

# GHCにおけるプログラム変換

上田 和紀 ・ 吉川 康一

(財) 新世代コンピュータ技術開発機構

本論文では、(Flat) GHCのためのプログラム変換規則を示す。この規則は、著者の一人が以前に提案したものを改良したもので、展開とたたみ込みに基づいている。改良点は、結合の向きを制限したベキ等な代入を用いて規則を表わしたこと、入出力モードの概念を不要にしたこと、およびたたみ込み規則を一般化したことである。規則を正当化するために、ゴールのマルチ集合を、それとの有限回のやりとりのあらゆる可能性の集合によってモデル化する意味論を呈示する。この意味論は、同一化の失敗やデッドロックのような異常なふるまいも扱うので、プログラム変換が、そのようなふるまいを導入しないことの証明にも使うことができる。示した変換規則は、通信しあう複数のプロセスの融合に用いることができる。

## Transformation Rules for GHC Programs

*Kazunori Ueda and Koichi Furukawa*

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

Transformation rules for (Flat) GHC programs are presented, which refine the previous ones proposed by one of the authors. The rules are based on unfolding/folding and are novel in that they are stated in terms of idempotent substitutions with preferred directions of bindings. They are more general than the old ones in that no mode system is assumed and that the rule of folding is included. A semantic model is given which associates with a multiset of goals the set of possible finite sequences of transactions (via substitutions) with it, with which it can be shown that a transformation preserves any transaction sequences in the absence of the possibility of failure. Since the model deals with anomalous behaviors like the failure of unification and deadlock, it is guaranteed that the transformation does not introduce such anomalies. Applications of the transformation technique include the fusion of communicating parallel processes.

## 1. Introduction

### 1.1. Motivations

GHC (Guarded Horn Clauses) [Ueda 1985] [Ueda 1986a] is a simple parallel programming language for programming with communicating processes. It has introduced into Horn-clause logic programming the notion of *guard* to be used for two kinds of control: synchronization and choice nondeterminism. Synchronization is realized by restricting information flow caused by unification, and introduces partial order on bindings generated in the course of computation. A goal is viewed as a process that communicates with other processes by observing and generating substitutions. Readers uninitiated in GHC programming or parallel logic programming in general are referred to [Ueda 1986b] and [Shapiro 1987].

Computation in GHC is different from computation in ordinary logic programming in the following points:

- (1) It is *directed*. A GHC program has an intended direction of computation. Moreover, it does not search solutions by backtracking or OR-parallelism. Hence the value of a variable once computed is never revoked.
- (2) It may be *interactive*. A process may interact with other processes, which is the most important aspect of the language. Input and output can be done within the framework of the language by regarding the outside world as a process.
- (3) It may be *infinite*. In an interactive program, each interaction or a sequence of interactions is the main concern and the program need not necessarily terminate.

Since a GHC program is just an ordinary logic program if one forgets the aspect of control, (s)he may be tempted to define a set of transformation rules for GHC programs by adapting the set of rules developed for logic programs (e.g., in [Tamaki and Sato 1984]). Having a set of transformation rules will be quite useful for deriving efficient parallel programs from naive ones. For example, it can be used for process fusion by which we can adjust the granularity (of parallelism) of a program to the granularity of an implementation [Furukawa and Ueda 1985]. However, such transformation should preserve the *behavior* of processes defined by the program, and it was not clear how to guarantee the correctness of transformation.

The goal of this paper is to give a sufficiently general set of transformation rules for (Flat) GHC programs based on unfolding and folding, and to justify it using a simple semantic model. Unfold/fold transformation has been commonly used in functional and logic programming; here we want to focus on the transformation of interactive, possibly non-terminating parallel programs written in GHC.

### 1.2. Approach and Related Works

Our set of transformation rules is based on two previous works: unfold/fold transformation for ordinary logic programs by Tamaki and Sato [1984] and unfold transformation for Flat GHC programs by Furukawa et al. [1987].

Tamaki and Sato showed that their rules preserve the least Herbrand model of the program. Although their framework gives us a good guideline, we have to find an alternative to the least model semantics that cannot be used for non-terminating programs (which may not have base case clauses). In addition, the control structure of GHC must be taken into account since it is an integral part of the language.

The set of rules by Furukawa et al. takes control into account, and is close to our set in its structure. However, it has several points to be improved. First, a mode system that attaches either of the input and output modes to each argument of a predicate is assumed to reason about information flow, which loses the generality of the rules to some extent. Second, it uses the notion of 'input relatedness' to judge whether certain information can come from the caller, but this notion is hard to formalize correctly. Third, the rule of folding does exist, but it is not powerful enough to be used for process fusion. Fourth, the discussion on correctness is informal, due to the lack of formal tools.

We propose to solve the above problems in two aspects: the presentation of rules and the semantic framework. As for the presentation, we adopt idempotent substitutions [Lassez et al. 1987] to model information exchanged by processes. Moreover, we use normal forms of clauses to simplify the rules.

As for the semantic framework, we use the notion of a *partial answer substitution* (pas) to model a fragment of a possibly infinite computation. An act of providing a multiset of goals with a possibly empty input substitution and getting an observable output substitution (= pas) is called a transaction, and a computation is modeled as a finite sequence of transactions. The set of all possible computations of a multiset of goals constitutes a model of that multiset. Any computation is preserved by the transformation rules in the absence of the possibility of failure. Anomalous behaviors like failure and deadlock are also modeled, unlike the least Herbrand model comprising successful cases only. This means that the transformation rules preserve such anomalous behaviors as well.

## 2. Preliminaries

This section briefly introduces notions related to substitution and unification, most of which can be found elsewhere (e.g., [Lloyd 1984]).

A *term* and an *atom(ic formula)* are defined as usual. We consider finite terms and atoms only. An

infinite computation may create infinite structures in the limit, but we are concerned with transactions that are of a finite nature. By  $VAR$  we denote the denumerable set of all variables.

An *expression* is a term, an atom, or a syntactic entity (legally) made up of terms, atoms and connectives in GHC (':-', ',', '|'). By  $\text{var}(e)$  we denote the set of all the variables appearing in an expression  $e$ . A *simple expression* is either a term or an atom.

A *substitution* is a mapping from the set of variables to the set of terms, which is extended to a mapping from the set of expressions to the set of expressions in the usual manner. A substitution acts as an identity except for finitely many variables, hence it is described as a finite set of the form

$$\{v_1 \triangleleft t_1, \dots, v_n \triangleleft t_n\}, \quad n \geq 0,$$

where  $v_i$ 's are distinct variables,  $t_i$ 's are terms, and  $v_i \neq t_i$  for  $1 \leq i \leq n$ . (We use  $\equiv$  and  $\neq$  to denote the syntactic equality and inequality of simple expressions.) Each element of a substitution is called a *binding*.

Given a substitution  $\sigma = \{v_1 \triangleleft t_1, \dots, v_n \triangleleft t_n\}$ , by  $\text{domain}(\sigma)$  we denote the set  $\{v_1, \dots, v_n\}$  and by  $\text{range}(\sigma)$  the set  $\text{var}(t_1) \cup \dots \cup \text{var}(t_n)$ .

The composition of two substitutions is defined as follows. Let  $\sigma = \{u_1 \triangleleft s_1, \dots, u_n \triangleleft s_n\}$  and  $\tau = \{v_1 \triangleleft t_1, \dots, v_n \triangleleft t_n\}$ . Then the composition  $\sigma\tau$  (substitutions will be used as postfix operators throughout) is a finite set obtained from  $\{u_1 \triangleleft s_1\tau, \dots, u_n \triangleleft s_n\tau\} \cup \tau$  by deleting

- (1) all the elements  $u_i \triangleleft s_i\tau$  such that  $u_i \equiv s_i\tau$  and
- (2) all the elements  $v_i \triangleleft t_i$  such that  $v_i \in \text{domain}(\sigma)$ .

A substitution  $\sigma$  is said to be *idempotent* iff  $\sigma\sigma = \sigma$ . An idempotent substitution enjoys a nice property [Lassez et al. 1987]:

**Proposition 2.1.**  $\sigma\sigma = \sigma \iff \text{domain}(\sigma) \cap \text{range}(\sigma) = \emptyset$  (empty set)

Idempotent substitutions are adequate for modeling information in logic programming. A non-idempotent substitution introduces a variable whose occurrences are about to be rewritten, contrary to the single-assignment property of logical variables. Fortunately, the unification algorithms employed in most logic programming systems calculate idempotent mgu's defined below.

Now we define unification. A substitution  $\theta$  is said to be an *idempotent most general unifier (idmgu)* of two simple expressions  $t_1$  and  $t_2$  iff

- (1)  $t_1\theta \equiv t_2\theta$  and
- (2)  $\forall \sigma (t_1\sigma \equiv t_2\sigma \rightarrow \sigma = \theta\sigma)$ .

It is easy to see that if such  $\theta$  exists for  $t_1$  and  $t_2$ , it is idempotent and is an mgu in any other proposed

senses [Lassez et al. 1987]. The following theorem, which directly follows from the Unification Theorem in [Robinson 1979], guarantees that we need only consider idmgu's:

**Theorem 2.2.** If two simple expressions  $t_1$  and  $t_2$  are unifiable then there exists an idmgu of them.

**Example.** We consider various unifiers of two variables  $X$  and  $Y$ :  $\{X \triangleleft Y\}$  and  $\{Y \triangleleft X\}$  are idmgu's;  $\{X \triangleleft W, Y \triangleleft W\}$  is an idempotent unifier that is not most general; and  $\{X \triangleleft W, Y \triangleleft W, W \triangleleft Y\}$  is a non-idempotent, most general unifier.

Hereafter, we assume all the substitutions we consider to be idempotent.

As the above example shows, an idmgu may not be unique if we take the renaming of variables into account. However, some idmgu's may be preferable to others. For example, in SLD-resolution employing a selected goal  $p(X)$  and a clause with the head  $p(A)$ , we usually prefer replacing  $A$  by  $X$  to the other way around both on paper and in implementation.

Accordingly, we introduce a means to obtain a substitution with preferable directions of bindings. Let  $V$  be a set of variables. A substitution  $\sigma$  is said to be *smallest with respect to  $V$*  iff  $\forall (v \triangleleft t) \in \sigma (v \in V \wedge t \in VAR \rightarrow t \in V)$ . In other words, no variable-to-variable binding in  $\sigma$  can replace a variable in  $V$  by a variable not in  $V$ .

**Proposition 2.3.** Given two simple expressions  $t_1$  and  $t_2$  and a set  $V$  of variables, there exists an idmgu of  $t_1$  and  $t_2$  smallest w. r. t.  $V$  iff  $t_1$  and  $t_2$  are unifiable.

**Proof.** By construction from an arbitrary idmgu of  $t_1$  and  $t_2$ . ■

Unification can be defined for a multiset of equations as well as for two simple expressions. Let  $S = \{s_1 = t_1, \dots, s_n = t_n\}$ , where  $s_i$ 's are terms. Then  $\theta$  is said to be an idmgu of  $S$  iff it is an idmgu of two terms  $f(s_1, \dots, s_n)$  and  $f(t_1, \dots, t_n)$ ,  $f$  being an arbitrary  $n$ -ary function symbol. The set  $S$  is said to be *solvable* iff it has an idmgu.

Some miscellaneous notions. Let  $\theta = \bigcup_{i=1}^n \{v_i \triangleleft t_i\}$ . Then by  $\bar{\theta}$  we denote the (multi)set of equations  $\bigcup_{i=1}^n \{v_i = t_i\}$ . It is easy to show that  $\theta$  is an idmgu of  $\bar{\theta}$ . Finally, we define the restriction  $\theta|_e$  of a substitution  $\theta$  to an expression  $e$  as follows:

$$\theta|_e = \{(v \triangleleft t) \in \theta \mid v \in \text{var}(e)\}.$$

### 3. Flat GHC

We first define a rather abstract syntax of GHC. A GHC program is a set of guarded clauses of the following form:

$$h :- G \mid B$$

where  $h$  is an atom and  $G$  and  $B$  are multisets of atoms. The atom  $h$  is called the *head* and an atom in  $G$  or  $B$  is called a goal. The part of a guarded clause before the commitment operator ‘|’ is called the *guard* and the part after ‘|’ is called the *body*.  $G$  and  $B$  are multisets rather than sets because two syntactically identical goals may commit to different clauses. One predefined binary predicate, ‘=’, is provided for unifying two terms.

A program is invoked by a goal clause

$$:- B$$

where  $B$  is a multiset of goals.

Flat GHC imposes restriction on the guard goals of each clause. A guard goal of a Flat GHC program must be either

- a unification goal of the form  $t_1 = t_2$ , or
- a call to a *test predicate*, where a test predicate is made up of clauses with empty bodies.

Note that a goal calling a test predicate never gets instantiated, and that its result (whether it succeeds or not) is uniquely determined depending on the values of the arguments.

For convenience, hereafter we use the following notational conventions:  $G_U$  denotes the multiset of the unification goals in a guard;  $G_N$  denotes the multiset of the non-unification goals in a guard;  $B_U$  denotes the multiset of the unification goals in a body; and  $B_N$  denotes the multiset of the non-unification goals in a body. Also, by  $C :: C'$  we denote a clause  $C$  of which  $C'$  is a variant using fresh variables.

Now we describe an operational semantics of Flat GHC. Note that this does *not* specify how a Flat GHC program must be executed.

To execute a multiset of goals means to execute each constituent goal in parallel. In this paper we assume that the goals in a multiset may be executed by unfair scheduling. The multiset of goals succeeds when the constituent goals all succeed.

A unification goal  $t_1 = t_2$  tries to unify  $t_1$  and  $t_2$ . What complicates things is that it may possibly be executed in parallel with other unification goals which may instantiate  $t_1$  and/or  $t_2$ . However, here we adopt the simplest scheme:  $t_1 = t_2$  is executed as an atomic action and their idmgu is applied to all the goals being executed. This overspecification of the operational semantics will be compensated in the semantic model in Section 4. The goal  $t_1 = t_2$  succeeds (also said to be *reduced*) when  $t_1$  and  $t_2$  become identical.

The execution of a non-unification goal  $g$  proceeds as follows: The goal  $g$  searches for a clause  $C :: (h :- G_U \cup G_N \mid B)$  such that

- $\{g\theta = h\} \cup G_U$  has an idmgu  $\sigma$  such that  $g\theta\sigma \equiv g\theta$  and

- $G_N\sigma$  succeeds,

where  $\theta$  is a substitution given in the meantime by other unification goals running in parallel with  $g$ . If  $g$  finds such  $C$ , then  $g$ , now instantiated to  $g\theta$ , *commits to  $C$*  and its body  $B\sigma$  is executed. We also say that  $g\theta$  is *reduced to  $B\sigma$  using  $C$* . We assume that if there exist clauses to which  $g\theta$  can commit,  $g\theta$  will eventually commit to one of them. This is true even if the guard of some clause falls into infinite computation. Our computation is fair with respect to the computation of guards in this sense. Note, however, that we still have choice nondeterminism in commitment and some clause may be ignored by every goal. A non-unification goal succeeds if and when it commits to some clause and its body succeeds.

Finally, we introduce notations for reducibility relations between a non-unification goal  $g$  and a clause  $C :: (h :- G_U \cup G_N \mid B)$ :

$$\begin{aligned} +(g, C) &\stackrel{\text{def}}{=} \exists \sigma ((\sigma \text{ is an idmgu of } \{g=h\} \cup G_U) \\ &\quad \wedge (g\sigma \equiv g) \wedge (G_N\sigma \text{ succeeds})) \\ -(g, C) &\stackrel{\text{def}}{=} \neg \exists \theta +(g\theta, C) \\ ?(g, C) &\stackrel{\text{def}}{=} \neg +(g, C) \wedge \exists \theta +(g\theta, C) \end{aligned}$$

They stand for “reducible as is”, “irreducible” and “possibly reducible with more information”, respectively. Exactly one of  $+$ ,  $-$  and  $?$  holds for  $g$  and  $C$ . When  $G_U = \emptyset$ , the definition of  $+(g, C)$  is simplified to

$$\exists \sigma ((g \equiv h\sigma) \wedge (G_N\sigma \text{ succeeds})).$$

Section 5.1 shows how to eliminate  $G_U$ . Recall that in Flat GHC, it is uniquely determined whether “ $G_N\sigma$  succeeds” or not.

#### 4. Semantic Model

This section gives a semantic model of a multiset of goals with which the correctness of program transformation will be discussed.

First of all, we clarify our approach to the semantic model. We choose to model the parallel execution of a multiset of goals as the set of all possible serialized computations (simply called *computations* henceforth). Each computation is considered to have two aspects: internal (concrete) and external (abstract).

Internally, a computation is a possible sequence of reductions to which substitutions may be incrementally given from outside. Consider the parallel execution of two goals  $g_1$  and  $g_2$ . The reduction of  $g_1$  using  $C_1$  must precede the reduction of  $g_2$  using  $C_2$  in any computation in the model containing them iff the reduction of  $g_2$  using  $C_2$  needs directly or indirectly a substitution generated by the body of  $C_1$  derived from  $g_1$ . If no such dependency exists, those reductions will appear in any order.

Externally, a computation starting with a multiset  $B_0$  of goals is viewed as a sequence  $\langle \alpha_1, \beta_1 \rangle \langle \alpha_2, \beta_2 \rangle \dots$  of transactions with  $B_0$ . A (*normal*) *transaction*  $\langle \alpha_i, \beta_i \rangle$  is an act of providing  $B_0$  with a possibly empty *input substitution*  $\alpha_i$  and getting an observable (see below) *output substitution*  $\beta_i$ . Each normal transaction is realized by a finite number of reductions. In addition to normal transactions, some special transactions are introduced to model the failure of unification, deadlock (the irreducibility of non-unification goals), and infinite reductions without observable substitutions.

Communication between a multiset  $B_0$  of goals and the rest of the goals is done only through the variables in  $B_0$ . Suppose the first transaction  $\langle \alpha_1, \beta_1 \rangle$  has been made and  $B_0$  is reduced to  $B_1$  that represents ‘the rest of the computation’. Then, the next transaction must be made through the variables in  $B_0\alpha_1\beta_1$ ; the variables in  $B_1 \setminus B_0\alpha_1\beta_1$  are not accessible. In general,  $\text{var}(B_0\alpha_1\beta_1 \dots \alpha_k\beta_k)$  is called the *interface* for the rest of the computation represented by  $B_k$ .

Now let us formalize the model of a multiset  $B_i$  of goals under a program  $\mathcal{P}$ , with which substitutions are communicated through an interface  $\text{var}(B)$ . The model is denoted as  $\llbracket B_i |_B \rrbracket_{\mathcal{P}}$ .  $\llbracket B |_B \rrbracket_{\mathcal{P}}$  is abbreviated to  $\llbracket B \rrbracket_{\mathcal{P}}$ .  $\llbracket B_i |_B \rrbracket_{\mathcal{P}}$  is a set of finite sequences of transactions and satisfies the following properties. Since we may want to make no transaction with  $B_i$ , we first have

- (0)  $\epsilon \in \llbracket B_i |_B \rrbracket_{\mathcal{P}}$ , where  $\epsilon$  is an empty sequence of transactions.

We then consider all possible concrete computations of  $B_i$  to which an input substitution  $\alpha_i$  is given initially but no subsequent input is given, and classify them into four cases (not mutually exclusive) according to their prefixes:

- (1) *Normal transaction*. There exists a sequence of reductions

$$B_i\alpha_i = B_{i,0} \longrightarrow B_{i,1} \longrightarrow \dots \longrightarrow B_{i,k} = B_{i+1}$$

such that

- $\beta'_i$  is an idmgu of the multiset of all the unification goals executed in the sequence that is smallest w. r. t.  $\text{var}(B\alpha_i)$ , and
- $\beta'_i$  is *observable*, i.e.,  $\beta_i \stackrel{\text{def}}{=} \beta'_i|_{B\alpha_i} \neq \emptyset$ .

Then, we divide  $\beta_i$  into two (idempotent) substitutions  $\beta_{i1}$  and  $\beta_{i2}$  such that

- (N1)  $\beta_i = (\beta_{i1}\beta_{i2})|_{B\alpha_i}$  (i.e.,  $\beta_{i1}\beta_{i2}$  and  $\beta_i$  have the same effect on  $B\alpha_i$ ),
- (N2)  $\beta_{i1} = \beta_{i1}|_{B\alpha_i}$ ,  $\beta_{i1} \neq \emptyset$  (i.e., observable), and  $\beta_{i1}$  is smallest w. r. t.  $\text{var}(B\alpha_i)$ ,
- (N3)  $\beta_{i2} = \beta_{i2}|_{B\alpha_i\beta_{i1}}$ , and
- (N4) those variables in  $\beta_{i1}$  and/or  $\beta_{i2}$  but not in  $\beta_i$  are all fresh variables.

We can always find at least one such pair of  $\beta_{i1}$  and  $\beta_{i2}$  by letting  $\beta_{i1} = \beta_i$  and  $\beta_{i2} = \emptyset$ , and there may be many others. Now for every possible pair of  $\beta_{i1}$  and  $\beta_{i2}$ , we have

$$\begin{aligned} \forall \vec{t} (\vec{t} \in \llbracket (\overline{\beta_{i2}} \cup B_{i+1}) |_{B\alpha_i\beta_{i1}} \rrbracket_{\mathcal{P}} \\ \rightarrow \langle \alpha_i, \beta_{i1} \rangle \vec{t} \in \llbracket B_i |_B \rrbracket_{\mathcal{P}}). \end{aligned}$$

- (2) *Failure*. There exists a possibly empty sequence of reductions  $B_i\alpha_i \longrightarrow \dots \longrightarrow B_{i+1}$  such that the multiset of the unification goals in  $B_{i+1}$  is unsolvable. Then

$$\langle \alpha_i, \top \rangle \in \llbracket B_i |_B \rrbracket_{\mathcal{P}}.$$

- (3) *Success and Deadlock*. There exists a possibly empty sequence of reductions  $B_i\alpha_i \longrightarrow \dots \longrightarrow B_{i+1}$  such that none of its prefixes (including the whole sequence) cause *normal transaction* or *failure* and that  $B_{i+1}$  allows no further reduction. Then

$$\begin{cases} \langle \alpha_i, \perp_{\text{success}} \rangle \in \llbracket B_i |_B \rrbracket_{\mathcal{P}}, & \text{if } B_{i+1} = \emptyset; \\ \langle \alpha_i, \perp_{\text{deadlock}} \rangle \in \llbracket B_i |_B \rrbracket_{\mathcal{P}}, & \text{otherwise.} \end{cases}$$

- (4) *Divergence*. There exists an infinite sequence of reductions  $B_i\alpha_i \longrightarrow \dots$  such that none of its prefixes cause *normal transaction*. Then

$$\langle \alpha_i, \perp_{\text{divergence}} \rangle \in \llbracket B_i |_B \rrbracket_{\mathcal{P}}.$$

Now  $\llbracket B_i |_B \rrbracket_{\mathcal{P}}$  is defined as the smallest set (or the intersection of all the sets) satisfying the above properties, where we regard two or more sequences as identical if their differences come only from the different naming of fresh variables (introduced in a sequence of reductions and in the division of  $\beta_i$ ). Note that different abstract computations may be obtained from a concrete computation, and that an abstract computation may be obtained from different concrete computations. Our notion of an abstract computation is at a similar level of abstraction to a *behavior* in [Lichtenstein et al. 1987], but the definitions are quite different.

The transactions  $\langle \alpha_i, \top \rangle$ ,  $\langle \alpha_i, \perp_{\text{success}} \rangle$ ,  $\langle \alpha_i, \perp_{\text{deadlock}} \rangle$  and  $\langle \alpha_i, \perp_{\text{divergence}} \rangle$  are called *special transactions*. Of these, success, deadlock and divergence mean the *inactivity* of a multiset of goals and cannot be distinguished from outside. Therefore, we may omit the subscripts of  $\perp$  when the differences are not important.

Some notes on the above model will be useful. First of all, each element of a model is a finite sequence. This means that we are modeling a multiset of goals using the set of all finite transactions with it. A consequence of this is that we cannot handle some properties that could be observed only at infinity. This, however, does not mean that we cannot model perpetual processes; it is quite possible to deal with programs that are useful but essentially non-terminating. This point will be discussed further in Section 8.

Item (1) says that we may observe the result of unification using two or more transactions even when the unification is specified by a single goal. We want to leave it to implementations how unification is performed and how output substitutions are applied. A variable being observed may be instantiated to an infinite term in the limit, but we require that each transaction be of a finite nature; it should return a finite set of bindings after finite computation.

Since Items (1) and (2) are not exclusive, the process of a computation may be observed from outside even if it is doomed to fail. However, we have given up the idea of specifying precisely what can be observed before a computation fails. This is because the behavior of a failing multiset of unification goals is hard to define and not very important. Note that we have 'occur check' as a consequence of the finiteness of terms.

Modeling inactivity as well as normal transactions is important, because we want to distinguish between a program that will eventually produce a non-empty output and one that may or may not produce it. For example, let  $\mathcal{P}$  be

$$\{(p(X) :- \emptyset \mid \{X=1\}), (p(X) :- \emptyset \mid \emptyset)\}$$

and  $\mathcal{Q}$  be

$$\{(p(X) :- \emptyset \mid \{X=1\})\}.$$

Then,  $\langle \emptyset, \perp \rangle$  is in  $\llbracket p(X) \rrbracket_{\mathcal{P}}$  but not in  $\llbracket p(X) \rrbracket_{\mathcal{Q}}$ , while  $\langle \emptyset, \{X \triangleleft 1\} \rangle$  is both in  $\llbracket p(X) \rrbracket_{\mathcal{P}}$  and in  $\llbracket p(X) \rrbracket_{\mathcal{Q}}$ .

Output substitutions  $\beta_1, \beta_2, \dots$  in an abstract computation  $\langle \alpha_1, \beta_1 \rangle \langle \alpha_2, \beta_2 \rangle \dots$  are called *partial answer substitutions (pas's)*, since if the computation ends with  $\langle \alpha_n, \perp_{success} \rangle$ , the universal closure  $\forall (B\alpha_1\beta_1\alpha_2\beta_2\dots\alpha_n)$  is a logical consequence of the declarative reading of the program.

The example below shows how the model circumvents the Brock-Ackerman anomaly [Brock and Ackerman 1981]. Let  $\mathcal{P}$  (in the syntax following DEC-10 Prolog) be:

```
d([A|_],0) :- true | 0=[A,A].
merge([A|X1],Y, Z) :- true |
  Z=[A|Z1], merge(X1,Y,Z1).
merge(X, [A|Y1],Z) :- true |
  Z=[A|Z1], merge(X,Y1,Z1).
merge([], Y, Z) :- true | Z=Y.
merge(X, [], Z) :- true | Z=X.

p1([A|Z1],0) :- true | 0=[A|01], p11(Z1,01).
p11([B|_],01) :- true | 01=[B].

p2([A,B|_],0) :- true | 0=[A,B].

g1(I,J,0) :- true |
  d(I,X), d(J,Y), merge(X,Y,Z), p1(Z,0).
g2(I,J,0) :- true |
  d(I,X), d(J,Y), merge(X,Y,Z), p2(Z,0).
```

Then, the computation

$$\langle \{I \triangleleft [5|_]\}, \{0 \triangleleft [5|01]\} \rangle$$

belongs both to  $\llbracket g1(I, J, 0) \rrbracket_{\mathcal{P}}$  and to  $\llbracket g2(I, J, 0) \rrbracket_{\mathcal{P}}$ , but

$$\langle \{I \triangleleft [5|_]\}, \{0 \triangleleft [5|01]\} \rangle \langle \{J \triangleleft [6|_]\}, \{01 \triangleleft [6]\} \rangle$$

belongs only to  $\llbracket g1(I, J, 0) \rrbracket_{\mathcal{P}}$  and not to  $\llbracket g2(I, J, 0) \rrbracket_{\mathcal{P}}$ .

## 5. Transformation Rules

Now we are in a position to describe the set of rules for transforming an initial program  $\mathcal{P}_0$  to  $\mathcal{P}_1, \mathcal{P}_2$  and so forth. Actual transformation will consist of the rewriting of existing clauses and the introduction of new, auxiliary clauses. However, we treat those new clauses as if they existed in  $\mathcal{P}_0$ , following the formulation of [Tamaki 1987]. One purpose of this is to justify the introduction of new clauses which usually introduces new possible computations. The set of those new clauses in  $\mathcal{P}_0$  is referred to as  $\mathcal{D}$ .  $\mathcal{D}$  must satisfy the following conditions:

(D1)  $\forall g \forall C \in \mathcal{D} \forall C' \in \mathcal{P}_0 \setminus \{C\} (+ (g, C) \rightarrow - (g, C'))$   
(i.e., a clause in  $\mathcal{D}$  does not 'overlap' with any other clauses.)

(D2)  $\forall (h :- G \mid B) \in \mathcal{P}_0 \forall g \in B \forall C \in \mathcal{D} (- (g, C))$  (i.e., a body goal in a clause in  $\mathcal{P}_0$  cannot commit to a clause in  $\mathcal{D}$ .)

(D3) Each clause in  $\mathcal{D}$  must be of the form

$$h' :- \emptyset \mid B'_N, \quad \text{var}(h') \subseteq \text{var}(B'_N).$$

These conditions are to guarantee the correctness of folding (Section 5.4).

### 5.1. Normalization

Normalization transforms a clause  $C$  in  $\mathcal{P}_j$  to a normal form by executing the unification goals in it as far as possible.

First, we 'execute' the unification goals in the guard. Let  $C$  be

$$C:: h :- G_U \cup G_N \mid B.$$

If  $G_U$  is unsolvable then let  $\mathcal{P}_{j+1}$  be  $\mathcal{P}_j \setminus \{C\}$ . Otherwise, let  $C'$  be

$$C':: h\theta :- G_N\theta \mid B\theta,$$

where  $\theta$  is an idmgu of  $G_U$ .

Then we 'execute' the unification goals in the body of  $C'$  now of the form

$$C':: h' :- G'_N \mid B'_U \cup B'_N.$$

If  $B'_U$  is unsolvable, stop the program transformation (we are not interesting in improving a program that

may fail). Otherwise, let  $\theta$  be an idmgu of  $B'_U$  smallest w. r. t.  $\text{var}(h')$ .  $\theta|_{h'}$  stands for the output substitution of  $C'$ . Finally, let  $\mathcal{P}_{j+1}$  be  $(\mathcal{P}_j \setminus \{C'\}) \cup \{C''\}$ , where  $C''$  is

$$C'':: h' :- G'_N \mid \overline{\theta|_{h'}} \cup B'_N \theta.$$

Note that we can ignore  $G'_N$  in simplifying  $B'_U$ , because  $G'_N$  has been solved when  $B'_U$  is executed.

The rest of the rules in the subsequent sections assume that all the clauses in a program have been normalized.

## 5.2. Immediate Execution of a Body Goal

We have two rules based on unfolding: Immediate execution (below) and case splitting (Section 5.3). The distinction is not made in the transformation of ordinary logic programs, but it is necessary in GHC in which guard plays a crucial role.

Consider a goal  $g$  and a clause  $C$ . In ordinary logic programming,  $g$  is either reducible or irreducible to the body of  $C$ , depending on whether  $g$  and the head of  $C$  are unifiable. In GHC, on the other hand, there exists the third case where  $g$  is reducible to the body of  $C$  only when  $g$  is *appropriately instantiated*. The unfolding rules must correctly deal with such a synchronization condition expressed in the guard of  $C$ . Case splitting is provided exactly for this purpose, while immediate execution deals with the unfolding which does not involve synchronization. Case splitting may look more powerful, but immediate execution can be applied in less limited contexts.

Now we state the rule of immediate execution. Let  $C \in \mathcal{P}_j$  be

$$C:: h :- G_N \mid B_U \cup B_N$$

and  $B_N = \{g\} \cup B'_N$ . If the conditions

$$(I1) \quad \forall C_k \in \mathcal{P}_j (+ (g, C_k) \vee - (g, C_k))$$

$$(I2) \quad \exists C_k \in \mathcal{P}_j (+ (g, C_k))$$

both hold, then for each  $C_k:: (h_k :- G_{Nk} \mid B_k) \in \mathcal{P}_j$  such that  $+ (g, C_k)$  holds, make a clause

$$C'_k:: h :- G_N \mid B_U \cup B_k \theta \cup B'_N$$

where  $\theta$  is an idmgu of  $g$  and  $h_k$  smallest w. r. t.  $\text{var}(g)$ , and let  $\mathcal{P}_{j+1}$  be  $(\mathcal{P}_j \setminus \{C\}) \cup \{C'_k \mid + (g, C_k)\}$ .

Let us see why Condition (I2) is necessary. Suppose  $C_1 = (\text{p}(X) :- \emptyset \mid \{q\})$ ,  $C_2 = (\text{p}(X) :- \emptyset \mid \{X=1\})$  and  $\mathcal{P}_j = \{C_1, C_2\}$ . The goal  $\text{p}(X)$  may either deadlock or get the substitution  $\{X \triangleleft 1\}$ . However, if we applied immediate execution to  $C_1$ ,  $\mathcal{P}_{j+1}$  would be  $\{C_2\}$ , under which the goal  $\text{p}(X)$  will not deadlock.

## 5.3. Case Splitting

Let  $C \in \mathcal{P}_j$  be

$$C:: h :- G_N \mid B_U \cup B_N$$

and  $B_N = \{g_1, \dots, g_n\}$ . Case splitting requires the following conditions:

$$(C1) \quad B_U = \emptyset$$

$$(C2) \quad \forall g \forall C' \in \mathcal{P}_j \setminus \{C\} (+ (g, C) \rightarrow - (g, C')) \text{ (i.e., } C \text{ does not 'overlap' with other clauses.)}$$

If both hold, make a clause  $C'_{ik}$  for each pair of  $g_i \in B_N$  and  $C_k \in \mathcal{P}_j$  of the form

$$C_k:: h_k :- G_{Nk} \mid B_k$$

as follows:

- If  $g_i$  and  $h_k$  are unifiable then let  $C'_{ik} = \perp$ . Otherwise, let  $\theta$  be an idmgu of them smallest w. r. t.  $\text{var}(g_i)$ . (Intuitively,  $\theta|_{g_i}$  stands for an input substitution necessary (and sufficient if  $G_{Nk} = \emptyset$ ) for  $g_i$  to commit to  $C_k$ .) If the condition

$$(C3) \quad (\text{domain}(\theta|_{g_i}) \cup \text{range}(\theta|_{g_i})) \setminus \text{var}(h_k) \subseteq \text{var}(h)$$

holds, let  $C'_{ik}$  be

$$C'_{ik}:: h \theta :- G_N \theta \cup G_{Nk} \theta \mid (B_N \setminus \{g_i\}) \theta \cup B_k \theta.$$

Otherwise, let  $C'_{ik} = \perp$ .

Finally, let  $\mathcal{P}_{j+1}$  be  $(\mathcal{P}_j \setminus \{C\}) \cup \{C'_{ik} \mid C'_{ik} \neq \perp\}$ .

For a goal reduced using  $C$  to generate an output substitution, at least one more reduction is necessary because  $C$  has no unification goals in its body. Case splitting enumerates all the possibilities for the first such reduction.

Condition (C1) is necessary because  $\theta|_{g_i}$  and  $G_{Nk}$  are promoted to the guard of  $C$ . If  $B_U \neq \emptyset$ , to observe the output substitution from  $B_U$  might require more input substitution under  $\mathcal{P}_{j+1}$  than under  $\mathcal{P}_j$ .

The purpose of Condition (C2) is just the same as that of Condition (I2). Without (C2), the behavior of a goal  $g$  might change in case  $g$  can commit to  $C$  but to none of the  $C'_{ik}$ 's.

Condition (C3) requires that the condition for  $g_i$  to commit to  $C_k$ , expressed as  $\theta|_{g_i}$ , be promoted to the guard of  $C'_{ik}$  without being diminished. Suppose  $g_i = \text{q}(X, Y, Z, U)$  and  $h_k = \text{q}(V, W, -, -)$ . Then  $\theta|_{g_i} = \{X \triangleleft Y, Z \triangleleft f(W)\}$ , which states that  $X$  must be identical to  $Y$  and the principal function symbol of  $Z$  must be unary  $f$ . Then,  $\text{p}(X, Y, Z)$  is a legal head of  $C$  which will be instantiated to  $\text{p}(Y, Y, f(W))$  in  $C'_{ik}$ , but if we dropped any of  $X, Y$  and  $Z$  from  $\text{p}(X, Y, Z)$ , the condition represented by  $\theta|_{g_i}$  would not be promoted correctly. Condition (C3) is the formalization of the 'input relatedness' condition in [Furukawa et al. 1987].

## 5.4. Folding

The Folding rule is similar to that of Tamaki and Sato. A clause used for folding must belong to  $\mathcal{D}$  but need not necessarily to  $\mathcal{P}_j$ .

Let  $C \in \mathcal{P}_j$  be

$$C:: h :- G_N \mid B_U \cup B_N$$

and  $B_N = K \cup B'_N$ , where  $K$  is the multiset of goals to be folded. Let the clause to be used for folding ( $\in \mathcal{D}$ ) be

$$C'':: h'' :- \emptyset \mid B''_N, \quad (\text{var}(h'') \subseteq \text{var}(B''_N)).$$

Folding requires the following four conditions to hold:

(F1)  $\exists \theta' (K = B''_N \theta')$

(F2) Let  $\theta$  be a substitution such that  $K = B''_N \theta$  and  $\theta = \theta|_{B''_N}$ . We intend to replace  $K$  in  $C$  by  $h''\theta$ . The condition for this is:

$$\begin{aligned} \exists \sigma ((K\sigma = B''_N \theta|_{h''}) \wedge (\text{domain}(\sigma) \\ \cap \text{var}(h :- G_N \mid B_U \cup B'_N) = \emptyset)), \end{aligned}$$

that is,  $K$  and  $B''_N \theta|_{h''}$  are variants and  $\sigma$  does not rewrite  $C$  except for  $K$ .

(F3)  $C$  is either the result of applying transformation rules to a clause of  $\mathcal{P}_0 \setminus \mathcal{D}$  zero or more times, or the result of applying case splitting to a clause in  $\mathcal{P}_0$  at least once.

If they all hold, then let  $\mathcal{P}_{j+1}$  be  $(\mathcal{P}_j \setminus \{C\}) \cup \{C'\}$ , where  $C'$  is

$$C':: h :- G_N \mid B_U \cup h''\theta \cup B'_N.$$

Conditions (F1) to (F3) may look complex, but are very similar to those in [Tamaki 1987]. Conditions (F1), (F2) and (D1) together guarantee that the immediate execution of  $h''\theta$  in  $C'$  yields  $C$  and only  $C$ . Condition (F3) is to avoid introducing infinite reductions by, say, folding a clause in  $\mathcal{D} \cap \mathcal{P}_j$  by itself. To preserve the semantics of a non-terminating program, we require the case-splitting, rather than the immediate execution, of a clause in  $\mathcal{D}$  to be folded in future. There may be various sets of conditions for correct folding (e.g., see [Kanamori and Fujita 1986]), but our set suffice at least for process fusion.

## 6. Correctness of the Transformation Rules

A transformation sequence  $\mathcal{P}_0, \mathcal{P}_1, \dots$  preserves the meaning of  $\mathcal{P}_0$  in the sense that for any multisets of goals  $B'$  and  $B$  and for any  $i > 0$  and  $n > 0$ , the following hold:

(1) A computation  $\langle \alpha_1, \beta_1 \rangle \dots \langle \alpha_{n-1}, \beta_{n-1} \rangle \langle \alpha_n, \beta_n \rangle$  belonging to  $\llbracket B' \mid B \rrbracket_{\mathcal{P}_0}$  belongs also to  $\llbracket B' \mid B \rrbracket_{\mathcal{P}_i}$  if none

of the computations  $\langle \alpha_1, \beta_1 \rangle \dots \langle \alpha_{k-1}, \beta_{k-1} \rangle \langle \alpha_k, \top \rangle$  for  $1 \leq k \leq n$  belong to  $\llbracket B' \mid B \rrbracket_{\mathcal{P}_0}$ .

(2) Item (1) holds also when we replace “ $\beta_n$ ” by any of “ $\perp_{\text{success}}$ ”, “ $\perp_{\text{deadlock}}$ ” and “ $\perp_{\text{divergence}}$ ”.

(3) A computation  $\langle \alpha_1, \beta_1 \rangle \dots \langle \alpha_{n-1}, \beta_{n-1} \rangle \langle \alpha_n, \top \rangle$  belonging to  $\llbracket B' \mid B \rrbracket_{\mathcal{P}_0}$  belongs also to  $\llbracket B' \mid B \rrbracket_{\mathcal{P}_i}$  if none of the computations  $\langle \alpha_1, \beta_1 \rangle \dots \langle \alpha_{k-1}, \beta_{k-1} \rangle \langle \alpha_k, \top \rangle$  for  $1 \leq k < n$  belong to  $\llbracket B' \mid B \rrbracket_{\mathcal{P}_0}$ .

These can be shown using the induction on the lengths of abstract computations. The basic task is to show that if a multiset  $B_0$  of goals allows a computation beginning with a normal transaction  $\langle \alpha_1, \beta_1 \rangle$  that leaves a multiset  $B_1$  of goals, then there exists under another program the same transaction that leaves the equivalent multiset of goals. Since each normal transaction is made up of a finite number of reductions, we can apply to it the proof technique used in [Tamaki and Sato 1984] which is based on manipulation of finite proof trees. The use of idempotent substitutions for the presentation of the rules serves to avoid the complication of the proof.

The same technique can be used also for proving that if  $\llbracket B_0 \mid B \rrbracket_{\mathcal{P}_0}$  contains  $\langle \alpha_1, \top \rangle$ ,  $\langle \alpha_1, \perp_{\text{success}} \rangle$  and/or  $\langle \alpha_1, \perp_{\text{deadlock}} \rangle$ ,  $\llbracket B_0 \mid B \rrbracket_{\mathcal{P}_i}$  also contains them, and vice versa. A proof on  $\langle \alpha_1, \perp_{\text{divergence}} \rangle$  requires somewhat different technique because it must handle infinite sequences of reductions. A complete proof will be found in [Ueda and Furukawa 1989].

## 7. Examples

This section illustrates how to apply our transformation rules to process fusion [Furukawa and Ueda 1985]. We consider a simple program that computes a sequence of the partial sums of an integer sequence.

```
integerSums(I,N,Sums) :- true |
  integers(I,N,Is), sums(Is,Sums). (1)
```

```
integers(I,N,Is) :- I=<N |
  Is=[I|Is1], I1:=I+1, integers(I1,N,Is1). (2)
integers(I,N,Is) :- I>N | Is=[]. (3)
```

```
sums(Is,Sums) :- true | sums1(Is,0,Sums). (4)
```

```
sums1([], _,Sums) :- true | Sums=[]. (5)
```

```
sums1([I|Is1],S,Sums) :- true |
  S1:=I+S, Sums=[S1|Sums1],
  sums1(Is1,S1,Sums1). (6)
```

Our objective is to obtain a single-process program which computes the same sequence. We start by executing the body goals of Clause (1) until we have two tail-recursive goals:

```
Clause (1)
  ↓ Immediate Execution
integerSums(I,N,Sums) :- true |
  integers(I,N,Is), sums1(Is,0,Sums). (7)
```



Then we introduce a new clause for the final single process by parameterizing the second argument of `sums1` and leaving `Is` local. This is the key step which requires heuristics; however, the only heuristics needed is to generalize parameters. The resulting clause is:

```
fused_integerSums(I,N,S,Sums) :- true |
  integers(I,N,Is), sums1(Is,S,Sums). (8)
```

The second argument of `sums1` is generalized to a variable `S`, and it is included in the clause head. Now we try to obtain a single tail-recursive program using case-splitting and folding:

Clause (8)

↓ *Case Splitting*

```
fused_integerSums(I,N,S,Sums) :- I=<N |
  Is=[I|Is1], I1:=I+1, integers(I1,N,Is1),
  sums1(Is,S,Sums). (9)
```

```
fused_integerSums(I,N,S,Sums) :- I >N |
  Is=[], sums1(Is,S,Sums). (10)
```

Clause (9)

↓ *Normalization*

```
fused_integerSums(I,N,S,Sums) :- I=<N |
  I1:=I+1, integers(I1,N,Is1),
  sums1([I|Is1],S,Sums).
```

↓ *Immediate Execution*

```
fused_integerSums(I,N,S,Sums) :- I=<N |
  I1:=I+1, integers(I1,N,Is1),
  S1:=I+S, Sums=[S1|Sums1],
  sums1(Is1,S1,Sums1).
```

↓ *Folding by (8)*

```
fused_integerSums(I,N,S,Sums) :- I=<N |
  I1:=I+1, S1:=I+S, Sums=[S1|Sums1],
  fused_integerSums(I1,N,S1,Sums1). (11)
```

Clause (10)

↓ *Normalization and Immediate Execution*

```
fused_integerSums(I,N,S,Sums) :- I >N |
  Sums=[]. (12)
```

The remaining task is to express the original predicate `integerSums` in terms of the newly introduced predicate `fused_integerSums`:

Clause (7)

↓ *Folding by (8)*

```
integerSums(I,N,Sums) :- true |
  fused_integerSums(I,N,0,Sums). (13)
```

The resulting clauses (11), (12) and (13) give a new definition of the `integerSums` program. This program contains only one tail-recursive process; the intermediate stream `Is` and the operations on it have been eliminated. If the program is to be executed on one processor, the compiled code of the new program will usually be better than the code obtained by compiling the original two tail-recursive procedures separately.

We will next show how we can deal with a stream transformer which may absorb some of the input elements.

```
evenSquare(Xs,Ys) :- true |
  evenseq(Xs,Es), squareseq(Es,Ys). (14)
```

```
evenseq([X|Xs1],Es) :- even(X) |
  Es=[X|Es1], evenseq(Xs1,Es1). (15)
```

```
evenseq([X|Xs1],Es) :- odd(X) |
  evenseq(Xs1,Es). (16)
```

```
squareseq([E|Es1],Ys) :- true |
  Y:=E^2, Ys=[Y|Ys1], squareseq(Es1,Ys1). (17)
```

Let us first split Clause (14), as we did in the `integerSums` example:

Clause (14)

↓ *Case splitting*

```
evenSquare([X|Xs1],Ys) :- even(X) |
  Es=[X|Es1], evenseq(Xs1,Es1),
  squareseq(Es,Ys). (18)
```

```
evenSquare([X|Xs1],Ys) :- odd(X) |
  evenseq(Xs1,Es), squareseq(Es,Ys). (19)
```

Clause (18)

↓ *Normalization and Immediate Execution*

```
evenSquare([X|Xs1],Ys) :- even(X) |
  evenseq(Xs1,Es1),
  Y:=X^2, Ys=[Y|Ys1], squareseq(Es1,Ys1).
```

↓ *Folding by (14) (Assume D = {Clause (14)}.)*

```
evenSquare([X|Xs1],Ys) :- even(X) |
  Y:=X^2, Ys=[Y|Ys1], evenSquare(Xs1,Ys1). (20)
```

Clause (19)

↓ *Folding by (14)*

```
evenSquare([X|Xs1],Ys) :- odd(X) |
  evenSquare(Xs1,Ys). (21)
```

Clauses (20) and (21) have replaced Clauses (14) to (17).

## 8. Discussions and Conclusions

The last section discusses three aspects of the technicalities presented above: applicability, presentation, and justification of the rules.

*Applicability of the rules.* Our transformation technique is interesting in that it can be used for programs with interaction. Both programs with two-way (demand-driven) communication and programs with one-way communication (pipelining) can be handled. The current set of rules is a fundamental tool for the simple improvement of programs, and various techniques proposed for ordinary logic programs could be adapted for inclusion in an enhanced set of rules. We did not argue the transformation of guard goals, but it could be introduced rather independently.

*Presentation of the rules.* The presentation of the rules should be interesting in its own right. Firstly,

they have been simplified owing to the use of normal forms. Secondly, the use of idempotent substitutions with preferred directions of bindings was helpful in formalizing the rules. This suggests that substitutions with such properties are an appropriate tool for modeling information in (concurrent) logic programming. The presentation using the algebra of terms and substitutions shows a good conformity with the synchronization rule of GHC. However, the use of unification for interprocess communication brings about some difficulty also. That is, the semantics and implementations must provide against the failure of unification, while a GHC program that fails is usually regarded as erroneous. The possibility of failure seems to be the price of flexible interprocess communication realized by unification.

*Justification of the rules.* We have given a simple semantics based on transactions with which to justify the rules. A transaction is a finite fragment of a computation. It is abstract enough, and because of its finite nature we could use existent tools to describe and reason about it. A transaction is a natural unit of computation from users' point of view. The strengths of our semantic model are that it handles essentially non-terminating programs and that it handles anomalous behaviors in contrast with the success set semantics of logic programs. These are very important for interactive programs. The current rules are designed so that they preserve all the behaviors in the absence of the possibility of failure. They could be simplified further if they were allowed to diminish the possibility of anomalous transactions like deadlock.

In designing the semantic model, we were faced with two problems: the modeling of failing computations (as described above) and the treatment of fairness. As for the latter, we chose to allow execution unfair with respect to the selection of body goals reduced from the initial goals. As a consequence, a normal transaction or failure that would necessarily happen under fair execution may not happen, and divergence that would not happen under fair execution may happen. This choice contributes much to the simplicity of the semantics. While a model of fair execution will be need as well, the proposed semantics should be useful because it is still controversial whether fairness should be assumed by the language rules and some implementations of GHC adopt unfair scheduling. Partial orders of transactions may be a better alternative to sequences since we need not consider the fairness of interleaving, but it is yet to be studied how to use partial orders in our framework in which information is modeled as substitutions.

#### Acknowledgments

The authors are indebted to Masaki Murakami and Akira Okumura for valuable discussions.

#### References

- Brock, J. D. and Ackerman, W. B. (1981) Scenarios: A Model of Non-determinate Computation. In *Formalization of Programming Concepts*, Diaz, J. and Ramos, I. (eds.), LNCS 107, Springer-Verlag, pp. 252-259.
- Furukawa, K., Okumura, A. and Murakami, M. (1987) Unfolding Rules for GHC programs. In *Proc. Workshop on Partial and Mixed Computation*, Bjørner, D. et al. (eds.), Gl. Avernæs, Denmark.
- Furukawa, K. and Ueda, K. (1985) GHC Process Fusion by Program Transformation. In *Second Conf. Proc. Japan Soc. Softw. Sci. Tech.*, pp. 89-92.
- Kanamori, T. and Fujita, H. (1986) Unfold/fold Transformation of Logic Programs with Counters. ICOT Tech. Report TR-179, ICOT, Tokyo.
- Lassez, J. -L., Maher, M. J. and Marriott, K. (1987) Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, Minker, J. (ed.), Morgan Kaufmann, pp. 587-625.
- Lichtenstein, Y., Codish, M. and Shapiro, E. (1987) Representation and Enumeration of Flat Concurrent Prolog Computations. In [Shapiro 1987], Chapter 27.
- Lloyd, J. W. (1984) *Foundations of Logic Programming*. Springer-Verlag.
- Robinson, J. A. (1979) *Logic: Form and Function*. Edinburgh University Press.
- Shapiro, E. Y. (ed.) (1987), *Concurrent Prolog: Collected Papers*, Vol. 1-2, The MIT Press.
- Tamaki, H. and Sato, T. (1984) Unfold/Fold Transformation of Logic Programs. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 127-138.
- Tamaki, H. (1987) Program Transformation in Logic Programming Languages. In *Program Transformation*, Fuchi, K. (editor-in-chief), Kyoritsu Shuppan, Tokyo, pp. 39-62 (in Japanese).
- Ueda, K. (1985) Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo (revised in 1986). Revised version in *Proc. Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, 1986, pp. 168-179. Also in [Shapiro 1987], Chapter 4.
- Ueda, K. (1986a) Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, ICOT Tech. Report TR-208, ICOT, Tokyo (revised in 1987). Also in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, 1988, pp. 441-456.
- Ueda, K. (1986b) Introduction to Guarded Horn Clauses. ICOT Tech. Report TR-209, ICOT, Tokyo.
- Ueda, K. and Furukawa, K. (1989) Transformation Rules for GHC Programs. To appear as ICOT Tech. Report, ICOT, Tokyo.