TR-434

A Preliminary Note on the Semantics of Guarded Horn Clauses

by
T. Kanamori and M. Maeji(Mitsubishi)

December, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Preliminary Note on the Semantics of Guarded Horn Clauses

Tadashi KANAMORI   Machi MAEJI

Mitsubishi Electric Corporation
Central Research Laboratory
8-1-1 Tsukaguchi-Honmachi
Amagasaki, Hyogo, JAPAN 661

Abstract

This paper presents a preliminary formalization of the semantics of Guarded Horn Clauses (GHC) particularly emphasizing on a compositional character. First, the execution of GHC programs is described using some tree structures, in which not only *actually* successful computation but also *potentially* successful computation are considered. Next, by simply abstracting this description, the semantics of a class of GHC programs, called monotonic GHC programs, is studied. The semantics considers the set of pairs consisting of initial goals and their answers, and it is shown that the semantics enjoys a compositional character. Then, after examining the problems of non-monotonic GHC programs, the semantics of general GHC programs is investigated. The semantics considers the partially ordered multiset of such pairs, and it is shown that the semantics still enjoys the compositional character. Last, the semantics of GHC programs is compared with that of Prolog programs. A reformulation of Kahn's results about functional data flow networks within the framework of parallel logic programming for finite computation is presented as well in Appendix.

Keywords : Semantics, Parallel Logic Program, Guarded Horn Clauses, Nondeterminacy.

Contents

1. Introduction

When one tries to formalize the semantics of Guarded Horn Clauses (GHC) programs [38], he/she immediately gets at a loss how to treat various new computational phenomena which do not appear for Prolog programs. (The readers unfamiliar with GHC are recommended to read Section 2 first.)

- Because applying substitutions to the variables occurring in callers is inhibited (except trivial renaming of the variables) in the guards of GHC programs, some goal never succeeds as a GHC goal, even if it can succeed as a Prolog goal. How should the phenomena be treated?
- Some goal can succeed when other goals coexist, even if it never succeeds as a single goal. In general, some goal behaves in a quite different manner according to its environment, i.e., the behaviors of coexisting goals. How should the phenomena be considered?
- How should the effects of the commitment operator "|" be accommodated into the semantics?
- How should the "parallel" execution of GHC programs be reflected in the semantics? Should it be different from the "nondeterministic" execution (possibly under some assumption) of GHC programs? What should be the essential difference from the parallel or nondeterministic execution of Prolog programs?

Let us first review the above problems in detail.

(1) A more general goal may be suspended even if a more specific goal may succeed.

Example 1.1 Let "*merge*" be the predicate for nondeterministically merging two input streams into one output stream defined by the following GHC program:

```
merge([A|Xs],Ys,Zs) :- | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
merge(Xs,[A|Ys],Zs) :- | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
merge([ ],Ys,Zs) :- | Zs=Ys.
merge(Xs,[ ],Zs) :- | Zs=Xs.
```

(a) Then, the execution of goal "?- $merge([1,3],[2,4],[1,2,3,4])$" can succeed. (It succeeds if the execution is first committed to the 1st, 2nd and 1st clauses in this order. The execution after these commitments never fails whichever clauses it may be committed to.)

(b) The execution of goal "?- $merge([1,3],[2,4],Z)$" can return answer substitution $< Z \Leftarrow [1,2,3,4] >$. (It can return 5 other answer substitutions. The execution never fails whichever clauses it may be committed to.)

(c) The execution of goal "?- $merge([1|X],Y,[1,2,3,4])$" is suspended, because no guard is solved for the goal after the commitment to the first clause.

In general, if the execution of "?- A" succeeds with answer substitution σ, then the execution of "?- $A\sigma$" may possibly succeed, while the execution of "?- B" does not necessarily succeed even if the execution of "?- $B\tau$" succeeds. That is, a more specific goal may possibly succeed if a more general goal may succeed, but not vice versa. If we would consider each goal "?- A" simply as an instruction to search a goal instance $A\theta$ that is a first-order logical consequence of a given program, we could not discriminate cases (a), (b) from case (c).

(2) A more general answer does not necessarily subsume a more specific answer.

Example 1.2 Let "*produce-none*," "*produce-one*" and "*consume-one*" be the predicates defined by the following GHC program:

```
produce-none(L).
```

```
      produce-one(L) :- | L=[X|M].
      consume-one([X|M]).
```
Then, the execution of goal
```
      ?- produce-none(L)
```
succeeds with answer substitution <>, while the execution of goal
```
      ?- produce-one(L)
```
succeeds with answer substitution $< L \Leftarrow [X|M] >$. Although "*produce-none*(L)" returns a more general answer substitution than "*produce-one*(L)" does, it does not necessarily mean that it makes more goals succeed when combined with other goals. For example, the execution of goal
```
      ?- produce-none(L), consume-one(L)
```
is suspended, while the execution of goal
```
      ?- produce-one(L), consume-one(L)
```
succeeds. If we would consider more specific answer substitutions to be subsumed by more general answer substitutions, we could not capture the phenomena above.

(3) A goal may succeed through the interaction with coexisting goals.

A goal, which is suspended due to the insufficient instantiation of its arguments, can be released from the suspension by the supply of the (arguments form) information from coexisting goals. This phenomena shows that it is not appropriate to consider only the execution result of an individual goal for characterizing the semantics of the goal.

Example 1.3 Let "*seesaw*" be the predicate defined by the following GHC program:
```
      seesaw([N|In1],Out) :- N>1 | subtract(N,1,N1), Out=[N1|Out1], seesaw(In1,Out1).
      seesaw([1|In1],Out) :- | Out=[0|Out1].
      seesaw([0|In1],Out) :- | In1=[ ], Out=[ ].
```
seesaw(In, Out) receives natural number "N" from the input stream, send "$N-1$" to the output stream and continues the same cycle as long as the received natural number is greater than 1. It outputs 0 and stops when 1 is received, and it closes the streams and stops when 0 is received. Then, goal
```
      ?- seesaw([100|X],Y), seesaw(Y,X).
```
succeeds, while individual goals
```
      ?- seesaw([100|X],Y).
      ?- seesaw(Y,X).
```
will never succeed. Hence, it is not sufficient to consider each individual goal by itself.

(4) Just answer substitutions when executed as single goals do not characterize the predicates.

Is it appropriate to consider the input/output behaviors of all instances of a goal for characterizing the semantics of the goal? Even more complicated, two goals show different input/output behaviors when other goals coexist, even if any instances of these two goals show an identical input/output behaviors as single goals.

Example 1.4 Let "$p1$" and "$p2$" be the predicates defined by the following GHC program:
```
      p1(X,Y,Z) :- | double(X,XX), double(Y,YY), merge(XX,YY,W), one-by-one(W,Z).
      p2(X,Y,Z) :- | double(X,XX), double(Y,YY), merge(XX,YY,W), two-at-once(W,Z).
      double(0,AA) :- | AA=[0,0].
      double(1,AA) :- | AA=[1,1].
      merge([A|Xs],Ys,Zs) :- | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
```

```
    merge(Xs,[A|Ys],Zs) :- | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
    merge([ ],Ys,Zs) :- | Zs=Ys.
    merge(Xs,[ ],Zs) :- | Zs=Xs.
    one-by-one([A|W],Z) :- | Z=[A|Z1], next-one(W,Z1).
    next-one([B|W],Z) :- | Z=[B].
    two-at-once([A,B|W],Z) :- | Z=[A|Z1], Z1=[B].
```

where "*double*" is a predicate to duplicate its input element (either 0 or 1) and make a list consisting of the two elements, and "*merge*" is the predicate as before. The only difference between "*p*1" and "*p*2" is that "*one-by-one*" invoked from "*p*1" pulls out first two elements from the merged stream one by one as soon as each element appears in the merged stream, while "*two-at-once*" invoked from "*p*2" pulls out first two elements from the merged stream only when two elements appear in the merged stream. Then, it is easy to see that the execution of goals "?- $p1(t_1, t_2, t_3)$" and "?- $p2(t_1, t_2, t_3)$" returns the same answer substitutions for any terms t_1, t_2, t_3. For example, the execution of "?- $p1(0,1,Z)$" and "?- $p2(0,1,Z)$" both return

$<Z \Leftarrow [0,0]>, <Z \Leftarrow [0,1]>, <Z \Leftarrow [1,0]>, <Z \Leftarrow [1,1]>.$

That is, their input/output behaviors are identical for any instances of $p1(X,Y,W)$ and $p2(X,Y,W)$. Now, let "*complement*" be the predicate defined by the following GHC program:

```
    complement([0|W],Y) :- | Y=1.
    complement([1|W],Y) :- | Y=0.
```

and consider two goals

```
    ?- p1(0,Y,Z), complement(Z,Y),
    ?- p2(0,Y,Z), complement(Z,Y).
```

Then, the first goal can return answer substitution

$<Y \Leftarrow 1, Z \Leftarrow [0,0]>, <Y \Leftarrow 1, Z \Leftarrow [0,1]>,$

while the second goal can return only one answer substitution

$<Y \Leftarrow 1, Z \Leftarrow [0,0]>,$

since the second element of the output "Z" of $p2$ is already decided to be 0 when its first element 0 is passed to "*complement*" (hence when the output of "*complement*" is passed to the input "Y" of "$p2$"). That is, "?- $p1(0,Y,Z), complement(Z,Y)$" and "?- $p2(0,Y,Z), complement(Z,Y)$" show different input/output behaviors, although "$p1$" and "$p2$" show an identical input/output behavior. (This is the famous anomaly by Brock and Ackerman [4].)

(5) The commitment operator of GHC plays two distinct roles.

One role is that it introduces some asymmetricity between the atoms in a guard and those in a body. It separates the guard from the body with the restriction that the guard be executed in a special manner. Intuitively, the execution of guards must suceed (i) regardless of the (arguments form) information possibly supplied in future through callers (ii) without utilizing the (arguments form) information possibly supplied from the execution of bodies.

Another role is that it prunes off the search space. Once the execution is committed to some clause, other clauses are discarded and not reconsidered even if using some other clause may lead to successful computation. This commitment operation makes the burden of managing (otherwise multiple) variables binding environments much lighter.

It has been sometimes pointed out that search incompleteness due to the commitment operation of GHC is the most important difference that separates GHC from Prolog, hence makes GHC extra-logical, because the lack of backtracking in GHC possibly misses some successful computation so that some goal expected to succeed may possibly fail, be suspended or loop indefinitely. But as far as the semantics of Prolog (at some abstraction level) is concerned, the backtracking mechanism can be ignored by assuming appropriate nondeterminism, and in fact it was ignored in van Emden and Kowalski's paper [6]. (The backtracking mechanism was considered a problem of implementation from the viewpoint of semantics at some abstraction level.) Moreover, even with the backtracking mechanism, search completeness is not guaranteed in general. Hence, it would be permissible at first to assume appropriate nondeterminism and ignore the second role of the commitment operation when some abstract semantics of GHC programs is aimed at following the spirit of van Emden and Kowalski's approach.

(6) Nondeterministic execution characterizes some semantic aspects of GHC programs.

When one tries to formalize the semantics of any parallel programs, he/she naturally wonders whether the semantics should have any characteristics that is inherent in the truly parallel execution, and not captured by the nondeterministic sequential execution. Here, by *truly parallel execution of GHC programs*, we mean that the execution proceeds in such a way that the execution segment of one goal may be conducted independently before the goal receives the information of the shared variables instantiation caused by the execution of other goals. By *nondeterministic sequential execution of GHC programs*, we mean that the execution proceeds sequentially by interleaving the execution segments of each goal in such a way that, before the execution segment of one goal is conducted, the goal always receives the information of the shared variables instantiation caused by the execution of other goals. The necessity of considering the truly parallel execution of GHC programs depends on the following two points:

(a) whether the different assumptions on the propagation of the shared variables instantiation in the truly parallel execution and in the nondeterministic sequential execution have any particular meaning to the intended semantics of GHC programs, and
(b) whether the independent execution of each goal in the truly parallel execution and the interleaved execution of each goal in the nondeterministic sequential execution have any particular meaning to the intended semantics of GHC programs.

As for the first point, if the intended semantics is concerned with only whether *some* execution of a goal *can* succeed, fail or be suspended, then there is no particular effect which depends on whether the execution is done in truly parallel. It seems obvious that some nondeterministic sequential execution can succeed whenever some truly parallel execution succeeds, because, if the truly parallel execution is committed to a clause, hence its guard is solved, then the guard is solved regardless of the further instantiation of the variables occurring in the caller, hence the nondeterministic sequential execution can be committed to the same clause. It also seems obvious that some nondeterministic sequential execution can fail whenever some truly parallel execution fails. One might be afraid that a wrong commitment is prevented in the nondeterministic sequential execution, since the information of the shared variables instantiation is propagated immediately before the next sequential execution step so that the guard for the wrong commitment is not solved. This is not true due to the same reason. If the truly parallel execution is committed to a clause, hence its guard is solved, then the guard cannot rule out further instantiation of the variables occurring in the caller, hence the nondeterministic sequential execution can be committed to the same clause to end in failure.

As for the second point, again if the intended semantics is concerned with only whether *some* execution of a goal *can* succeed, fail or be suspended, any assumption about nondeterministic execution, e.g., fairness, does not need to be considered.

Example 1.5 Let *a-or-b* and *b-or-a* be the predicates defined by the following GHC program:

```
    a-or-b(X) :- | X=a.
    a-or-b(X) :- | X=b.
    b-or-a(X) :- | X=b.
    b-or-a(X) :- | X=a.
```

Consider the goal

```
    ?- a-or-b(X), b-or-a(X).
```

One might expect that this goal can fail when the truly parallel execution is employed, e.g., the execution of the first goal *a-or-b*(X) is committed to the first clause and that of the second goal *b-or-a*(X) is committed to the third clause in parallel, and it never fails whatever nondeterministic sequential execution may be employed. The latter expectation is, however, wrong. The following sequential execution leads to failure as well.

1. The execution of *a-or-b*(X) is committed to the first clause.
2. X and a is unified.
3. The execution of *b-or-a*(X) is committed to the third clause.
4. The unification of a and b fails.

These problems call for reflection even about our understanding of the semantics of Prolog programs. The semantics of Prolog programs by van Emden and Kowalski [6] is absolutely useful abstraction to consider various semantic properties of Prolog programs without worrying too detailed operational behaviors of actual Prolog interpreters. However, which aspects of Prolog computation are indeed reflected in van Emden-Kowalski style semantics, and which aspects are ignored (or abstracted)? And, which aspects of GHC computation should be reflected in the semantics of GHC programs, and which aspects should be ignored (or abstracted), if the spirit of van Emden and Kowalski's approach is followed? So far, we have considered differences of answer substitutions, since they are superficially observable behavior of GHC programs. However, if the least Herbrand model semantics of Prolog programs is re-examined, it is noticed that it does not necessarily characterize differences of answer substitutions. Moreover, even if the least Herbrand models of programs P_1 and P_2 are identical and Q defines predicates not in P_1 or P_2, the least Herbrand models of programs $P_1 \cup Q$ and $P_2 \cup Q$ might be different when Q contains constant or function symbol not in P_1 or P_2.

Example 1.6 Let P_1 and P_2 be the following two Prolog programs:

```
    P1 : p(a).
         p0(a).
    P2 : p(X).
         p0(a).
```

The predicate "$p0$" is just used for introducing constant "a." Then, because the Herbrand universes of P_1 and P_2 are $\{a\}$, these two Prolog programs are equivalent in the sense of the least Herbrand model semantics. However, these two programs respond in different manners to a query

```
    ?- p(X).
```

P_2 returns the empty susbtitution $<>$ as its answer substitution, while P_1 returns $< X \Leftarrow a >$. Moreover, these programs may not be replaced with each other when the program to be combined has a different Herbrand universe. For example, let Q be the following Prolog program:

Q : q(b).

Then, the least Herbrand model of $P_1 \cup Q$ is

$\{p(a), p0(a), q(b)\}$,

while that of $P_2 \cup Q$ is

$\{p(a), p(b), p0(a), q(b)\}$.

This paper presents a preliminary formalization of the semantics of Guarded Horn Clauses (GHC) particularly emphasizing on a compositional character. First, Section 3 describes the execution of GHC programs using some tree structures, in which not only *actually* successful computation but also *potentially* successful computation are considered. Next, Section 4 studies the semantics of a class of GHC programs, called monotonic GHC programs, by simply abstracting the description. The semantics considers the set of pairs consisting of initial goals and their answers, and it is shown that the semantics enjoys a compositional character. Then, after examining the problems of non-monotonic GHC programs, Section 5 investigates the semantics of general GHC programs. The semantics considers the partially ordered multiset of such pairs, and it is shown that the semantics still enjoys the compostional character. Last, Section 6 compares the semantics of GHC programs with that of Prolog programs. Appendix shows a reformulation of Kahn's results about functional data flow networks within the framework of parallel logic programming for finite computation. (In the following, paragraphs begining with "*Remark*" are supplementary explanations so that the readers may skip them at first reading.)

2. Guarded Horn Clauses (GHC)

This section introduces the parallel programming language GHC following the explanation in Ueda [41]. (See [40] for thorough discussion.) In the following, symbols beginning with uppercase letters are used for variables, and ones beginning with lower case letters for constant, function and predicate symbols, following the syntactic convention of DECsystem10 Prolog [3].

(1) GHC Program

A *GHC program* is a finite set of expressions, called simply *clauses*, of the following form:

$$\mathrm{H} \text{ :- } \mathrm{G}_1, \mathrm{G}_2, \ldots, \mathrm{G}_m \mid \mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_n. \qquad (m, n \geq 0)$$

where H, G_i's and B_j's are atoms. H is called a *clause head*, the G_i's are called *guard goals*, and the B_j's are called *body goals*. The symbol "|" is called a *commitment operator*. (When $m = n = 0$, ":-" and "|" are ommitted.) The part of a clause before "|" is called a *guard*, and the part after "|" is called a *body*. Note that the clause head is included in the guard.

One binary predicate "=" is predefined by the language. The predicate "=" is used for unifying two terms.

Example 2.1 A predicate "*join*" for merging two ordered streams into an ordered stream is defined in GHC as follows:

C_1: join([A|Xs],[B|Ys],Zs) :- A≤B | Zs=[A|Zs1], join(Xs,[B|Ys],Zs1).
C_2: join([A|Xs],[B|Ys],Zs) :- A>B | Zs=[B|Zs1], join([A|Xs],Ys,Zs1).
C_3: join([ ],Ys,Zs) :- | Zs=Ys.
C_4: join(Xs,[ ],Zs) :- | Zs=Xs.

where ≤ and > are defined as follows:

C_5: 0≤Y.
C_6: suc(X)≤suc(Y) :- | X≤Y.
C_7: suc(X)>0.

C_8: suc(X)>suc(Y) :- | X>Y.

(2) GHC Goal

A *GHC goal* is an expression of the following form:

?- A_1, A_2, ..., A_k. $(k \geq 0)$.

A goal is called an empty goal when k is equal to 0.

Example 2.2 The following are GHC goals.

?- join([1],[ ],X), join(X,[2],Z).

?- join(X,[2],Z).

where "1" and "2" are abbreviations of *suc*(0) and *suc*(*suc*(0)), respectively.

(3) Execution

The execution of a GHC goal with respect to a given GHC program tries to solve the goal, i.e., reduce the goal to the empty goal, using the clauses in the GHC program in the same way as Prolog but possibly a fully parallel manner provided that the following "rules of suspension" and "rule of commitment" are observed.

Rules of Suspension

(a) Unification invoked directly or indirectly in the guard of a clause C called by a goal G (i.e., unification of G with the head of C and any unification invoked by solving the guard goals of C) cannot instantiate the goal G.
(b) Unification invoked directly or indirectly in the body of a clause C called by a goal G cannot instantiate the guard of C or G until C is selected for commitment (see below).

A piece of unification that can succeed only by causing such instantiation is suspended until it can succeed without causing such instantiation.

Rule of Commitment

When some clause C called by a goal G succeeds in solving its guard, that clause C tries to be selected for subsequent computation of G. To be selected, C must first confirm that no other clause in the program have been selected for G. If confirmed, C is selected indivisibly, and the execution of G is said to be committed to the clause C.

Example 2.3 The execution of GHC goal

?- join([1],[ ],X), join(X,[2],Z).

never fails whichever clauses it may be committed to, and will return $<X \Leftarrow [1], Z \Leftarrow [1,2]>$ as its answer substitution. The execution of goal

?- join(X,[2],Z).

is suspended, because no guard is solved for the goal.

3. A Preliminary Operational Semantics of GHC Programs

This section first introduces the notion of computation forest, then introduces a partial ordering relation between computation forests.

The following sections assume familiarity with the basic terminology of first order logic, such as term, atom (atomic formula), formula, substitution, most general unifier (m.g.u.) and so on. Syntactic variables are X, Y, Z for variables; s, t, for terms; A, B for atoms; C for clauses, possibly with primes and subscripts.

A *goal* is a multiset of atoms. A goal consisting of only one atom is called a *singleton goal*. Goals are denoted by Γ, Δ, Π, and the empty goal is denoted by $\Box$. Two goals are considered identical when they are identical modulo renaming of variables.

A *substitution* is defined as usual, and denoted by

$$<X_1 \Leftarrow t_1, X_2 \Leftarrow t_2, \ldots, X_l \Leftarrow t_l>.$$

A substitution is called a *renaming substitution* when it assigns a distinct variable to each variable, and called a *non-renaming substitution* when it is not a renaming substitution. Substitutions are denoted by $\sigma, \tau, \theta, \eta$, and the empty substitution is denoted by $<>$.

3.1 Computation Forest

The notion of computation forest is obtained by modelling the nondeterministic sequential execution of GHC just as the notion of proof tree in Prolog.

Definition Computation Forest

A *computation forest* is a multiset of trees satisfying the following conditions:

(a) Each node is labelled with a trio of the form (A, R, C), where A is either an atom or an equation, R is either the empty set { } or a singleton set of atoms, and C is either a clause of P or a special unit clause

C_0: X=X.

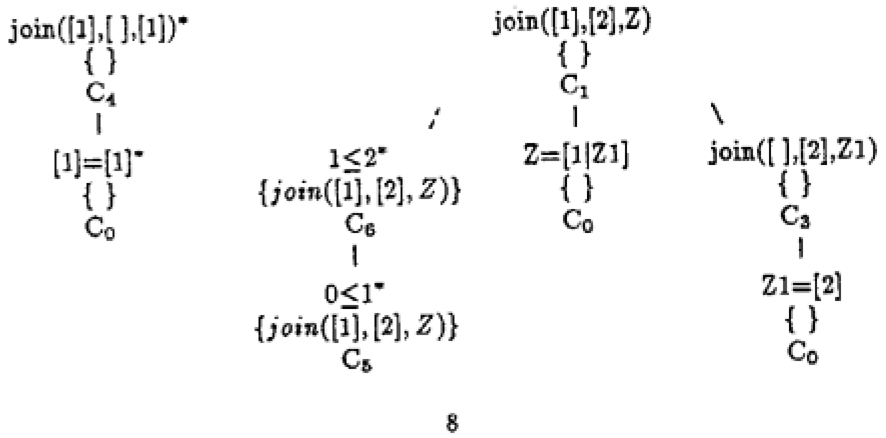
not in program P. (A is called the *atom part*, R is called the *instantiation restriction*, and C is called the *clause part* of the label.)

(b) Each node is classified into either *guard node* or *body node*.

(c) Each node is either marked "solved" or unmarked.

Computation forests are denoted by $\mathcal{F}, \mathcal{G}, \mathcal{H}$, possibly with primes and subscripts. Two computation forests are considered identical when they are identical modulo renaming of variables occurring in the labels. Each tree in a computation forest is called the *component computation tree* of the computation forest. The computation forest obtained by juxtaposing two computation forests $\mathcal{F}$ and $\mathcal{G}$ is denoted by $\mathcal{F} \sqcup \mathcal{G}$.

Example 3.1.1 $\mathcal{F}_6$ below is a computation forest. (Due to space limit, label (A, R, C) are shown in three consecutive rows.)



The dotted line denotes an edge to a guard node, and the solid lines denote edges to body nodes. The superscript "*" denotes the "solved" mark.

Definition Initial Computation Forest

A computation forest $\mathcal{F}$ is an *initial computation forest* of goal $\Gamma = \{A_1, A_2, \ldots, A_k\}$, when $\mathcal{F}$ consists of just k root unmarked body nodes labelled with $(A_1, \{\ \}, C_{i_1})$, $(A_2, \{\ \}, C_{i_2})$, ... , $(A_k, \{\ \}, C_{i_k})$, where each C_{i_l} is any clause in P defining the predicate of A_l for $l = 1, 2, \ldots, k$. These body nodes have no sibling guard nodes. (See the next explanation of *immediate extension*.)

Example 3.1.2 $\mathcal{F}_0$ below is an initial computation forest of $\{join([1],[\ ],X), join(X,[2],Z)\}$.

```
join([1],[ ],X)                  join(X,[2],Z)
     { }                              { }
     C4                               C1
```

Definition Immediate Extension of Computation Forest

A computation forest $\mathcal{G}$ is an *immediate extension of* $\mathcal{F}$ when $\mathcal{G}$ is obtained from $\mathcal{F}$ by the following operation: Let v be an unmarked terminal node of $\mathcal{F}$ which is

(a) either a guard node

(b) or a body node whose all sibling guard nodes are marked "solved."

Let (A, R, C) be the label of v.

Case 1 : When A is not an equation, the terminal node v is said to be *GHC resolvable* if the GHC clause C is of the form

$$H \text{ :- } G_1 , G_2 , \ldots , G_m \mid B_1 , B_2 , \ldots , B_n \qquad (m, n \geq 0)$$

such that A is an instance of H, say by substitution η. Then

(1a) add m unmarked child guard nodes of v labelled with $(G_1\eta, H\eta, C_{i_1}), (G_2\eta, H\eta, C_{i_2})$, $\ldots, (G_m\eta, H\eta, C_{i_m})$ to v, where each C_{i_l} is any clause in P defining the predicate of G_l for $l = 1, 2, \ldots, m$. (These nodes are called *sibling guard nodes* of other child body nodes defined in (1b).)

(1b) add n unmarked child body nodes labelled with $(B_1\eta, R, C_{j_1}), (B_2\eta, R, C_{j_2}), \ldots, (B_n\eta, R, C_{j_n})$ to v, where each C_{j_l} is any clause in P defining the predicate of B_l for $l = 1, 2, \ldots, n$.

(1c) when $m = n = 0$, mark the just GHC resolved node "solved," and repeat marking any unmarked node "solved" when all its child nodes are marked "solved."

In this case, $\mathcal{G}$ is called an immediate extension of $\mathcal{F}$ *with substitution* $<>$.

Case 2 : When A is an equation $s = t$ (and C is "$X = X$"), the terminal node v is said to be *GHC resolvable* if s and t are unifiable, say by m.g.u. θ, and θ does not instantiate the variables occurring in R, i.e., $R\theta = R$. Then

(2a) modify the label (A', R', C') of each node to $(A'\theta, R'\theta, C')$, and

(2b) mark the just GHC resolved node "solved," and repeat marking any unmarked node "solved" when all its child nodes are marked "solved."

In this case, $\mathcal{G}$ is called an immediate extension of $\mathcal{F}$ *with substitution* θ.

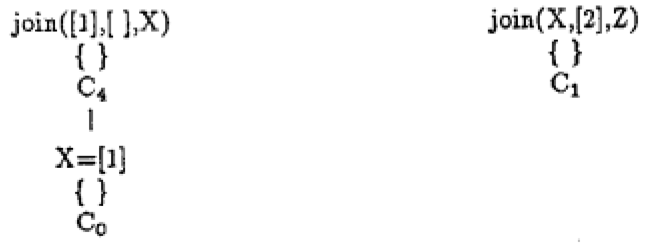
One will immediately understand why $H\eta$ is the instantiation restriction of each child guard node. One might, however, wonder why R is the instantiation restriction of each child body node. The reason is as follows: Suppose that a node is GHC resolved to generate a child guard node labelled with $(G_i\eta, H\eta, C)$, and the child guard node is to be GHC resolved. Then, some of the variables occurring in $G_i\eta$, which occur only in the guards

$G_1\eta, G_2\eta, \ldots, G_m\eta$, may be instantiated when $G_i\eta$ is solved, while those in $H\eta$ may not be instantiated, since the goals invoked from $G_i\eta$ are invoked from $H\eta$ indirectly.

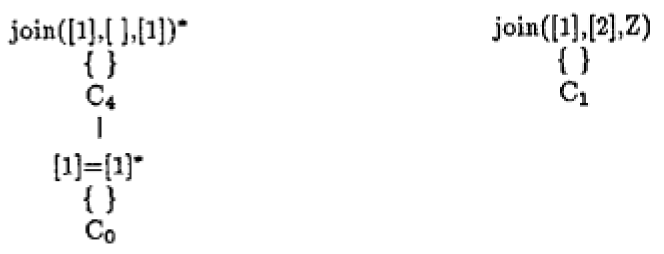
Definition Extension of Computation Forest

A computation forest $\mathcal{G}$ is an *extension of* a computation forest $\mathcal{F}$ with substitution σ', when $\mathcal{G}$ is obtained from $\mathcal{F}$ through (possibly no) successive application of immediate extensions with substitutions $\theta_1, \theta_2, \ldots, \theta_l$ and σ' is the composition $\theta_1\theta_2\cdots\theta_l$.

Example 3.1.3 Computation forest $\mathcal{F}_1$ below is an immediate extension of $\mathcal{F}_0$ in Example 3.1.2.



Computation forest $\mathcal{F}_2$ below is an immediate extension of $\mathcal{F}_1$ above.



Both $\mathcal{F}_1$ and $\mathcal{F}_2$ are extensions of $\mathcal{F}_0$

Definition Committed Forest and Committed Extension

A computation forest $\mathcal{F}$ is called a *committed forest* when any node whose instantiation restriction R is not the empty set are marked "solved."

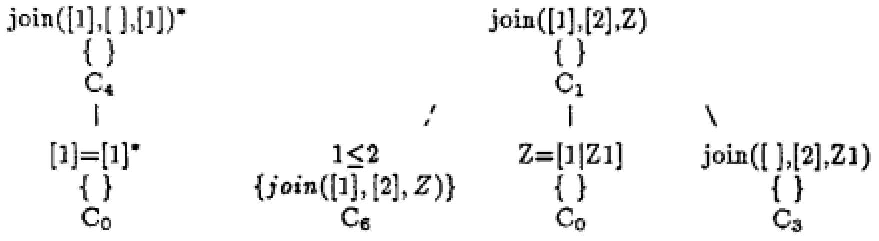
A committed forest $\mathcal{F}$ is called a *committed extension of* goal Γ *with answer substitution* σ, when $\mathcal{F}$ is an extension of an initial computation forest of Γ with substitution σ', and σ is the restriction of σ' to the variables occurring in Γ.

Definition Maximal Committed Forest and Maximal Committed Extension

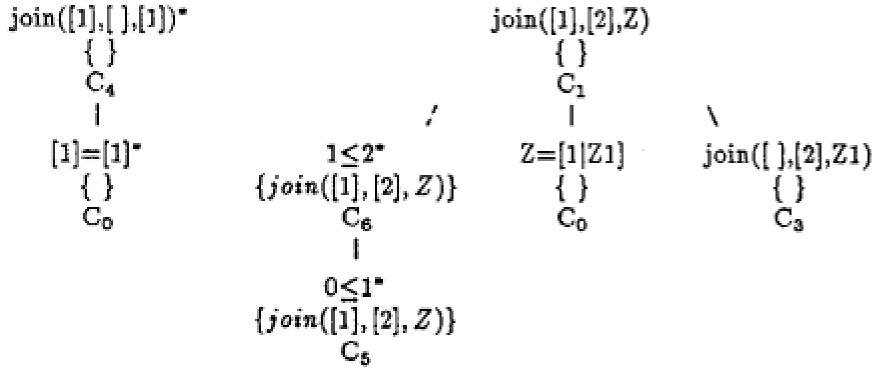
A committed forest $\mathcal{F}$ is called a *maximal committed forest*, when there exists no committed forest (except itself) which is an extension of $\mathcal{F}$.

A maximal committed forest $\mathcal{F}$ is called a *maximal committed extension of* goal Γ *with answer substitution* σ, when $\mathcal{F}$ is a committed extension of Γ with answer substitution σ.

Example 3.1.4 Computation forest $\mathcal{F}_1$ is a committed extension of goal $\Gamma = \{join([1],[\,],X), join(X,[2],Z)\}$. Computation forest $\mathcal{F}_2$ is also a committed extension of Γ. Although computation forest $\mathcal{F}_3$ below is an immediate extension of $\mathcal{F}_2$, it is not a committed extension of Γ, since the node labelled with $(1 \leq 2, \{join([1],[2],Z)\}, C_6)$ is not marked "solved."



However, computation forest $\mathcal{F}_5$ below is a committed extension of Γ.



Remark. Our operational semantics is different from the original execution mechanism by Ueda [41] in several respects.

(a) Our operational semantics just specifies possible construction of computation forests. The original execution mechanism by Ueda discards alternative choice of clauses once all sibling guard nodes are marked "solved."

(b) Our operational semantics inhibits GHC resolution at body nodes untill all sibling guard nodes are marked "solved." The original execution mechanism by Ueda permits it as far as it does not instantiate the variables occurring in the guard atoms, although no actual implementation employs this mechanism.

(c) Our operational semantics assumes that the application of m.g.u. θ to the label of each node in a computation forest (the operation at step (2a)) can be done as an indivisible operation. The original execution mechanism by Ueda does not assume such indivisibility (cf. Ueda [39]). Such indivisibility is problematic, in particular when a distributed implementation, which may assign these nodes to different processor elements, is employed. It is a well-known pragmatics in the circle of GHC programming that the programming styles assuming such indivisibility should be avoided as much as possible, e.g., the use of meta-predicate $var(X)$ for shared variable X. However, this assumption does not affect the semantics to be explained in the following.

(d) The clause used at GHC resolution is fixed as the third element of each label before the node is actually GHC resolved. This is just for simplicity of explanation, and does not affect the semantics to be explained in the following.

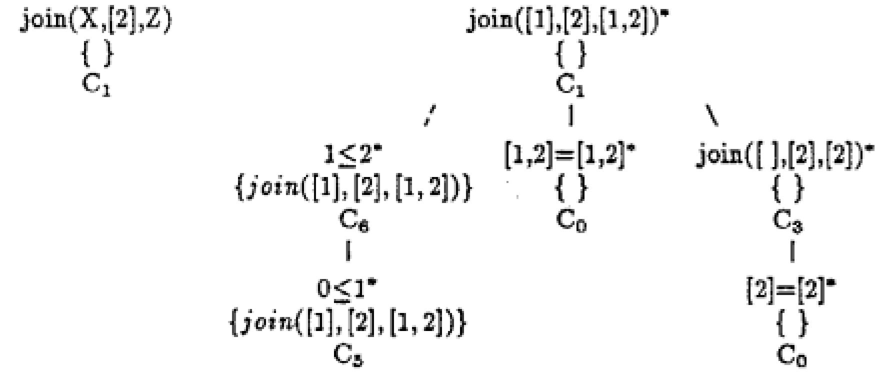
3.2 Success Forest and Suspension Forest

The definition of computation forest naturally introduces a partial ordering relation between committed forests. (Intuitively, this ordering means that computation forest $\mathcal{F}$ can be extended to $\mathcal{G}$ when additional instantiation is applied to all the node labels of $\mathcal{F}$.)

Definition Extension Ordering between Committed Forests

Let $\mathcal{F}$ be a committed extension of goal Γ with answer substitution σ and $\mathcal{G}$ be a committed extension of goal $\Gamma\theta$ with answer substitution τ. Then, $\mathcal{F}$ is said to be *extensible* to $\mathcal{G}$ and denoted by $\mathcal{F} \preceq \mathcal{G}$ when there exists a substitution η for $\Gamma\sigma$ such that $\mathcal{G}$ is an extension of $\mathcal{F}\eta$, where $\mathcal{F}\eta$ is a computation forest obtained by applying η to all the node labels (except the clause parts) of $\mathcal{F}$.

Example 3.2.1 Consider the GHC program of Example 2.1 defining "*join*," "$\leq$" and "$>$." Let $\mathcal{F}$ and $\mathcal{G}$ be two computation forests (trees) below:



Then $\mathcal{F}$ is extensible to $\mathcal{G}$.

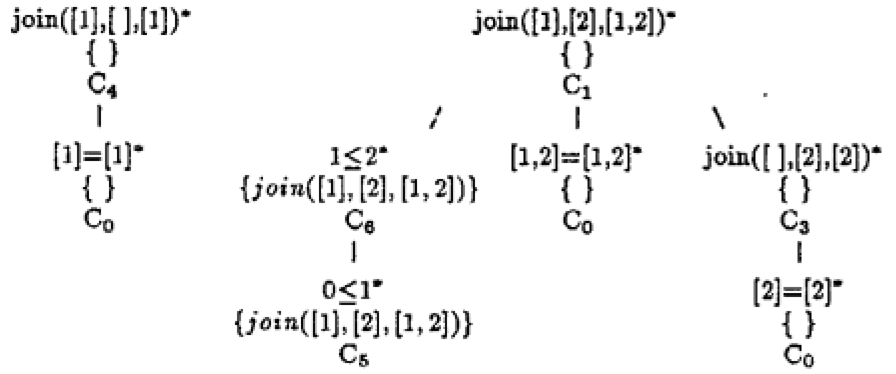
Definition Maximal Committed Subextension

Let Γ and Δ be goals, $\mathcal{F} \cup \mathcal{G}$ be a maximal committed extension of goal $\Gamma\theta \cup \Delta\theta$, where the root nodes of $\mathcal{F}$ and $\mathcal{G}$ are originated from the root nodes with atom parts in $\Gamma\theta$ and $\Delta\theta$, respectively. A committed extension $\mathcal{H}$ of goal Γ is called the *maximal committed subextension of* Γ *in* $\mathcal{F} \cup \mathcal{G}$ when

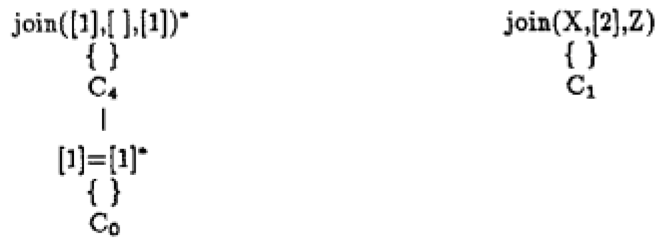
(a) $\mathcal{H}$ is extensible to $\mathcal{F}$, and

(b) there is no committed extension $\mathcal{H}'$ of goal Γ (except $\mathcal{H}$) such that $\mathcal{H} \preceq \mathcal{H}' \preceq \mathcal{F}$.

Example 3.2.2 Consider the GHC program of Example 2.1 defining "*join*," "$\leq$" and "$>$." Let $\mathcal{F} \cup \mathcal{G}$ be a maximal committed forest of $\{join([1],[\,],X), join(X,[2],Z)\}$ below:



Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be two committed forests (trees) of $\{join([1],[\,],X)\}$ and $\{join(X,[2],Z)\}$ below:



Then, $\mathcal{H}_1$ and $\mathcal{H}_2$ above are maximal committed subextensions of $\{join([1],[\,],X)\}$ and $\{join(X,[2],Z)\}$ in $\mathcal{F} \cup \mathcal{G}$.

In the discussion in Section 4 and 5, we will focus our attention on the following class of computation forests.

Definition Success Forest, Suspension Forest and Success-Suspension Forest

A committed extension $\mathcal{F}$ of goal Γ with answer substitution σ is called a *success forest* of Γ with answer substitution σ, when all nodes of $\mathcal{F}$ are marked "solved."

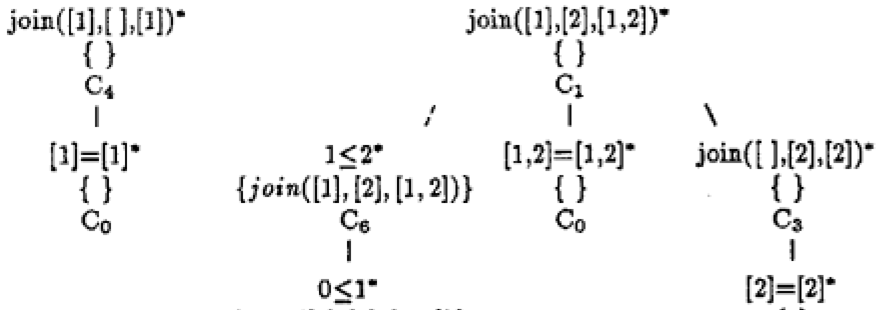
A committed extension $\mathcal{F}$ of goal Γ with answer substitution σ is called a *suspension forest* of Γ with answer substitution σ, when

(a) $\mathcal{F}$ is not yet a success forest,

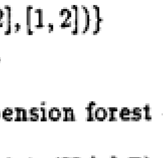
(b) $\mathcal{F}$ is a maximal committed subextension of $\mathcal{G}$ for some success forest $\mathcal{G}$ of $\Gamma\theta$.

A committed extension $\mathcal{F}$ of goal Γ with answer substitution σ is called a *success-suspension forest* of Γ with answer substitution σ, when it is either a success forest or a suspension forest of Γ with answer substitution σ. The multiset of all the atom parts of $\mathcal{F}$'s unmarked terminal nodes is called the *unsolved goal of $\mathcal{F}$*.

Example 3.2.3 $\mathcal{F}_8$ below is a success forest of $\{join([1],[\,],X),\ join(X,[2],Z)\}$.



The computation forest below is a suspension forest of $\{join(X,[2],Z)\}$.



Note that, for any success-suspension forest $\mathcal{F} \cup \mathcal{G}$ of goal $\Gamma\theta \cup \Delta\theta$, there exists a unique maximal committed subextension of Γ in $\mathcal{F} \cup \mathcal{G}$, and it is either a success forest or a suspension forest. The following lemma is to be used extensively in the following inductive proofs.

Lemma 3.2 Let $\mathcal{F}$ be a maximal committed extension of goal $\{A_1, A_2, \ldots, A_k\}$ with answer substitution σ. If σ is not a renaming substitution, then there exists an atom A_l $(1 \leq l \leq k)$ such that the answer substitution of the maximal committed subextension of $\{A_l\}$ in $\mathcal{F}$ is not a renaming substitution.

Proof. If there is no such A_l, then let $\mathcal{G}$ be the computation forest consisting of maximal committed subextensions of $\{A_1\}, \{A_2\}, \ldots, \{A_k\}$ in $\mathcal{F}$. Then, $\mathcal{G}$ is a maximal committed extension of $\{A_1, A_2, \ldots, A_k\}$ whose answer substitution is a renaming substitution and which is extensible to $\mathcal{F}$. This contradicts the assumption that $\mathcal{G}$ is a maximal committed extension of $\{A_1, A_2, \ldots, A_k\}$.

Remark. One might have thought that it is a little unnatural that the substitution θ is applied to the label of each node at step (2a) in the definition of "immediate extension." Indeed, it seems more natural to modify the operation at step (2a) as follows:

(2a) modify the label (A', R', C') of each *unmarked terminal* node (except the just GHC resolve node) to $(A'\theta, R'\theta, C')$,

because not only it is useless to apply θ to all the nodes but also the modification above leaves the tracks of how GHC resolution is applied. However, according to this modification, the labels of corresponding nodes might be different for two computation forests that are *substantially* identical but different only in the order of GHC resolutions applied, which brings a little complication so that we have employed the definition before.

4. Semantics of Monotonic GHC Programs

This section first introduces a class of GHC programs, called monotonic GHC programs, for which the operational semantics described in the previous section can be very simply abstracted. Then, it is shown that the denotations of composed goals are computed from those of composing atoms, and two equivalent subgoals can be replaced with each other w.r.t. this semantics.

4.1 Monotonic GHC Programs

(1) Monotonicity of GHC Programs

Definition Monotonicity of Goals, Atoms and Predicates

Let P be a GHC program. A goal Γ is said to be *monotonic in* P when, for any instance Π of Γ such that

(a) there exists a success-suspension forest $\mathcal{F}$ of Π, say with answer substitution σ, and

(b) there exists a success-suspension forest $\mathcal{G}$ of $\Pi\sigma$, say with answer substitution τ,

there exists a success-suspension forest $\mathcal{G}'$ such that

(a) $\mathcal{G}'$ is a success-suspension forest of Π with answer substitution $\sigma\tau$, and

(b) the unsolved goal of $\mathcal{G}'$ is identical to that of $\mathcal{G}$.

An atom "A" is said to be *monotonic in* P when singleton goal $\{A\}$ is monotonic in P. An n-ary predicate "p" is said to be *monotonic in* P when atom $p(X_1, X_2, \ldots, X_n)$ is monotonic in P, where $X_1, X_2, \ldots, X_n$ are distinct variables.

Definition Monotonicity of GHC Programs

A GHC program P is said to be *monotonic* when any predicate is monotonic in P.

Example 4.1.1 Let P_{join} be the GHC program of Example 2.1 defining "*join*," "$\leq$" and "$>$." Then, P_{join} is monotonic. For example, "*join*" is monotonic. The reason is as follows:

Suppose that there exists a success-suspension forest $\mathcal{F}$ of $\{join(t_1, t_2, t_3)\}$ with answer substitution σ and a success-suspension forest $\mathcal{G}$ of $\{join(t_1, t_2, t_3)\sigma\}$ with answer substitution τ. Then,

(a) if the clause part of a non-terminal node of $\mathcal{F}$ is C_1 or C_2, the same clause is the clause part of the corresponding non-terminal node of $\mathcal{G}$, and

(b) if the clause part of a non-terminal node of $\mathcal{F}$ is C_3 or C_4,

  (b1) either the same clause is the clause part of the corresponding non-terminal node of $\mathcal{G}$,

  (b2) or the opposite clause is the clause part of the corresponding non-terminal node of $\mathcal{G}$ when both the first and the second arguments of the atom parts are [ ] in $\mathcal{G}$.

Let $\mathcal{G}'$ be the success-suspension forest obtained from $\mathcal{G}$ by changing the clause part of the non-terminal nodes to conform the clause part of the corresponding non-terminal nodes of $\mathcal{F}$. Then, $\mathcal{G}'$ is a success-suspension forest of $\{join(t_1, t_2, t_3)\}$ with answer substitution $\sigma\tau$, and the unsolved goal of $\mathcal{G}'$ is identical to that of $\mathcal{G}$.

Example 4.1.2 Let P_{echo} be the following GHC program:

```
C1: shout-wait(X,Y) :- | X=0, wait0(Y).
C2: shout-wait(X,Y) :- | X=1, wait1(Y).
C3: echo-back(0,Y) :- | Y=0.
C4: echo-back(1,Y) :- | Y=1.
C5: wait0(0).
C6: wait1(1).
```

Then, P_{echo} is monotonic. For example, "*shout-wait*" is monotonic. The reason is as follows: Suppose that there exist a success-suspension forest $\mathcal{F}$ of $\{shout\text{-}wait(s, t)\}$ with answer substitution σ, and a success-suspension forest $\mathcal{G}$ of $\{shout\text{-}wait(s, t)\sigma\}$ with answer substitution τ. Then,

(a) s is a variable or 0, $s\sigma$ is 0, τ is <>, and

  (a1) when t is a variable, $t\sigma$ is a variable, or

  (a2) when t is 0, $t\sigma$ is 0, and

(b) s is a variable or 1, $s\sigma$ is 1, τ is <>, and

  (b1) when t is a variable, $t\sigma$ is a variable, or

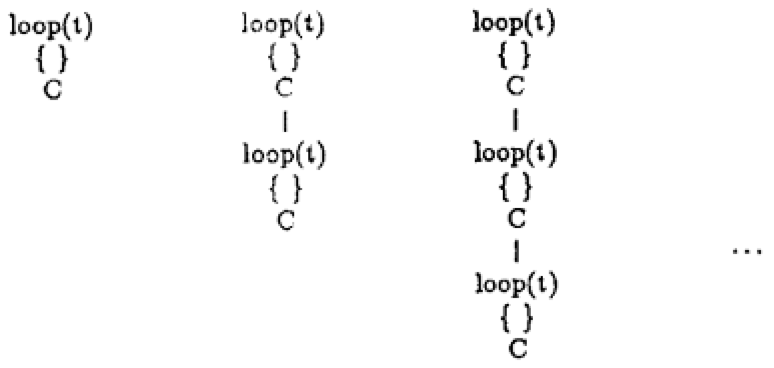
  (b2) when t is 1, $t\sigma$ is 1.

Hence, for any success-suspension forests $\mathcal{F}, \mathcal{G}$ in the definition of monotonicity of "*shout-wait*," the computation forest $\mathcal{G}$ itself is a computation forest of $\{shout\text{-}wait(s, t)\}$ with answer substitution σ.

Example 4.1.3 Let P_{loop} be the following GHC program:

```
C: loop(X) :- | loop(X).
```

Then, P_{loop} is monotonic. For example, "*loop*" is monotonic, because the initial computation forest of $\{loop(t)\}$ is extended to the committed extensions below, hence there is no success-suspension forest of $\{loop(t)\}$.



Example 4.1.4 Recall the GHC program P_{BA} of Example 1.4 as follows:

```
C01: p1(X,Y,Z) :- | double(X,XX),double(Y,YY),merge(XX,YY,W),one-by-one(W,Z).
C02: p2(X,Y,Z) :- | double(X,XX),double(Y,YY),merge(XX,YY,W),two-at-once(W,Z).
C03: double(0,AA) :- | AA=[0,0].
C04: double(1,AA) :- | AA=[1,1].
C05: merge([A|Xs],Ys,Zs) :- | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
C06: merge(Xs,[A|Ys],Zs) :- | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
C07: merge([ ],Ys,Zs) :- | Zs=Ys.
C08: merge(Xs,[ ],Zs) :- | Zs=Xs.
C09: one-by-one([A|W],Z) :- | Z=[A|Z1], next-one(W,Z1).
C10: next-one([B|W],Z) :- | Z=[B].
C11: two-at-once([A,B|W],Z) :- | Z=[A|Z1], Z1=[B].
C12: complement([0|W],Y) :- | Y=1.
C13: complement([1|W],Y) :- | Y=0.
```

Then, P_{BA} is monotonic. For example, "*merge*" is monotonic. The reason is as follows: Suppose that there exists a success-suspension forest $\mathcal{F}$ of $\{merge(t_1,t_2,t_3)\}$ with answer substitution σ and a success-suspension forest $\mathcal{G}$ of $\{merge(t_1,t_2,t_3)\sigma\}$ with answer substitution τ. Then,

(a) if the clause part of a non-terminal node of $\mathcal{F}$ is C_{05} or C_{06},
   - (a1) either the same clause is the clause part of the corresponding non-terminal node of $\mathcal{G}$,
   - (a2) or the opposite clause is the clause part of the corresponding non-terminal nodes of $\mathcal{G}$ when the head elements of the first and the second arguments of the atom parts are identical in $\mathcal{G}$, and

(b) if the clause part of a non-terminal node of $\mathcal{F}$ is C_{07} or C_{08},
   - (b1) either the same clause is the clause part of the corresponding non-terminal nodes of $\mathcal{G}$,
   - (b2) or the opposite clause is the clause part of the corresponding non-terminal nodes of $\mathcal{G}$ when both the first and the second arguments of the atom parts are [ ] in $\mathcal{G}$.

Let $\mathcal{G}'$ be the computation forest obtained from $\mathcal{G}$ by changing the clause part of non-terminal nodes to conform the clause part of the corresponding nodes of $\mathcal{F}$. Then, $\mathcal{G}'$ is a success-suspension forest of $\{merge(t_1,t_2,t_3)\}$ with answer substitution $\sigma\tau$, and the unsolved goal of $\mathcal{G}'$ is identical to that of $\mathcal{G}$.

Example 4.1.5 Let P_{non} be the GHC program as follows:

```
C1: non(X,Y) :- | X=0, wait(Y).
C2: non(X,1).
```

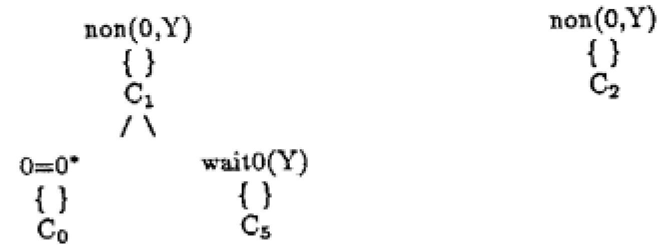
C_3: any(0,Y) :- | Y=0.
C_4: any(0,Y) :- | Y=1.
C_5: wait0(0).

This program is not monotonic, because, although

(a) computation forests $\mathcal{F}$ left below is a suspension forest of $\{non(X,Y)\}$ with answer substitution $<X \Leftarrow 0>$, and

(b) computation forest $\mathcal{G}$ right below is a suspension forest of $\{non(0,Y)\}$ with answer substitution $<>$,

there exists no success-suspension forest $\mathcal{G}'$ such that $\mathcal{G}'$ is a success-suspension forest of $\{non(X,Y)\}$ with answer substitution $<X \Leftarrow 0>$ and the unsolved goal of $\mathcal{G}'$ is identical to that of $\mathcal{G}$.



(2) Monotonicity for Composed Goals

The definition of monotonicity does not immediately guarantee that the monotonicity holds for composed goals.

Theorem 4.1 If a GHC program P is monotonic, then any goal Γ is monotonic in P.

Proof. For two substitutions σ and σ', we will define $\sigma' \ll \sigma$ when there exists a non-renaming substitution η such that $\eta\sigma' = \sigma$. This partial ordering $\ll$ is obviously well-founded. (The least substitution w.r.t. this partial ordering is a renaming substitution. For simplicity, $<>$ is used as a representative of renaming substitutions in the following proof, since the empty substitution $<>$ is a special renaming substitution.) The theorem is proved by induction on this well-founded ordering of answer substitutions.

Let $\mathcal{F}$ be a success-suspension forest of Π with answer substitution σ, and let $\mathcal{G}$ be a success-suspension forest of $\Pi\sigma$ with answer substitution τ.

Base Case: When σ is $<>$, computation forest $\mathcal{G}$ itself is a success-suspension forest of Π with answer substitution $\sigma\tau$.

Induction Step: When σ is not $<>$, from Lemma 3.2, there exists an atom A in Π such that the maximal committed subextension $\mathcal{F}_A$ of $\{A\}$ in $\mathcal{F}$ has a non-renaming answer substitution σ_A. Let σ' be the substitution such that $\sigma_A\sigma'$ is σ, and Π' be $\Pi\sigma_A$. Then, $\mathcal{F}$ is a success-suspension forest of Π' with answer substitution σ', and $\mathcal{G}$ is a success-suspension forest of $\Pi'\sigma'$ with answer substitution τ. From the induction hypothesis for σ', there exists a success-suspension forest $\mathcal{G}'$ of Π' with answer substitution $\sigma'\tau$ and the unsolved goal of $\mathcal{G}'$ is identical to that of $\mathcal{G}$.

Let $\mathcal{G}'_{A\sigma_A}$ be the maximal committed subextension of $\{A\sigma_A\}$ in $\mathcal{G}'$ with answer substitution τ_A. From the monotonicity of P, there exists a success-suspension forest $\mathcal{G}''_A$ such that $\mathcal{G}''_A$ is a success-suspension forest of $\{A\}$ with answer substitution $\sigma_A\tau_A$ and the unsolved goal of $\mathcal{G}''_A$ is identical to that of $\mathcal{G}'_{A\sigma_A}$.

Let $\mathcal{H}''$ be a success-suspension forest of Π obtained by

(a) first applying GHC resolution in the same way as $\mathcal{G}''_A$, and

(b) then applying GHC resolution in the same way as $\mathcal{G}'$.

Then $\mathcal{H}''$ is a success-suspension forest of Π with answer substitution $\sigma\tau$ and the unsolved goal of $\mathcal{H}''$ is identical to that of $\mathcal{G}$.

4.2 Success-Suspension Set

Now on in Section 4, let P be a monotonic GHC program.

(1) Success-Suspension Atom

Definition Success-Suspension Atom

An atom A is called a *success-suspension atom* (resp. *success atom, suspension atom*) *in* P when there exists a success-suspension forest (resp. *success forest, suspension forest*) of goal $\{A\}$ in P. The success-suspension forest is called the *associated forest* of A.

(2) Success-Suspension Set for Composed Goals

Definition Success-Suspension Pair for Goals

A pair $(\Gamma, \Gamma\sigma)$ is called a *success-suspension pair* (resp. *success pair, suspension pair*) *for goals of* P when there exist a success-suspension forest (resp. success forest, suspension forest) $\mathcal{F}$ of goal Γ with answer substitution σ in P. The success-suspension forest $\mathcal{F}$ is called the *associated forest of* $(\Gamma, \Gamma\sigma)$. Two success-suspension pairs are considered identical when they are identical modulo renaming of variables.

Definition Success-Suspension Set for Goals

The set of all the success-suspension pairs (resp. success pairs, suspension pairs) for goals of P is called the *success-suspension set* (resp. *success set, suspension set*) *for goals of* P, and denoted by $\overline{SS}(P)$ (resp. $\overline{success}(P)$, $\overline{suspension}(P)$).

The overlining is used to indicate that the quantities are concerned with the computation whose initial goals are *multisets of possibly many atoms*, while no overlining is to be used to indicate that the quantities are concerned with the computation whose initial goals are *singleton goals*. Note that two pairs corresponding to different computation forests with the same answer substitutions are considered identical elements in success-suspension sets for goals.

Example 4.2.1 Consider the GHC program P_{join} of Example 2.1 defining "*join*," "$\leq$" and "$>$." Then, $\overline{success}(P_{join})$ includes

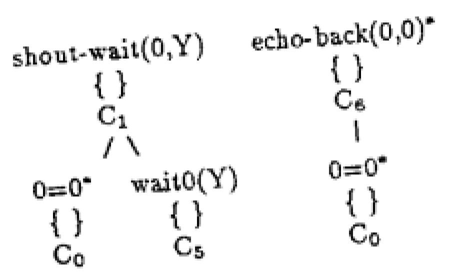
$(\{merge([1],[\,],X), merge(X,[2],Z)\}, \{merge([1],[\,],[1]), merge([1],[2],[1,2])\})$,

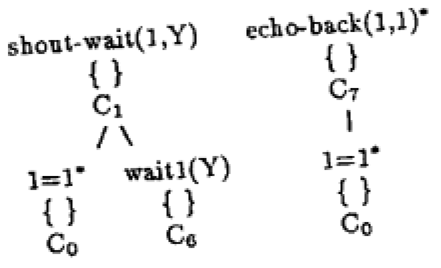
that is, $\{join([1],[\,],X), join(X,[2],Z)\}$ succeeds with answer substitution $< X \Leftarrow [1], Z \Leftarrow [1,2] >$. $\overline{suspension}(P_{join})$ includes

$(\{join([1],Y,X), join(X,[2],Z)\}, \{join([1],Y,X), join(X,[2],Z)\})$.

that is, $\{join([1],Y,X), join(X,[2],Z)\}$ is suspended with answer substitution $<>$.

Example 4.2.2 Consider the GHC program P_{echo} of Example 4.1.2 defining "*shout-wait*," "*echo-back*," "*wait0*" and "*wait1*." Then, an initial computation forest of $\{shout\text{-}wait(X,Y), echo\text{-}back(X,Z)\}$ is, for example, extended to the committed forests below:





The left computation forest can be extended to a success forest if variable "Y" is instantiated to constant "0", while the right computation forest can be extended to a success forest if variable "Y" is instantiated to constant "1". Hence, $\overline{suspension}(P_{echo})$ includes

$(\{shout\text{-}wait(X,Y), echo\text{-}back(X,Z)\}, \{shout\text{-}wait(0,Y), echo\text{-}back(0,0)\})$,

$(\{shout\text{-}wait(X,Y), echo\text{-}back(X,Z)\}, \{shout\text{-}wait(1,Y), echo\text{-}back(1,1)\})$,

that is, $\{shout\text{-}wait(X,Y), echo\text{-}back(X,Z)\}$ is suspended with answer substitutions $<X \Leftarrow 0, Z \Leftarrow 0>$ or $<X \Leftarrow 1, Z \Leftarrow 1>$. If variable "Z" were "Y," the computation forests above are extended to success forests so that $\overline{success}(P_{echo})$ includes

$(\{shout\text{-}wait(X,Y), echo\text{-}back(X,Y)\}, \{shout\text{-}wait(0,0), echo\text{-}back(0,0)\})$,

$(\{shout\text{-}wait(X,Y), echo\text{-}back(X,Y)\}, \{shout\text{-}wait(1,1), echo\text{-}back(1,1)\})$,

that is, $\{shout\text{-}wait(X,Y), echo\text{-}back(X,Y)\}$ succeeds with answer substitutions $< X \Leftarrow 0, Y \Leftarrow 0>$ and $<X \Leftarrow 1, Y \Leftarrow 1>$.

Example 4.2.3 Consider the GHC program P_{loop} defining "*loop*." Then, $\overline{suspension}(P_{loop})$ is empty, while $\overline{success}(P_{loop})$ includes only pairs of the form

$(\{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}, \{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}\tau)$,

where τ is an m.g.u. of $\{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}$. (This set is included in $\overline{success}(P)$ for any program P.)

(3) Success-Suspension Set for Atoms

Definition Success-Suspension Pair

A pair $(A, A\sigma)$ is called a *success-suspension pair* (resp. *success pair*, *suspension pair*) *of* P when $(\{A\}, \{A\sigma\})$ is a success-suspension pair (resp. success pair, suspension pair) for goals of P.

Definition Success-Suspension Set

The set of all the success-suspension pairs (resp. success pair, suspension pair) of P is called the *success-suspension set* (resp. *success set*, *suspension set*) of P, and denoted by $SS(P)$ (resp. $success(P)$, $suspension(P)$).

Note again that two pairs corresponding to different computation forests with the same answer substitutions are considered identical elements in success-suspension sets.

Example 4.2.4 Consider the GHC program P_{join}. Then, $success(P_{join})$ includes

$(join([1],[2],Z), join([1],[2],[1,2]))$.

$suspension(P_{join})$ includes

$(join(X,[2],Z), join(X,[2],Z))$.

Example 4.2.5 Consider the GHC program P_{echo}. Then, $success(P_{echo})$ includes

$(shout\text{-}wait(X,0), shout\text{-}wait(0,0))$,

$(shout\text{-}wait(X,1), shout\text{-}wait(1,1))$.

$suspension(P_{echo})$ includes

$(shout\text{-}wait(X,Y), shout\text{-}wait(0,Y))$,
$(shout\text{-}wait(X,Y), shout\text{-}wait(1,Y))$.

Example 4.2.6 Consider the GHC program P_{loop}. Then, $success(P_{loop})$ includes only pairs of the form
$(s = t, s\sigma = t\sigma)$,
where σ is an m.g.u. of s and t. $suspension(P_{loop})$ is empty.

(4) Denotation of Monotonic GHC Programs

Definition Denotation of Goals, Atoms and Predicates of Monotonic GHC Programs
Let P be a monotonic GHC program. The *denotation of goal* Γ *in* P is the following *set* of pairs:
$D_P(\Gamma) = \{ (\Pi, \Pi\sigma) \mid \Pi$ is an instance of Γ and $(\Pi, \Pi\sigma) \in \overline{SS}(P) \}$.
The *denotation of atom "A" in P* is the following *set* of pairs:
$D_P(A) = \{ (B, B\sigma) \mid B$ is an instance of A and $(B, B\sigma) \in SS(P) \}$.
The *denotation of n-ary predicate "p" in P* is the denotation of atom $p(X_1, X_2, \ldots, X_n)$, i.e.,
$D_P(p) = \{ (p(t_1, t_2, \ldots, t_n), p(t_1, t_2, \ldots, t_n)\sigma) \mid$
$(p(t_1, t_2, \ldots, t_n), p(t_1, t_2, \ldots, t_n)\sigma) \in SS(P)$
where $t_1, t_2, \ldots, t_n$ are any terms $\}$.
The subscript P of D_P is ommitted when P is obvious from the context.

Definition Denotation of Monotonic GHC Programs
The *denotation of monotonic GHC program P* is the union of all the denotations of predicates $\bigcup_p D_P(p)$, i.e., $SS(P)$.

Note that we have thrown away the details of associated forests except their initial goals and their answer substitutions so that only the sets of atom (or goal) pairs are considered in the denotations.

Example 4.2.7 Consider P_{join}. Then $D(join)$ includes, for example
(join([1],[ ],Z), join([1],[ ],[1])),
(join([1],Y,[1]), join([1],Y,[1])).
$D(\leq)$ includes any pair $(t_1 \leq t_2, t_1 \leq t_2)$ when some instances of t_1 and t_2 are $suc^i(0)$ and $suc^j(s)$, where $i \leq j$ and s is any term. Similarly $D(>)$ includes any pair $(t_1 \leq t_2, t_1 \leq t_2)$ when some instances of t_1 and t_2 are $suc^i(s)$ and $suc^j(0)$, where $i < j$ and s is any term.

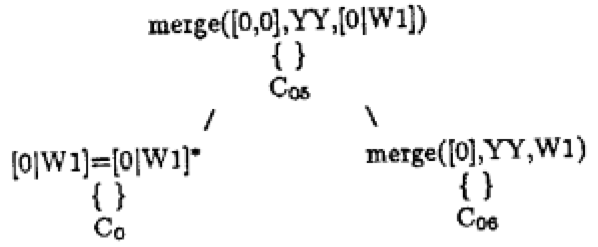
Example 4.2.8 Consider P_{echo}. Then $D(shout\text{-}wait)$ includes
(shout-wait(X,Y), shout-wait(0,Y)),
(shout-wait(X,Y), shout-wait(1,Y)),
(shout-wait(0,Y), shout-wait(0,Y)),
(shout-wait(1,Y), shout-wait(1,Y)),
(shout-wait(X,0), shout-wait(0,0)),
(shout-wait(X,1), shout-wait(1,1)),
(shout-wait(0,0), shout-wait(0,0)),
(shout-wait(1,1), shout-wait(1,1)).
$D(echo\text{-}back)$ includes
(echo-back(X,Y), echo-back(X,Y)),
(echo-back(X,0), echo-back(X,0)),
(echo-back(X,1), echo-back(X,1)),

(echo-back(0,Y), echo-back(0,0)),
(echo-back(1,Y), echo-back(1,1)),
(echo-back(0,0), echo-back(0,0)),
(echo-back(1,1), echo-back(1,1)).

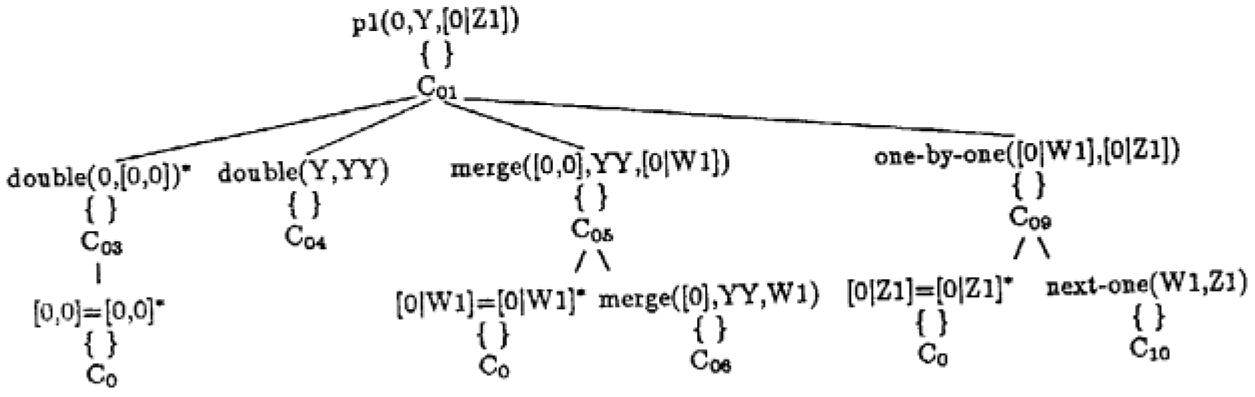
$D(wait0)$ and $D(wait1)$ include
(wait0(Y), wait0(Y)),
(wait0(0), wait0(0)),
(wait1(Y), wait1(Y)),
(wait1(1), wait1(1)).

Example 4.2.9 Consider P_{loop}. Then $D(loop)$ is empty.

Example 4.2.10 Consider the GHC program P_{BA} of Example 1.4 defining "*p1*," "*p2*," "*double*," "*merge*," "*one-by-one*," "*next-one*," "*two-at-once*" and "*complement*." Then $D(merge)$ includes, for example,
(merge([0,0],YY,W), merge([0,0],YY,[0|W1]))
which corresponds to the suspension forest below. (Notice the clause part C_{06} of the right child node.)



$D(p1)$ includes, for example,
(p1(0,Y,Z), p1(0,Y,[0|Z1])),
which corresponds to the suspension forest below:



while $D(p2)$ does not include
(p2(0,Y,Z), p2(0,Y,[0|Z2])).
Hence, "*p1*" and "*p2*" are discriminated in our success-suspension set semantics.

Remark. One might expect that we may adopt the success-suspension set for goals $\overline{SS}(P)$ as a semantics of GHC programs. However, it is too *global*. The set $\overline{SS}(P)$ considers all possible changes of goals whose initial goals possibly consist of many atoms. If this line were followed, the *denotation* of n-ary predicate "p" would be defined as follows:

$$\overline{D}_P(p) = \{(\Gamma, \Gamma\sigma) \mid (\Gamma, \Gamma\sigma) \in \overline{SS}(P) \text{ and }$$
$$\Gamma \text{ contains } p(t_1, t_2, \ldots, t_n) \text{ , where } t_1, t_2, \ldots, t_n \text{ are any terms } \}.$$

However, according to this definition, the denotation of "p" is concerned with all possible execution environment, i.e., co-existing goals, so that it lacks a *compositional* character. This set $\overline{D}(p)$ depends on other predicates in P, which might have no relation with "p."

4.3 Goal Composition for Monotonic GHC Programs

(1) Computation of Success-Suspension Set for Composed Goals

In the previous section, the denotations of atoms were defined using the success-suspension set for atoms. Then, how can the denotations of composed goals be restored from those of composing atoms?

We will prepare one more notion, which is to be used when it is necessary to obtain Δ from a given $\Delta\sigma$.

Definition Descendant Goal

Let P be a monotonic GHC program, Γ be a multiset of success-suspension atoms in P of the form

$\{A_1, A_2, \ldots, A_k\}$.

A multiset of success-suspension atoms Δ in P of the form

$\{B_1, B_2, \ldots, B_k\}$.

is called a *descendant goal* of Γ when A_l is an instance of B_l by a common substitution σ for $l = 1, 2, \ldots, k$.

Note that, for any given $\Delta\sigma$ and substitution σ, descendant goal Δ always exists, although it is not always unique. In the following goal composition, when Δ appears in the same context as $\Delta\sigma$, it denotes any of the descendant goals of $\Delta\sigma$.

By abuse of terminology, we will use the following terminology:

Definition Goal Interpretation

Let P be a monotonic GHC program. A *goal interpretation* of P is a set of expressions of the form

$(\{A_1, A_2, \ldots, A_k\}, \{A_1\sigma, A_2\sigma, \ldots, A_k\sigma\})$

where A_l and $A_l\sigma$ are success-suspension atoms for $l = 1, 2, \ldots, k$.

Then the transformation for goal composition is defined as follows:

Definition Transformation for Composing Goals of Monotonic GHC Programs

Let P be a monotonic GHC program, and $T_{compose}$ be the transformation of goal interpretations defined as follows:

$$T_{compose}(I) = \{ \ (\Gamma \cup \Delta, \Gamma\sigma \cup \Delta\sigma) \mid (\Gamma, \Gamma\sigma) \text{ and } (\Delta\sigma, \Delta\sigma) \text{ are in } I \ \}$$
$$\cup \{ \ (\Gamma \cup \Delta, \Gamma\sigma\tau \cup \Delta\sigma\tau) \mid (\Gamma, \Gamma\sigma) \text{ and } (\Gamma\sigma \cup \Delta\sigma, \Gamma\sigma\tau \cup \Delta\sigma\tau) \text{ are in } I \ \}.$$

The first set in the right-hand side, which corresponds to the base case of the inductive definition, says that, if goal Γ is maximally extended with answer substitution σ, and goal Δ is maximally extended without instantiation when the instantiation σ is received, then the composed goal $\Gamma \cup \Delta$ is maximally extended with answer substitution σ. The second set in the right-hand side, which corresponds to the induction step of the inductive definition, says that, if goal Γ is maximally extended with answer substitution σ, and goal $\Gamma \cup \Delta$ is maximally extended with answer substitution τ (possibly including further instantiation of $\Gamma\sigma$) when the instantiation σ is received, then the composed goal $\Gamma \cup \Delta$ is maximally extended with answer substitution $\sigma\tau$.

Theorem 4.3.1 Let P be a monotonic GHC program. Then

$$\overline{SS}(P) = \bigcup_{l=0}^{\infty} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\}),$$

where atom pair $(A, A\sigma)$ in $SS(P)$ is identified with goal pair $(\{A\}, \{A\sigma\})$.

Proof. We will show the following two inclusion relations:

Right Inclusion: $\overline{SS}(P) \supseteq \bigcup_{l=0}^{\infty} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$,

Left Inclusion: $\overline{SS}(P) \subseteq \bigcup_{l=0}^{\infty} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$.

Note that the monotonicity only affects the proof of "Right Inclusion."
Right Inclusion: We will show that any element in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ is an element in $\overline{SS}(P)$ by induction on n.
Base Case: When $n = 0$, then $\overline{SS}(P) \supseteq SS(P) \cup \{(\Box, \Box)\}$ is obvious.
Induction Step: Suppose that $\overline{SS}(P) \supseteq T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$.

As for the first set in the right-hand side of the definition of $T_{compose}$, let $(\Gamma, \Gamma\sigma)$ and $(\Delta\sigma, \Delta\sigma)$ be pairs in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$. From the induction hypothesis, there exist their associated forests $\mathcal{F}$ and $\mathcal{G}$. Then the union $\mathcal{F} \cup \mathcal{G}$ is a computation forest of $\Gamma \cup \Delta$ with answer substitution σ.

As for the second set in the right-hand side of the definition of $T_{compose}$, let $(\Gamma, \Gamma\sigma)$ and $(\Gamma\sigma \cup \Delta\sigma, \Gamma\sigma\tau \cup \Delta\sigma\tau)$ be pairs in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$. From the induction hypothesis, there exist their associated forests $\mathcal{F}$ and $\mathcal{G} \cup \mathcal{H}$. Let $\mathcal{G}'$ be the maximal committed subextension of $\Gamma\sigma$ in $\mathcal{G} \cup \mathcal{H}$ with answer substitution τ'. Then, from the monotonicity of P and Theorem 4.1, there must exist a success-suspension forest $\mathcal{G}''$ such that $\mathcal{G}''$ is a success-suspension forest of Γ with answer substitution $\sigma\tau'$ and the unsolved goal of $\mathcal{G}''$ is identical to that of $\mathcal{G}'$. Let $\bar{\mathcal{G}} \cup \bar{\mathcal{H}}$ be the success-suspension forest of $\Gamma \cup \Delta$ obtained by

(a) first applying GHC resolution in the same way as $\mathcal{G}''$, and
(b) then applying GHC resolution in the same way as $\mathcal{G} \cup \mathcal{H}$.

Then, $\bar{\mathcal{G}} \cup \bar{\mathcal{H}}$ is a success-suspension forest of $\Gamma \cup \Delta$ with answer substitution $\sigma\tau$.
Left Inclusion: We will show that any element in $\overline{SS}(P)$ is in $\bigcup_{l=0}^{\infty} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ by induction on the well-founded ordering $\ll$ of answer substitutions. (See the proof of Theorem 4.1.) Let $(\Gamma, \Gamma\sigma)$ be a pair in $\overline{SS}(P)$, and $\mathcal{F}$ be its associated forest.
Base Case: When σ is $<>$, let Γ be $\{A_1, A_2, \ldots, A_k\}$. Then the component computation trees of $\mathcal{F}$, whose atom parts of the root labels are $A_1, A_2, \ldots, A_k$, are success-suspension forests of $\{A_1\}, \{A_2\}, \ldots, \{A_k\}$ with answer substitution $<>$. Because the corresponding pairs are all in $SS(P)$, by applying $T_{compose}$ to $SS(P)$ k times, (Γ, Γ) is in $T^{k}_{compose}(SS(P) \cup \{(\Box, \Box)\})$.
Induction Step: When σ is not $<>$, from Lemma 3.2, there exists an atom A in Γ such that the maximal committed subextension $\mathcal{F}_A$ of $\{A\}$ in $\mathcal{F}$ has non-renaming answer substitution σ_A. Hence, pair $(\{A\}, \{A\sigma_A\})$ is in $T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ for any $l \geq 0$. Let σ' be the substitution such that $\sigma_A\sigma'$ is σ. From the induction hypothesis for σ', pair $(\Gamma\sigma_A, \Gamma\sigma_A\sigma')$ is

in $T^n_{compose}(SS(P) \cup \{(\Box,\Box)\})$ for some n. From the definition of transformation $T_{compose}$, pair $(\Gamma, \Gamma\sigma_A\sigma')$, i.e., $(\Gamma, \Gamma\sigma)$ is in $\bigcup_{l=0}^{\infty} T^l_{compose}(SS(P) \cup \{(\Box,\Box)\})$.

Example 4.3.1 Consider P_{join}. Since $SS(P_{join})$ includes
  (join([1],[2],Z), join([1],[2],[1,2])),
  (join([1],[ ],[1]), join([1],[ ],[1])),
$\overline{SS}(P_{join})$ includes
  ({join([1],[ ],[1]), join([1],[2],Z)}, {join([1],[ ],[1]), join([1],[2],[1,2])}).
Again, since $SS(P_{join})$ includes
  (join([1],[ ],X), join([1],[ ],[1])),
$\overline{SS}(P_{join})$ includes
  ({join([1],[ ],X), join(X,[2],Z)}, {join([1],[ ],[1]), join([1],[2],[1,2])}).

Example 4.3.2 In the previous example, the information of instantiation is propagated from one process to the other process only once. In general, the execution proceeds by passing information to each other more than once, e.g., "*seesaw*" of Example 1.3. Here, consider P_{echo}. Since $SS(P_{echo})$ includes
  (echo-back(0,Y), echo-back(0,0)),
  (echo-back(1,Y), echo-back(1,1)),
  (shout-wait(0,0), shout-wait(0,0)),
  (shout-wait(1,1), shout-wait(1,1)),
$\overline{SS}(P_{echo})$ includes
  ({shout-wait(0,Y), echo-back(0,Y)}, {shout-wait(0,0), echo-back(0,0)}),
  ({shout-wait(1,Y), echo-back(1,Y)}, {shout-wait(1,1), echo-back(1,1)}).
Again, since $SS(P_{echo})$ includes
  (shout-wait(X,Y), shout-wait(0,Y)),
  (shout-wait(X,Y), shout-wait(1,Y)),
$\overline{SS}(P_{echo})$ includes
  ({shout-wait(X,Y), echo-back(X,Y)}, {shout-wait(0,0), echo-back(0,0)}),
  ({shout-wait(X,Y), echo-back(X,Y)}, {shout-wait(1,1), echo-back(1,1)}).
Note that these pairs in $\overline{success}(P_{echo})$ would not be computed if $suspension(P_{echo})$ were not included in the denotation.

Example 4.3.3 Consider P_{loop}. Since $SS(P_{loop})$ includes only pairs of the form
  $(s = t, s\sigma = t\sigma)$
where σ is an m.g.u. of s and t, $\overline{SS}(P_{loop})$ includes only pairs of the form
  $(\{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}, \{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}\tau)$,
where τ is an m.g.u. of $\{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}$.

Example 4.3.4 Consider the GHC program P_{BA}. First, since $SS(P_{BA})$ includes
  (p1(0,1,[0|Z1]), p1(0,1,[0,1])),
  (complement([0,1],1), complement([0,1],1)),
$\overline{SS}(P_{BA})$ includes
  ({p1(0,1,[0|Z1]),complement([0|Z1],1)}, {p1(0,1,[0,1]),complement([0,1],1)}),
Next, since $SS(P_{BA})$ includes
  (complement([0|Z1],Y), complement([0|Z1],1)),
$\overline{SS}(P_{BA})$ includes
  ({p1(0,Y,[0|Z1]),complement([0|Z1],Y)}, {p1(0,1,[0,1]),complement([0,1],1)}).
Then, since $SS(P_{BA})$ includes

(p1(0,Y,Z), p1(0,Y,[0|Z1])),
$\overline{SS}(P_{BA})$ includes
({p1(0,Y,Z),complement(Z,Y)}, {p1(0,1,[0,1]),complement([0,1],1)}),
while $\overline{SS}(P_{BA})$ does not include
({p2(0,Y,Z),complement(Z,Y)}, {p2(0,1,[0,1]),complement([0,1],1)}),
since $SS(P_{BA})$ does not include
(p2(0,Y,Z), p2(0,Y,[0|Z1])).

Remark. It is not difficult to modify the definition of $T_{compose}$ above so as to respect the distinction between success pairs and suspension pairs. Let $T_{compose}$ be the transformation of pairs of goal interpretations defined as follows:

$T_{compose}(I_1, I_2) = (J_1, J_2)$ where
$J_1 = \{ (\Gamma \cup \Delta, \Gamma\sigma \cup \Delta\sigma) \mid (\Gamma, \Gamma\sigma), (\Delta\sigma, \Delta\sigma)$ are in $I_1 \}$
$\cup \{ (\Gamma \cup \Delta, \Gamma\sigma\tau \cup \Delta\sigma\tau) \mid (\Gamma, \Gamma\sigma)$ is in $I_1 \cup I_2$, and $(\Gamma\sigma \cup \Delta\sigma, \Gamma\sigma\tau \cup \Delta\sigma\tau)$ is in $I_1 \}$.
$J_2 = \{ (\Gamma \cup \Delta, \Gamma\sigma \cup \Delta\sigma) \mid (\Gamma, \Gamma\sigma), (\Delta\sigma, \Delta\sigma)$ are in $I_1 \cup I_2$ and at least one is in $I_2 \}$
$\cup \{ (\Gamma \cup \Delta, \Gamma\sigma\tau \cup \Delta\sigma\tau) \mid (\Gamma, \Gamma\sigma)$ is in $I_1 \cup I_2$, and $(\Gamma\sigma \cup \Delta\sigma, \Gamma\sigma\tau \cup \Delta\sigma\tau)$ is in $I_2 \}$.

Then, $(\overline{success}(P), \overline{suspension}(P)) = \bigcup_{l=0}^{\infty} T^{l}_{compose}(success(P) \cup \{(\Box, \Box)\}, suspension(P))$.

(2) Replacement with Equivalence

Is it permissible to replace one atom in a goal with another atom, when they have the same denotations?

Definition Equivalent Goals, Atoms and Predicates of Monotonic GHC Programs
Let P be a monotonic GHC program. Two goals Γ and Δ are said to be *equivalent in* P when
(a) the set of variables occurring in Γ and that in Δ are identical, and
(b) $(\Gamma\sigma, \Gamma\sigma\tau)$ is in $D(\Gamma)$ if and only if $(\Delta\sigma, \Delta\sigma\tau)$ is in $D(\Delta)$ for any substitutions σ and τ.
Two atoms "A" and "B" are said to be *equivalent in* P when two singleton goals $\{A\}$ and $\{B\}$ are equivalent in P. Two n-ary predicates "p" and "q" are said to be *equivalent in* P when two atoms $p(X_1, X_2, \ldots, X_n)$ and $q(X_1, X_2, \ldots, X_n)$ are equivalent in P, where $X_1, X_2, \ldots, X_n$ are distinct variables.

Theorem 4.3.2 Let P be a monotonic GHC program. Then, for any two atoms A and B equivalent in P, goals $\Gamma \cup \{A\}$ and $\Gamma \cup \{B\}$ are equivalent in P for any goal Γ.

Proof. It suffices to show that $((\Pi \cup \{A\})\sigma, (\Pi \cup \{A\})\sigma\tau)$ is in $\overline{SS}(P)$ if and only if $((\Pi \cup \{B\})\sigma, (\Pi \cup \{B\})\sigma\tau)$ is in $\overline{SS}(P)$ for any Π and σ, τ. Then from Theorem 4.3.1, it suffices to show that $((\Pi \cup \{A\})\sigma, (\Pi \cup \{A\})\sigma\tau)$ is in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ if and only if $((\Pi \cup \{B\})\sigma, (\Pi \cup \{B\})\sigma\tau)$ is in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ for any Π, σ, τ and n. This is easily proved by induction on n using the definition of transformation $T_{compose}$.

To summarize, for monotonic GHC programs,
(a) the denotations of composed goals can be computed from the denotations of composing atoms, and
(b) if two atoms A and B are equivalent, then any two goals obtained by adding A or B to any common goal are also equivalent.

Hence, the success-suspension set $SS(P)$ (or something containing the equivalent information) is one of the qualified candidates of the semantics of monotonic GHC programs.

5. Semantics of General GHC Programs

Unfortunately, neither the goal composition method would work nor the semantic property in Section 4.3 would hold for general GHC programs if the success-suspension sets were adopted as their semantics. This section first examines why the semantics is not appropriate for general GHC programs, then defines partially ordered success-suspension multisets as their semantics, and last generalizes the results in Section 4.3

5.1 Problems of Non-monotonic GHC Programs

What is the reason the previous goal composition method does not work for non-monotonic GHC programs?

Example 5.1.1 Recall the GHC program P_{non} of Example 4.1.5.

```
    C1: non(X,Y) :- | X=0, wait0(Y).
    C2: non(X,1).
    C3: any(0,Y) :- | Y=0.
    C4: any(0,Y) :- | Y=1.
    C5: wait0(0).
```

Then, since $SS(P_{non})$ includes

(any(0,Y), any(0,1)),
(non(0,1), non(0,1)),

$T_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$ includes

({non(0,Y),any(0,Y)}, {non(0,1),any(0,1)}),

Again, since $SS(P_{non})$ includes

(non(X,Y), non(0,Y)),

$T^2_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$ includes

({non(X,Y),any(X,Y)}, {non(0,1),any(0,1)}).

Hence $\overline{SS}(P_{non})$ would include a wrong pair if the goal composition method for monotonic GHC programs were adopted for non-monotonic GHC programs.

Is it the responsibility of the goal composition method, or the responsibility of the semantics itself? The following example says that it is the responsibility of the semantics itself.

Example 5.1.2 Consider the following two programs:

```
    P1 : non(X,Y) :- | X=0, wait0(Y).
         non(X,1).
         non(X,1) :- | X=0.
         any(0,Y) :- | Y=0.
         any(0,Y) :- | Y=1.
         wait0(0).
    P2 : non(X,Y) :- | X=0, wait(Y).
         non(X,1).
         non(X,1) :- | X=0.
         any(0,Y) :- | Y=0.
         any(0,Y) :- | Y=1.
         wait(0).
         wait(1).
```

Then, although $D(non)$ and $D(any)$ are identical for P_1 and P_2, goal $\{non(X,Y), any(X,Y)\}$ cannot succeed with answer substitution $< X \Leftarrow 0, Y \Leftarrow 1 >$ in P_1, while it can succeed with answer substitution $< X \Leftarrow 0, Y \Leftarrow 1 >$ in P_2.

In general, if two computation forests with an identical answer substitution are considered identical, then the extension ordering between success-suspension forests is not reflected in the semantics so that pairs at different positions w.r.t. the extension ordering are considered identical, which leads the goal composition method in Section 4.3 to computing wrong pairs.

5.2 Partially Ordered Success-Suspension Multiset

Now on in Section 5, let P be a general GHC program.

(1) Success-Suspension Atom with Superscript

Definition Success-Suspension Atom with Superscript

A parenthesized superscript of atom A is used as an identifier for success-suspension forests of goal $\{A\}$. Two parenthesized superscripts of atom A are identical if and only if the computation forests of goal $\{A\}$ identified by those parenthesized superscripts are identical. (Two parenthesized superscripts of different atoms may be identical even if the computation forests identified by those parenthesized superscripts are different.) Parenthesized superscripts, simply *superscript* hereafter, are denoted by $(a), (b), (c), (d), (e), (f)$.

An atom with superscript $A^{(a)}$ is called a *success-suspension atom* (resp. *success atom*, *suspension atom*) *in* P when there exists a success-suspension forest (resp. success forest, suspension forest) of goal $\{A\}$ in P identified by superscript (a). The success-suspension forest is called the *associated forest* of $A^{(a)}$.

Let $A^{(a)}$ and $B^{(b)}$ be success-suspension atoms in P whose associated forests are $\mathcal{F}$ and $\mathcal{G}$. Then $A^{(a)} \preceq B^{(b)}$ if and only if $\mathcal{F}$ is extensible to $\mathcal{G}$.

Note that, an atom $A^{(a)}$ satisfying $A^{(a)} \preceq A\sigma^{(b)}$ always exists uniquely for any given success-suspension atom $A\sigma^{(b)}$ and atom A, because, if $\mathcal{G}$ is the associated forest of $A\sigma^{(b)}$, then $A^{(a)} \preceq A\sigma^{(b)}$ if and only if the associated forest $\mathcal{F}$ of $A^{(a)}$ is the maximal committed subextension of $\{A\}$ in $\mathcal{G}$.

Example 5.2.1 Consider program P_{non}. Superscripts (1) and (2) are attached to $non(0,Y)$ to make a distinction between two seemingly identical atoms, because two computation forests below are different computation forests of $non(0,Y)$.

```
                non(0,Y)                          non(0,Y)
                  { }                               { }
                  C1                                C2
                  / \
      0=0*                wait0(Y)
      { }                   { }
      C0                    C5
```

Similarly, superscripts (1) and (2) are attached to $non(X,Y)$, because two computation forests below are different computation forests of $non(X,Y)$.

$$\begin{tikzcd}[column sep=small]
 & \begin{array}{c}\mathrm{non(0,Y)}\\ \{\,\}\\ C_1\end{array} \arrow[dl, dash] \arrow[dr, dash] & & & \begin{array}{c}\mathrm{non(X,Y)}\\ \{\,\}\\ C_2\end{array} \\
\begin{array}{c}0{=}0^{*}\\ \{\,\}\\ C_0\end{array} & & \begin{array}{c}\mathrm{wait0(Y)}\\ \{\,\}\\ C_5\end{array} & &
\end{tikzcd}$$

Note that, due to the distinction by their superscripts,

$non(X,Y)^{(1)} \preceq non(0,Y)^{(1)}$,
$non(X,Y)^{(2)} \preceq non(0,Y)^{(2)}$,
$non(X,Y)^{(1)} \not\preceq non(0,Y)^{(2)}$,
$non(X,Y)^{(2)} \not\preceq non(0,Y)^{(1)}$.

Also note that $non(0,Y)^{(1)} \not\preceq non(0,1)$, while $non(0,Y)^{(2)} \preceq non(0,1)$. (In the following, when atom A has only one success-suspension forest, its superscript is ommitted for notational simplicity.)

(2) Partially Ordered Success-Suspension Multiset for Composed Goals

Definition Success-Suspension Pair for Goals with Superscript

A pair $(\Gamma^{(a)}, \Gamma\sigma^{(b)})$ is called a *success-suspension pair* (resp. *success pair*, *suspension pair*) *for goal of* P when there exists a success-suspension forest (resp. success forest, suspension forest) $\mathcal{F}$ of goal Γ with answer substitution σ in P such that

(a) the superscript (a) is an abbreviation of the sequence of the superscripts $(a_1)(a_2)\cdots(a_k)$ of Γ's component atoms $A_1, A_2, \ldots, A_k$, where (a_l) indicates that the maximal committed subextension of $\{A_l\}$ in $\mathcal{F}$ is the success-suspension forest of A_l identified by (a_l) for $l = 1, 2, \ldots, k$, and

(b) the superscript (b) is an abbreviation of the sequence of the superscripts $(b_1)(b_2)\cdots(b_k)$ of $\Gamma\sigma$'s component atoms $A_1\sigma, A_2\sigma, \ldots, A_k\sigma$, where (b_l) indicates that the l-th component computation tree of $\mathcal{F}$ is the success-suspension forest of $A_l\sigma$ identified by (b_l) for $l = 1, 2, \ldots, k$.

The success-suspension forest $\mathcal{F}$ is called the *associated forest* of the pair. Two success-suspension pairs with superscripts are considered identical when the corresponding goals are identical modulo renaming of variables and the corresponding superscripts are identical.

Let $(\Gamma^{(a)}, \Gamma\sigma^{(b)})$ and $(\Delta^{(d)}, \Delta\tau^{(e)})$ be two success-suspension pairs of the form

$(\{A_1^{(a_1)}, A_2^{(a_2)}, \ldots, A_k^{(a_k)}\}, \{A_1\sigma^{(b_1)}, A_2\sigma^{(b_2)}, \ldots, A_k\sigma^{(b_k)}\})$,
$(\{B_1^{(d_1)}, B_2^{(d_2)}, \ldots, B_k^{(d_k)}\}, \{B_1\tau^{(e_1)}, B_2\tau^{(e_2)}, \ldots, B_k\tau^{(e_k)}\})$.

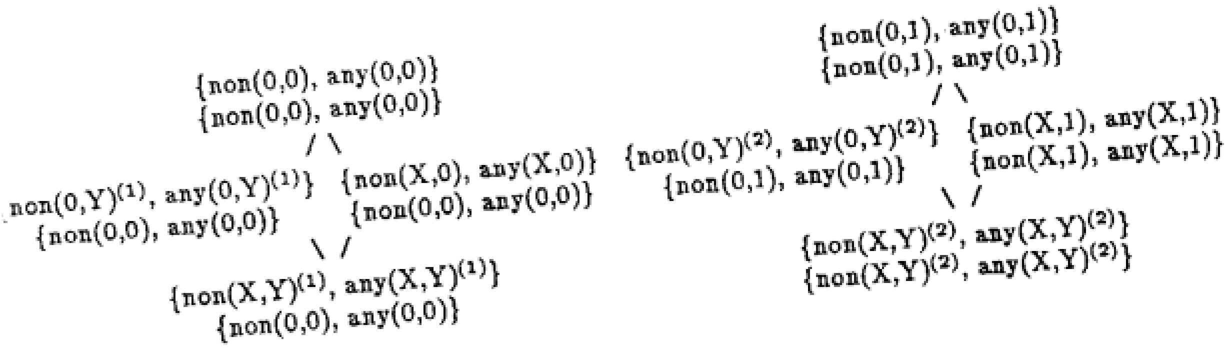
Then $(\Gamma^{(a)}, \Gamma\sigma^{(b)}) \preceq (\Delta^{(d)}, \Delta\tau^{(e)})$ if and only if $A_l^{(a_l)} \preceq B_l^{(d_l)}$ and $A_l\sigma^{(b_l)} \preceq B_l\tau^{(e_l)}$ for $l = 1, 2, \ldots, k$.

Definition Success-Suspension Multiset for Goals

The partially ordered set of all the success-suspension pairs (resp. success pairs, suspension pairs) for goals of P is called the *success-suspension multiset* (resp. *success multiset*, *suspension multiset*) *for goals of* P, and denoted by $\overline{SS}(P)$ (resp. $\overline{success}(P)$, $\overline{suspension}(P)$).

Although the term "multiset" is a little confusing, we have used it to mean that seemingly identical pairs of goals are distinguished only by their superscripts.

Example 5.2.2 Consider P_{non}. Then, the figure below is a part of $\overline{SS}(P_{non})$.



(3) Partially Ordered Success-Suspension Multiset for Atoms

Definition Success-Suspension Pair with Superscript

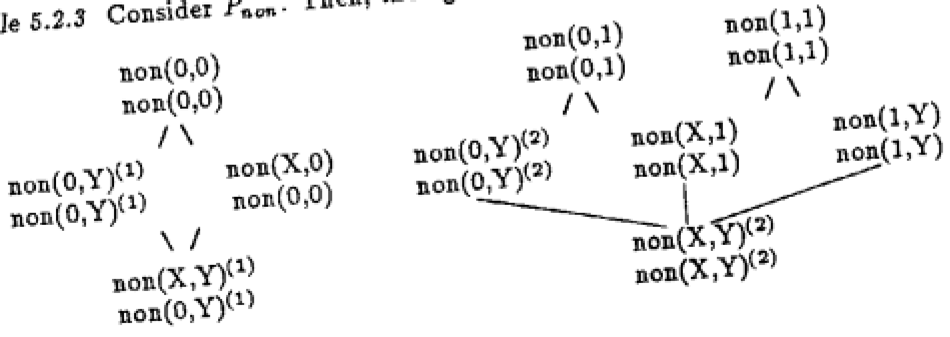
A pair $(A^{(a)}, A\sigma^{(b)})$ is called a *success-suspension pair* (resp. *success pair*, *suspension pair*) *of P* when $(\{A\}^{(a)}, \{A\sigma\}^{(b)})$ is a success-suspension pair (resp. success pair, suspension pair) for goals of P.

Let $(A^{(a)}, A\sigma^{(b)})$ and $(B^{(d)}, B\tau^{(e)})$ be two success-suspension pairs. Then $(A^{(a)}, A\sigma^{(b)}) \preceq (B^{(d)}, B\tau^{(e)})$ if and only if $(\{A\}^{(a)}, \{A\sigma\}^{(b)}) \preceq (\{B\}^{(d)}, \{B\tau\}^{(e)})$.

Definition Success-Suspension Multiset

The partially ordered set of all the success-suspension pairs (resp. success pairs, suspension pairs) of P is called the *success-suspension multiset* (resp. *success multiset*, *suspension multiset*) *of P*, and denoted by $SS(P)$ (resp. $success(P)$, $suspension(P)$).

Example 5.2.3 Consider P_{non}. Then, the figure below is a part of $SS(P_{non})$.



(4) Denotation of General GHC Programs

Definition Denotation of Goals, Atoms and Predicates of General GHC Programs

Let P be a general GHC program. The *denotation of goal* Γ *in P* is the following partially ordered multiset of pairs:

$D_P(\Gamma) = \{\ (\Pi^{(a)}, \Pi\sigma^{(b)}) \mid \Pi$ is an instance of Γ and $(\Pi^{(a)}, \Pi\sigma^{(b)}) \in \overline{SS}(P)\ \}$.

The *denotation of atom "A" in P* is the following set of pairs:

$D_P(A) = \{\ (B^{(a)}, B\sigma^{(b)}) \mid B$ is an instance of A and $(B^{(a)}, B\sigma^{(b)}) \in SS(P)\ \}$.

The *denotation of n-ary predicate "p" in P* is the denotation of atom $p(X_1, X_2, \ldots, X_n)$, i.e.,

$D_P(p) = \{\ (p(t_1, t_2, \ldots, t_n)^{(a)}, p(t_1, t_2, \ldots, t_n)\sigma^{(b)}) \mid$
$(p(t_1, t_2, \ldots, t_n)^{(a)}, p(t_1, t_2, \ldots, t_n)\sigma^{(b)}) \in SS(P)$
where $t_1, t_2, \ldots, t_n$ are any terms $\}$.

The subscript P of D_P is ommitted when P is obvious from the context.

Definition Denotation of General GHC Program

The *denotation of general GHC program P* is the union of all the denotations of predicates $\bigcup_{p} D_P(p)$, i.e., $SS(P)$.

Note that we have thrown away the details of associated forests except their initial goals, their answer substitutions and the extension ordering between those computation forests so that only the partially ordered multisets of atom (or goal) pairs are considered in the denotations.

Example 5.2.4 Consider P_{non}. Then, $D(non)$ is the partially ordered multiset of Example 5.2.3, and $D(any)$ and $D(wait0)$ are the partially ordered multisets below.

$$\begin{tikzcd}
& \substack{\mathrm{any}(0,0)\\ \mathrm{any}(0,0)} \arrow[dl, dash] \arrow[dr, dash] & \\
\substack{\mathrm{any}(0,Y)^{(1)}\\ \mathrm{any}(0,0)} \arrow[dr, dash] & & \substack{\mathrm{any}(X,0)\\ \mathrm{any}(X,0)} \arrow[dl, dash] \\
& \substack{\mathrm{any}(X,Y)^{(1)}\\ \mathrm{any}(X,Y)^{(1)}} &
\end{tikzcd}
\qquad
\begin{tikzcd}
& \substack{\mathrm{any}(0,1)\\ \mathrm{any}(0,1)} \arrow[dl, dash] \arrow[dr, dash] & \\
\substack{\mathrm{any}(0,Y)^{(2)}\\ \mathrm{any}(0,1)} \arrow[dr, dash] & & \substack{\mathrm{any}(X,1)\\ \mathrm{any}(X,1)} \arrow[dl, dash] \\
& \substack{\mathrm{any}(X,Y)^{(2)}\\ \mathrm{any}(X,Y)^{(2)}} &
\end{tikzcd}$$

$$\begin{tikzcd}
\substack{\mathrm{wait0}(0)\\ \mathrm{wait0}(0)} \arrow[d, dash] \\
\substack{\mathrm{wait0}(Y)\\ \mathrm{wait0}(Y)}
\end{tikzcd}$$

Note that superscripts (1) and (2) are attached to $any(0,Y)$ to make a distinction between two seemingly identical atoms. Also note that $any(0,Y)^{(1)} \preceq any(0,0)$ and $any(X,Y)^{(1)} \preceq any(0,Y)^{(1)}$.

Remark. Let $(A, A\sigma)$ be a success pair in $D(p)$. (For simplicity, we have omitted the superscripts of success-suspension atoms.) Let σ', σ'' be any substitutions such that $\sigma'\sigma''$ is σ, and θ be any substitution for the variables occurring in $A\sigma$. Then, $(A\sigma', A\sigma'\sigma'')$ and $(A\sigma\theta, A\sigma\theta)$ are success pairs in $D(p)$, and

$(A, A\sigma) \preceq (A\sigma', A\sigma'\sigma'') \preceq (A\sigma\theta, A\sigma\theta)$.

Hence, the partial ordering structure of $D(P)$ above success pairs are determined independently from program P. The interesting part, which depends on program P, in the partial ordering structure of $D(p)$ is that below success pairs.

5.3 Goal Composition for General GHC Programs

(1) Computation of Partially Ordered Success-Suspension Multiset for Composed Goals

Then, how can the denotations of composed goals be restored from the denotations of composing atoms for general GHC programs?

Again, we will prepare one more notion, which is to be used when it is necessary to obtain Δ from a given $\Delta\sigma$.

Right Inclusion: We will show that any element in $\bigcup_{l=0}^{n} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ is an element in $\overline{SS}(P)$ by induction on n.

Base Case: When $n = 0$, then $\overline{SS}(P) \supseteq SS(P) \cup \{(\Box, \Box)\}$ is obvious.

Induction Step: Suppose that $\overline{SS}(P) \supseteq T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$.

As for the first set in the right-hand side of the definition of $T_{compose}$, let $(\Gamma^{(a)}, \Gamma\sigma^{(b)})$ be a pair in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ and $(\Delta\sigma^{(c)}, \Delta\sigma^{(c)})$ be a pair in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$. From the induction hypothesis, there exist their associated forests $\mathcal{F}$ and $\mathcal{G}$. Then the union $\mathcal{F} \cup \mathcal{G}$ is a computation forest of $\Gamma^{(a)} \cup \Delta^{(d)}$ with answer substitution σ, since $\Delta^{(d)}$ is a descendant goal of $\Delta\sigma^{(c)}$.

As for the second set in the right-hand side of the definition of $T_{compose}$, let $(\Gamma^{(a)}, \Gamma\sigma^{(b)})$ and $(\Gamma\sigma^{(b)} \cup \Delta\sigma^{(c)}, \Gamma\sigma\tau^{(c)} \cup \Delta\sigma\tau^{(f)})$ be pairs in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$. From the induction hypothesis, there exist their associated forests $\mathcal{F}$ and $\mathcal{H}$. Then $\mathcal{H}$ is a success-suspension forest of $\Gamma^{(a)} \cup \Delta^{(d)}$ with answer substitution $\sigma\tau$, since $\Delta^{(d)}$ is a descendant goal of $\Delta\sigma^{(c)}$.

Left Inclusion: We will show that any element in $\overline{SS}(P)$ is in $\bigcup_{l=0}^{\infty} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ by induction on the well-founded ordering $\ll$ of answer substitutions. (See the proof of Theorem 4.1.) Let $(\Gamma^{(a)}, \Gamma\sigma^{(c)})$ be an element in $\overline{SS}(P)$, and $\mathcal{G}$ be its associated forest.

Base Case: When σ is $<>$, let $\Gamma^{(a)}$ be $\{A_1^{(a_1)}, A_2^{(a_2)}, \ldots, A_k^{(a_k)}\}$. Then the component computation trees of $\mathcal{G}$, whose atom parts of the root labels are $A_1^{(a_1)}, A_2^{(a_2)}, \ldots, A_k^{(a_k)}$, are success-suspension forests of $\{A_1^{(a_1)}\}, \{A_2^{(a_2)}\}, \ldots, \{A_k^{(a_k)}\}$ with answer substitution $<>$. Because the corresponding pairs are all in $SS(P)$, by applying $T_{compose}$ to $SS(P)$ k times, $(\Gamma^{(a)}, \Gamma^{(a)})$ is in $T^{k}_{compose}(SS(P) \cup \{(\Box, \Box)\})$.

Induction Step: When σ is not $<>$, from Lemma 3.2, there exists an atom $A^{(a_0)}$ in $\Gamma^{(a)}$ such that the maximal committed subextension $\mathcal{G}_A$ of $\{A^{(a_0)}\}$ in $\mathcal{G}$ has non-renaming answer substitution σ_A. Hence, pair $(\{A^{(a_0)}\}, \{A\sigma_A^{(b_0)}\})$ is in $T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ for any $l \geq 0$. Let σ' be the substitution such that $\sigma_A\sigma'$ is σ. From the induction hypothesis for σ', pair $(\Gamma\sigma_A^{(b)}, \Gamma\sigma_A\sigma'^{(c)})$ is in $T^{n}_{compose}(SS(P) \cup \{(\Box, \Box)\})$ for some n, where $\Gamma\sigma_A^{(b)}$ is a descendant goal of $\Gamma\sigma^{(c)}$. From the definition of transformation $T_{compose}$, pair $(\Gamma^{(a)}, \Gamma\sigma_A\sigma'^{(c)})$, i.e., $(\Gamma^{(a)}, \Gamma\sigma^{(c)})$ is in $\bigcup_{l=0}^{\infty} T^{l}_{compose}(SS(P) \cup \{(\Box, \Box)\})$.

Example 5.3 Consider how $D(\{non(X,Y), any(X,Y)\})$ is computed from $SS(P_{non})$. First, $T_{compose}(SS(P_{non}) \cup \{(\Box, \Box)\})$ includes the partially ordered multiset below:

$$\begin{tikzcd}
 & \begin{array}{c}\{\text{non(0,0), any(0,0)}\}\\ \{\text{non(0,0), any(0,0)}\}\end{array} \arrow[dl, dash] \arrow[dr, dash] & & & \begin{array}{c}\{\text{non(0,1), any(0,1)}\}\\ \{\text{non(0,1), any(0,1)}\}\end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{array}{c}\{\text{non(0,Y)}^{(1)}\text{, any(0,Y)}^{(1)}\}\\ \{\text{non(0,0), any(0,0)}\}\end{array} & & \begin{array}{c}\{\text{non(X,0), any(X,0)}\}\\ \{\text{non(0,0), any(0,0)}\}\end{array} & \begin{array}{c}\{\text{non(0,Y)}^{(2)}\text{, any(0,Y)}^{(2)}\}\\ \{\text{non(0,1), any(0,1)}\}\end{array} \arrow[dr, dash] & & \begin{array}{c}\{\text{non(X,1), any(X,1)}\}\\ \{\text{non(X,1), any(X,1)}\}\end{array} \arrow[dl, dash] \\
 & & & & \begin{array}{c}\{\text{non(X,Y)}^{(2)}\text{, any(X,Y)}^{(2)}\}\\ \{\text{non(X,Y)}^{(2)}\text{, any(X,Y)}^{(2)}\}\end{array} &
\end{tikzcd}$$

For example, pair

$(\{\text{non(X,0)}^{(1)}\text{,any(X,0)}\}, \{\text{non(0,0),any(0,0)}\})$,

is in $T_{compose}(SS(P_{non}) \cup \{(\Box, \Box)\})$, because

$(\text{non(X,0)}^{(1)}, \text{non(0,0)})$,

Definition Descendant Goal

Let P be a general GHC program, and $\Gamma^{(a)}$ be a multiset of success-suspension atoms in P of the form

$$\{A_1^{(a_1)}, A_2^{(a_2)}, \ldots, A_k^{(a_k)}\}.$$

A multiset of success-suspension atoms $\Delta^{(b)}$ in P of the form

$$\{B_1^{(b_1)}, B_2^{(b_2)}, \ldots, B_k^{(b_k)}\}.$$

is called a *descendant goal* of $\Gamma^{(a)}$ when A_l is an instance of B_l by a common substitution σ and $B_l^{(b_l)} \preceq A_l^{(a_l)}$ for $l = 1, 2, \ldots, k$.

Note that, for any given $\Delta\sigma^{(e)}$ and substitution σ, descendant goal $\Delta^{(d)}$ always exists, although Δ is not always unique. However, once Δ is fixed, $\Delta^{(d)}$ is uniquely determined. In the following goal composition, when $\Delta^{(d)}$ appears in the same context as $\Delta\sigma^{(e)}$, it denotes any of the descendant goals of $\Delta\sigma^{(e)}$.

Again, by abuse of terminology, we will use the following terminology:

Definition Goal Interpretation

Let P be a general GHC program. A *goal interpretation* of P is a partially ordered set of expressions of the form

$$(\{A_1^{(a_1)}, A_2^{(a_2)}, \ldots, A_k^{(a_k)}\}, \{A_1\sigma^{(b_1)}, A_2\sigma^{(b_2)}, \ldots, A_k\sigma^{(b_k)}\}),$$

where $A_l^{(a_l)}$ and $A_l\sigma^{(b_l)}$ are success-suspension atoms for $l = 1, 2, \ldots, k$.

Let $(\Gamma^{(a)}, \Gamma\sigma^{(b)})$ and $(\Delta^{(d)}, \Delta\tau^{(e)})$ be two pairs in a goal interpretation of the form

$$(\{A_1^{(a_1)}, A_2^{(a_2)}, \ldots, A_k^{(a_k)}\}, \{A_1\sigma^{(b_1)}, A_2\sigma^{(b_2)}, \ldots, A_k\sigma^{(b_k)}\}),$$
$$(\{B_1^{(d_1)}, B_2^{(d_2)}, \ldots, B_k^{(d_k)}\}, \{B_1\tau^{(e_1)}, B_2\tau^{(e_2)}, \ldots, B_k\tau^{(e_k)}\}).$$

Then $(\Gamma^{(a)}, \Gamma\sigma^{(b)}) \preceq (\Delta^{(d)}, \Delta\tau^{(e)})$ if and only if $A_l^{(a_l)} \preceq B_l^{(d_l)}$ and $A_l\sigma^{(b_l)} \preceq B_l\tau^{(e_l)}$ for $l = 1, 2, \ldots, k$.

Then the transformation for goal composition is defined as follows:

Definition Transformation for Composing Goals of General GHC Programs

Let P be a general GHC program, and $T_{compose}$ be the transformation of goal interpretations defined as follows:

$$\begin{aligned} T_{compose}(I) = \{\ & (\Gamma^{(a)} \cup \Delta^{(d)}, \Gamma\sigma^{(b)} \cup \Delta\sigma^{(e)}) \mid \\ & (\Gamma^{(a)}, \Gamma\sigma^{(b)}) \text{ and } (\Delta\sigma^{(e)}, \Delta\sigma^{(e)}) \text{ are in } I\ \} \\ \cup \{\ & (\Gamma^{(a)} \cup \Delta^{(d)}, \Gamma\sigma\tau^{(c)} \cup \Delta\sigma\tau^{(f)}) \mid \\ & (\Gamma^{(a)}, \Gamma\sigma^{(b)}) \text{ and } (\Gamma\sigma^{(b)} \cup \Delta\sigma^{(e)}, \Gamma\sigma\tau^{(c)} \cup \Delta\sigma\tau^{(f)}) \text{ are in } I\ \}. \end{aligned}$$

Note that, in the definition above, $\Delta^{(d)}$ is a descendant goal of $\Delta\sigma^{(e)}$.

Theorem 5.3.1 Let P be a general GHC program. Then

$$\overline{SS}(P) = \bigcup_{l=0}^{\infty} T_{compose}^{l}(SS(P) \cup \{(\square, \square)\}),$$

where atom pair $(A^{(a)}, A\sigma^{(b)})$ in $SS(P)$ is identified with goal pair $(\{A\}^{(a)}, \{A\sigma\}^{(b)})$.

Proof. The theorem is proved along the same line as Theorem 4.3.1 taking the distinction by superscripts into consideration. Again, we will show the following two inclusion relations:

Right Inclusion: $\overline{SS}(P) \supseteq \bigcup_{l=0}^{\infty} T_{compose}^{l}(SS(P) \cup \{(\square, \square)\})$

Left Inclusion: $\overline{SS}(P) \subseteq \bigcup_{l=0}^{\infty} T_{compose}^{l}(SS(P) \cup \{(\square, \square)\})$

(any(0,0), any(0,0))
are in $SS(P_{non})$ and $\{any(X,0)\}$ is a descendant goal of $\{any(0,0)\}$. Note that the denotation of $\{non(X,Y), any(X,Y)\}$ computed by $T_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$ is the partially ordered multiset of Example 5.2.2 except that one pair at the left bottom is missed. Then next, $T^2_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$ includes the pair below which has been missed.

```
              \ /
   {non(X,Y)^(1), any(X,Y)^(1)}
       {non(0,0), any(0,0)}
```

Note that pair
({non(X,Y)$^{(2)}$,any(X,Y)$^{(2)}$}, {non(0,1),any(0,1)}),
is not in $T^2_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$, because,
({non(X,Y)$^{(2)}$}, {non(0,Y)$^{(2)}$})
is not in $T_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$, although
({non(0,Y)$^{(2)}$, any(0,Y)$^{(2)}$}, {non(0,1),any(0,1)})
is in $T_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$. Note also that, although
({non(X,Y)$^{(1)}$}, {non(0,Y)$^{(1)}$})
is in $T_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$,
({non(0,Y)$^{(1)}$, any(0,Y)$^{(2)}$}, {non(0,1),any(0,1)})
is not in $T_{compose}(SS(P_{non}) \cup \{(\Box,\Box)\})$.

Remark. As before, it is not difficult to modify the definition of $T_{compose}$ above so as to respect the distinction between success pairs and suspension pairs.

(2) Replacement with Equivalence

Definition Equivalent Goals, Atoms and Predicates of General GHC Programs

Let P be a general GHC program. Two goals Γ and Δ are said to be *equivalent in P* when
(a) the set of variables occurring in Γ and that in Δ are identical, and
(b) there exists a one-to-one correspondence between the set of the goals with superscript occurring in $D(\Gamma)$ and that in $D(\Delta)$ such that
(b1) $\Gamma\sigma^{(a)}$ correspond to $\Delta\sigma^{(a')}$ for any substitution σ,
(b2) $(\Gamma\sigma^{(a)}, \Gamma\sigma\tau^{(b)})$ is in $D(\Gamma)$ if and only if $(\Delta\sigma^{(a')}, \Delta\sigma\tau^{(b')})$ is in $D(\Delta)$ for any substitutions σ and τ, where $\Gamma\sigma^{(a)}$ corresponds to $\Delta\sigma^{(a')}$ and $\Gamma\sigma\tau^{(b)}$ corresponds to $\Delta\sigma\tau^{(b')}$, and
(b3) $(\Gamma\sigma_1^{(a_1)}, \Gamma\sigma_1\tau_1^{(b_1)}) \preceq (\Gamma\sigma_2^{(a_2)}, \Gamma\sigma_2\tau_2^{(b_2)})$ in $D(\Gamma)$ if and only if $(\Delta\sigma_1^{(a'_1)}, \Delta\sigma_1\tau_1^{(b'_1)}) \preceq (\Delta\sigma_2^{(a'_2)}, \Delta\sigma_2\tau_2^{(b'_2)})$ in $D(\Delta)$, where $\Gamma\sigma_l^{(a_l)}$ corresponds to $\Delta\sigma_l^{(a'_l)}$ and $\Gamma\sigma_l\tau_l^{(b_l)}$ corresponds to $\Delta\sigma_l\tau_l^{(b'_l)}$ for $l = 1, 2$.

Two atoms "A" and "B" are said to be *equivalent in P* when two singleton goals $\{A\}$ and $\{B\}$ are equivalent in P. Two n-ary predicates "p" and "q" are said to be *equivalent in P* when two atoms $p(X_1, X_2, \ldots, X_n)$ and $q(X_1, X_2, \ldots, X_n)$ are equivalent in P, where $X_1, X_2, \ldots, X_n$ are distinct variables.

Theorem 5.3.2 Let P be any GHC program. Then, for any two atoms A and B equivalent in P, goals $\Gamma \cup \{A\}$ and $\Gamma \cup \{B\}$ are equivalent in P for any goal Γ.

Proof. It suffices to show that

(a) $(\Pi\sigma^{(a)}\cup\{A\sigma\}^{(c)}, \Pi\sigma\tau^{(b)}\cup\{A\sigma\tau\}^{(d)})$ is in $\overline{SS}(P)$ if and only if $(\Pi\sigma^{(a)}\cup\{B\sigma\}^{(c')}, \Pi\sigma\tau^{(b)}\cup\{B\sigma\tau\}^{(d')})$ is in $\overline{SS}(P)$ for any Π and σ, τ, where $\{A\sigma\}^{(c)}$ corresponds to $\{B\sigma\}^{(c')}$ and $\{A\sigma\tau\}^{(d)}$ corresponds to $\{B\sigma\tau\}^{(d')}$, and

(b) $(\Pi\sigma_1^{(a_1)}\cup\{A\sigma_1\}^{(c_1)}, \Pi\sigma_1\tau_1^{(b_1)}\cup\{A\sigma_1\tau_1\}^{(d_1)}) \preceq (\Pi\sigma_2^{(a_2)}\cup\{A\sigma_2\}^{(c_2)}, \Pi\sigma_2\tau_2^{(b_2)}\cup\{A\sigma_2\tau_2\}^{(d_2)})$ in $\overline{SS}(P)$ if and only if $(\Pi\sigma_1^{(a_1)} \cup \{B\sigma_1\}^{(c'_1)}, \Pi\sigma_1\tau_1^{(b_1)} \cup \{B\sigma_1\tau_1\}^{(d'_1)}) \preceq (\Pi\sigma_2^{(a_2)} \cup \{B\sigma_2\}^{(c'_2)}, \Pi\sigma_2\tau_2^{(b_2)}\cup\{B\sigma_2\tau_2\}^{(d'_2)})$ in $\overline{SS}(P)$ for any Π and $\sigma_1, \sigma_2, \tau_1, \tau_2$, where $\{A\sigma_l\}^{(c_l)}$ corresponds to $\{B\sigma_l\}^{(c'_l)}$ and $\{A\sigma_l\tau_l\}^{(d_l)}$ corresponds to $\{B\sigma_l\tau_l\}^{(d'_l)}$ for $l = 1, 2$.

Then from Theorem 5.3.1, it suffices to show that

(a) $(\Pi\sigma^{(a)} \cup \{A\sigma\}^{(c)}, \Pi\sigma\tau^{(b)} \cup \{A\sigma\tau\}^{(d)})$ is in $T^n_{compose}(SS(P) \cup \{(\Box, \Box)\})$ if and only if $(\Pi\sigma^{(a)} \cup \{B\sigma\}^{(c')}, \Pi\sigma\tau^{(b)} \cup \{B\sigma\tau\}^{(d')})$ is in $T^n_{compose}(SS(P) \cup \{(\Box, \Box)\})$ for any Π, σ, τ and n, where $\{A\sigma\}^{(c)}$ corresponds to $\{B\sigma\}^{(c')}$ and $\{A\sigma\tau\}^{(d)}$ corresponds to $\{B\sigma\tau\}^{(d')}$, and

(b) $(\Pi\sigma_1^{(a_1)}\cup\{A\sigma_1\}^{(c_1)}, \Pi\sigma_1\tau_1^{(b_1)}\cup\{A\sigma_1\tau_1\}^{(d_1)}) \preceq (\Pi\sigma_2^{(a_2)}\cup\{A\sigma_2\}^{(c_2)}, \Pi\sigma_2\tau_2^{(b_2)}\cup\{A\sigma_2\tau_2\}^{(d_2)})$ in $T^n_{compose}$ $(SS(P)\cup\{(\Box, \Box)\})$ if and only if $(\Pi\sigma_1^{(a_1)}\cup\{B\sigma_1\}^{(c'_1)}, \Pi\sigma_1\tau_1^{(b_1)}\cup\{B\sigma_1\tau_1\}^{(d'_1)})$ $\preceq (\Pi\sigma_2^{(a_2)} \cup \{B\sigma_2\}^{(c'_2)}, \Pi\sigma_2\tau_2^{(b_2)} \cup \{B\sigma_2\tau_2\}^{(d'_2)})$ in $T^n_{compose}(SS(P) \cup \{(\Box, \Box)\})$ for any $\Pi, \sigma_1, \sigma_2, \tau_1, \tau_2$ and n, where $\{A\sigma_l\}^{(c_l)}$ corresponds to $\{B\sigma_l\}^{(c'_l)}$ and $\{A\sigma_l\tau_l\}^{(d_l)}$ corresponds to $\{B\sigma_l\tau_l\}^{(d'_l)}$ for $l = 1, 2$.

This is proved by induction on n using the definition of transformation $T_{compose}$.

To summarize, for general GHC programs,

(a) the denotations of composed goals can be computed from the denotations of composing atoms, and

(b) if two atoms A and B are equivalent, then any two goals obtained by adding A or B to any common goal are also equivalent.

Hence, the partially ordered success-suspension multiset $SS(P)$ (or something containing the equivalent information) is one of the qualified candidates of the semantics of general GHC programs.

6. Comparison with the Semantics of Prolog Programs

So far, we have focused our attention on the semantics of GHC programs. It is an interesting problem whether there exists some connection between the semantics of GHC programs and that of Prolog programs. This section first shows how we compare the semantics of them, then introduces three formulations of the semantics of Prolog programs, and last shows that these semantics enjoy the same compositional character and the same semantic property as GHC programs.

6.1 Prolog and GHC

As was mentioned in Section 1, the second role of the commitment operation has been ignored in this paper. Hence, as far as our semantics is concerned, it is natural to compare the semantics of GHC programs with that of Prolog programs after the following conversion.

Definition GHC Program Corresponding to Prolog Program

Let C be a Prolog clause of the form

$p(t_1, t_2, \ldots, t_m)$:- $B_1, B_2, \ldots, B_n$.

Then the GHC clause C_{GHC} of the form

$p(X_1, X_2, \ldots, X_m)$:- $\mid X_1 = t_1, X_2 = t_2, \ldots, X_m = t_m, B_1, B_2, \ldots, B_n$

is called the *GHC clause corresponding to* Prolog clause C, where $X_1, X_2, \ldots, X_m$ are distinct new variables.

Let P be a Prolog program. Then the GHC program P_{GHC} obtained from P by converting each Prolog clause in P to the corresponding GHC clause is called the *GHC program corresponding to* Prolog program P.

First, note that $suspension(P_{GHC})$ is empty so that $SS(P_{GHC}) = success(P_{GHC})$, because the guard of each GHC clause in P_{GHC} never renders any computaion suspended. Hence, the partially ordered success-suspension multiset semantics is reduced to the success multiset semantics for P_{GHC}, and the definition of $T_{compose}$ is reduced as below:

$$\begin{aligned} T_{compose}(I) = \{\ & (\Gamma^{(a)} \cup \Delta^{(d)}, \Gamma\sigma\tau^{(c)} \cup \Delta\sigma\tau^{(f)}) \mid \\ & (\Gamma^{(a)}, \Gamma\sigma^{(b)}) \text{ and } (\Delta\sigma^{(e)}, \Delta\sigma\tau^{(f)}) \text{ are in } I\ \}. \end{aligned}$$

Second, note that we may ignore the distinction (using superscripts) between seemingly identical success goals as well as the partial ordering relation between success goals, because the consideration on intermediate suspension forests is no longer necessary. Hence the partially ordered success multiset semantics is redneed to the success set semantics for P_{GHC}.

6.2 Semantics of Prolog Programs

(1) Atom-Answer Set Semantics

Definition Atom-Answer Set Semantics of Prolog Programs

A *program* is a set of definite clauses. A *labelled tree* is a finite tree whose nodes are labelled with expressions of the form "$A = B$", where A and B are unifiable atoms. Substitutions $\sigma_1, \sigma_2, \ldots, \sigma_n$ are said to be *unifiable* when there exists a substitution σ such that, for each σ_i, there exists a substitnion τ_i satisfying $\sigma = \sigma_i\tau_i$. A substitution τ is called the *most general unifier of* $\sigma_1, \sigma_2, \ldots, \sigma_n$ when τ is the most general substitution among such substitutions.

Let P be a program, T be a labelled tree and $T_1, T_2, \ldots, T_n$ be its immediate subtrees. The labelled tree T is called a *proof tree of* atom A *with answer substitution* σ *by* P when there exists a clause C in P of the form

B :- $B_1, B_2, \ldots, B_n$

such that

(a) A and B are unifiable, say by an m.g.u. θ,
(b) the root node of T is labelled with "$A = B$",
(c) $T_1, T_2, \ldots, T_n$ are proof trees of $B_1, B_2, \ldots, B_n$ with answer substitutions $\sigma_1, \sigma_2, \ldots, \sigma_n$ by P respectively, and
(d) σ is the restriction of an m.g.u. of $\theta, \sigma_1, \sigma_2, \ldots, \sigma_n$ to the variables occurring in A.

The clause C is said to be *used at the root node*.

Let $T_1, T_2, \ldots, T_k$ be proof trees of atoms $A_1, A_2, \ldots, A_k$ with answer substitutions $\sigma_1, \sigma_2, \ldots, \sigma_k$. A multiset $F = \{T_1, T_2, \ldots, T_k\}$ is called a *proof forest* of atom multiset $\{A_1, A_2, \ldots, A_k\}$ with answer substitution σ when σ is an m.g.u. of $\sigma_1, \sigma_2, \ldots, \sigma_k$.

The set of all the pairs $(\Gamma, \Gamma\sigma)$ such that there exists a proof forest of Γ with answer substitution σ by P is called the success set for composed goals of P, and denoted by $\overline{success_1}(P)$. The set of all the pairs $(A, A\sigma)$ such that there exists a proof tree of A with answer substitntion σ by P is called the success set of P, and denoted by $success_1(P)$.

The denotation of goal Γ is the set of all the pairs $(\Pi, \Pi\sigma)$ in $\overline{success_1}(P)$ such that Π is an instance of Γ. The denotation of atom "A" is the set of all the pairs $(B, B\sigma)$ in $success_1(P)$ such that B is an instance of A. The denotation of an n-ary predicate "p" is the denotation of atom $p(X_1, X_2, \ldots, X_n)$, where $X_1, X_2, \ldots, X_n$ are distinct variables. The denotation of program P is $success_1(P)$.

Example 6.2.1 Let P_1, P_2 and P_3 be the following three Prolog programs:

```
P1 : p(a).
     p0(a).
P2 : p(X).
     p0(a).
P3 : p(X).
     p(a).
     p0(a).
```

Then $success_1(P_1)$ includes

(p(X), p(a)),

$success_1(P_2)$ includes

(p(X), p(X)),

and $success_1(P_3)$ includes both of

(p(X), p(X)),
(p(X), p(a)).

It is obvious that this semantics directly corresponds to the success-suspension set semantics for converted GHC programs.

(2) Extended Herbrand Model Semantics

Definition Extended Herbrand Model Semantics of Prolog Programs

A *program* is a set of definite clauses. A *labelled tree* is a finite tree whose nodes are labelled with atoms.

Let P be a program, T be a labelled tree and $T_1, T_2, \ldots, T_n$ be its immediate subtrees. The labelled tree T is called a *proof tree of* atom A *by* P when there exists an instance of a clause C in P of the form

A :- $A_1, A_2, \ldots, A_n$

such that $T_1, T_2, \ldots, T_n$ are proof trees of $A_1, A_2, \ldots, A_n$, respectively. The clause C is said to be *used at the root node*.

Let $T_1, T_2, \ldots, T_k$ be proof trees of atoms $A_1, A_2, \ldots, A_k$. A multiset $F = \{T_1, T_2, \ldots, T_k\}$ is called a *proof forest* of atom multiset $\{A_1, A_2, \ldots, A_k\}$.

The set of all the goals Γ such that there exists a proof forest of Γ by P is called the success set for composed goals of P, and denoted by $\overline{success_2}(P)$. The set of all the atoms A such that there exists a proof tree of A by P is called the success set of P, and denoted by $success_2(P)$.

The denotation of goal Γ is the set of all the goals Π in $success_2(P)$ such that Π is an instance of Γ. The denotation of atom "A" is the set of all the atoms B in $success_2(P)$ such that B is an instance of A. The denotation of an n-ary predicate "p" is the denotation of atom $p(X_1, X_2, \ldots, X_n)$, where $X_1, X_2, \ldots, X_n$ are distinct variables. The denotation of program P is $success_2(P)$.

Example 6.2.2 Let P_1, P_2 and P_3 be the three Prolog programs as before. Then $success_2(P_1)$ includes

$p(a)$,

while $success_2(P_2)$ and $success_2(P_3)$ includes

$p(X), p(a)$.

Note that $success_2$ does not make a distinction between P_2 and P_3.

It is obvious that, if $success_1(P) = success_1(Q)$ for two Prolog programs P and Q, then $success_2(P) = success_2(Q)$, since

$success_2(P) = \{A\sigma \mid (A, A\sigma) \in success_1(P)\}$,

$success_2(Q) = \{B\tau \mid (B, B\tau) \in success_1(Q)\}$,

but not vice versa.

(3) Herbrand Model Semantics

Definition Herbrand Model Semantics of Prolog Programs

A *program* is a set of definite clauses. A *labelled tree* is a finite tree whose nodes are labelled with ground atoms.

Let P be a program, T be a labelled tree and $T_1, T_2, \ldots, T_n$ be its immediate subtrees. The labelled tree T is called a *proof tree of* ground atom A *by* P when there exists a ground instance of a clause C in P of the form

A :- $A_1, A_2, \ldots, A_n$

such that $T_1, T_2, \ldots, T_n$ are proof trees of $A_1, A_2, \ldots, A_n$, respectively. The clause C is said to be *used at the root node*.

Let $T_1, T_2, \ldots, T_k$ be proof trees of ground atoms $A_1, A_2, \ldots, A_k$. A multiset $F = \{T_1, T_2, \ldots, T_k\}$ is called a *proof forest* of ground atom multiset $\{A_1, A_2, \ldots, A_k\}$.

The set of all the ground goals Γ such that there exists a proof forest of Γ by P is called the success set for composed goals of P, and denoted by $\overline{success_3}(P)$. The set of all the ground atoms A such that there exists a proof tree of A by P is called the success set of P, and denoted by $success_3(P)$.

The denotation of goal Γ is the set of all the ground goals Π in $\overline{success_3}(P)$ such that Π is an instance of Γ. The denotation of atom "A" is the set of all the ground atoms B in $success_3(P)$ such that B is a ground instance of A. The denotation of an n-ary predicate "p" is the denotation of atom $p(X_1, X_2, \ldots, X_n)$, where $X_1, X_2, \ldots, X_n$ are distinct variables. The denotation of program P is $success_3(P)$.

Example 6.2.3 Let P_1, P_2 and P_3 be the three Prolog programs as before. Then $success_3(P_1)$, $success_3(P_2)$ and $success_3(P_3)$ include

$p(a)$.

Note that $success_3$ does not make a distinction between P_1, P_2 and P_3.

It is obvious that, if $success_2(P) = success_2(Q)$ for two Prolog programs P and Q, then $success_3(P) = success_3(Q)$, since

$success_3(P) = \{$ ground instance of $A \mid A \in success_2(P)\}$,

$success_3(Q) = \{$ ground instance of $B \mid B \in success_2(Q)\}$,

but not vice versa.

Remark. For each definition of the semantics, we have the corresponding multset semantics by modifying as follows:

(a) A *program* is a set of pairs consisting of clause identifiers and definite clauses. (We assume that no pair has an identical clause identifier so that two definite clauses of the same form are distinguished.)

(b) Each node label of a *labelled tree* has an additional second component C, where C is the clause identifier of a pair in P.
(c) Each node label of a *proof tree* has an additional second component C, which is the clause identifier of the definite clause used at the node.
(d) The multiset of all the atom pairs (resp. atoms, ground atoms) corresponding to different proof trees is called the *success multiset* of P, and denoted by $success_1(P)$ (resp. $success_2(P)$, $success_3(P)$). The success multiset for composed goals and denotaions of atoms and predicates are defined accordingly.

6.3 Goal Composition for Prolog Programs

(1) Computation of the Semantics for Composed Goals

Then, how can the denotations of composed goals be restored from the denotations of composing atoms?

Definition Transformation for Composing Goals of Prolog Programs
Let P be a Prolog program. For the atom-answer set semantics, a set of expressions of the form $(\Gamma, \Gamma\sigma)$ is called a goal interpretation. $T_{compose}$ is the transformation of goal interpretations defined as follows:
$T_{compose}(I) = \{ (\Gamma \cup \Delta, \Gamma\sigma\tau \cup \Delta\sigma\tau) \mid (\Gamma, \Gamma\sigma) \text{ and } (\Delta\sigma, \Delta\sigma\tau) \text{ are in } I \}$.
For the extended Herbrand model semantics, a set of atoms is called a goal interpretation. $T_{compose}$ is the transformation of goal interpretations defined as follows:
$T_{compose}(I) = \{ \Gamma \cup \Delta \mid \Gamma \text{ and } \Delta \text{ are in } I \}$.
For the Herbrand model semantics, a set of ground atoms is called a goal interpretation. $T_{compose}$ is the transformation of goal interpretations defined as follows:
$T_{compose}(I) = \{ \Gamma \cup \Delta \mid \Gamma \text{ and } \Delta \text{ are in } I \}$.

Theorem 6.3.1 Let P be a Prolog program. Then

$$\overline{success}_1(P) = \bigcup_{l=0}^{\infty} T^l_{compose}(success_1(P) \cup \{(\Box, \Box)\}),$$
$$\overline{success}_2(P) = \bigcup_{l=0}^{\infty} T^l_{compose}(success_2(P) \cup \{\Box\}),$$
$$\overline{success}_3(P) = \bigcup_{l=0}^{\infty} T^l_{compose}(success_3(P) \cup \{\Box\}),$$

where atom pair $(A, A\sigma)$ in $success_1(P)$ is identified with goal pair $(\{A\}, \{A\sigma\})$, and atom "A" in $success_2(P)$ or $success_3(P)$ is identified with goal $\{A\}$.

Proof. Obvious.

(2) Replacement with Equivalence

Definition Equivalent Goals, Atoms and Predicates of Prolog Programs
Let P be a Prolog program. For the atom-answer set semantics, two goals Γ and Δ are said to be *equivalent in P* when
(a) the set of variables occurring in Γ and that in Δ are identical, and
(b) $(\Gamma\sigma, \Gamma\sigma\tau)$ is in $D(\Gamma)$ if and only if $(\Delta\sigma, \Delta\sigma\tau)$ is in $D(\Delta)$ for any substitutions σ and τ.
For the extended Herbrand model semantics, two goals Γ and Δ are said to be *equivalent in P* when
(a) the set of variables occurring in Γ and that in Δ are identical, and
(b) $\Gamma\sigma$ is in $D(\Gamma)$ if and only if $\Delta\sigma$ is in $D(\Delta)$ for any substitutions σ.

For the Herbrand model semantics, two goals Γ and Δ are said to be *equivalent in* P when

(a) the set of variables occurring in Γ and that in Δ are identical, and

(b) $\Gamma\sigma$ is in $D(\Gamma)$ if and only if $\Delta\sigma$ is in $D(\Delta)$ for any substitution σ which substitutes ground terms to the variables occurring in Γ and Δ.

Two atoms "A" and "B" are said to be *equivalent in* P when two singleton goals $\{A\}$ and $\{B\}$ are equivalent in P. Two n-ary predicates "p" and "q" are said to be *equivalent in* P when two atoms $p(X_1, X_2, \ldots, X_n)$ and $q(X_1, X_2, \ldots, X_n)$ are equivalent in P, where $X_1, X_2, \ldots, X_n$ are distinct variables.

Theorem 6.3.2 Let P be a Prolog program. Then, for any two atoms A and B equivalent in P, goals $\Gamma \cup \{A\}$ and $\Gamma \cup \{B\}$ are equivalent in P for any goal Γ.

Proof. Obvious from Theorem 6.3.1.

Remark. The semantics of Prolog programs was first established by employing the Herbrand model semantics [6]. More detailed discussions following it, e.g., those on strong completeness of SLD-resolution, naturally considered answer substitutions separately [5],[1],[20]. The extended Herbrand model semantics was investigated in [7]. Following the trends of the semantics, theoretical works on reasoning about Prolog programs have usually employed the Herbrand model semantics. For example, the meaning of "equivalence preservation" of the program transformation originally shown by Tamaki and Sato [37] is that the least Herbrand model of the initial Prolog program and that of the final Prolog program in their transformation sequence are identical. Recently, it was proved that Tamaki-Sato's transformation also preserves equivalence in the sense of the atom-answer set semantics [15], and even in the sense of the atom-answer multiset semantics [14].

7. Discussion

(1) Related Works on the Semantics of Data Flow Networks

A data flow network is a directed graph whose nodes are *operators* (or *processes*) and whose edges are *communication channels* (or *communication lines*). An operator represents either an indivisible primitive operator or a composed defined sub-network. A communication channel represents a one-way data path between operators.

Each operator operates when necessary data are ready at its (not necessary all) input ports, and transmits the computed results to its (not necessary all) output ports after an unspecified amount of time. The input ports of each operator are always ready to receive data, and the (potentially infinite) received data are buffered at the input ports for later use. These operators autonomously operate in parallel, and communicate only by the asynchronous transmission of data through their input and output ports.

Each communication channel links an output port at one node to an input port at the other (or possibly identical) node. They transmit data within an impredictable, but finite amount of time.

A (possibly infinite) sequence of data observed on each edge is called the *history* (or *stream*) of the communication channel. The empty history is denoted by Λ. If a history $\mathbf{X}$ is a prefix (initial subsequence) of history $\mathbf{Y}$, then relation $\mathbf{X} \subseteq \mathbf{Y}$ holds. This relation is a partial ordering relation with its minimal element Λ, and called the *prefix ordering*. Moreover, this partial ordering relation is *(chain) complete*, i.e., any increasing chain $\mathbf{X}_1 \subseteq \mathbf{X}_2 \subseteq \cdots \subseteq \mathbf{X}_n \subseteq \cdots$ has a least upper bound $\lim_{n\to\infty} \mathbf{X}_n$, i.e., a history which is larger than any $\mathbf{X}_n$ and smaller than any such history.

The problem of the semantics of such data flow networks is to characterize the possible behaviors of the entire network in terms of the behaviors of its operators so as to make various reasoning about such data flow networks possible.

Kahn and MacQueen [11] [12] considered a class of data flow networks, called *functional data flow networks*, which satisfy the following two conditions:

functionality:

Each operator with n input ports and m output ports is a *function* f from n-tuples of histories to m-tuples of histories, called the *history function* of the operator,

continuity:

$f(\lim_{n\to\infty} X_n) = \lim_{n\to\infty} f(X_n)$ for each history function f.

Here the latter condition "continuity" immediately implies

monotonicity:

$f(X) \subseteq f(Y)$ if $X \subseteq Y$ for each history function f.

Intuitively, monotonicity means that receiving more input can only provoke the operator to send an output that extends the output provoked by the current input, and continuity means furthermore that each operator never waits for infinite amount of input before it decides to send any finite output. With such a data flow network, Kahn and MacQueen associated a set of mutual recursive equations relating the histories on the communication channels using the history functions, and chracterized the behavior of the entire network by the unique least fixpoint of the set of equations, whose existence is guaranteed by the least fixpoint theorem for continuous functions on complete partially ordered sets. They then utilized Scott's theory for proving inductive properties of such a data flow network as well as for showing that the composed network still has a continuous history function.

Following the success of Kahn and MacQueen's semantics, several attempts had been made for generalizing their approach to nondeterministic data flow networks, whose operators are not necessarily functions, hence, do not always yield the same output when given the same input. The attempts, however, immediately exposed several inherent difficulties, which then leaded to the proposals of several alternative approaches to overcome these difficulties.

Keller [16] discussed the problems of the semantics of nondeterministic data flow networks in detail, in particular, pointed out that, when the output port of a nondeterministic operator (e.g., the *merge* operator) is linked to its input port in a network, the operator does not necessarily shows the same behavior in the entire network as it shows as a single separated component. To solve the problem, he then devised several preliminary technieques for representing the semantics of such data flow networks and proving properties of them.

To solve the same problem, Kosinski [17] presented an idea to associate a tag with each datum in each history for identifying the sequence of arbitrary decisions made by nondeterministic operators (e.g., the "*merge*" operator) which have contributed to the existence of that datum in that history. By considering a set of such *tagged histories* satisfying some causality conditions for the generation of these tagged histories, and by defining a complete partial order on the tagged history set, he characterized the behavior of the entire network using the least fixpoint technieque in the same way as Kahn and MacQueen.

A natural generalization of a *history function* for nondeterministic data flow networks is a *history relation*. Brock and Ackerman [4] found a very clear example, called *Brock-Ackerman's Anomaly*, which demonstrates that a naive generalization of history functions to history relations does not adequately characterize the behaviors of nondeterministic data flow networks. Instead of naive history relations, they proposed an alternative characterization, called *scenarios*, which incorporates the causality relations between data received at input ports and data transmitted from output ports. Two data in the histories are causally ordered if the event of producing one must precede that of producing the other. A scenario is a pairs of input histories and output histories augmented with a partial ordering on data in the histories showing this causality constraint. Brock and Ackerman showed that two data flow networks indistinguishable by the naive history relation semantics are distinguishable by this scenario model semantics, and gave a composition rule for scenarios of data flow networks.

Pratt [30] considered a set of *traces*. A trace in his paper is a partial ordered multisets of events, an event is either production or consumption of a datum at some port, and the partial ordering denotes necessary temporal precedence between the events. He defined a network composition rule based on the notion called *consistent network trace*. This notion, similarly to the least fixpoint operator, works for excluding never occuring computation by ruling out the cycles which a naive composition rule might generate in the partial ordering of events in the composed networks.

Staples and Nguyen [34] considered partially ordered multisets of input/output history relations. In their approach, the behavior of each operator is specified by equations (or term rewriting rules). Two seemingly identical input/output history relations are distinguished when they differ in the choice of the equations on the computation paths. Two input/output histories are partially ordered if the least computation returning the output histories of the first can be extended to the least computation returning the output histories of the second. They then defined an operation for network composition, called **Link**, using three operations **Step**, **Iter** and **Closit**.

Nguyen, Gries and Owicki [29] treated the problem within the framework of Manna-Pnueli style temporal logic [23]. They considered infinite sequences of *observations*. An observation in their paper is a pair consisting of finite sequences of events at all ports of a network up to some point in an execution of the network (called a *trace* in their paper), and port-state information about on which ports the network is ready to communicate at that point (called *communication functions*). Their specification language is that of (linear) predicate temporal logic with port-variables, communication functions and the binary predicate corresponding to the precedence relation between events in the traces. They then gave a proof system, which includes an inference rule for network composition based on the notion "consistent network traces" by Pratt, as well as a proof of its soundness and relative completeness, and demonstrated that two inequivalent assertions are derived for the two networks of Brock-Ackerman's anomaly.

(2) Motivations of Our Approach

Let us show how we have reached the formulation presented in this paper. First, we started reformulating Kahn's results [11] within the framework of GHC. Following van Emden and Kowalski's approach, we first considered pairs (A, σ) consisting of initial goals and their answer substitutions corresponding to *actually* successful computation. We then immediately found that, even if each atom has no actually successful computation, composed goals consisting of such atoms might have successful computation. (Moreover, even if each atom has at most one actually successful computation, composed goals consisting of such atoms might have more than two actually successful computation. See Example A.4 in

Appendix.) Hence, we decided to consider not only *actually* successful computation but also *potentially* successful computation.

Next, however, we encountered a difficulty in computing the success-suspension set for composed goals from the success-suspension sets for atoms. Because we had then employed the notion of maximal committed forest without using the clause part of each label, some path through which goals can succeed if appropriate instantiations are given from outside were missed.

There were two alternative ways to overcome this difficulty. One is to consider not just the changes to *maximal* committed forests but all the changes to any committed forests. However, this would lead us to considering the very details of computation almost step by step. The other way is to define suspension forests as maximal committed subextensions just as we did.

Then, we found an example of non-monotonic GHC program for which the success-suspension set for composed goals could not be computed exactly from the success-suspension sets for atoms by the naive goal composition method presented in Section 4.3. This time, unsuccessful computation could be included in the computed success-suspension set for composed goals.

Again, there were two alternative ways. One is to consider not the pairs of initial goals and answer substitutions, but the trios of initial goals, unsolved goals and answer substitutions. The unsolved goals can show how far the execution has proceeded and with what goals the execution should resume. However, the unsolved goals would expose the internal details, i.e., what goals are called how from the upper-level goal. This characteristics is undesirable for the *abstractness* of the semantics we would like. The other way is to introduce some *structure* into success-suspension multisets, and assume some condition for the structure so that the naive goal composition method of Section 4.3 works for the simple case. The condition is exactly the *monotonicity* we have defined in Section 4.1. (Then, we noticed that the key notion for the success-suspension set semantics is not functionality but monotonicity. This is the reason the reformulation of Kahn's results about data flow networks is in Appendix.)

Last, we tried to generalize the goal composition method for general GHC programs without assuming monotonicity. Defining the goal composition method, while taking the partial ordering structure into consideration, was the last step. After several hand-simulations, we found that the alternative representation $(A, A\sigma)$ is more convinient for explaining the transformation for goal composition, since the relation between generated pairs is more easily captured using this representation.

Our approach is very closely related to Staples and Nguyen's approach. The main differences are

(a) their approach considers the least computation returning given output histories, while our approach considers the maximal computation for given input instantiations, and
(b) their approach considers all partial computation, while our approach consideres only maximal partial computation, which makes the denotations of deterministic predicates (or operators) simpler.

Very roughly speaking, the **Step** operation in their paper corresponds to the inverse of $T_{compose}$ in our paper, the **Iter** operation to that of $T^{l}_{compose}$, and the **Closit** operation to that of $\bigcup_{l=0}^{\infty} T^{l}_{compose}$. (Cf. the top-down goal composition in Appendix (3).)

Our approach is also related to Brock and Ackerman's approach. Let $(A\theta', A\theta'\tau')$, $(A\theta, A\theta\tau)$ be pairs in $SS(P)$ such that

$(A\theta', A\theta'\tau') \preceq (A\theta, A\theta\tau)$

and let θ'' be the substitution such that $\theta'\theta''$ is θ. (For simplicity, we have omitted the superscripts of success-suspension atoms.) Then there exists an m.g.u. of $\theta'\tau'$ and θ (see Section 6.2 (1)), say σ, and a substitution τ'' such that τ is $\sigma\tau''$. This means that the additional input instantiation θ'' has caused the additional output substitution τ''. By considering all such θ' and τ', we can decompose substitution θ into $\theta_1\theta_2\cdots\theta_n$, substitution τ into $\tau_1\tau_2\cdots\tau_m$, and define a causality ordering between $\theta_1, \theta_2, \ldots, \theta_n$ and $\tau_1, \tau_2, \ldots, \tau_m$. This is an alternative formulation of the idea behind Takeuchi's approach [36], which is an adaptation of the scenario set model by Brock and Ackerman to GHC.

(3) Other Works on the Semantics of Parallel Logic Programs

There are several attempts for formalizing the semantics of parallel logic programs, either operational or declarative.

Operational semantics of GHC was first examined by Ueda [39]. He focused his attention on the effects of the truly parallel execution. He discussed, in particular, the problems of distributed unification, and by assuming an appropriate distributed unification mechanism, explained the "rule of commitments" and the "rule of suspension" using some layered nested box structures.

Beckman [2] discussed the semantics of parallel logic programs based on the CCS (Calculus of Communicating Systems) by Milner [24],[25] by translating parallel logic programs into CCS-type transition systems. Saraswat [31],[32] also studied the semantics of (some class of) Concurrent Prolog and GHC programs based on Plotkin-style transition systems.

Takeuchi [36] has been investigating the semantics of GHC by adapting the "scenario" model of Brock and Ackerman [4] to GHC. Shibayama [33] has been studying a *compositional* semantics of GHC. Murakami [27] has been constructing a Hoare-style axiomatic system for GHC so as to utilize it for verification of GHC programs [26].

The declarative semantics has not yet been fully studied. Within the framework of a constraint committed-choice language, Maher [22] investigated when the committed-choice language can be said to have *logical* semantics in the same sense as Prolog, and gave some sufficient condition for it, *data sufficiency* (no suspension due to insufficient instantiation of arguments) and *determinacy* (no two guards satisfying an identical call).

Levi and Palamidessi [18] have studied the declarative semantics of synchronization in particular, by introducing functor annotation "producing" and "consuming" into the term structures. Levi [19] specialized and simplified the above approach for the semantics of *flat* GHC programs by considering the set of trios consisting of atoms with distinct variables arguments, their argument form restriction (input guard equations) and their instantiation substitutions (output equations), and charcterized the set as a least fixpoint of some transformation. Murakami [28] has extended it for flat GHC programs with perpetual processes.

Several attempts for reasoning about GHC programs, e.g., verification [13],[26],[27], transformation [8],[9],[10],[42], and debugging [21],[35], have been made as well.

8. Conclusions

A preliminary formalization of the semantics of Guarded Horn Clauses (GHC) has been presented particularly emphasizing on a compositional character. Further refinement and logical formalization of the semantics are left for future.

Acknowledgements

This research was done as a part of the Fifth Generation Computer Systems project of Japan. We would like to thank Dr. K. Fuchi (Director of ICOT) for the opportunity of doing this research, and Dr. K. Furukawa (Deputy Director of ICOT), Dr. R. Hasegawa (Chief of ICOT 1st Laboratory) and Dr. H. Ito (Chief of ICOT 3rd Laboratory) for their advice and encouragement.

References

[1] Apt, K.R. and M.H.van Emden, "Contribution to the Theory of Logic Programming," J. ACM, Vol.29, No.3, pp.841-862, 1982.

[2] Beckman, L., "Towards a Formal Semantics for Concurrent Logic Programming Languages," Proc. of 3rd International Conference on Logic Programming, pp.335–349, London, July 1986.

[3] Bowen, D.L, L.Byrd, F.C.N.Pereira, L.M.Pereira and D.H.D.Warren, "DECsystem-10 Prolog User's Manual," Department of Artificial Intelligence, University of Edinburgh, 1983.

[4] Brock, J.D. and W.B.Ackerman, "Scenario : A Model of Nondeterminate Computation," in *Formalization of Programming Concepts* (J.Diaz and I.Ramos Eds.), Lecture Notes in Computer Science 107, pp.252–259, Springer-Verlag, 1981.

[5] Clark, K.L., "Predicate Logic as a Computational Formalism," Research Monograph : 79/59, TOC, Imperial College, 1979.

[6] van Emden, M. and R.Kowalski, "The Semantics of Predicate Logic as a Programming Language," J. of ACM, Vol.23, No.4, pp.733–742, 1976.

[7] Falaschi, M., G.Levi, M.Martelli and C.Palamidessi, "A More General Declarative Semantics for Logic Programming Languages," to appear, 1988.

[8] Fujita, H., A.Okumura and K.Furukawa, "Partial Evaluation of GHC Programs Based on UR-set with Constraint Solving," Proc. of Logic Programming '88, Seattle, August 1988.

[9] Furukawa, K. and K.Ueda, "GHC Process Fusion by Program Transformation," Proc. of 2nd Conference of Japan Society for Software Science and Technology, pp.89–92, Tokyo, November 1985.

[10] Furukawa, K., A.Okumura and M.Murakami, "Unfolding Rules for GHC Programs," Proc. of 2nd France-Japan Artificial Intelligence and Computer Science Symposium, Cannes, November 1987. Also Proc. of 4th Conference of Japan Society for Software Science and Technology, pp.267–270, Kyoto, November 1987.

[11] Kahn, G., "The Semantics of A Simple Language for Parallel Programming," in Information Processing 74 (J.L.Rosenfeld Ed.), pp.471–475, North Holland, 1974.

[12] Kahn, G. and D.B.MacQueen, "Coroutines and Networks of Parallel Processes," in Information Processing 77 (B.Gilchrist Ed.), pp.993–998, North Holland, 1977.

[13] Kameyama, Y., "Axiomatic System for Concurrent Logic Programming Languages," Master's Thesis of The University of Tokyo, 1987. Also an extended abstract was presented at U.S.-Japan Workshop on Logic of Programs, Honolulu, May 1987.

[14] Kanamori, T. and T.Kawamura, "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation (II)," ICOT Technical Report TR-403, ICOT, Tokyo, June 1988.

[15] Kawamura, T. and T.Kanamori, "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation," ICOT Technical Report TR-399, ICOT, Tokyo, June

1988. Also to appear Proc. of International Conference on Fifth Generation Computer Systems 1988, Tokyo, November 1988.

[16] Keller, R.M., "Denotational Models for Parallel Programs with Indeterminate Operators," in *Formal Description of Programming Concepts* (E.J.Neuhold Ed.), pp.337–366, North-Holland, 1977.

[17] Kosinski, P.R., "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs," in Conf. Rec. of 5th ACM Symposium on Principles of Programming Languages, pp.214–221, Tucson, January 1978.

[18] Levi, G. and C.Palamidessi, "An Approach to the Declarative Semantics of Synchronization in Logic Language," Proc. of 4th International Conference on Logic Programming, pp.877–893, Melbourne, May 1987.

[19] Levi, G., "A New Declarative Semantics of Flat Guarded Horn Clauses," to appear, January 1988.

[20] Lloyd, J.W., "Foundations of Logic Programming," Springer-Verlag, 1984.

[21] Lloyd, J.W. and A.Takeuchi, "A Framework of Debugging GHC Programs," ICOT Technical Report TR-186, ICOT, Tokyo, 1986.

[22] Maher, M.J., "Logic Semantics for A Class of Committed-choice Programs," Proc. of 4th International Conference on Logic Programming, pp.858–876, Melbourne, May 1987.

[23] Manna, Z. and A.Pnueli, "Verification of Concurrent Programs: The Temporal Framework," in *The Correctness Problem in Computer Science* (R.S.Boyer and J S Moore Eds.), pp.215-273, Academic Press, 1981.

[24] Milner, R., "A Calculus of Communicating Systems," Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[25] Milner, R., "Lectures on a Calculus for Communicating Systems," in *Control Flow and Data Flow : Concepts of Distributed Programming* (M.Broy Ed.), pp.205–228, Springer-Verlag, 1985.

[26] Murakami, M., "Proving Partial Correctness of Guarded Horn Clauses," Proc. of The Logic Programming Conference '87, pp.117–124, Tokyo, June 1987. Also to appear in *Logic Programming '87* (E.Wada Ed.), 1988.

[27] Murakami, M., "On an Axiomatic Semantics of GHC Programs," (in Japanese) Proc. of 4th Conference of Japan Society for Software Science and Technology, pp.259–262, Kyoto, November 1987.

[28] Murakami, M., "A Declarative Semantics of Parallel Logic Programs with Perpetual Processes," to appear Proc. of International Conference on Fifth Generation Computer Systems 1988, Tokyo, November 1988.

[29] Nguyen,V., D.Gries and S.Owicki, "A Model and Temporal Proof Systems for Networks of Processes," Conf. Rec. of the 12th Annual ACM Symposium on Principles of Programming Languages, pp.121–131, New Orleans, January 1985.

[30] Pratt, V., "On the Composition of Processes," Conf. Rec. of the 9th Annual ACM Symposium on Principles of Programming Languages, pp.213–223, Albuquerque, January 1982.

[31] Saraswat, V.A., "Partial Correctness Semantics for CP(!,−)," Proc. of the Foundation of Software Technology and Theoretical Computer Sciences Conference, Lecture Notes in Computer Science 206, pp.347–368, Springer-Verlag, 1985.

[32] Saraswat, V.A., GHC : Operational Semantics, Problems and Relationship with CP(↓ ,|)," Proc. of 1987 Symposium on Logic Programming, pp.347–358, San Francisco, August 1987.

[33] Shibayama, E., "A Compositional Semantics of GHC," Proc. of 4th Conference of Japan Society for Software Science and Technology, pp.255–258, Kyoto, November 1987.

[34] Staples, J. and V.L.Nguyen, "A Fixpoint Semantics of Nondeterministic Data Flow," J. of ACM, Vol.32, No.2, pp.411–444, 1985.

[35] Takeuchi, A., "Algorithmic Debugging of GHC Programs," ICOT Technical Report TR-185, ICOT, Tokyo, 1986.

[36] Takeuchi, A., "Towards A Semantic Model of GHC," Technical Report of "Foundation of Software" Research Group, Information Processing Society of Japan, 1986.

[37] Tamaki, H. and T.Sato, "Unfold/Fold Transformation of Logic Programs", Proc. of 2nd International Logic Programming Conference, pp.127–138, Uppsala, July 1984.

[38] Ueda, K., "Guarded Horn Clauses," ICOT Technical Report TR-103, ICOT, Tokyo, 1985. Also in Proc. of The Logic Programming Conference '85, pp.225–236, 1985. Also in *Logic Programming '85* (E.Wada Ed.), Lecture Note in Computer Science 221, pp.168–179, Springer-Verlag, 1986. Also in *Concurrent Prolog: Collected Papers* (E.Y.Shapiro Ed.), Vol. 1, Chap. 4, The MIT Press, 1987.

[39] Ueda, K., "On the Operational Semantics of Guarded Horn Clauses," Presented at RIMS Simposium on Mathematical Methods in Software Science and Engineering '85, Kyoto, October 1985. Also RIMS Research Report 586, pp.263–283, Research Institute for Mathematical Sciences, Kyoto University, March 1986.

[40] Ueda, K., "Guarded Horn Clauses," Doctorial Thesis, Information Engineering Course, Faculty of Engineering, University of Tokyo, 1986.

[41] Ueda, K., "Guarded Horn Clauses : A Parallel Logic Programming Language with the Concept of a Guard," Proc. of 1st France-Japan Artificial Intelligence and Computer Science Symposium, pp.127–138, Tokyo, October 1987. Also in *Programming of Future Generation Computers* (M.Nivat and K.Fuchi Eds.), North-Holland, 1988.

[42] Ueda, K. and K.Furukawa, "Transformation Rules for GHC Programs," to appear as ICOT Technical Report, ICOT, Tokyo, 1988. Also to appear Proc. of International Conference on Fifth Generation Computer Systems 1988, Tokyo, November 1988.

Appendix. Semantics of Functional GHC Programs

This appendix shows a reformulation of Kahn's results about data flow networks [11],[12] within the framework of parallel logic programming for finite computation based on our success-suspension set semantics.

(1) Functionality of GHC Programs

Definition Functionality of Goals, Atoms and Predicates

Let P be a GHC program. A goal Γ is said to be *functional in P* when, for any instance Π of Γ, there exist no two success-suspension forests of Π with different answer substitutions. An atom "A" is said to be *functional in P* when singleton goal $\{A\}$ is functional in P. An n-ary predicate "p" is said to be *functional in P* when atom $p(X_1, X_2, \ldots, X_n)$ is functional in P, where $X_1, X_2, \ldots, X_n$ are distinct variables.

Definition Functionality of GHC Programs

A GHC program P is said to be *functional* when any predicate is functional in P.

Note that, when a goal Γ is functional in P, there might exist two different proof forests with an identical answer substitutions, hence, $SS(P)$ is regarded as a *set* rather than a *multiset* when functionality is considered.

Example A.1 Consider P_{join}. The predicates "*join*," "$\leq$" and "$>$" are functional.

Example A.2 Consider P_{echo}. The predicates "*wait0*," "*wait1*" and "*echo-back*" are functional, while "*shout-wait*" is not functional, since

(shout-wait(X,Y), shout-wait(0,Y)),
(shout-wait(X,Y), shout-wait(1,Y))

are both in $suspension(P_{echo})$.

Example A.3 Consider P_{loop}. The predicate "*loop*" is functional.

(2) Functionality for Composed Goals

The definition of functionality does not immediately guarantee that the functionality holds for composed goals.

Theorem A.1 If a GHC program P is functional, then any goal Γ is functional in P.

Proof. Suppose that, for some goal Γ, there exist proof forests $\mathcal{F}_1$ and $\mathcal{F}_2$ with answer substitution σ_1 and σ_2, respectively. We will show that σ_1 and σ_2 are identical (modulo renaming substitution) by induction on the well-founded ordering $\ll$ of answer substitutions. (See the proof of Theorem 4.1.)
Base Case: When σ_1 and σ_2 are $<>$, they are the same answer substitutions, hence the theorem trivially holds.
Induction Step: Otherwise, from Lemma 3.2, there must exist an atom A in Γ such that the maximal committed subextension of $\{A\}$ in $\mathcal{F}_1$ or $\mathcal{F}_2$, say in $\mathcal{F}_1$, has a non-renaming answer substitution σ_A. From the functionality of P, the maximal committed subextension of $\{A\}$ in $\mathcal{F}_2$ also has answer substitution σ_A. Let σ_1' be the substitution such that $\sigma_A\sigma_1'$ is σ_1, and σ_2' be the substitution such that $\sigma_A\sigma_2'$ is σ_2. Then, $\mathcal{F}_1$ and $\mathcal{F}_2$ are success-suspension forests of $\Gamma\sigma_A$ with answer substitution σ_1' and σ_2', respectively. From the induction hypothesis for σ_1' and σ_2', substitutions σ_1' and σ_2' must be identical modulo renaming substitution, hence σ_1 and σ_2 are also identical modulo renaming substitution.

Example A.4 One might expect that the definition of functionality can be weakened as follows:

"A goal Γ is said to be *functional in* P when, for any instance Π of Γ, there exist no two success forests with different answer substitutions."

With this definition, however, the theorem above does not hold. For example, consider the following GHC program:

```
shout-wait(X,Y) :- | X=0, wait0(Y).
shout-wait(X,Y) :- | X=1, wait1(Y).
echo-back(0,Y) :- | Y=0.
echo-back(1,Y) :- | Y=1.
wait0(0).
wait1(1).
```

Although $shout\text{-}wait(X,Y)$ and $echo\text{-}back(X,Y)$ are functional if this definition is adopted, a query

```
?- shout-wait(X,Y), echo-back(X,Y)
```

can return two answer substitutions

$<X \Leftarrow 0, Y \Leftarrow 0>$,
$<X \Leftarrow 1, Y \Leftarrow 1>$.

Similarly, one might expect that the definition of functionality can be weakened as follows:

"A goal Γ is said to be *functional in P* when, for any instance Π of Γ, there exist no two success forests or no two suspension forests with different answer substitutions."

Again, this definition does not work. For example, consider the following GHC program:

```
shout-wait(X,Y) :- | X=0, wait0(Y).
shout-wait(X,Y) :- | X=1, Y=1.
echo-back(0,Y) :- | Y=0.
echo-back(1,Y) :- | Y=1.
wait0(0).
```

Although *shout-wait*(X, Y) and *echo-back*(X, Y) are functional if this definition is adopted, a query

```
?- shout-wait(X,Y), echo-back(X,Y)
```

can return two answer substitutions

$< X \Leftarrow 0, Y \Leftarrow 0 >$,

$< X \Leftarrow 1, Y \Leftarrow 1 >$.

(3) Success-Suspension Set Semantics of Functional GHC Programs

Due to the following theorem, the success-suspension set semantics in Section 4 is valid for functional GHC programs.

Theorem A.2 If a GHC program P is functional, then P is monotonic.

Proof. Let A be an atom such that

(a) there exists a success-suspension forest $\mathcal{F}$ of $\{A\}$, say with answer substitution σ, and

(b) there exists a success-suspension forest $\mathcal{G}$ of $\{A\sigma\}$, say with answer substitution τ.

Let $\mathcal{G}'$ be the maximal committed subextension of $\{A\}$ in $\mathcal{G}$. Then $\mathcal{G}'$ is a success-suspension forest of $\{A\}$ with answer substitution σ, since P is functional. Hence τ is $<>$ and $\mathcal{G}'$ is identical to $\mathcal{G}$ itself, which means that P is monotonic.

In particular, the denotation of each predicate contains no two pairs with identical first elements. Hence, not only the bottom-up goal composition method in Section 4.3 but also the following nondeterministic *top-down* goal composition method, which corresponds to Kahn's method for functional data flow networks [11], works for functional GHC programs, where argument Γ is a goal some instance of which can succeed.

```
compose(Γ:goal): pair-of-goals;
  $\Gamma_0 := \Gamma$;  $i := 0$;
  while there exists $A \in \Gamma_i$ such that $(A, A\sigma) \in SS(P)$ and $A \neq A\sigma$ do
    $\Gamma_{i+1} := \Gamma_i\sigma$;   $i := i + 1$;
  endwhile
  return $(\Gamma, \Gamma_i)$
```