TR-331

# Detecting Functionality of Logic Programs Based on Abstract Hybrid Interpretation

by

T. Kanamori, K.Horiuchi &
T. Kawamura

January, 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

# Detecting Functionality of Logic Programs Based on Abstract Hybrid Interpretation

Tadashi KANAMORI    Kenji HORIUCHI    Tadashi KAWAMURA

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

## Abstract

This paper presents a framework for detecting functionality of Prolog programs by abstract interpretation. The framework is based on mode analysis of Prolog programs. The mode analysis is in turn based on OLDT resolution by Tamaki and Sato, a hybrid of the top-down and the bottom-up interpretations of Prolog programs. By directly abstracting the hybrid interpretation according to the mode structure, we can infer mode patterns of goals without either diving into infinite looping or wasting time for mode patterns of irrelevant goals. Functionality is detected by overestimating the solution number of each goal for each mode pattern during the mode analysis process, and by guaranteeing that the solution number is at most 1. This method is a refinement of the one by Debray and Warren.

Keywords : Program Analysis, Functionality, Abstract Interpretation, Prolog.

## Contents

## 1. Introduction

Though relationality (or nondeterminacy) is one of the prominent features of Prolog, many Prolog programs in practice are functional (or deterministic) [3],[10],[12]. When a goal is executed with some arguments instantiated to ground terms, the form of the goal at success time is often unique. If such functionality is detected, the space prepared for later backtracking is no longer necessary to be kept so that much memory space is reduced. Due to its importance, many researches have been done on functionality detection [3],[10],[12].

In this paper, a framework for detecting functionality of Prolog programs by abstract interpretation is presented. The framework is based on mode analysis of Prolog programs. The mode analysis is in turn based on OLDT resolution by Tamaki and Sato, a hybrid of the top-down and the bottom-up interpretations of Prolog programs. By directly abstracting the hybrid interpretation according to the mode structure, we can infer mode patterns of goals without either diving into infinite looping or wasting time for mode patterns of irrelevant goals. Functionality is detected by overestimating the solution number of each goal for each mode pattern during the mode analysis process, and by guaranteeing that the solution number is at most 1. This method is a refinement of the one by Debray and Warren [3].

This paper is organized as follows: After presenting the hybrid interpretation of Prolog programs in Section 2, we will show a mode analysis method in Section 3, because the mode information plays a crucial role in our functionality detection. Then, we will show a functionality detection method based on the mode analysis method in Section 4.

The following sections assume familiarity with the basic terminologies of first order logic such as term, atom (atomic formula), definite clause, negative clause, substitution, most general unifier (m.g.u.) and so on is assumed. The syntax of DEC-10 Prolog is followed. Negative clauses are often confused with sequences of atoms. Syntactical variables are $X, Y, Z$ for variables, $s, t$ for terms and $A, B$ for atoms, possibly with primes and subscripts. In addition, $t[Z]$ is used for a term containing some occurrence of variable $Z$, and $\theta, \sigma, \tau$ for substitutions.

## 2. Standard Hybrid Interpretation of Logic Programs

In this section, we will first present a basic hybrid interpretation method of Prolog programs [13], then a modified hybrid interpretation method suitable for the basis of the abstract interpretation presented later.

### 2.1 Basic Hybrid Interpretation of Logic Programs

#### (1) Search Tree

A *search tree* is a tree with its nodes labelled with negative or null clauses, and with its edges labelled with substitutions. A *search tree of* negative clause $G$ is a search tree whose root node is labelled with $G$. The relation between a node and its child nodes in a search tree is specified in various ways depending on various strategies of "resolution". In this paper, the class of "ordered linear" strategies is assumed. (See the explanations of OLDT resolution in the following subsection (4), and of OLD resolution in Section 3.)

A *refutation* of negative clause $G$ is a path in a search tree of $G$ from the root to a node labelled with the null clause $\square$. Let $\theta_1, \theta_2, \ldots, \theta_k$ be the labels of the edges on the path.

Then, the *answer substitution of the refutation* is the composed substitution $\tau = \theta_1\theta_2\cdots\theta_k$, and the *solution of the refutation* is $G\tau$.

Consider a path in a search tree from one node to another node. Intuitively, when the leftmost atom of the starting node's label is refuted just at the ending node, the path is called a unit subrefutation of the atom. More formally, let $G_0, G_1, \ldots, G_k$ be a sequence of labels of the nodes and $\theta_1, \theta_2, \ldots, \theta_k$ be the labels of the edges on the path. The path is called a *unit subrefutation* of atom $A$ when $G_0, G_1, G_2, \ldots, G_{k-1}, G_k$ are of the form

"$A, G$",
"$H_1, G\theta_1$",
"$H_2, G\theta_1\theta_2$",
$\vdots$
"$H_{k-1}, G\theta_1\theta_2\cdots\theta_{k-1}$"
"$G\theta_1\theta_2\ldots\theta_k$",

respectively, where $G, H_1, H_2, \ldots, H_{k-1}$ are sequences of atoms. Then, the *answer substitution of the unit subrefutation* is the composed substitution $\tau = \theta_1\theta_2\cdots\theta_k$, and the *solution of the unit subrefutaion* is $A\tau$.

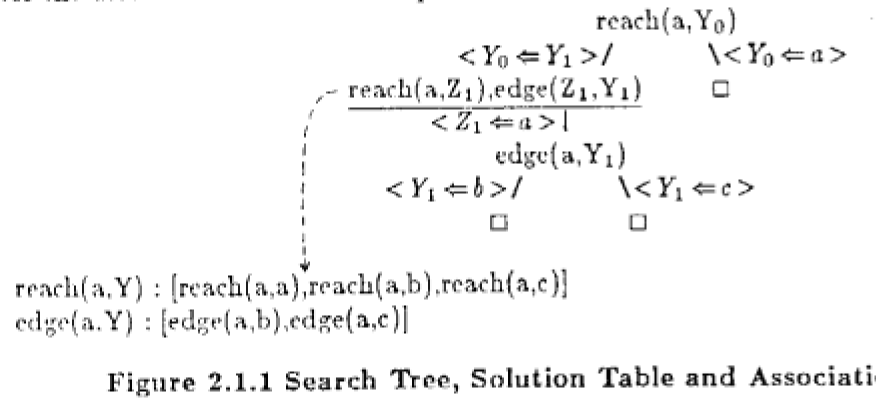
## (2) Solution Table

A *solution table* is a set of entries. Each entry is a pair of the *key* and the *solution list*. The key is an atom such that there is no other identical key (modulo renaming of variables) in the solution table. The solution list is a list of atoms, called *solutions*, such that each solution in it is an instance of the corresponding key.

## (3) Association

Let $Tr$ be a search tree whose nodes labelled with non-null clauses are classified into either *solution nodes* or *lookup nodes*, and let $Tb$ be a solution table. (The solution nodes and lookup nodes are explained later.) An *association* of $Tr$ and $Tb$ is a set of pointers pointing from each lookup node in $Tr$ *into* some solution list in $Tb$ such that the leftmost atom of the lookup node's label and the key of the solution list are variants of each other.

*Example 2.1.1* An association of a search tree of "$reach(a, Y_0)$" and a solution table is depicted in the figure below. The underline denotes the lookup node, and the dotted line denotes the association from the lookup node.



reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c)]
edge(a,Y) : [edge(a,b),edge(a,c)]

**Figure 2.1.1 Search Tree, Solution Table and Association**

## (4) OLDT Structure

The hybrid Prolog interpreter is modeled by OLDT resolution. An *OLDT structure* of negative clause $G$ is a triple $(Tr, Tb, As)$ satisfying the following conditions:

(a) $Tr$ is a search tree of $G$. The relation between a node and its child nodes in a search tree is specified by the following *OLDT resolution*. Each node of the search tree labelled with non-null clause is classified into either a *solution node* or a *lookup node*.

(b) $Tb$ is a solution table.

(c) $As$ is an association of $Tr$ and $Tb$. The tail of the solution list pointed from a lookup node is called the *associated solution list* of the lookup node.

Let $G$ be a negative clause of the form "$A_1, A_2, \ldots, A_n$" ($n \geq 1$). A node of OLDT structure $(Tr, Tb, As)$ labelled with negative clause $G$ is said to be *OLDT resolvable* when it satisfies either of the following conditions:

(a) The node is a terminal solution node of $Tr$, and there is some definite clause "$B_0$ :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program $P$ such that $A_1$ and $B_0$ are unifiable, say by an m.g.u. $\theta$. (Without loss of generality, we assume that the m.g.u. $\theta$ substitutes a term consisting of fresh variables for every variable in $A_1$ and the definite clause.) The negative clause (or possibly null clause) "$B_1\theta, B_2\theta, \ldots, B_m\theta,\ A_2\theta, \ldots, A_n\theta$" is called the *OLDT resolvent*.

(b) The node is a lookup node of $Tr$, and there is some solution $B\tau$ in the associated solution list of the lookup node such that $B\tau$ is an instance of $A_1$, say by an instantiation $\theta$. (Again, we assume that the instantiation $\theta$ substitutes a term consisting of fresh variables for every variable in $A_1$, that is, a fresh variant of $B\tau$ is an instance of $A_1$ by $\theta$.) The negative clause (or possibly null clause) "$A_2\theta, \ldots, A_n\theta$" is called the *OLDT resolvent*.

The restriction of the substitution $\theta$ to the variables of $A_1$ is called the *substitution of the OLDT resolution*.

The *initial OLDT structure* of negative clause $G$ is the triple $(Tr_0, Tb_0, As_0)$, where $Tr_0$ is a search tree consisting of just the root solution node labelled with $G$, $Tb_0$ is the solution table consisting of just one entry whose key is the leftmost atom of $G$ and solution list is the empty list, and $As_0$ is the empty set of pointers.

An *immediate extension* of OLDT structure $(Tr, Tb, As)$ in program $P$ is the result of the following operations, when a node $v$ of OLDT structure $(Tr, Tb, As)$ is OLDT resolvable.

(a) When $v$ is a terminal solution node, let $C_1, C_2, \ldots, C_k$ ($k \geq 0$) be all the clauses with which the node $v$ is OLDT resolvable, and $G_1, G_2, \ldots, G_k$ be the respective OLDT resolvents. Then add $k$ child nodes of $v$ labelled with $G_1, G_2, \ldots, G_k$, to $v$. The edge from $v$ to the node labelled with $G_i$ is labelled with $\theta_i$, where $\theta_i$ is the substitution of the OLDT resolution with $C_i$. When $v$ is a lookup node, let $B_1\tau_1$, $B_2\tau_2$, $\ldots$, $B_k\tau_k$ ($k \geq 0$) be all the solutions with which the node $v$ is OLDT resolvable, and $G_1, G_2, \ldots, G_k$ be the respective OLDT resolvents. Then add $k$ child nodes of $v$ labelled with $G_1, G_2, \ldots, G_k$, to $v$. The edge from $v$ to the node labelled with $G_i$ is labelled with $\theta_i$, where $\theta_i$ is the substitution of the OLDT resolution with $B_i\tau_i$. A new node labelled with a non-null clause is a lookup node when the leftmost atom of the new negative clause is a variant of some key in $Tb$, and is a solution node otherwise.

(b) Replace the pointer from the OLDT resolved lookup node with the one pointing to the last of the associated solution list. Add a pointer from the new lookup node to the head of the solution list of the corresponding key.

(c) When a new node is a solution node, add a new entry whose key is the leftmost atom of the label of the new node and whose solution list is the empty list. When a new node is a lookup node, add no new entry. For each unit subrefutation of atom $A$ (if any) starting from a solution node and ending with some of the new nodes, add its solution $Ar$ to the last of the solution list of $A$ in $Tb$, if $Ar$ is not in the solution list.

An OLDT structure $(Tr', Tb', As')$ is an extension of OLDT structure $(Tr, Tb, As)$ if $(Tr', Tb', As')$ is obtained from $(Tr, Tb, As)$ through successive application of immediate extensions.

*Example 2.1.2* Consider the following "graph reachability" program by Tamaki and Sato [13].

```
reach(X,Y) :- reach(X,Z), edge(Z,Y).
reach(X,X).
edge(a,b).
edge(a,c).
edge(b,a).
edge(b,d).
```

Then, the hybrid interpretation generates the following OLDT structures of "$reach(a, Y_0)$".

First, the initial OLDT structure below is generated. The root node of the search tree is a solution node. The solution table contains only one entry with its key $reach(a, Y)$ and its solution list [ ].

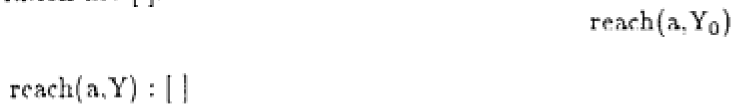


**Figure 2.1.2 Basic Hybrid Interpretation at Step 1**

Secondly, the root node "$reach(a, Y_0)$" is OLDT resolved using the program to generate two child nodes. The generated left child node is a lookup node, because its leftmost atom is a variant of the key in the solution table. The association associates the lookup node to the head of the solution list of $reach(a, Y)$. The generated right child node is the end of a unit subrefutation of $reach(a, Y_0)$. Its solution $reach(a, a)$ is added to the solution list of $reach(a, Y)$.

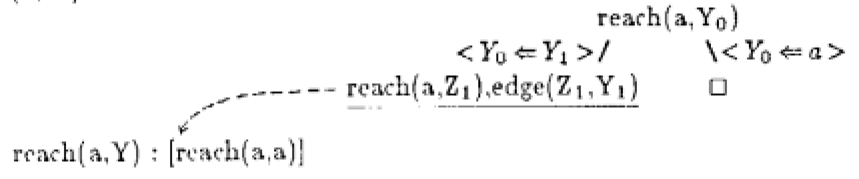


**Figure 2.1.3 Basic Hybrid Interpretation at Step 2**

Thirdly, the lookup node is OLDT resolved using the solution table to generate one child solution node. The association associates the lookup node to the last of the solution list.

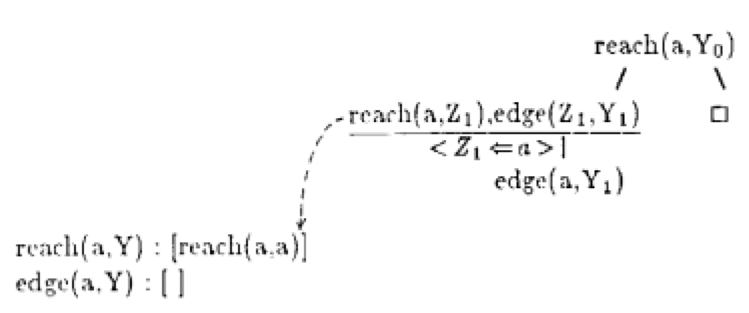
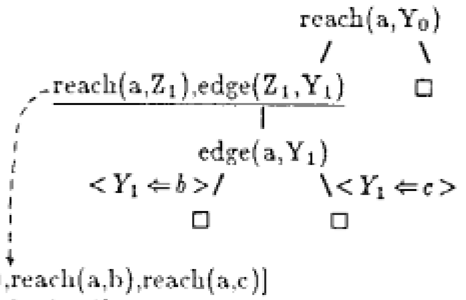


**Figure 2.1.4 Basic Hybrid Interpretation at Step 3**

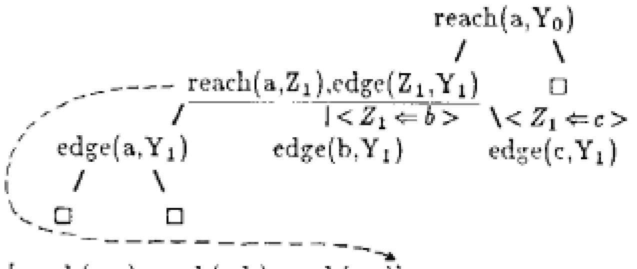
Fourthly, the generated solution node is OLDT resolved further using the program to generate two new nodes labelled with the null clauses. These two nodes add two solutions *reach*$(a, b)$ and *reach*$(a, c)$ to the last of the solution list of *reach*$(a, Y)$, and two solutions *edge*$(a, b)$ and *edge*$(a, c)$ to the last of the solution list of *edge*$(a, Y)$.



reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c)]
edge(a,Y) : [edge(a,b),edge(a,c)]

**Figure 2.1.5 Basic Hybrid Interpretation at Step 4**

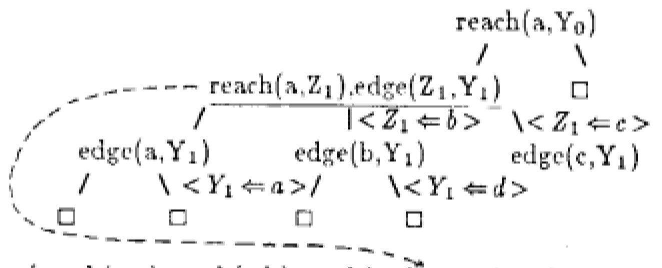
Fifthly, the lookup node is OLDT resolved using the solution table, since new solutions were added to the solution list of *reach*$(a, Y)$.



reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c)]
edge(a,Y) : [edge(a,b),edge(a,c)]
edge(b,Y) : [ ]
edge(c,Y) : [ ]

**Figure 2.1.6 Basic Hybrid Interpretation at Step 5**

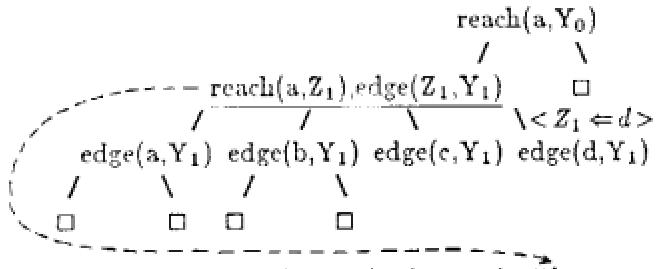
Sixthly, the left new solution node "*edge*$(b, Y_1)$" is OLDT resolved, and one new solution *reach*$(a, d)$ is added to the solution list.



reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c),reach(a,d)]
edge(a,Y) : [edge(a,b),edge(a,c)]
edge(b,Y) : [edge(b,a),edge(b,d)]
edge(c,Y) : [ ]

**Figure 2.1.7 Basic Hybrid Interpretation at Step 6**

Lastly, the lookup node is OLDT resolved once more using the solution table, and the extension process stops, because the solution nodes labelled with $edge(c, Y_1)$ and $edge(d, Y_1)$ are not OLDT resolvable.



reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c),reach(a,d)]
edge(a,Y) : [edge(a,b),edge(a,c)]
edge(b,Y) : [edge(b,a),edge(b,d)]
edge(c,Y) : [ ]
edge(d,Y) : [ ]

**Figure 2.1.8 Basic Hybrid Interpretation at Step 7**

Though all solutions were found under the depth-first from-left-to-right extension strategy in this example, the strategy is not complete in general. The reason of the incompleteness is two-fold. One is that there might be generated infinitely many different solution nodes. Another is that some lookup node might generate infinitely many child nodes so that extensions at other nodes right to the lookup node might be inhibited forever.

### (5) Soundness and Completeness of OLDT Resolution

Let $G$ be a negative clause. An *OLDT refutation* of $G$ in program $P$ is a refutation in the search tree of some extension of OLDT structure of $G$. The *answer substitution of the OLDT refutation* and the *solution of the OLDT refutation* are defined in the same way as before. Then, OLDT resolution is sound and complete. (Do not confuse the completeness of the general OLDT resolution with the incompleteness of the one under a specific extension strategy, e.g., the depth-first from-left-to-right strayegy.)

**Theorem 2.1 (Soundness and Completeness of OLDT Resolution)**
If $G\tau$ is a solution of an OLDT refutation of $G$ in $P$, its universal closure $\forall X_1 X_2 \cdots X_n$ $G\tau$ is a logical consequence of $P$.

If a universal closure $\forall Y_1 Y_2 \cdots Y_m$ $G\sigma$ is a logical consequence of $P$, there is $G\tau$ which is a solution of an OLDT refutation of $G$ in $P$ and $G\sigma$ is an instance of $G\tau$.

*Proof.* Though our hybrid interpretation is different from the original OLDT resolution by Tamaki and Sato [13] in two respects (see [7]), these differences do not affect the proof of the soundness and the completeness. See Tamaki and Sato [13] pp.93-94.

## 2.2 Modified Hybrid Interpretation of Logic Programs

In order to make the conceptual presentation of the hybrid interpretation simpler, we have not considered the details of how it is implemented. In particular, it is not obvious in the "immediate extension of OLDT structure"

(a) how we can know whether a new node is the end of a unit subrefutation starting from some solution node, and

(b) how we can obtain the solution of the unit subrefutation efficiently if any.
It is an easy solution to insert a special call-exit marker $[\![A_1, \theta]\!]$ between $B_1\theta$, $B_2\theta$, ..., $B_m\theta$ and $A_2\theta$, ..., $A_n\theta$ when a solution node is OLDT resolved using an m.g.u. $\theta$, and obtain the unit subrefutation of $A_1$ and its solution $A_1\tau$ when the leftmost of a new OLDT resolvent is the special call-exit marker $[\![A_1, \tau]\!]$. But, we will use the following modified framework. (Though such redefinition might be confusing, it is a little difficult to grasp the intuitive meaning of the modified framework without the explanation in Section 2.1.)

A *search tree* of OLDT structure in the modified framework is a tree with its nodes labelled with a pair of a (generalized) negative clause and a substitution. (We have said "generalized", because it might contain non-atoms, i.e., call-exit markers. The edges are not labelled with substitutions any more.) A *search tree of* $(G, \sigma)$ is a serach tree whose root node is labelled with $(G, \sigma)$. The clause part of each label is a sequence "$\alpha_1$, $\alpha_2$, ..., $\alpha_n$" consisting of either atoms in the body of the clauses in $P \cup \{G\}$ or *call-exit markers* of the form $[\![A, \sigma']\!]$. A *refutation of* $(G, \sigma)$ is a path in a search tree of $(G, \sigma)$ from the root to a node labelled with $(\Box, \tau)$. The *answer substitution of the refutation* is the substitution $\tau$, and the *solution of the refutation* is $G\tau$. A *solution table* and an *association* are defined in the same way as before.

An *OLDT structure* is a triple of a search tree, a solution table and an association. The relation between a node and its child nodes in search trees of OLDT structures is specified by the following modified OLDT resolution.

A node of OLDT structure $(Tr, Tb, As)$ labelled with ("$\alpha_1$, $\alpha_2$, ..., $\alpha_n$", $\sigma$) is said to be *OLDT resolvable* when it satisfies either of the following conditions:

(a) The node is a terminal solution node of $Tr$, and there is some definite clause "$B_0$ :- $B_1, B_2, \ldots, B_m$" $(m \geq 0)$ in program $P$ such that $\alpha_1\sigma$ and $B_0$ are unifiable, say by an m.g.u. $\theta$.

(b) The node is a lookup node of $Tr$, and there is some solution $B\tau$ in the associated solution list of the lookup-node such that (a fresh variant of) $B\tau$ is an instance of $\alpha_1\sigma$, say by an instantiation $\theta$.

The OLDT resolvent is obtained through the following two phases, called *calling phase* and *exiting phase* since they correspond to a "Call" (or "Redo") line and an "Exit" line in the messages of the conventional DEC10 Prolog tracer. A call-exit marker is inserted in the calling phase when a node is OLDT resolved using the program, while no call-exit marker is generated when a node is OLDT resolved using the solution table. When there is a call-exit marker at the leftmost of the clause part in the exiting phase, it means that some unit subrefutation is obtained.

(a) (Calling Phase) When a node labelled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\sigma$) is OLDT resolved, the intermediate label is generated as follows:

a-1. When the node is OLDT resolved using a definite clause "$B_0$ :- $B_1, B_2, \ldots, B_m$" in program $P$ and an m.g.u. $\theta$, the intermediate clause part is "$B_1$, $B_2$, ..., $B_m$, $[\![\alpha_1, \sigma]\!]$, $\alpha_2$, ..., $\alpha_n$", and the intermediate substitution part $\tau_0$ is $\theta$.

a-2. When the node is OLDT resolved using a solution $B\tau$ in the solution table and an instantiation $\theta$, the intermediate clause part is "$\alpha_2, \ldots, \alpha_n$", and the intermediate substitution part $\tau_0$ is $\sigma\theta$.

(b) (Exiting Phase) When there are $k$ call-exit markers $[\![A_1, \sigma_1]\!]$, $[\![A_2, \sigma_2]\!]$, ..., $[\![A_k, \sigma_k]\!]$ at the leftmost of the intermediate clause part, the label of the new node is generated as follows:

b-1. The clause part is obtained by eliminating all these call-exit markers. The substitution part is $\sigma_k \cdots \sigma_2\sigma_1\tau_0$.

b-2. Add $A_1\sigma_1\tau_0$, $A_2\sigma_2\sigma_1\tau_0$, ..., $A_k\sigma_k \cdots \sigma_1\tau_0$ to the last of the solution lists of $A_1\sigma_1$, $A_2\sigma_2$, ..., $A_k\sigma_k$, respectively, if they are not in the solution lists.

The precise algorithm is shown in Figure 2.2.1. The processing at the calling phase is performed in the first **case** statement, while that of the exiting phase is performed in the second **while** statement successively.

Note that, when a node is labelled with $(G, \sigma)$, the substitution part $\sigma$ always shows the instantiation of atoms to the left of the leftmost call-exit marker in $G$. When there is a call-exit marker $[\![A_j, \sigma_j]\!]$ at the leftmost of clause part in the exiting phase, we need to update the substitution part by composing $\sigma_j$ in order that the property above still holds after eliminating the call-exit marker. The sequence $\tau_1, \tau_2, \ldots, \tau_i$ denotes the sequence of updated substitutions. In addition, when we pass a call-exit marker $[\![A_j, \sigma_j]\!]$ in the **while** loop above with substitution $\tau_j$, the atom $A_j\tau_j$ denotes the solution of the unit subrefutation of $A_j\sigma_j$. The solution $A_j\tau_j$ is added to the solution list of $A_j\sigma_j$.

A node labelled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\sigma$) is a lookup node when a variant of atom $\alpha_1\sigma$ already exists as a key in the solution table, and is a solution node otherwise ($n \geq 1$).

```
OLDT-resolve(("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\sigma$) : label) : label ;
    $i$ := 0;
    case
      when a solution node is OLDT resolved with "$B_0$ :- $B_1, B_2, \ldots, B_m$" in $P$
          let $\theta$ be the m.g.u. of $\alpha_1\sigma$ and $B_0$ ;
          let $G_0$ be a negative clause "$B_1, B_2, \ldots, B_m, [\![\alpha_1, \sigma]\!], \alpha_2, \ldots, \alpha_n$" ;
          let $\tau_0$ be the substitution $\theta$ ;                                    — (A)
      when a lookup node is OLDT resolved with "$B\tau$" in $Tb$
          let $\theta$ be the instantiation of $\alpha_1\sigma$ to (a fresh variant of) $B\tau$ ;
          let $G_0$ be a negative caluse "$\alpha_2, \ldots, \alpha_n$" ;
          let $\tau_0$ be the composed substitution $\sigma\theta$ ;                     — (B)
    endcase
    while the leftmost of $G_i$ is a call-exit marker $[\![A_{i+1}, \sigma_{i+1}]\!]$ do
          let $G_{i+1}$ be $G_i$ other than the leftmost call-exit marker ;
          let $\tau_{i+1}$ be $\sigma_{i+1}\tau_i$ ;                                     — (C)
          add $A_{i+1}\tau_{i+1}$ to the last of $A_{i+1}\sigma_{i+1}$'s solution list if it is not in it ;
          $i$ := $i$ + 1 ;
    endwhile
    $(G_{new}, \sigma_{new})$ := $(G_i, \tau_i)$ ;
    return $(G_{new}, \sigma_{new})$.
```

**Figure 2.2.1 Modified Hybrid Interpretation**

The *initial OLDT structure* of $(G, \sigma)$ is a triple $(Tr_0, Tb_0, As_0)$, where $Tr_0$ is a search tree of $G$ consisting of just the root solution node labelled with $(G, \sigma)$, $Tb_0$ is a solution table consisting of just one entry whose key is the leftmost atom of $G$ and solution list is [ ], and $As_0$ is the empty set of pointers. The *immediate extension of OLDT structure*, *extension of OLDT structure*, *answer substitution of OLDT refutation* and *solution of OLDT refutation* are defined in the same way as before.

*Example 2.2* Consider the example in Section 2.1 again. The modified hybrid interpretation generates the following OLDT structures of $reach(a, Y_0)$.

First, the initial OLDT structure below is generated. Now, the root node is labelled with ("$reach(a, Y_0)$",<>).

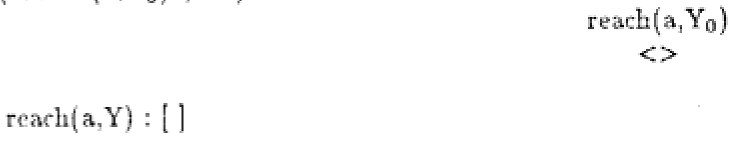


**Figure 2.2.2 Modified Hybrid Interpretation at Step 1**

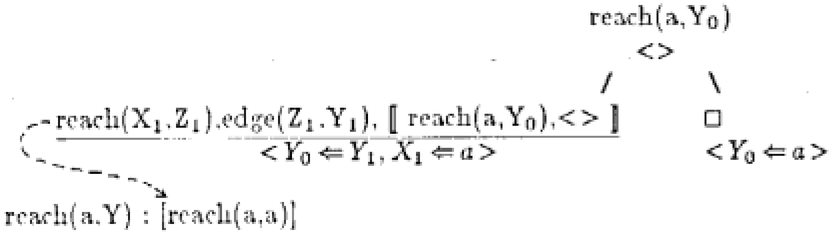
Secondly, the root node ("$reach(a, Y_0)$",<>) is OLDT resolved using the program to generate two child nodes. The intermediate label of the left child node is

("$reach(X_1, Z_1), edge(Z_1, Y_1), [\![reach(a, Y_0), <> ]\!]$", $< Y_0 \Leftarrow Y_1, X_1 \Leftarrow a >$).

It is the new label immediately, since its leftmost is not a call-exit marker. The intermediate label of the right child node is

("$[\![reach(a, Y_0), <> ]\!]$", $< Y_0 \Leftarrow a, X_1 \Leftarrow a >$).

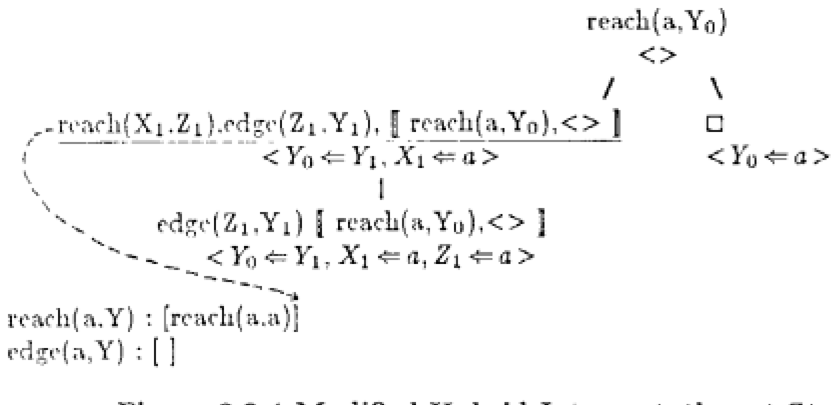
By eliminating the leftmost call-exit marker and composing the substitution, the new label is ($\square$, $< Y_0 \Leftarrow a, X_1 \Leftarrow a >$). (When the clause part of the label is $\square$, we will omit the assignments irrelevant to the top-level goal in the following figures, e.g., $< X_1 \Leftarrow a >$.) During the elimination of the call-exit marker, $reach(a, a)$ is added to the solution table.



reach(a,Y₀)
<>
reach(X₁,Z₁),edge(Z₁,Y₁), [ reach(a,Y₀),<> ]
<Y₀ ⇐ Y₁, X₁ ⇐ a>
□
<Y₀ ⇐ a>
reach(a,Y) : [reach(a,a)]

**Figure 2.2.3 Modified Hybrid Interpretation at Step 2**

Thirdly, the left lookup node is OLDT resolved using the solution table to generate one child solution node.



reach(a,Y₀)
<>
reach(X₁,Z₁),edge(Z₁,Y₁), [ reach(a,Y₀),<> ]
<Y₀ ⇐ Y₁, X₁ ⇐ a>
□
<Y₀ ⇐ a>
edge(Z₁,Y₁) [ reach(a,Y₀),<> ]
<Y₀ ⇐ Y₁, X₁ ⇐ a, Z₁ ⇐ a>
reach(a,Y) : [reach(a,a)]
edge(a,Y) : [ ]

**Figure 2.2.4 Modified Hybrid Interpretation at Step 3**

Fourthly, the generated solution node is OLDT resolved using a unit clause "$edge(a, b)$" in program $P$ to generate the intermediate label

("$[\![edge(Z_1, Y_1), < Y_0 \Leftarrow Y_1, X_1 \Leftarrow a, Z_1 \Leftarrow a > ]\!], [\![reach(a, Y_0), <> ]\!]$", $< Y_1 \Leftarrow b >$).

By eliminating the leftmost call-exit markers and composing substitutions, the new label is $(\Box, <Y_0 \Leftarrow b, X_1 \Leftarrow a,\ Z_1 \Leftarrow a, Y_1 \Leftarrow b>)$. During the elimination of the call-exit markers, $edge(a,b)$ and $reach(a,b)$ are added to the solution table.

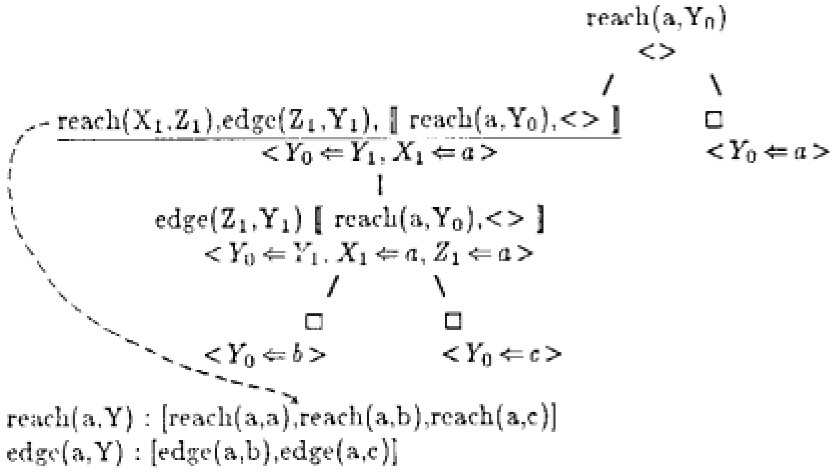
Similarly, the node is OLDT resolved using a unit clause "$edge(a,c)$" in program $P$ to generate the intermediate label

("$[\![edge(Z_1,Y_1), <Y_0 \Leftarrow Y_1, X_1 \Leftarrow a, Z_1 \Leftarrow a>]\!], [\![reach(a,Y_0), <>]\!]$", $<Y_1 \Leftarrow c>$)

By eliminating the leftmost call-exit markers and composing substitutions similarly, the new label is $(\Box, <Y_0 \Leftarrow c, X_1 \Leftarrow a,\ Z_1 \Leftarrow a, Y_1 \Leftarrow c>)$. This time, $edge(a,b)$ and $reach(a,b)$ are added to the solution table during the elimination of the call-exit markers.

The process of extension proceeds similarly to obtain all the solutions as in Example 2.1.2.

*Remark.* Note that we no longer need to keep the edges and the non-terminal solution nodes of search trees. In addition, we can throw away assignments in $\theta$ for the variables in $B\tau$ at step (B), and those in $\tau_i$ for variables not in $A_{i+1}\sigma_{i+1}$ at step (C) in Figure 2.2.1.



reach(a,Y₀)
<>
reach(X₁,Z₁),edge(Z₁,Y₁), ⟦ reach(a,Y₀),<> ⟧
<Y₀ ⇐ Y₁, X₁ ⇐ a>
□
<Y₀ ⇐ a>
edge(Z₁,Y₁) ⟦ reach(a,Y₀),<> ⟧
<Y₀ ⇐ Y₁, X₁ ⇐ a, Z₁ ⇐ a>
□
<Y₀ ⇐ b>
□
<Y₀ ⇐ c>
reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c)]
edge(a,Y) : [edge(a,b),edge(a,c)]

**Figure 2.2.5 Modified Hybrid Interpretation at Step 4**

## 3. Mode Analysis by Abstract Hybrid Interpretation

Suppose we would like to know that, when a goal "$isort(L_0,M)$" with its first argument $L_0$ instantiated to a ground term succeeds, the form of its second argument $M$ after the success is unique, where *isort* and *insertp* are defined by

```
isort([ ],[ ]).
isort([X|L],M) :- isort(L,N), insertp(X,N,M).
insertp(X,[ ],[X]).
insertp(X,[Y|N],[X,Y|M]) :- X≤Y.
insertp(X,[Y|N],[Y|M]) :- X>Y, insertp(X,N,M).
```

When the first clause (unit clause $isort([\ ],[\ ])$) is used first to succeed, the desired functionality is obvious. When the second clause is used first, the goal is reduced to "$isort(L_1,N_1)$, $insertp(X_1,N_1,M)$". If we can assume the functionality for $isort(L_1,N_1)$ when its first argument $L_1$ is instantiated to a ground term, the detection of the desired functionality is reduced to that of $insertp(X_1,N_1,M)$. It is, however, crucial to know that the second argument $N_1$ is instantiated to a ground term after the first subgoal $isort(L_1,N_1)$ succeeds in order to know that the form of $M$ after the success of $insertp(X_1,N_1,M)$ is unique. More generally, the first argument of *isort* invoked from the top-level goal is always a ground term at calling time and the second argument is always a ground term at exiting time. Similarly, so are the first and the second arguments of *insertp* at calling time, and the third argument at exiting time. How can we show it mechanically?

In this section, we will reformulate the work by Mellish [10],[11] and Debray [4] from the point of view in Section 2.2.

## 3.1 Mode Analysis

A *mode* is one of the following 3 sets of terms:

$\underline{any}$ : the set of all terms,
$\underline{ground}$ : the set of all ground terms,
$\underline{\emptyset}$ : the emptyset of terms.

Modes are ordered by the instantiation ordering $\preceq$ depicted below.

$$\begin{tikzcd}
\underline{\emptyset} \arrow[d, dash] \\
\underline{ground} \arrow[d, dash] \\
\underline{any}
\end{tikzcd}$$

Note that this is the reverse of the set inclusion ordering below.

$$\begin{tikzcd}
\underline{any} \arrow[d, dash] \\
\underline{ground} \arrow[d, dash] \\
\underline{\emptyset}
\end{tikzcd}$$

A *mode substitution* is an expression of the form
$< X_1 \Leftarrow \underline{m_1}, X_2 \Leftarrow \underline{m_2}, \ldots, X_l \Leftarrow \underline{m_l} >$,
where $\underline{m_1}, \underline{m_2}, \ldots, \underline{m_l}$ are modes. The mode assigned to variable $X$ by mode substitution $\mu$ is denoted by $\mu(X)$. We stipulate that a mode substitution assigns $\underline{any}$, the minimum element w.r.t. the instantiation ordering, to variable $X$ when $X$ is not in the domain of the mode substitution explicitly. Hence the empty mode substitution $<>$ assigns $\underline{any}$ to every variable.

Let $A$ be an atom in the body of some clause in $P \cup \{G\}$, $\mu$ be a mode substitution of the form
$< X_1 \Leftarrow \underline{m_1}, X_2 \Leftarrow \underline{m_2}, \ldots, X_l \Leftarrow \underline{m_l} >$.
Then $A\mu$ is called a *mode-abstracted atom*, and denotes the set of all atoms obtained by replacing each $X_i$ in $A$ with a term in $\underline{m_i}$. Two mode-abstracted atoms $A\mu$ and $B\nu$ are said to be *unifiable* when $A\mu \cap B\nu \neq \underline{\emptyset}$. A list of mode-abstracted atoms $[A_1\mu_1, A_2\mu_2, \ldots, A_n\mu_n]$ denotes the set union $\cup_{i=1}^{n} A_i\mu_i$. Similarly, $G\mu$ (or the pair $(G, \mu)$) is called a *mode-abstracted negative clause*, and denotes the set of negative clauses obtained by replacing each $X_i$ in $G$ with a term in $\underline{m_i}$. When $G$ is of the form "$A_1, A_2, \ldots, A_n$", the mode-abstracted atom $A_1\mu$ is called the leftmost mode-abstracted atom of $G\mu$.

The purpose of mode analysis is to obtain possible mode patterns of goals appearing in the top-down execution of a given goal. Let us formalize the top-down execution here. The top-down Prolog interpreter is modeled by OLD resolution. The OLD resolution is defined using just search trees, called OLD trees. (Because there is neither a solution table nor an association, we have no distinction of solution nodes and lookup nodes. All nodes are solution nodes.) The relation between a node and its child nodes in OLD trees is specified in the same way as the OLDT resolution in Section 2.1, except that we have no resolution using lookup nodes and solution tables, hence no manipulation of solution tables and associations.

An atom $A$ appearing at the leftmost of the label of a node in some OLD tree of $G$ is called *calling pattern of* $G$. Note that any calling pattern of $G$ is an instance of some atom in the body of some clause in $P \cup \{G\}$. Each calling pattern corresponds to some key in the solution table of OLDT structure.

A solution $A\tau$ of a subrefutation in an OLD tree of $G$ is called an *exiting pattern of* $G$. Note that any exiting pattern of $G$ is also an instance of some atom in the body of some clause in $P \cup \{G\}$. Each exiting pattern corresponds to some element in the solution lists of OLDT structure.

Let $G\mu$ be a mode-abstracted negative clause, $\mathcal{C}(G\mu)$ be the set of all calling patterns of negative clauses in $G\mu$ and $\mathcal{E}(G\mu)$ be the set of all exiting patterns of negative clauses in $G\mu$. The *mode analysis* w.r.t. $G\mu$ is the problem to compute

(a) some list of mode-abstracted atoms which is a superset of $\mathcal{C}(G\mu)$,
(b) some list of mode-abstracted atoms which is a superset of $\mathcal{E}(G\mu)$.

*Remark.* We have adopted the simplest mode structure consisting of just 3 modes $\underline{any}$, $\underline{ground}$, $\emptyset$ (cf. Section 3 of [4]). In order to include an additional mode $\underline{variable}$ representing the set of all variables, we need to take one more quantity (called *sharing*) into consideration to infer modes correctly. (See Section 6 of [7] for details.) But, these 3 modes are enough for the functionality detection in Section 4.

## 3.2 Abstract Hybrid Interpretation for Mode Analysis

### 3.2.1 OLDT Structure for Mode Analysis

A *search tree for mode analysis* is a tree with its nodes labeled with a pair of a (generalized) negative clause and a mode substitution. (For brevity, we will sometimes omit the term "for mode analysis" hereafter in Section 3.) A *search tree of* $(G, \mu)$ is a search tree whose root node is labeled with $(G, \mu)$. The clause part of each label is a sequence "$\alpha_1$, $\alpha_2$, ..., $\alpha_n$" consisting of either atoms in the body of some clause in $P \cup \{G\}$ or call-exit markers of the form $[\![A, \mu', \eta]\!]$. A *refutation* of $(G, \mu)$ is a path in a search tree of $(G, \mu)$ from the root to a node labelled with $(\Box, \nu)$. The *answer substitution of the refutation* is the mode substitution $\nu$ and the *solution of the refutation* is $G\nu$.

A *solution table for mode analysis* is a set of entries. Each entry consists of the *key* and the *solution list*. The key is a mode-abstracted atom. The solution list is a list of mode-abstracted atoms, called *solutions*, whose all solutions are greater than the key w.r.t. the instantiation ordering.

Let $Tr$ be a search tree whose nodes labeled with non-null clauses are classified into either *solution nodes* or *lookup nodes*, and let $Tb$ be a solution table. An *association for mode analysis* of $Tr$ and $Tb$ is a set of pointers pointing from each lookup node in $Tr$ into some solution list in $Tb$ such that the leftmost mode-abstracted atom of the lookup node' label and the key of the solution list are variants of each other.

An *OLDT structure for mode analysis* is a triple of search tree, solution table and association. The relation between a node and its child nodes in a search tree is specified by *OLDT resolution for mode analysis* in Section 3.2.3.

### 3.2.2 Overestimation of Modes

Because the purpose of mode analysis is to compute supersets of the sets of calling patterns and exiting patterns using lists of mode-abstracted atoms, we need to overestimate them somehow by manipulating mode-abstracted atoms. We would like to do it by specifying the operations for mode analysis corresponding to those at step (A),(B) and (C) in Figure 2.2.1. In order to specify them, we need to consider the following situation: Let $A$ be an atom, $X_1, X_2, \ldots, X_k$ all the variables in $A$, $\mu$ a mode substitution of the form
$<X_1 \Leftarrow \underline{m_1}, X_2 \Leftarrow \underline{m_2}, \ldots, X_k \Leftarrow \underline{m_k}>$,
$B$ an atom, $Y_1, Y_2, \ldots, Y_l$ all the variables in $B$, and $\nu$ a mode substitution of the form
$<Y_1 \Leftarrow \underline{n_1}, Y_2 \Leftarrow \underline{n_2}, \ldots, Y_l \Leftarrow \underline{n_l}>$,
Then

(a) How can we know whether $A\mu$ and $B\nu$ are unifiable, i.e., whether there is an atom in $A\mu \cap B\nu$?

(b) If there is such an atom $A\sigma = B\tau$, what terms are expected to be assigned to each $Y_j$ by $\tau$?

*Example 3.2.2.1* There is a common atom of
$p(X, Y) <X \Leftarrow \underline{ground}, Y \Leftarrow \underline{any}>$,
$p(f(U), g(V)) <U \Leftarrow \underline{any}, V \Leftarrow \underline{any}>$.
Hence, they are unifiable. Let $p(f(U), g(V))\tau$ be a common atom. Then $U$ must be instantiated to a term in $\underline{ground}$, and $V$ must be instantiated to a term in $\underline{any}$.

#### (1) Overestimation of Unifiability

When two mode-abstracted atoms $A\mu$ and $B\nu$ are unifiable, two atoms $A$ and $B$ must be unifiable in the usual sense. Let $\eta$ be an m.g.u of $A$ and $B$ of the form
$<X_1 \Leftarrow t_1, X_2 \Leftarrow t_2, \ldots, X_k \Leftarrow t_k, Y_1 \Leftarrow s_1, Y_2 \Leftarrow s_2, \ldots, Y_l \Leftarrow s_l>$.
If we can overestimate the mode assigned to each occurrence of $Z$ in $t_i$ from the mode substitution $\mu$ and that of $Z$ in $s_j$ from the mode substitution $\nu$, we can overestimate the mode assigned to the variable $Z$ by taking the join $\vee$ w.r.t. the instantiation ordering for all occurrences of $Z$. If it is the emptyset $\emptyset$ for some variable $Z$, we can't expect that there exists a common atom $A\sigma = B\tau$ in $A\mu \cap B\nu$.

A mode containing all instances of some occurrence of $Z$ when an instance of term $t[Z]$ is in mode $\underline{m}$ is denoted by $Z/ < t[Z] \Leftarrow m >$. Due to the choice of modes (see [4]), it is computed simply as follow:

$$Z/ < t[Z] \Leftarrow \underline{m} > = \underline{m}.$$

*Example 3.2.2.2* Let $t$ be $[X|L]$ and $\underline{m}$ be *ground*. Then
$X/ < [X|L] \Leftarrow \underline{ground} > = \underline{ground}, \quad L/ < [X|L] \Leftarrow \underline{ground} > = \underline{ground}.$
Let $t$ be $[X|L]$ and $\underline{m}$ be $\underline{any}$. Then
$X/ < [X|L] \Leftarrow \underline{any} > = \underline{any}, \quad L/ < [X|L] \Leftarrow \underline{any} > = \underline{any}.$

Note that $Z/ < t[Z] \Leftarrow \underline{m} >$ is not $\emptyset$ when $m$ is not $\emptyset$. Because the join $\vee$ of non-empty modes w.r.t. the instantiation ordering is always non-empty, the mode assigned to each variable is non-empty when $\mu$ and $\nu$ do not assign $\emptyset$ to any variable. This means that the unifiability of $A\mu$ and $B\nu$ can be reduced to the unifiability of $A$ and $B$.

## (2) One Way Propagation of Mode Substitutions

Recall the situation we are considering. First, we will restrict our attentions to the case where $\nu = <>$ first. Suppose there is an atom $A\sigma = B\tau$ in $A\mu \cap B <>$. Then, what terms are expected to be assigned to variables in $B$ by $\tau$?

As has been just shown, we can overestimate the mode assigned to each variable $Z$ in $t_i$ from the mode substitution $\mu$. By collecting these modes assignment for all variables, we can overestimate the mode substitution $\lambda$ for the variables in $t_1, t_2, \ldots, t_k$. If we can overestimate the mode assigned to $s_j$ from the mode substitution $\lambda$ obtained above, we can obtain the mode substitution $\nu'$

$< Y_1 \Leftarrow \underline{n'_1}, Y_2 \Leftarrow \underline{n'_2}, \ldots, Y_l \Leftarrow \underline{n'_l} >$

by collecting the modes for all variables $Y_1, Y_2, \ldots, Y_l$.

Let $\lambda$ be a mode substitution. A mode containing all instances of $s$ when each variable $X$ is assigned a term in mode $\lambda(X)$ is denoted by $s/\lambda$, and computed as follows:

$$s/\lambda = \begin{cases} \emptyset, & \lambda(X) = \emptyset \text{ for some } X \text{ in } s; \\ \underline{ground}, & \text{when } \lambda(X) = \underline{ground} \text{ for every variable } X \text{ in } s; \\ \underline{any}, & \text{otherwise.} \end{cases}$$

*Example 3.2.2.3* Let $s$ be $[X|L]$ and $\lambda$ be $<X \Leftarrow \underline{ground}, L \Leftarrow \underline{ground}>$. Then

$s/\lambda = \underline{ground}$.

Let $s$ be $[X|L]$ and $\lambda$ be $<X \Leftarrow \underline{any}, L \Leftarrow \underline{ground}>$. Then

$s/\lambda = \underline{any}$.

Let $A$, $B$ be atoms, $\mu$ a mode-substitution for the variables in $A$, and $\eta$ an m.g.u. of $A$ and $B$. The mode-substitution for the variables in $B$, that is obtained from $\mu$ and $\eta$ using $Z/ < t[Z] \Leftarrow \underline{t} >$ and $s/\lambda$ above, is denoted by $propagate(\mu, \eta)$. (Note that $propagate(\mu, \eta)$ depends on just $\mu$ and $\eta$.)

## (3) Overestimation of Mode Substitutions

As for the operation at step (A) for mode analysis, we can adopt the one way propagation

$$propagate(\mu, \eta)$$

since the destination side mode substitution is $<>$. As for the operations at step (B) and (C) for mode analysis, where the destination side mode substitution is not necessarily $<>$, we can adopt the join $\vee$ w.r.t. the instantiation ordering

$$\mu \vee propagate(\nu, \eta),$$

i.e., variable-wise join of the mode assigned by the previous mode substitution $\mu$ and the one by the one-way propagation $propagate(\nu, \eta)$.

*Example 3.2.2.4* Let $\nu$ and $propagate(\mu, \eta)$ be mode substitutions :

$< X_1 \Leftarrow \underline{any}, Y_1 \Leftarrow \underline{any} >$,

$< X_1 \Leftarrow \underline{ground}, Y_1 \Leftarrow \underline{any} >$.

Then, $\nu \vee propagate(\mu, \eta)$ is a mode substitution

$< X_1 \Leftarrow \underline{ground}, Y_1 \Leftarrow \underline{any} >$.

## 3.2.3 OLDT Resolution for Mode Analysis

The relation between a node and its child nodes in a search tree is specified by *OLDT resolution for mode analysis* as follows:

A node of OLDT structure $(Tr, Tb, As)$ labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\mu$) is said to be *OLDT resolvable* $(n \geq 1)$ when it satisfies either of the following conditions:

(a) The node is a terminal solution node of $Tr$, and there is some definite clause "$B_0$ :- $B_1, B_2, \ldots, B_m$" $(m \geq 0)$ in program $P$ such that $\alpha_1$ and $B_0$ is unifiable, say by an m.g.u. $\eta$.

(b) The node is a lookup node of $Tr$, and there is some mode-abstracted atom $B\nu$ in the associated solution list of the lookup node such that $\alpha_1$ and $B$ are variants of each other. Let $\eta$ be the renaming of $B$ to $\alpha_1$.

The precise algorithm of OLDT resolution for mode analysis is shown in Figure 3.2.3. Note that the operations at steps (A), (B) and (C) in Figure 2.2.1 are modified.

```
OLDT-resolve(("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\mu$) : label) : label ;
    $i := 0$ ;
    case
      when a solution node is OLDT resolved with "$B_0$ :- $B_1, B_2, \ldots, B_m$" in $P$
          let $\eta$ be the m.g.u. of $\alpha_1$ and $B_0$ ;
          let $G_0$ be a negative clause "$B_1, B_2, \ldots, B_m, [\![\alpha_1, \mu, \eta]\!], \alpha_2, \ldots, \alpha_n$" ;
          let $\nu_0$ be $propagate(\mu, \eta)$ ;                                    — (A)
      when a lookup node is OLDT resolved with "$B\nu$" in $Tb$
          let $\eta$ be the renaming of $B$ to $\alpha_1$ ;
          let $G_0$ be a negative caluse "$\alpha_2, \ldots, \alpha_n$" ;
          let $\nu_0$ be $\mu \vee propagate(\nu, \eta)$ ;                           — (B)
    endcase
    while the leftmost of $G_i$ is a call-exit marker $[\![A_{i+1}, \mu_{i+1}, \eta_{i+1}]\!]$ do
          let $G_{i+1}$ be $G_i$ other than the leftmost call-exit marker ;
          let $\nu_{i+1}$ be $\mu_{i+1} \vee propagate(\nu_i, \eta_{i+1})$ ;          — (C)
          add $A_{i+1}\nu_{i+1}$ to the last of $A_{i+1}\mu_{i+1}$'s solution list if it is not in it ;
          $i := i + 1$ ;
    endwhile
    $(G_{new}, \mu_{new}) := (G_i, \nu_i)$ ;
    return $(G_{new}, \mu_{new})$.
```

**Figure 3.2.3 OLDT Resolution for Mode Analysis**

A node labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\mu$) is a lookup node when a variant of $\alpha_1\mu$ is a key in the solution table, and is a solution node otherwise $(n \geq 1)$.

The *initial OLDT structure*, *immediate extension of OLDT structure*, *extension of OLDT structure*, *answer substitution of OLDT refutation* and *solution of OLDT refutation* are defined in the same way as in Section 2.2.

## 3.3 Examples of Mode Analysis

We will show two examples of mode analysis, which are used again in Section 4.

*Example 3.3.1* Let *isort*, *insertp*, $\leq$ and $<$ be predicates defined by

```
isort([ ],[ ]).
isort([X|L],M) :- isort(L,N), insertp(X,N,M).
insertp(X,[ ],[X]).
insertp(X,[Y|N],[X,Y|N]) :- X≤Y.
insertp(X,[Y|N],[Y|M]) :- X>Y, insertp(X,N,M).
0≤X.
suc(X)≤suc(Y) :- X≤Y.
suc(X)>0.
suc(X)>suc(Y) :- X>Y.
```

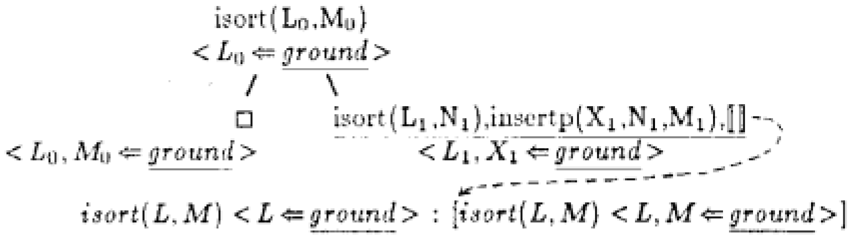
Then, extension of OLDT structure proceeds similarly to *reach* in Example 2.2. (This *isort* is the "insertion sort".)

First, the initial OLDT structure below is generated. The root node of the search tree is a solution node. The solution table contains only one entry with its key $isort(L,M) < L \Leftarrow \underline{ground}>$ and its solution list is [ ].

$$\begin{array}{c} \text{isort}(L_0,M_0) \\ < L_0 \Leftarrow \underline{ground} > \end{array}$$

$$isort(L,M) < L \Leftarrow \underline{ground}> \ : \ [\ ]$$

**Figure 3.3.1. Mode Analysis at Step 1**



isort(L0,M0)
< L0 ⇐ ground >
□
< L0, M0 ⇐ ground >
isort(L1,N1),insertp(X1,N1,M1),[]
< L1, X1 ⇐ ground >
isort(L, M) < L ⇐ ground > : [isort(L, M) < L, M ⇐ ground >]

**Figure 3.3.2. Mode Analysis at Step 2**

Secondly, the root node is OLDT resolved using the program to generate two child nodes. Since the intermediate clause part of the left child node is

"$[isort(L_0,M_0), < L_0 \Leftarrow \underline{ground}>, < L_0, M_0 \Leftarrow [\ ]> ]$",

the node is the end of a unit subrefutation. Its solution $isort(L_0,M_0) < L_0, M_0 \Leftarrow \underline{ground}>$ is added to the solution list. Since the intermediate clause part of the right child node is

"$isort(L_1,N_1),\ insertp(X_1,N_1,M_1),$

$[isort(L_0,M_0), < L_0 \Leftarrow \underline{ground}>, < L_0 \Leftarrow [X_1|L_1], M_0 \Leftarrow M_1 > ]$",

it is immediately the clause part of the generated node. The generated node is a lookup node, because the leftmost mode-abstracted atom is a variant of that of the root node. The association associates the lookup node to the head of the solution list of $isort(L,M) < L \Leftarrow$

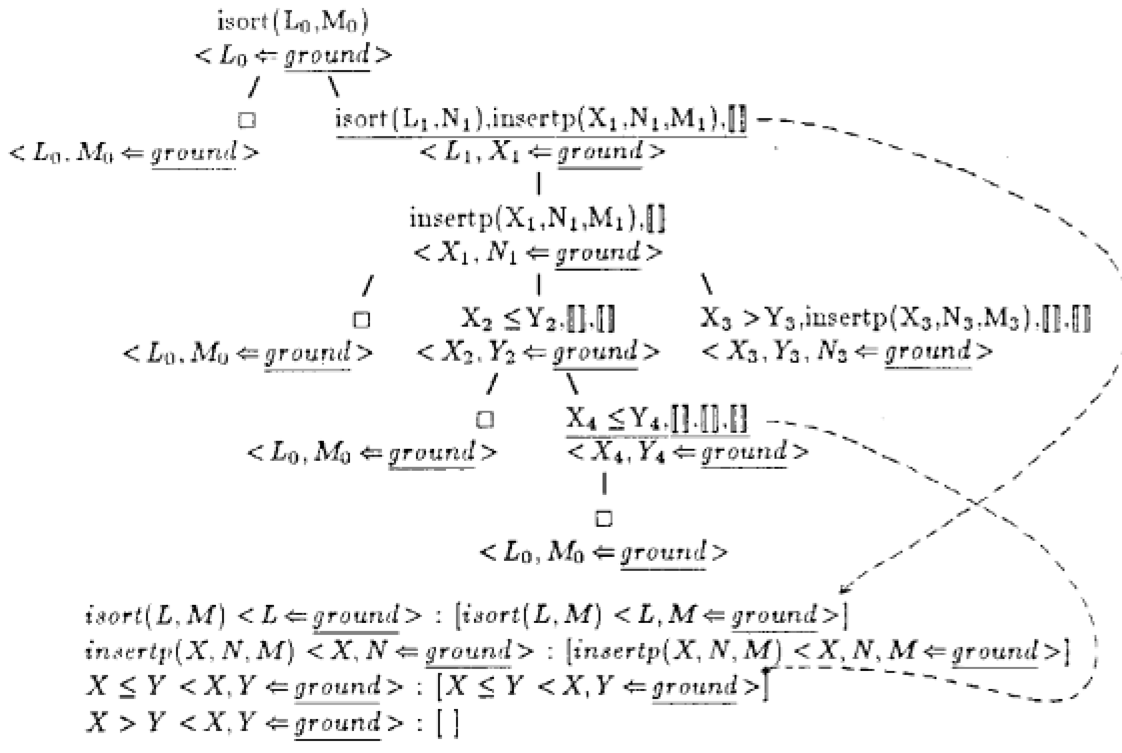
$\underline{ground}>$. (From now on, the quantities inside call-exit markers are omitted due to space limit so that they are depicted simply by [].)

Thirdly, the lookup node is OLDT resolved using the solution table to generate one child solution node. The association associates the lookup node to the last of the solution list.

Fourthly, the new solution node is OLDT resolved using the program to generate three child nodes. The generated left child node adds a solution $insertp(X_2, N_2, M_2) < X_2, N_2, M_2 \Leftarrow \underline{ground} >$ to the solution table. The generated center child node and right child node are solution nodes.

Fifthly, the center solution node is OLDT resolved using the program to generate two child nodes. The generated left child node labeled with the null clause gives three solutions, of which a new solution $X_3 \le Y_3 < X_3, Y_3 \Leftarrow \underline{ground}>$ is added to the solution table. The generated right child node is a lookup node. The association associates the right lookup node to the head of the solution list of $X \le Y < X, Y \Leftarrow \underline{ground}>$.

Sixthly, the lookup node is OLDT resolved using the solution table to geneate one child node. The generated child node labeled with the null clause □ gives three solutions, all of which are already in the solution table.



$isort(L, M) < L \Leftarrow \underline{ground}>$ : $[isort(L, M) < L, M \Leftarrow \underline{ground}>]$
$insertp(X, N, M) < X, N \Leftarrow \underline{ground}>$ : $[insertp(X, N, M) < X, N, M \Leftarrow \underline{ground}>]$
$X \le Y < X, Y \Leftarrow \underline{ground}>$ : $[X \le Y < X, Y \Leftarrow \underline{ground}>]$
$X > Y < X, Y \Leftarrow \underline{ground}>$ : [ ]

**Figure 3.3.3. Mode Analysis at Step 6**

The process proceeds in the same way. Lastly at step 10, all nodes are OLDT resolved up. The search tree below the rightmost solution node in Figure 3.3.3 and the final solution table are as in Figure 3.3.4. The final solution table says that

(a) *isort* is always called with its first argument instantiated to a ground term. *insertp* is always called with its first and second arguments instantiated to ground terms. $\le$ and $>$ are always called with their arguments instantiated to ground terms.

(b) When *isort* succeeds, its first and second arguments are instantiated to ground terms. So are all arguments of *insertp*, $\leq$ and $>$.

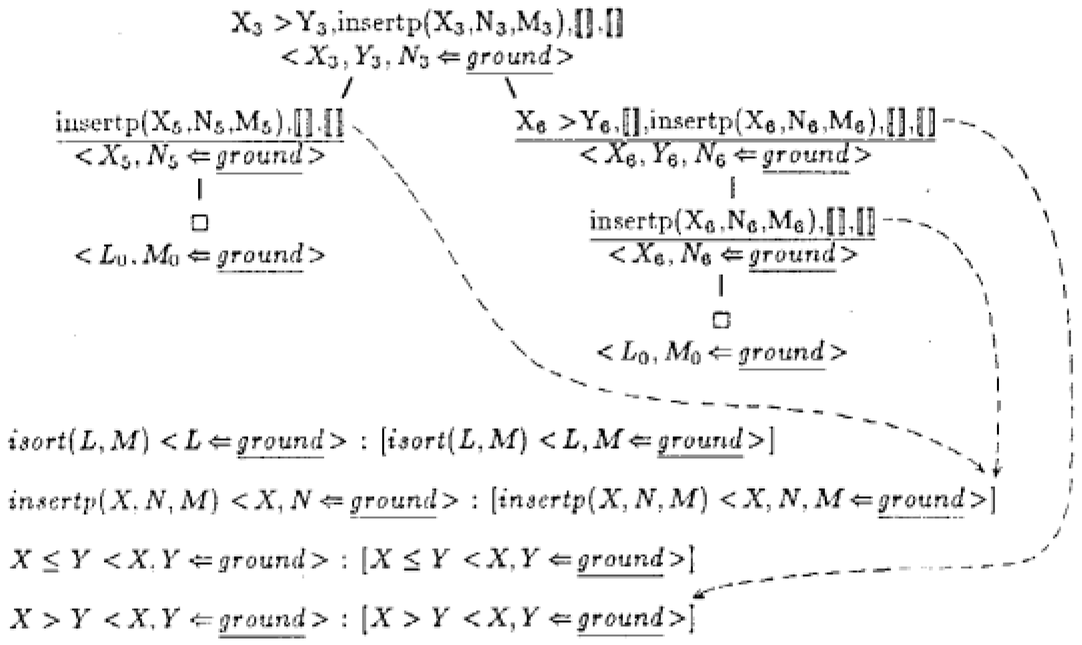

X3 >Y3,insertp(X3,N3,M3),[],[]
< X3, Y3, N3 ⇐ ground >
insertp(X5,N5,M5),[],[]
< X5, N5 ⇐ ground >
< L0, M0 ⇐ ground >
X6 >Y6,[],insertp(X6,N6,M6),[],[]
< X6, Y6, N6 ⇐ ground >
insertp(X6,N6,M6),[],[]
< X6, N6 ⇐ ground >
< L0, M0 ⇐ ground >
isort(L,M) < L ⇐ ground > : [isort(L,M) < L,M ⇐ ground >]
insertp(X,N,M) < X,N ⇐ ground > : [insertp(X,N,M) < X,N,M ⇐ ground >]
X ≤ Y < X,Y ⇐ ground > : [X ≤ Y < X,Y ⇐ ground >]
X > Y < X,Y ⇐ ground > : [X > Y < X,Y ⇐ ground >]

**Figure 3.3.4. Mode Analysis at Step 10**

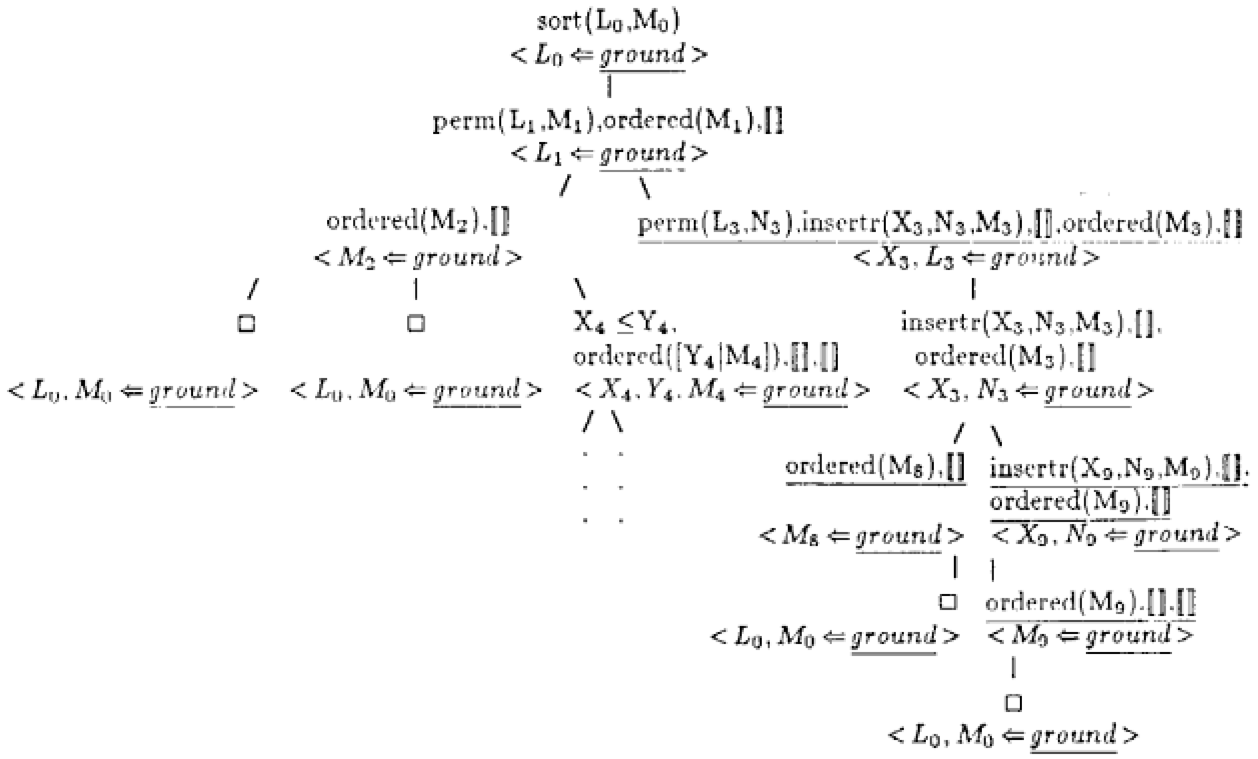


**Figure 3.3.5. Mode Analysis of *sort***

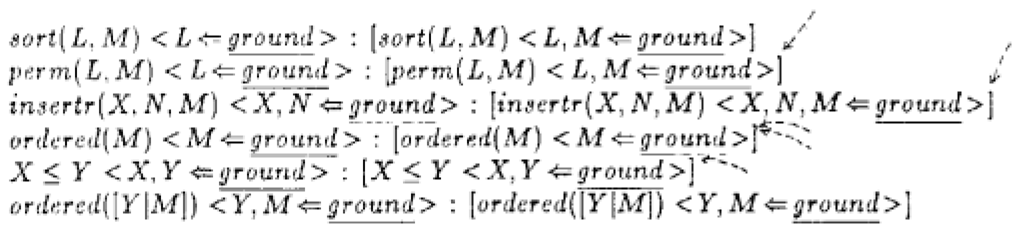

$sort(L,M) < L \Leftarrow \underline{ground} > : [sort(L,M) < L,M \Leftarrow \underline{ground} >]$
$perm(L,M) < L \Leftarrow \underline{ground} > : [perm(L,M) < L,M \Leftarrow \underline{ground} >]$
$insertr(X,N,M) < X,N \Leftarrow \underline{ground} > : [insertr(X,N,M) < X,N,M \Leftarrow \underline{ground} >]$
$ordered(M) < M \Leftarrow \underline{ground} > : [ordered(M) < M \Leftarrow \underline{ground} >]$
$X \leq Y < X,Y \Leftarrow \underline{ground} > : [X \leq Y < X,Y \Leftarrow ground >]$
$ordered([Y|M]) < Y,M \Leftarrow \underline{ground} > : [ordered([Y|M]) < Y,M \Leftarrow \underline{ground} >]$

**Figure 3.3.5. Mode Analysis of** *sort* (Continued)

*Example 3.3.2* Let *sort*, *perm*, *insertr* and *ordered* be predicates defined by

```
sort(L,M) :- perm(L,M),ordered(M).
perm([ ],[ ]).
perm([X|L],M) :- perm(L,N), insertr(X,N,M).
insertr(X,N,[X|N]).
insertr(X,[Y|N],[Y|M]) :- insertr(X,N,M).
ordered([ ]).
ordered([X]).
ordered([X,Y|M]) :- X≤Y, ordered([Y|M]).
0≤Y.
suc(X)≤suc(Y) :- X≤Y.
```

Then the OLDT resolution for mode analysis generates the OLDT structure in Figure 3.3.5. (This *sort* is a specification of sorting.)

## 4. Functionality Detection Based on Mode Analysis

Because variables in Prolog are freely instantiatable, we need to be a little careful in defining the functionality of Prolog programs. In this paper, we will define it assuming mode information. (See Section 5 for another definition assuming type information.) Then, functionality of a given mode-abstracted atom can be detected by overestimating the number of its solution for each atom satisfying the mode restriction. If the number is guaranteed to be at most 1, we can conclude that the mode-abstracted atom is functional.

### 4.1 Overestimating Solution Numbers

A mode-abstracted atom $A\mu$ is said to be *functional*, if, when any goal $A\sigma$ in $A\mu$ succeeds with its form $A\tau$, the atom $A\tau$ is unique up to renaming of variables, that is, the input form (the form at calling time) $A\sigma$ uniquely determines the output form (the form at exiting time) $A\tau$. A mode-abstracted atom $A\mu$ is said to be *relational* otherwise. (This is a definition obtained by simplifying the one by Debray and Warren [3].)

This definition of functionality is different from the usual definition of *determinacy* (cf. [10],[11],[12]). When the execution of a goal never succeeds in two different ways, i.e., there is at most one OLD refutation of the goal, the goal is said to be *deterministic*. (Hence, even if the backtracking is forced after the first success, it never succeeds twice.) The following example is due to Debray and Warren [3]. Let *again-and-again* be a predicate defined by

```
again-and-again(a).
again-and-again(X) :- again-and-again(X).
```

When the execution of a goal *again-and-again*$(X)$ succeeds using the first clause, the argument must be "$a$". When the backtracking is forced after the success, the second clause is used to recurse. When the execution succeeds again, the argument must be "$a$" again. Hence, *again-and-again*$(X)$ is not deterministic, but functional.

## (1) Solution Number

In order to detect functionality for a given mode-abstracted atom, we need to count the number of solutions somehow during the mode analysis process in Section 3. But, it is difficult to exactly count the numbers of solutions. Moreover, it is unnecessary for functionality detection to know whether the solution number is 2 or 3. Our additional domain of abstract interpretation consists of the following three elements:

2 : more than 2
1 : 1
0 : 0

Three operations $+$, $\times$ and *max* are defined for these elements as follows:

| $+$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 1 | 2 |
| 1 | 1 | 2 | 2 |
| 2 | 2 | 2 | 2 |

| $\times$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 |
| 2 | 0 | 2 | 2 |

$$max\{x_1, x_2, \ldots, x_k\} = \begin{cases} 2 & \text{when some } x_i \text{ is 2;} \\ 1 & \text{when no } x_i \text{ is 2 and some } x_j \text{ is 1;} \\ 0 & \text{when no } x_i \text{ is 2 and no } x_j \text{ is 1.} \end{cases}$$

*Example 4.1.1* The results of the following operations

$max\{1, 1 \times 1\}$,
$max\{1, 1, 1 \times 1\}$,
$max\{1, 1\}$

are all 1.

## (2) Solution Number Expression

During the mode analysis process, we will obtain expressions which overestimate the numbers of each solution in the solution table. A *solution number expression* (or simply *expression*) is defined as follows:

(a) The solution number constants 0, 1 and 2 are factors.
(b) $\#(A\nu)$ is a factor when $A\nu$ is a mode-abstracted atom. (Intuitively, $\#(A\nu)$ is used to show that each atom in some mode-abstracted atom has at most $\#(A\nu)$ solutions in $A\nu$. It often takes much space to write $\#(A\nu)$ for actual $A\nu$ so that we use $\gamma$ (possibly with subscript) for its abbreviation.)
(c) $0 \times f$, $1 \times f$, $2 \times f$ and $\gamma_i \times f$ are factors when $f$ is a factor.
(d) A factor is a simple expression.
(e) $f + e$ is a simple expression when $f$ is a factor and $e$ is a simple expression.
(f) $max\{e_1, e_2, \ldots, e_k\}$ is an expression when $e_1, e_2, \ldots, e_k$ are simple expressions ($k \geq 1$).

*Example 4.1.2* Let $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ be abbreviations for

$\#(\text{isort(L,M)} < L, M \Leftarrow \underline{ground} >)$,
$\#(\text{insertp(X,N,M)} < X, \overline{N} \Leftarrow \underline{ground} >)$,
$\#(\text{X} \leq \text{Y} < X, Y \Leftarrow \underline{ground} >)$,
$\#(\text{X} > \text{Y} < X, Y \Leftarrow \underline{ground} >)$,

respectively. Then

$max\{1, \gamma_1 \times \gamma_2\}$,
$max\{1, \gamma_3, \gamma_4 \times \gamma_2\}$,
$max\{1, \gamma_3\}$,
$max\{1, \gamma_4\}$

are solution number expressions.

Hereafter, the following simplification of solution number expressions are implicitly assumed.

$$\begin{array}{ll} 0 + e \rightarrow e, & e + 0 \rightarrow e, \\ 2 + e \rightarrow 2, & e + 2 \rightarrow 2, \\ 0 \times e \rightarrow 0, & e \times 0 \rightarrow 0, \\ 1 \times e \rightarrow e, & e \times 1 \rightarrow e. \end{array}$$

## (3) Overestimating Solution Numbers from OR Goals

There are two kinds of OR-branching in the search tree of OLDT resolution for mode analysis.

(i) Clauses of the program is applied to a solution node.
(ii) Solutions of the solution table is applied to a lookup node.

How can we overestimate the solution number of the goal at OR-branching from those of the OR-branches.

### (i) Overestimating Solution Numbers from OR Goals Using Programs

For simplicity, we assume that the following *consistency matrix C* is known for each mode-abstracted atom.

$$C_{ij} = \begin{cases} 1 & \text{there might exist a different solution of a common goal using clause } i \text{ and clause } j; \\ 0 & \text{there is no different solution of a common goal using clause } i \text{ and clause } j; \end{cases}$$

When the predicate of a mode-abstracted atom is defined using $k$ clauses, its consistency matrix $C$ is a $k \times k$ symmetric matrix with its all diagonal elements 1.

*Example 4.1.3* Let *insertp* be a predicate for inserting an element *properly* defined by

insertp(X,[ ],[X]).
insertp(X,[Y|N],[X,Y|N]) :- X≤Y.
insertp(X,[Y|N],[Y|M]) :- Y<X, insertp(X,N,M).

Then the consistency matrix of $insertp(X, N, M) < X, N \Leftarrow \underline{ground} >$ is

$$\begin{pmatrix} 1,0,0 \\ 0,1,0 \\ 0,0,1 \end{pmatrix},$$

because the second arguments of the heads of the 1st clause is not unifiable with those of the 2nd and 3rd clauses, and $X \leq Y$ and $X > Y$ are inconsistent when $X$ and $Y$ are ground. Now, let *insertr* be a predicate for inserting an element *randomly* defined by

insertr(X,N,[X|N]).
insertr(X,[Y|N],[Y|M]) :- insertr(X,N,M).

Then, the consistency matrix of mode-abstracted atom $insertr(X, N, M) < X, N \Leftarrow \underline{ground} >$ is

$$\begin{pmatrix} 1,1 \\ 1,1 \end{pmatrix}.$$

How to obtain consistency matrices is discussed in Debray and Warren [3].

Now, suppose that the solution numbers of a mode-abstracted negative clause "$A_1$, $A_2$, ..., $A_n$" $\mu$ are overestimated by solution number expressions $e_1, e_2, \ldots, e_k$ when 1st, 2nd, ..., $k$-th clauses are used first respectively. Then, if $C = (C_{ij})$ is the consistency matrix of $A_1\mu$, the solution number of the negative clause is at most

$$\begin{aligned} max\{&C_{11} \times e_1 + C_{12} \times e_2 + \cdots + C_{1k} \times e_k, \\ &C_{21} \times e_1 + C_{22} \times e_2 + \cdots + C_{2k} \times e_k, \\ &\vdots \\ &C_{k1} \times e_1 + C_{k2} \times e_2 + \cdots + C_{kk} \times e_k\}. \end{aligned} \qquad (*)$$

*Example 4.1.4* Suppose that the solution number of any atom in $insertp(X, N, M) < X, N \Leftarrow \underline{ground} >$ is at most 1 when 1st, 2nd and 3rd clauses are used first respectively. Then the solution number of the atom is at most

$max\{1 \times 1 + 0 \times 1 + 0 \times 1, 0 \times 1 + 1 \times 1 + 0 \times 1, 0 \times 1 + 0 \times 1 + 1 \times 1\} = 1.$

Now suppose that the solution number of any atom in $insertr(X, N, M) < X, N \Leftarrow \underline{ground} >$ is at most 1 when 1st and 2nd clauses are used first respectively. Then the solution number of the atom is at most

$max\{1 \times 1 + 1 \times 1, 1 \times 1 + 1 \times 1\} = 2.$

### (ii) Overestimating Solution Numbers from OR Goals Using Solution Tables

Suppose that the solution numbers of a mode-abstracted negative clause "$A_1, A_2, \ldots,$ $A_n$" $\mu$ are overestimated by solution number expressions $e_1, e_2, \ldots, e_k$ when $k$ solutions in $A_1\mu$'s solution list are used first respectively. Then, because we have no information about whether these solutions in the solution table overlap or not, we can only overestimate the solution number of the original negtive clause by

$$e_1 + e_2 + \cdots + e_k. \qquad (**)$$

*Example 4.1.5* Suppose that a goal $multiply(X, Y, W) < X \Leftarrow \underline{ground} >$ has two solutions using the solutions in the solution list below first.

multiply(X,Y,Z)$< X \Leftarrow \underline{ground} >$ : $[multiply(X, Y, Z) < X, Z \Leftarrow \underline{ground} >,$
$multiply(X, Y, Z) < X, Y, Z \Leftarrow \underline{ground} >]$

and the numbers of these solutions are overestimated by

$\gamma_1 \times \gamma_3$,

$\gamma_2 \times \gamma_4$.

Then the number of the solutions of the original goal is overestimated by

$\gamma_1 \times \gamma_3 + \gamma_2 \times \gamma_4$.

### (4) Overestimating Solution Numbers from AND Goals

Suppose that a mode-abstracted negative clause "$A_1, A_2, \ldots, A_n$" $\mu_1$ has a refutation in which the $i$-th mode-abstracted atom is of the form $A_i\mu_i$ when it is at the leftmost, and the solution numbers of the mode-abstracted atoms $A_1\mu_1$, $A_2\mu_2$, $\ldots$, $A_n\mu_n$ are overestimated by $e_1, e_2, \ldots, e_n$. Then the solution number of the original negative clause is overestimated by their product

$$e_1 \times e_2 \times \cdots \times e_n . \qquad (***)$$

*Example 4.1.6* Consider an AND goal in

"$isort(L,N), insertp(X,N,M)$" $<L \Leftarrow \underline{ground}>$.

Suppose that the number of $isort(L,N)$ $<L \Leftarrow \underline{ground}>$'s solutions returning an atom in $isort(L,N)$ $<L, N \Leftarrow \underline{ground}>$ is overestimated by

$\gamma_1$,

and the number of $insertp(X,N,M)$ $<X, N \Leftarrow \underline{ground}>$'s solution returning an atom in $insertp(X,N,M)$ $<X,N,M \Leftarrow \underline{ground}>$ is overestimated by

$\gamma_2$.

Then the number of $isort(L,N), insertp(X,N,M)$ $<L \Leftarrow \underline{ground}>$'s solutions returning an atom in $isort(L,N), insertp(X,N,M)$ $<L,N,X,M \Leftarrow \underline{ground}>$ is overestimated by their product

$\gamma_1 \times \gamma_2$.

## (5) Join of Solution Number Expressions

Recall the mode analysis process in Section 3.3. In Figure 3.3.3 of Example 3.3.1, we have one solution of the top-level goal

$isort(L,M)$ $<L,M \Leftarrow \underline{ground}>$

in the solution table, but the solution has been obtained through 4 different paths in the search tree. In general, one solution in a solution table is obtained through many different paths in a search tree. (In Figure 3.3.4 at step 10, the solution is obtained through 6 paths.) In order to overestimate the number of the solution, we need to take all paths into consideration. Though we can obtain the overestimation of the solution number according to the rules $(*)$, $(**)$ and $(***)$ after having constructed up all the search tree, it is more efficient if we can obtain the overestimation incrementally as the mode analysis proceeds. How should we update the new overestimation when a new additional overestimation of the solution number is obtained?

*Example 4.1.7* Consider the four refutations in Figure 3.3.3 which appeared during the mode analysis of $isort(L_0, M_0)$ $<L_0 \Leftarrow \underline{ground}>$ in Example 3.3.1.

Though the solution number expression overestimating the number of the solution $isort(L,M)$ $<L,M \Leftarrow \underline{ground}>$ is obtained from 6 refutations in Figure 3.3.4 at step 10, which give the solution, by following the rule of $(*)$, $(**)$ and $(***)$, we can just obtain the partial information of the overestimation at step 6

$max\{1+?, 0+?\}$,

$max\{? + 0, ? + \gamma_1 \times \gamma_2\}$,

$max\{? + 0, ? + \gamma_1 \times \gamma_2\}$,

$max\{? + 0, ? + \gamma_1 \times \gamma_2\}$,

at step 2, 4, 5 and 6, where $\gamma_1$ and $\gamma_2$ are abbreviations for

$\#$(isort(L,M) $<L,M \Leftarrow ground>$),

$\#$(insertp(X,N,M) $<X,N,M \Leftarrow \underline{ground}>$),

and the "?" parts should be decided from other refutations. As far as the process proceeds until step 6, we can just summarize the partial information obtained so far to $max\{1, \gamma_1 \times \gamma_2\}$.

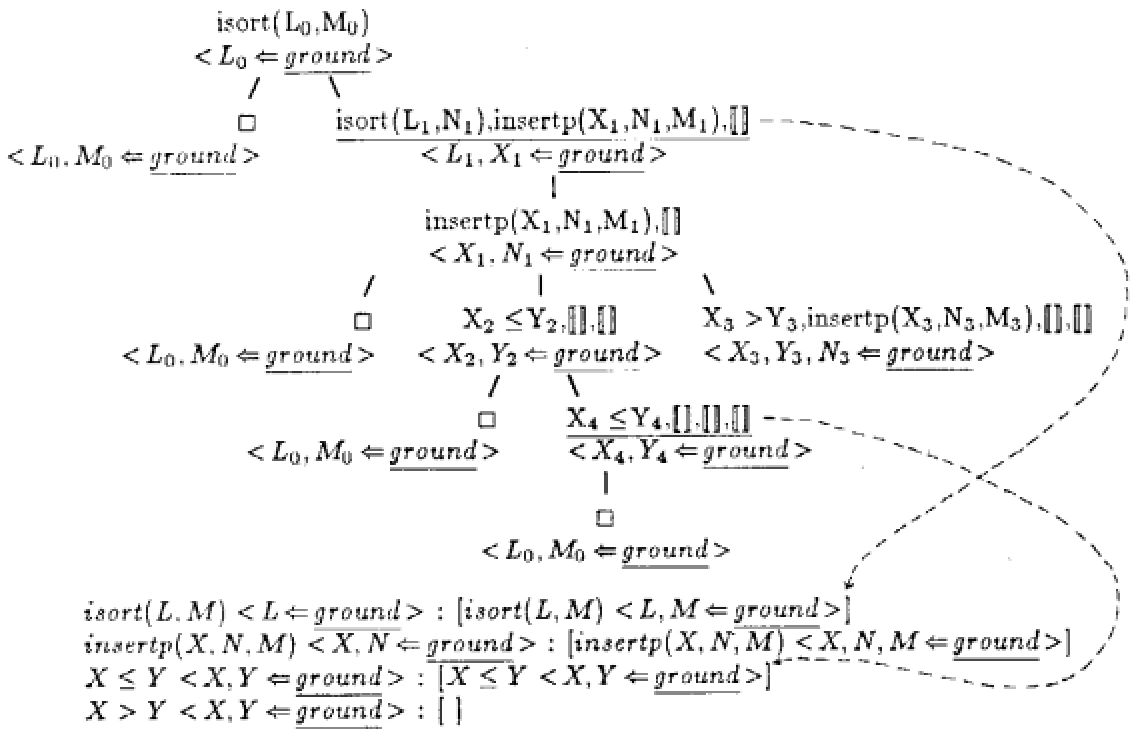

isort(L$_0$,M$_0$)
$< L_0 \Leftarrow \underline{ground} >$
$\Box$
$< L_0, M_0 \Leftarrow \underline{ground} >$
isort(L$_1$,N$_1$),insertp(X$_1$,N$_1$,M$_1$),[]
$< L_1, X_1 \Leftarrow \underline{ground} >$
insertp(X$_1$,N$_1$,M$_1$),[]
$< X_1, N_1 \Leftarrow \underline{ground} >$
$\Box$
$< L_0, M_0 \Leftarrow \underline{ground} >$
X$_2$ ≤Y$_2$,[],[]
$< X_2, Y_2 \Leftarrow \underline{ground} >$
X$_3$ >Y$_3$,insertp(X$_3$,N$_3$,M$_3$),[],[]
$< X_3, Y_3, N_3 \Leftarrow \underline{ground} >$
$\Box$
$< L_0, M_0 \Leftarrow \underline{ground} >$
X$_4$ ≤Y$_4$,[],[],[]
$< X_4, Y_4 \Leftarrow \underline{ground} >$
$\Box$
$< L_0, M_0 \Leftarrow \underline{ground} >$
$isort(L, M) < L \Leftarrow \underline{ground} >$ : $[isort(L, M) < L, M \Leftarrow \underline{ground} >]$
$insertp(X, N, M) < X, N \Leftarrow \underline{ground} >$ : $[insertp(X, N, M) < X, N, M \Leftarrow \underline{ground} >]$
$X \leq Y < X, Y \Leftarrow \underline{ground} >$ : $[X \leq Y < X, Y \Leftarrow \underline{ground} >]$
$X > Y < X, Y \Leftarrow \underline{ground} >$ : [ ]

**Figure 4.1.1 Four Refutations Obtained at Step 6**

The *join of solution number expressions* $e \vee f$ is devised for summarizing the partial overestimation from each overestimation so far obtained as follows:

$$max\{e_1, e_2, \ldots, e_k\} \vee max\{f_1, f_2, \ldots, f_k\} = max\{e_1 \vee f_1, e_2 \vee f_2, \ldots, e_k \vee f_k\},$$

$$e \vee f = \begin{cases} e & \text{the simple expression } e \text{ contains the factor } f \text{ ;} \\ e + f & \text{the simple expression } e \text{ does not contain the factor } f \text{ ;} \end{cases}$$

Note that, the first rule corresponds to the following column by column decomposition.

$$\begin{aligned} & max\{C_{11} \times e_1 + C_{12} \times e_2 + \cdots + C_{1k} \times e_k, \\ & \qquad C_{21} \times e_1 + C_{22} \times e_2 + \cdots + C_{2k} \times e_k, \\ & \qquad \ldots \\ & \qquad C_{k1} \times e_1 + C_{k2} \times e_2 + \cdots + C_{kk} \times e_k\}. \\ = \; & max\{C_{11} \times e_1, C_{21} \times e_1, \ldots, C_{k1} \times e_1\} \vee \\ & max\{C_{12} \times e_2, C_{22} \times e_2, \ldots, C_{k2} \times e_2\} \vee \\ & \qquad \ldots \\ & max\{C_{1k} \times e_k, C_{2k} \times e_k, \ldots, C_{kk} \times e_k\}. \end{aligned}$$

*Example 4.1.8* Suppose that the number of solutions obtained through the four paths in Figure 3.3.3 are overestimated by

$max\{1, 0\}$ at step 2,

$max\{0, \gamma_1 \times \gamma_2\}$ at step 4, 5 and 6,

respectively. Then the overestimation of the number of the solution is obtained incrementally as the mode analysis proceeds as follows:

$max\{1,0\}$ at step 2,
$max\{1,0\} \vee max\{0, \gamma_1 \times \gamma_2\} = max\{1, \gamma_1 \times \gamma_2\}$ at step 4,
$max\{1, \gamma_1 \times \gamma_2\} \vee max\{0, \gamma_1 \times \gamma_2\} = max\{1, \gamma_1 \times \gamma_2\}$ at step 5 and 6.

## (6) Calculating Solution Numbers from Solution Number Expressions

Let $\gamma_1, \gamma_2, \ldots, \gamma_k$ be abbreviations for $\#(A_1\mu_1), \#(A_2\mu_2), \ldots, \#(A_k\mu_k)$, and $e_1, e_2, \ldots, e_k$ be the solution number expressions (possibly containing $\gamma_1, \gamma_2, \ldots, \gamma_k$) overestimating $\gamma_1, \gamma_2, \ldots, \gamma_k$ respectively. We denote the $k$-vector $(\gamma_1, \gamma_2, \ldots, \gamma_k)$ by $\gamma$, the $k$-vector of solution number expressions $(e_1, e_2, \ldots, e_k)$ by $\mathbf{e}(\gamma)$, the $k$-vector $(1, 1, \ldots, 1)$ by $\mathbf{1}$ and a $k$-vector $(n_1, n_2, \ldots, n_k)$ by $\mathbf{n}$. Then the overestimation of these solution numbers are obtained as follows:

```
$\mathbf{n}_0 := \mathbf{1}$ ;
repeat $i := i + 1$; $\mathbf{n}_{i+1} := \mathbf{e}(\mathbf{n}_i)$ until $\mathbf{n}_{i+1} \leq \mathbf{n}_i$
return $\mathbf{n}_i$
```

**Figure 4.1.2 Bottom-up Approximation of Solution Numbers**

Because each solution $A\mu_i$ in the solution table is obtained in the mode analysis process, some atom in the key mode-abstracted atom has at least one solution in $A\mu_i$. Hence, the number of solutions in $A\mu_i$ is estimated by 1 first. Then, using the relation represented by each $e_i$, the estimation is repeatedly updated until there occurs no change.

We will show how the previous two examples continue.

*Example 4.1.9* Suppose that the numbers of solutions represented by $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ are overestimated as follows:

$\gamma_1 : max\{1, \gamma_1 \times \gamma_2\}$,
$\gamma_2 : max\{1, \gamma_3, \gamma_4 \times \gamma_2\}$
$\gamma_3 : max\{1, \gamma_3\}$
$\gamma_4 : max\{1, \gamma_4\}$

At the beginning of the calculation, $n_1$, $n_2$, $n_3$ and $n_4$ are initialized to 1. After the first repetition of the computation of the right-hand sides, they are updated to $1, 1, 1, 1$, which do not change henceafter. After all, we have detected that these solution numbers are at most 1.

*Example 4.1.10* Suppose that the numbers of solutions represented by $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6$ are overestimated as follows:

$\gamma_1 : \gamma_2 \times \gamma_3$
$\gamma_2 : max\{1, \gamma_2 \times \gamma_3\}$
$\gamma_3 : max\{1 + \gamma_3, 1 + \gamma_3\}$
$\gamma_4 : max\{1, 1, \gamma_5 \times \gamma_6\}$
$\gamma_5 : max\{1, \gamma_5\}$
$\gamma_6 : max\{1, 1, \gamma_5 \times \gamma_6\}$

At the beginning of the calculation, $n_1$, $n_2$, $n_3$, $n_4$, $n_5$ and $n_6$ are initialized to 1. After the first repetition of the computation of the right-hand sides, they are updated to $2, 2, 2, 1, 1, 1$, which do not change henceafter. After all, we could not detect that these solution numbers are at most 1.

In the previous examples, we had to perform the bottom-up calculation after the solution number expressions overestimating solution numbers were known. Is it possible to know whether $\gamma_i$ is overestimated by 1 or 2 from the solution number expressions directly? Careful examination shows that $\gamma_i$ is overestimated by 2 if and only if the corresponding expression $e_i$ contains either the plus symbol "+" or $\gamma_j$ overestimated by 2. For example, $e_3$ contains "+", and $e_1$ and $e_2$ contains $\gamma_3$ in Example 4.1.10. Hence, we define $\bar{e}$ as follows:

$$\bar{e} = \begin{cases} 2 & \text{when } e \text{ contains either the plus symbol "+"} \\ & \text{or } \gamma_j \text{ overestimated by 2;} \\ e & \text{otherwise.} \end{cases}$$

## 4.2 Abstract Hybrid Interpretation for Functionality Detection

A *search tree for functionality detection* is a tree with its node labeled with a triple of a (generalized) negative clause, a mode substitution and a factor. (For brevity, we will sometimes omit the term "for functionality detection" hereafter in Section 4.) A *search tree of* $(G, \mu, 1)$ is a search tree whose root node is labeled with $(G, \mu, 1)$. The clause part of each triple is a sequence $\alpha_1, \alpha_2, \ldots, \alpha_n$ consisting of either atoms in the body of some clause in $P \cup \{G\}$ or call-exit markers of the form $[A, \mu, e, \eta, h\text{-th}]$. A *refutation* of $(G, \mu, 1)$ is a path in a search tree of $(G, \mu, e)$ from the root to a node labelled with $(\Box, \nu, f)$. The *answer substitution of the refutation* is the mode substitution $\nu$, the *answer factor* is the factor $f$, and the *solution of the refutation* is $G\nu f$.

A *solution table for functionality detection* is a set of entries. Each entry consists of the *key* and the *solution list*. The key is a mode-abstracted atom $A\mu$. The solution list is a list of triples of the form $A\nu e$, called *solutions*, where $e$ is a solution number expression.

$\#(A\nu)$ is said to be *overestimated by 2* in a solution table $Tb$, when there is a solution of the form $A\nu 2$ in $Tb$. $\#(A\mu)$ is said to *immediately depend on* $\#(B\nu)$ in a solution table $Tb$, when there is a solution $A\mu e$ in $Tb$ such that $\#(B\nu)$ appears in $e$. $\#(A\mu)$ is said to *depend on* $\#(B\nu)$ when $\#(A\mu)$ is connected to $\#(B\nu)$ through successive "immediate depend" relation, i.e., they are in the transitive closure of the "immediate depend" relation.

Let $Tr$ be a search tree whose nodes labeled with non-null clauses are classified into either *solution nodes* or *lookup nodes*, and let $Tb$ be a solution table. An *association for functionality detection* of $Tr$ and $Tb$ is a set of pointers pointing from each lookup node in $Tr$ into some solution list in $Tb$.

An *OLDT structure for functionality detection* is a triple of search tree, solution table and association. The relation between a node and its child nodes in a search tree is specified by *OLDT resolution for functionality detection* as follows:

A node of OLDT structure $(Tr, Tb, As)$ labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\mu$, $e$) is said to be *OLDT resolvable* $(n \geq 1)$ when it satisfies either of the following conditions:

(a) The node is a terminal solution node of $Tr$ and there is some definite clause "$B_0$ :- $B_1, B_2, \ldots, B_m$" $(m \geq 0)$ in program $P$ such that $\alpha_1$ and $B_0$ is unifiable, say by an m.g.u. $\eta$.

(b) The node is a lookup node of $Tr$ and there is some solution $B\nu f$" in the associated solution list of the lookup node such that $\alpha_1$ and $B$ are variants of each other. Let $\eta$ be the renaming of $B$ to $\alpha_1$.

The precise algorithm of OLDT resolution for functionality detection is shown in Figure 4.2. Note that the operations at steps (A), (B) and (C) are modified.

```
OLDT-resolve(("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\mu$, $e$) : label) : label ;
    $i := 0$;
    case
      when a solution node is OLDT resolved with the $h$-th clause "$B_0$ :- $B_1, B_2, \ldots, B_m$"
            in $P$ defining the predicate of $\alpha_1$
        let $\eta$ be the m.g.u. of $\alpha_1$ and $B_0$ ;
        let $G_0$ be a negative clause "$B_1, B_2, \ldots, B_m$, $[\![\alpha_1, \mu, e, \eta, h\text{-th}]\!]$, $\alpha_2, \ldots, \alpha_n$" ;
        let $\nu_0$ be $propagate(\mu, \eta)$ and $f_0$ be 1 ;                                   — (A)
      when a lookup node is OLDT resolved with "$B\nu f$" in $Tb$
        let $\eta$ be renaming of $B$ to $\alpha_1$ ;
        let $G_0$ be a negative caluse "$\alpha_2, \ldots, \alpha_n$" ;
        let $\nu_0$ be $\mu \vee propagate(\nu, \eta)$ and $f_0$ be $e \times \#(B\nu)$ ;           — (B)
    endcase
    while the leftmost of $G_i$ is a call-exit marker $[\![A_{i+1}, \mu_{i+1}, e_{i+1}, \eta_{i+1}, h\text{-th}]\!]$ do
        let $G_{i+1}$ be $G_i$ other than the leftmost call-exit marker ;
        let $\nu_{i+1}$ be $\mu_{i+1} \vee propagate(\nu_i, \eta_{i+1})$ ;
        let $f_{i+1}$ be $e_{i+1} \times \#(A_{i+1}\nu_{i+1})$ ;                                   — (C)
        add-solution($A_{i+1}\nu_{i+1} max\{C_{1h} \times f_i, C_{2h} \times f_i, \ldots, C_{kh} \times f_i\}$, $A_{i+1}\mu_{i+1}$)
            where $C$ is the consistency matrix of $A_{i+1}\mu_{i+1}$ ;
        $i := i + 1$;
    endwhile
    $(G_{new}, \mu_{new}, e_{new}) := (G_i, \nu_i, f_i)$;
    return $(G_{new}, \mu_{new}, e_{new})$.

add-solution($A\nu e$, $A\mu$) ;
    case
      when there exists a solution $A\nu e'$ in $A\mu$'s solution list :
          replace the solution $A\nu e'$ with $A\nu \overline{e' \vee e}$ ;
          if $e' \vee e = 2$
          then for all $B\lambda f$ in the solution table such that $\#(B\lambda)$ depends on $\#(A\nu)$,
                  replace the expression part $f$ with 2 ;
      when there exists no solution of the form $A\nu e'$ in $A\mu$'s solution list :
          add $A\nu e$ to the last of the solution list ;
    endcase
```

**Figure 4.2 OLDT Resolution for Functionality Detection**

A node labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\mu$, $e$) is a lookup node when the mode-abstracted atom $\alpha_1\mu$ is a key in the solution table, and is a solution node otherwise ($n \geq 1$).

The *initial OLDT structure*, *immediate extension of OLDT structure*, *extension of OLDT structure*, *answer substitution of OLDT refutation* and *solution of OLDT refutation* are defined in the same way as in Section 2.3.

## 4.3 Examples of Functionality Detection

We will show two examples of how functionality detection proceeds.

*Example 4.3.1* Let *isort*, *insertp*, $\leq$ and $<$ be predicates defined as before in Example 3.3.1. Then, extension of OLDT structure proceeds similarly to the mode analysis.

First, the initial OLDT structure below is generated. The expression part of the root node is initialized to 1.

$$\begin{array}{c} \mathrm{isort}(L_0,M_0) \\ < L_0 \Leftarrow \underline{ground} > \\ \hline 1 \end{array}$$

$isort(L, M) < L \Leftarrow \underline{ground} > : [\ ]$

**Figure 4.3.1 Functionality Detection at Step 1**

Secondly, the root node is OLDT resolved using the program to generate two child nodes. Since the clause part of the intermediate label of the left child node is

"$[\![isort(L_0, M_0), < L_0 \Leftarrow \underline{ground} >, 1, < L_0, M_0 \Leftarrow [\ ] >, 1st]\!]$",

the node is the end of a unit subrefutation and its solution $isort(L_0, M_0) < L_0, M_0 \Leftarrow \underline{ground} > max\{1,0\}$ is added to the solution list, because the expression part of the intermediate label is 1 and the consistency matrix of $isort(L_0, M_0) < L_0 \Leftarrow \underline{ground} >$ is

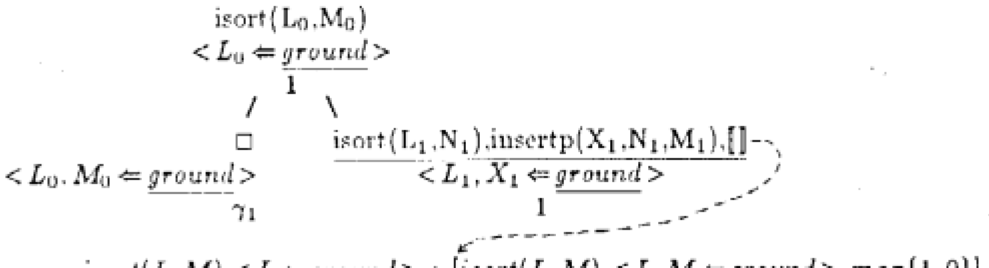
$$\begin{pmatrix} 1,0 \\ 0,1 \end{pmatrix}.$$

The expression part of the label is set to $\gamma_1$, where $\gamma_1$ is an abbreviation for $isort(L, M) < L, M \Leftarrow \underline{ground} >$. Since the clause part of the intermediate label of the right child node is

"$isort(L_1, N_1), insertp(X_1, N_1, M_1)$,

$[\![isort(L_0, M_0), < L_0 \Leftarrow \underline{ground} >, 1, < L_0 \Leftarrow [X_1|L_1], M_0 \Leftarrow M_1 >, 2nd]\!]$",

it is immediately the clause part of the generated node. (Again, the quantities inside call-exit markers are omitted due to space limit so that they are depicted simply by $[\![\,]\!]$.)



isort(L0,M0)
< L0 ⇐ ground >
1
□
< L0, M0 ⇐ ground >
γ1
isort(L1,N1),insertp(X1,N1,M1),[[]]
< L1, X1 ⇐ ground >
1
isort(L, M) < L ⇐ ground > : [isort(L, M) < L, M ⇐ ground > max{1, 0}]

**Figure 4.3.2 Functionality Detection at Step 2**

Thirdly, the lookup node is OLDT resolved using the solution table to generate one child solution node. Since the number of the solution used at the OLDT resolution is denoted by $\gamma_1$, the expression part of the new label is $1 \times \gamma_1 = \gamma_1$.

Fourthly, the new solution node is OLDT resolved using the program to generate three child nodes. The generated left child node adds two solutions to the solution table while two call-exit markers

$[\![insertp(X_1, N_1, M_1), < X_1, N_1 \Leftarrow \underline{ground} >, \gamma_1, < X_1 \Leftarrow X, N_1 \Leftarrow [\ ], M_1 \Leftarrow [X] >, 1st]\!]$,

$[\![isort(L_0, M_0), < L_0 \Leftarrow \underline{ground} >, 1, < L_0 \Leftarrow [X_1|L_1], M_0 \Leftarrow M_1 >, 2nd]\!]$

are eliminated during the exiting phase. The first one gives a solution $insertp(X, N, M) < X, N, M \Leftarrow \underline{ground} >$ with its expression $max\{1, 0, 0\}$, since the consistency matrix of $insertp(X, N, M) < X, N \Leftarrow \underline{ground}>$ is

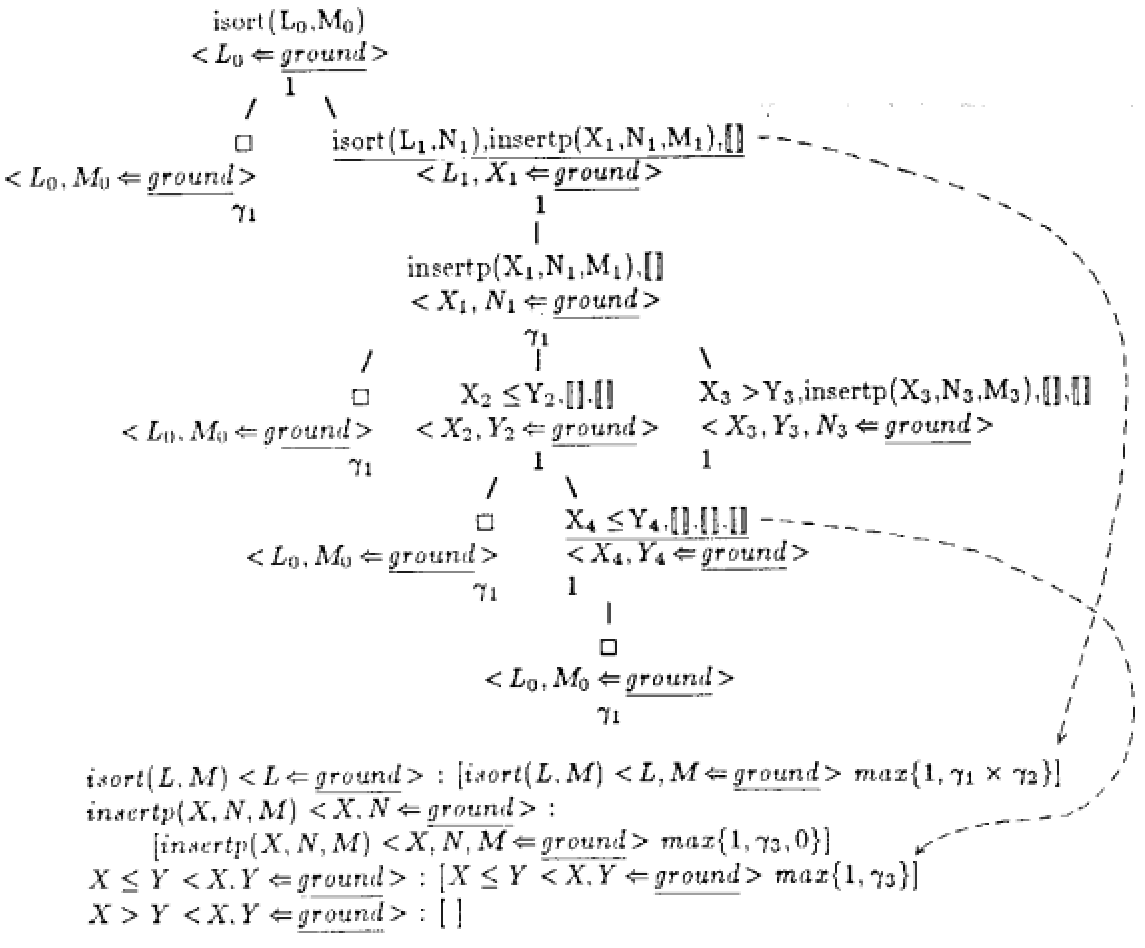
$$\begin{pmatrix} 1,0,0 \\ 0,1,0 \\ 0,0,1 \end{pmatrix}.$$

The second one gives a solution $isort(L, M) < L, M \Leftarrow \underline{ground} >$ with its expression $max\{0, \gamma_1 \times \gamma_2\}$, where $\gamma_2$ is an abbreviation for $\#(insertp(X, N, M) < X, N, M \Leftarrow \underline{ground}>)$. The expression part of the solution in the solution table is updated to the join of the new expression and the previous one

$max\{1, 0\} \vee max\{0, \gamma_1 \times \gamma_2\} = max\{1, \gamma_1 \times \gamma_2\}$.

The generated center child node and right child node are solution nodes. Their expression part are both 1.

Fifthly, the center node is OLDT resolved using the program to generate two child nodes. The generated left child node labeled with the null clause □ gives three solutions, which are added to the solution table as before. The generated right child node is a lookup node.



isort(L₀,M₀) <L₀ ⇐ ground> 1
□ <L₀, M₀ ⇐ ground> γ₁
isort(L₁,N₁),insertp(X₁,N₁,M₁),[] <L₁, X₁ ⇐ ground> 1
insertp(X₁,N₁,M₁),[] <X₁, N₁ ⇐ ground> γ₁
□ <L₀, M₀ ⇐ ground> γ₁
X₂ ≤Y₂,[],[] <X₂, Y₂ ⇐ ground> 1
X₃ >Y₃,insertp(X₃,N₃,M₃),[],[] <X₃, Y₃, N₃ ⇐ ground> 1
□ <L₀, M₀ ⇐ ground> γ₁
X₄ ≤Y₄,[],[],[] <X₄, Y₄ ⇐ ground> 1
□ <L₀, M₀ ⇐ ground> γ₁

$isort(L, M) < L \Leftarrow \underline{ground}>$ : $[isort(L, M) < L, M \Leftarrow \underline{ground}>\ max\{1, \gamma_1 \times \gamma_2\}]$

$insertp(X, N, M) < X, N \Leftarrow \underline{ground}>$ :

$[insertp(X, N, M) < X, N, M \Leftarrow \underline{ground}>\ max\{1, \gamma_3, 0\}]$

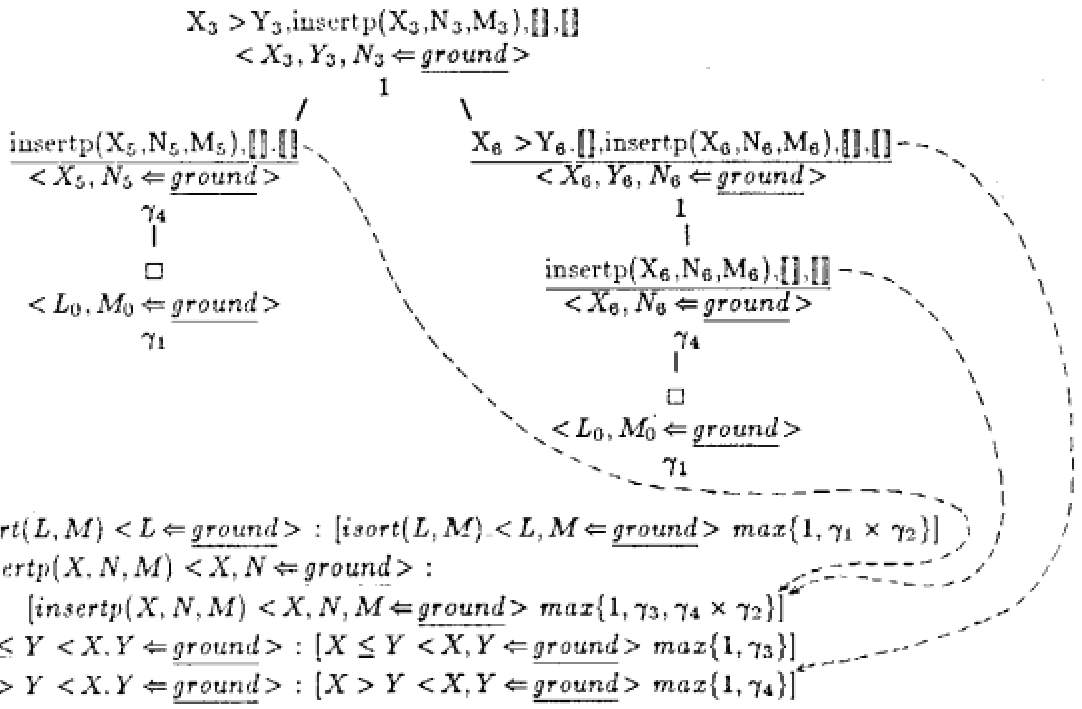
$X \leq Y < X, Y \Leftarrow \underline{ground}>$ : $[X \leq Y < X, Y \Leftarrow \underline{ground}>\ max\{1, \gamma_3\}]$

$X > Y < X, Y \Leftarrow \underline{ground}>$ : $[\ ]$

**Figure 4.3.3 Functionality Detection at Step 6**

Sixthly, the lookup node is OLDT resolved using the solution table to generate one child node. The generated child node labeled with the null clause □ gives three solutions, which are added to the solution table as before.

The process proceeds in the same way. Lastly at step 10, all nodes are OLDT resolved up. The search tree below the rightmost solution node in Figure 4.3.3 and the final solution table are as follows:


$X_3 > Y_3$,insertp($X_3$,$N_3$,$M_3$),[],[]
$<X_3, Y_3, N_3 \Leftarrow \underline{ground}>$
1
insertp($X_5$,$N_5$,$M_5$),[],[]
$<X_5, N_5 \Leftarrow \underline{ground}>$
$\gamma_4$
□
$<L_0, M_0 \Leftarrow \underline{ground}>$
$\gamma_1$
$X_6 > Y_6$,[],insertp($X_6$,$N_6$,$M_6$),[],[]
$<X_6, Y_6, N_6 \Leftarrow \underline{ground}>$
1
insertp($X_6$,$N_6$,$M_6$),[],[]
$<X_6, N_6 \Leftarrow \underline{ground}>$
$\gamma_4$
□
$<L_0, M_0 \Leftarrow \underline{ground}>$
$\gamma_1$

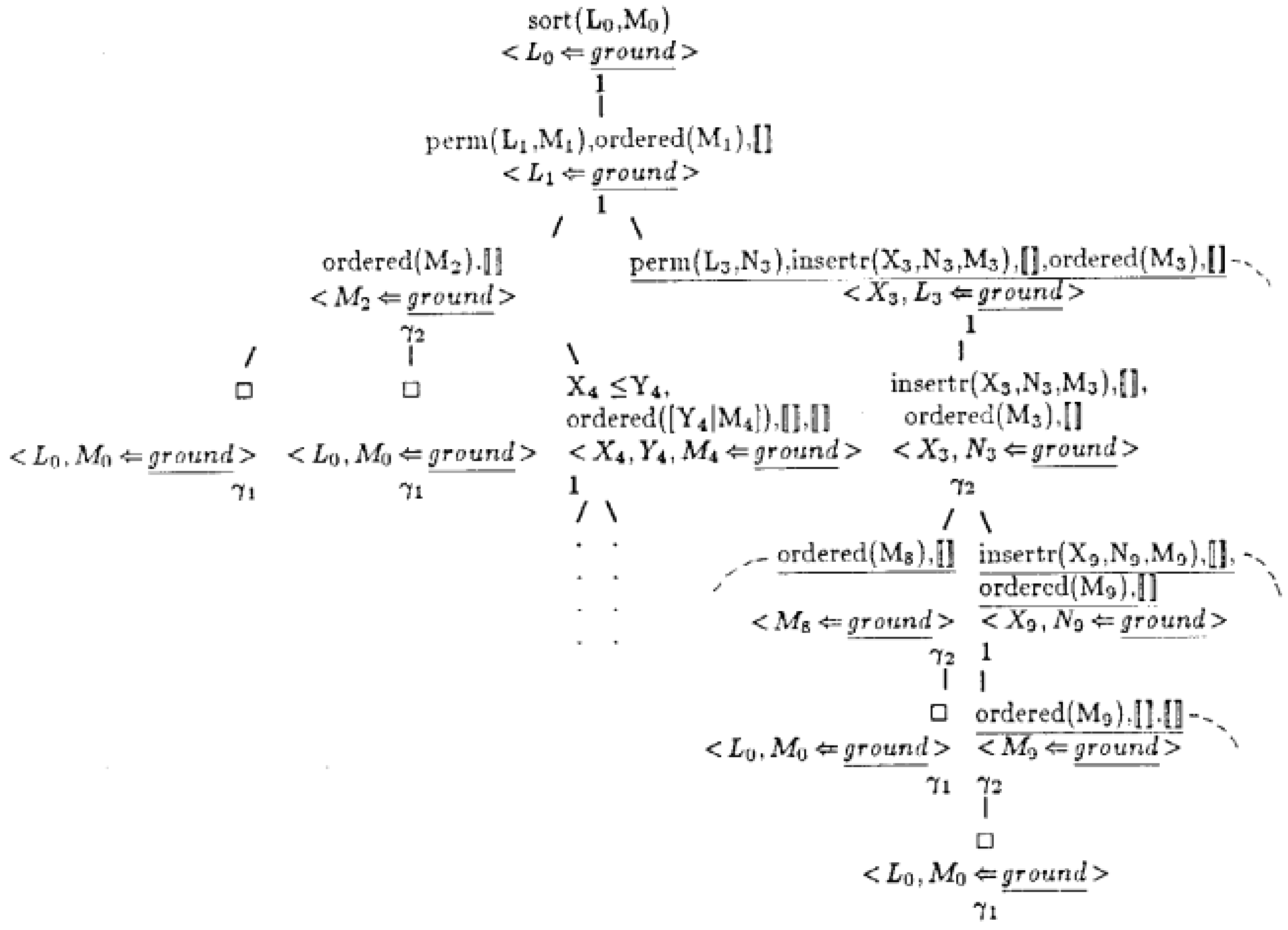
$isort(L,M) <L \Leftarrow \underline{ground}>$ : $[isort(L,M) <L,M \Leftarrow \underline{ground}>\ max\{1, \gamma_1 \times \gamma_2\}]$
$insertp(X,N,M) <X,N \Leftarrow \underline{ground}>$ :
    $[insertp(X,N,M) <X,N,M \Leftarrow \underline{ground}>\ max\{1, \gamma_3, \gamma_4 \times \gamma_2\}]$
$X \leq Y <X,Y \Leftarrow \underline{ground}>$ : $[X \leq Y <X,Y \Leftarrow \underline{ground}>\ max\{1, \gamma_3\}]$
$X > Y <X,Y \Leftarrow \underline{ground}>$ : $[X > Y <X,Y \Leftarrow \underline{ground}>\ max\{1, \gamma_4\}]$

**Figure 4.3.4 Functionality Detection at Step 10**

The final solution table says that, when an atom *isort* is called with its first argument instantiated to a ground term, the number of its solution is at most 1, because there is only one solution in the corresponding solution list. Similarly, solution numbers of atoms in other entries are overestimated by (the sums of) the solution numbers in the corresponding solution lists. After all, we have detected the functionality of *isort*.

*Example 4.3.2* Let *sort*, *perm*, *insertr* and *ordered* be predicates defined as before in Example 3.3.2. Then the OLDT resolution for functionality detection generates the following OLDT structures for functionality detection, where $\gamma_1$, $\gamma_2$, $\gamma_3$, $\gamma_4$, $\gamma_5$ and $\gamma_6$ are abbreviations for

$\#$(sort(L,M) $<L,M \Leftarrow \underline{ground}>$),
$\#$(perm(L,M) $<L,M \Leftarrow \underline{ground}>$),
$\#$(insertr(X,N,M) $<X,N,M \Leftarrow \underline{ground}>$),
$\#$(ordered(M) $<M \Leftarrow \underline{ground}>$),
$\#$(X$\leq$ Y $<X,Y \Leftarrow \underline{ground}>$),
$\#$(ordered([Y|M]) $<Y,M \Leftarrow \underline{ground}>$).



$sort(L, M) < L \Leftarrow \underline{ground} >$ : $[sort(L, M) < L, M \Leftarrow \underline{ground} > 2]$
$perm(L, M) < L \Leftarrow \underline{ground} >$ : $[perm(L, M) < L, M \Leftarrow \underline{ground} > 2]$
$insertr(X, N, M) < X, N \Leftarrow \underline{ground} >$ : $[insertr(X, N, M) < X, N, M \Leftarrow \underline{ground} > 2]$
$ordered(M) < M \Leftarrow \underline{ground} >$ : $[ordered(M) < M \Leftarrow \underline{ground} > max\{1, 1, \gamma_5 \times \gamma_6\}]$
$X \le Y < X, Y \Leftarrow \underline{ground} >$ : $[X \le Y < X, Y \Leftarrow \underline{ground} > max\{1, \gamma_5\}]$
$ordered([Y|M]) < Y, M \Leftarrow \underline{ground} >$ :
$\quad [ordered([Y|M]) < Y, M \Leftarrow \underline{ground} > max\{0, 1, \gamma_5 \times \gamma_6\}]$

**Figure 4.3.5 Functionality Detection of *sort***

After all, we could not detect the functionality of *sort*. (We guess that, in general, the more easily the functionality can be detected by abstract interpretation, the more efficiently the program can perform the same task.)

*Remark.* One might wonder that the simultaneous mode analysis is redundant, since the solution number expressions overestimating solution numbers are directly corresponding to the programs, e.g., Example 4.3.1 and 4.3.2. This is not a case in general. (cf. Example 4.1.5.) Moreover, the consistency matrices are dependent on the modes.

## 5. Discussion

Functionality (or determinacy) detection of Prolog programs has been studied by several researchers extensively. A technique of detecting determinacy of Prolog programs was first proposed by Mellish [10]. Though his approach was operational (e.g., the operational characteristics of the definition of determinacy and strong dependence on the impure effect of the cut operator) and rather informal, he recently proposed a unified framework for abstract interpretation of Prolog programs (cf. [1], [2]) in order to give a theoretical foundation to his various techniques for analyzing determinacy, mode and shared structures [11]. His approach derives simultaneous recurrence equations for goals at calling time and exiting time during the top-down execution of a given top-level goal, and obtains a solution greater than the least solution of the simultaneous recurrence equations using a bottom-up approximation. Our approach is less operational and based on the standard hybrid interpretation directly, which makes the mode analysis simpler.

Debray and Warren [3] employed a less operational definition of functionality and developed a method for detecting functionality of Prolog programs. We owes to their approach both the definition of functionality and the use of consistency matrix ("mutual exclusiveness" in [3]). But their explanation of the algorithm is rather informal and implicit in several respects, though they refered to the use of mode information [4] and extension table [3] (solution table in this paper).

Sawamura and Takeshima [12] employed an operational definition of determinacy and proved its recursive unsolvability in general. Then, they clarified several solvable cases (*absolute determinacy* and *relative determinacy*) to utilize them for Prolog program optimization.

Our approach can be improved or extended in several respects. First, detection of functionality can be more efficient when functionality of some atoms is already known. If some mode-abstracted atom is known to be functional, we delete the program defining the predicates and register the mode-abstracted atoms in the solution table with its solution number expression 1. When the mode-abstracted atom appears at the leftmost, the node is always considered a lookup node.

*Example 5.1* Any atom is functional when all its arguments are instantiated to ground terms. In particular, a unary atom is functional when its argument is instantiated to a ground term, e.g., $ordered(M) < M \Leftarrow \underline{ground} >$. Functionality of some primitive predicates is obvious, e.g., $add(X, Y, Z) < X, Y \Leftarrow \underline{ground} >$. In the mode analysis process, we can delete the clauses defining *add* and modifies the initial solution table to include the entry

add(X,Y,Z)$< X, Y \Leftarrow \underline{ground} >$ : [add(X,Y,Z) $< X, Y, Z \Leftarrow \underline{ground} >$ 1]

Secondly, detection of functionality can be extended for the case when types of some arguments are given. Instead of the mode analysis, we can combine the type inference with the overestimation. (See [5], [6], [7] for details of type inference.)

*Example 5.2* Suppose we would like to know that, when a goal "$reverse(L, M)$" with its first argument $L$ instantiated to a list succeeds, the form of its second argument $M$ after the success is unique, where *reverse* and *append* are defined by

```
reverse([ ],[ ]).
reverse([X|L],M) :- reverse(L,N), append(N,[X],M).
append([ ],M,M).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Then, we only need to consider the following *type-abstracted* atom reverse(L,M)$< L \Leftarrow \underline{list} >$.

## 6. Conclusions

We have shown a framework for detecting functionality of logic programs. This method is an element of our system for analysis of Prolog programs Argus/A under development [5],[7],[8],[9].

## Acknowledgements

Our analysis system Argus/A 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 (Vice Director of ICOT), Dr. T. Yokoi (Vice Director of ICOT) and Dr. H. Ito (Chief of ICOT 3rd Laboratory) for their advice and encouragement.

## References

[1] Cousot,P.and R.Cousot, "Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", Conference Record of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, pp.238-252, 1977.

[2] Cousot,P.and R.Cousot, "Static Determination of Dynamic Properties of Recursive Procedures", in Formal Description of Programming Concepts (E.J.Neuhold Ed), pp. 237-277, North Holland, 1978.

[3] Debray,S.K.and D.S.Warren, "Detection and Optimization of Functional Computation in Prolog", Proc. of 3rd International Conference on Logic Programming, London, 1986.

[4] Debray,S.K., "Automatic Mode Inference for Prolog Programs," Proc. of 1986 Symposium on Logic Programming, Salt Lake City, 1986.

[5] Horiuchi,K.and T.Kanamori, "Polymorphic Type Inference in Prolog by Abstract Interpretation," Proc. The Logic Programming Conference '87, pp.107-116, Tokyo, 1987.

[6] Kanamori,T. and K.Horiuchi, "Type Inference in Prolog and its Applications", Proc. of 9th International Joint Conference on Artificial Intelligence, Los Angels, 1985.

[7] Kanamori,T. and T.Kawamura, "Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation", ICOT Technical Report, TR-279, 1987.

[8] Kanamori,T., K. Horiuchi and T.Kawamura, "Detecting Termination of Logic Programs Based on Abstract Hybrid Interpretation", to appear, ICOT Technical Reports, 1987.

[9] Maeji,M. and T.Kanamori, "Top-down Zooming Diagnosis of Logic Programs" to appear, ICOT Technical Reports, 1987.

[10] Mellish,C.S., "Some Global Optimizations for A Prolog Compiler", J. Logic Programming, pp.43-66, 1985.

[11] Mellish,C.S., "Abstract Interpretation of Prolog Programs", Proc. of 3rd International Conference on Logic Programming, London, 1986.

[12] Sawamura,H.and T.Takeshima, "Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and their Application to Prolog Optimization", Proc. of the Symposium on Logic Programming, pp.200-207, 1985.

[13] Tamaki,H.and T.Sato, "OLD Resolution with Tabulation", Proc. of 3rd International Conference on Logic Programming, pp.84-98, London, 1986.