ソフトウェア基礎論 19−10
プログラミング言語 9−10
（1986. 12. 12）

GHCプログラムの意味について
TOWARDS A SEMANTIC MODEL OF GHC

竹内彰一
Akikazu Takeuchi
三菱電機 中央研究所
Central Research Laboratory, Mitsubishi Electric Corporation

あらまし　GHCにおける計算の意味は入力サブスティチューションと出力サブスティチューションの間の対応関係としてとらえることができる。しかし一般に非決定性をもつ言語の意味としては入出力関係だけでは不十分なことが知られている。本論文は入出力関係の他に入出力要素間の因果関係も抽出するBrockらのシナリオ集合モデルに基づくGHCの意味モデルを与える。このモデルは宣言的デバッギングや等価変換等に適したものになっている。

Abstract. A semantic model of GHC programs based on the scenario set model of Brock *et al.* is presented. It provides as a meaning of a program a set of input–output histories each associated with causality relation between input and output substitutions and can be regarded as a basis for developing theory of parallel logic programming such as declarative debugging and equivalence preserving transformation.

1 Introduction

The semantics of logic programs has been extensively investigated [EK76], [AE82], [LL84]. These provide a rigid basis for various mathematical manipulations of logic programs such as program verification, equivalent program transformation and declarative debugging. Logical foundations for parallel logic programming languages are also indispensable for the development of the theory of parallel logic programming including verification, transformation and debugging. However, the results for pure logic programs are not directly applicable to parallel logic programming languages because of the new control primitives such as commitment.

Given a program P (a set of Horn clause), the success set of the program is defined to be the set of all A in the Herbrand base of P such that $P \cup \{\leftarrow A\}$ has an SLD-refutation. The finite-failure set is defined to be the set of all A in the Herbrand base of P such that there exists a finitely-failed SLD-tree with $\leftarrow A$ as root. It is well known that the success set, the least model and the least fixpoint of the function associated with the program are equivalent. The finite-failure set is characterized by the greatest fixpoint under a certain condition. If a goal succeeds under sound computation rules, the result is assumed of being included in the success set. If a goal finitely fails, then the result is ensured to be included in the finite-failure set.

Some literatures [CG84] recommends reading a guarded clause as just a Horn clause. This is sufficient as long as a goal succeeds, but this does not happen sufficient in many cases. Suppose that a goal failed. This implies neither that the result is not in the success set, nor that the result is in the finite-failure set, since the goal may fail even if there is a possibility of success because of commitment to an inappropriate clause. Such semantics becomes insufficient also if two programs with different input/output behavior need to be distinguished.

Parallel logic programming languages have two control primitives not appearing in pure logic programs. These are a commitment operator and a synchronization primitive. Parallel logic programming relies heavily on these control primitives. However, a commitment operator changes the semantics of failure and a synchronization primitive introduces procedural flavor. It is now obvious that declarative semantics for pure logic programs cannot characterize such aspects of parallel logic programs as failure and input/output behavior.

Let us consider the algorithmic debugging for parallel logic programming languages, where the intended interpretation of a program plays an important role in guiding debugging. Declarative semantics such as a success set is no longer sufficient. Intended interpretations should be abstract semantics characterizing all aspects which programmers intend to express. The author developed an algorithmic debugger for GHC, where the intended interpretation with procedural flavor of a GHC program was defined [TA86]. Lloyd and Takeuchi refined the framework for the above algorithmic debugging and discussed some difficult cases to handle [LT86].

Another application of the semantics of parallel logic programs is to establish the equivalence relation between two programs, one of which is, for example, the result of fold/unfold transformation to the other. The way to establish such equivalence is to demonstrate that the meanings of two programs are the same. Tamaki and Sato [TS84] have proved that their program transformation system of logic programs has equivalence preserving property using the following scenario. First they adopt the least model semantics as meaning of a logic program. Secondly in order to demonstrate the equivalence of two models they show that a ground literal *A* in one model is included in the another model by constructing a proof tree from the proof tree of *A* in the former program and vise varsa.

The above framework can be applicable to establish equivalence relation in parallel logic programs. However, there are two problems; One is [illegible]eaning of a parallel logic program. The other relates to concept of a proof tree. As for meaning of a parallel logic program, the first approximation is to introduce input/output concept. It leads to total history model of Kahn [KA74] which was developed for deterministic functional languages.

For a pure logic program, the success set, the least model and the least fixpoint of the function associated with the program are equivalent. The success set is a set of ground literals that can be finitely derived from the program. It is possible to imagine the success set of a parallel logic program. A success set of a parallel logic program corresponds to a set of literals which are final forms of goals after the computation successfully terminates. Usually it is called a set of total histories since the final form of a goabl represents in itself all the inputs received and all the output sent. However, it is known that a set of total histories is insufficient for the semantics of nondeterministic data flow languages [BA81]. The example used to prove this proposition is also valid in parallel logic programs. Hence, the set of total histories is insufficient for the semantics of parallel logic progams. We illustrate this using the same example in GHC form in [BA81].

Anomaly:

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

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

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

merge([A|Ix],Iy,O) :-
        true | O=[A|Out], merge(Ix,Iy,Out).
merge(Ix,[A|Iy],O) :-
        true | O=[A|Out], merge(Ix,Iy,Out).
merge(Ix,[],O) :- true | Ix=O.
merge([],Iy,O) :- true | Iy=O.
```

For i=1,2

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

For i=1,2

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

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

The first arguments of **p1** and **p2** are used as input ports and the second arguments as output ports. In p_1, the first two elements of a list received are output one by one as they are received. On the other hand, in p_2, the first two elements are output after both elements are received. s_i has two input ports, the first and second arguments, and one output port, the third argument. Internally it invokes four goals, two **dup**'s, **merge** and p_i. **dup** outputs doubleton, the elements of which are duplicates of the first element of the list received. The total history set of s_i is:

```
{si([X|Ix],_,[X,X]), si(_,[Y|Iy],[Y,Y]),
 si([X|Ix],[Y|Iy],[X,Y]),
 si([X|Ix],[Y|Iy],[Y,X])}
```

where X and Y denote the first element of lists received at the first and second arguments, respectively. Note that the total history set reveals several possibilities of outputs for the same input, which results from the nondeterminism of the merge operator. t_i consists of s_i and **plus1** where the output of s_i is connected to the second input port of s_i through **plus1**. **plus1** outputs a singleton at the second argument, the element of which is equal to the value of the first element of the first argument plus one. The total history set of t_1 when it receives [5] is:

```
{t1([5],[5,5]), t1([5],[5,6])}
```

However, the total history set of t_2 is:

```
{t1([5],[5,5])}.
```

Consequently, the example illustrates that, even if the meanings of the intermediate modules (s_i) are equivalent with respect to the total history set, the meanings of the entire modules (t_i) can be different in the sense of the total history set. It implies that the extraction of the meaning in terms of the total history set is insufficient. The difficulty in modelling computation of parallel logic programs and nondeterministic data flow languages results from their nondeterminism introduced by, for example, the merge operation.

What we need is not only total histories, but also information about dynamic behavior of a program. The latter information should be abstract enough in order not to extract implementation detail. There are two preceding works. Namely the scenario set model by Brock and Ackermann [BA81] and the fixpoint model by Staples and Nguyen [SN85]. The first approach extracts causality relation between input and output data. The second extracts transition relation (partial order) among subcomputations.

In this paper, we present a semantic model of GHC programs. It is based on scenario set model and it provides as a meaning of a program a set of total histories each associated with causality relation between input and output data. The method to construct semantic model is procedural from the viewpoint of logic programming, but it appears unavoidable.

The section 2 briefly introduces GHC. In section 3, the concept of a derivation tree is introduced. A derivation tree is defined syntactically and constructed from a program. The concept of causality relation between input and output is also defined over a derivation tree. The section 4 establishes the tight relation between a feasible derivation tree and a trace tree. We define a meaning of a program in the section 5. The anomaly presented in this section is revisited in section 6 with our new semantic model and it is shown that the new model can discriminate two modules with the same total history sets. The section 7 discussed about a couple of applications of this new semantic model. Related works are discussed in section 8.

2 Guarded Horn Clauses

Guarded Horn Clauses (hereafter GHC) is a parallel logic programming language [UE85]. A program of GHC consists of a finite set of guarded clauses. A guarded clause has the form:

$$H :- G_1, ..., G_n \mid B_1, ..., B_m.$$

where "H", "$G_1, ..., G_n$" and "$B_1, ..., B_m$" are called the head, the guard part and the body part of the clause, respectively. Each G_i and B_j have the form, $P(T_1, ..., T_k)$ ($k \geq 0$), *true* or $U = V$, where "=" is unification. We do not deal with other built-in predicates than a unification and *true* in this paper.

Without loss of generality, we assume that a head of a guarded Horn clause has a *skeletal form*, that is, all the arguments are distinct variables.

Definition 1 *A substitution set is a set of the form:*

$$\{v_1/t_1, \ldots, v_n/t_n\}$$

where v_i is a variable and t_i is a term.

3 A Feasible Derivation Tree

3.1 A derivation tree

We introduce a derivation tree for a program P. A derivation tree is a syntactic object and there is no explicit relation to real computation in this section. In later section, we will establish tight relation between a derivation tree and a trace tree of an actual computation.

Definition 2 *Let a node be a triplet (R, θ_I, θ_O). R is a skeletal predicate $p(v_1, \ldots, v_n)$ where p is a predicate symbol and $v_1, \ldots, v_n$ are distinct variables. θ_I and θ_O are both substitution sets applicable to R.*

Given a program P, we say that a tree T is ***a derivation tree*** *constructed from P iff*

(1) T is a node $(true, \phi, \phi)$, or

(2) T is a node $(X = Y, \theta_I, \theta_O)$, or

(3) T is a node (R, θ_I, θ_O), where R is a skeletal predicate appearing in P. θ_I and θ_O are substitution sets.

(4) T is a tree for which there exists a clause $H :- G_1, ..., G_n | B_1, ..., B_m$ in P satisfying the following conditions, $\Gamma 1$ and $\Gamma 2$.
Let $Tg_1, ..., Tg_n$, $T_1, ..., T_m$ be immediate substrees of T and $V_1, ..., V_n, W_1, ..., W_m$ be their root nodes.
Let (H, θ_I, θ_O) be a root of T.

($\Gamma 1$) $Tg_1, ..., Tg_n$ are derivation trees such that
for $i = 1, ..., n$
$V_i = (skel(G_i), \sigma(G_i) \cdot \theta_I \cdot \gamma_{I_i}, \gamma_{O_i})$,
$\gamma_{I_i} = \bigcup_{j \neq i} \gamma_{O_j}$,
$\gamma_{O_i} \parallel H$,

($\Gamma 2$) $T_1, \ldots, T_m$ are derivation trees such that
for $i = 1, ..., m$
$W_i = (skel(B_i), \sigma(B_i) \cdot \theta_I \cdot \gamma_O \cdot \beta_{I_i}, \beta_{O_i})$,
$\gamma_O = \bigcup_{i=1,..,n} \gamma_{O_i}$,
$\beta_{I_i} = \bigcup_{j \neq i} \beta_{O_j}$,
$\theta_O = \bigcup_{i=1,..,m} \beta_{O_i}$

$\theta_I, \theta_O, \gamma_{I_i}, \gamma_{O_i}, \beta_{I_i}, \beta_{O_i}$ are all substitution sets. $skel(F)$ is a skeletal form of F and $\sigma(F)$ is a substitution set which creates F when applied to the skeletal form of F. $S \parallel T$ means that no element of substitution set S substitutes variables in a term T. Note that $\gamma_{O_i} \parallel H$ in the above definition express synchronization condition of GHC.

Here we show some examples.

Example 1

Program

```
p(X) :- true | q(X,Y,Z), r(Y,Z).
q(X,Y,Z) :- X=a | Y=b, qq(Z).
qq(Z) :- Z=c | true.
r(Y,Z) :- Y=b | Z=c.
```

Derivation tree

```
              (p(X),{X/a},{})
                /          \
  (q(X,Y,Z),                  (r(Y,Z),
    {X/a,Z/c},{Y/b})            {Y/b},{Z/c})
         |
  (qq(Z),{Z/c},{})
```

Example 2

Program

```
p2 :- true | q(X,Y,Z), r2(X,Y,Z).
q(X,Y,Z) :- X=a | Y=b, qq(Z).
qq(Z) :- Z=c | true.
r2(X,Y,Z) :- Y=b | X=a, Z=c.
```

Derivation tree

```
             (p2,{},{})
               /     \
 (q(X,Y,Z),             (r2(X,Y,Z),
   {X/a,Z/c},{Y/b})       {Y/b},{X/a,Z/c})
         |
 (qq(Z),{Z/c},{})
```

In the first example, the derivation tree is similar to the trace tree of an actual computation invoked by p(a). However, in the second example, no computation corresponds to the derivation tree. The reason why such derivation tree can be constructed is that the definition of a derivation tree does not consider the causality relation between subcomputations. In the next, we introduce the causality relation among subcomputations as those of input and output substitution sets.

3.2 Causality relation among substitutions

In our derivation tree modelling, both input and output are substitution sets. Between elements of an input substitution set and elements of an output substitution set, there exists causality relation inferred from the least input substitution set for commitment.

Definition 3 *Let T and U be a derivation tree and a node in T, respectively. Let θ_I be a subset of the input substitution set at the node U.*

We say $\theta_I \xrightarrow{0} \theta_O$ at U in T iff θ_I is the least input substitution set for the commitment to the clause invoked at U and θ_O is equal to the set of all substitutions created by unifications $(=)$ in the body of the clause.

Definition 4 *Let T and U be a derivation tree and a node in T, respectively. Let θ_I be a subset of the input substitution set at the node U.*

We say θ_I at U **causes** *θ_O iff*

(1) $\theta_I \xrightarrow{0} \theta_O$ at U

(2) $\theta_I \xrightarrow{0} \psi$ at U, and $\psi'(\subseteq \psi)$ at V **causes** *θ_O, where V is one of immediate descendants of U.*

(3) $\theta_I = \psi \bigcup \chi$, $\psi \bigcap \chi = \phi$, ψ at U **causes** *ψ', χ at U* **causes** *χ', and $\psi' \bigcup \chi'$ at V* **causes** *θ_O, where V is one of descendants of U in T.*

α at U **causes** β is denoted by $\alpha@U \rightarrow \beta$. Note that β is a set of all substitutions which can be generated by unifications at some clause and that α is the least set which can generate β. Terms literally placed at argument positions of a goal are regarded as parts of an input substitution set. Therefore, as well as substitutions that are generated at run-time, such fixed term patterns are also regarded as contributing to cause output substitutions.

Definition 5 *Let T and U be a derivation tree and a node in T, respectively. Let θ_I and θ_O be subsets of input and output substitution sets at the node U.*

We say $\theta_I \mapsto \theta_O$ at U iff $\theta_I@U \rightarrow \theta'$ and θ_O is a subset of θ' including all the substitutions relevant to the node U.

Relation $\mapsto$ introduces causality structure into input and output substitution sets of a node.

Proposition 1 *Given a derivation tree, for any node (R, θ_I, θ_O) in the derivation tree, there exists decomposition $\{\theta_{O_1}, \ldots, \theta_{O_n}\}$ (i.e., $\theta_{O_i} \bigcap \theta_{O_j} = \phi$ and $\bigcup_{i=1}^{n} \theta_{O_i} = \theta_O$) of θ_O such that there exists $\{\theta_{I_1}, \ldots, \theta_{I_n}\}$ such that θ_{I_i} (possibly empty) $\subset \theta_I$ and $\theta_{I_i} \mapsto \theta_{O_i}$.*

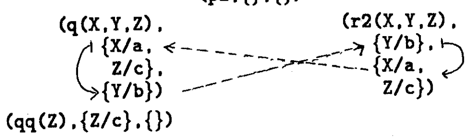
In other words, a pair of θ_{I_i} and θ_{O_i} is a causally dependent pair. Note that $\bigcup_{i=1}^{n} \theta_{I_i} \subseteq \theta_I$. $\theta_I - \bigcup_{i=1}^{n} \theta_{I_i}$ is necessary for advancing computation, but it creates no output substitution.

We define a causally consistent derivation tree

Definition 6 *Let N be a node in a derivation tree, $(R_i, \theta_{I_i}, \theta_{O_i})$ $(i = 1, \ldots, n)$ be immediate descendants of N. We say N is* **causally consistent** *iff there is no directed loop among subsets of θ_{I_i} and θ_{O_i} with respect to $\mapsto$ relation.*

Definition 7 *We say a derivation tree T is causally consistent iff every node in T is* **causally consistent.**

Example 2 illustrated above is not a causally consistent derivation tree.



3.3 A feasible derivation tree

Definition 8 *A derivation tree T is called a* **guard tree** *iff*

(1) the root of T corresponds to a goal in a guard part of a clause, or

(2) T is a subtree of a guard tree.

Definition 9 *A node with label U is a* ***success node*** *iff*

(1) $P = (true, \phi, \phi)$, or

(2) $P = (X = Y, \theta_I, \theta_O)$ where $X \cdot \theta_I \cdot \theta_O$ is equivalent to $Y \cdot \theta_I \cdot \theta_O$

Definition 10 *A derivation tree T is a* ***success derivation tree*** *iff*

(1) T is finite, and

(2) T is causally consistent, and

(3) all its leaf nodes are success nodes.

Definition 11 *A node with label U is a* ***failure node*** *iff*

(1) $U = (X = Y, \theta_I, -)$ where there is no substitution making $X \cdot \theta_I$ and $Y \cdot \theta_I$ identical, or

(2) a node which has no clause satisfying the condition (T1) such that all its associated guard trees are finite and causally consistent.

Analyzing the case (2), we can classify the second type of failure into the following three. A node is called an ***infinite guard*** iff there is at least one clause which only violates finite condition. A node is called ***blocking*** iff it is not an infinite guard and there is at least one clause which only violates consistency condition. A node is called ***finite failure*** iff it is neither blocking nor infinite.

Definition 12 *A derivation tree T is a* ***failure derivation tree*** *iff*

(1) T is finite and causally consistent, and

(2) all its guard trees are success derivation trees, and

(3) there exists at least one failure node except in the guard tree.

Note that a failure node in a failure derivation tree is either a blocking or a finite failure node.

Definition 13 *We say a failure derivation tree is a* ***finite failure derivation tree*** *iff at least one of its failure nodes is finite failure.*

Definition 14 *We say a failure derivation tree is a* ***suspension derivation tree*** *iff all the failure nodes are blocking.*

Definition 15 *We say a derivation tree T is a* ***feasible derivation tree*** *iff T is a success derivation tree, a finite failure derivation tree or a suspension derivation tree.*

4 A Feasible Derivation Tree = A Trace Tree

So far, we have syntactically defined a feasible derivation tree. In this section we prove that there is a clear correspondence between a feasible derivation tree and a trace tree formed by an actual computation. First we define a trace tree which is slightly different from ordinary definition.

A trace tree is an AND tree formed by a computation. Each node is represented by triplet (R, θ_I, θ_O), where R is a head of the clause invoked at the node. θ_I and θ_O are substitution sets comming from outside and comming up from inside, respectively.

Theorem 1 *Let P and A be a program and a goal. A trace tree of any legal finite computation invoked by A on P is a feasible derivation tree constructed from P.*

It can be easily proved from the computation rule and the immutableness of trace.

In some sense, the above theorem states about the soundness of a feasible derivation tree. As completeness of a feasible derivation tree, we obtain the following theorem.

Theorem 2 *Let P and A be a program and a goal. For any feasible derivation tree rooted at A constructed from P, there exsits a computation invoked by A on P, the trace tree of which is equivalent to the tree.*

Proof: It will be demonstrated by giving a sequence of scheduling which creates the same trace tree as the given derivation tree. By the term *scheduling*, we mean the selection of a goal for resolution from a set of goals. A sequence of scheduling is a sequence of selection of goals. Let T be a feasible derivation tree whose root is A. For T, we give a sequence of scheduling by induction on the height of T. We omit the trivial nodes such as $(true, \phi, \phi)$ and $(X = Y, \theta_I, \theta_O)$. In other words, *true* and unification nodes are counted as 0.

Base case: Height is equal to 1. Suppose that T consists of a single node $A = (R, \theta_I, \theta_O)$ and let $R : -G_1, \ldots, G_n | B_1, \ldots, B_m$ be a clause used at A in the feasible derivation tree. Note that $\{G_i\}$ and $\{B_j\}$ are sets of unifications that satisfy θ_I and θ_O, respectively. In the computation invoked by the goal $R \cdot \theta_I$, clearly the scheduler selects A and the computation creates the same trace tree as T when the clause $R : -G_1, \ldots, G_n | B_1, \ldots, B_m$ is invoked.

Induction step: Assume that, for any feasible derivation tree of height $k (< N)$, given the same goal as the root of the tree provided with the whole input substitutions, then there exists a sequence of scheduling which creates the same trace as the feasible derivation tree.

Consider the feasible derivation tree T of height $N + 1$. Let $A = (H, \theta_I, \theta_O)$ be a root of T and $Tg_1, \ldots, Tg_n, T_1, \ldots, T_m$ be immediate substrees. Let Q_i and R_j be roots of Tg_i and T_j, respectively. Let C be a clause used at A.

Given a goal $H \cdot \theta_I$, it is the first goal to be selected and C can be invoked. From the induction hypothesis, for each $Tg_i (i = 1, \ldots, n)$ and $T_j (j = 1, \ldots, m)$, there exists a sequence of scheduling that creates the same trace tree as it, provided that the goal is invoked with the whole input substitutions.

From the following proposition and that there is no loop in the directed graph specified by $\mapsto$ relation over subsets of input and output substitution sets of Q_i and R_j, we can interleave the sequences of scheduling in the way causal ordering over substitutions is preserved in the interleaved sequence of scheduling.

Therefore the top-level goal followed by the interleaved sequence is the sequence of scheduling of T. ∎

Proposition 2 *For a goal, sequence of scheduling is divided into set of subsequences each associated with causality pair of input and output substitution sets.*

5 Meaning of a Program

Since what we are concerned with as a meaning of a program is only behavior of a program which can be observable from outside, we abstract our model, namely a success derivation tree, a suspension derivation tree and a finite failure derivation tree.

Given two substitution sets θ_I and θ_O at some node, causality structure between θ_I and θ_O can be specified by the $\mapsto$ relation over pairs of subsets of θ_I and θ_O. Instead of considering the whole derivation trees, it is enough to consider the $\mapsto$ relation between input and output substitutions.

A success set $Msuc(P)$ of a program P is defined to be a set of triplets (G,θ_I,θ_O)'s which have success derivation trees, together with the $\mapsto$ relations over subsets of θ_I and θ_O.

Definition 16
$Msuc(P) =$

$$\left\{ (H,\theta_I,\theta_O,\mapsto) \;\middle|\; \begin{array}{l} (H,\theta_I,\theta_O) \text{ has a success} \\ \textit{derivation tree with } \mapsto \\ \text{relation over } \theta_I \text{ and } \theta_O \end{array} \right\}$$

A suspension set $Msus(P)$ and a finite failure set $Mf(P)$ of a program P are defined in the same way.

Definition 17
$Msus(P) =$

$$\left\{ (H,\theta_I,\theta_O,\mapsto) \;\middle|\; \begin{array}{l} (H,\theta_I,\theta_O) \text{ has a suspension} \\ \textit{derivation tree with } \mapsto \\ \text{relation over } \theta_I \text{ and } \theta_O \end{array} \right\}$$

Definition 18
$Mf(P) =$

$$\left\{ (H,\theta_I,\theta_O,\mapsto) \;\middle|\; \begin{array}{l} (H,\theta_I,\theta_O) \text{ has a finite failure} \\ \text{derivation tree with } \mapsto \\ \text{relation over } \theta_I \text{ and } \theta_O \end{array} \right\}$$

Now the meaning of a program P is defined to be a triplet of these three sets.

Definition 19

$$M(P) = (Msuc(P), Msus(P), Mf(P))$$

The definition corresponds to a success set and a finite failure set modelling of semantics of logic programs. The new definition is augmented by $Msus(P)$, since suspension is an important class of results of computation in GHC. Furthermore, since, compared with Horn clauses, concept of commitment and notion of input and output are introduced in GHC, the semantics of GHC is augmented by notion of input and output and causality relation among them. It should be also noted that some elements belong to two or three sets, which reflects nondeterministic choice mechanism of GHC.

6 Anomaly Revisited

Let us see the anomaly again in order to see how the problem can be solved in our model. The problem was that by the total history set model we cannot discriminate $\mathbf{s}_1$ and $\mathbf{s}_2$. The definition of $\mathbf{s}_i$ is shown below.

```
s_i(Ix, Iy, Out) :- true |
    dup(Ix, Ox), dup(Iy, Oy),
    merge(Ox, Oy, Oz), p_i(Oz, Out).
```

In the derivation tree of $\mathbf{s}_1$`([A|_],[B|_],[A,B])`:

$$\begin{array}{ll}
\{\texttt{Ix/[A|\_]}\} \rightarrow \{\texttt{Ox/[A|M1],M1/[A|M2],M2/[]}\} & \textit{in dup} \\
\{\texttt{Ox/[A|M1]}\} \rightarrow \{\texttt{Oz/[A|M3]}\} & \textit{in merge} \\
\{\texttt{Oz/[A|M3]}\} \rightarrow \{\texttt{Out/[A|M4]}\} & \textit{in } p_1
\end{array}$$

$$\begin{array}{ll}
\{\texttt{Iy/[B|\_]}\} \rightarrow \{\texttt{Oy/[B|N1],N1/[B|N2],N2/[]}\} & \textit{in dup} \\
\{\texttt{Oy/[B|N1]}\} \rightarrow \{\texttt{M3/[B|N3]}\} & \textit{in merge} \\
\{\texttt{M3/[B|N3]}\} \rightarrow \{\texttt{M4/[B|N4],N4/[]}\} & \textit{in } p_1
\end{array}$$

(Note that the variables, M3 and M4, appearing twice in causality relation in **merge** and $\mathbf{p}_1$, respectively, reflect the temporal order between causality relations.) Therefore at the node ($\mathbf{s}_1$`(Ix,Iy,Out)`,$\{$`Ix/[A|_],Iy/[B|_]`$\}$, $\{$`Out/[A,B]`$\}$):

$$\begin{array}{l}
\{\texttt{Ix/[A|\_]}\} \mapsto \{\texttt{Out/[A|M4]}\} \\
\{\texttt{Iy/[B|\_]}\} \mapsto \{\texttt{M4/[B|N4],N4/[]}\}
\end{array}$$

On the other hand, in the derivation tree of $\mathbf{s}_2$`([A|_],[B|_], [A,B])`:

$$\begin{array}{ll}
\{\texttt{Ix/[A|\_]}\} \rightarrow \{\texttt{Ox/[A|M1],M1/[A|M2];M2/[]}\} & \textit{in dup} \\
\{\texttt{Ox/[A|M1]}\} \rightarrow \{\texttt{Oz/[A|M3]}\} & \textit{in merge} \\
\{\texttt{Iy/[B|\_]}\} \rightarrow \{\texttt{Oy/[B|N1],N1/[B|N2],N2/[]}\} & \textit{in dup} \\
\{\texttt{Oy/[B|N1]}\} \rightarrow \{\texttt{M3/[B|N3]}\} & \textit{in merge}[2] \\
\{\texttt{Oz/[A|M3],M3/[B|N3]}\} & \\
\qquad \rightarrow \{\texttt{Out/[A|M4],M4/[B|N4],N4/[]}\} & \textit{in } p_2
\end{array}$$

Therefore at the node ($\mathbf{s}_2$`(Ix,Iy,Out)`, $\{$`Ix/[A|_],Iy/[B|_]`$\}$, $\{$`Out/[A,B]`$\}$):

$$\{\texttt{Ix/[A|\_]}\} \bigcup \{\texttt{Iy/[B|\_]}\} \mapsto \{\texttt{Out/[A|M4],M4/[B|N4],N4/[]}\}$$

Now it is obvious that $\mathbf{s}_1$`([A|_], [B|_], [A,B])` and $\mathbf{s}_2$`([A|_], [B|_], [A,B])` have different causality structures, that is, $\mathbf{s}_1$ and $\mathbf{s}_2$ have different meanings.

7 Applications

As important applications of our semantics model of GHC, we briefly mention about two projects, which the author is involved in.

The first is an algorithmic debugging of GHC programs. The author developed an algothmic debugger of GHC programs [TA86] and its formal framework was refined in [LT86]. In [LT86], it is proved that the debugger can find a bug, provided that some condition holds. The paper discusses about some critical cases which the debugger cannot handle when computation results in an unexpected result.

The reason why this happens is regarded as that an intended interpretation defined in [TA86] is rough compared with our new model. By adopting our new semantics model as an intended interpretation of GHC programs, it is expected to be able to prove the stronger theorem, which guarantees that the debugger can find a bug when a user happens to know in a natural sense that computation results in an erroneous result.

The second is the equivalence transformation of GHC programs. It is obvious that we need some semantic model to discuss about the equivalence preserving property of a transformation system, however, currently we have no such formal model. Our semantic model provides a basis for discussion about equivalence of two programs. One of the advantages of our approach is that each computation is abstracted as a feasible derivation tree. This is useful for proving that one computation in one program is possible in the other program in terms of induction on the computation tree, which is the main technique used when proving the equivalence preserving property of fold/unfold transformation system of logic programs by Tamaki and Sato [TS84]. There are many things to be done for this application and discussion about them is out of scope of this paper.

8 Related works

As mentioned earlier in this paper, there are two preceding works on semantic models of nondeterminstic parallel languages. In this section we briefly review these works in relation to our model.

8.1 A Scenario Set Model [BA81]

A scenario model was proposed as a formal model for a nondeterminate dataflow computation. The definition of a scenario is: A scenario is a pair consisting of an input stream tuple and an output stream tuple, together with a causality order relating each element of the input and output streams to those elements which played a role in its creation. The advantage of the scenario model is that it is static and it can be found in the whole history of computation.

A scenario model can be extended to a model for parallel logic programming languages. In fact, our model is based on this model. In our model, input and output stream tuples are replaced by input and output substitutions. A meaning of a GHC program is given as a set of literals with causality order relating each element of input and output substitutions to those elements which played a role in its creation. Causality relation can be syntactically extracted from a clause by defining primitive causality which relates input substitutions contributing to the commitment to the clause to output substitutions created in the body part of the clause.

8.2 Fixpoint Model [SN85]

The model consists of the partially ordered multiset of input-output histories. An input-output history may be incomplete which corresponds to computation not yet terminating. To one (possibly incomplete) input-output history, many states of computation may correspond, since an input-output history is abstract view for computation. Partial order between two elements in the multiset is interpreted to mean that the least computation achieving one history can be extended to the least computation achieving the other. The disadvantage of this model is that the model captures dynamic property of computation. It may make the demonstration of equivalence of two programs difficult. For the demonstration, it is desired that we could focus on one element of a set at a time. But in this model, when we discuss about behavior of a program, we have to always consider all the possible input-output histories with partial order, since this partially order structure is the way this model characterizes nondeterministic computation.

8.3 Relation between two models

Usually nondeterministic computation can be characterized as an AND-OR tree, where nondeterministic choices are modelled as OR branches. Scenario set model captures the AND tree component of the tree, in which choices made are reflected as causality relation among input and output at a node. Fixpoint model captures the nondeterminism in the form of OR tree, the nodes of which correspond to ongoing (AND) computations. In this approach, choice points are represented explicitly as nodes in the OR tree. But each node carries no information about its history.

9 Conclusion

We have presented a semantic model of GHC programs. It is based on scenario set model and provides as a meaning of a program a set of total histories each associated with causality relation between input and output data. A feasible derivation tree can be syntactically constructed from a program. In this sense, a feasible derivation tree is a syntactic object. However, in section

4, we have shown that there is a tight relation between a feasible derivation tree and a trace tree. Therefore a feasible derivation tree can be regarded as an abstract image of computation.

Owing to the fact that a feasible derivation tree can be constructed syntactically from a program, we can extract a meaning of a program statically. This will help the development of theory of parallel logic programming such as declarative debugging and equivalence preserving transformation.

We believe that the framework can be also applicable to other parallel logic languages such as PARLOG [CG84], Oc [HI85] and Concurrent Prolog [SH83]. The current model is procedurally defined. Declarative reconstruction of the model and its applications to equivalence preserving transformation system and refinement of declarative debugging are the next research themes.

10 Acknowledgement

We would like to thank Kazunori Ueda and Hirohisa Seki for informing about two important papers and helpful discussion. We would also like to thank Koichi Furukawa and all the other members of 1st Lab. of ICOT, both for discussion and for providing a stimulating atmosphere in which to work.

References

[AE82] K. R. Apt and M. H. van Emden, Contributions to the Theory of Logic Programming, *J. ACM*, vol. 29, no. 3, 1982, pp. 841-862.

[BA81] J. D. Brock and W. B. Ackermann, Scenario: A Model of Nonderminate Computation, In *Formalization of Programming Concepts*, J. Diaz and I. Ramos (ed.), Lecture Notes in Computer Science, vol. 107, Springer-Verlag, 1981, pp. 252-259.

[CG84] K. L. Clark and S. Gregory, *PARLOG: Parallel Programming in Logic*, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London, 1984.

[EK76] M. H. van Emden and R. Kowalski, The Semantics of Predicate Logic as a Programming Language, *J. ACM*, Vol. 23, No. 4 (1976), pp. 733-742.

[HI85] M. Hirata, Self-Description of Oc and Its Applications, In *Proc. Second National Conf. of Japan Society of Software Science and Technology*, 1985, pp. 153-156. (in Japanese)

[KA74] G. Kahn, The Semantics of a Simple Language for Parallel Programming, *Proceedings of IFIP Congress 74*, J. L. Rosenfeld, Ed. , 1974, pp. 471-475.

[LL84] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[LT86] J. W. Lloyd and A. Takeuchi, *A Framework of Debugging GHC*, ICOT Tech. Report TR-186, Institute for New Generation Computer Technology, Tokyo, 1986.

[SH83] E. Y. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter*, ICOT Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.

[SN85] J. Staples, V. L. Nguyen, A Fixpoint Semantics for Nondeterministic Data Flow, *J. ACM*, Vol. 32, No. 2 (1985), pp. 411-444.

[TA86] A. Takeuchi, *Algorithmic Debugging of GHC programs*, ICOT Tech. Report TR-185, Institute for New Generation Computer Technology, Tokyo, 1986.

[TS84] H. Tamaki and T. Sato, Unfold/Fold Transformation of Logic Programs, In *Proc. The Second Inter. Conf. on Logic Programming*, Uppsala University, 1984, pp. 127-138.

[UE85] K. Ueda, *Guarded Horn Clauses*, ICOT Tech. Report TR-103, Institute for New Generation Computer Technology, 1985. Also appearing in *Lecture Notes in Computer Science*, Springer-Verlag, 1986.