ICOT Technical Report: TR-398

TR-398

Detecting Termination of Logic Programs Based on Abstract Hybrid Interpretation

by
T. Kanamori, T. Kawamura
and K. Horiuchi(Mitsubishi)

December, 1987

Detecting Termination of Logic Programs Based on Abstract Hybrid Interpretation

Tadashi KANAMORI   Tadashi KAWAMURA   Kenji HORIUCHI

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

Abstract

This paper presents a framework for detecting termination 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. Termination is detected by finding a subset of arguments for each mode pattern during the mode analysis process such that some measure computed from the symbolic complexities of the arguments is bounded downwards and always decreases in recursions. This method for Prolog programs is an adaptation of Boyer and Moore's method for Lisp programs.

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

Contents

1. Introduction

Termination is a basic property of programs. Detection of termination is important not only from the theoretical point of view but also from the practical point of view, since nonterminating programs are very problematic in debugging. Though the termination property is undecidable in general, many techniques for guaranteeing it have been developed in various frameworks [1],[5]. As for functional programs, Boyer and Moore [1] investigated how to detect termination of Lisp programs, which is crucial for well-founded induction in their verification. Dershowitz [5] studied termination of term rewriting systems by considering more general well-founded ordering on terms. As for relational programs like Prolog, because variables of Prolog programs are freely instantiatable [6], detection of termination is harder than the usual one so that just a few works have been done ([6], [15]).

This paper presents a framework for detecting termination of Prolog programs by abstract interpretation ([2],[3]). 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. Termination is detected by finding a subset of arguments for each mode pattern during the mode analysis process such that some measure computed from the symbolic complexities of the arguments is bounded downwards and always decreases in recursions. This method for Prolog programs is an adaptation of Boyer and Moore's method for Lisp programs.

This paper is organized as follows: After presenting the hybrid interpretation of Prolog programs in Section 2, we will show a method for inferring modes in Section 3, because the mode analysis plays a crucial role in our termination detection. Then, in Section 4, we will show a termination detection method based on the mode analysis method. In Section 5, different formulations of termination and their detection are discussed.

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. 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 an occurence of variable Z, and θ, σ, τ for substitutions.

2. Standard Hybrid Interpretation of Logic Programs

In this section, we will first present a basic hybrid interpretation method of Prolog programs [16], 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, ..., G_k be a sequence of labels of the nodes and θ_1, θ_2, ..., θ_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, ..., 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, ..., 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.

(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.

Example 2.1.1 An OLDT structure of "$reach(a, Y_0)$" is depicted in Figure 2.1.1. The underline denotes the lookup node, and the dotted line denotes the association from the lookup node.

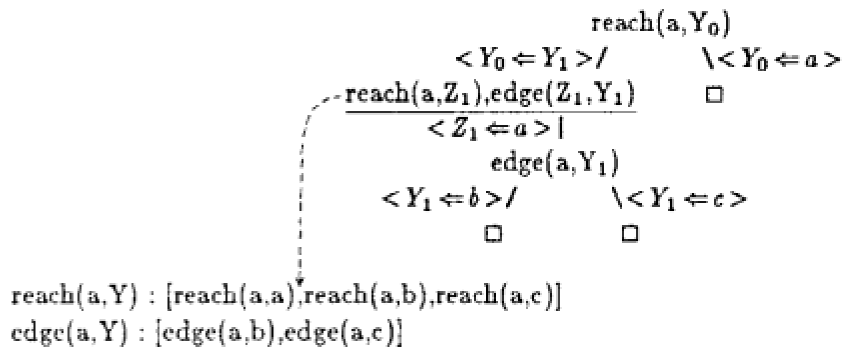


Figure 2.1.1 OLDT Structure

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. θ. (Without loss of generality, we assume that the m.g.u. θ 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 A_1 and $B\tau$ are unifiable, say by an m.g.u. θ. (Again, we assume that the m.g.u. θ substitutes a term consisting of fresh variables for every variable in A_1 and the definite clause.) The negative clause (or possibly null clause) "$A_2\theta, \ldots, A_n\theta$" is called the *OLDT resolvent*.

The restriction of the substitution θ 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 θ_i, where θ_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 θ_i, where θ_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 an instance 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 $A\tau$ to the last of the solution list of A in Tb, if $A\tau$ 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 [16].

```
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 [ ].

reach(a,Y$_0$)

reach(a,Y) : [ ]

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 an instance (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 $reach(a, Y)$.

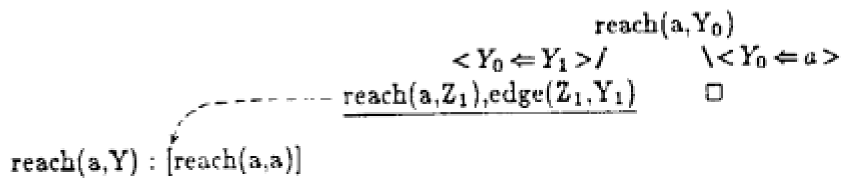


Figure 2.1.3 Basic Hybrid Interpretation at Step 2

† Note that it is a lookup node when the leftmost atom of the new negative clause is an *instance*, not a *variant*, of some key in Tb. (cf. the definition in our previous papers [9],[10].)

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.

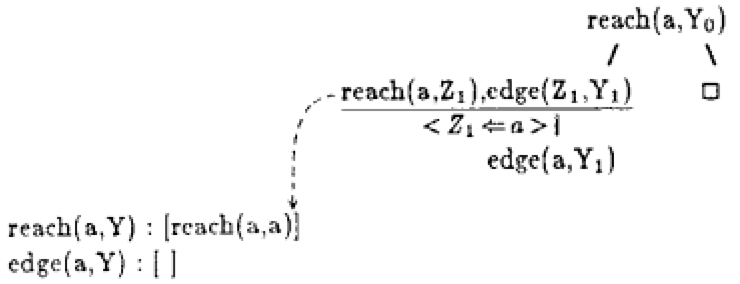
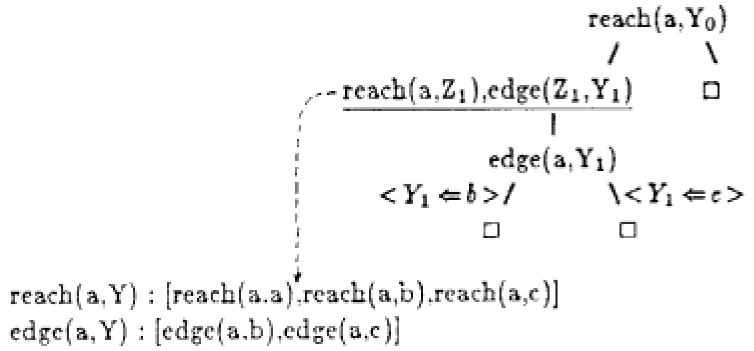


Figure 2.1.4 Basic Hybrid Interpretation at Step 3

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,Y₀)
reach(a,Z₁),edge(Z₁,Y₁)
edge(a,Y₁)
<Y₁ ⇐ b>
<Y₁ ⇐ c>
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 $reach(a,Y)$.

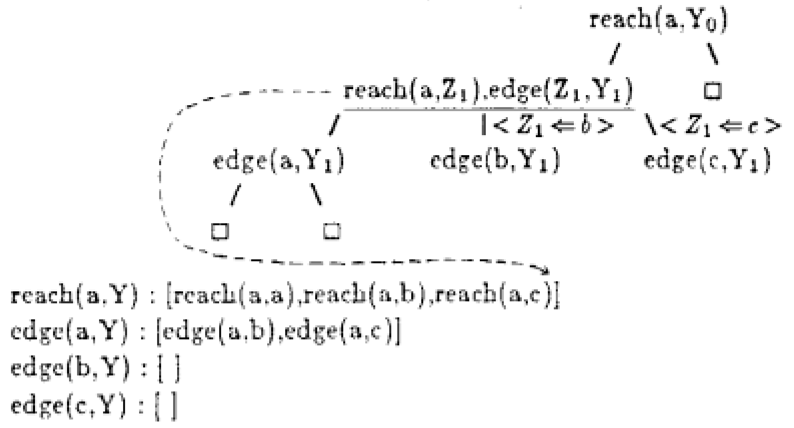
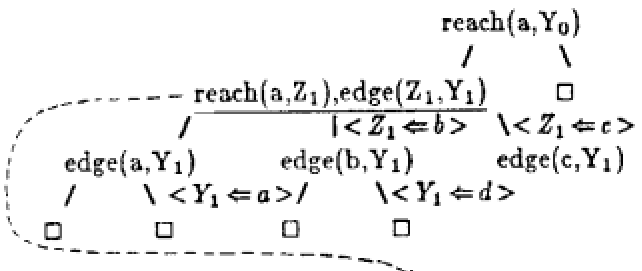


Figure 2.1.6 Basic Hybrid Interpretation at Step 5

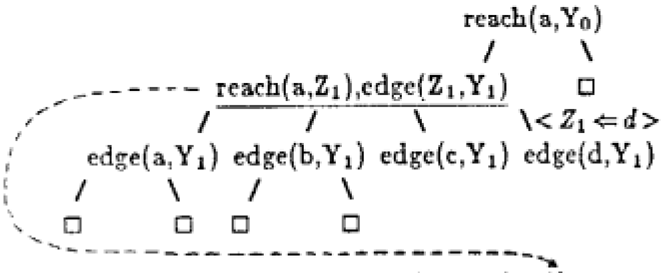
Sixthly, the left new solution node "$edge(b, Y_1)$" is OLDT resolved, and one new solution $reach(a, d)$ is added to the solution list of $reach(a, Y)$.



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. It is a basis of the abstract interpretation in this paper that 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 strategy.)

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 [16] in one respect (see [9]), the differences does not affect the proof of the soundness and the completeness. See Tamaki and Sato [16] 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. θ, 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, σ) is a serach tree whose root node is labelled with (G, σ). The clause part of each label is a sequence "α_1, α_2, ..., α_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, σ) is a path in a search tree of (G, σ) from the root to a node labelled with $(\Box, \tau)$. The answer *substitution of the refutation* is the substitution τ, 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 ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", σ) 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. θ.
(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 $\alpha_1\sigma$ and B_0 are unifiable, say by an m.g.u. θ.

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 ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", σ) 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. θ, the intermediate clause part is "$B_1, B_2, \ldots, B_m$, $[\alpha_1, \sigma]$, $\alpha_2, \ldots, \alpha_n$", and the intermediate substitution part τ_0 is θ.

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

```
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 m.g.u. of $\alpha_1\sigma$ and $B\tau$ ;
            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

Note that, when a node is labelled with (G, σ), the substitution part σ 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 σ_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 τ_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 ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", σ) is a lookup node when the leftmost atom $\alpha_1\sigma$ is an instance of an already existing key in the solution table, and is a solution node otherwise $(n \geq 1)$.

The *initial OLDT structure* of (G, σ) 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, σ), 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 ("$reach(a, Y_0)$",<>).

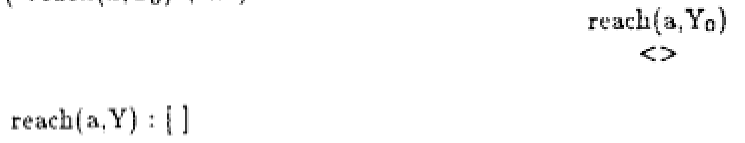


Figure 2.2.2 Modified Hybrid Interpretation at Step 1

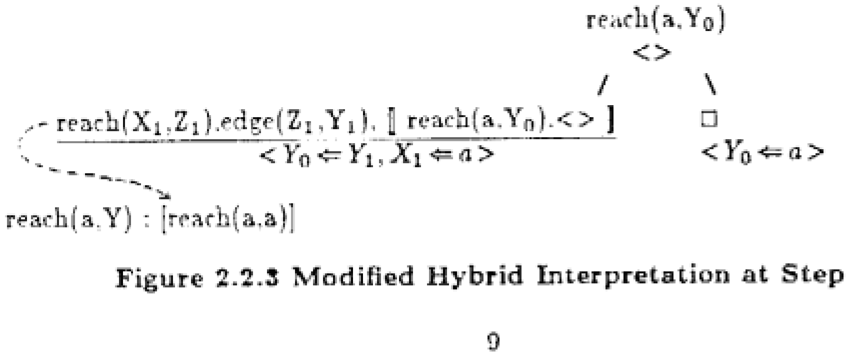
Secondly, the root node ("$reach(a, Y_0)$",<>) is OLDT resolved using the program to generate two child nodes. The intermediate label of the left child node is

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

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 $(\Box, < Y_0 \Leftarrow a, X_1 \Leftarrow a >)$. (When the clause part of the label is $\Box$, 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.

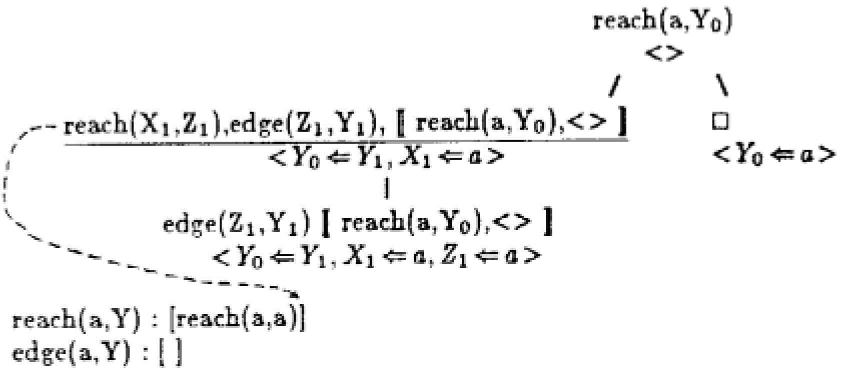

reach(a,Y₀)
<>
reach(X₁,Z₁),edge(Z₁,Y₁), [reach(a,Y₀),<>]
<Y₀ ⇐ Y₁, X₁ ⇐ a>
edge(Z₁,Y₁) [reach(a,Y₀),<>]
<Y₀ ⇐ Y₁, X₁ ⇐ a, Z₁ ⇐ a>
□
<Y₀ ⇐ a>
reach(a,Y) : [reach(a,a)]
edge(a,Y) : [ ]

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.

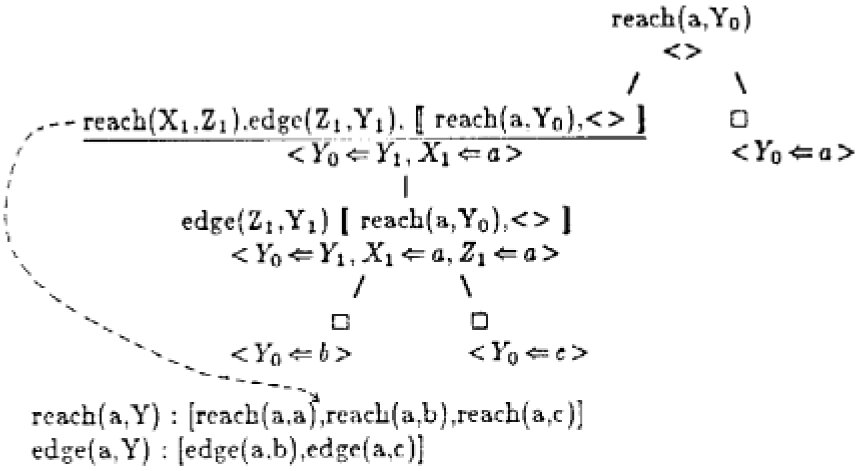

reach(a,Y₀)
<>
reach(X₁,Z₁),edge(Z₁,Y₁), [reach(a,Y₀),<>]
<Y₀ ⇐ Y₁, X₁ ⇐ a>
edge(Z₁,Y₁) [reach(a,Y₀),<>]
<Y₀ ⇐ Y₁, X₁ ⇐ a, Z₁ ⇐ a>
□
<Y₀ ⇐ b>
□
<Y₀ ⇐ c>
□
<Y₀ ⇐ a>
reach(a,Y) : [reach(a,a),reach(a,b),reach(a,c)]
edge(a,Y) : [edge(a,b),edge(a,c)]

Figure 2.2.5 Modified Hybrid Interpretation at Step 4

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 θ for the variables in $B\tau$ at step (B), and those in τ_i for variables not in $A_{i+1}\sigma_{i+1}$ at step (C) in Figure 2.2.1.

3. Mode Analysis by Abstract Interpretation

Suppose we would like to know that, when a goal "$div(X, D+1, Q)$" with its first argument X and its second argument $D+1$ instantiated to ground terms is executed, it either succeeds or fails finitely. Here div, $<$ and sub are defined by

```
div(X,D+1,0) :- X<D+1.
div(X,D+1,Q+1) :- sub(X,D+1,Y), div(Y,D+1,Q).
0<D+1.
X+1<D+1 :- X<D.
sub(X,0,X).
sub(X+1,D+1,Y) :- sub(X,D,Y).
```

(For notational convinience, we will use $X+1$ and $X+2$ instead of $suc(X)$ and $suc(suc(X))$.) When the first clause is used first to succeed, the desired termination property is reduced to that of $X < D+1$. When the second clause is used first, the goal is reduced to $sub(X, D+1, Y), div(Y, D+1, Q')$. If we can assume the termination property for $div(Y, D+1, Q')$ when its first argument Y is instantiated to a ground term, the detection of the desired termination is reduced to that of $sub(X, D+1, Y)$. It is, however, crucial to know that the third argument Y is instantiated to a ground term after the first subgoal $sub(X, D+1, Y)$ succeeds in order to resort to the termination property of goal $div(Y, D+1, Q')$. More generally, the first and the second arguments of div invoked from the top-level goal are always ground terms at calling time and the third argument is always a ground term at exiting time. Similarly, so are the first and the second arguments of sub 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 [12],[13] and Debray [4] from the point of view in Section 2.3.

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,
$\emptyset$: the emptyset of terms.

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

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

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

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

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 μ 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 $P \cup \{G\}$, μ 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 $A\mu \cap B\nu \neq \emptyset$. 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, μ)) 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 nalysis 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 termination 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, μ) is a search tree whose root node is labeled with (G, μ). The clause part of each label is a sequence "α_1, α_2, $\dots$, α_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, μ) is a path in a search tree of (G, μ) from the root to a node labelled with $(\Box, \nu)$. The *answer substitution of the refutation* is the mode substitution ν 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, $X_1, X_2, \dots, X_k$ all the variables in A, μ a mode substitution of the form
$<X_1 \Leftarrow \underline{m_1}, X_2 \Leftarrow \underline{m_2}, \dots, X_k \Leftarrow \underline{m_k}>$,
B an atom, $Y_1, Y_2, \dots, Y_l$ all the variables in B, and ν a mode substitution of the form
$<Y_1 \Leftarrow \underline{n_1}, Y_2 \Leftarrow \underline{n_2}, \dots, Y_l \Leftarrow \underline{n_l}>$,
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 τ?

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 η 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 μ and that of Z in s_j from the mode substitution ν, 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 $Z/ < t[Z] \Leftarrow \underline{m} >$. 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 $\underline{ground}$. 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 $\underline{m}$ 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 μ and ν 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 τ?

As has been just shown, we can overestimate the mode assigned to each variable Z in t_i from the mode substitution μ. By collecting these modes assignment for all variables, we can overestimate the mode substitution λ 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 λ obtained above, we can obtain the mode substitution ν'

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

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

Let λ 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/λ, 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 λ be $< X \Leftarrow \underline{ground}, L \Leftarrow \underline{ground} >$. Then

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

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

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

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

(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 μ and the one by the one-way propagation $propagate(\nu, \eta)$.

Example 3.2.2.4 Let ν 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 ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", μ) 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 α_1 and B_0 is unifiable, say by an m.g.u. η.

(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 α_1 is a variant of B, say by a renaming η.

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.

A node labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", μ) 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.

```
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

3.3. An Example of Mode Analysis

We will show an example of mode analysis, which is going to be used in Section 4.3.

Example 3.3 Let *div*, < and *sub* be predicates defined as before. Then, the mode analysis generates the following OLDT structure of $div(X, D+1, Q) < X, D \Leftarrow \underline{ground} >$ similarly to *reach* in Example 2.3. First, the initial OLDT structure is generated. The root node of the search tree is a solution node. The solution table contains only one entry with its key $div(X, D+1, Q) < X, D \Leftarrow \underline{ground} >$ and its solution list [ ].

$$\begin{tikzcd}[column sep=tiny]
& \begin{array}{c} \mathrm{div}(X_0,D_0+1,Q_0) \\ < X_0, D_0 \Leftarrow \underline{ground} > \end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
\begin{array}{c} X_1 < D_1+1, [\![\,]\!] \\ < X_1, D_1 \Leftarrow \underline{ground} > \end{array} & & \begin{array}{c} \mathrm{sub}(X_2,D_2+1,Y_2),\mathrm{div}(Y_2,D_2+1,Q_2),[\![\,]\!] \\ < X_2, D_2 \Leftarrow \underline{ground} > \end{array}
\end{tikzcd}$$

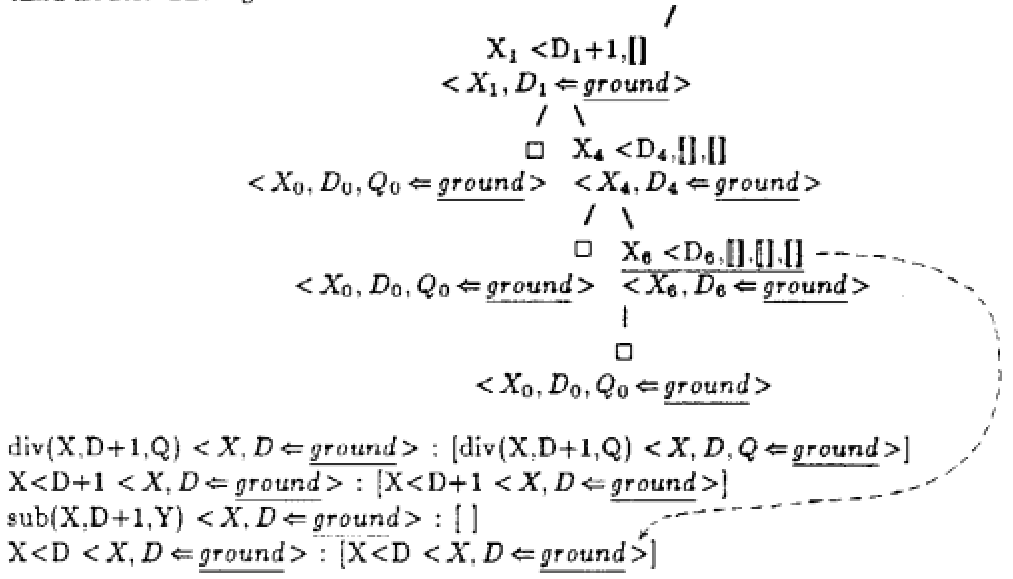
div(X,D+1,Q) $< X, D \Leftarrow \underline{ground} >$: [ ]
X<D+1 $< X, D \Leftarrow \underline{ground} >$: [ ]
sub(X,D+1,Y) $< X, D \Leftarrow \underline{ground} >$: [ ]

Figure 3.3.1. Mode Analysis at Step 2

Secondly, the root node is OLDT resolved using the program to generate two child nodes. The intermediate clause part of the left child node is
"$X_1 < D_1 + 1$,
$[\![div(X_0, D_0+1, Q_0), < X_0, D_0 \Leftarrow \underline{ground} >, < X_0 \Leftarrow X_1, D_0 \Leftarrow D_1, Q_0 \Leftarrow 0 >]\!]$",
hence it is immediately the clause part of the generated node. The intermediate clause part of the right child node is

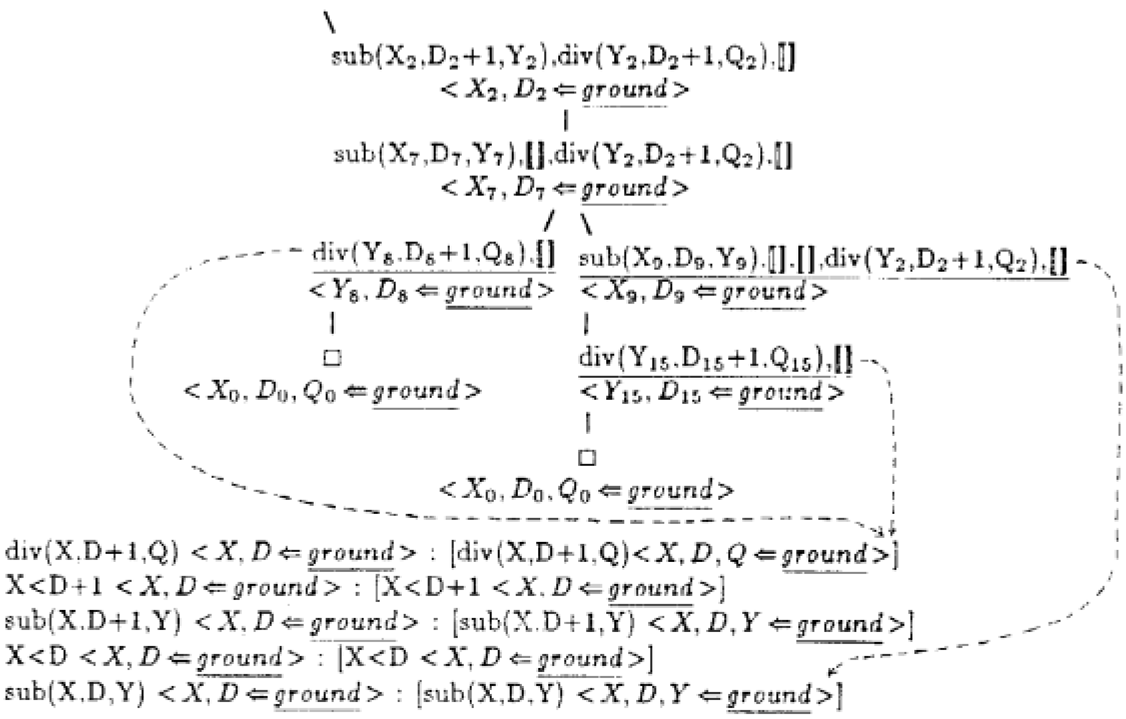
"$sub(X_2, D_2 + 1, Y_2)$, $div(Y_2, D_2 + 1, Q_2)$,
[$div(X_0, D_0 + 1, Q_0), < X_0, D_0 \Leftarrow \underline{ground} >, < X_0 \Leftarrow X_2, D_0 \Leftarrow D_2, Q_0 \Leftarrow Q_2 + 1 >$]",
hence it is immediately the clause part of the generated node as well. Both of these child nodes are solution nodes. (The quantities inside call-exit markers are omitted due to space limit so that they are depicted simply by [].)

Thirdly, the left solution node is OLDT resolved using the program to generate two child nodes. The right node is a new solution node.



X1 <D1+1,[]
<X1, D1 ⇐ ground>
X4 <D4,[],[]
<X0, D0, Q0 ⇐ ground>
<X4, D4 ⇐ ground>
X6 <D6,[],[],[]
<X0, D0, Q0 ⇐ ground>
<X6, D6 ⇐ ground>
<X0, D0, Q0 ⇐ ground>
div(X,D+1,Q) <X, D ⇐ ground> : [div(X,D+1,Q) <X, D, Q ⇐ ground>]
X<D+1 <X, D ⇐ ground> : [X<D+1 <X, D ⇐ ground>]
sub(X,D+1,Y) <X, D ⇐ ground> : [ ]
X<D <X, D ⇐ ground> : [X<D <X, D ⇐ ground>]

Figure 3.3.2. Mode Analysis at Step 5



sub(X2,D2+1,Y2),div(Y2,D2+1,Q2),[]
<X2, D2 ⇐ ground>
sub(X7,D7,Y7),[],div(Y2,D2+1,Q2),[]
<X7, D7 ⇐ ground>
div(Y8,D8+1,Q8),[]
<Y8, D8 ⇐ ground>
sub(X9,D9,Y9),[],[],div(Y2,D2+1,Q2),[]
<X9, D9 ⇐ ground>
<X0, D0, Q0 ⇐ ground>
div(Y15,D15+1,Q15),[]
<Y15, D15 ⇐ ground>
<X0, D0, Q0 ⇐ ground>
div(X,D+1,Q) <X, D ⇐ ground> : [div(X,D+1,Q)<X, D, Q ⇐ ground>]
X<D+1 <X, D ⇐ ground> : [X<D+1 <X, D ⇐ ground>]
sub(X,D+1,Y) <X, D ⇐ ground> : [sub(X,D+1,Y) <X, D, Y ⇐ ground>]
X<D <X, D ⇐ ground> : [X<D <X, D ⇐ ground>]
sub(X,D,Y) <X, D ⇐ ground> : [sub(X,D,Y) <X, D, Y ⇐ ground>]

Figure 3.3.3. Mode Analysis at Step 10

Fourthly, the right solution node is OLDT resolved using the program to generate two child node. The right child node is a lookup node. The association associates the lookup node to the head of $X < D < X, D \Leftarrow \underline{ground} >$'s solution list.

Fifthly, the lookup node is OLDT resolved using the solution table.

Sixthly, the right solution node is OLDT resolved using the program to generate one child solution node.

Seventhly, the solution node is OLDT resolved to generate two nodes. Both the right node and the left node are lookup nodes.

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

The final solution table says that

(a) *div*, *sub* and < are always called with its first and second arguments instantiated to ground terms.

(b) When *div*, *sub* and < succeed, their arguments are instantiated to ground terms.

4. Termination Detection Based on Abastract Hybrid Interpretation

In this section, we will show a termination detection method to check recursions during the process of mode analysis.

4.1 Termination of Prolog Programs

(1) OLD Termination

Because variables in Prolog programs are freely instantiatable, we need to define the termination property of Prolog programs in a slightly different way.

Example 4.1.1 Let *loop* be a predicate defined by

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

When any goal of the form *loop*(t) is executed, it never terminates.

Example 4.1.2 Let *reverse* be a predicate 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).
```

When a goal *reverse*($[X|L], M$) is executed with its all variables uninstantiated, it returns infinite number of solutions

reverse($[X_1]$,$[X_1]$),
reverse($[X_1,X_2]$,$[X_2,X_1]$),
reverse($[X_1,X_2,X_3]$,$[X_3,X_2,X_1]$),
⋮

and there exist execution paths with arbitrary length. Hence, when it is combined with other goals, it might never terminate. For example,

```
    ?- reverse([X|L],M), length(M,0).
```

Example 4.1.3 Let *esrever* be a predicate defined by

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

Even if a goal of the form $esrever([X|L], M)$ is executed with its first argument instantiated to a ground term, there exist execution paths with arbitrary length, because $append(N, [X], M)$ returns infinite number of solutions

```
append([ ],[X],[X]),
append([X₁],[X],[X₁,X]),
append([X₂,X₁],[X],[X₂,X₁,X]),
  ⋮
```

Hence, when it is combined with other goals, it might never terminate. For example,

?- esrever([X|L],M), length(M,0).

Example 4.1.4 Let *reverse* be the predicate as before. When a goal of the form $reverse(L, M)$ is executed with its first argument instantiated to a ground term, it always terminates. Here, it is crucial that *reverse* instantiates its second argument to a ground term when it succeeds.

Example 4.1.5 Let *append* be the predicate as before. When a goal of the form $append(N, [X], M)$ is executed with its first argument instantiated to a ground term, it always terminates.

An atom A is said to be *(OLD) terminating* when there is no infinite execution path in the OLD tree of A, i.e., it succeeds or fails finitely. A mode-abstracted atom $A\mu$ is said to be *(OLD) terminating* when any atom in the mode-abstracted atom is (OLD) terminating.

Example 4.1.6 The following mode-abstracted atoms

loop(X)$<X \Leftarrow any>$,
reverse([X|L],M)$<L \Leftarrow any>$,
esrever([X|L],M)$<L \Leftarrow ground>$

are not (OLD) terminating, while

reverse(L,M)$<L \Leftarrow ground>$,
append(N,[X],M)$<N, X \Leftarrow ground>$,

are (OLD) terminating.

(2) Symbolic Complexity Measure

A mapping m is called a *measure of* atom A, when it satisfies the following conditions.

(a) m is a mapping from the set of atoms to a well-founded set $(W, \prec)$.

(b) For any atom B and any substitution θ, $m(B\theta) \preceq m(B)$.

(c) Suppose that there is a path in the OLD tree of A starting from a node with its leftmost atom $p(t_1, t_2, \ldots, t_n)$ and ending with a node with its leftmost atom $p(s_1, s_2, \ldots, s_n)$. Let $\theta_1, \theta_2, \ldots, \theta_i$ be the labels of the edges on the path, and θ the composed substitution $\theta_1\theta_2 \cdots \theta_i$. Then $m(p(s_1, s_2, \ldots, s_n)) \prec m(p(t_1, t_2, \ldots, t_n)\theta)$.

An atom A is said to be *(OLD) terminating by* $\mathcal{M}$ when $\mathcal{M}$ is a set of all A's measures. In particular, when an atom A is terminating by $\{\}$, it is said to be *diverging*.

Theorem 4.1 An atom A is terminating if and only if there exists a measure of A.

Proof. Suppose that the OLD tree of atom A extended as far as possible is finite. Let $m(B)$ be the multiset of the lengths of all paths from the root to the leaves in the OLD tree of atom B. (We assume the multiset ordering over the usual ordering $<$, i.e., a multiset S_1 is smaller than a multiset S_2 when S_1 is obtained from S_2 by replacing an element of S_2 with (possibly zero) elements smaller than the element.) Then, the measure m ovbiously satisfies the conditions (a),(b) and (c). The inverse direction of the proof is trivial. (cf. Shapiro [15] p.59, Lemma 3.9.)

Hence, existence of measure is a necessary and sufficient condition for guaranteeing termination. But, detection of termination is undecidable in general. In this section, we restrict our attentions to the following simplest class of measures. (The class of measures is easily extended so that more general ordering can be taken into consideration. See Section 7.) The *symbolic complexity* of term t is the number of symbols contained in t, and denoted by $|t|$. The *symbolic complexity measure* m of atom $p(t_1, t_2, \ldots, t_n)$ is the symbolic complexity of some argument $|t_i|$, and the i-th argument is called the *measured argument* of m. (Note that the symbolic complexities are bounded downwards by constant 0.)

Example 4.1.7 Let *sub* be a predicate defined by

```
sub(X,0,X).
sub(X+1,D+1,Y) :- sub(X,D,Y).
```

When X and D are ground terms, $|X|$ and $|D|$ are symbolic complexity measures of $sub(X, D+1, Y)$, and the first argument and the second argument are the measured arguments, respectively.

(3) Termination Lemmas

Termination of a given mode-abstracted atom can be detected by finding a symbolic complexity measure for each mode such that it always decreases in recursions.

In order to guarantee it, we resort to *termination lemmas*. A termination lemma is a theorem of the form ($l \geq 0$)

termination-lemma(lemma-name).
  $|s| < |t|$:- $B_1, B_2, \ldots, B_l$.
end.

which means that, when $B_1, B_2, \ldots, B_l$ hold and s, t are ground terms, the symoloc complexity $|s|$ is smaller than $|t|$.

Example 4.1.8 Let *suc-decrease* and *sub-decrease* be termination lemmas

termination-lemma(suc-decrease).
  |X| < |X+1|.
end.
termination-lemma(sub-decrease).
  |Y| < |X| :- sub(X,D+1,Y).
end.

These lemmas are utilized for detecting termination of *div* later.

Let $p(s_1, s_2, \ldots, s_n)\mu$ and $p(t_1, t_2, \ldots, t_n)\nu$ be two mode-abstracted atoms, and Γ be a set of atoms we can assume to hold. Suppose we have a termination lemma

$|u| < |v|$:- $B_1, B_2, \ldots, B_l$.

Then we can guarantee that

$m(p(s_1, s_2, \ldots, s_n)) < m(p(t_1, t_2, \ldots, t_n))$

under the mode substitutions μ, ν and the antecedant Γ if there exists a substitution λ such that

(a) $\lambda(B_1)$, $\lambda(B_2)$, $\ldots$, $\lambda(B_l)$ are all contained in Γ,

(b) $\lambda(u)$ is s_i, and $\lambda(v)$ is t_i, and

(c) s_i is ground under μ, and t_i is ground under ν.

Example 4.1.9 Consider mode-abstracted atoms

$X+1 < D+1 < X, D \Leftarrow \underline{ground}>$,

$X < D < X, D \Leftarrow \underline{ground}>$.

Then, by utilizing the termination lemma *suc-decrease*, we can guarantee that

$m(X < D) < m(X+1 < D+1)$

under the mode assumption $< X, D \Leftarrow \underline{ground}>$, $< X, D \Leftarrow \underline{ground}>$ and the antecedant $\{\ \}$, where m is either the symbolic complexity of the first argument or the symbolic complexity of the second argument. Similarly, by utilizing the termination lemma *sub-decrease*, we can guarantee that

$m(div(Y, D+1, Q)) < m(div(X, D+1, Q+1))$

under the mode assumption $< Y, D \Leftarrow \underline{ground} >$, $< X, D \Leftarrow \underline{ground} >$ and the antecedant $\{sub(X, D+1, Y)\}$, where m is the symbolic complexity of the first argument. (Note that the symbolic complexity of the third argument is not a measure due to the mode assumption.)

Remark. The termination lemmas play the same role as *induction lemmas* of Boyer-Moore Theorem Prover (BMTP), but the measured arguments found through the termination detection are not used for formulation of induction formulas in our system unlike BMTP. In this paper, we assume that the termination lemmas are prepared before termination detection.

4.2 Abstract Hybrid Interpretation for Termination Detection

A *search tree for termination detection* is a tree with its node labeled with a quadruple of a (generalized) negative clause, a mode substitution, a usual substitution and a set of atoms called *antecedant*. (For brevity, we will sometimes omit the term "for termination detection" hereafter in Section 4.) A *search tree of* (G, σ, μ, Γ) is a search tree whose root node is labeled with (G, σ, μ, Γ). The clause part of each qudruple is a sequence "α_1, α_2, ..., α_n" consisting of either atoms in the body of $P \cup \{G\}$ or call-exit markers of the form $[\![A, \sigma', \mu', \eta]\!]$. A *refutation* of (G, σ, μ, Γ) is a path in a search tree of (G, σ, μ, Γ) from the root to a node labelled with $(\Box, \tau, \nu, \Delta)$. The answer *substitution of the refutation* is the substitution τ, the answer *mode substitution of the refutation* is the mode substitution ν, the answer *antecedant* is the antecedant Δ, and the *solution of the refutation* is $G\tau\nu$.

A *solution table for termination detection* is a set of entries. Each entry consists of the *key*, *measure set*, and the *solution list*. The key is a trio $A\sigma\mu$. The measure set is a set of symbolic complexity measures. The solution list is a list of mode-abstracted atoms $A\mu_i$, called *solutions*, whose all elements are greater than the corresponding mode-abstracted atom of 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 termination 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 termination 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 termination detection* as follows.

```
OLDT-resolve(("$\alpha_1, \alpha_2, \ldots, \alpha_n$", $\sigma$, $\mu$, $\Gamma$) : 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 $\eta$ be the m.g.u. of $\alpha_1$ and $B_0$ ;
        let $G_0$ be a negative clause "$B_1, B_2, \ldots, B_m, [\![\alpha_1, \sigma, \mu, \eta]\!], \alpha_2, \ldots, \alpha_n$";
        let $\tau_0$ be $\theta$ ;
        let $\nu_0$ be $propagate(\mu, \eta)$ ;                                  — (A)
        let $\Delta_0$ be $\Gamma$ ;
      when a lookup node is OLDT resolved with "$B\nu$" in $Tb$
        let $\theta$ be the empty substitution <> ;
        let $\eta$ be the renaming of $B$ to $\alpha_1$ ;
        let $G_0$ be a negative caluse "$\alpha_2, \ldots, \alpha_n$" ;
        let $\tau_0$ be $\sigma\theta$ ;
        let $\nu_0$ be $\mu \vee propagate(\nu, \eta)$ ;                           — (B)
        let $\Delta_0$ be $\Gamma \cup \{\alpha_1\}$ ;
    endcase
    while the leftmost of $G_i$ is a call-exit marker $[\![A_{i+1}, \sigma_{i+1}, \mu_{i+1}, \eta_{i+1}]\!]$ do
        let $G_{i+1}$ be $G_i$ other than the leftmost call-exit marker ;
        let $\tau_{i+1}$ be $\sigma_{i+1}\tau_i$ ;
        let $\nu_{i+1}$ be $\mu_{i+1} \vee propagate(\nu_i, \eta_{i+1})$ ;              — (C)
        let $\Delta_{i+1}$ be $\Delta_i \cup \{A_{i+1}\}$ ;
        add $A_{i+1}\nu_{i+1}$ to the last of $A_{i+1}\sigma_{i+1}\mu_{i+1}$'s solution list if it is not in it ;
        $i := i + 1$ ;
    endwhile
    $(G_{new}, \sigma_{new}, \mu_{new}, \Gamma_{new}) := (G_i, \tau_i, \nu_i, \Delta_i)$ ;
    return recursion-check(($G_{new}$, $\sigma_{new}$, $\mu_{new}$, $\Gamma_{new}$)).

recursion-check(($G$, $\sigma$, $\mu$, $\Gamma$) : label) : label ;
    let $A$ be the leftmost atom of $G$ and $p$ be its predicate;
    if $G$ contains call-exit markers with predicate $p$ more than a fixed bound $limit$
    then stop with warning "So many call patterns of $p$?"
    else let $[\![A_k, \sigma_k, \mu_k, \eta_k]\!]$ be the leftmost call-exit marker such that $A_k$'s predicate is $p$ ;
        let $[\![A_1, \sigma_1, \mu_1, \eta_1]\!]$, $[\![A_2, \sigma_2, \mu_2, \eta_2]\!]$, ..., $[\![A_{k-1}, \sigma_{k-1}, \mu_{k-1}, \eta_{k-1}]\!]$ be
            all the call-exit markers left to the leftmost call-exit marker ;
        let $\tau$ be the composed substitution $\sigma_k\sigma_{k-1}\cdots\sigma_2\sigma_1\sigma$ ;
        let $\mathcal{M} = \{m_1, m_2, \ldots, m_l\}$ be $A_k\sigma_k\mu_k$'s measure ;
        for $i$ from 1 to $l$
            case
              $m_i(A\sigma) < m_i(A_k\tau)$ is guaranteed
              under $\mu$, $\mu_k$ and $\Gamma\tau$ by some termination lemma : keep $m_i$ in $\mathcal{M}$ as it is ;
              $m_i(A\sigma) < m_i(A_k\tau)$ is not guaranteed
              under $\mu$, $\mu_k$ and $\Gamma\tau$ by any termination lemma : eliminate $m_i$ from $\mathcal{M}$ ;
            end
        if $\mathcal{M}$ is $\emptyset$ then stop with warning "$A_k\tau$ calls $A\sigma$ ?" else return $(G, \sigma, \mu, \Gamma)$ ;
```

Figure 4.2 OLDT Resolution for Termination Detection

A node of OLDT structure (Tr, Tb, As) labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", σ, μ, Γ) 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 $\alpha_1\sigma$ and B_0 are unifiable, say by an m.g.u. θ.

(b) The node is a lookup node of Tr and there is some solution $B\nu$ in the associated solution list of the lookup node such that α_1 is a variant of B, say by a renaming η.

The precise algorithm of OLDT resolution for termination detection is shown in Figure 4.2.

A node labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", σ, μ, Γ) is a lookup node when there already exeist a key $B\tau\nu$ such that $\alpha_1\sigma$ is an instance of $B\tau$ and $\alpha_1\mu$ is a variant of $B\nu$, and is a solution node otherwise ($n \geq 1$). When a triple $\alpha\sigma\mu$ of a solution node is registered to the solution table, its measure set is initialized to all the symbolic complexity measures of the arguments of α.

The *initial OLDT structure for termination detection* is a triple (Tr_0, Tb_0, As_0), where Tr_0 is a search tree consisting of just the root node labeled with ("$\alpha_1, \alpha_2, \ldots, \alpha_n$", σ, μ, $\{\ \}$), Tb_0 is a solution table consisting of just one entry whose key is $\alpha_1\sigma\mu$, solution list is [ ], and measure set is all the symbolic complexity measures of the arguments of α_1.

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 in Section 2.3.

4.3 Examples of Termination Detection

We will show two examples of how termination detection proceeds.

Example 4.3.1 Let *div*, < and *sub* be predicates defined as before in Example 3.3. Then, the termination detection generates the following OLDT structure of $div(X, D+1, Q) <><X \Leftarrow \underline{ground}>$ similarly to Example 3.3.

First, the initial OLDT structure below is generated. The antecedant part of the root node is initialized to { }.

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

"$X_1 < D_1 + 1$,
$[div(X_0, D_0 + 1, Q_0), <>, <X_0, D_0 \Leftarrow \underline{ground}>, <X_0 \Leftarrow X_1, D_0 \Leftarrow D_1, Q_0 \Leftarrow 0>]$",

hence it is immediately the clause part of the generated node. The clause part of the intermediate label of the right child node is

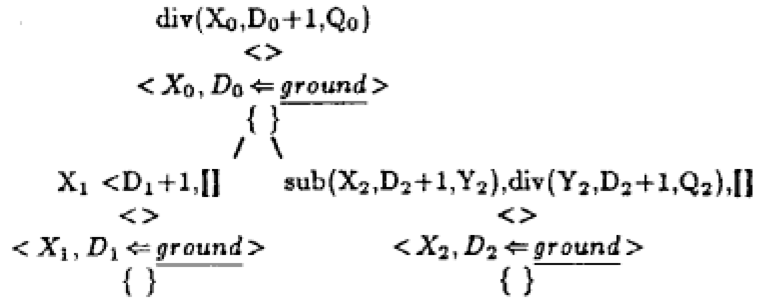
"$sub(X_2, D_2 + 1, Y_2)$, $div(Y_2, D_2 + 1, Q_2)$,
$[div(X_0, D_0+1, Q_0), <>, <X_0, D_0 \Leftarrow \underline{ground}>, <X_0 \Leftarrow X_2, D_0 \Leftarrow D_2, Q_0 \Leftarrow Q_2+1>]$",

hence it is immediately the clause part of the generated node.

Thirdly, the left solution node is OLDT resolved using the program to generate two child nodes. Since the label of the right child node is

"$X_4 < D_4$.$[X_1 < D_1 + 1, <>, <X_1, D_1 \Leftarrow \underline{ground}>, <X_1 \Leftarrow X_4 + 1, D_1 \Leftarrow D_4>]$.[]",

the atom $X_4 < D_4$ and the atom $X_1 < D_1 + 1 \overline{<X_1 \Leftarrow X_4 + 1, D_1 \Leftarrow D_4>}$ are compared for recursion check. Because $|X_4| < |X_4+1|$ when X_4 is a ground term, and $|D_4| < |D_4+1|$ when D_4 is a ground term, by the termination lemma *suc-decrese*, the 1st and the 2nd arguments remain measures of $X < D + 1$.

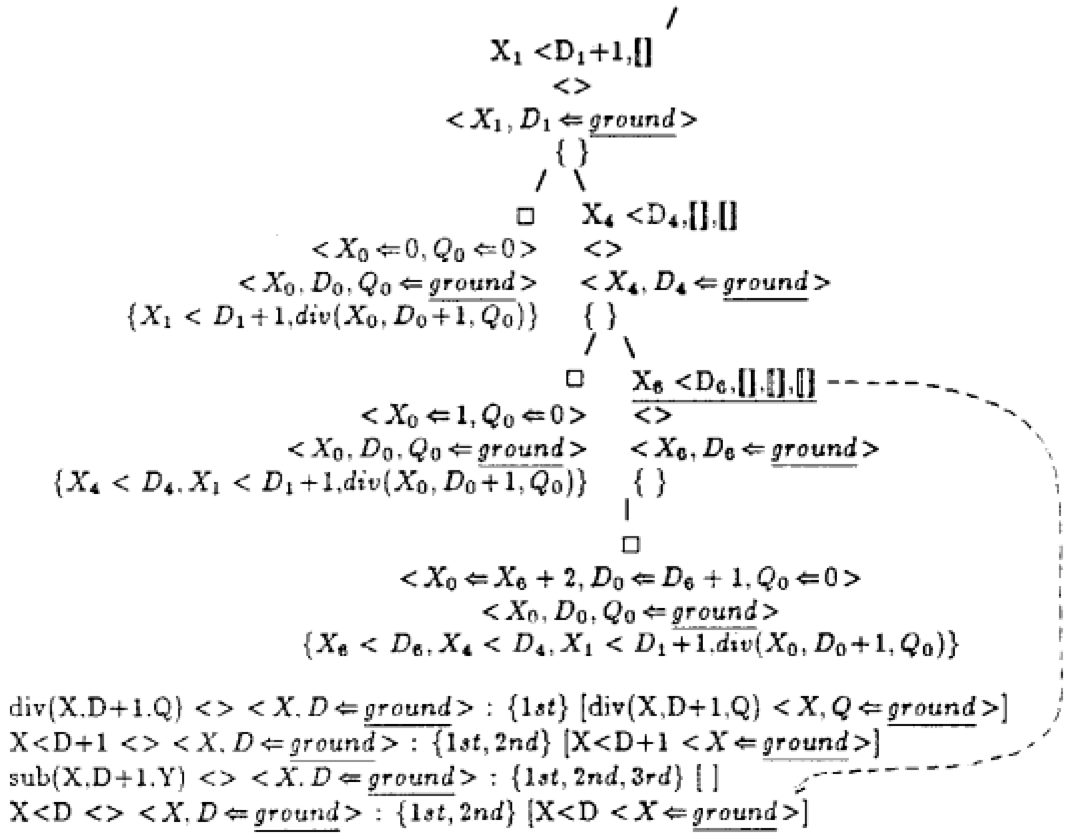


div(X,D+1,Q) <> $<X, D \Leftarrow \underline{ground}>$: $\{1st, 2nd, 3rd\}$ [ ]
X<D+1 <> $<X, D \Leftarrow \underline{ground}>$: $\{1st, 2nd\}$ [ ]
sub(X,D+1,Y) <> $<X, D \Leftarrow \underline{ground}>$: $\{1st, 2nd, 3rd\}$ [ ]

Figure 4.3.1 Termination Detection at Step 2

Fourthly, the right solution node is OLDT resolved using the program to generate two child node. The right child node is a lookup node, for which a similar recursion check is done.

Fifthly, the lookup node is OLDT resolved using the solution table.



div(X,D+1,Q) <> $<X, D \Leftarrow \underline{ground}>$: $\{1st\}$ [div(X,D+1,Q) $<X, Q \Leftarrow \underline{ground}>$]
X<D+1 <> $<X, D \Leftarrow \underline{ground}>$: $\{1st, 2nd\}$ [X<D+1 $<X \Leftarrow \underline{ground}>$]
sub(X,D+1,Y) <> $<X, D \Leftarrow \underline{ground}>$: $\{1st, 2nd, 3rd\}$ [ ]
X<D <> $<X, D \Leftarrow \underline{ground}>$: $\{1st, 2nd\}$ [X<D $<X \Leftarrow \underline{ground}>$]

Figure 4.3.2 Termination Detection at Step 5

Sixthly, the right solution node is OLDT resolved using the program to generate one child solution node. A similar recursion check is done using *suc-decrease*.

Seventhly, the solution node is OLDT resolved to generate two nodes. The left child node is a solution node, for which recursion check is done using *sub-decrese*. The right node is a lookup node.

The process proceeds in the same way. The final search tree (search subtree rooted with the right son of the root node) and solution table are shown in Figure 4.3.3.

The final solution table says that $div(X, D+1, Q)$ is terminating when it is executed with its first and second arguments instantiated to ground terms, since the first argument is the symbolic complexity measure common to all calling patterns of *div*. (It is terminating even if it is executed with only the first argument instantiated to a ground term, though the generated OLDT structure for termination detection will be twice as big as that of this example.)

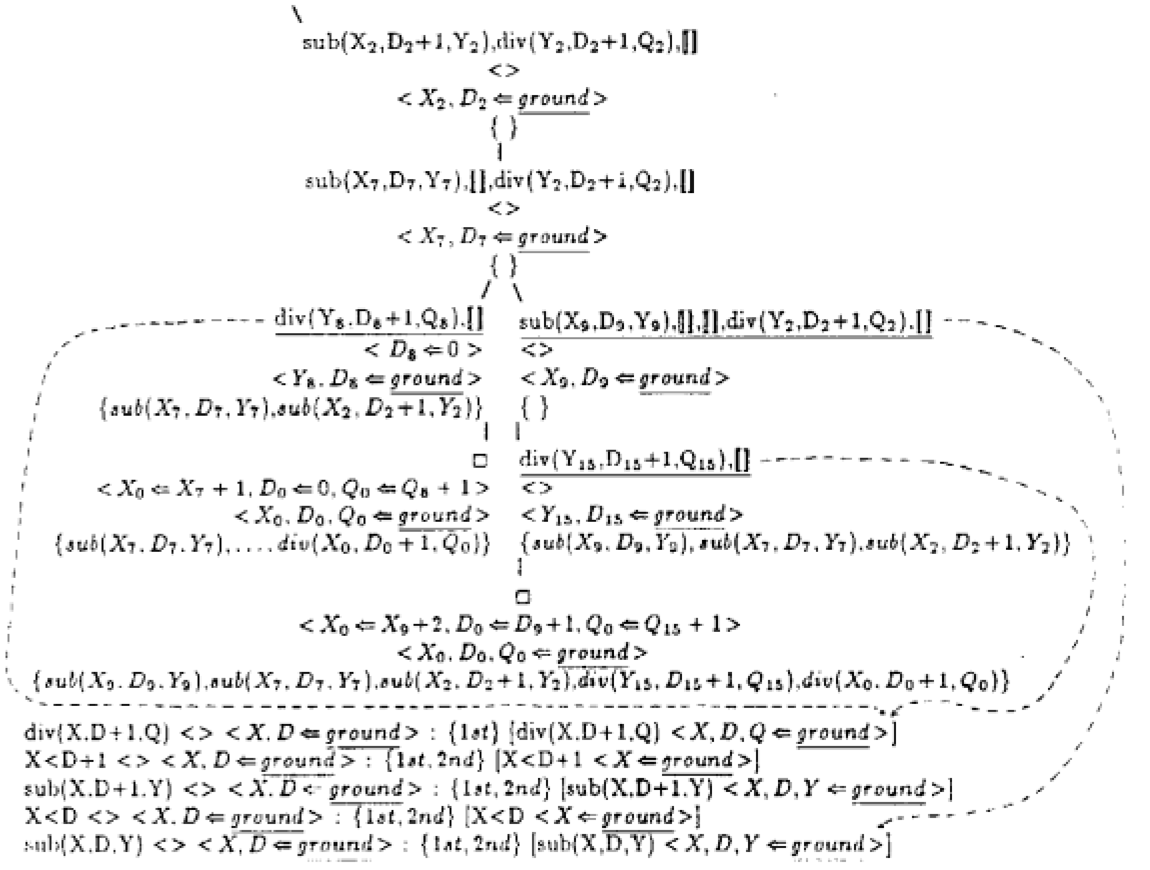

sub(X₂,D₂+1,Y₂),div(Y₂,D₂+1,Q₂),[]
<>
< X₂, D₂ ⇐ ground >
{ }
sub(X₇,D₇,Y₇),[],div(Y₂,D₂+1,Q₂),[]
<>
< X₇, D₇ ⇐ ground >
{ }
div(Y₈,D₈+1,Q₈),[]
< D₈ ⇐ 0 >
< Y₈, D₈ ⇐ ground >
{sub(X₇, D₇, Y₇),sub(X₂, D₂+1, Y₂)}
□
< X₀ ⇐ X₇ + 1, D₀ ⇐ 0, Q₀ ⇐ Q₈ + 1 >
< X₀, D₀, Q₀ ⇐ ground >
{sub(X₇, D₇, Y₇), . . . , div(X₀, D₀ + 1, Q₀)}
sub(X₉,D₉,Y₉),[],[],div(Y₂,D₂+1,Q₂),[]
<>
< X₉, D₉ ⇐ ground >
{ }
div(Y₁₅,D₁₅+1,Q₁₅),[]
<>
< Y₁₅, D₁₅ ⇐ ground >
{sub(X₉, D₉, Y₉), sub(X₇, D₇, Y₇), sub(X₂, D₂+1, Y₂)}
□
< X₀ ⇐ X₉+2, D₀ ⇐ D₉+1, Q₀ ⇐ Q₁₅ + 1 >
< X₀, D₀, Q₀ ⇐ ground >
{sub(X₉, D₉, Y₉), sub(X₇, D₇, Y₇), sub(X₂, D₂+1, Y₂), div(Y₁₅, D₁₅+1, Q₁₅), div(X₀, D₀+1, Q₀)}
div(X,D+1,Q) <> < X, D ⇐ ground > : {1st} [div(X,D+1,Q) < X, D, Q ⇐ ground >]
X<D+1 <> < X, D ⇐ ground > : {1st, 2nd} [X<D+1 < X ⇐ ground >]
sub(X,D+1,Y) <> < X, D ⇐ ground > : {1st, 2nd} [sub(X,D+1,Y) < X, D, Y ⇐ ground >]
X<D <> < X, D ⇐ ground > : {1st, 2nd} [X<D < X ⇐ ground >]
sub(X,D,Y) <> < X, D ⇐ ground > : {1st, 2nd} [sub(X,D,Y) < X, D, Y ⇐ ground >]

Figure 4.3.3 Termination Detection at Step 14

Remark. Though we have assumed that the measure sets in the solution table are initialized to the set of all symbolic complexities of the arguments of the key, it is more efficient to initialize them to that of the *ground* arguments of the key. For example, the measure set of $div(X, D+1, Q)$ in Example 4.3.1 is initialized to $\{1st, 2nd\}$, not $\{1st, 2nd, 3rd\}$.

Example 4.3.2 Let *qsort*, *part* and *append* be predicates defined by

```
qsort([ ],[ ]).
qsort([X|L],M) :- part(L,X,LA,LB), qsort(LA,MA), qsort(LB,MB), append(MA,[X|MB],M).
part([ ],X,[X],[ ]).
part([Y|L],X,[Y|LA],LB) :- X>Y, part(L,X,LA,LB).
part([Y|L],X,LA,[Y|LB]) :- X≤Y, part(L,X,LA,LB).
append([ ],M,M).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Then, the termination detection proceeds as follows. (Note that the first clause of *part* is wrong. This example is taken from Shapiro [15] p.60.)

Because of the wrong clause of *part*, we can't guarantee termination of *qsort* by symbolic complexity measure. ($qsort([X],M)$ is not smaller than $qsort([X],MA)$ when their first arguments are ground.) In general, the failure of guaranteeing termination does not always say that the atom is diverging, since we are considering a very restricted class of measures. However, a warning that termination of *qsort* is not guaranteed by simple measures due to some clause is helpful for diagnosis of nonterminating programs. (cf. Shapiro's approach in Section 3.4. of [15].)

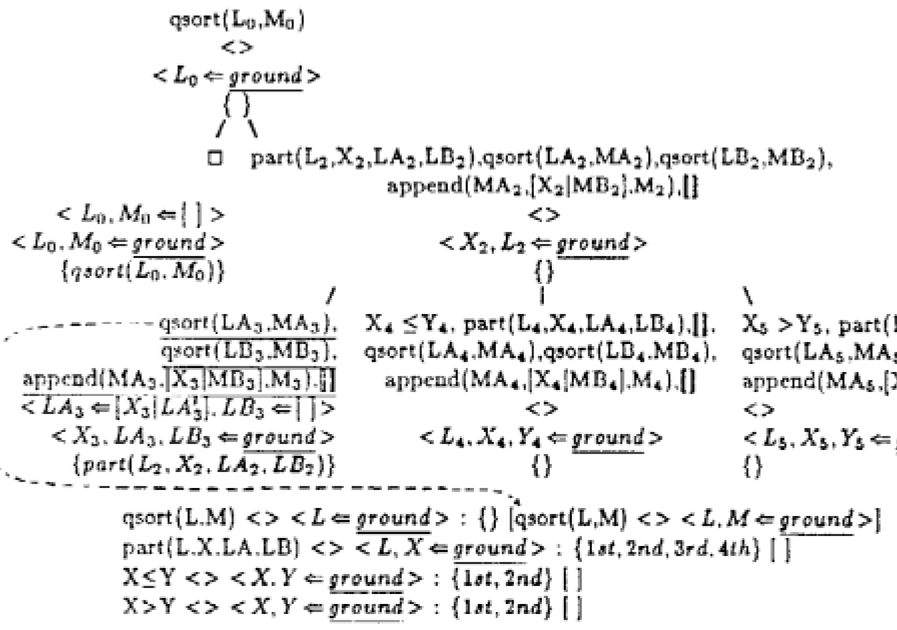

qsort(L0,M0)
<>
< L0 ⇐ ground >
{ }
□ part(L2,X2,LA2,LB2),qsort(LA2,MA2),qsort(LB2,MB2), append(MA2,[X2|MB2],M2),[]
< L0, M0 ⇐ [ ] >
< L0, M0 ⇐ ground >
{qsort(L0, M0)}
<>
< X2, L2 ⇐ ground >
{}
qsort(LA3,MA3), qsort(LB3,MB3), append(MA3,[X3|MB3],M3),[]
< LA3 ⇐ [X3|LA'3], LB3 ⇐ [ ] >
< X3, LA3, LB3 ⇐ ground >
{part(L2, X2, LA2, LB2)}
X4 ≤Y4, part(L4,X4,LA4,LB4),[], qsort(LA4,MA4),qsort(LB4,MB4), append(MA4,[X4|MB4],M4),[]
<>
< L4, X4, Y4 ⇐ ground >
{}
X5 >Y5, part(L5,X5,LA5,LB5),[] qsort(LA5,MA5),qsort(LB5,MB5), append(MA5,[X5|MB5],M5),[]
<>
< L5, X5, Y5 ⇐ ground >
{}
qsort(L,M) <> < L ⇐ ground > : {} [qsort(L,M) <> < L, M ⇐ ground >]
part(L,X,LA,LB) <> < L, X ⇐ ground > : {1st, 2nd, 3rd, 4th} [ ]
X≤Y <> < X, Y ⇐ ground > : {1st, 2nd} [ ]
X>Y <> < X, Y ⇐ ground > : {1st, 2nd} [ ]

Figure 4.3.4 Termination Detection of *qsort* at Step 3

5. Detection of Universal Termination and Existential Termination

5.1 Universal Termination and Existential Termination

In Section 2.1, atoms resolved in OLD resolution are always the leftmost ones in negative clauses. If it is permited to select arbitrary atoms in negative clauses, we have SLD-resolution. Then, what differences occur in the termination property ? Recall the five examples in Section 4.

Example 5.1.1 Let *loop* be the predicate as before. When any goal of the form $loop(t)$ is executed, it never terminates in any SLD-resolution.

Example 5.1.2 Let *reverse* be the predicate as before. When a goal $reverse([X|L], M)$ is executed by SLD-resolution with its all variables uninstantiated, it returns infinite number of solutions just like OLD-resolution in Example 4.1.2.

Example 5.1.3 Let *esrever* be the predicate as before. Suppose that the rightmost atom is always selected. Then, if a goal of the form $esrever([X|L], M)$ is executed with its first argument instantiated to a ground term, it always terminates finitely. Now, suppose that the leftmost atoms are always selected. Then, as Example 4.1.3, there exist execution paths with arbitrary length.

Example 5.1.4 Let *reverse* be the predicate as before, and suppose that the rightmost atoms are always selected. When a goal of the form $reverse(L, M)$ is executed with its first argument instantiated to a ground term, there exist execution path with arbitrary length, just as *esrever* in Example 4.1.4.

Example 5.1.5 Let *append* be the predicate as before. When a goal of the form $append(N, [X], M)$ is executed with its first argument instantiated to a ground term, it always terminates in any SLD-resolution.

An atom A is said to be *universally terminating* when there is no SLD tree of A with infinite execution path. A mode-abstracted atom $A\mu$ is said to be *universal terminating* when any atom in the mode-abstracted atom is universal terminating. An atom A is said to be *existentially terminating* when there is some SLD tree with no infinite execution path. A mode-abstracted atom $A\mu$ is said to be *existentially terminating* when any atom in the mode-abstracted atom is existentially terminating. When an atom (or mode-abstracted atom) is universally terminating, it is OLD terminating. When an atom (or mode-abstracted atom) is OLD terminating, it is existentially terminating.

Example 5.1.6 The mode-abstracted atom
    loop(X)$< X \Leftarrow \underline{any} >$,
    reverse([X|L],M)$< L \Leftarrow \underline{any} >$,
are not existentially terminating. The mode-abstracted atom
    esrever([X|L],M)$< L \Leftarrow \underline{ground} >$
is existentially terminating, but not OLD terminating. The mode-abstracted atom
    reverse(L,M)$< L \Leftarrow \underline{ground} >$,
is existentially terminating as well as OLD terminating, but not universally terminating. The mode-abstracted atom
    append(N,[X],M)$< N, X \Leftarrow \underline{ground} >$,
is both existentially terminating, OLD terminating and universally terminating.

5.2 Detection of Universal Termination

In order to detect universal termination, it is enough to check all SLD-trees whether they contain no infinite execution path.

Example 5.2 Let *append* be the predicate as before. Then there is only one OLDT structure for termination detection of $append(N, [X], M) < N, X \Leftarrow \underline{ground} >$ as below.

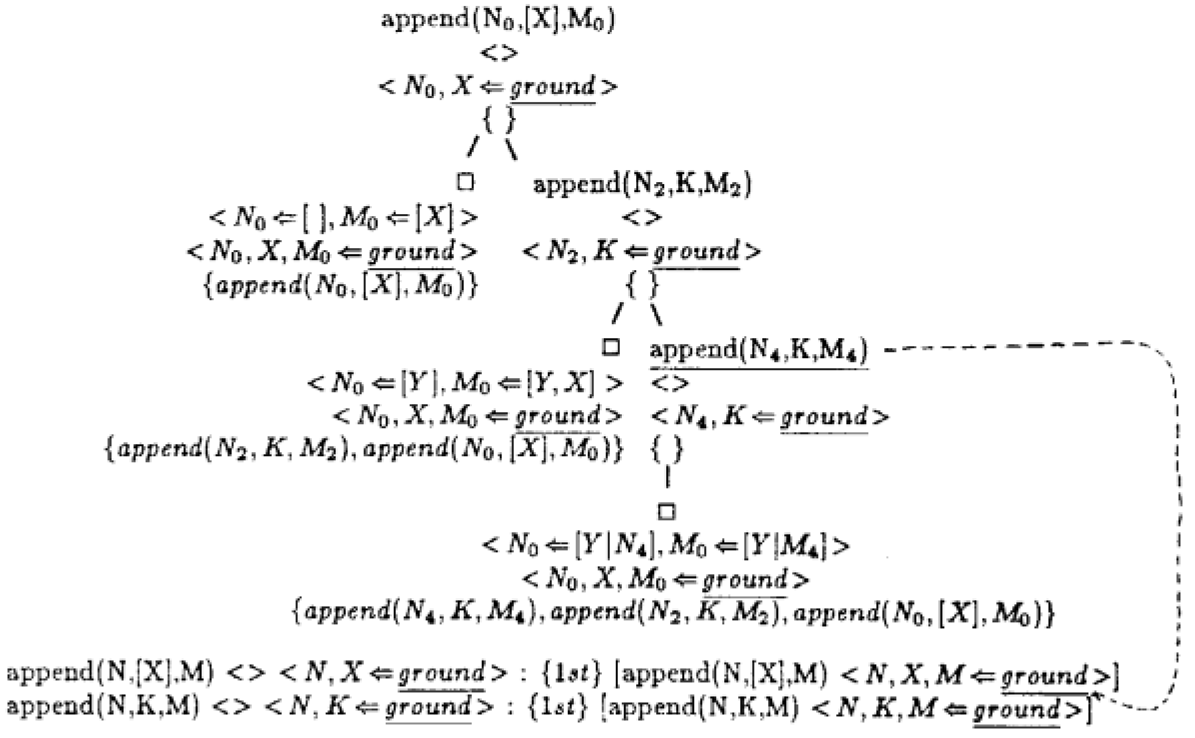

append(N0,[X],M0)
<>
< N0, X ⇐ ground >
{ }
□
append(N2,K,M2)
< N0 ⇐ [ ], M0 ⇐ [X] >
< N0, X, M0 ⇐ ground >
{append(N0, [X], M0)}
<>
< N2, K ⇐ ground >
{ }
□
append(N4,K,M4)
< N0 ⇐ [Y], M0 ⇐ [Y, X] >
< N0, X, M0 ⇐ ground >
{append(N2, K, M2), append(N0, [X], M0)}
<>
< N4, K ⇐ ground >
{ }
□
< N0 ⇐ [Y|N4], M0 ⇐ [Y|M4] >
< N0, X, M0 ⇐ ground >
{append(N4, K, M4), append(N2, K, M2), append(N0, [X], M0)}
append(N,[X],M) <> < N, X ⇐ ground > : {1st} [append(N,[X],M) < N, X, M ⇐ ground >]
append(N,K,M) <> < N, K ⇐ ground > : {1st} [append(N,K,M) < N, K, M ⇐ ground >]

Figure 5.2 Detection of Universal Termination

5.3 Detection of Existential Termination

In order to detect existential termination, it is enough to find an appropriate SLD-tree, which contains no infinite execution path, either by exhaustive search (with backtracking) or by heuristic search.

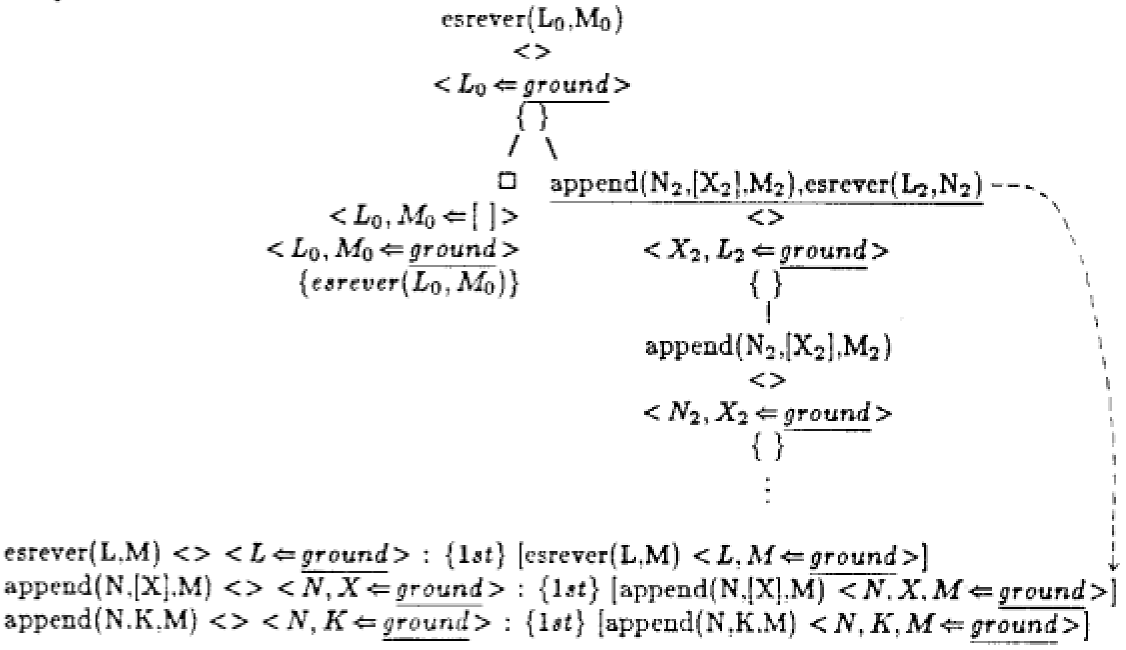

esrever(L0,M0)
<>
< L0 ⇐ ground >
{ }
□
append(N2,[X2],M2),esrever(L2,N2)
< L0, M0 ⇐ [ ] >
< L0, M0 ⇐ ground >
{esrever(L0, M0)}
<>
< X2, L2 ⇐ ground >
{ }
append(N2,[X2],M2)
<>
< N2, X2 ⇐ ground >
{ }
⋮
esrever(L,M) <> < L ⇐ ground > : {1st} [esrever(L,M) < L, M ⇐ ground >]
append(N,[X],M) <> < N, X ⇐ ground > : {1st} [append(N,[X],M) < N, X, M ⇐ ground >]
append(N,K,M) <> < N, K ⇐ ground > : {1st} [append(N,K,M) < N, K, M ⇐ ground >]

Figure 5.3 Detection of Existential Termination

Example 5.3 Let *esrever* the predicate as before. Then, termination detection proceeds similarly as in Figure 5.3 except selection of the second atom at the right child node of the root node instead of the leftmost atom. Since we have found one complete SLD search tree for termination detection without violating the termination condition, mode-abstracted atom $esrever(L, M) < L \Leftarrow \underline{ground} >$ is existentially terminating.

6. Discussion

As is mentioned in Section 1, a few techniques for detecting termination of logic programs have been developed. Francez, Grumberg, Katz and Pnueli [6] discussed formal methods to guarantee termination of Prolog programs. But, the class of goals they considered is restricted to those with no instantiation of variables during execution. Shapiro [15] discussed detection of diverging Prolog programs in the context of program debugging. His method checks the actual stack when the size of the stack overceeds a fixed depth during the execution of a given goal. Our approach is different from theirs in the following respects.

(a) Our approach considers the termination of the class of goals, whose variables may be instantiated during the execution (cf. [6]).

(b) Our approach checks not the actual stack, but an abstracted stack for a class of goals in a mode-abstracted negative clause (cf. [15]).

Our approach can be extended or generalized in the following four respects. First the class of measures considered can be generalized. For example, we can easily extend the class of symbolic complexity measures to those of the form $\epsilon_1|t_1| + \epsilon_2|t_2| + \cdots + \epsilon_n|t_n|$ ($\epsilon_i = +1, 0, -1$), or more general arithmetic expressions of symbolic complexities bounded downwards, if we can prepare corresponding termination lemmas of the form

termination-lemma(lemma-name).

$c \preceq m(s_1, s_2, \ldots, s_n) \prec m(t_1, t_2, \ldots, t_n)$:- $B_1, B_2, \ldots, B_l$.

end.

or tuples of symbolic complexities with the lexicographic ordering. It is even possible to employ a measure not based on symbolic complexities, e.g., any user-specified basic measure for each data structures like *length* for list structures to count just the top-level *cons*'s, or mappings to more novel well-founded sets. (The arguments actually contributing to the measure m are called the *measured arguments* of m.) But, the wider class of measures is allowed, the more time and the more termination lemmas are needed.

Example 6.1 Let *ackermann* be a predicate defined by

```
ackermann(0,N,N+1).
ackermann(M+1,0,X) :- ackermann(M,1,X).
ackermann(M+1,N+1,X) :- ackermann(M+1,N,Y), ackermann(M,Y,X).
```

In order to guarantee the termination of $ackermann(M, N, X)$ when M and N are ground, we need to use the tuple of the first and the second arguments with the lexicographic ordering.

Secondly, termination detection can be more powerful by utilizing the result of size analysis and transitivity of $\leq$ and $<$ (cf. [1] Chap.14).

Example 6.2 From the size analysis of *part*, we can derive

$0 \leq |LA| \leq |L|$:- part(L,X,LA,LB), $0 \leq |LB| \leq |L|$:- part(L,X,LA,LB).

Combined with the transitivity of $<$ and $\leq$, we can derive the following two termination lemmas, which are utilized to guarantee termination in other branches of Example 4.3.2.

termination-lemma(part-decreaseA).

```
    0 ≤ |LA| < |[X|L]| :- part(L,X,LA,LB).
end.
termination-lemma(part-decreaseB).
    0 ≤ |LB| < |[X|L]| :- part(L,X,LA,LB).
end.
```

Thirdly, we can store more information in the solution table. In the OLDT resolution for termination detection in Section 4, we recorded only mode-abstracted atoms as solutions in the solution lists, i.e., neglected the usual substitution part, because the number of solutions would be sometimes infinite if we did not neglect the usual substitution part. One might wonder, however, whether such neglection might loose too much information (of instantiation) so that termination cannot be detected in some cases. By employing depth-abstraction, not only more information can be saved but also the number of solutions is kept finite. A term t is a *level* 0 subterm of t itself. $t_1, t_2, \ldots, t_n$ are *level* $d+1$ subterms of t, when $f(t_1, t_2, \ldots, t_n)$ is a level d subterm of t. A term obtained from term t by replacing every level d non-variable subterm of t with a newly created distinct variable is called the *depth d abstraction of* t, and denoted by $[t]_d$. Let θ be a substitution of the form

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

Then the substitution

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

is called the *depth d abstraction of* θ, and denoted by $[\theta]_d$.

Example 6.3 Let t be a term $f(g(X,a),Y,b)$ and Z,U,V,W be fresh variables ([14] p.642). Then

$$[t]_d = \begin{cases} Z, & \text{when } d = 0; \\ f(U,Y,V), & \text{when } d = 1; \\ f(g(X,W),Y,b), & \text{when } d = 2; \\ t, & \text{when } d \geq 2. \end{cases}$$

Note that Y is not replaced with a new variable when $d = 1$, and neither is X when $d = 2$.

Then, we just need to modify the OLDT resolution for termination detection in two respects.

(1) The second condition for OLDT resolution is modified as follows: The node is a lookup node of Tr, and there is some solution $B\tau\nu$ in the associated solution list of the lookup node such that $\alpha_1\sigma$ and $B\tau$ are unifiable say by an m.g.u. θ, and α_1 is a variant of B, say by a renaming η.

(2) The statement for registration of solutions in OLDT resolution is modified as follows: add $A[\tau_{i+1}]_d\nu_{i+1}$ to the last of $A\sigma_{i+1}\mu_{i+1}$'s solution list if it is not in it.

Fourthly, it is possible to define the termination property assuming type information instead of assuming mode information. Then, we just need to use type inference [7],[8] instead of mode analysis.

Example 6.4 Suppose that a type predicate *list* is defined by

```
type.
    list([ ]).
    list([X|L]) :- list(L).
end.
```

Let *list* denote the set of all terms satisfying the definition above. If we would like to know that, when a goal of the form $reverse(L,M)$ is executed with its first argument instantiated to a list, it always terminates, we will consider a *type-abstracted atom*

reverse(L,M)$< L \Leftarrow \underline{list} >$.

7. Conclusions

We have shown a framework for detecting termination of logic programs. This method is an element of our system for analysis of Prolog programs Argus/A under development [7],[9],[10],[11].

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] Boyer,R. and J.S.Moore, "A Computational Logic," Academic Press, 1978.

[2] 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.

[3] 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.

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

[5] Dershowitz,N., "Orderings for Term Rewriting Systems," Theoretical Computer Science, 1982.

[6] Francez,N., O.Grumberg, S.Katz and A.Pnueli, "Proving Termination of Prolog Programs," in Logic of Programs (R.Parikh Ed.), Lecture Notes in Computer Science 193, pp.89–105, 1985.

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

[8] Kanamori,T. and K.Horiuchi, "Type Inference in Prolog and its Applications," Proc. of 9th International Joint Conference on Artificial Intelligence, pp.704–707, Los Angels, 1985.

[9] Kanamori,T. and T.Kawamura, "Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation," to appear, ICOT Technical Reports, 1987.

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

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

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

[13] Mellish,C.S., "Abstract Interpretation of Prolog Programs," Proc. of 3rd International Conference on Logic Programming, pp.463-474, London, 1986.

[14] Sato,T.and H.Tamaki, "Enumeration of Success Patterns in Logic Programming," Proc. of International Colloquium of Automata, Language and Programming, pp.640–652, 1984.

[15] Shapiro,E.Y., "Algorithmic Program Debugging," Ph.D Thesis, Dapartment of Computer Science, Yale University, 1982.

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