ICOT Technical Report: TR-550

TR-550

GHC Program Diagnosis Using Atom Behavior

by
M. Ueno & T. Kanamori (Mitsubishi)

April, 1990

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

GHC Program Diagnosis Using Atom Behavior

Machi UENO Tadashi KANAMORI

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

Abstract

This paper presents a diagnosis algorithm for flat GHC programs. The diagnosis algorithm finds two types of bugs (*wrong clause instance* and *wrong suspension atom*) by comparing an aspect of actual computation with that of the intended one. Roughly speaking, the aspect abstracted and compared here, called an *atom behavior*, is a set of pairs $(A\sigma, A\tau)$ such that atom $A\sigma$ can be instantiated to atom $A\tau$ when $A\sigma$ is executed *as far as possible* along a given course of computation. What behavior of GHC programs is incorrect and what bug is responsible for the incorrect behavior are explained based on this notion. Then, the diagnosis algorithm for GHC programs is presented. Human programmers just need to answer "Yes/No" to the queries which the diagnosis system asks them while tracing the *abstracted* computation process in a top-down manner. The power and the implementation of the algorithm are discussed as well.

Keywords : Program Diagnosis, Program Debugging, GHC.

Contents

1. Introduction

One of the reasons logic programming attracts one's attention is that it shows the possibilities of more sophisticated programming environment in future. The "algorithmic debugger" by Shapiro [20],[21] has been widely known as a successful example for the diagnosis of (pure) Prolog programs. While tracing (or backtracing) the execution of Prolog programs, his debugger asks the programmer whether each procedure (predicate) call just traced has returned a correct result or what answer should have been returned if the program is the intended one. The programmer just needs to answer the queries according to his/her declarative knowledge about each procedure, i.e., he/she does not need to follow the operational behavior of the execution in his/her mind. Moreover, due to the automatic database capability for recording the previous answers and utilizing them for answering the same queries, he/she needs to answer much less queries. Shapiro's approach has been further extended in various directions by various researchers so far [2],[3],[4],[5],[8],[13],[15],[16],[17],[18]. But, what have made his approach work so well?

One reason is the declarative character of the semantics of (pure) Prolog programs. Although the style of his debugger seems drastically novel at first glance, it is just automatically providing the locations at which we need to check the results so that it is not *completely* discontinuous with the debugging which the programmers usually do by following the execution traces. However, owing to the declarative character, whether the computation result is correct or not can be checked at each location in the execution trace independently of the other locations. It is also the declarative character that enables us to utilize the previous answers in the database independent of the locations in the execution traces.

The other reason, which is taken for granted when the diagnosis of Prolog programs is considered, is that the Prolog execution is search complete due to the backtracking mechanism as far as the execution terminates. When the execution of an atom is intended to succeed, but the program at hand is wrong, the execution of the atom using the program actually does fail (as far as the execution terminates), hence we can have a wrong execution trace for the diagnosis.

Simultaneously as Prolog has been widely accepted, more ambitious attempts have been made for designing Prolog-like languages suitable for parallel execution. Guarded Horn Clauses (GHC), as well as Concurrent Prolog and Parlog, is a programming language originated from such attempts [27],[28]. As for GHC, two remarkable features brought by the notion of guard distinguish GHC from Prolog.

One remarkable feature of GHC is the suspension (and synchronization) mechanism in the guards by prohibiting the instantiation of the variables appearing in caller goals. Because of this mechanism, the execution of a goal consisting of several atoms proceeds by importing and exporting the instantiation information, so that some goal can succeed when other goals co-exist, even if it never succeeds as a single goal. In general, some goal behaves in a quite different manner according to the behavior of co-existing goals, hence, if only the final form of the execution as a single goal is considered, the execution result of the goal is not necessarily synthesized from the execution results of the individual atoms in the goal.

The other remarkable feature of GHC is the committed-choice mechanism by throwing away alternative courses at passing the guards. Because of this mechanism, once we have found a clause to which the execution of a goal is committed, there occurs no backtracking. So, even if there exist several solutions to a goal, only one of them is obtained as its solution. Moreover, even the same initial goal with the same final form might succeed, fail or be suspended depending on to which clause the execution is committed.

As for (so called) concurrent logic programming languages Concurrent Prolog, Parlog and GHC, several attempts have just begun to apply Shapiro's approach to those languages [6],[10],[11],[12],[14],[19],[23],[24],[25],[26]. The two remarkable features mentioned above, however, make the attempts for GHC not necessarily easy.

The first feature of GHC makes it more difficult to understand GHC computation by *directly* tracing the execution process than to understand Prolog computation using the "trace" command of DEC10 Prolog, since we usually cannot focus our attention on local intermediate goals without considering the effects of other goals for GHC programs, while we can do it for (pure) Prolog programs. It is not easy to check the computation result at each location independently of the other locations. And worse, even if a usual trace of the whole execution process is given, understanding GHC computation from the execution trace is not easy. (This problem leads us to a notion, called an *atom behavior*, in Section 3.)

Example 1.1 Let P_1 be a GHC program consisting of the following clauses:

```
C1: t-and-c(O,S) :- | tailor(O,S), customer(O,S).
C2: tailor(O,S) :- | tailor(1,O,S).
C3: tailor(N,[order|Onext],S) :- N<3 |
        S=[suit|Snext], tailor(N+1,Onext,Snext).
C4: tailor(N,O,S) :- N=3 | S=[ ].
C5: tailor(N,[ ],S) :- | S=[ ].
C6: customer([order|O],[suit|S]) :- | customer(O,S).
C7: customer(O,[ ]).
```

Here, "1", "2", "3" and "$N+1$" are abbreviations for "$suc(zero)$", "$suc(suc(zero))$", "$suc(suc(suc(zero)))$" and "$suc(N)$", respectively. The predicate "*t-and-c*" denotes the world consisting of a tailor and a customer (clause C_1). The tailor makes suits in response to orders. He is a proud artisan so that he never tailors the same suits more than 2 (clauses C_3, C_4), and he stops his work when the order is stopped (clause C_5). The customer receives each tailored suit confirming the correspondence between the orders and the suits (clause C_6), and he gives up receiving the suits when the tailor stops his work (clause C_7).

Let P_2 be a GHC program consisting of the same clauses as P_1 except that C_3 is replaced with the following C_3':

```
C3': tailor(N,O,S) :- N<3 |
        O=[order|Onext], S=[suit|Snext], tailor(N+1,Onext,Snext).
```

The tailor defined using C_3' won't wait until the next order comes, and he tailors 2 suits expecting that there are 2 orders unless he notices that the order is stopped. (Note that, if the commitment operator "|" is considered just the logical conjunction "$\wedge$," those two programs P_1 and P_2 are logically equivalent.) Consider a query

```
?- t-and-c([order|O1],S).
```

which denotes the state that one suit is ordered (and other suits may be ordered). The execution of this goal in P_1 is suspended with an answer

```
t-and-c([order|O1],[suit|S1])
```

while the execution of the same goal in P_2 succeeds with an answer

```
t-and-c([order,order|O2],[suit,suit]).
```

Suppose that P_1 is the intended program. Then the success in P_2 is incorrect. It is due to the success of the atom $tailor(2, O1, S1)$ in P_2 which should have been suspended until the variable "$O1$" is instantiated, say, by substitution $< O1 \Leftarrow [order|O2] >$.

The debugging methods for (so called) concurrent logic programming languages in [6],[12],[23],[24],[25] do not detect any bug for program P_2, because those methods consider only the final form of goals, and the execution of the goal in the final form

```
?- t-and-c([order,order|02],[suit,suit])
```

with answer

```
t-and-c([order,order|02],[suit,suit])
```

conforms to our intention. We, however, would like to detect an instance of clause C_3' as the wrong clause instance containing a bug.

The second feature of GHC makes it impossible to guarantee that some course of computation can happen or cannot happen by (repeatedly) executing a GHC goal, since only one specific course of computation is considered when the GHC program is executed, while all courses of computation can be enumerated when a Prolog program is executed. (This problem leads us to a formalization of the correctness of GHC programs in Section 4.)

Example 1.2 Let Q_1 be the (meaningless) GHC program consisting of the following clauses:

```
C1:  ss(X).
C2:  ss(X) :- | always-suspend(X).
```

Then, the execution of goal

```
?- ss(X).
```

in Q_1 either succeeds or is suspended depending on to which clause the execution is committed.

Let Q_2 be a GHC program consisting of the same clauses as Q_1 except that C_1 is missed. Then, the execution of the same goal in Q_2 is always suspended.

Now suppose that Q_1 is the intended program. Though the execution of $ss(X)$ never succeeds in Q_2, the suspension in Q_2 itself does not contradict our intention. Moreover, however repeatedly the execution of the goal may be tried, we cannot judge from the superficial behavior whether Q_2 is wrong so that the intended success never occurs, or the execution has chosen a wrong path accidentally so that it has not succeeded.

This paper presents a diagnosis algorithm for flat GHC programs. The diagnosis algorithm finds two types of bugs (*wrong clause instance* and *wrong suspension atom*) by comparing an aspect of actual computation with that of the intended one. Roughly speaking, the aspect abstracted and compared here, called an *atom behavior*, is a set of pairs $(A\sigma, A\tau)$ such that atom $A\sigma$ can be instantiated to atom $A\tau$ when $A\sigma$ is executed *as far as possible* along a given course of computation. What behavior of GHC programs is incorrect and what bug is responsible for the incorrect behavior are explained based on this notion. Then, the diagnosis algorithm for GHC programs is presented. Human programmers just need to answer "Yes/No" to the queries which the diagnosis system asks them while tracing the *abstracted* computation process in a top-down manner. The power and the implementation of the algorithm are discussed as well.

The rest of this paper is organized as follows: Section 2 explains flat GHC programs and their execution. Section 3 introduces the notion of atom behaviors to represent (some abstracted aspect of) the execution in flat GHC programs. Section 4 defines incorrect flat GHC programs and two types of bugs in incorrect flat GHC programs. Then Section 5 presents a diagnosis algorithm for flat GHC programs. Last, Section 6 and Section 7 discusses the power and the implementation of the algorithm.

2. Flat GHC

This section explains GHC mostly following the explanation of Ueda [28]. 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 [1].

(1) Program

A *clause* is an expression of the form:

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

where H, G_i's and B_j's are atoms ($1 \leq i \leq m, 1 \leq j \leq n$). H is called a *clause head*, the G_i's are called *guard atoms*, and the B_j's are called *body atoms*. The symbol "|" is called a *commitment operator*. The part of a clause before "|" is called a *guard*, and the part after "|" is called a *body*. (When $m = n = 0$, ":-" and "|" are omitted. Note that the clause head is included in the guard.)

One primitive (infix) binary predicate "=" for unifying two terms is predefined by the language. Other primitive predicates are predefined using a (possibly infinite) set of clauses such that

- each clause is of the form "$H :- |B_1, B_2, \ldots, B_n$" ($n \geq 0$),
- H is not unifiable with the head of any other clause in the set, and
- each body atom B_j is an equation of the form "$s_j = t_j$" ($1 \leq j \leq n$).

The primitive predicates used in practice are not excluded by this condition. Atoms with primitive predicates are called *primitive atoms*.

A clause is called a *flat clause* when each guard atom G_i is a primitive atom ($1 \leq i \leq m$). A *program* is a finite set of flat clauses.

Example 2.1 Let P_1 be the set of the following flat clauses:

```
C1: t-and-c(0,S) :- | tailor(0,S), customer(0,S).
C2: tailor(0,S) :- | tailor(1,0,S).
C3: tailor(N,[order|Onext],S) :- N<3 |
        S=[suit|Snext], tailor(N+1,Onext,Snext).
C4: tailor(N,0,S) :- N=3 | S=[ ].
C5: tailor(N,[ ],S) :- | S=[ ].
C6: customer([order|Onext],[suit|Snext]) :- | customer(Onext,Snext).
C7: customer(0,[ ]).
```

Here, "<" is a primitive predicate. Then, P_1 is a program.

(2) Goal

A *goal* is an expression of the form:

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

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

Example 2.2 The following are GHC goals.

```
?- t-and-c([order|01],S).
?- tailor([order|01],S), customer([order|01],S).
```

(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 an atom G (i.e., unification of G with the head of C and any unification invoked by solving the guard atoms of C) cannot instantiate the atom G.

(b) Unification invoked directly or indirectly in the body of a clause C called by an atom 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 an atom 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 goal

```
?- t-and-c([order|O1],S)
```

in P_1 is suspended with answer

```
t-and-c([order|O1],[suit|S1]).
```

(4) Success, Failure and Suspension

Let A be an atom and C be a clause called by A. When the guard of C is solved with answer substitution, say θ, for the variables appearing in the guard of C without instantiating A, then the execution of A is said to *succeed in the guard of clause C with substitution θ*. Otherwise, the execution of A is said to *be suspended in the guard of clause C*. (The latter case includes two cases. One is the case when the unification invoked, either directly or indirectly, in the guard of C instantiates the atom A. The other is the case when the guard cannot be solved even if the instantiation of A is permitted. Though it might look unnatural, we will not make a distinction between them hereafter.)

Note that, due to the restriction on the primitive predicates, the execution of atom A *can succeed* in the guard of a clause C with substitution θ by committing to appropriate clauses in the guard, if and only if the execution of atom A *does succeed* in the guard of C with θ by committing to any committable clauses in the guard. Similarly, the execution of goal A *can be suspended* in the guard of clause C, if and only if the execution of goal A *is suspended* in the guard of clause C.

An atom A is said to *succeed immediately in program P* when

- A is an equation for two unifiable terms, or
- there exists a clause C with no body atoms in P such that the execution of A succeeds in the guard of C.

An atom A is said to *be suspended immediately in program* P when the execution of A is suspended in the guard of any clause in P. An atom A is said to *fail immediately in program* P when A is an equation for two non-unifiable terms.

Notice the difference between the suspension in the guard of a specific clause and the immediate suspension in a program. Note also that the execution of non-primitive atoms can be still non-deterministic, though the execution of primitive atoms is deterministic.

Example 2.4 Let A be an atom of the form

$tailor(1, [order|O1], S)$.

Then, the execution of A succeeds in the guard of clause C_3, while the execution of A is suspended in the guards of clauses C_4 and C_5.

Let B be an atom of the form

$tailor(2, O1, S1)$.

The execution of B is suspended in program P_1.

Remark. The execution of a GHC program may continue infinitely in some cases. In this paper, we will focus our attention on the cases in which the execution terminates finitely.

3. Atom Behavior

In this section, a notion, called an *atom behavior*, is introduced for representing an abstracted aspect of how the form of an atom have been changed (i.e., instantiated) during the execution of the atom until it succeeds, fails or is suspended with its final form. The actual computation process of the atom is represented by a tree, called a *computation tree*, based on the non-deterministic sequential execution. (Note that, though we will use the non-deterministic sequential execution to formalize the necessary notions, most of the final notions formalized are independent of the sequentiality. See Section 8 for the details.) A more abstracted aspect of the computation process is represented by a tree, called a *behavior tree*.

The following sections assume familiarity with the basic terminology of first order logic, such as term, atom (atomic formula), formula and so on. Syntactic variables are X, Y, Z for variables; s, t for terms; C for clauses, possibly with primes and subscripts. "$\equiv$" is used to denote the syntactical identity of two expressions.

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

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

where $X_1, X_2, \ldots, X_l$ are distinct variables. A substitution is called a *renaming substitution* when it assigns a distinct variable to each variable. Substitutions are denoted by $\sigma, \tau, \mu, \nu, \theta, \eta$, and the empty substitution is denoted by $<>$.

An *atom* is defined as usual. Atoms are denoted by A, B, possibly with primes and subscripts. Two atoms are considered identical when they are identical up to renaming of the variables in the atoms.

An atom A is said to be *less instantiated than or equal to* an atom B, and denoted by $A \leq B$, when there exists a substitution θ such that $A\theta$ is identical to B. An atom A is said to be *less instantiated than* an atom B, and denoted by $A < B$, when $A \leq B$ and $B \not\leq A$.

3.1 Atom Behavior in General

Suppose that a given atom $A\mu$ is executed together with other atoms. Then, the variables in the atom might be instantiated by the execution of the atom itself (and the atom exports the instantiation to the other atoms). Or, the variables in the atom might be instantiated by the execution of the other atoms (and the atom imports the instantiation from the other atoms). Suppose that the initial atom has succeeded or been suspended with the form $A\nu$ possibly after the interactions with the other atoms, where we assume $A\mu < A\nu$. Then, let $A\sigma$ be an atom such that $A\mu \leq A\sigma < A\nu$, and let us execute the atom $A\sigma$ as *far as possible* separately from the other atoms along the same course as the execution of $A\mu$ mentioned first. Then, since $A\sigma$ cannot import the instantiation from the other atoms, it would reach an atom $A\tau$ such that $A\tau \leq A\nu$, and stop there. Then, the pair $(A\sigma, A\tau)$ denotes an interval the atom can pass autonomously. Let us collect all such intervals $(A\sigma, A\tau)$ such that $A\sigma < A\tau$ and there exists no other such interval subsuming it. Then, the set will represent some structure of the execution of $A\mu$ mentioned first. (Recall Example 1.1 in Section 1 for the reason such a set is considered.)

Definition Atom Pair

A pair of atoms $(A\sigma, A\tau)$ is called an *atom pair* when $A\sigma < A\tau$. Two atom pairs are considered identical when they are identical up to renaming of the variables appearing in the pairs.

Definition Atom Behavior

A finite set $\mathcal{B}$ of atom pairs is called an *atom behavior of* $A\nu$ when

- for any atom pair $(A\sigma, A\tau)$ in $\mathcal{B}$, there holds $A\tau \leq A\nu$,
- for any two atom pairs $(A\sigma_1, A\tau_1)$ and $(A\sigma_2, A\tau_2)$ in $\mathcal{B}$, if there exists $A\theta$ such that $A\sigma_1 \leq A\theta \leq A\tau_1$ and $A\sigma_2 \leq A\theta \leq A\tau_2$, then $A\tau_1$ and $A\tau_2$ are identical, and neither $A\sigma_1 < A\sigma_2$ nor $A\sigma_2 < A\sigma_1$, and
- $\mathcal{B}$ is marked "success," "failure" or "suspension."

An atom behavior is called a *success atom behavior* (resp. *failure atom behavior*, *suspension atom behavior*) when it is marked "success" (resp. "failure," "suspension"). Atom behaviors are denoted by $\mathcal{B}_{A\nu}$, possibly with primes and subscripts, when it is necessary to explicitly show $A\nu$, and simply by $\mathcal{B}$ when $A\nu$ is obvious from the context.

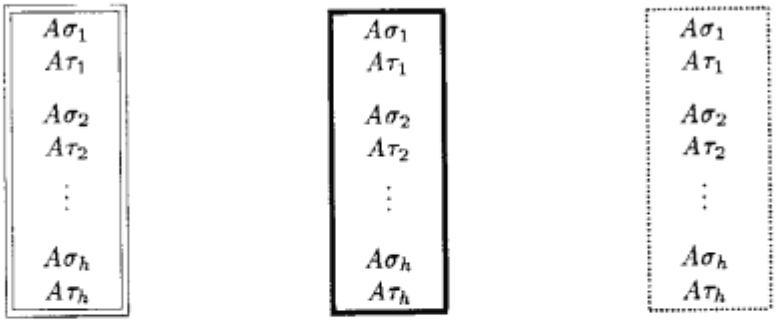
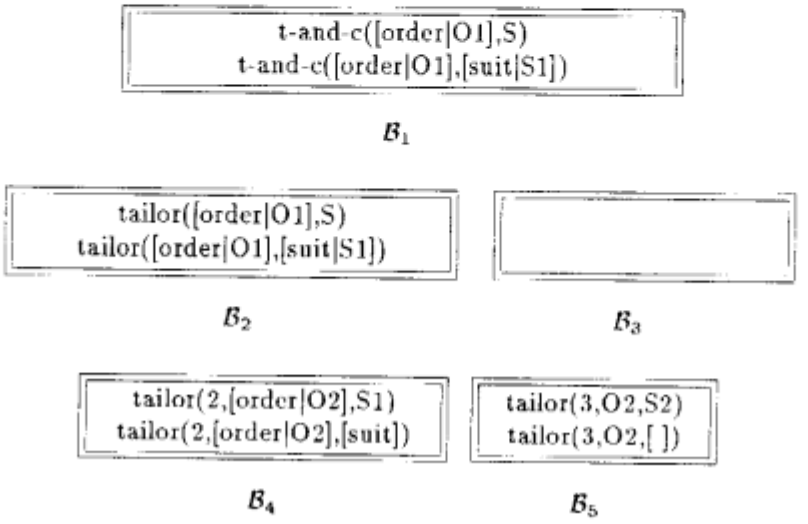


Figure 3.1 Success, Failure and Suspension Atom Behaviors

Though an atom behavior is, in general, a partially ordered set, most of the atom behaviors in this paper are totally ordered. (The precise definition of the partial ordering is irrelevant to the following discussion so that we will omit it.) Hereafter, success atom

behaviors are depicted by arranging their component pairs lengthways (upside down w.r.t. the instantiation ordering, hence the lower a pair is located, the more its second atom is instantiated), and by surrounding them with doubled lines. Similarly, failure atom behaviors are surrounded with thick lines, and suspension atom behaviors with dotted lines. For example, they are depicted as in Figure 3.1 above.

Example 3.1 $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4, \mathcal{B}_5$ below are success atom behaviors.



Though the atom behaviors above consist of zero or one pair, atom behaviors in general may consist of more pairs with more complicated partial ordering.

3.2 Computation Tree

In this section, we will introduce the notion of *computation tree*. (Cf. [7].)

(1) Labelled Tree

A computation tree is a special labelled tree. Hereafter, we will assume that

- each clause in program P is assigned a distinct clause identifier C_i $(i > 0)$,
- a unit clause "$X = X$" (not in program P) is assigned a clause identifier C_0, and
- a special clause identifier $C_?$ is prepared.

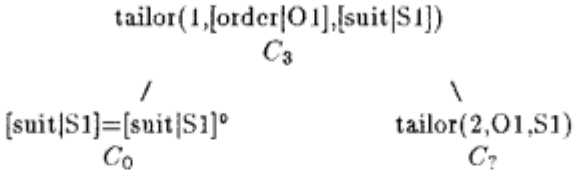
Definition Labelled Tree

A tree T is called a *labelled tree* when

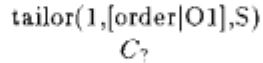
- the nodes of T are labelled with pairs of an atom and a clause identifier, and
- the terminal nodes of T are marked "success," "failure" or unmarked.

The atom part of the root node label of T is called the *root atom of* T. Labelled trees are denoted by T, possibly with primes and subscripts. Two labelled trees are considered identical when they are identical up to renaming of the variables appearing in the labels.

Example 3.2.1 T_2 below is a labelled tree. The superscript "$\circ$" denotes the "success" mark. (The failure marks are denoted by superscripts "$\bullet$.")



T_0 below, consisting of only one root node, is also a labelled tree.



(2) Extension of Labelled Trees

By modelling the non-deterministic sequential GHC execution, the extension of labelled trees is defined as follows:

Definition Immediate Extension of Labelled Trees

Let T and T' be labelled trees. T' is called an *immediate extension of* T in program P when T' is obtained from T by the following operation:

Case 1 : When there exist an unmarked terminal node v in T labelled with $(s = t, C_?)$,

- replace the label of the node with $(s = t, C_0)$.

When s and t are unifiable, say by m.g.u. θ,

- modify the label (A', C') of each node in T to $(A'\theta, C')$, and
- mark the node v "success."

In this case, T' is called an immediate extension of T *with substitution* θ. When s and t are not unifiable,

- mark the node v "failure."

In this case, T' is called an immediate extension of T *with substitution* <>.

Case 2 : When there exist an unmarked terminal node v in T labelled with $(A, C_?)$ and a flat clause C in P of the form

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

such that the execution of A succeeds in its guard with substitution θ, then let η be an instantiation of H to A, and

- replace the label of the node v with (A, C),
- add n child nodes of v labelled with $(B_1\eta\theta, C_?), (B_2\eta\theta, C_?), \ldots, (B_n\eta\theta, C_?)$ to v, and
- if $n = 0$, mark the node v "success."

In this case, T' is called an immediate extension of T *with substitution* θ.

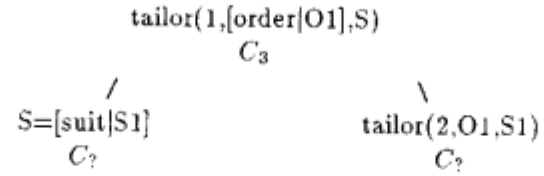
Definition Extension of Labelled Tree

Let T and T' be labelled trees. Then, T' is called an *extension* of T with substitution σ in program P when there exist labelled trees $T_0, T_1, \ldots, T_k$ $(k \geq 0)$ such that

- T_0 is T,
- T_i is an immediate extension of T_{i-1} with substitution θ_i in P for $i = 1, 2 \ldots, k$,
- T_k is T', and
- σ is $\theta_1\theta_2 \cdots \theta_k$.

In particular, T' is called a *proper extension* of T when $k > 0$.

Example 3.2.2 Labelled tree T_1 below is an immediate extension of the labelled tree T_0 of Example 3.2.1 in the flat GHC program P_1.



The labelled tree T_2 of Example 3.2.1 is an immediate extension of T_1 in P_1, so that both T_1 and T_2 are extensions of T_0 in P_1.

(3) Maximal Labelled Tree

Extending a labelled tree as far as possible corresponds to applying GHC execution as far as possible.

Definition Maximal Labelled Tree
A labelled tree T is called a *maximal labelled tree* in program P when there exists no proper extension of T in P.

Example 3.2.3 The labelled tree T_2 of Example 3.2.1 is a maximal labelled tree.

(4) Computation Tree

A computation tree models the GHC execution applied to an atom.

Definition Initial Tree
A labelled tree is called the *initial tree* of atom A when it consists of a single unmarked node labelled with $(A,C_?)$.

Definition Computation Tree
A labelled tree is called a *computation tree* of atom A with solution $A\theta$ in program P when it is an extension of the initial tree of A in P with substitution θ.

Definition Uncommitted Node and Committed Node
Let T be a computation tree, and v be a node in T. Then, v is called an *uncommitted node* when the clause part of the node label is $C_?$, and called a *committed node* otherwise.

Example 3.2.4 The labelled tree T_0 of Example 3.2.1 is an initial tree, so that T_1 and T_2 are computation trees. The right child node of T_2 is an uncommitted node, while the root and the left child nodes are committed nodes.

(5) Clause Instance Used at a Committed Node

Definition Most General Atom for Commitment
Let C be a flat clause of the form

$$H \text{ :- } G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n,$$

$A\nu$ be an atom such that the execution of $A\nu$ succeeds in the guard of C, and $A\mu$ be an atom such that $A\mu \leq A\nu$. Then, atom $H\eta$ is called the *most general atom for commitment from $A\mu$ to $A\nu$* when

(a) $A\mu \leq H\eta \leq A\nu$,
(b) $H\eta$ succeeds in the guard of C in the same way as $A\nu$ (i.e., using the same clauses for primitive predicates), and

(c) there exists no other $H\eta'$ satisfying (a), (b) and more general than $H\eta$.

If $H\eta$ succeeds in the guard of C with substitution θ, the flat clause instance

$$H\eta\theta \text{ :- } G_1\eta\theta, G_2\eta\theta, \ldots, G_m\eta\theta \mid B_1\eta\theta, B_2\eta\theta, \ldots, B_n\eta\theta$$

is called the *most general clause instance for commitment from $A\mu$ to $A\nu$*.

Definition Clause Instance Used at a Committed Node

Let T be a computation tree with root label $(A\nu, C)$, $A\mu$ be an atom such that $A\mu \leq A\nu$, v be a committed node in T, and $C\eta\theta$ be the most general clause instance for commitment from $A\mu$ to $A\nu$, where C is of the form

$$H \text{ :- } G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n.$$

A clause D is called the *clause instance used at v in T w.r.t. $A\mu$* when

- v is the root node of T, and D is $C\eta\theta$, or
- v is a node in some immediate subtree T_i of T $(1 \leq i \leq n)$, and D is the clause instance used at v in T_i w.r.t. $B_i\eta\theta$.

Let T be a computation tree of $A\mu$, and v be a node in T. Then, the clause instance used at v in T w.r.t. $A\mu$ is called simply the *clause instance used at v in T*.

Note that, due to the restriction on primitive predicates, the most general atom for commitment is unique up to renaming of variables, hence so is the clause instance used at node v, when a maximal computation tree T is given.

Example 3.2.5 The clause instance used at the root node in T_2 is

$$tailor(1, [order|O1], S) \text{ :- } 1 < 3 \mid S = [suit|S1], tailor(2, O1, S1).$$

(6) Success Tree, Failure Tree and Suspension Tree

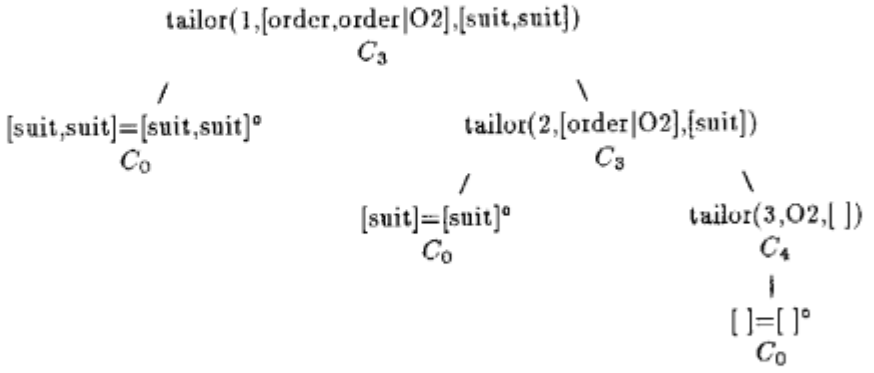
Depending on how the terminal nodes are marked, maximal computation trees are classified into *success tree*, *failure tree* and *suspension tree*.

Definition Success Tree, Failure Tree and Suspension Tree

A maximal computation tree T in program P is called

- a *success tree* in P when all terminal nodes in T are marked "success,"
- a *failure tree* in P when some terminal node in T is marked "failure,"
- a *suspension tree* in P otherwise.

Example 3.2.6 T_2 is a suspension tree in P_1. T_6 below is a success tree in P_1.



(7) Extension Ordering between Computation Trees

The definition of computation tree naturally introduces a partial ordering relation between computation trees. (Intuitively, this ordering means that computation tree T can be extended to T' when additional instantiation θ is applied to all the node labels of T.)

Definition Extension Ordering between Computation Trees

Let T and T' be computation trees in program P. Then, T is said to be *extensible* to T' and denoted by $T \preceq T'$ when there exists a substitution θ for the variables in the atom of the root label of T such that T' is an extension of $T\theta$ in P, where $T\theta$ denotes the tree obtained from T by applying θ to the atom part of every node label.

Example 3.2.7 T_0 and T_2 of Example 3.2.1, and T_6 of Example 3.2.4 are computation trees in P_1, and $T_0 \preceq T_2 \preceq T_6$ holds.

3.3 Behavior Tree

In this section, we will define a more abstracted aspect of the GHC execution (Cf. [9]).

(1) Maximal Subextension

Definition Maximal Subextension

Let T and T' be computation trees in program P such that $T \preceq T'$. Then, a computation tree T'' is called a *maximal subextension* of T in T' when

(a) T'' is an extension of T in P,
(b) T'' is extensible to T' in P, and
(c) there exists no other computation tree satisfying (a), (b) to which T'' is extensible in P.

Example 3.3.1 T_2 is the maximal subextension of T_0 in T_6.

(2) Computed Atom Behavior

Definition Computed Atom Behavior

Let A be an atom of the form $p(X_1, X_2, \ldots, X_n)$ where $X_1, X_2, \ldots, X_n$ are distinct variables, and T be a maximal computation tree in program P with root atom $A\nu$. Then, $(A\sigma, A\tau)$ is called an *atom pair of T in P*, when the following conditions are satisfied:

(a) $A\sigma < A\tau \preceq A\nu$.
(b) Let T_0 be the initial tree of $A\sigma$. Then, there exists a maximal subextension of T_0 in T with solution $A\tau$.
(c) There exists no other pair $(A\sigma', A\tau')$ satisfying (a),(b) and $A\sigma' < A\sigma$, $A\tau \leq A\tau'$.

The set of all atom pairs of T is called the *computed atom behavior of T in P*. A computed atom behavior is called a computed *success* (resp. *failure*, *suspension*) atom behavior when it is a computed atom behavior of a success (resp. failure, suspension) tree.

Example 3.3.2 The set of atom pairs

$\{ (tailor(1, [order|O1], S), tailor(1, [order|O1], [suit|S1]),$
$(tailor(1, [order, order|O2], S), tailor(1, [order, order|O2], [suit, suit]) \}$

is the computed atom behavior of T_6.

(3) Behavior Tree

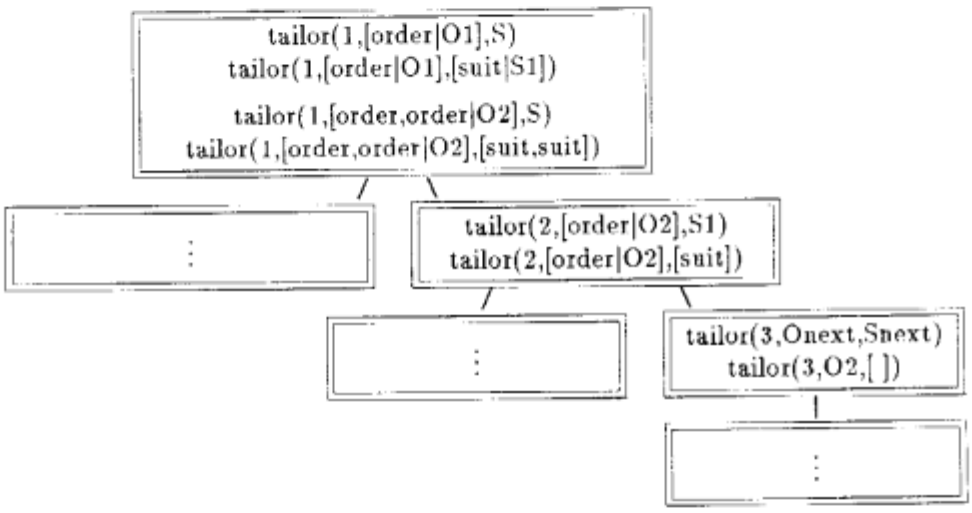
Definition Behavior Tree

Let T be a maximal computation tree in program P, and $T_1, T_2, \ldots, T_n$ be the immediate subtrees of T ($n \geq 0$). A tree $\tilde{T}$ with its immediate subtrees $\tilde{T}_1, \tilde{T}_2, \ldots, \tilde{T}_n$ is called a *behavior tree corresponding to* T when

- the label of the root node of $\tilde{T}$ is the computed atom behavior of T, and
- each $\tilde{T}_i$ is the behavior tree corresponding to T_i ($1 \leq i \leq n$).

If a flat clause D is the clause instance used at the root of T, then D is also said to be *used at the root of* $\tilde{T}$. A tree $\tilde{T}$ is called a *behavior tree* when there exists a maximal computation tree T such that $\tilde{T}$ is a behavior tree corresponding to T. If T has an uncommitted (resp. committed) root, then $\tilde{T}$ is said to *have an uncommitted* (resp. *committed*) *root*. Behavior trees are denoted by $\tilde{T}$, possibly with primes and subscripts.

Example 3.3.3 The tree $\tilde{T}_6$ below is the bahavior tree corresponding to the computation tree T_6 of Example 3.2.4. The details of the atom behaviors of $[suit, suit] = [suit, suit]$, $[suit] = [suit]$ and $[\,] = [\,]$ are omitted.



Then, the clause instance

tailor(1,[order|O1],S) :- 1<3 | S=[suit|S1], tailor(2,O1,S1)

is used at the root node.

A behavior tree is an abstraction obtained from a computation tree by focusing on the maximal autonomous transitions of the form of the root atom. Each transition (i.e., atom pair) in the atom behaviors of child nodes is propagated to their parent node through the clause instance used at the parent node, and all those propagated transitions (after merging overlapped transitions) constitute the transitions in the atom behavior of the parent node. It is easy to see that, when all child atom behaviors are success atom behaviors, so is the parent atom behavior, when some child atom behavior is a failure atom behavior, so is the the parent atom behavior, and otherwise, the parent atom behavior is a suspension atom behavior.

4. Incorrect Flat GHC Programs and their Bugs

What bahvior should the execution of an atom in a GHC program show when the program conforms to the intention in our mind? In this section, we will define what behavior of flat GHC programs is incorrect and what bug is responsible for the incorrect behavior.

4.1 Computed Interpretation and Intended Interpretation

For our diagnosis, a program is characterized by a set of atom behaviors.

Definition Behavior Interpretation

A *behavior interpretation* is a set of atom behaviors, and denoted by I. An atom behavior is said to be *true in* I when it is in I. Otherwise, it is said to be *false in* I.

Definition Computed Behavior Interpretation

A behavior interpretation is called the *computed (behavior) interpretation* of program P, and denoted by $C(P)$, when it consists of all the atom behaviors that is a computed atom behavior of a maximal computation tree in program P.

Definition Intended Behavior Interpretation

A behavior interpretation is called the *intended (behavior) interpretation* of P, and denoted by $M(P)$, when it consists of all the atom behaviors that represents the intended execution behavior for P, i.e., that conforms to the intention in our mind.

Example 4.1.1 Let P_2 be a program consisting of the same flat clauses as P_1 except that C_3 is replaced with C_3' as below:

```
C1:  t-and-c(O,S) :- | tailor(O,S), customer(O,S).
C2:  tailor(O,S) :- | tailor(1,O,S).
C3': tailor(N,O,S) :- N<3 |
        O=[order|Onext], S=[suit|Snext], tailor(N+1,Onext,Snext).
C4:  tailor(N,O,S) :- N=3 | S=[ ].
C5:  tailor(N,[ ],S) :- | S=[ ].
C6:  customer([order|Onext],[suit|Snext]) :- | customer(Onext,Snext).
C7:  customer(O,[ ]).
```

Suppose that P_1 is the intended program for P_2, i.e., $M(P_2) = C(P_1)$. Then, $C(P_2)$ includes a success atom behavior

{ (t-and-c(O,S), t-and-c([order,order|O2],[suit,suit])) },

while $M(P_2)$ does not includes the success atom behavior.

Example 4.1.2 Let P_3 be a program consisting of the same clauses as P_1 except that C_5 is missed as below:

```
C1:  t-and-c(O,S) :- | tailor(O,S), customer(O,S).
C2:  tailor(O,S) :- | tailor(1,O,S).
C3:  tailor(N,[order|Onext],S) :- N<3 |
        S=[suit|Snext], tailor(N+1,Onext,Snext).
C4:  tailor(N,O,S) :- N=3 | S=[ ].
C6:  customer([order|Onext],[suit|Snext]) :- | customer(Onext,Snext).
C7:  customer(O,[ ]).
```

Suppose that P_1 is the intended program for P_3, i.e., $M(P_3) = C(P_1)$. Then, $C(P_3)$ includes a suspension atom behavior

{ (t-and-c([order],S),t-and-c([order],[suit|S1])) }

but $M(P_3)$ does not include the suspension atom behavior.

4.2 Incorrect Flat GHC Programs

We will define the incorrectness of a flat GHC program as follows. (Recall Example 1.2 in Section 1 for the reason "$C(P) = M(P)$" is not used.)

Definition Incorrect Program

Let $C(P)$ be the computed interpretation of program P, and $M(P)$ be the intended interpretation of P in our mind. Then, P is said to be *(partially) correct w.r.t.* $M(P)$ when $C(P) \subseteq M(P)$. Otherwise, P is said to be *(partially) incorrect w.r.t.* $M(P)$.

Example 4.2.1 Suppose that we have written the program P_2 by mistake in place of the intended program P_1. As was mentioned in Section 1, the tailor defined using C_3' won't wait until the next order comes, and he tailors 2 suits expecting that there are 2 orders unless he notices that the order is stopped. Consider a query

```
?- t-and-c([order|O1],S).
```

which denotes the state in which one suit is ordered (and other suits may be ordered). The execution of this goal in P_1 is suspended with an answer

```
t-and-c([order|O1],[suit|S1])
```

while the execution of the same goal in P_2 succeeds with an answer

```
t-and-c([order,order|O2],[suit,suit]).
```

Then the success in P_2 is incorrect. It is due to the success of the atom $tailor(2, O1, S1)$ in P_2 which should have been suspended until the variable "$O1$" is instantiated, say, by substitution $< O1 \Leftarrow [order|O2] >$. In fact, $C(P_2)$ includes the success atom behavior

{(t-and-c(O,S), t-and-c([order,order|O2],[suit,suit]))},

while $M(P_2)$ does not include the success atom behavior. Hence, program P_2 of Example 4.1.1 is incorrect w.r.t. $M(P_2)$.

Example 4.2.2 Suppose that we have written the program P_3 by mistake in place of the intended program P_1. The tailor defined without C_5 won't stop his work even if the order is stopped. Consider a query

```
?- t-and-c([order],S).
```

which denotes the state in which *only* one suit is ordered. The execution of this atom in P_1 succeeds with an answer

```
t-and-c([order],[suit])
```

while the execution of the same atom in P_3 is suspended with an answer

```
t-and-c([order],[suit|S1]).
```

Then, the suspension in P_3 is incorrect. It is due to the suspension of the atom $tailor([\,], S1)$ in P_3 which should have succeeded with instantiation $< S1 \Leftarrow [\,] >$. In fact, $C(P_3)$ includes the suspension atom behavior

{(t-and-c([order],S), t-and-c([order],[suit|S1]))},

while $M(P_3)$ does not include the atom behavior. Hence, program P_3 of Example 4.1.2 is incorrect w.r.t. $M(P_3)$.

In this paper, we will assume that primitive predicates are correct with respect to our intention from the beginning so that the diagnosis of primitive atoms is unnecessary.

4.3 Bugs in Incorrect Flat GHC Programs

When a program is incorrect w.r.t. our intention, what bug is responsible for the incorrect execution? There are two types of bugs in incorrect programs.

(1) Wrong Clause Instance

One is the case when the execution of an atom is committed to some clause C and the body of C succeeds (fails, or is suspended) with some answer which conforms to our intention, but the success (failure, or suspension) of the head atom contradicts our intention.

Definition Wrong Clause Instance

A flat clause C in program P of the form

"H :- $G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n$"

is said to be *wrong w.r.t.* $M(P)$ when there exists a bahavior tree in P such that

- the root node is labelled with $\mathcal{H}$, and $\mathcal{H}$ is not in $M(P)$,
- the child nodes are lebbeled with $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$, and $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ are all in $M(P)$, and
- a clause instance $C\theta$ is used at the root node.

Then the clause instance $C\theta$ is called a *wrong clause instance* in P w.r.t. $M(P)$.

Example 4.3.1 Let P_2 be the program of Example 3.3.1 consisting of the same flat clauses as P_1 except that C_3 is replaced with C_3'. Then, the clause C_3' in P_2 is wrong w.r.t. $M(P_2)$, and the clause instance

tailor(2,O1,S1) :- | S1=[suit|Snext], O1=[order|Onext], tailor(3,Onext,Snext))

is a wrong clause instance in P_2 w.r.t. $M(P_2)$, because the tree below is a part of a behavior tree in program P_2, and

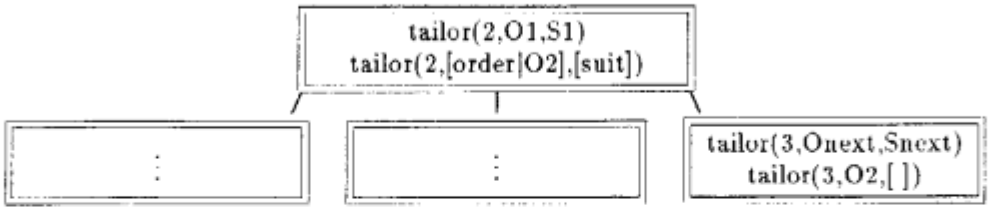
- the success atom bahavior

  { (tailor(2,O1,S1), tailor(2,[order|O2],[suit])) }

  is not in $M(P_2)$,
- the success atom behavior

  { (tailor(3,Onext,Snext), tailor(3,Onext,[ ])) }

  is in $M(P_2)$ (and the atom behaviors of $[suit|S2] = [suit|S2]$) and $[order|O2] = [order|O2]$ are assumed to be true in any $M(P)$ from the beginning, since "=" is a primitive predicate), and
- the clause instance is used at the root node.



(2) Wrong Suspension Atom

The other is the case when the execution of an atom is not committed to any clause in the program, but the suspension of the atom contradicts our intention. Then there must exist a clause missed in the program to which the execution of the atom should be committed.

Definition Wrong Suspension Atom

An atom A is called a *wrong suspension atom in program* P *w.r.t.* $M(P)$ when the execution of A is suspended immediately in P, and the empty suspension atom behavior of A is not in $M(P)$.

Example 4.3.2 Let P_3 be the program of Example 3.3.2 consisting of the same clauses as P_1 except that C_5 is missed. Then, the atom "*tailor*$(2,[\,],S1)$" is a wrong suspension atom in P_3 w.r.t. $M(P_3)$, because the empty suspension atom behavior of *tailor*$(2,[\,],S1)$ is in $C(P_3)$, but not in $M(P_3)$.

(3) Bug Detection Theorem

The following theorem guarantees that, when a GHC program shows an incorrect behavior, either wrong clause instances or wrong suspension atoms are responsible for it.

Theorem 4 *(Bug Detection Theorem)*
Let P be a flat GHC program and $M(P)$ be an intended behavior interpretation. Then, P is (partially) incorrect w.r.t. $M(P)$ if and only if there exists either a wrong clause instance in P w.r.t. $M(P)$ or a wrong suspension atom in P w.r.t. $M(P)$.

Proof. The proof is divided into the "if" part and the "only if" part.
"If" Part: If there exists a wrong clause instance "H :- $\Gamma \mid \Delta$" in P w.r.t. $M(P)$, some $\mathcal{H}$ is in $C(P)$, but not in $M(P)$, hence P is incorrect w.r.t. $M(P)$.
If there exists a wrong suspension atom A w.r.t. $M(P)$, some empty suspension atom behavior is in $C(P)$, but not in $M(P)$, hence P is incorrect w.r.t. $M(P)$.
"Only If" Part: Suppose that there exists an atom behavior in $C(P)$, but not in $M(P)$. Then, consider the bahavior tree T whose root label is the atom behavior. The proof is by induction on the structure of T.
If T is a behavior tree with uncommitted root, then the root node is labelled with an empty suspension atom behavior of some atom A, which is a wrong suspension atom in P w.r.t. $M(P)$.
If T is a bahavior tree with committed root and the labels of the child nodes are all in $M(P)$, then there exists a clause instance in P used at the root node of T, which is a wrong clause instance in P w.r.t. $M(P)$.
If T is a behavior tree with committed root and some label of the child node is not in $M(P)$, then, from the induction hypothesis, there exists either a wrong clause instance in P w.r.t. $M(P)$ or a wrong suspension atom in P w.r.t. $M(P)$.

5. Diagnosis of Flat GHC Programs

In this section, we will show a diagnosis algorithm assuming that a computation tree is given. (The GHC system usually leaves a log-file containing the information about the computation tree.)

5.1 Queries for the Diagnosis

The diagnosis algorithm in this section asks whether an atom behavior is in the intended behavior interpretation or not, and proceeds the diagnosis in response to the answers. We assume that a device, called "*oracle*," always answers correctly to the queries, e.g., a human programmer who knows the diagnosed program well.

To examine whether a success atom behavior

$$\{(A\sigma_1, A\tau_1), (A\sigma_2, A\tau_2), \ldots, (A\sigma_k, A\tau_k)\}$$

of $A\nu$ is in the intended behavior interpretation or not, our diagnosis system needs to ask a question which means

Until the execution succeeds with $A\nu$, the following transitions have occured:

$A\sigma_1$ is instantiated to $A\tau_1$ by itself,
$A\sigma_2$ is instantiated to $A\tau_2$ by itself,
⋮
$A\sigma_k$ is instantiated to $A\tau_k$ by itself.

Does it conform to your intention? :

In the actual terminal sessions, the question is abbreviated for simplicity as follows:

$A\sigma_1 \rightarrow A\tau_1$
$A\sigma_2 \rightarrow A\tau_2$
⋮
$A\sigma_k \rightarrow A\tau_k$
$A\nu$ `is in success. O.K.?` :

The answer to this query is either "yes" or "no." The answer "yes", or simply "y", means that the atom behavior is in the intended behavior interpretation, while the answer "no", or simply "n", means that it is not. As for failure or suspension atom behaviors, the questions are the same except that the last line "$A\nu$ is in success" is replaced with "$A\nu$ is in failure" or "$A\nu$ is in suspension"

Example 5.1 To examine whether the success atom behavior B_4 of Example 3.1 is in $M(P_2)$ or not, our diagnosis system asks as below:

```
tailor(2,01,S1) → tailor(2,[order|02],[suit])
tailor(2,[order|02],[suit]) is in success. O.K.?  :
```

The answer to this query is "no", or simply "n", because this atom behavior is not in $M(P_2)$.

5.2 A Diagnosis Algorithm Using Atom Behavior

The diagnosis algorithm "*diagnose*" receives a computation tree and an atom. When a computation tree T of atom A is given, "*diagnose*(T, A)" returns either a bug or a message "no bug is found."

diagnose(T: computation tree; A: atom): bug-message;
  if the root atom of T is a primitive atom
  then return "no bug is found"
  else let B be the result of "*compute-atom-behavior*" applied to T;
    confirm whether B is true or not (i.e., it is in $M(P)$ or not);
    if it is true
    then return "no bug is found"
    else let $T_1, T_2, \ldots, T_n$ be the immediate subtrees of T (if exist);
      let $B_1, B_2, \ldots, B_n$ be the body of the clause instance used at T's root (if exists);
    apply "*diagnose*" to each T_i and B_i $(1 \leq i \leq n)$;
    if some application returns a bug
    then return it
    else if T is a computation tree with uncommitted root
      then return the atom A as a bug
      else return the clause instance used at the root of T as a bug

Figure 5.2.1 A Diagnosis Algorithm Using Atom Behavior

The auxiliary procedure "*compute-atom-behavior*" (naively) computes the atom behavior of a given computation tree.

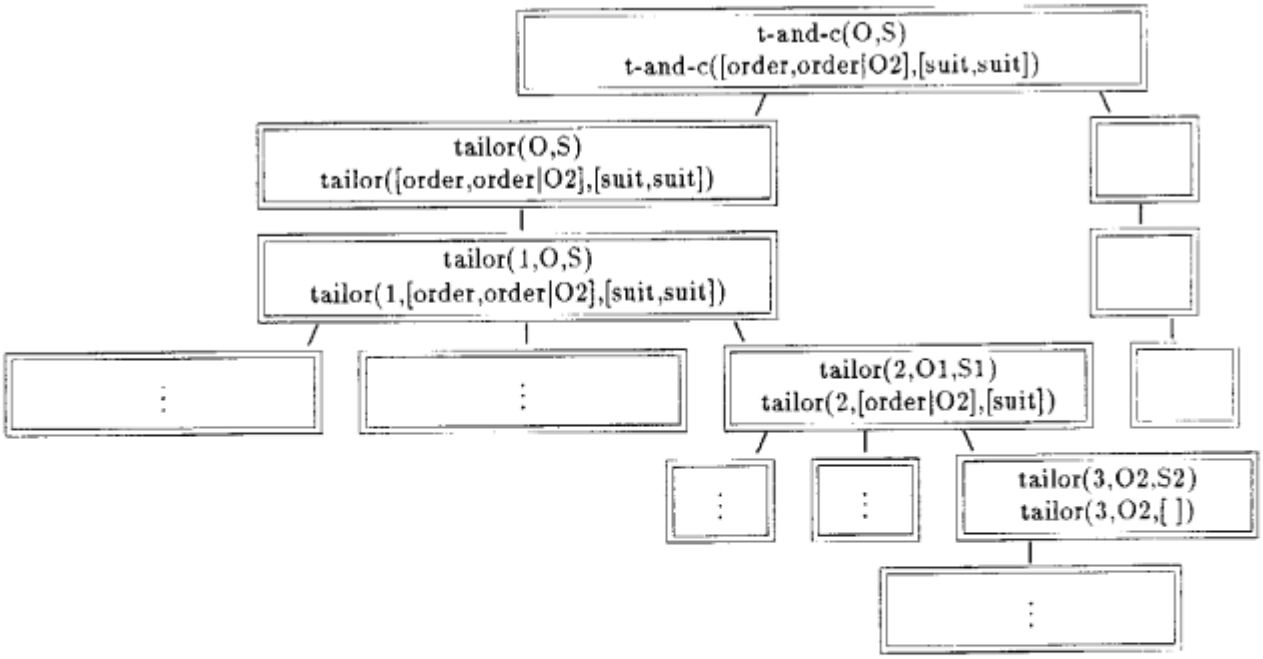
compute-atom-behavior(T: computation tree): atom behavior;
  let $A\nu$ be the root atom of T;
  initialize $\mathcal{A}_0$ to $\{B \mid B \leq A\nu\}$;
  initialize $\mathcal{B}_0$ to $\{\ \}$;
  initialize i to 0;
  repeat
    select $A\sigma$ in $\mathcal{A}_i$ such that $A\sigma$ is minimal in $\mathcal{A}_i$;
    maximally extend the initial tree of $A\sigma$ in T if possible, and let its solution be $A\tau$;
    let $\mathcal{A}_{i+1}$ be $\mathcal{A}_i - \{B | A\sigma \leq B \leq A\tau\}$;
    if $A\sigma \equiv A\tau$ **then** let $\mathcal{B}_{i+1}$ be $\mathcal{B}_i$ **else** let $\mathcal{B}_{i+1}$ be $\mathcal{B}_i + \{(A\sigma, A\tau)\}$;
    increment i by 1;
  until $\mathcal{A}_i$ is empty;
  return $\mathcal{B}_i$;

Figure 5.2.2 Computation of Atom Behavior

5.3 Two Examples of the Diagnosis

Let us show two examples of the application of the diagnosis algorithm.

Example 5.3.1 Let us diagnose the behavior tree in the program P_2 of Example 3.3.1. The tree below is a behavior tree in program P_2.



By traversing a path from the root node in the behavior tree, our system asks the programmer as below to detect a bug. (The first prompt "diagnose?-" instead of "?-" denotes that the execution is under the diagnosis mode.)

```
diagnose?- t-and-c([order|O1],S).
succeeded with t-and-c([order,order|O2],[suit,suit]).

%%% DIAGNOSIS STARTED %%%
```

$t\text{-}and\text{-}c(O,S) \rightarrow t\text{-}and\text{-}c([order,order|O2],[suit,suit])$
$t\text{-}and\text{-}c([order,order|O2],[suit,suit])$ `is in success. O.K.?:  n`

$tailor(O,S) \rightarrow tailor([order,order|O2],[suit,suit])$
$tailor([order,order|O2],[suit,suit])$ `is in success. O.K.?:  n`

$tailor(1,O,S) \rightarrow tailor(1,[order,order|O1],[suit,suit])$
$tailor(1,[order,order|O2],[suit,suit])$ `is in success. O.K.?:  n`

$tailor(2,O1,S1) \rightarrow tailor(2,[order|O2],[suit])$
$tailor(2,[order|O2],[suit])$ `is in success. O.K.?:  n`

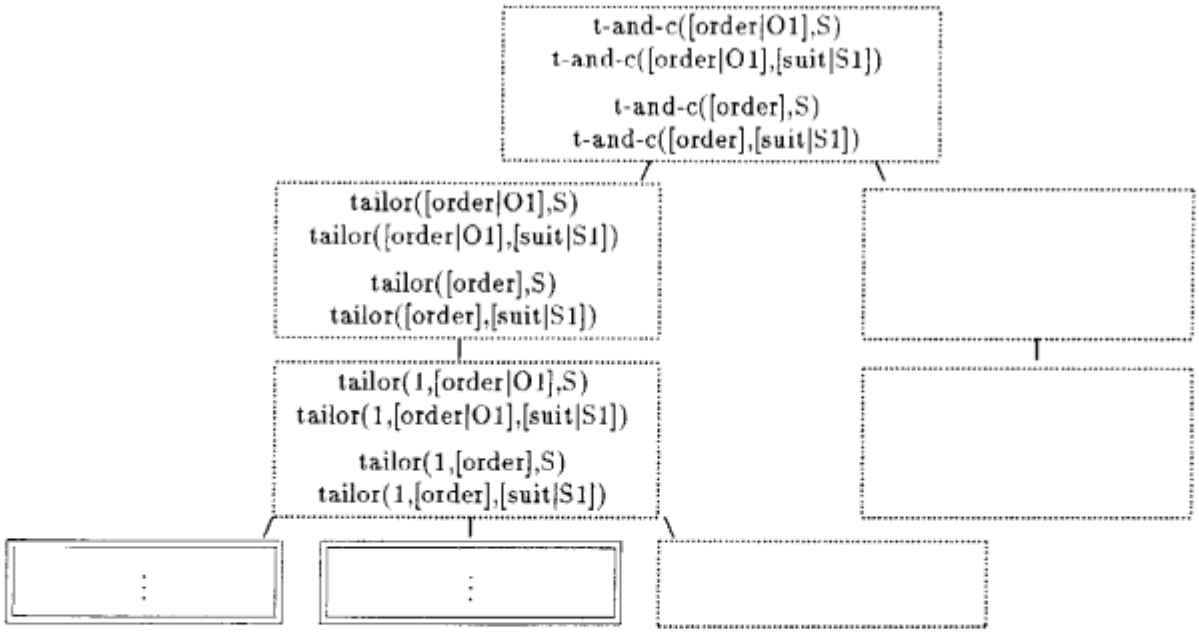
$tailor(3,O2,S2) \rightarrow tailor(3,O2,[\ ])$
$tailor(3,O2,[\ ])$ `is in success. O.K.?:  y`

```
%%% WRONG CLAUSE INSTANCE %%%

        tailor(2,O1,S1) :- 2<3 |
            S1=[suit|Snext], O1=[order|Onext], tailor(3,Onext,Snext).
```

Note that the atom behaviors of primitive atoms are always in our intended behavior interpretation, so that it is unnecessary to ask about the primitive atoms.

Example 5.3.2 Let us diagnose the behavior tree in the program P_3 of Example 3.3.2. Then, the tree below is a behavior tree in program P_3.



By traversing a path from the root node in the behavior tree, our system asks the programmer as below to detect a bug:

```
diagnose?- t-and-c([order],S).
```

```
suspended with t-and-c([order],[suit|S1]).

%%% DIAGNOSIS STARTED %%%
```

$t\text{-}and\text{-}c([order|O1], S) \rightarrow t\text{-}and\text{-}c([order|O1], [suit|S1])$
$t\text{-}and\text{-}c([order], S) \rightarrow t\text{-}and\text{-}c([order], [suit|S1])$
$t\text{-}and\text{-}c([order], [suit|S1])$ is in suspension. O.K.?: n

$tailor([order|O1], S) \rightarrow tailor([order|O1], [suit|S1])$
$tailor([order], S) \rightarrow tailor([order], [suit|S1])$
$tailor([order], [suit|S1])$ is in suspension. O.K.?: n

$tailor(1, [order|O1], S) \rightarrow tailor(1, [order|O1], [suit|S1])$
$tailor(1, [order], S) \rightarrow tailor(1, [order], [suit|S1])$
$tailor(1, [order], [suit|S1])$ is in suspension. O.K.?: n

$tailor(2, [\,], S1)$ is in suspension. O.K.?: n

```
%%% WRONG SUSPENSION ATOM %%%
        tailor(2,[ ],S1).
```

6. Soundness and Completeness of the Diagnosis Algorithm

The soundness and completeness of our diagnosis algorithm is stated as below:

Theorem 6.1 (Soundness of the Diagnosis Algorithm)
If "*diagnose*" returns a clause C, then it is a wrong clause instance. If "*diagnose*" returns an atom A, then it is a wrong suspension atom.

Proof. Obvious from the definitions of wrong clause instances and wrong suspension atoms.

Theorem 6.2 (Completeness of the Diagnosis Algorithm)
If the root label of a behavior tree is not in $M(P)$, then "*diagnosis*" returns either a wrong clause instance or a wrong suspension atom w.r.t. $M(P)$.

Proof. Obvious, since "*diagnose*" simulates the "only if" part of the bug detection theorem.

7. Implementation of the Diagnosis Algorithm

In this section, we will discuss several improvements of (and extensions to) the diagnosis algorithm in Section 5.

(1) Diagnosis Using Truncated Atom Behavior

The atom behaviors checked in the diagnosis algorithm of Section 5 consists of any atom pairs representing autonomous transition intervals that starts from any atom more general than the final atom. For example, suppose that the execution of a top-level goal "$p([1,2,3,\ldots,100], Y)$" is diagnosed, and the program of "p" processes the first list arguement from the first element one by one, and instantiate the second argument one by one. Then, the atom behavior checked in our algorithm includes all the atom pairs of the form

$(p([1|X1], Y), p([1|X1], t_1))$,
$(p([1,2|X2], Y), p([1,2|X2], t_2))$,
$\vdots$
$(p([1,2,\ldots,100|X100], Y), p([1,2,\ldots,100|X100], t_{100}))$,

$(p([1,2,\ldots,100],Y),p([1,2,\ldots,100],t))$.
In the actual execution of the top-level goal, however, the partially instantiated atom "$p([1,2,\ldots,i|Xi])$" never appears as far as we can assume that each arguement is passed all at once ($1 \leq i \leq 100$). To restrict our attention to the transition intervals that actually occur, we will truncate the less instantiated atom pairs as follows:

Definition Truncated Atom Behavior

Let $\mathcal{B}_{A\nu}$ be an atom behavior and $A\mu$ be an atom such that $A\mu \leq A\nu$. Then, a set of atom pairs

$\{ (A\sigma', A\tau') \mid A\sigma' < A\tau'$, and
there exists an atom behavior $(A\sigma, A\tau)$ in $\mathcal{B}_{A\nu}$ s.t.
$A\sigma'$ is a most general unification of $A\sigma$ and $A\mu$,
$A\tau'$ is a most general unification of $A\tau$ and $A\mu$ $\}$

is called the *truncated atom behavior of* $\mathcal{B}_{A\nu}$ *by* $A\mu$ and denoted by ${}_{A\mu}\mathcal{B}_{A\nu}$.

A usual atom behavior $\mathcal{B}_{p(t_1,t_2,\ldots,t_n)}$ is considered as ${}_{p(X_1,X_2,\ldots,X_n)}\mathcal{B}_{p(t_1,t_2,\ldots,t_n)}$, where $X_1, X_2, \ldots, X_n$ are distinct variables.

When this truncated atom behavior is employed instead of the original atom behavior, the size of each query is much reduced. The algorithm in Section 5 is modified in the following three points:

- The 4-th line of "*diagnose*" is modified to
  let $\mathcal{B}$ be the result of "*compute-atom-behavior*" applied to T and A;
- The query is modified to
  $A\mu$ is executed.
  $A\sigma_1 \rightarrow A\tau_1$
  $A\sigma_2 \rightarrow A\tau_2$
  $\vdots$
  $A\sigma_k \rightarrow A\tau_k$
  $A\nu$ is in success. O.K.? :
- The subprocedure "*compute-atom-behavior*" takes one more argument $A\mu$, and the 2nd line is modified to
  initialize $\mathcal{A}_0$ to $\{B|A\mu \leq B \leq A\nu\}$

(2) Diagnosis Using Difference Atom Behavior

The atom behaviors checked in the diagnosis algorithm of Section 5 consists of any atom pairs representing maximal transition intervals. Each atom pair $(A\sigma, A\tau)$ in an atom behavior says that, if an atom is instantiated at least to the form $A\sigma$, it can autonomously instantiated at most to the form $A\tau$. Though such an atom behavior represents an inherent characteristics of the atom, two transitions sometimes overlap. For example, suppose that $(A\sigma, A\tau)$ is an atom pair in $\mathcal{B}$, and there exists another atom pair $(A\theta, A\eta)$ such that $A\theta$ is less instantiated than $A\sigma$. Then, some portion of the transition from $A\sigma$ to $A\tau$ is already included in the transition from $A\theta$ to $A\eta$. That is, let $A\sigma'$ be a most general unification of $A\sigma$ and $A\eta$. Then, the transition from $A\sigma$ to $A\sigma'$ is implicitly included in $(A\theta, A\eta)$, and the remaining transition is represented by $(A\sigma', A\tau)$. (If $A\sigma'$ and $A\tau$ are identical, $(A\sigma, A\tau)$ is turned out to have no inherent information.) To extract the portion inherent to the transition itself, we will define as follows:

Definition Difference Atom Behavior

Let $\mathcal{B}$ be an atom behavior, $(A\sigma, A\tau)$ be an atom pair in $\mathcal{B}$, and $A\sigma'$ be an atom such that $A\sigma \leq A\sigma' < A\tau$. Then, an atom pair $(A\sigma', A\tau)$ is called a *difference atom pair of* $(A\sigma, A\tau)$ *in* $\mathcal{B}$ when

(a) there exists no atom pair $(A\theta, A\eta)$ in $\mathcal{B}$ such that $A\theta < A\sigma$, and $A\sigma'$ is identical to $A\sigma$, or

(b) there exists an atom pair $(A\theta, A\eta)$ in $\mathcal{B}$ such that $A\theta < A\sigma$, and $A\sigma'$ is a most general unification of $A\sigma$ and $A\eta$, and

(c) there exists no other $A\sigma''$ satisfying (a) or (b) and more general than $A\sigma'$.

An atom pair $(A\sigma', A\tau)$ is called a *difference atom pair in* $\mathcal{B}$ when there exists an atom pair $(A\sigma, A\tau)$ such that $(A\sigma', A\tau)$ is a difference atom behavior of $(A\sigma, A\tau)$ in $\mathcal{B}$. A set of atom pairs $\bar{\mathcal{B}}$ is called the difference atom behaviors of $\mathcal{B}$ when it consists of all the difference atom pairs in $\mathcal{B}$.

Again, when this difference atom behavior is employed instead of the original atom behavior, the size of each query is sometimes much reduced, because atom pairs with no inherent information are eliminated. Moreover, the meaning of query is sometimes easier to grasp, because overlapping between two transitions is eliminated. The algorithm in Section 5 is modified only in the following one point:

- The **repeat** loop in the subprocedure "*compute-atom-behavior*" is modified to

repeat
  select $A\sigma$ in $\mathcal{A}_i$ such that $A\sigma$ is minimal in $\mathcal{A}_i$;
  if there exists an atom pair $(A\theta, A\eta)$ in $\mathcal{B}_i$
    s.t. $A\theta < A\sigma$, $A\sigma'$ is an m.g.u. of $A\sigma$ and $A\eta$, and $A\sigma < A\sigma'$
  then let $\mathcal{A}_{i+1}$ be $\mathcal{A}_i - \{B | A\sigma \leq B < A\sigma'\}$;
      let $\mathcal{B}_{i+1}$ be $\mathcal{B}_i$;
  else maximally extend the initial tree of $A\sigma$ in T if possible, and let its solution be $A\tau$;
      let $\mathcal{A}_{i+1}$ be $\mathcal{A}_i - \{B | A\sigma \leq B \leq A\tau\}$;
      if $A\sigma \equiv A\tau$ **then** let $\mathcal{B}_{i+1}$ be $\mathcal{B}_i$ **else** let $\mathcal{B}_{i+1}$ be $\mathcal{B}_i + \{(A\sigma, A\tau)\}$;
  increment i by 1;
until $\mathcal{A}_i$ is empty;

(3) Diagnosis Utilizing Answer Database

One of the reasons Shapiro's algorithmic debugging has appealed so strongly [20],[21] is that the number of queries human programmers need to answer can be much lessened due to the use of an "answer database" by accumulating answers to previous queries. A new query is first posed to the "answer database," asked to the programmer only if the "answer database" fails to answer it, and the answer is added to the "answer database" so that the programmer need not answer the same query over again.

The "answer database" can be also utilized for our diagnosis in the same way as Shapiro's algorithmic debugging, though atom behaviors, the objects diagnosed in our diagnosis, is a little complicated than atoms, the objects diagnosed in Shapiro's algorithmic debugging. For example, if $\mathcal{B}$ is a recorded success atom behavior of $A\nu$, it contains the information about an atom behavior of $A\nu'$ when $A\nu \leq A\nu'$.

(4) Diagnosis in the "Spy" Manner

The diagnosis algorithm described in this paper is in the "trace" manner, i.e., it descends a behavior tree from the parent atom behavior to its immediate child atom behaviors

step by step. We can modify the diagnosis algorithm to behave in the "spy" manner (of DEC10 Prolog), i.e., it asks queries about atom behaviors with the same predicates continually (and diagnoses other immediate child atom behaviors only when no bug is found for the predicate) [15]. Queries asked by such a diagnosis algorithm are easier for human programmers to answer, since we need not frequently change our attention to different predicates. (In general, for each priority rule for deciding which node in a behavior tree is diagnosed prior to the other nodes, e.g., top-down, bottom-up or "spy", we have a corresponding diagnosis algorithm. It is even possible to decide the priority according to the user's direction, e.g., defer the check of atom behavior in response to the user's answer meaning "I don't know now, or don't want to check now, whether the atom behavior is true or not" [15].)

In particular, the diagnosis algorithm in the "spy" manner is suitable for the diagnosis of the GHC programs in the object-oriented style. When GHC is used for object-oriented style programming, we usually prepare a predicate for each class of objects. For example, a class of objects "counter" is programmed in GHC as below [22]:

```
counter([clear|Msgs],State) :- | counter(Msgs,0).
counter([up|Msgs],State) :- |
            add(State,1,NewState), counter(Msgs,NewState).
counter([down|Msgs],State) :- |
            subtract(State,1,NewState), counter(Msgs,NewState).
counter([ ],State).
```

Such a predicate usually takes

- arguments representing the streams of messages sent from other objects,
- (possibly 0) arguments representing the streams of messages sent to other objects, and
- an argument representing the current state of the object.

In addition, the clauses defining the predicate are usually linear recursive, i.e., each clause contains at most one atom with the head predicate in the body. For such a GHC program, the diagnosis in the "spy" manner proceeds very naturally, since the queries asked continually in the "spy" manner correspond to the continual state transitions of the same object.

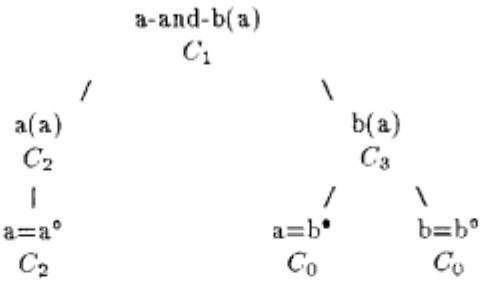
8. Discussion

(1) Truly Parallel v.s. Non-deterministic Sequential

Recall that we have formalized the notion of computation tree based on the non-deterministic sequential GHC execution, in which the instantiation caused by the extension at one node is propagated immediately to all the other nodes. (Let us call such execution *non-deterministic sequential*.) At first glance, it seems unnatural, since we cannot guarantee that the instantiation is propagated in such a way. If nodes are assigned to different processor, the extension at some node might be done before the instantiation caused at other nodes has been propagated. (Let us call such execution *truly parallel*.) For example, consider the following program:

```
C1: a-and-b(X) :- | a(X), b(X).
C2: a(X) :- | X=a.
C3: b(X) :- | X=b, X=b.
```

Suppose that the substitution $< X \Leftarrow a >$ caused by the execution of "$X = a$" is propagated before the extensions at other nodes are tried except the extension at the right node labelled with "$X = b$". Then, we will have the tree below, which is not our maximal computation tree.



However, note that, if we can assume that the instantiation caused at each node is eventually propagated to all the other nodes, we can say that

- a labelled tree is a success tree by non-deterministic sequential execution if and only if it is a success tree by truly parallel execution,
- a labelled tree is a suspension tree by non-deterministic sequential execution if and only if it is a suspension tree by truly parallel execution, and
- the execution of an atom fails by non-deterministic sequential execution if and only if it fails by truly parallel execution.

Hence, the notions of success tree and suspension tree do not depend on the non-deterministic sequential execution mechanism we have employed in Section 3.2. Moreover, once a maximal computation tree (by non-deterministic sequential execution) is given, any maximal subextension in it, hence its atom behavior, is also independent of the sequentiality. (Note that the notions of truncated atom behavior and difference atom behavior in Section 7 are also independent of the sequentiality.)

(2) Comparison with the Diagnosis of Prolog Programs

As for Prolog programs, since the advocacy by Shapiro [20],[21], many studies to extend and refine his approach have been made [2],[3],[4],[5],[8],[13],[15],[16],[17],[18].

Suppose that we identify any Prolog clause of the form

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

with a GHC clause of the form

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

Then, as far as the predicate of an atom is not undefined, the execution of an atom in such a GHC program is never suspended immediately since there always exists some clause to which the execution is committable. The failure in such a GHC program corresponds to the backtracking in the original Prolog program.

When our diagnosis algorithm is applied to such a converted GHC program, it can detect "wrong suspension atom" only when the predicate is undefined, and otherwise it can only detect "wrong clause instances," which correspond to the "wrong clause instances" and some of the "uncovered atoms" detected by Shapiro's diagnosis algorithm applied to the original Prolog program.

(3) Comparison with Other Diagnosis of Concurrent Logic Programs

As for (so called) concurrent logic programs, several researches have been just begun [6],[10],[11],[12],[14],[19],[23],[24],[25],[26].

Lloyd and Takeuchi [12] have tried to generalize Shapiro's approach for the diagnosis of GHC programs by considering 3 sets of ground atoms $M_{success}$, $M_{failure}$, $M_{suspension}$ instead of considering only the least Herbrand model $M_{success}$, and naturally noticed that some bugs inherent to GHC cannot be detected if only ground atoms are considered. See Takeuchi [24] for its implementation in GHC. (Maeda, Uoi and Tokura [14], Sato, Aida and Saitou [19] are for the diagnosis of GHC program, and Tatemura and Tanaka [26] is for the diagnosis of their committed-choice language FLENG similar to GHC.)

Lichtenstein and Shapiro [10] has generalized Shapiro's approach for the diagnosis of Flat Concurrent Prolog (FCP) based on the notion of atom behavior. (We borrowed the term "atom behavior" from their paper.) Their atom behavior, however, does not correspond to a set of maximal transitions, but a sequence of transitions observed whenever an immediate extension (in our terminology of Section 3.2) is applied or an instantiation is imported from outside. Naturally, their atom behaviors contain much finer, hence much more, information than ours and depends on each specific course of non-deterministic computation. (The information involved in our diagnsosis, however, suffices for the diagnosis, if, for checking whether a GHC program is erroneous or not, we only observe how an initial top-level goal is instantiated in the final success, suspension or failure state.)

To lessen the information involved in the diagnosis, Lichitenstein and Shapiro [11] have later proposed a method to diagnose FCP programs using rougher information obtained by applying abstaction to their atom behaviors. Though we have shown the diagnosis algorithm using atom behaviors, our notion of atom behavior is already an abstraction of GHC computation. Similarly to Lichtenstein and Shapiro's approach, if a different abstraction of the GHC computation is employed, a diagnosis alogorithm for the different aspect of the GHC computation is obtained.

Huntbach [6] tried to apply Shapiro's approach to PARLOG. However, because his approach focuses its attention on computation trees, not behavior trees, his diagnosis algorithm is naturally unable to detect the bugs inherent to the synchronization mechanism of committed-choice languages based on how far the variables in the goals are instantiated.

9. Conclusions

We have presented a framework for the diagnosis of GHC programs. This method is an element of our system for modification of GHC programs under development.

Acknowledgements

Our modification system under development is a subproject of the Fifth Generation Computer System (FGCS) "Intelligent Programming System." The authors 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 Lab.) for their advice and encouragement.

References

[1] 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.

[2] Dershowitz, N. and Y-J Lee, "Deductive Debugging," Proc. of 1987 Symposium on Logic Programming, pp.298–306, San Francisco, August 1987.

[3] Drabent, W., S.Nadjm-Tehrani and J.Maluszynski, "The Use of Assertions in Algorithmic Debugging," Proc. of the International Conference on Fifth Generation Computer Systems 1988, Tokyo, November 1988.

[4] Edman, A. and S.A.Tärnlund, "Mechanization of An Oracle in A Debugging System," Proc. of 8th International Joint Conference on Artificial Intelligence, pp.553–555, Karlsruhe, August 1983.

[5] Ferrand, G., "Error Diagnosis in Logic Programming," J. of Logic Programming, Vol.4, pp.177–198, 1987.

[6] Huntbach, M., "Algorithmic Parlog Debugging," Proc. 4th Symposium on Logic Programming, pp.288–297, San Francisco, August 1987.

[7] Kanamori, T. and M.Maeji, "A Preliminary Note on the Semantics of Guarded Horn Clauses," ICOT Technical Report TR-434, ICOT, Tokyo, December 1988.

[8] Kanamori, T., T.Kawamura, M.Maeji and K.Horiuchi, "Logic Program Diagnosis from Specifications," ICOT Technical Report TR-447, ICOT, Tokyo, February 1989.

[9] Kanamori, T. and M.Maeji, "A Fixpoint Semantics of GHC Programs," ICOT Technical Report TR-5??, ICOT, Tokyo, March 1990.

[10] Lichtenstein, Y. and E.Shapiro, "Concurrent Algorithmic Debugging," The Weizmann Institute of Science, Department of Computer Science, Technical Note CS87-20, 1987.

[11] Lichtenstein, Y. and E.Shapiro, "Abstract Algorithmic Debugging," Proc. of 1987 Symposium on Logic Programming, pp. 512–531, San Francisco, August 1987.

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

[13] Lloyd, J.W., "Declarative Program Diagnosis," Technical Report 86/3, Department of Computer Science, University of Melbourne, 1986. Also New Generartion Computing, Vol.5, No.2, pp.133–154, 1987.

[14] Maeda, M., H.Ueoi and N.Tokura, "A Debugging Method for GHC Programs," (in Japanese) Research Report of SIG Foundation of Software 28-4, Japan Information Processing Society, Tokyo, March 1989.

[15] Maeji, M. and T.Kanamori, "Top-down Zooming Diagnosis of Logic Programs," Presented at RIMS Symposium on Mathematical Methods in Software Science and Engineering '87, Kyoto, September 1987. Also RIMS Research Report 655, pp. 147–166, Research Institute for Mathematical Sciences, Kyoto University, April 1988. Also ICOT Technical Report TR-290, ICOT, Tokyo, August 1987.

[16] Pereira, L.M., "Rational Debugging in Logic Programming," Proc. of 3rd International Conference on Logic Programming, pp. 203–210, London, July 1986.

[17] Pereira, L.M. and M.Calejo, "A Framework for Prolog Debugging," Proc. of 5th International Conference and Symposium on Logic Programming," pp.481–495, Seatle, August 1988.

[18] Plaisted, D., "An Efficient Bug Location Algorithm," Proc. of 2nd International Logic Programming Conference, pp. 151–157, Uppsala, July 1984.

[19] Sato, S., H.Aida and T.Saitou, "An Experimental Debugging Tool for Concurrent Logic Programs," Proc. of 4th National Conference of Japan Society for Software Science and Technology, pp. 447–450, Kyoto, November 1987.

[20] Shapiro, E.Y., "Algorithmic Program Debugging," An ACM Distinguished Dissertation 1982, The MIT Press, 1983. Also Research Report 237, Yale University, Department of Computer Science, 1982.

[21] Shapiro, E.Y., "Algorithmic Program Diagnosis," Conf. Rec. of the 9th ACM Symposium on Principles of Programming Languages, pp.299–308, 1984.

[22] Shapiro, E.Y. and A.Takeuchi, "Object Oriented Programming in Concurrent Prolog," New Generation Computing, Vol.1, pp.25–48, OHMSHA, LTD and Springer Verlag, 1983.

[23] Takeuchi, A., "On the Algorithmic Debugging of GHC Programs," (in Japanese) Proc. of Spring National Conference of Japan Information Processing Society, pp.499-501, Tokyo, March 1986.

[24] Takeuchi, A., "Algorithmic Debugging of GHC Programs and its Implementation in GHC," ICOT Technical Report TR-185, ICOT, Tokyo, June 1986.

[25] Takeuchi, A., "GHC Programming Environment," (in Japanese), in *A Concurrent Logic Programming Language GHC and its Applications* (K.Fuchi, K.Furukawa and F.Mizoguchi Eds.), pp.191–215, Kyouritsu Pub. Co., September 1987.

[26] Tatemura, J. and H.Tanaka, "Debugger for the Parallel Logic Programs : FLENG," Proc. of the Logic Prpgramming '89, pp.133–142, Tokyo, July 1989.

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

[28] 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.