ソフトウェア基礎論 21－6
(1987. 6. 26)

Unfolding Rules for GHC Programs

GHCプログラムのunfold変換について

Koichi Furukawa, Akira Okumura, and Masaki Murakami

古川康一、奥村 晃、村上昌己

Institute for New Generation Computer Technology

(財)新世代コンピュータ技術開発機構

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

あらまし 本論文ではGHC(Guarded Horn Clauses)プログラムを変換するためのunfoldを基にした変換規則の集合を与える。ここで与える規則集合(UR-set)は、到達し得る解の集合を保存し、かつ新たなデッドロックを導入しないことが示される。UR-setは、プロセス融合や部分計算などのGHCプログラムの種々のプログラム変換のための基礎を与えるものと思われる。

1. Introduction

It is expected that fruitful results will follow program transformation research in parallel logic languages such as GHC [Ue85], PARLOG [CG86] and Concurrent Prolog [Sh83]. 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 [Sh86] and program transformation to fuse two concurrent processes to increase efficiency [FU85]. 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 parts 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), by the goal, r(In,Out), the following clause is obtained. (The definition of unfolding used in this note is in the next section.)

```
(C1)' q(A,[B|In],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 or deadlock freedom 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 derived by goal substitution of the source program, mainly by unfolding. First, a term definition of unfolding is introduced for the following discussion.

Definition: *unfolding*

Consider clauses P and Q as

$P :: H_p \text{:-} G_p \mid B_p$
$Q :: H_q \text{:-} G_q \mid B_q$

where each of H_p and H_q is an atomic formula which has all distinct variables for its arguments, and each of G_p, B_p, G_q, and B_q is a sequence of goals. If there is a substitution, θ, which makes H_q the same as a goal, A, in B_p, *unfolding* clause P *at* goal A *by* clause Q is defined as to obtain a merged clause, R, which is

$R :: H_p \text{:-} G_p, G_q\theta \mid B_p', B_q\theta$

where B_p' is B_p without goal A.

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*

Explicit unifications (=) appearing in a body part of a clause, C, can be symbolically executed within the body part; that is, a further instantiated value can substitute corresponding variable occurrences within the body. Furthermore, if neither side of = includes an output variable of the clause, the unification goal can be 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) :- true | X = a, q(X))
        → (p(X) :- true | X = a, q(a))
(p :- true | X = a, q(X))
        → (p :- true | q(a))
```

Rule 2 *Unfolding at an Immediately Executable Goal*

Each of the clauses for a given goal may take any of the following forms.

satisfied	its guard is already satisfied.
candidate	its guard is not yet satisfied,but may be in future.
unsatisfiable	its guard is already known as not able to be satisfied.

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.

Let a clause, C, be of the form

$A \text{:-} G_1, G_2, \ldots, G_m \mid A_1, A_2, \ldots, A_n$

C can be unfolded at an immediately executable body goal, A_i, by all satisfied clauses, C_{ij} ($1 \leq j \leq l$; l is the number of satisfied clauses). The resulting clause,

D_{ij}, is obtained from the original clause, C, by replacing goal A_i by the body of C_{ij}. D_{ij} is a guarded resolvent of C and C_{ij} whose guard goals are the same as C, because the guards of C_{ij} must be true.) Thus, a new program is derived by replacing clause C of the original program by all of D_{ij}.

Example:

```
{(p :- true | q,a(1),r)}
       → {   (p :- true | q,b,c,r),
             (p :- true | q,d,e,r) }
by {   (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 :- $G_1,G_2,...,G_m \mid U_1,U_2,...,U_p,N_1,N_2,...,N_q$

where U_i ($0 \leqq i \leqq p$) are output unifications and N_j ($0 \leqq j \leqq q$) others. Furthermore, let the intersection of a set of variables appearing in $G_1,G_2,...,G_m,U_1,U_2,...,U_p$ and that appearing in $N_1,N_2,...,N_q$, be $X_1,X_2,...,X_r$. Then, a new clause, C_1, of newP is introduced as

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

Then, the sequence of N_j of C can be folded by C_1 and a transformed clause, C', is obtained as

P :- $G_1,G_2,...,G_m \mid U_1,U_2,...,U_p$,newP$(X_1,X_2,...,X_r)$.

Thus, a new program is derived by replacing clause C of the original program by C_1 and C'. This rule is used to transform clauses into forms where the next rule can be applied.

Example:

Clause
`(p(X,Y) :- X>0 | Y=[X|Z], q(Z), r)` is replaced by a pair of clauses as

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

Rule 4 *Unfolding across Guard*

If the guard condition of a clause, C, is true and there are no immediately executable goals in the body part, then C can be unfolded at the body goals which share input variables with the head all together. Let a clause, C, be

A :- true | $A_1,A_2,...,A_n,B_1,B_2,...,B_l$

where each A_i shares input variables with A, and each B_i does not. Let a clause of Ai be C_{ij} ($1 \leqq j \leqq m_i$; m_i is the number of clauses whose heads can be unified with A_i). Then, D_{ij}, the result of unfolding C at a goal, A_i, by C_{ij}, is a guarded resolvent of C and C_{ij}. The guard part of D_{ij} comes from the guard part of C_{ij}. Thus, a new program is derived by replacing clause C of the original program by all of D_{ij} ($1 \leqq i \leqq n$, $1 \leqq j \leqq m_i$).

Example:

`(p(X) :- true | q(X), r(X))` can be 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. Correctness of UR-set

A rule of transformation must provide some equivalence. We expect the following attributes between the original program, P, and a transformed one, 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, neither can it in P'.

The above attributes do not allow for cases of deadlock, failure, and infinite loops. However, we consider those programs as mistakes, and have made them out of consideration. This section gives a brief explanation showing each of UR-set provides those attribute.

Rule 1

This rule allows the execution of body unification in advance. However, it changes neither the head nor the guard part, so it has the same effect as the case where the unification is executed immediately after commitments. Consider a clause, C, in P, and its corresponding C' in P' as

```
C :: (p(X,Y) :- true | X=a, r(X,Y)),
C':: (p(X,Y) :- true | X=a, r(a,Y)),
```

where `X` and `Y` are output variables. The only difference between C and C' is whether the first argument of `r` is instantiated to `a` or not. a2) holds obviously. As to a1) and a3), every commitment under `r(X,Y)` is also possible for `r(a,Y)`, because an instantiation never prevents any commitment.

Rule 2

This rule substitutes a body goal, A, by goals which should be derived by resolution of A. It makes only such the reductions in advance as can be done eventually for A. It generates clauses for each possible resolvent and no clauses for impossible resolvents. It does not tighten any guard condition, and does not instantiate any variable out of goal A. Therefore, a1), a2), and a3) hold clearly.

Rule 3

An introduced clause of a new predicate has true guard, and a goal of that predicate can commit immediately. This means that the transformed program gives the same behavior as the original one except the excessive commitment. So a1), a2), and a3) hold clearly.

Rule 4

Re-examine the same example shown in the previous section.

C :: (`p(X,Y) :- true | q(X), r(Y)`) can be replaced by

```
C'::{  (p(X,Y) :- X>3 | q1, r(Y)),
       (p(X,Y) :- X≤3 | q2, r(Y)),
       (p(X,Y) :- Y<2 | q(X), r1),
       (p(X,Y) :- Y≥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)  }.
```

For `p(X,Y)` in C to have a solution, one of the guard conditions of `q(X)` or `r(Y)` must be true by an instantiation of X or Y. At that time, the guard condition of the corresponding clause of p in C' is true, and the converse is also true. Therefore, a1), a2), and a3) are satisfied.

4. Brock-Ackerman Problem

This section presents an example of the application of the proposed set of rules, called the Brock-Ackerman Problem [BA81]. Consider the program 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) :- true |
        A1 := A+1, Out = [A1].
```

The clause of p_2 could be derived, if we would allow to unfold the clause of p_1 at `p11(In,Out)` by the clause of p_{11}. s_1 and s_2 have the same set of solutions. So, in this sense, that unfolding is correct. However, t_1 and t_2 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 any context. However, the rules provide fair transformations in certain contexts.

If we start with the clause of t_i with a mode declaration as `ti(+,-)`, then contexts for p_i 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).
------------------------------------------------------ Rule2
ti(In,Out) :- true | ~~si(In,Mid,Out),~~
   dup(In,Ox), dup(Mid,Oy), merge(Ox,Oy,Oz),
   pi(Oz,Out), plus1(Out,Mid).
------------------------------------------------------ Rule4
ti([A|_],Out) :- true | ~~dup(In,Ox),~~
   Ox=[A,A], dup(Mid,Oy), merge(Ox,Oy,Oz),
   pi(Oz,Out), plus1(Out,Mid).
------------------------------------------------------ Rule1
ti([A|_],Out) :- true | ~~Ox=[A,A],~~
   dup(Mid,Oy), merge([A,A],Oy,Oz),
   pi(Oz,Out), plus1(Out,Mid).
------------------------------------------------------ Rule2
ti([A|_],Out) :- true | ~~merge([A,A],Oy,Oz),~~
   dup(Mid,Oy), merge([A],Oy,Oz1),
   Oz=[A|Oz1], pi(Oz,Out), plus1(Out,Mid).
------------------------------------------------------ Rule1
ti([A|_],Out) :- true | ~~Oz=[A|Oz1],~~
   dup(Mid,Oy), merge([A],Oy,Oz1),
   pi([A|Oz1],Out), plus1(Out,Mid).
------------------------------------------------------ <α>
```

In the case of i = 1

```
-------------------------------------------------- i → 1
t1([A|_],Out) :- true |
   dup(Mid,Oy), merge([A],Oy,Oz1),
   p1([A|Oz1],Out), plus1(Out,Mid).
-------------------------------------------------- Rule2
t1([A|_],Out) :- true | ~~p1([A|Oz1],Out),~~
   dup(Mid,Oy), merge([A],Oy,Oz1),
   Out=[A|Out1], p11(Oz1,Out1),
   plus1(Out,Mid).
-------------------------------------------------- Rule1
t1([A|_],Out) :- true |
   dup(Mid,Oy), merge([A],Oy,Oz1),
   Out=[A|Out1], p11(Oz1,Out1),
   plus1([A|Out1],Mid).
-------------------------------------------------- Rule2
t1([A|_],Out) :- true |
   dup(Mid,Oy), merge([A],Oy,Oz1),
   Out=[A|Out1], p11(Oz1,Out1),
   ~~plus1([A|Out1],Mid),~~ A1:=A+1, Mid=[A1].
-------------------------------------------------- Rule1
t1([A|_],Out) :- true |
   dup([A1],Oy), merge([A],Oy,Oz1),
   Out=[A|Out1], p11(Oz1,Out1),
   A1:=A+1 ~~,Mid=[A1]~~.
-------------------------------------------------- Rule2
t1([A|_],Out) :- true | ~~dup([A1],Oy),~~
   Oy=[A1,A1], merge([A],Oy,Oz1),
   Out=[A|Out1], p11(Oz1,Out1), A1:=A+1.
-------------------------------------------------- Rule1
t1([A|_],Out) :- true | ~~Oy=[A1,A1],~~
   merge([A],[A1,A1],Oz1), Out=[A|Out1],
   p11(Oz1,Out1), A1:=A+1.
-------------------------------------------------- Rule2
t1([A|_],Out) :- true |
   ~~merge([A],[A1,A1],Oz1),~~
   Oz1=[A|Oz2], merge([],[A1,A1],Oz2),
   Out=[A|Out1], p11(Oz1,Out1), A1:=A+1.
t1([A|_],Out) :- true |
   ~~merge([A],[A1,A1],Oz1),~~
   Oz1=[A1|Oz2], merge([A],[A1],Oz2),
   Out=[A|Out1], p11(Oz1,Out1), A1:=A+1.
---------------------------------------- Rule1 to each
t1([A|_],Out) :- true | ~~Oz1=[A|Oz2],~~
   merge([],[A1,A1],Oz2), Out=[A|Out1],
   p11([A|Oz2],Out1), A1:=A+1.
t1([A|_],Out) :- true | ~~Oz1=[A1|Oz2],~~
   merge([A],[A1],Oz2), Out=[A|Out1],
   p11([A1|Oz2],Out1), A1:=A+1.
---------------------------------------- Rule2 to each
t1([A|_],Out) :- true | ~~p11([A|Oz2],Out1),~~
   merge([],[A1,A1],Oz2), Out=[A|Out1],
   Out1=[A], A1:=A+1.
t1([A|_],Out) :- true | ~~p11([A1|Oz2],Out1),~~
   merge([A],[A1],Oz2), Out=[A|Out1],
   Out1=[A1], A1:=A+1.
--------------------------------------------------
```

Each merge/3 is transformed to some unifications and is eliminated. So the final result is as follows.

```
--------------------------------------------------
t1([A|_],Out) :- true |
   ~~merge([],[A1,A1],Oz2), A1:=A+1,~~
   Out=[A|Out1], Out1=[A].
t1([A|_],Out) :- true | ~~merge([A],[A1],Oz2),~~
   Out=[A|Out1], Out1=[A1], A1:=A+1.
-------------------------------------------------- Rule1
t1([A|_],Out) :- true | ~~Out1=[A],~~
   Out=[A,A].
t1([A|_],Out) :- true | ~~Out1=[A1],~~
   Out=[A,A1], A1:=A+1.
-------------------------------------------------- [t1 end]
```

In the case of i = 2 (from <α>)

```
-------------------------------------------------- i → 2
t2([A|_],Out) :- true |
   dup(Mid,Oy), merge([A],Oy,Oz1),
   p2([A|Oz1],Out), plus1(Out,Mid).
-------------------------------------------------- Rule2
t2([A|_],Out) :- true | ~~merge([A],Oy,Oz1),~~
   dup(Mid,Oy), Oz1=[A|Oz2],
   merge([],Oy,Oz2),
   p2([A|Oz1],Out), plus1(Out,Mid).
-------------------------------------------------- Rule1
t2([A|_],Out) :- true | ~~Oz1=[A|Oz2],~~
   dup(Mid,Oy), merge([],Oy,Oz2),
   p2([A,A|Oz2],Out), plus1(Out,Mid).
-------------------------------------------------- Rule2
t2([A|_],Out) :- true | ~~p2([A,A|Oz2],Out),~~
   dup(Mid,Oy), merge([],Oy,Oz2), Out=[A,A],
   plus1(Out,Mid).
--------------------------------------------------
```

dup/2, merge/3, and plus1/2 are transformed to some unifications and are eliminated. So the final result is as follows.

```
--------------------------------------------------
t2([A|_],Out) :- true |
   ~~dup(Mid,Oy), merge([],Oy,Oz2),~~
   Out=[A,A]~~,plus1(Out,Mid)~~.
-------------------------------------------------- [t2 end]
```

Thus, the clauses of ti are partially evaluated to single unifications. They implement the success sets of each program directly.

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 various applications. To evaluate its efficiency, we need to perform further experiments such as process fusion, leveling of a meta-interpreter and its object program, or program synthesis from naive definition.

UR-set has a difficult problem. To judge whether a goal is immediately executable, we must confirm that no more instantiation can occur to the variables which allow the goal to be committed to another clause. If the goal has an input variable from another goal, we must know whether an instantiation for that variable can occur before the commitment. We are now considering a rather tight condition, which is to check that no other goals related to input variables of the object goal can ever commit before that goal. It can be derived from mode analysis.

To achieve 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.

Acknowledgment

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

[BA81] J. D. Brock and W. B. Ackerman: "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.

[CG86] K. L. Clark and S. Gregory: "PARLOG: Parallel Programming in Logic," ACM Trans. Program. Lang. Syst. 8, 1.

[FU85] K. Furukawa and K. Ueda: "GHC Process Fusion by Program Transformation," in Proc. the Second Annual Conference of Japan Society of Software Science and Technology, 1985.

[Sh83] E. Y. Shapiro: "A Subset of Concurrent Prolog and Its Inter- preter," ICOT Tech. Report TR-003.

[Sh86] E. Y. Shapiro: "Concurrent Prolog: A Progress Report," IEEE Computer 19(8), 1986.

[Ue85] K. Ueda: "Guarded Horn Clauses," in Proc. Logic Programming '85, (Lecture Notes in Computer Science, Vol.221), Springer-Verlag, 1986.

盛光印刷所