR-set is a kind of unfold/fold rules of Prolog programs. Therefore, the same argument as [TS 84] can be used to show its partial correctness except for the control issue.

There are three problems related to the control issue. The first problem is that some of the nonterminating GHC programs, such as an operating system, are still useful and cannot be exclude from the discussion. One possible way to avoid the problem is to approximate every noterminating program by an appropriate terminating program just by adding a clause for termination. The resulting program will terminate if its input is finite.

The second problem is that commit operators will reduce the solution space by throwing uncommitted clauses away. It is necessary to show that any transformed program, P′, will not compute any solution which is not included in the solution set of the original program, P. If guard conditions in P′ are weaker than those in P, then P′ might compute a larger solution set than P.

Since Rules 1 and 2 do not change any guards, they will not weaken guards. In Rule 4, an attempt is made to form a new guard by combining the guard of the original clause, C, with the guard of a clause, Cij, to be called from some goal in the body of C. In some cases, the original guard may be combined with a guard having a unification with no input related variables of C. If the clause is committed, then the meaningless unification will compute solutions other than those in the original program, P. However, since resulting clauses containing meaningless unification are discarded, this problem does not arise.

The third problem is that some program which will cause deadlock may be transformed to a terminating program. Since these programs are excluded from this discussion, there are no problem.

3.2 Total Correctness (a2)

To prove the total correctness of the UR-set, it must be shown that the solution space will not be shrunk by the transformation. There are three possible causes of shrunk solution space:

(1) the guard conditions have been strengthened,
(2) scheduling nondeterminism has been lost, and
(3) commitment nondeterminism has been lost.

Since the usual unfold/fold rules ignoring the treatment of guards are used, guard conditions in total are not logically changed. Therefore, the guard conditions are not strengthened by the applications of UR-set.

Some considerations for the scheduling nondeterminism are required. In the case of Rules 1 and 2, any ordering can be selected by applying Rules 1 and 2 in a certain order, because unifications within a guard or body and immediately executable goals can be executed at any time independent of other goals' status.

In the case of Rule 4, each new clause obtained, Dij, is a result of the (guarded) resolution of the original clause, C, and a clause, Cij, to be called from some body goal of C. To solve the guard of Dij, the first goal to be executed after committing C in the original program, P, must be specified. Therefore, it seems to lose scheduling nondeterminism. However, since the application of Rule 4 unfolds at all the possible body goals of C, there are also candidate clauses corresponding to specify other goals to be executed first.

Next, commitment nondeterminism is discussed. To avoid too early commitment to some of the clauses for a nondeterministic goal, we excluded those goals having "candidate" clauses from immediately executable goals. Therefore, those goals are unfolded by Rule 4. Since unfolding by Rule 4 does not perform any commitment, no nondeterminacy is lost.

3.3 Preserving Freedom from Deadlock

Preserving freedom from deadlock is the most critical problem, particularly for concurrent languages like GHC. Note that, the entire behavior of a program is determined by conditions of such output unification, because only output variables affect to other processes. Therefore, unless those conditions are changed by the transformations, freedom from deadlock is preserved. The only rule changing guard part is Rule 4. However, the clauses for which Rule 4 is applicable are only those without output unifications. Therefore, no rule application causes deadlock between the goal which calls the transformed clause and other and-goals. Also, no rule produces any clauses the call to which will deadlock if the original program, P, is deadlock-free.

One further comment is on the possible interaction between nondeterminism and change of output unification conditions. Suppose that the unfolding of an executable branch of a nondeterministic goal produces a new output unification. Then, application of Rule 4 at another goal will change the guard conditions for the output unification to be executed. However, the nondeterministic goal is also unfolded at during the same application of Rule 4, and there is another clause which will perform the output unification at some proper time. Therefore, deadlock can be avoided in this situation.

§4 Brock-Ackerman Problem

This section presents an example of the application of the proposed set of rules, called the Brock-Ackerman Problem.[1] Consider the porgram below (i= 1, 2).

```
p1([A|In], Res) :- true | p11(In, Out), Res=[A|Out].
p11([A|In], Res) :- true | Res=[A].

p2([A, B|_], Res) :- true | Res=[A, B].

dup([A|I], Res) :- true | Res=[A, A].

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

si(Ix, Iy, Out) :- true |
        dup(Ix, Ox), dup(Iy, Oy), merge(Ox, Oy, Oz), pi(Oz, Out).

ti(In, Out) :- true | si(In, Mid, Out) plus1(Out, Mid).

plus1([A|In], Out) :- A1:=A+1 | Out=[A1].
```

The clause of p2 can be derived, if the clause of p1 at p11(In, Out) is unfolded by the clause of p11. s1 and s2 have the same set of solutions. Therefore, in this sense, that unfolding is correct. However, t1 and t2 has a different set of solutions, and it turns out that the transformation may cause trouble.

Our rules cannot provide unfolding at the goal, p11(In, Out). We consider it impossible to obtain an unfolded clause which behaves correctly in all contexts. However, the rules provide fair transformations in certain contexts.

If we start with the clause of ti with a mode declaration as ti(+, −), then the contexts for pi are limited and therefore the clause can be transformed as part of the total transformation.

The following shows the transformation sequence. (Clauses are handled in their normal form.)

```
ti(In, Out) :- true | si(In, Mid, Out), plus1(Out, Mid).
------------------------------------------------------------------ R2
ti(In, Out) :- true | /*si(In, Mid, Out),*/
        dup(In, Ox), dup(Mid, Oy), merge(Ox, Oy, Oz), pi(Oz, Out),
        plus1(Out, Mid).
------------------------------------------------------------------ R4
ti([A|_], Out) :- true | /*dup(In, Ox),*/
        Ox=[A, A], dup(Mid, Oy), merge(Ox, Oy, Oz), pi(Oz, Out),
        plus1(Out, Mid).
------------------------------------------------------------------ R1
ti([A|_], Out) :- true | /*Ox=[A, A],*/
        dup(Mid, Oy), merge([A, A], Oy, Oz), pi(Oz, Out),
        plus1(Out, Mid).
------------------------------------------------------------------ R4
ti([A|_], Out) :- true | dup(Mid, Oy),
        /*merge([A, A], Oy, Oz),*/
        Oz=[A|Oz1], merge([A], Oy, Oz1),
        pi(Oz, Out), plus1(Out, Mid).
------------------------------------------------------------------ R1
ti([A|_], Out) :- true | dup(Mid, Oy),
        /*Oz=[A|Oz1],*/merge([A], Oy, Oz1),
        pi([A|Oz1], Out), plus1(Out, Mid).
------------------------------------------------------------------ ⟨α⟩

i→1
------------------------------------------------------------------

t1([A|_], Out) :- true |
        dup(Mid, Oy), merge([A], Oy, Oz1),
        p1([A|Oz1], Out), plus1(Out, Mid).
------------------------------------------------------------------ R2
t1([A|_], Out) :- true | dup(Mid, Oy), merge([A], Oy, Oz1),
```

```
        /*pl([A|Oz1], Out),*/pl1(Oz1, Out1), Out=[A|Out1],
        plus1(Out, Mid).
--------------------------------------------------------------------- R1
tl([A|_], Out) :- true | dup(Mid, Oy), merge([A], Oy, Oz1),
        pl1(Oz1, Out1), Out=[A|Out1],
        plus1([A|Out1], Mid).
--------------------------------------------------------------------- R3
tl([A|_], Out) :- true | tl1(A, Out1), Out=[A|Out1].
tl1(A, Out1) :- true |
        dup(Mid, Oy), merge([A], Oy, Oz1),
        pl1(Oz1, Out1), plus1([A|Out1], Mid).
--------------------------------------------------------------------- R4
tl1(A, Out1) :- true | dup(Mid, Oy),
        /*merge([A], Oy, Oz1),*/Oz1=[A|Oz2], merge([], Oy, Oz2),
        pl1(Oz1, Out1), plus1([A|Out1], Mid).
tl1(A, Out1) :- A1:=A+1 |
        dup(Mid, Oy), merge([A], Oy, Oz1),
        pl1(Oz1, Out1),
        /*plus1([A|Out1], Mid).*/Mid=[A1].
--------------------------------------------------------------------- R1 to each
tl1(A, Out1) :- true | dup(Mid, Oy),
        /*Oz1=[A|Oz2],*/merge([], Oy, Oz2),
        pl1([A|Oz2], Out1), plus1([A|Out1], Mid).
tl1(A, Out1) :- A1:=A+1 |
        dup([A1], Oy), merge([A], Oy, Oz1),
        pl1(Oz1, Out1)./*Mid=[A1].*/
--------------------------------------------------------------------- R2 to each
tl1(A, Out) :- true |
        dup(Mid, Oy), merge([], Oy, Oz2),
        /*pl1([A|Oz2], Out1),*/Out1=[A],
        plus1([A|Out1], Mid).
tl1(A, Out1) :- A1:=A+1 |
        /*dup([A1], Oy),*/Oy=[A1, A1],
        merge([A], Oy, Oz1), pl1(Oz1, Out1).
--------------------------------------------------------------------- R1 to each
tl1(A, Out1) :- true |
        dup(Mid, Oy), merge([], Oy, Oz2),
        Out1=[A], plus1([A, A], Mid).
tl1(A, Out1) :- A:=A+1 |
        /*Oy=[A1, A1],*/
        merge([A], [A1, A1], Oz1), pl1(Oz1, Out1).
--------------------------------------------------------------------- R3 to the 1st
tl1(A, Out1) :- true | tl2(A), Out1=[A].
```

```
t11(A, Out1) :- A1:=A+1 |
        merge([A], [A1, A1], Oz1), p11(Oz1, Out1).
t12(A) :- true |
        dup(Mid, Oy), merge([], Oy, Oz2), plus1([A, A], Mid).
------------------------------------------------------------------- R4
t12(A) :- true | dup(Mid, Oy),
        /*merge([], Oy, Oz2),*/Oz2=Oy,
        plus1([A, A], Mid).
t12(A) :- A1:=A+1 |
        dup(Mid, Oy), merge([], Oy, Oz2),
        /*plus1([A, A], Mid).*/Mid=[A1].
------------------------------------------------------------------- R1 to each
t12(A) :- true | dup(Mid, Oy),
        /*Oz2=Oy,*/plus1([A, A], Mid).
t12(A) :- A1:=A+1 |
        dup([A1], Oy), merge([], Oy, Oz2).
        /*Mid=[A1].*/
------------------------------------------------------------------- R4 to the 1st
t12(A) :- A1:=A+1 | dup(Mid, Oy),
        /*plus1([A, A], Mid).*/Mid=[A1].
t12(A) :- A1:=A+1 |
        dup([A1], Oy), merge([], Oy, Oz2).
------------------------------------------------------------------- R1 to the 1st
t12(A) :- A1=A:+1 |
        dup([A1], Oy)./*Mid=[A1].*/
t12(A) :- A1:=A+1 |
        dup([A1], Oy), merge([], Oy, Oz2).
------------------------------------------------------------------- R2 to each
t12(A) :- A1:=A+1 |
        /*dup([A1], Oy).*/Oy=[A1, A1].
t12(A) :- A1 :=A+1 |
        /*dup([A1], Oy).*|/Oy=[A1, A1],
        merge([], Oy, Oz2).
------------------------------------------------------------------- R1 to each
t12(A) :- A1:=A+1 | true.
        /*Oy=[A1, A1].*/
t12(A) :- A1:=A+1 |
        /*Oy=[A1, A1],*/merge([], [A1, A1], Oz2).
-------------------------------------------------------------------
```

R2 to the 2nd at merge repeatedly, and unification for Oz2 is eliminated.

```
-------------------------------------------------------------------
t12(A) :- A1:=A+1 | true.
t12(A) :- A1:=A+1 | /*merge([], [A1, A1], Oz2).*/true.
```

```
---------------------------------------------------------------------
--------------------------------------------------------------------- recall t11
t11(A, Out1) :- true | t12(A), Out1=[A].
t11(A, Out1) :- A1:=A+1 | merge([A], [A1, A1], Oz1), p11(Oz1, Out1).
---------------------------------------------------------------------
```

R2 to the 2nd at merge repeatedly.

```
---------------------------------------------------------------------
t11(A, Out1) :- true | t12(A), Out1=[A].
t11(A, Out1) :- A1:=A+1 | /*merge([A], [A1, A1], Oz1),*/
        Oz1=[A, A1, A1], p11(Oz1, Out1).
t11(A, Out1) :- A1:=A+1 | /*merge([A], [A1, A1], Oz1),*/
        Oz1=[A1, A, A1], p11(Oz1, Out1).
t11(A, Out) :- A1:=A+1 | /*merge([A], [A1, A1], Oz1),*/
        Oz1=[A1, A1, A], p11(Oz1, Out1).
--------------------------------------------------------------------- R1 to 2-4th
t11(A, Out1) :- true | t12(A), Out1=[A].
t11(A, Out1) :- A1:=A+1 | /*Oz1=[A, A1, A1],*/p11(A, A1, A1], Out1).
t11(A, Out1) :- A1:=A+1 | /*Oz1=[A1, A, A1],*/p11([A1, A, A1], Out1).
t11(A, Out1) :- A1:=A+1 | /*Oz1=[A1, A1, A],*/p11([A1, A1, A], Out).
--------------------------------------------------------------------- R2 to 2-4th
t11(A, Out1) :- true | t12(A), Out1=[A].
t11(A, Out1) :- A1:=A+1 | /*p11([A, A1, A1], Out1).*/Out1=[A].
t11(A, Out1) :- A1:=A+1 | /*p11([A1, A, A1], Out1).*/Out1=[A1].
t11(A, Out1) :- A1:=A+1 | /*p11([A1, A1, A], Out1).*/Out1=[A1].
---------------------------------------------------------------------
```

Thus the program of the case 1 is as following.

```
---------------------------------------------------------------------
t1([A|_], Out) :- true | t11(A, Out1), Out=[A|Out1].
t11(A, Out1) :- true | t12(A), Out1=[A].
t11(A, Out1) :- A1:=A+1 | Out1=[A].
t11(A, Out1) :- A1:=A+1 | Out1=[A1].
t12(A) :- A1:=A+1 | true.
--------------------------------------------------------------------- [case 1 end]
```

i→2(from ⟨α⟩)

```
---------------------------------------------------------------------
t2([A|_], Out) :- true |
        dup(Mid, Oy), merge([A], Oy, Oz1),
        p2([A|_Oz1], Out), plus1(Out, Mid).
--------------------------------------------------------------------- R4
t2([A|_], Out) :- true | dup(Mid, Oy),
        /*merge([A], Oy, Oz1),*/Oz1=[A|Oz2], merge([], Oy, Oz2),
        p2([A|Oz1], Out), plus1(Out, Mid).
```

```
----------------------------------------------------------------------- R1
t2([A|_], Out) :- true | dup(Mid, Oy),
        /*Oz1=[A|Oz2],*/merge([], Oy, Oz2),
        p2([A, A|Oz2], Out), plus1(Out, Mid).
----------------------------------------------------------------------- R2
t2([A|_], Out) :- true | dup(Mid, Oy), merge([], Oy, Oz2),
        /*p2([A, A|Oz2], Out),*/Out=[A, A],
        plus1(Out, Mid).
----------------------------------------------------------------------- R1
t2([A|_], Out) :- true | dup(Mid, Oy), merge([], Oy, Oz2),
        Out=[A, A], plus1([A, A], Mid).
----------------------------------------------------------------------- R3
t2([A|_], Out) :- true | t21(A), Out=[A, A].
t21(A) :- true |
        dup(Mid, Oy), merge([], Oy, Oz2), plus1([A, A], Mid).
----------------------------------------------------------------------- R4
t21(A) :- true | dup(Mid, Oy),
        /*merge([], Oy, Oz2),*/Oz2=Oy,
        plus1([A, A], Mid).
t21(A) :- A1:=A+1 | dup(Mid, Oy), merge([], Oy, Oz2),
        /*plus1([A, A], Mid).*/Mid=[A1].
----------------------------------------------------------------------- R1 to each
t21(A) :- true | dup(Mid, Oy),
        /*Oz2=Oy,*/plus1([A, A], Mid).
t21(A) :- A1:=A+1 | dup([A1], Oy), merge([], Oy, Oz2).
        /*Mid=[A1].*/
----------------------------------------------------------------------- R2 to 2nd
t21(A) :- true | dup(Mid, Oy), plus1([A, A], Mid).
t21(A) :- A1:=A+1 | /*dup([A1], Oy),*/
        Oy=[A1, A1], merge([], Oy, Oz2).
----------------------------------------------------------------------- R1 to 2nd
t21(A) :- true | dup(Mid, Oy), plus1([A, A], Mid).
t21(A) :- A1:=A+1 |
        /*Oy=[A1, A1],*/merge([], [A1, A1], Oz2).
-----------------------------------------------------------------------
R2 to the 2nd at merge repeatedly.
-----------------------------------------------------------------------
t21(A) :- true | dup(Mid, Oy), plus1([A, A], Mid).
t21(A) :- A1:=A+1 |
        /*merge([], [A1, A1], Oz2).*/Oz2=[A1, A1].
----------------------------------------------------------------------- R4 to the 1st
t21(A) :- A1:=A+1 | dup(Mid, Oy),
        /*plus1([A, A], Mid).*/Mid=[A1].
```

```
t21(A) :- A1:=A+1 | Oz2=[A1, A1].
---------------------------------------------------------------- R1 to each
t21(A) :- A1:=A+1 | dup([A1], Oy)./*Mid=[A1].*/
t21(A) :- A1:=A+1 | /*Oz2=[A1, A1].*/true.
---------------------------------------------------------------- R2 to the 2nd
t21(A) :- A1:=A+1 | /*dup([A1], Oy).*/Oy=[A1, A1].
t21(A) :- A1:=A+1 | true.
---------------------------------------------------------------- R1 to the 2nd
t21(A) :- A1:=A+1 | /*Oy=[A1, A1].*/true.
t21(A) :- A1:=A+1 | true.
----------------------------------------------------------------
```

Thus the program of the case 2 is as following.

```
----------------------------------------------------------------
t2([A|_], Out) :- true | t21(A), Out=[A, A].
t21(A) :- A1:=A+1 | true.
---------------------------------------------------------------- [case 2 end]
```

§5 Conclusion

This paper presented a set of rules, called UR-set, for the transformation of GHC programs. It seems to be powerful enough for many applications. To evaluate its efficiency, we need to perform further experiments such as process fusion, leveling of a metainterpreter and its object program, or program synthesis from naive definition.

Recently, we found a problem related to the notion of "input related". It is now being solved based on the elaborate formalization analyzing the direction of unification. This formalization also allows us to leave the input/output modes of variables unspecified.[7]

To realize an automatic partial evaluation system, we must find a valid control strategy to apply UR-set. We are interested in implementing such a system in GHC. We believe it will take the form of cooperation of several unfolding processes.

Acknowledgements

We would like to express special thanks to Yuji Matsumoto. This study owed a great deal to his detailed investigation. We would also like to thank Kazunori Ueda and Akikazu Takeuchi for their helpful discussion.

References

1) Brock, J. D. and Ackerman, W. B., "Scenario: A Model of Nondeterminate Computation," in *Formalization of Programming Concepts* (J. Diaz and I. Ramos, ed.), *Lecture Notes in Computer Science*, *Vol. 107*, Springer-Verlag, 1981.

2) Clark, K. L. and Gregory, S., "PARLOG: Parallel Programming in Logic," *ACM Trans. Program. Lang. Syst. 8, 1.*
3) Furukawa, K. and Ueda, K., "GHC Process Fusion by Program Transformation," *Proc. the Second Annual Conference of Japan Society of Softwate Science and Technology,* 1985.
4) Shapiro, E. Y., "A Subset of Concurrent Prolog and Its Interpreter," *ICOT Tech. Report, TR-003.*
5) Shapiro, E. Y., "Concurrent Prolog: A Progress Report," *IEEE Computer, Vol. 19, No. 8,* 1986.
6) Ueda, K., "Guarded Horn Clauses," *Proc. Logic Programming '85* (*Lecture Notes in Computer Science, Vol. 221*), Springer-Verlag, 1986.
7) Ueda, K., et al., "Transformation Rules for FGHC Programs," forthcoming, 1988.