ソフトウェア基礎論 37－6
プログラミング言語 27－6
(1990. 12. 13)

拡張された unification アルゴリズムに基づくフル GHC の操作的意味論

沈 涵　　　田中 二郎

富士通（株）国際情報社会科学研究所

あらまし：　並列論理型言語の意味論について、幾つかの手法がすでに提案されている。それらの手法は主にフラットな並列論理型言語に限定したものである。本論文では、拡張された代入（solved 代入、suspended 代入）と unification アルゴリズムに基づくフル GHC における操作的意味論を提案する。まず、variant form variable という概念、そして plus- 演算子、minus- 演算子を導入し、拡張された unification アルゴリズムを定義する。それによって、フル GHC における head ゴール, guard ゴールと body ゴールの unification を同一の枠組で簡単に扱えるようになった。最後に、拡張された unification アルゴリズムに基づいてフル GHC における操作的意味論を提案する。

An Operational Semantics for Full GHC Based on Extended Unification Algorithm

Han SHEN　　　Jiro TANAKA

International Institute for Advanced Study of Social Information Science, FUJITSU LIMITED

17-25, SHINKAMATA, 1-CHOME,OTA-KU, TOKYO 144, JAPAN

Abstract:　Though several approaches have already been proposed for the semantics of concurrent logic programming languages, they mainly concern for the flat subset of the languages, where only the system-defined predicates appear in the guard part. This paper proposes an operational semantics for full GHC. Our approach is based on the extended substitution involving *solved substitution* and *suspended substitution*, and the extended unification algorithm. We introduce the concept of *variant form variables* and define two operations of *plus-operation* and *minus-operation*. Two important notions of solved substitution and suspended substitution are also defined. Then we describe an *extended unification algorithm* which can deal with GHC head-unification, guard-unification and body-unification in a uniform way. Based on these concept, operations, notions and algorithm, we propose an operational semantics for full GHC.

1 Introduction

Various kinds of *concurrent logic programming languages* have been proposed so far[Shapiro 89]. PARLOG [Clark 85], Concurrent Prolog [Shapiro 83] and GHC [Ueda 85] are examples of such languages. Though there are differences, these languages are very similar to each other. Horn clauses with guards are used for defining predicates, goals can be executed concurrently, and they have a synchronization mechanism between goals.

Logic programming is often claimed that it is well-founded in semantics [Lloyd 84]. However, this was not true for concurrent logic programming languages. It seems that the language features, such as *guard, commit, suspension* and *concurrency*, have prevented the development of semantics as a natural extension of Prolog semantics.

Though several approaches have already been proposed[Ueda 90] [Levi 87] [Murakami 90a] [Gerth 88] [Murakami 90b][Maher 87], they mainly concern for the flat subset of the languages, where only the system-defined predicates appear in guard part.

In this paper, we propose an operational semantics for full GHC. Our approach is based on the extended substitution (involving *solved substitution* and *suspended substitution*) and the extended unification algorithm. In the following section, we introduce the concept of *variant form variables* and extend the notions of *term, GHC clause, program* and *goal* to include *variant form variables.* In Section 3, we define two operations, i.e., plus-operation and minus-operation, on these extended structures. We also give two important notions of *solved substitution* and *suspended substitution*, then describe an extended unification algorithm which can deal with head-unification, guard-unification and body-unification in a uniform way. In section 4, we propose an operational semantics for full GHC, based on transition relation of states by using the extended unification algorithm. Finally, we would like to state some concluding remarks.

The paper assumes a basic knowledge of concurrent logic programming language GHC[Ueda 85] and some semantics aspects of Prolog[Lloyd 84].

2 Preliminary Definitions

This section gives some preliminary terminologies and definitions used in the whole paper.

We use $VARS$ to denote a set of all the variables appearing in a system.

In addition to the variables belonging to $VARS$, we also introduce the concept of *variant form variables*, which is related to our operational semantics for full GHC defined later:

Definition 1 (Variant Form Variable)
Let $X \in VARS$. The variant form variables of x are defined as follows:

- *x^* is a variant form variable of x;*
- *if y is a variant form variable of x, then y^* is also a variant form variable of x, and y^* is a variant form variable of y.* ■

For example, assume a variable $x \in VARS$, x^*, x^{**}, x^{***} are called the variant form variables of x, and x^{***} can be called a variant form variable of x, x^* or x^{**}. So there exist many variant form variables corresponding to a single variable or variant form variable. A variant form variable may be both a variant form variable of a variable x in $VARS$ and several other variant form variables of x.

Definition 2 (Original Form Variable)
x is said to be an original form of y if $x \in VARS$ and y is a variant form variable of x. ■

For example: Let $x \in VARS$. x^*, x^{**}, x^{***}, ..., have the only one original form variable x.

Variant form variables and original form variables are used in GHC operational semantics defined later. With these concepts, we can easily extend unification algorithm of pure logic programming into GHC, and can easily deal with head-unification, guard-unification and body-unification in a uniform frame. In a unification process or some phase of a computation process, a variant form variable is treated as a constant. Number of $*$ marked in an original form variable records such a phase.

All the variant form variables with their original form variables in $VARS$ constitutes a set, here we use $VARS^*$ to denote it.

Furtherly, the predicate symbols occurring in a system are partitioned into two sets:

- P_S, which contains all the system-defined predicate symbols;
- P_U, which contains all the user-defined predicate symbols.

We stipulate $PRED =_{df} P_S \cup P_U$, which contains all the predicate symbols appearing in a system.

Next, We define the concept of *term* which is the same one as in pure logic programming except our definition allows variant form variables:

Definition 3 (Term)
A term is defined inductively as:

(a) a is a term, where a is a constant;

(b) x is term, where $x \in (VARS \cup VARS^*)$;

(c) $f(t_1 \dots t_n)$ is a term, where f is an n-ary function symbol and t_i are terms $(i = 1, \dots, n)$. ■

Furtherly, we denote the set of all the terms containing $x \in VARS^*$ by $TERMS^*$, and the set of all the terms containing no $x \in VARS^*$ by $TERMS$.

For convenience, we use $\hat{t}$ to stand for an n-tuple of terms $t_1, \dots, t_n$, and $p(\hat{t})$ to stand for an n-ary literal, where $p \in PRED$ and $\hat{t}$ is an n-tuple of terms. All of the literals is denoted by $LITERALS$. Moreover, $t \equiv s$ is assumed to express that term t is syntactically equal to term s.

Definition 4 (GHC clause)
Let $p \in P_U$, $q_i, r_j \in PRED$, $\hat{t}$, $\hat{t}_i$, $\hat{s}_j$ are tuples of terms $(i = 1, \dots, m,\ j = 1, \dots, n)$.

$$p(\hat{t}) \leftarrow q_1(\hat{t}_1), \dots, q_m(\hat{t}_m) | r_1(\hat{s}_1), \dots, r_n(\hat{s}_n)$$

is referred to as a user-defined GHC clause C. $p(\hat{t})$ is called the head of C, the conjunction of the literals $q_1(\hat{t}_1), \dots, q_m(\hat{t}_m)$ is called the guard part *of C, and the conjunction of the literals $r_1(\hat{s}_1), \dots, r_n(\hat{s}_n)$ is called the* body part *of C. The operator "|" is called a commitment operator.* ■

Definition 5 (Program)
A finite set of GHC clauses is called a program. ■

This paper assumes that variable sets occurring in different GHC clauses are disjoint.

Finally, a goal in our scheme is defined as:

Definition 6 (Goal:)

$$\leftarrow p_1(\hat{t}_1), \dots, p_m(\hat{t}_m)$$

, where $p_i \in PRED$, $\hat{t}_i$ are tuples of terms $(i = 1, \dots, m)$. ■

The above notions of GHC clause, program and goal are almost the same as those defined in [Ueda 85], except that our notions allow variant form variables to appear in a term.

3 An Extended Unification Algorithm for Full GHC

We extend unification algorithm of pure logic programming in order to deal with GHC head-unification, guard-unification and body-unification in a uniform way.

In the previous section, the concept of variant form variables is introduced to annotate variables which can not be instantiated by unification in some phrase of a computing process. In this section, we define *plus-operation* and *minus-operation* which are used to make some adjustments in a phrase of a computing process (see the definition of operational semantics), and then give two important notions of *solved substitution* and *suspended substitution*. Based on them, we give an extended unification algorithm applicable for GHC head-unification, guard-unification and body-unification.

First, we need some definitions of *pair set, primitive term-pair set, solved substitution* and *suspended substitution*:

Definition 7 (Pair Set)
A finite set of the following form:

$$S = \{< t_1, s_1 > \ \dots \ < t_m, s_m >\}$$

is called a pair set, *where $t_i, s_i \in (TERMS \cup TERMS^* \cup LITERALS)$ $(i = 1, \dots, m)$.* ■

Definition 8 (Primitive Term-Pair Set)
A finite set of the following form:

$$S = \{< t_1, x_1 > \ \dots \ < t_m, x_m >\}$$

is called a primitive term-pair set, *where $x_i \in (VARS \cup VARS^*)$, $t_i \in (TERMS \cup TERMS^*)$, $x_i \notin t_i$, $x_i \not\equiv x_j$ $(i \neq j)$, $(i = 1, \dots, m,\ j = 1, \dots, m)$.* ■

Definition 9 (Solved Substitution)
A primitive term-pair set which furtherly satisfies the following condition: $x_i \in VARS$ $(i = 1, \dots, m)$ is referred to as a solved substitution. ■

Definition 10 (Suspended Substitution)
A primitive term-pair set is called a suspended substitution *if $\exists x_l \in VARS^*$ $(1 \leq l \leq m)$.* ■

From the above definitions, it is clearly that a primitive term-pair set is either a solved substitution or a suspended substitution.

Solved substitutions are just what we try to obtain in a computing process, while suspended ones correspond to suspended actions, resulting from GHC head-unification or guard-unification in a direct or indirect way (In our extended unification algorithm shown later, a suspension happens when trying to instantiate some variant form variables). This paper uses θ to denote a solved substitution and $\theta^{(s)}$ to annotate a suspended substitution.

Next, we define two operations: plus-operation $^+$ and minus-operation $^-$ to be used in the extended unification algorithm and commitment operation respectively, which are related to our operational semantics. The plus-operation is defined on a set of literals, and the minus-operation is defined on a union set of conjunctions of literals and primitive-term pair sets.

Definition 11 (plus-operation $^+$)

- $^+(c) =_{df} c$ *, where c is a constant;*
- $^+(x) =_{df} x^*$*, where* $x \in (VARS \cup VARS^*)$*;*
- $^+(f(t_1, ..., t_n)) =_{df} f(^+(t_1), ..., ^+(t_n))$, *where f is an n-ary functor symbol and* $t_i \in (TERMS^* \cup TERMS)$ $(i = 1, ..., n)$*;*
- $^+(p(t_1 \ ... \ t_m)) =_{df} p(^+(t_1) \ ... \ ^+(t_m))$, *where* $p \in PRED$ *and* $t_i \in (TERMS^* \cup TERMS)$ $(i = 1, ..., m)$. ■

Definition 12 (minus-operation $^-$)

- $^-(c) =_{df} c$ *, where c is a constant;*
- $^-(x^*) =_{df} x$*, where* $x \in (VAR \cup VARS^*)$*;*
- $^-(x) =_{df} x$*, where* $x \in VARS$*;*
- $^-(f(t_1, ..., t_n)) =_{df} f(^-(t_1), ..., ^-(t_n))$, *where f is an n-ary functor symbol and* $t_i \in (TERMS^* \cup TERMS)$ $(i = 1, ..., n)$*;*
- $^-(p(t_1 \ ... \ t_m)) =_{df} p(^-(t_1) \ ... \ ^-(t_m))$, *where* $p \in PRED$ *and* $t_i \in (TERMS^* \cup TERMS)$ $(i = 1, ..., m)$*;*
- $^-(L_1, ..., L_n) =_{df} {}^-(L_1), ..., ^-(L_n)$, *where* $L_i \in LITERALS$ $(i = 1, ..., n)$*;*
- $^-(\{< t_1, s_1 > \ ... \ < t_m, s_m >\}) = \{<^-(t_1), ^-(s_1) > \ ... \ <^-(t_m), ^-(s_m) >\}$, *where* $t_i, s_i \in (TERMS \cup TERMS^*)$ $(i = 1, ..., m)$. ■

Other concepts such as most general unifier (mgu), renaming equivalence and answer substitution are very similar to those defined in pure logic programming [Lloyd 84] [Lass 88], we omit their definitions in this paper.

Based on the concept of variant form variables, the plus-operation and the minus-operation, together with the notions of solved substitution and suspended substitution, we can easily define an extended unification algorithm for the computation of a most general unifier of a given pair set in a uniform frame:

Extended Unification Algorithm

Let S be a pair set. Repeatedly choose a pair in S of the following form and perform the corresponding action until it terminates with failure or nothing can be done for S furtherly.

1. $< x, x >$ (or $< c, c >$), where $x \in (VARS \cup VARS^*)$ (or c is a constant): delete the pair from S;
2. $< c_1, c_2 >$, where c_1, c_2 are constants and $c_1 \not\equiv c_2$: terminate with failure;
3. $< c_1, x >$, where $x \in VAR^*$ and c_1 is a constant: if exists $< c_2, x > \in (S - \{< c_1, x >\})$, $c_1 \not\equiv c_2$, then terminate with failure;
4. $< f(u_1 \ ... \ u_m), g(v_1 \ ... \ v_n) >$, where f is a m-ary function symbol and g is an n-ary function symbol, and $u_i, v_j \in (TERMS^* \cup TERMS)$ $(i = 1, ..., m, \ j = 1, ..., n)$: if $f = g$ and $m = n$ then replace it by the pairs: $< u_1, v_1 >$,..., $< u_m, v_m >$ else terminate with failure;
5. $< p(u_1 \ ... \ u_m), q(v_1 \ ... \ v_n) >$, where $p, q \in P_U$ and $u_i, v_j \in (TERMS^* \cup TERMS)$ $(i = 1, ..., m, \ j = 1, ..., n)$: if $p = q$ and $m = n$ replace it by the pairs $< u_1, v_1 >$, ..., $< u_m, v_m >$ else terminate with failure;
6. $< x, \ t >$, where $x \in (VARS^* \cup VARS)$, $t \in (TERMS^* \cup TERMS)$ and $t \notin VARS$, $x \notin VARS^*$ or $t \notin VARS^*$: substitute the pair by $< t, x >$;
7. $< t, \ x >$, where $x \in VARS$, $t \in (TERMS^* \cup TERMS)$, $x \in (S - \{< t, x >\})$: if $x \in t$ then terminate with failure else replace x in the other pairs of S by t.

Finally, if there exists $< t, x > \in \theta$, where $x \in VARS^*$, $t \in (TERMS^* \cup TERMS)$, then rename S by $\theta^{(s)}$ (a suspended substitution), else rename S by θ (a solved substitution). ■

In this paper, the concept of variant form variables plays an important role in coping with the constraint upon GHC head-unification, guard-unification and body-unification in a uniform way. When trying to unify a user-defined goal G with the head H of a

GHC clause, we first apply a plus-operation to G (i.e., $^+(G)$), and then apply the above extended unification algorithm for the pair set $\{<^+ (G), H >\}$. Whenever a variable in G (i.e., a variant form variable in $^+(G)$) is attempted to be instantiated in a direct or indirect way, our algorithm yields a suspended substitution. A suspension can be explained as a correspondence to suspension action in real GHC implementation. For a pair set which may yield a suspension, our algorithm makes the greatest efforts of trying to compute a nearest result to the expected mgu.

It can be said that variables in a goal are finally instantiated only via system-defined predicate " $=$", or some system-defined computing predicates like "*sum*". Here we give the following main procedure of computing unification for a goal. It is composed of the following three parts:

Main Procedure of Unification

- For a user-defined goal G:
  firstly apply the plus-operation on G, then make a pair set S: $\{^+(G),\ H\}$ (here H is the head of a GHC clause which is tried to be unified with G). Furtherly, apply the extended unification algorithm to the pair set S;
- For a goal of the form: $G_1 = G_2$:
  Create a pair set $S = \{G_1, G_2\}$, and then apply the above extended unification algorithm to it;
- For a system-defined computing goal G:
  Simply compute G, and suppose a result, say c, is to be obtained, and then create a primitive term-pair set $< c, x >$ for G, here c is the constant which is a computing result of x, where $x \in (VARS \cup VARS^*)$. ■

Now, we give several simple examples to show how our unification mechanism works:

First, we show an example for a goal corresponding to system-defined computing predicate "*sum*":

Example 1

For a goal G_1: $\leftarrow sum(2\ 3\ v_1)$

$sum(x\ y\ z)$ *means to add* x *to* y *and set the result to* z, *so we can get a solved substitution for* G_1:

$$\theta_1 = \{< 5, v_1 >\}$$

■

Then, we show a unification procedure corresponding to the system defined predicate $=$:

Example 2

For a goal G_2: $\leftarrow v_2 = [v_1]$

$t_1 = t_2$ *means to unify* t_1 *and* t_2 *in a similar way as in pure logic programming, we can obtain the following solved substitution for* G_2:

$$\theta_2 = \{< [v_1], v_2 >\}$$

■

Following is an example corresponds to a user-defined predicate:

Example 3

For a goal G_3: $\leftarrow append(v_2\ [6]\ x)$

, *here "append" is defined in a usual way:*

$$C_1:\ append([x_1|x_2]\ x_3\ x_4) \leftarrow true \mid x_4 = [x_1|x_5], append(x_2\ x_3\ x_5)$$
$$C_2:\ append([\ ]\ x_6\ x_7) \leftarrow true \mid x_6 = x_7.$$

Apply the plus-operation to G_3, *we get:*

$$G_3' = {}^+(append(v_2\ [6]\ x)) = append(v_2^*\ \ [6]\ \ x^*)$$

Generate a pair set for G_3' *and the head of clause* C_1, *we get:*

$$S_1 = \{< G_3',\ append([x_1|x_2]\ \ x_3\ x_4) >\}$$

Apply the extended unification algorithm to it, we get the following suspended substitution:

$$\theta_{31}^{(s)} = \{\underline{< [x_1|x_2], v_2^* >}, < [6], x_3 >, < x^*, x_4 >\}$$

Similarly,
For the unification of G_3' *and the head of clause* C_2, *we also get a suspended substitution:*

$$\theta_{32}^{(s)} = \{\underline{< [\ ], v_2^* >}, < [6], x_6 >, < x^*, x_7 >\}$$

■

In our extended unification algorithm, the relationship among the several occurrences of a variable in different arguments of a predicate can still be maintained without the head unification constraint being violated:

Example 4

$$C:\ p(x\ x) \leftarrow true|true$$

and a goal G: $\leftarrow p(a\ y)$:
Using the extended unification algorithm for the pair set: $S = \{<^+ (p(a\ y)), p(x\ x) >\}$, *we get:*

$$\theta^{(s)} = \{< a, x >, \underline{< a, y^* >}\}$$

, *which is a suspended substitution.* ■

From the extended unification algorithm described above, it holds that:

Proposition 1

For a given pair set S, the extended unification algorithm can obtain a solved substitution which is a mgu of S, a suspended substitution towards a mgu of S, or terminates with failure in a finite time. ■

Furtherly, it is quite easy to define a concatenation rule with which concatenates two substitutions resulted in a computing process. It is needed by our operational semantics:

Definition 13 (Concatenation $\theta_1 \circ \theta_2$)

Let θ_1 be a solved substitution, θ_2 be a primitive term-pair set. We use $\theta_1 \circ \theta_2$ to express the concatenation of θ_1 and θ_2. It is computed by performing the following actions:

- *Firstly, for each $< t, x > \in \theta_2$, where $x \in VARS$, $t \in (TERMS^* \cup TERMS)$, repace all the variant form variables of x in θ_1 by t;*
- *Then apply the extended unification algorithm to the set of $\theta_1 \cup \theta_2$.* ■

4 An Operational Semantics for GHC

In this section, we propose an operational semantics for full GHC, by using transition relation of states based on extended unification algorithm.

Now, we define the concept of transition relation of states upon which our operational semantics is based.

First, we need to define the concept of substitution restricted by goal.

Definition 14 (Restricted Substitution)

Let θ be a primitive term-pair set, G be a goal, and $V(G)$ be the set of all the original form variables appearing in G. Restrict θ by G is defined as:

$$\theta|_G =_{df} \{< t, x > \mid < t, x > \in \theta \wedge x \in V(G)\}$$

■

Thanks to the concept of variant form variable, the plus-operation, the minus-operation and the extended unification algorithm, together with concatenation rule, the work of defining an operational semantics for full GHC becomes quite simple and intuitive.

Let *Goal* represent a set of all the goals, and *Sub* represent a set containing all the solved substitutions and suspended substitutions. We express a state by an element in $Sta = Goal * Sub$.

Definition 15 (Transition Relation)

A transition relation $(\longrightarrow, Sta, Sta)$ is a relation satisfying the following conditions:

Let $s_i, s_k \in Sta$, $s_i =< G_i, \theta_i >$ and $s_k =< G_k, \theta_k >$, here θ_i is a solved substitution.

1. G_i is a single goal: ie., $G_i \equiv q(\bar{t})$

1a. $<\longrightarrow, s_i, s_k >$ holds, where $s_k \equiv s_i$;

1b. q is the system-defined predicate " $=$", i.e., $G_i : G_{i1} = G_{i2}$: if exists a primitive term-pair set σ for the pair set: $\{< G_{i1}, G_{i2} >\}$, then $<\longrightarrow, s_i, s_k >$ holds, where $s_k =< [\,],\ \theta_i \circ \sigma >$;

1c. q is a system-defined computing predicate: if exists a primitive term-pair set σ corresponding to the computing result of G_i[1], then $<\longrightarrow, s_i, s_k >$ holds, where $s_k =< [\,], \theta_i \circ \sigma >$;

1d. $q \in P_U$ (user-defined predicate): if exists a suspended substitution σ for the pair set: $\{<^+(q),\ H\ >\}$, where H is the head of a clause C: $H \leftarrow Guard\,|Body$, then $<\longrightarrow, s_i, s_k >$ holds, where $s_k =< G_i, \theta_i \circ \sigma >$;
else if exists a solved substitution σ for the pair set: $\{<^+(q),\ H\ >\}$, where H is the head of a clause C: $H \leftarrow Guard\,|Body$, and furtherly if $< (Guard)\sigma,\ \sigma > \longrightarrow\ \dots \longrightarrow$ $< [\,],\ \sigma \circ \gamma >$, then $<\longrightarrow, s_i, s_k >$ holds, where $s_k =<^- ((Body)(\sigma \circ \gamma)),\ \theta_i \circ \sigma \circ \gamma >$.

2. G_i is composed of several subgoals, i.e., $G_i \equiv G_{i_1}, \dots, G_{i_m}$: if $< G_{i_j}, \theta_i > \longrightarrow < Q_{i_j},\ S_{i_j} >$ (S_{i_j} are primitive term-pair sets, $j = 1, \dots, m$), and exists k such that $< Q_{i_k}, S_{i_k} > \not\equiv < G_{i_k}, \theta_i >$ $(1 \leq k \leq m)$, then $<\longrightarrow, s_i, s_k >$ holds, where

$$s_k =< (Q_{i_1}, \dots, Q_{i_m})\Lambda(S_{i_1}|_{G_{i_1}}, \dots, S_{i_m}|_{G_{i_m}}), \Lambda(S_{i_1}|_{G_{i_1}}, \dots, S_{i_m}|_{G_{i_m}}) >.$$

$\Lambda(S_{i_1}|_{G_{i_1}}, \dots, S_{i_m}|_{G_{i_m}})$ is a solved substitution which is a result of the AND-combination of $S_{i_1}|_{G_{i_1}}, \dots, S_{i_m}|_{G_{i_m}}$. The AND-combination operator Λ is defined as follows:

Let S_i $(i = 1, \dots, n)$ be primitive term-pair sets, which correspond to subgoals $G_1, \dots, G_n$ in a goal: $\leftarrow G_1, \dots, G_n$. Initially, create a set: $S = S_1 \cup \dots \cup S_n$. The AND-combination $\Lambda(S_1, \dots, S_n)$ can be obtained by repeatedly performing the following actions:

- *For each $< t, x > \in S$ ($x \in VARS$, $t \in (TERMS^* \cup TERMS)$, replace all the variant form variables of x by t;*
- *Then apply the extended unification algorithm for S.* ■

[1] the element of σ has the form of $< c, x >$, $x \in (VAR \cup VAR^*)$, c is a constant

The AND-combination operator Λ is used to combine several solved substitutions and possibly suspended substitutions corresponding to concurrently executed subgoals of a goal. The computation for Λ not only embodies the consistency check but also models the synchronization ability of GHC as well.

Here we show a simple example of how to compute AND-combination:

Example 5
For a goal G:

$$\leftarrow sum(2\ 3\ v_1), v_2 = [v_1], append(v_2\ [6]\ x)$$

Assume $\theta_1, \theta_2, \theta_{31}^{(s)}$ to be the same as those in Example 1, Example 2 and Example 3 respectively, compute the AND-combination of $\theta_1, \theta_2, \theta_{31}^{(s)}$ corresponding to the concurrently executed results of the subgoals of G. First, creat a union set S_1:

$$S_1 = \{< 5, v_1 >, < [v_1], v_2 >, < [x_1|x_2], v_2^* > \\ < [6], x_3 >, < x^*, x_4 >\};$$

then it is transformed into:

$$S_1' = \{< 5, v_1 >, < [5], v_2 >, < [x_1|x_2], [5] > \\ < [6], x_3 >, < x^*, x_4 >\};$$

Apply the extended unification algorithm to S_1', we get:

$$R_1 = \{< 5, v_1 >, < [5], v_2 >, < [5], x_1 >, \\ < [\,], x_2 >, < [6], x_3 >, < x^*, x_4 >\};$$

Similarly, compute $\Lambda(\theta_1, \theta_2, \theta_3, \theta_{32}^{(s)})$, we get:

$$R_2 = \{< 5, v_1 >, < [5], v_2 >, \underline{< [\,], [5] >}, \\ < [6], x_6 >, < x^*, x_7 >\} = failure \ ■$$

By using AND-combination and our active strategy for suspended substitutions, sometimes it is possible to avoid some deadlock situation. For example: for a goal: $\leftarrow p(x), q(x)$ with the definitions of its subgoals:
$p(5) \leftarrow true|true. \quad q(6) \leftarrow true|true.$
two suspended substitutions θ_1, θ_2 are generated for the concurrently executions of $p(x)$ and $q(x)$:
$\theta_1^{(s)} = \{< 5, x^* >\}; \quad \theta_2^{(s)} = \{< 6, x^* >\}$,
and then $\Lambda(\theta_1^{(s)}, \theta_2^{(s)})$ yields a failure (by action(3) of the extended unification algorithm). So a deadlock is avoided by our active computation for suspended substitution and AND-combination.

Obviously, AND-combination is associative, i.e.

Proposition 2
$\Lambda(\theta_1, \theta_2, \theta_3) = \Lambda(\Lambda(\theta_1, \theta_2), \theta_3)) = \Lambda(\theta_1, \Lambda(\theta_2, \theta_3))$

Definition 16 (Operational Semantics)
Let G, G_i $(i = 1, ..., m)$ be goals, P be a program, ε be a null substitution, θ_i $(i = 1, ..., m)$ be primitive term-pair sets. We call $< G, \varepsilon > \rightarrow < G_1, \theta_1 > \rightarrow \ldots \rightarrow < G_m, \theta_m >$ an execution *of G with the remaining goal G_m and substitution θ_m. Furtherly,*

- *If $G_m = null$, i.e.,*
  $< G, \varepsilon > \rightarrow < G_1, \theta_1 > \rightarrow \ldots \rightarrow < [\,], \theta_m >$ then it is called a successful execution of G, and θ_m is referred to as an answer substitution of G;
- *If $G_m \neq null$,*
  If θ_m is a suspended substitution, then it is called a suspended execution. Furtherly, if all the executions starting from goal G are suspended executions, we say the execution of G yields a deadlock;

  else if there exists no transition relation for the state $< G_m, \theta_m >$ furtherly, then it is called a failure execution. ■

Notice that our operational semantics can also explain non-terminating programs by some meaningful executions of goals (see Example 8).

Now, we give several examples to show how our operational semantics works for full GHC in various cases:

Example 6

Clause C_1: $p(y) \leftarrow q(z\ y)\ |\ r(y\ z)$.
Clause C_2: $q(w_1\ w_2) \leftarrow true\ |w_1 = a, w_2 = b$.
Clause C_3: $r(v_1\ v_2) \leftarrow true\ |v_1 = b$.

and a goal G: $\leftarrow p(x)$.

For G and the head of clause C_1, we set:

$$S_1 = \{<^+ (p(x)),\ p(y) >= \{< p(x^*),\ p(y) >\}$$

Apply the extended unification algorithm to it, we get:
$\theta_1 = \{< x^*, y >\}$;

The clause $C_1\theta_1$ is:
$p(x^*) \leftarrow\ q(z\ x^*)\ |\ r(x^*\ z)$.

Similarly, use the same extended unification algorithm for its guard goal $q(z\ x^*)$ and the head of clause C_2, get:
$\theta_2 = \{< z^*, w_1 >, < x^{**}, w_2 >\}$;

The clause $C_2\theta_2$ is committed:
$q(z^*\ x^{**}) \leftarrow true\ |\ z^* = a,\ x^{**} = b$.

So make minus-operation on the body part of $C_2\theta_2$:
$^-(z^* = a,\ x^{**} = b)$, *get* $z = a,\ x^* = b$,
the suspended substitution corresponding to it is:
$\theta_3^{(s)} = \{< a, z >, \underline{< b, x^* >}\}$

, here a suspension happens. ■

Now let us see another example which does not generate suspension:

Example 7

Clause C_1: $p(y) \leftarrow q(z\ y) \mid r(y\ z)$.
Clause C_2: $q(w_1\ w_2) \leftarrow true \mid w_1 = a$.
Clause C_3: $r(v_1\ v_2) \leftarrow true \mid v_1 = b$.

and a goal G: $\leftarrow p(x)$.

We obtain a solved substitution for the unification of G *and the head of clause* C_1: $\theta_1 = \{< x^*, y >\}$;

The clause $C_1\theta_1$ *is:*
$p(x^*) \leftarrow q(z\ x^*) \mid r(x^*\ z)$.

We get a solved substitution θ_2 *for the unification of* $q(z\ x^*)$ *and the head of clause* C_2:
$\theta_2 = \{< z^*, w_1 >, < x^{**}, w_2 >\}$;

The clause $C_2\theta_2$ *is committed:*
$q(z^*\ x^{**}) \leftarrow true \mid z^* = a$

Make minus-operation on the body part of $C_2\theta_2$:
${}^-(z^* = a)$, *we get:* $z = a$. *So the corresponding solved substitution is:* $\theta_3 = \{< a, z >\}$.

The concatenation result of $\theta_2 \circ \theta_3$ *is:*
$\theta_{2\circ 3} = \{< a, w_1 >, < x^{**}, w_2 >, < a, z >\}$;

The concatenation result of θ_1 *and* $\theta_{2\circ 3}$ *is:*
$\theta_{1\circ 2\circ 3} =$
$\{< x^*, y >, < a, w_1 >, < x^{**}, w_2 >, < a, z >\}$;

The clause $C_1\theta_{1\circ 2\circ 3}$ *is committed:*
$p(x^*) \leftarrow q(a\ \ x^*) \mid r(x^*\ a)$.

Make minus-operation on its body part: ${}^-(r(x^*\ a))$, *we get:* $r(x\ a)$

Furtherly, we get θ_4 *for the unification of* $r(x\ a)$ *and the head of* C_3: $\theta_4 = \{< x^*, v_1 >, < a, v_2 >\}$;

The clause $C_3\theta_4$ *is committed:*
$r(x^*\ a) \leftarrow true \mid x^* = b$.

So make minus-operation on the its body part:
${}^-(x^* = b)$, *get:* $(x = b)$, *and the corresponding solved substitution is:* $\theta_5 = \{< b, x >\}$;

The concatenation of θ_4 *and* θ_5 *is:*
$\theta_{4\circ 5} = \{< b, v_1 >, < a, v_2 >, < b, x >\}$;

So the concatenation result of $\theta_{1\circ 2\circ 3}$ *and* $\theta_{4\circ 5}$ *is:*
$R = \theta_{1\circ 2\circ 3\circ 4\circ 5} =$
$\{< b, y >, < a, w_1 >, < b, w_2 >, < a, z >,$
$< b, v_1 >, < a, v_2 >, < b, x >\}$.

The answer substitution for the goal G *is:*
$R|_G = \{< b, x >\}$.

Different from the above example, this one doesn't yield a suspended substitution. In this case, clause C_2 *is used only to initiate the variable* z. *It doesn't initiate the variable* y, *so doesn't initiate the variable* x *in goal* $p(x)$ *via* y *of the guard part of clause* C_1 *either. Finally, the variable* x *is successfully initiated by the body part of clause* C_1 *after it is committed.* ■

Unlike pure logic programming only concerning about success and failure, our operational semantics for full GHC can also characterize some nonterminating program. Below is a demand-driven example of computing *Fibonacci number*, we show a meaningful execution by our operational semantics with some detailed transition relations omitted.

Example 8

$$\begin{aligned}
C_1 :\ & initialgoal \leftarrow true| \\
& \qquad fiboinitial(DFs), \\
& \qquad driver(DFs\ DOs). \\
C_2 :\ & fiboinitial(Ns) \leftarrow true| \\
& \qquad fibonacci(0\ 1\ Ns). \\
C_3 :\ & fibonacci(N_1\ N_2\ \ [N|Ns_1]) \leftarrow true| \\
& \qquad N = N_2, N_3 = N_1 + N_2, \\
& \qquad fibonacci(N_2\ N_3\ Ns_1). \\
C_4 :\ & driver(Fs\ IOs) \leftarrow true| \\
& \qquad IOs = [read(X)|IOs_1], \\
& \qquad check(Fs\ IOs_1\ X). \\
C_5 :\ & check(Fso\ IOso\ more) \leftarrow true| \\
& \qquad Fso = [N|Fs_1], \\
& \qquad IOso = [write(N)|IOs_2], \\
& \qquad driver(Fs_1\ IOs_2).
\end{aligned}$$

Although the program starting from initialgoal will never terminate, it can result in meaningful results from its executions. Here we simply show a meaningful execution using transition relation of states.

$< initialgoal, \varepsilon >$, *where* ε *is a null substitution.*
$\rightarrow < fiboinitial(DFs), driver(DFs\ DOs),\ \varepsilon >$
$\rightarrow < fibonacci(0\ 1\ DFs), driver(DFs\ DOs),\ \theta_1 >$
$\rightarrow < fibonacci(0\ 1\ DFs), check(DFs\ IOs_1\ X),\ \theta_2 >$
Assume end-user send an instruction of 'more', then:
$\rightarrow < fibonacci(0\ 1\ [N|Fs_1]), driver(Fs_1\ IOs_2),\ \theta_3 >$
$\rightarrow < fibonacci(1\ 1\ Fs_1]), driver(Fs_1\ IOs_2),\ \theta_4 >$,
where
$\theta_4 = \{< 0, N_1 >, < 1, N_2 >, < 1, N_3 >, < 1, N >,$
$< Fs_1, Ns_1 >, < [1|Fs_1], DFs >, < [1|Fs_1], Fso >,$
$< [1|Fs_1], Ns >, < [1|Fs_1], Fs >, < [read(more)$
$write(1)|IOs_2], IOs >, < [read(more)\ write(1)|$
$IOs_2], DOs >, < [write(1)|IOs_2], IOs_1 >,$
$< [write(1)|IOs_2], IOs_0 >, < more, X >\}$
This is a meaningful execution. ■

5 Conclusions and Final Remarks

After giving several preliminary definitions together with the concept of variant form variables, we have defined the plus-operation and the minus-operation, and also given two important notions of solved substitution and suspended substitution. Then we have described an extended unification algorithm which deals with GHC head-unification, guard-unification and body-unification in a uniform frame. We have proposed an operational semantics by using the transition relation of states in GHC based on extended unification algorithm.

There remains much work to be done, which is related to the semantics of meta and reflective concurrent logic programming languages[Suga 90] [Tanaka 88][Tanaka 90]. We would try to characterize some issues about the semantics for meta and reflective GHC. And prove some correctness of practical systems by using the proposed semantics.

6 Acknowledge

The authors would like to express our thanks to Hiroyasu SUGANO for his joining to the discussions on the draft version of this paper. The research underlying this paper has benefited from his thoughtful comments and valuable advices. We would also like to thank to Masaki MURAKAMI for his discussions and comments.

References

[Clark 85] K. Clark and S. Gregory: PARLOG, Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, Revised 1985.

[de Boer 89] Frank S. de Boer, Joost N.Kok, Catuscia Palamidessi and Jan J.M.M.Rutten: Semantic Models for a Version of PARLOG, Proceedings of the 6th International Conference on Logic Programming, pp. 621-636, J.-L.Lassez, Ed. MIT Press, Cambridge, Mass., 1989.

[Gerth 88] R. Gerth, M. Codish, Y. Lichtenstein and E. Shapiro: Fully Abstract Denotational Semantics for Flat Concurrent Prolog, Third Annual Symposium on Logic in Computer Science, pp. 320-335, Edinburgh, Scottland, IEEE, July 5-8, 1988.

[Lass 88] J.L. Lassez, M.J. Maher and K. Marriot: Unification Revisited, Foundations of Deductive Databases and Logic Programming (J.Minker,ed), pp.587-625, Morgan Kaufman Publishers, Los Altos, CA, 1988.

[Levi 87] G. Levi and C. Palamidessi: An Approach to the Declarative Semantics of Synchronization in Logic Languages, Logic Programming: Proc. of the Fourth International Conference, Vol. 2, pp. 877-893, The MIT Press, 1987.

[Lloyd 84] J.W. Lloyd: Foundations of Logic Programming, Springer-Verlag, 1984.

[Maher 87] M.J. Maher: Logic Semantics for a Class of Committed-Choice Programs, Proceedings of the 4th International Conference on Logic Programming, J.-L.Lassez, pp.858-876, Ed. MIT Press, Cambridge, Mass., 1987.

[Murakami 90a] M. Murakami: A Declarative Semantics of Flat Guarded Horn Clauses for Programs with Perpetual Processes, Theoretical Computer Science 75, pp.67-83, North-Holland, 1990.

[Murakami 90b] M. Murakami: Formal Semantics of Concurrent logic Programs, Journal of Information Processing Society (in Japanese, to appear).

[Shapiro 83] E. Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, 1983.

[Shapiro 89] E. Shapiro: The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, pp.413-510, Vol.21, No.3 (1989).

[Suga 90] H.Sugano: Meta and Reflective Computation in Logic Programming, Workshop on Meta-Programming in Logic programming, Leuven, Belgium, April 1990.

[Tanaka 88] J. Tanaka: Meta-interpreters and Reflective Operations in GHC, Proc. of International Conference on FGCS, ICOT, 1988.

[Tanaka 90] J. Tanaka: Reflective GHC and Its Implementation, Proc. of the Logic Programming Conference'90, ICOT (in Japanese).

[Ueda 85] K. Ueda: Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985.

[Ueda 90] K. Ueda: Designing Concurrent Programming Language, Proceedings of an International Conference organized by the IPSJ to Commemorate the 30th Anniversary, pp. 87-94, IPSJ 1990.