TR-485

Logic Program Analysis by Abstract Hybrid Interpretation

by
T. Kanamori, T. Kawamura, M. Maeji
& Horiuchi (Mitsubishi Electric)

July, 1989

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

Logic Program Analysis by Ab stract Hybrid Interpretation

Tadashi KANAMORI[†] Tadashi KAWAMURA[†]
Machi MAEJI[†] Kenji HORIUCHI[‡]

[†]Mitsubishi Electric Corporation
Central Research Laboratory
1-1 Tsukaguchi-Honmachi 8-Chome
Amagasaki, Hyogo, 661 Japan

[‡]ICOT Research Center
Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome, Minato-ku, Tokyo, 108 Japan

Abstract

This paper presents a unified framework for analyzing Prolog programs. The framework is based on a hybrid of the top-down and the bottom-up interpretations of Prolog programs. An execution-time property of Prolog goals can be analyzed when the goals are executed using an interpretation that is obtained by abstracting the hybrid interpretation according to the property. Due to its hybrid character, the execution neither dives into infinite loop nor wastes time for irrelevant goals. In addition, the behavior of the abstract hybrid interpreter is very close to the way human programmers usually analyze the property in their mind. Type inference, depth-abstracted term inference and mode inference are exemplified as the analysis of Prolog programs when different abstraction is employed.

Keywords : Program Analysis, Abstract Interpretation, Prolog, Hybrid Interpretation

Contents

1. Introduction

(1) What is Abstract Interpretation?

Automatic analysis of the execution-time properties of programs from their texts is useful not only for human programmers to find program bugs but also for meta-processing systems to manipulate programs effectively. For example, the information of data types sometimes plays an important role in the verification of Prolog programs [15]. The information of the form of Prolog goals appearing in their successfull execution can eliminate unnecessary backtracking from the Prolog execution [24]. The information of modes provides the Prolog compiler with a chance to generate optimized codes [21]. Besides these properties, the functionality and the termination properties are of special importance [9],[17],[18].

But, why can we analyze such execution-time properties of programs *without executing* them? The answer of the abstract interpretation approach is that we can analyze such properties *by approximately executing* them in greater or lesser degree. The framework of the abstract interpretation approach can be depicted schematically as below:

standard interpreter
standard domain
abstract interpreter
abstract domain

Figure 1.1 General Idea of the Abstract Interpretation Approach

The left half of the figure shows the standard domain of data to which the usual execution (the standard interpretation) is applied, while the right half shows the abstract domain for which some approximate execution (the abstract interpretation) is defined. The abstract interpretation approach executes programs in the abstract domain to extract useful information about the execution in the standard domain by utilizing the correspondence between the standard and the abstract domains.

For example, let the standard domain and the standard interpretation be the set of integers and the multiplication of integers, and let the abstract domain and the abstract interpretation be the set of signs $\{+, 0, -\}$ and the multiplication of signs as below:

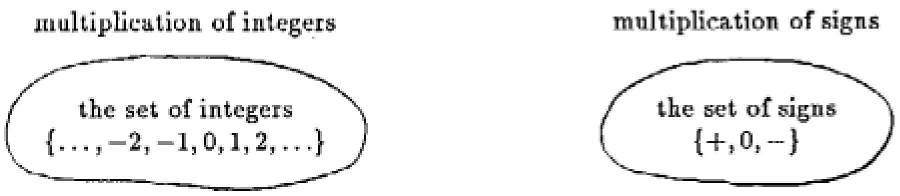


Figure 1.2 A Simple Example of Abstract Interpretation

Then, without exactly calculating the result $+221$, we can know that $(-13) \times (-17)$ is positive by abstracting the signs of the multiplicand and multiplier and by conducting the multiplication of signs $(-) \times (-) = (+)$.

Though the example above is too trivial, it gives us some flavor of the abstract interpretation approach. Then, how is the abstract interpretation approach applied to Prolog programs?

(2) How Do Human Programmers Analyze Programs?

Before considering the framework for the abstract interpretation of Prolog programs, let's first reflect on how we usually analyze Prolog programs in our mind? Suppose that we are asked: "When the execution of $reverse(L_0, N_0)$ succeeds, what data types of terms are variables L_0 and N_0 instantiated to?" Here, following the syntax of DEC-10 Prolog, "*reverse*" is defined as below:

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

Human programmers can easily answer the question after examining the program for a while, although they might not be precisely conscious of how they have reached the answer. Probably, they have done as follows:

1. If the first clause of "*reverse*" is used first when the execution of $reverse(L_0, N_0)$ succeeds, L_0 and N_0 are instantiated to [ ], hence L_0 and N_0 are lists.
2. If the second clause of "*reverse*" is used first, L_0 is instantiated to $[X_1|L_1]$, and N_0 to M_1, hence we need to answer the question: "When the execution of $reverse(L_1, N_1)$, $append(N_1, [X_1], M_1)$ succeeds, what data types of terms are $[X_1|L_1]$ and M_1 instantiated to?"
3. Now, we first need to answer the question: "When the execution of $reverse(L_1, N_1)$ succeeds, what data type of term are L_1 and N_1 instantiated to?" Because the question is identical to the original one, we would dive into infinite loop if we repeated the same process. However, we usually procceed as follows. As far as we know so far, L and N are lists when the execution of $reverse(L, N)$ succeeds. Let's temporarily assume it.
4. Then we need to answer the question: "When N_1 is a list and the execution of $append(N_1, [X_1], M_1)$ succeeds, what data types of terms are N_1, X_1 and M_1 instantiated to?" If the first clause of "*append*" is used first when the execution of $append(N_1, [X_1], M_1)$ succeeds, N_1 is instantiated to [ ], X_1 to X_2, and M_1 to $[X_2]$, hence N_1 is a list, X_1 may be any term, and M_1 is a list.
5. If the second clause of "*append*" is used first, N_1 is instantiated to $[Y_3|N_3]$, X_1 to X_3, and M_1 to $[Y_3|M_3]$, hence we need to answer the question: "When N_3 is a list and the execution of $append(N_3, [X_3], M_3)$ succeeds, what data types of terms are $[Y_3|N_3], X_3$ and $[Y_3|M_3]$ instantiated to?"
6. The analysis proceeds in the same way by following the execution in the domain of types. After several steps, we know that N_1 needed at step 4 is a list, X_1 may be any term, and M_1 is a list.
7. Hence, L_0 and N_0 needed at the beginning are lists. Because this result has not enlarged the data types of L_1 and N_1 temporarily assumed at step 3, we can conclude that L_0 and N_0 are lists when the execution of $reverse(L_0, N_0)$ succeeds.

Though we have not emphasized it, note that we needed to propagate the type information. For example, we needed to know at step 5 that, when N_1 is a list and N_1 is instantiated to $[Y_3|N_3]$, then N_3 is a list. Similarly, we needed to know at step 7 that, when L_1 is a list and L_0 is instantiated to $[Y_1|L_1]$, then L_0 is a list.

(3) What Interpreter is Appropriate for Prolog Abstract Interpretation?

Now, what framework is appropriate if we would analyze Prolog programs using the abstract interpretation approach? The figure below depicts the framework for type inference when the framework of Figure 1.1 is applied directly. Then, what interpretation should we employ for the abstract interpretation of Prolog programs?

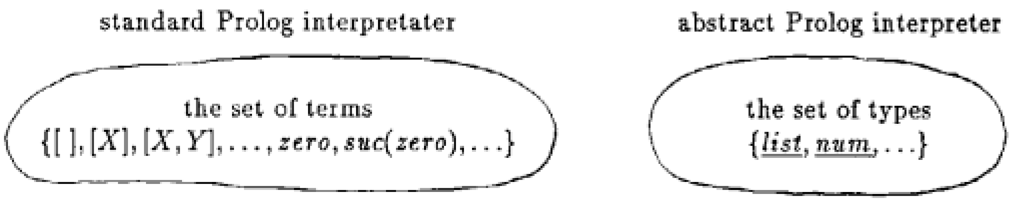

standard Prolog interpretater
the set of terms
$\{[\,], [X], [X,Y], \ldots, zero, suc(zero), \ldots\}$
abstract Prolog interpreter
the set of types
$\{\underline{list}, \underline{num}, \ldots\}$

Figure 1.3 Framework for Type Inference by Abstract Interpretation

One Prolog intepreter familiar to us is the top-down interpreter, which starts with a given top-level goal and repeats the resolution operation continually until an empty goal is obtained. However, if we had used the top-down interpretation to approximately execute the goal in the domain of types, we would have dived into infinite loop in the example just examined. For example, if we had not made the assumption at steps 3, we could not have proceeded any further. (In general, due to the abstraction, the top-down excution in abstract domains is more likely to dive into infinite loop than the usual one in the domain of terms. Sensitive readers might have hesitated to make the assumption at step 3 without rigorous justifications. Making the assumptions is, however, crucial to answer the question at the beginning.)

The other Prolog interpreter, which is also simple but not so familiar to us as the top-down interpreter, is the bottom-up interpreter, which starts with the set of all instances of unit clauses and repeats the generation of the head instances whose body instances are already generated. However, if we had employed the bottom-up interpretation, we would have generated many goals irrelevant to the top-level goal. For example, we have considered only necessary goals to know the data types of L_0 and M_0 when $reverse(L_0, M_0)$ succeeds, so that, say, a goal of the form $append(N, suc(K), suc(K))$ have not been considered in the example just examined.

Thus, the previous reflection on how we analyze Prolog programs in our mind has shown the different behavior from both the top-down interpreter and the bottom-up interpreter. This suggests us that it might be more appropriate to adopt another Prolog interpreter from the beginning.

This paper presents a unified framework for analyzing Prolog programs. The framework is based on a hybrid of the top-down and the bottom-up interpretations of Prolog programs. An execution-time property of Prolog goals can be analyzed when the goals are executed using an interpretation that is obtained by abstracting the hybrid interpretation according to the property. Due to its hybrid character, the execution neither dives into infinite loop nor wastes time for irrelevant goals. In addition, the behavior of the abstract hybrid interpreter is very close to the way human programmers usually analyze the property in their mind. Type inference, depth-abstracted term inference and mode inference are exemplified as the analysis of Prolog programs when different abstraction is employed.

The rest of this paper is organized as follow: First, Section 2 and 3 show the standard hybrid interpretation and the abstract hybrid interpretation for "type inference" emphasizing their correspondence. Next, Section 4 and 5 give their implementations emphasizing their correspondence again. Then, Section 6 and 7 show that "depth-abstracted term inference" and "mode inference" can be done using the same framework as "type inference" by modifying just 3 steps in its implementation. Last, Section 8 discusses the various analysis of logic programs by abstract interpretation according to their interpretation methods and their target properties, and suggests that "functionality detection" and "termination detection" can be done as well based on our framework by enriching the abstract domains appropriately.

2. Standard Hybrid Interpretation

In this section, we will first present an example of standard hybrid interpretation [27], then formalize the notions of the standard hybrid interpretation, and last point out the correspondence between the standard hybrid interpretation and the standard top-down interpretation.

2.1 An Example of Standard Hybrid Interpretation

Let us first see an example of the standard hybrid interpretation. Consider the following "graph reachability" program by Tamaki and Sato [27].

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

$$\begin{tikzcd} a \arrow[r, bend left] \arrow[d] & b \arrow[l, bend left] \arrow[d] \\ c & d \end{tikzcd}$$

The first clause of "*reach*" says that node Y is reachable from node X if node Z is reachable from X and there is an edge from Z to Y, while the second clause says that any node is reachable from itself. The unit clauses of "*edge*" gives the edges of the directed graph as right above. The program is a typical *left recursive program* so that the execution of a goal is likely to dive into infinite loop. For example, the execution of a top-level goal "$reach(a, Z_0)$" immediately calls "$reach(a, Z_1)$" recursively at the leftmost in the body of the first clause to repeat the execution of the goal of the same form.

The standard hybrid interpretation was devised by Tamaki and Sato [27] to avoid such infinite loop. It manipulates

- a tree,
- a table, and
- pointers connecting from some nodes of the tree into the table.

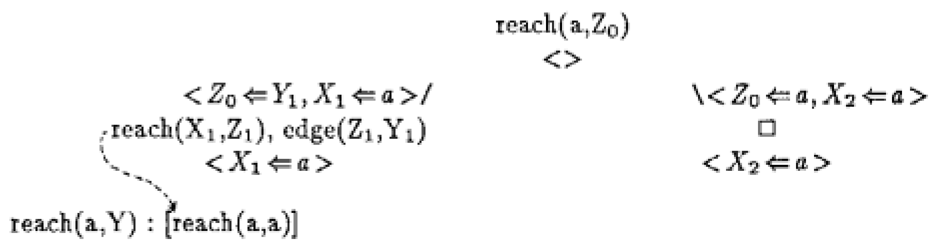
Roughly speaking, the tree corresponds to the top-down interpretation, the table corresponds to the bottom-up interpretation, and the pointers connecting them enables us to enjoy the advantages of the both interpretations. Let us see how the standard hybrid interpreter returns solutions to the top-level goal "$reach(a, Z_0)$."

reach(a,Z_0)
<>

reach(a,Z) : [ ]

Figure 2.1.1 Standard Hybrid Interpretation at Step 1

First, an initial tree consisting of only the root node labelled with a pair of goal $reach(a, Z_0)$ and an empty substitution <> is generated. (In general, each node of the tree is labelled with a pair of a goal and a substitution. Due to space limit, the goal and the substitution are arranged in two consecutive rows in the figure.) There is also generated an initial table containing only one pair of atom $reach(a, Z)$ and an empty list [ ]. (In general, the first element of each pair in the table is called a *key*, while the second element a *solution list* of the key. The key and solution list are delimited by ":" in the figure.) No pointer is generated yet.



reach(a,Z_0)
<>
$< Z_0 \Leftarrow Y_1, X_1 \Leftarrow a >$
reach(X_1,Z_1), edge(Z_1,Y_1)
$< X_1 \Leftarrow a >$
$< Z_0 \Leftarrow a, X_2 \Leftarrow a >$
□
$< X_2 \Leftarrow a >$
reach(a,Y) : [reach(a,a)]

Figure 2.1.2 Standard Hybrid Interpretation at Step 2

Secondly, the root node is expanded using the program to generate two child nodes in the same way as the top-down interpretation. The left child node generated using the first clause of "*reach*" is labelled with a pair of "$reach(X_1, Z_1), edge(Z_1, Y_1)$" and "$< X_1 \Leftarrow a >$". The edge to the left child node is labelled with the m.g.u. used in the resolution. Note that, $reach(a, Z_1)$, the leftmost atom under the substitution, is a variant of $reach(a, Z)$, a key in the table. Such a node is classified into a *lookup node*. (When the root node was generated, there was no such key in the table. Such a node is classified into a *solution node*.) A new pointer connecting from the lookup node to the head of the solution list of $reach(a, Z)$ is generated. (The pointer is depicted by the dotted line in the figure.) This means that the solutions in the solution list obtained by solving another atom are to be utilized for solving $reach(a, Z_1)$ instead of solving itself. The right child node generated using the second clause of "*reach*" is labelled with a pair of an empty goal □ and a substitution $< X_2 \Leftarrow a >$. The edge to the right child node is also labelled with the m.g.u. When this node is generated, goal $reach(a, Z_0)$ has been just solved instantiating Z_0 to "a" so that its solution $reach(a, a)$ is added to the solution list of $reach(a, Z)$. (In general, as for a solution node, the usual top-down interpretation is applied to the leftmost atom under the substituition.)

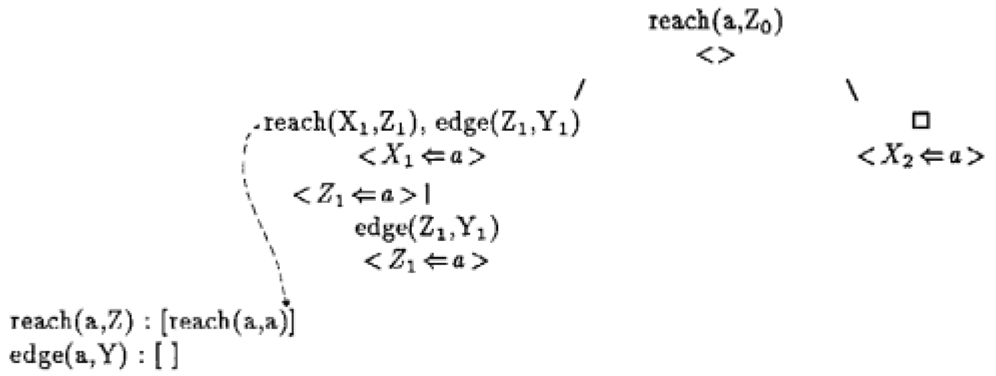


Figure 2.1.3 Standard Hybrid Interpretation at Step 3

Thirdly, the lookup node is expanded using the table to generate one child node. Because the solution in the list pointed from the lookup node is an instance of the leftmost atom under the substitution, i.e., $reach(a, a)$ is an instance of $reach(a, Z_1)$, the atom $reach(a, Z_1)$ is solved utilizing the solution to generate a child node labelled with a pair of "$edge(Z_1, Y_1)$" and $< Z_1 \Leftarrow a >$. The edge to the child node is labelled with the instantiation. The pointer from the lookup node is shifted to the last of the solution list. Because *edge*(a, Y_1), the leftmost atom under the substitution, is not a variant of any key in the table, the new node is a solution node so that a new pair of key *edge*(a, Y) and solution list [ ] is added to the table.

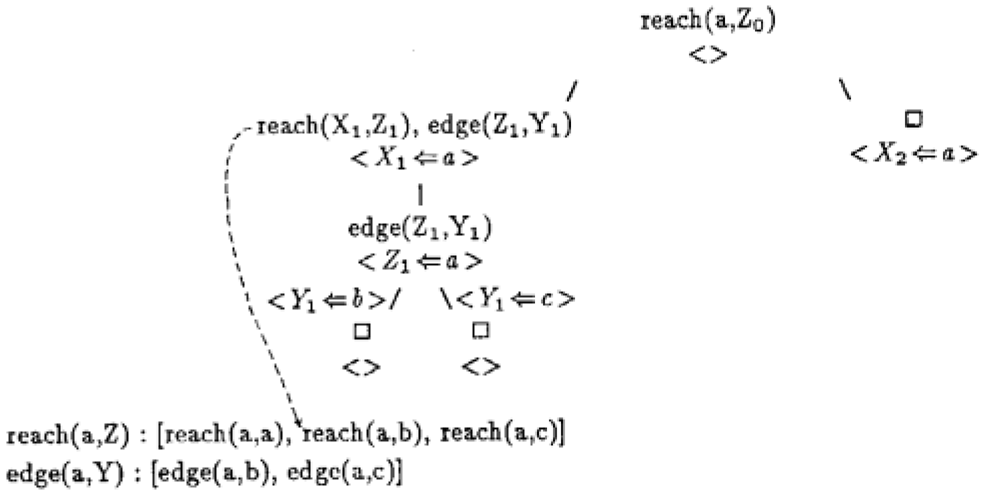


Figure 2.1.4 Standard Hybrid Interpretation at Step 4

Fourthly, the generated solution node is expanded further using the program to generate two child nodes labelled with a pair of □ and <>. These two nodes add two solutions *edge*(a, b) and *edge*(a, c) to the last of the solution list of *edge*(a, Y), and two solutions *reach*(a, b) and *reach*(a, c) to the last of the solution list of *reach*(a, Z).

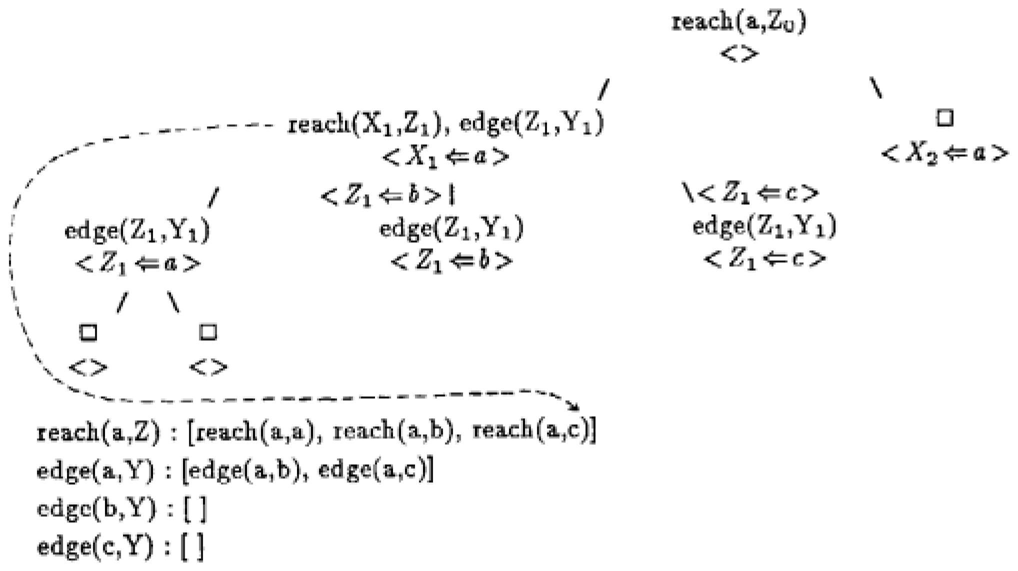
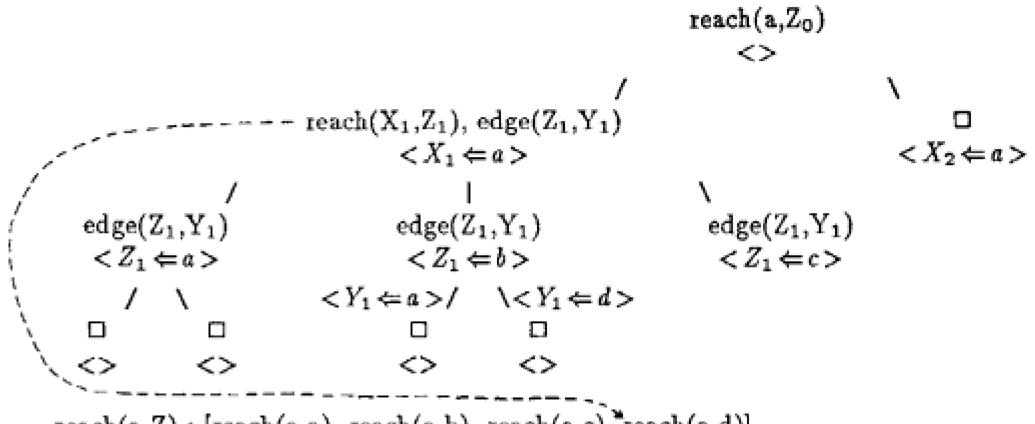


Figure 2.1.5 Standard Hybrid Interpretation at Step 5

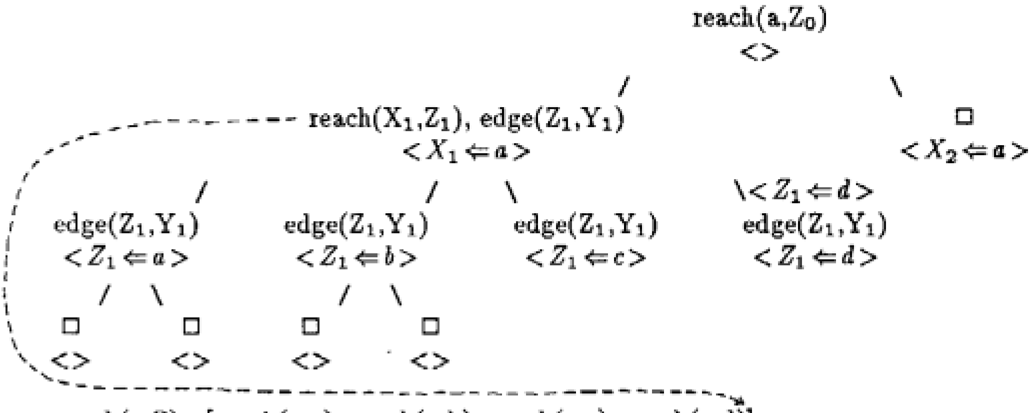
Fifthly, the lookup node is expanded using the solution table to generate two child nodes, since new solutions were added to the solution list of *reach*(a, Z) so that the list pointed from the lookup node is not empty, that is, there exist solutions not yet utilized.



reach(a,Z) : [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 2.1.6 Standard Hybrid Interpretation at Step 6

Sixthly, the left new solution node is expanded using the program to generate two child nodes. This time, goal $edge(b, Y_1)$ has been solved with solutions $edge(b, a)$ and $edge(b, d)$, and goal $reach(a, Z_0)$ with solutions $reach(a, a)$ and $reach(a, d)$, of which $reach(a, a)$ is already in the solution list of $reach(a, Z)$. Two new solutions $edge(b, a)$ and $edge(b, d)$ are added to the last of the solution list of $edge(a, Y)$, and one new solution $reach(a, d)$ to the last of the solution list of $reach(a, Z)$.



reach(a,Z) : [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 2.1.7 Standard Hybrid Interpretation at Step 7

Lastly, the lookup node is expanded once more using the table, since the list pointed from the lookup node is again not empty.

The standard hybrid interpreter stops here, because no solution node is expansible and the list pointed from the lookup node is empty.

2.2 Standard Hybrid Interpretation of Prolog Programs

Let us formalize the notions used in the example just examined.

(1) Term and Substitution

A *term* is defined as usual, and denoted by s, t, possibly with primes and subscripts. In particular, variables are denoted by X, Y, Z.

An *assignment of* term t *to* variable X is a pair (X, t), and hereafter represented by $X \Leftarrow t$. A *substitution* is a finite set of assignments such that there is no two assignements to the same variable, and hereafter represented by

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

where $X_1, X_2, \ldots, X_l$ are distinct variables, called the *domain variables* of the substitution. Substitutions are denoted by $\sigma, \tau, \theta, \eta$. The *restriction of* σ *to the set of variables* $\mathcal{V}$ is a substitution consisting of all the assignments in σ to the variables in $\mathcal{V}$.

The term assigned to variable X by substitution σ is denoted by $\sigma(X)$. We assume that a substitution assigns X itself to variable X when X is not in the domain variables of the substitution explicitly. Hence the empty substitution $<>$ assigns itself to every variable.

The *composed substitution* of σ and τ, denoted by $\sigma\tau$, is defined as usual.

(2) Atom and Goal

An atom is defined as usual, and denoted by A, B. Let A be an atom and σ be a substitution of the form

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

Then $A\sigma$ denotes the atom obtained by replacing each variable X_i in A with term t_i. (Note that, when $A\sigma$ is considered, assignments other than those in the restriction of σ to the variables in A do not matter.) An atom $A\tau$ is called an *instance* of an atom $A\sigma$ when there exists a substitution θ such that $A\tau$ is $A\sigma\theta$. An atom B is called a *variant* of an atom A when B is obtained from A by renaming the variables in A.

A goal is a finite sequence of atoms, and denoted by G, H. An empty goal, i.e., an empty sequence of atoms, is denoted by $\Box$. $G\sigma$ is defined in the same way as $A\sigma$.

(3) Unification of Atoms

Two atoms $A\sigma$ and $B\tau$ are said to be *unifiable* when there exists a common instance of $A\sigma$ and $B\tau$. A *most general unification* and a *most general unifier (m.g.u.)* of two atoms $A\sigma$ and $B\tau$ are defined as usual. Let θ be an m.g.u. of $A\sigma$ and $B\tau$. Then $B\tau\theta$ is a most general unification of $A\sigma$ and $B\tau$.

(4) Search Tree, Solution Table and Association

A *search tree* is a tree satisfying the following conditions:

- Each node is classified into either a *solution node* or a *lookup node*, and is labelled with a pair of a (possibly empty) goal and a substitution. (The distinction between solution nodes and lookup nodes is defined later.)
- Each edge is labelled with a substitution.

A *search tree of* $G\sigma$ is a search tree whose root node is labelled with (G, σ). A node in a search tree is called a *null node* when the goal part of the label is $\Box$. When a node in a search tree is labelled with ("$A_1, A_2, \ldots, A_n$", σ), the atom $A_1\sigma$ is called the *head atom of* the node.

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 variant key 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.

Let Tr be a search tree and Tb be a solution table. An *association* of Tr and Tb is a set of pointers connecting from each lookup node in Tr *into* some solution list in Tb such that the head atom of the lookup node and the key of the solution list are variants of each other. The tail of the solution list pointed from a lookup node is called the *associated solution list* of the lookup node.

(5) OLDT Structure

An *OLDT structure* of $G\sigma$ is a trio (Tr, Tb, As) satisfying the following conditions:

- Tr is a search tree of $G\sigma$.
- Tb is a solution table.
- As is an association of Tr and Tb.

(6) OLDT Resolution

A node in a search tree of OLDT structure (Tr, Tb, As) labelled with ("$A, A_2, \ldots, A_n$", σ) is said to be *OLDT resolvable* when it satisfies either of the following conditions:

- The node is a terminal solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that $A\sigma$ and B are unifiable, say by an m.g.u. θ. (We assume that, whenever each clause is used, a fresh variant of the clause is used.) The pair of the (possibly empty) goal "$B_1, B_2, \ldots, B_m, A_2, \ldots, A_n$" and the substitution $\sigma\theta$ (or the restriction of $\sigma\theta$ to the variables in "$B_1, B_2, \ldots, B_m, A_2, \ldots, A_n$") is called the *OLDT resolvent*.
- The node is a lookup node of Tr, and for some substitution θ (for the variables in $A\sigma$), there is a variant of $A\sigma\theta$ in the associated solution list of the lookup node. The pair of the (possibly empty) goal "$A_2, \ldots, A_n$" and the substitution $\sigma\theta$ (or possibly the restriction of $\sigma\theta$ to the variables in "$A_2, \ldots, A_n$") is called the *OLDT resolvent*.

In either cases, the substitution θ is called the *substitution of the OLDT resolution*.

(7) OLDT Subrefutation

An *OLDT subrefutation of* an atom and an *OLDT subrefutation of* a goal are paths in a search tree (not necessarily starting from the root node) which are simultaneously defined inductively as follows:

(a1) A path with length more than 0 starting from a solution node is an *OLDT subrefutation of* an atom $A\sigma$ with solution $A\tau$ when

- the initial node is labelled with a pair of the form ("A, G", σ), the initial edge with, say substitution θ, and the last node with a pair of the form ("G", σ'),
- the node next to the initial node is labelled with a pair of the form ("$A_1, A_2, \ldots, A_n$, G", $\sigma\theta$), and the path except the initial node and the initial edge is a subrefutation of $(A_1, A_2, \ldots, A_n)\theta$ with solution $(A_1, A_2, \ldots, A_n)\theta\tau'$ ($n \geq 0$), and
- τ is $\sigma\theta\tau'$.

(a2) A path with length 1 starting from a lookup node is an *OLDT subrefutation of* an atom $A\sigma$ with solution $A\tau$ when

- the initial node is labelled with a pair of the form ("A, G", σ), the initial edge with, say substitution θ, and the last node with a pair of the form ("G", σ''), and
- τ is $\sigma\theta$.

(b1) A path with length 0, i.e., a path consisting of only one node, is an *OLDT subrefutation of* $\Box\ \sigma$ with solution $\Box\ \sigma$.

(b2) A path with length more than 0 is an *OLDT subrefutation of* a goal $(A_1, A_2, \ldots, A_n)\sigma$ with solution $(A_1, A_2, \ldots, A_n)\tau$ $(n > 0)$ when

- the initial node is labelled with a pair of the form ("$A_1, A_2, \ldots, A_n, H$", σ), and the last node with a pair of the form ("H", σ'),
- the path is the concatination of a subrefutation of $A_1\sigma$ with solution $A_1\sigma\tau_1$, a subrefutation of $A_2\sigma\tau_1$ with solution $A_2\sigma\tau_1\tau_2, \ldots$, a subrefutation of $A_n\sigma\tau_1\tau_2\cdots\tau_{n-1}$ with solution $A_n\sigma\tau_1\tau_2\cdots\tau_{n-1}\tau_n$, and
- τ is $\sigma\tau_1\tau_2\cdots\tau_{n-1}\tau_n$.

In particular, a subrefutation of $A\sigma$ is called a *unit subrefutation of* $A\sigma$.

(8) Initial OLDT Structure and Extension of OLDT Structure

The *initial OLDT structure* of $G\sigma$ is the OLDT structure (Tr_0, Tb_0, As_0), where Tr_0 is a search tree consisting of only the root solution node labelled with (G, σ), Tb_0 is the solution table consisting of only one entry whose key is the head atom of the root node and whose solution list is an empty list [ ], and As_0 is an 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.

1. When v is a terminal solution node, let $C_1, C_2, \ldots, C_k$ $(k \geq 1)$ be all the clauses with which the node v is OLDT resolvable, and $(G_1, \sigma_1), (G_2, \sigma_2), \ldots, (G_k, \sigma_k)$ be the respective OLDT resolvents. Then add k child nodes of v labelled with $(G_1, \sigma_1), (G_2, \sigma_2), \ldots, (G_k, \sigma_k)$ to v. When v is a lookup node, let $A\sigma\theta_1, A\sigma\theta_2, \ldots, A\sigma\theta_k$ $(k \geq 1)$ be all (the variants of) the solutions with which the node v is OLDT resolvable, and $(G_1, \sigma_1), (G_2, \sigma_2), \ldots, (G_k, \sigma_k)$ be the respective OLDT resolvents. Then add k child nodes of v labelled with $(G_1, \sigma_1), (G_2, \sigma_2), \ldots, (G_k, \sigma_k)$ to v. In both cases, the edge from v to the node labelled with (G_i, σ_i) is labelled with θ_i, where θ_i is the substitution of the OLDT resolution. A new node is a lookup node when the head atom is a variant of some key in Tb, and is a solution node otherwise.
2. Replace the pointer from the OLDT resolved lookup node with the one connecting 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.
3. When a new node is a solution node, add a new entry whose key is the head atom of the new node and whose solution list is an empty list. For each unit subrefutation of atom $A\sigma$ (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 $A\sigma$ 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.

(9) OLDT Refutation

An *OLDT refutation* of $G\sigma$ in program P is a path in the search tree of some extension of the initial OLDT structure of $G\sigma$ from the root node to a null node. The *solution of an OLDT refutation* is defined in the same way as that of an OLDT subrefutation.

2.3 Correctness of the Standard Hybrid Interpretation

The readers have probably guessed that the standard hybrid interpretation just avoids repeating the same computation in the top-down interpretation by utilizing the solutions of the atoms of the same form so that, in the execution of any top-level goal, it calls the same atoms and returns the same solutions as the top-down interpretation. Let us first formalize the top-down interpretation.

(1) OLD Tree

An *OLD tree* is a tree such that each node is labelled with a pair of a (possibly empty) goal and a substitution, and each edge is labelled with a substitution. An *OLD tree of* $G\sigma$ is an OLD tree whose root node is labelled with (G, σ). A *null node* and a *head atom* are defined in the same way as before.

(2) OLD Resolution

A node of OLD tree Tr labelled with ("$A, A_2, \ldots, A_n$", σ) is said to be *OLD resolvable* when it satisfies the following condition:

- The node is a terminal node of Tr, and there is some definite clause "B :- $B_1, B_2, \ldots, B_m$" $(m \geq 0)$ in program P such that $A\sigma$ and B are unifiable, say by an m.g.u. θ. The pair of the (possibly empty) goal "$B_1, B_2, \ldots, B_m, A_2, \ldots, A_n$" and the restriction of $\sigma\theta$ to the variables in $B_1, B_2, \ldots, B_m, , A_2, \ldots, A_n$ is called the *OLD resolvent*.

The substitution θ is called the *substitution of the OLD resolution*.

A *subrefutation* and a *unit subrefutation* in an OLD tree are defined in the same way as those in a search tree. A path in a search tree starting from a node labelled with ("G, H", σ) is called a *partial subrefutation of* $G\sigma$ when it does not contain any subrefutation of $G\sigma$ as its prefix.

(3) OLD Refutation

The *initial OLD tree* of $G\sigma$ is the OLD tree Tr_0 consisting of only the root node labelled with (G, σ).

An *immediate extension* of OLD tree Tr in program P is the result of the following operations, when a node v of OLD tree Tr is OLD resolvable.

- Let $C_1, C_2, \ldots, C_k$ $(k \geq 1)$ be all the clauses with which the node v is OLD resolvable, and $(G_1, \sigma_1), (G_2, \sigma_2), \ldots, (G_k, \sigma_k)$ be the respective OLD resolvents. Then add k child nodes of v labelled with $(G_1, \sigma_1), (G_2, \sigma_2), \ldots, (G_k, \sigma_k)$ to v. The edge from v to the node labelled with (G_i, σ_i) is labelled with θ_i, where θ_i is the substitution of the OLD resolution with C_i.

An OLD tree Tr' is an *extension* of OLD tree Tr if Tr' is obtained from Tr through successive application of immediate extensions.

An *OLD refutation* of $G\sigma$ in program P is a path in some extension of the initial OLD tree of $G\sigma$ from the root node to a null node. The *solution of a refutation* is defined in the same way as that of a *subrefutation*.

(4) Correctness of the Standard Hybrid Interpretation

It is the basis of our abstract interpretation that any OLD extension is subsumed by an OLDT extension. (It is easy prove the reverse direction so that we will omit it.)

Theorem 2.3 (Correctness of the Standard Hybrid Interpretation)

(a) Let $G_0\sigma_0$ be a goal, T be an extension of the initial OLD tree of $G_0\sigma_0$, and $(T\tau, Tb, As)$ be an extension of the initial OLDT structure of $G_0\sigma_0$. If $A\sigma$ is the head atom of a node in T, then there exists an extension of $(T\tau, Tb, As)$ such that the search tree of the extension contains a node with head atom $A\sigma'$ which is a variant of $A\sigma$. (Correctness for Calling Patterns)

(b) Let T be an extension of an initial OLD tree, $(T\tau, Tb, As)$ be an extension of an initial OLDT structure, $A\sigma$ and $A\sigma'$ be atoms such that $A\sigma$ is a variant of $A\sigma'$. If T contains a unit subrefutation of $A\sigma$ with its solution $A\tau$, and $A\sigma'$ is the head atom of a node in $T\tau$, then there exists an extension of $(T\tau, Tb, As)$ such that the search tree of the extension contains a unit subrefutation of $A\sigma'$ with its solution $A\tau'$ which is a variant of $A\tau$. (Correctness for Exiting Patterns)

Proof. Although our standard hybrid interpretation is slightly different from the original OLDT resolution by Tamaki and Sato [27], these differences do not affect the proof of the following lemma:

(a) Let γ be an OLD partial subrefutation of $(A_1, A_2, \ldots, A_n)\sigma$ ending with a node whose head atom is $A\tau$, $\mathcal{S} = (T\tau, Tb, As)$ be an extension of an initial OLDT structure, and v be a node in $T\tau$ labelled with ("$A_1, A_2, \ldots, A_n, H$", σ') such that $(A_1, A_2, \ldots, A_n)\sigma$ is a variant of $(A_1, A_2, \ldots, A_n)\sigma'$. Then there exists an extension of $(T\tau, Tb, As)$ such that the search tree of the extension contains a node with head atom $A\tau'$ which is a variant of $A\tau$.

(b) Let γ be an OLD subrefutation of $(A_1, A_2, \ldots, A_n)\sigma$ with its solution $(A_1, A_2, \ldots, A_n)\tau$, $\mathcal{S} = (T\tau, Tb, As)$ be an extension of an initial OLDT structure, and v be a node in $T\tau$ labelled with ("$A_1, A_2, \ldots, A_n, H$", σ') such that $(A_1, A_2, \ldots, A_n)\sigma$ is a variant of $(A_1, A_2, \ldots, A_n)\sigma'$. Then there exists an extension of $(T\tau, Tb, As)$ such that the search tree of the extension contains an OLDT subrefutation of $(A_1, A_2, \ldots, A_n)\sigma'$ starting from v with its solution $(A_1, A_2, \ldots, A_n)\tau'$ which is a variant of $(A_1, A_2, \ldots, A_n)\tau$.

Then, the theorem is an immediate consequence of the lemma by letting H be $\Box$ and v be the root node for the part (a), and by letting n be 1 for the part (b). The proof of the lemma is almost the same as that of the lemma in Theorem 3.3. See Section 3.3.

Though all solutions were found under the depth-first from-left-to-right extension strategy in the example of Section 2.1, it is not always the case in general, that is, some solution might not be found forever. The reason is two-fold. One is that there might be generated infinitely many different solution nodes (hence possibly infinitely many lookup nodes). The other 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. (However, when this hybrid interpretation is applied to the abstract domain with finite elements, it always terminates under any strategy. See Section 3.3.)

3. Abastrct Hybrid Interpretation for Type Inference

The readers have probably noticed the similarity in the behavior between the type inference in Section 1 and the standard hybrid interpretation in Section 2.

3.1 An Example of Type Inference

Let us first re-examine the type inference process in Section 1 using the notions similar to those in Section 2. Recall the "*reverse*" program in Section 1.

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

And suppose that we are asked: "When the execution of $reverse(L_0, N_0)$ succeeds, what data types of terms are L_0 and N_0 instantiated to?"

First, an initial tree consisting of only the root node labelled with a pair of goal $reverse(L_0, N_0)$ and an empty substitution <> is generated. (In general, each node of the tree is labelled with a pair of a goal and a substitution of data types.) There is also generated an initial table containing only one pair of $reverse(L, N)$ <> and an empty list [ ]. (In general, the first element of the pair in the table is called a *key*, while the second element a *solution list* of the key.) No pointer is generated yet.

$$\begin{array}{c} \text{reverse}(L_0,N_0) \\ <> \end{array}$$

reverse(L,N)<> : [ ]

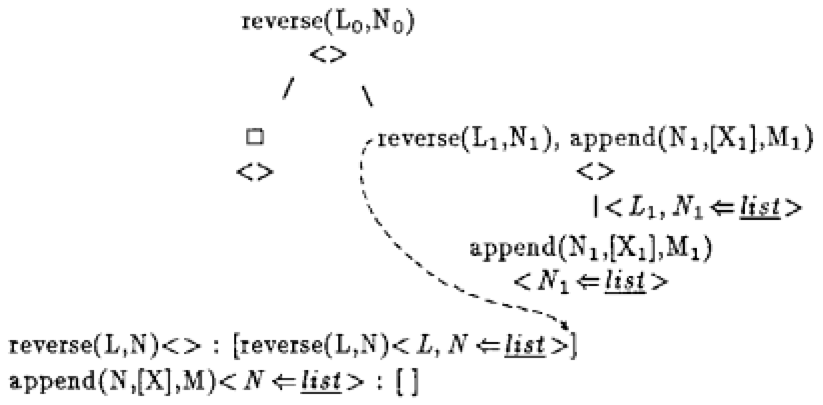
Figure 3.1.1 Type Inference at Step 1

Secondly, the root node is expanded using the program to generate two child nodes. The left child node generated using the first clause of "*reverse*" is labelled with a pair of an empty goal □ and an empty substitution <>. The edge to the left child node is labelled with the m.g.u. When this node is generated, goal $reverse(L_0, N_0)$ has been just solved instantiating L_0 and N_0 to lists so that its solution $reverse(L, N) < L, N \Leftarrow \underline{list} >$ is added to the solution list of $reverse(L, N)$ <>. The right child node generated using the second clause of "*reverse*" is labelled with a pair of "$reverse(L_1, N_1), append(N_1, [X_1], M_1)$" and <>. The edge to the right child node is also labelled with the m.g.u. Note that, $reverse(L_1, N_1)$ <>, the leftmost atom with the substitution, is a variant of $reverse(L, N)$ <>, a key in the table. Such a node is classified into a *lookup node*. (When the root node was generated, there was no such key in the table. Such a node is classified into a *solution node*.) The new pointer connecting from the lookup node to the head of the solution list of $reverse(L, N)$ <> is generated.

$$\begin{tikzcd}
 & \begin{array}{c}\text{reverse}(L_0,N_0)\\ <>\end{array} \arrow[dl, "{<L_0,N_0 \Leftarrow [\,]>}"'] \arrow[dr, "{<L_0 \Leftarrow [X_1|L_1], N_0 \Leftarrow M_1>}"] & \\
\begin{array}{c}\square\\ <>\end{array} & & \begin{array}{c}\text{reverse}(L_1,N_1),\ \text{append}(N_1,[X_1],M_1)\\ <>\end{array} \arrow[dll, dashed] \\
\text{reverse}(L,N)<> : [\text{reverse}(L,N)< L, N \Leftarrow \underline{list} >] & &
\end{tikzcd}$$

Figure 3.1.2 Type Inference at Step 2

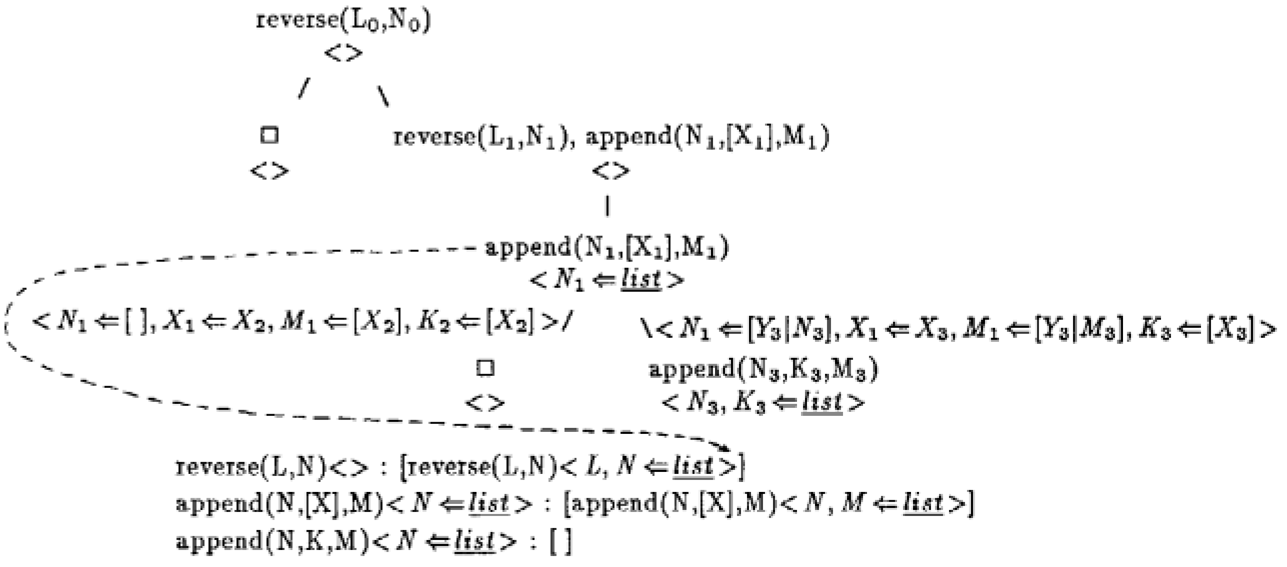
Thirdly, the lookup node is expanded using the table to generate one child node. Because the solution in the list pointed from the lookup node is more restricted w.r.t. data types than the leftmost atom with the type substitution, i.e., $reverse(L,N) < L, N \Leftarrow \underline{list} >$ is more restricted w.r.t. data types than $reverse(L,N)$ $<>$, the atom is solved utilizing the solution to generate a child node labelled with a pair of "$append(N_1,[X_1],M_1)$" and $< N_1 \Leftarrow \underline{list} >$. The edge to the child node is labelled with $< L_1, N_1 \Leftarrow \underline{list} >$. The pointer from the lookup node is shifted to the last of the solution list. Because $append(N_1,[X_1],M_1)$ $< N_1 \Leftarrow \underline{list} >$, the leftmost atom with the type substitution, is not a variant of any key in the table, the new node is a solution node so that a new pair of key $append(N,[X],M)$ $< N \Leftarrow \underline{list} >$ and solution list [ ] is added to the table.



reverse(L₀,N₀) <>
□ <>
reverse(L₁,N₁), append(N₁,[X₁],M₁) <>
<L₁, N₁ ⇐ list>
append(N₁,[X₁],M₁) <N₁ ⇐ list>
reverse(L,N)<> : [reverse(L,N)<L, N ⇐ list>]
append(N,[X],M)<N ⇐ list> : [ ]

Figure 3.1.3 Type Inference at Step 3

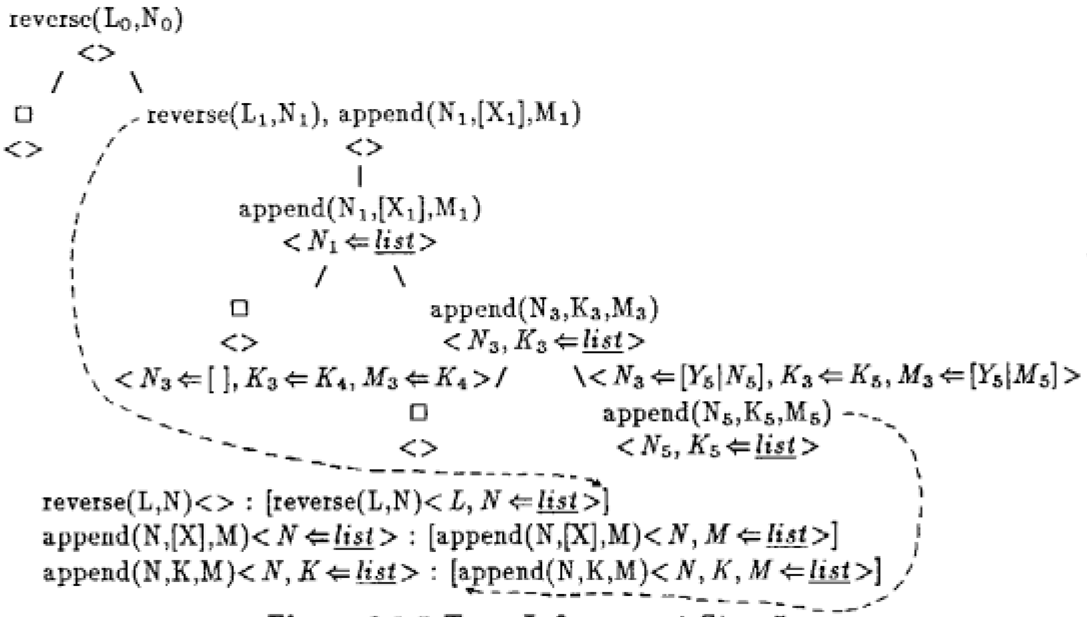
Fourthly, the new solution node is expanded further using the program to generate two child nodes labelled with pair $(\square, <>)$ and pair $(append(N_3, K_3, M_3), < N_3, K_3 \Leftarrow \underline{list} >$. The left node adds a solutions $append(N,[X],M) < N, M \Leftarrow \underline{list} >$ to the last of the solution list of $append(N,[X],M) < N \Leftarrow \underline{list} >$. (The solution $reverse(L,N) < L, N \Leftarrow \underline{list} >$ is already in the solution list of $reverse(L,N)$ $<>$.) The right node is a solution node so that a new entry is added to the solution table.



reverse(L₀,N₀) <>
□ <>
reverse(L₁,N₁), append(N₁,[X₁],M₁) <>
append(N₁,[X₁],M₁) <N₁ ⇐ list>
<N₁ ⇐ [ ], X₁ ⇐ X₂, M₁ ⇐ [X₂], K₂ ⇐ [X₂]>
<N₁ ⇐ [Y₃|N₃], X₁ ⇐ X₃, M₁ ⇐ [Y₃|M₃], K₃ ⇐ [X₃]>
□ <>
append(N₃,K₃,M₃) <N₃, K₃ ⇐ list>
reverse(L,N)<> : [reverse(L,N)<L, N ⇐ list>]
append(N,[X],M)<N ⇐ list> : [append(N,[X],M)<N, M ⇐ list>]
append(N,K,M)<N ⇐ list> : [ ]

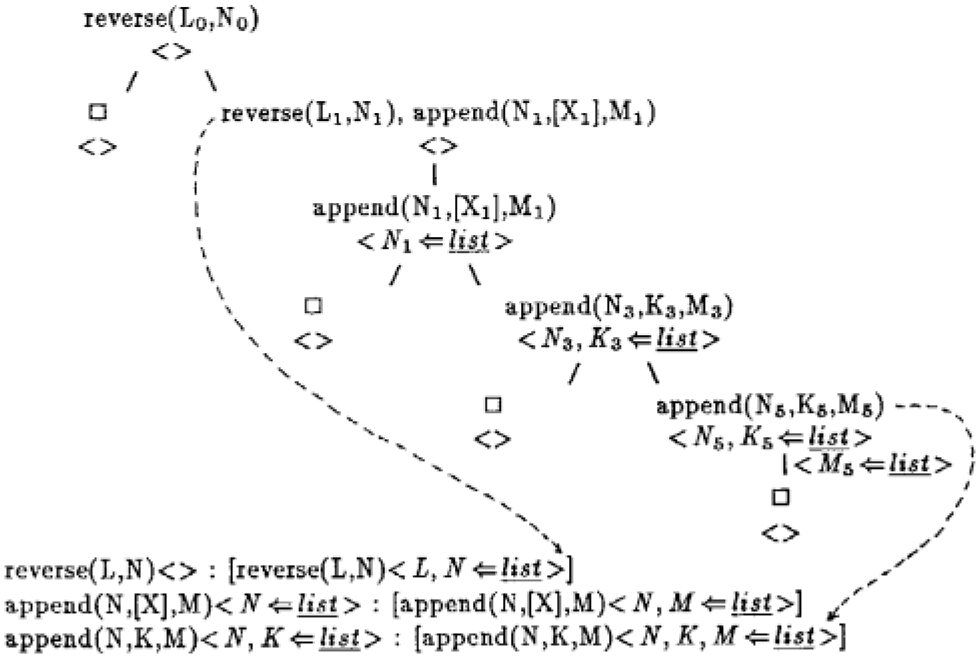
Figure 3.1.4 Type Inference at Step 4

Fifthly, the new solution node is expanded further using the program to generate two child nodes labelled with pair $(\square, <>)$ and pair $(append(N_5, K_5, M_5), < N_5, K_5 \Leftarrow \underline{list}>$. The left node adds a solutions $append(N, K, M) < N, K, M \Leftarrow \underline{list}>$ to the last of the solution list of $append(N, K, M) < N, K \Leftarrow \underline{list}>$. (The solutions $append(N, [X], M) < N, M \Leftarrow \underline{list}>$ and $reverse(L, N) < L, N \Leftarrow \underline{list}>$ are already in the solution lists.) The right node is a lookup node so that the new pointer connecting from the lookup node to the head of the solution list of $append(N, K, M) < N, K \Leftarrow \underline{list}>$ is generated.



reverse(L0,N0)
<>
□
<>
reverse(L1,N1), append(N1,[X1],M1)
<>
append(N1,[X1],M1)
<N1 ⇐ list>
□
<>
append(N3,K3,M3)
<N3, K3 ⇐ list>
<N3 ⇐ [ ], K3 ⇐ K4, M3 ⇐ K4>
<N3 ⇐ [Y5|N5], K3 ⇐ K5, M3 ⇐ [Y5|M5]>
□
<>
append(N5,K5,M5)
<N5, K5 ⇐ list>
reverse(L,N)<> : [reverse(L,N)< L, N ⇐ list>]
append(N,[X],M)< N ⇐ list> : [append(N,[X],M)< N, M ⇐ list>]
append(N,K,M)< N, K ⇐ list> : [append(N,K,M)< N, K, M ⇐ list>]

Figure 3.1.5 Type Inference at Step 5



reverse(L0,N0)
<>
□
<>
reverse(L1,N1), append(N1,[X1],M1)
<>
append(N1,[X1],M1)
<N1 ⇐ list>
□
<>
append(N3,K3,M3)
<N3, K3 ⇐ list>
□
<>
append(N5,K5,M5)
<N5, K5 ⇐ list>
<M5 ⇐ list>
□
<>
reverse(L,N)<> : [reverse(L,N)< L, N ⇐ list>]
append(N,[X],M)< N ⇐ list> : [append(N,[X],M)< N, M ⇐ list>]
append(N,K,M)< N, K ⇐ list> : [append(N,K,M)< N, K, M ⇐ list>]

Figure 3.1.6 Type Inference at Step 6

Lastly, the lookup node is expanded using the solution table to generate one child node labelled with pair $(\Box, < N_5, K_5, M_5 \Leftarrow \underline{list} >)$, since the list pointed from the lookup node is not empty, that is, there exist solutions not yet utilized. (Although $append(N_3, K_3, M_3) < N_3, K_3 \Leftarrow \underline{list} >$, $append(N_1, [X_1], M_1) < N_1 \Leftarrow \underline{list} >$ and $reverse(L_0, N_0) <>$ have been solved when this node is generated, their solutions are already in the solution lists.)

The type inference stops here, because no solution node is expansible and the lists pointed from the lookup nodes are all empty.

3.2 Type Inference for Prolog Programs

Let us formalize the notions used in the example just examined.

(1) Type and Type Substitution

A *type definition* is a set of definite clauses enclosed by **type** and **end** satisfying the following conditions:

- The head of each definite clause is an atom with its predicate p and with its argument either a constant b or a term of the form $c(X_1, X_2, \ldots, X_n)$, where the unary predicate p is called a *type predicate*, b a *bottom element* and c a *constructor* of the type predicate.
- The body of each definite clause consists of atoms whose predicate is a type predicate and whose argument is X_i in the head arguments.

The *type of* a type predicate p is the set of all terms t such that the execution of $p(t)$ succeeds without instantiating the variables in it, and denoted by $\underline{p}$.

Example 3.2.1 A type predicate "*list*" is defined by

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

Similarly, a type predicate "*num*" is defined by

```
    type.
        num(zero).
        num(suc(N)) :- num(N).
    end.
```

Then $\underline{list}$ is $\{[\ ], [X], [X, Y], \ldots\}$, and $\underline{num}$ is $\{zero, suc(zero), suc(suc(zero)), \ldots\}$. Note that terms in each type are not necessarily ground, since the execution of $p(t)$ sometimes succeeds without instantiating the variables in t. For example, we include $[X]$ in $\underline{list}$, since the execution of $list([X])$ succeeds without instantiating the variable X.

Suppose that there exist k type predicates $p_1, p_2, \ldots, p_k$ in program P such that bottom elements and constructors of each p_i are disjoint, hence $\underline{p_1}, \underline{p_2}, \ldots, \underline{p_k}$ are disjoint. (To make our explanation simple, we will consider the simplest type structure here so that more complicated type structure, e.g., types with non-empty intersections or polymorphic types [13], are not discussed.) A *type* of program P is one of the following $k + 2$ sets of terms.

$\underline{any}$: the set of all terms,
$\underline{p_1}$: the set of all terms satisfying the definition of type predicate p_1,
$\underline{p_2}$: the set of all terms satisfying the definition of type predicate p_2,
$\vdots$
$\underline{p_k}$: the set of all terms satisfying the definition of type predicate p_k,
$\underline{\emptyset}$: the empty set.

The *instantiation ordering of types* is the ordering $\prec$ depicted left below, while the *set-inclusion ordering of types* is the ordering $\subset$ depicted right below:

$$\begin{tikzcd}
& \emptyset \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\underline{p_1} & \underline{p_2} \cdots & \underline{p_k} \\
& \underline{any} \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& \underline{any} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\underline{p_1} & \underline{p_2} \cdots & \underline{p_k} \\
& \emptyset \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] &
\end{tikzcd}$$

In general, a set of terms T_1 is *smaller than or equal to* a set of terms T_2 *w.r.t. the instantiation ordering*, and denoted by $T_1 \preceq T_2$, when

- for any unifiable terms t_1 in T_1 and t_2 in T_2, their most general unification is in T_2, and
- for any term t_2 in T_2, there exists a term t_1 in T_1 such that t_2 is an instance of t_1.

T_1 is *smaller than* T_2 *w.r.t. the instantiation ordering*, when $T_1 \preceq T_2$ but $T_2 \not\preceq T_1$. As the execution of a goal proceeds, the arguments of the goal ascend this instantiation ordering. (Hence, $\emptyset$ denotes over-instantiation, or failure.) Note that the instantiation ordering of types is just the reverse of the set inclusion ordering, hence if $\underline{t_1} \preceq \underline{t_2}$, then $\underline{t_1} \supseteq \underline{t_2}$. (This is not always the case for some analysis by abstract hybrid interpretation. See Section 7 (5).)

An *assignment of* type $\underline{t}$ *to* variable X is a pair $(X, \underline{t})$, and hereafter represented by $X \Leftarrow \underline{t}$. A *type substitution* is a finite set of type assignments such that there is no two type assignements to the same variable, and hereafter represented by

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

where $X_1, X_2, \ldots, X_l$ are distinct variables, called the *domain variables* of the type substitution. Type substitutions are denoted by μ, ν, λ in this section. The *restriction of* μ *to the set of variables* $\mathcal{V}$ is the type substitution consisting of all the type assignments in μ to the variables in $\mathcal{V}$.

The type assigned to variable X by type substitution μ is denoted by $\mu(X)$. We assume that a type substitution assigns $\underline{any}$, the minimum element w.r.t. the instantiation ordering, to variable X when X is not in the domain variables of the type substitution explicitly. Hence the empty type substitution $<>$ assigns $\underline{any}$ to every variable.

The *joined type substitution* of μ and ν, denoted by $\mu \vee \nu$, is the type substitution such that the domin variables is the union of those of μ and ν, and $\mu \vee \nu(X)$ is the least upper bound of $\mu(X)$ and $\nu(X)$ w.r.t. the instantiation ordering for each domain variable X. (In particular, if $\nu(X)$ is greater than or equal to $\mu(X)$ w.r.t. the instantiation ordering for each domain variable X of ν, then $\mu \vee \mu$ is the type substitution obtained from μ by just replacing $\mu(X)$ with $\nu(X)$ for each domain variable X of ν.)

(2) Type-abstracted Atom and Type-abstracted Goal

Let A be an atom and μ be a type substitution of the form

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

Then $A\mu$ (or pair (A, μ)) is called a *type-abstracted atom*, and denotes the set of all atoms obtained by replacing each variable X_i in A with a term in $\underline{t_i}$. (Hereafter, we will consider only the restriction of μ to the variables in A when $A\mu$ is considered.) A type-abstracted atom $A\nu$ is called an *instance* of a type-abstracted atom $A\mu$ when there exists a type substitution λ such that $A\nu$ is $A(\mu \vee \lambda)$. A type-abstracted atom $B\nu$ is called a *variant* of a type-abstracted atom $A\mu$ when B is a variant of A and ν is obtained from μ by renaming the variables in the domain of μ accordingly.

Similarly, $G\mu$ (or pair (G,μ)) is called a *type-abstracted goal*, and denotes the set of all goals obtained by replacing each X_i in G with a term in $\underline{t_i}$.

(3) Unification of Type-Abstracted Atoms

Two type-abstracted atoms $A\mu$ and $B\nu$ are said to be *unifiable* when $A\mu \cap B\nu \neq \emptyset$. Let A be an atom, $X_1, X_2, \ldots, X_k$ all the variables in A, μ a type substitution

$< X_1 \Leftarrow \underline{t_1}, X_2 \Leftarrow \underline{t_2}, \ldots, X_k \Leftarrow \underline{t_k}, \ldots >$,

B an atom, $Y_1, Y_2, \ldots, Y_l$ all the variables in B, and ν a type substitution

$< Y_1 \Leftarrow \underline{s_1}, Y_2 \Leftarrow \underline{s_2}, \ldots, Y_l \Leftarrow \underline{s_l}, \ldots >$.

Then, how can we know whether $A\mu$ and $B\nu$ are unifiable, that is, whether there exists a unification of $A\sigma$ in $A\mu$ and $B\tau$ in $B\nu$? And, if there exists such a unification, what types of terms are expected to be assigned to $Y_1, Y_2, \ldots, Y_l$ by the unifier?

When two type-abstracted atoms $A\mu$ and $B\nu$ are unifiable, two atoms A and B must be unifiable in the usual sense. Hence the unifiability of A and B can be temporarily used as an easy overestimation of the unifiability of $A\mu$ and $B\nu$. (This estimation might be inexact, e.g., the unifiability of $p(X) < X \Leftarrow \underline{list} >$ and $p(suc(Y)) < Y \Leftarrow \underline{list} >$.)

When A and B are unifiable, let η be an m.g.u. of A and B of the form

$< X_1 \Leftarrow t_1, X_2 \Leftarrow t_2, \ldots, X_k \Leftarrow t_k, Y_1 \Leftarrow s_1, Y_2 \Leftarrow s_2, \ldots, Y_l \Leftarrow s_l >$.

The type information of μ is propagated to the variables in B through η. Let's divide the type propagation through η into two phases, *inwards type propagation* and *outwards type propagation*.

When a term t containing an occurrence of term s is instantiated to a term in $\underline{t}$, a type containing all instances of *the occurrence* of term s is called an *inwards type propagation of* $\underline{t}$ *to* s, denoted by $s/ < t \Leftarrow \underline{t} >$. (Exactly speaking, some notation denoting the *occurrence* of s should be used instead of term s itself.) It is computed as below:

$$s/ < t \Leftarrow \underline{t} > = \begin{cases} \underline{t}, & \text{when } s \text{ is } t; \\ \underline{any}, & \text{when } \underline{t} \text{ is } \underline{any}; \\ s/ < t_i \Leftarrow \underline{t_i} >, & \text{when } \underline{t} \text{ is a type } \underline{p}, \\ & t \text{ is of the form } c(t_1, t_2, \ldots, t_n), \\ & c \text{ is a constructor of the data type } p, \\ & \text{the occurrence of } s \text{ is in } t_i, \text{ and} \\ & \underline{t_i} \text{ is the type assigned to the } i\text{-th argument } t_i; \\ \emptyset, & \text{otherwise.} \end{cases}$$

Example 3.2.2 Let t be $[X|L]$ and $\underline{t}$ be $\underline{list}$. Then

$X/ < [X|L] \Leftarrow \underline{list} > = \underline{any}, \quad L/ < [X|L] \Leftarrow \underline{list} > = \underline{list}.$

Let t be $[X|L]$ and $\underline{t}$ be $\underline{num}$. Then

$X/ < [X|L] \Leftarrow \underline{num} > = \emptyset, \quad L/ < [X|L] \Leftarrow \underline{num} > = \emptyset.$

When each variable Z in term s is instantiated to a term in $\lambda(Z)$, a type containing all instances of s is called an *outwards type propagation of* λ *to* s, denoted by s/λ. It is computed as below:

$$s/\lambda = \begin{cases} \emptyset, & \lambda(Z) = \emptyset \text{ for some } Z \text{ in } s; \\ \lambda(s), & \text{when } s \text{ is a variable;} \\ \underline{p}, & \text{when } s \text{ is a bottom element } b \text{ of a type } p \text{ or} \\ & \text{when } s \text{ is of the form } c(s_1, s_2, \ldots, s_n), \\ & c \text{ is a constructor of a data type } p \text{ and} \\ & s_1/\lambda, s_2/\lambda, \ldots, s_n/\lambda \text{ satisfy the type conditions;} \\ \underline{any}, & \text{otherwise.} \end{cases}$$

Example 3.2.3 Let s be $[X|L]$ and λ be $<X \Leftarrow \underline{any}, L \Leftarrow \underline{list}>$. Then
$s/\lambda = \underline{list}$.
Let s be $[X|L]$ and λ be $<X \Leftarrow \underline{any}, L \Leftarrow \underline{any}>$. Then
$s/\lambda = \underline{any}$.

Let A, $X_1, X_2, \ldots, X_k$, μ, B, $Y_1, Y_2, \ldots, Y_l$ and ν be as before. Then, we can overestimate the unification of $A\mu$ and $B\nu$ as follows:

1. First, we can check the unifiability of $A\mu$ and $B\nu$ by the unifiability of A and B. If A and B are not unifiable, $A\mu$ and $B\nu$ are not unifiable. Otherwise, let η be an m.g.u. of A and B of the form
$<X_1 \Leftarrow t_1, X_2 \Leftarrow t_2, \ldots, X_k \Leftarrow t_k, Y_1 \Leftarrow s_1, Y_2 \Leftarrow s_2, \ldots, Y_l \Leftarrow s_l>$.
2. Next, for each occurrence of a bottom element b in $t_1, t_2, \ldots, t_k$, we can compute the type assigned to the occurrence using the inwards type propagation of μ. Similarly, for each occurrence of variable Z in $t_1, t_2, \ldots, t_k$, we can compute a type containing all instances of the occurrence using the inwards type propagation. By taking their least upper bound w.r.t. the instantiation ordering for all the occurrences of Z in t, we can compute a type containing all instances of Z. If
   - the type assigned to some occurrence of the bottom element is not the type of the bottom element, or
   - the type assigned to some variable is $\emptyset$,

   $A\mu$ and $B\nu$ are not unifiable. Otherwise, we can compute the type substitution λ for all the variables in $t_1, t_2, \ldots, t_n$ by collecting these type assignments for the variables.
3. Then, we can overestimate the type $\underline{s'_j}$ assigned to s_j using the outwards type propagation of λ, hence, we can obtain a type substitution ν'
$< Y_1 \Leftarrow \underline{s'_1}, Y_2 \Leftarrow \underline{s'_2}, \ldots, Y_l \Leftarrow \underline{s'_l} >$
by collecting the types for all variables $Y_1, Y_2, \ldots, Y_l$ in B.
4. Last, $A\mu \cap B\nu$ is overestimated by $B(\nu \vee \nu')$.

The type substitution $\nu \vee \nu'$ is called the *propagated type substitution from μ to ν through η*, and denoted by "$\mu \overset{\eta}{\rightarrow} \nu$" or "$\nu \overset{\eta}{\leftarrow} \mu$."

Note that, even if σ assigns two terms containing a common variable to two different variables in A for some $A\sigma$ in $A\mu$, $B(\mu \overset{\eta}{\rightarrow} \nu)$ is a superset of $A\mu \cap B\nu$. For example, let $A\mu$ be $p(X_1, X_2) <>$ and $B\nu$ be $p(Y_1, Y_2) < Y_1 \Leftarrow \underline{list}>$. (Hence η is, e.g., $< X_1 \Leftarrow Y_1, X_2 \Leftarrow Y_2 >$.) When $p(Z, Z)$ in $A\mu$ and $p([\,], W)$ in $B\nu$ are unified, their unification $p([\,],[\,])$ is in $p(Y_1, Y_2) < Y_1, Y_2 \Leftarrow \underline{list}>$, which is still included in $B(\mu \overset{\eta}{\rightarrow} \nu)$, i.e., $p(Y_1, Y_2) < Y_1 \Leftarrow \underline{list}>$. Though the fact that Y_2 has been instantiated to $[\,]$, i.e., the type assigned to Y_2 has ascended w.r.t. the instantiation ordering, is not precisely reflected in the computation of "$\mu \overset{\eta}{\rightarrow} \nu$," the final estimation $B(\mu \overset{\eta}{\rightarrow} \nu)$ is a superset of $A\mu \cap B\nu$, since $\underline{t_1} \supseteq \underline{t_2}$ if $\underline{t_1} \preceq \underline{t_2}$.

(4) Search Tree, Solution Table and Association for Type Inference

A *search tree* is a tree satisfying the following conditions:

- Each node is classified into either a *solution node* or a *lookup node*, and is labelled with a pair of a (possibly empty) goal and a type substitution. (The distinction between solution nodes and lookup nodes is defined later.)
- Each edge from a solution node is labelled with a substitution, and each edge from a lookup node is labelled with a type substitution.

A *search tree of* (G, μ) is a search tree whose root node is labelled with (G, μ). A node is called a *null node* when the goal part of the label is $\Box$. When a node is labelled with ("$A_1, A_2, \ldots, A_n$", μ), the type-abstracted atom $A_1\mu$ is called the *head type-abstracted atom of* the node.

A *solution table* is a set of entries. Each entry is a pair of the *key* and the *solution list*. The key is a type-abstracted atom such that there is no other variant key in the solution table. The solution list is a list of type-abstracted atoms, called *solutions*, such that each solution in it is an instance of the corresponding key.

Let Tr be a search tree and Tb be a solution table. An *association* of Tr and Tb is a set of pointers connecting from each lookup node in Tr *into* some solution list in Tb such that the head atom of the label of the lookup node and the key of the solution list are variants of each other. The tail of the solution list pointed from a lookup node is called the *associated solution list* of the lookup node.

(5) OLDT Structure for Type Inference

An *OLDT structure* of $G\mu$ is a trio (Tr, Tb, As) satisfying the following conditions:

- Tr is a search tree of $G\mu$.
- Tb is a solution table.
- As is an association of Tr and Tb.

(6) OLDT Resolution for Type Inference

A node in a search tree of OLDT structure (Tr, Tb, As) labelled with ("$A, A_2, \ldots, A_n$", μ) is said to be *OLDT resolvable* when $\mu(X) \neq \emptyset$ for any variable X in $A, A_2, \ldots, A_n$, and $A\mu$ satisfies either of the following conditions:

- The node is a terminal solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that A and B are unifiable, say by an m.g.u. η. (We assume that, whenever each clause is used, a fresh variant of the clause is used.) The pair of the (possibly empty) goal "$B_1, B_2, \ldots, B_m, A_2, \ldots, A_n$" and the type substitution "$\mu \vee (\mu \xrightarrow{\eta} <>)$" (or possibly the restriction of "$\mu \vee (\mu \xrightarrow{\eta} <>)$" to the variables in "$B_1, B_2, \ldots, B_m, A_2, \ldots, A_n$") is called the *OLDT resolvent*. The substitution η is called the *substitution of the OLDT resolution*.
- The node is a lookup node of Tr, and for some type substitution λ (for the variables in A), there is some variant of $A(\mu \vee \lambda)$ in the associated solution list of the lookup node. The pair of the (possibly empty) goal "$A_2, \ldots, A_n$" and the type substitution "$\mu \vee \lambda$" (or possibly the restriction of "$\mu \vee \lambda$" to the variables in "$A_2, \ldots, A_n$") is called the *OLDT resolvent*. The type substitution λ is called the *type substitution of the OLDT resolution*.

(7) OLDT Subrefutation for Type Inference

An *OLDT subrefutation of* a type-abstracted atom and an *OLDT subrefutation of* a type-abstracted goal are paths in a search tree (not necessarily starting from the root node) which are simultaneously defined inductively as follows:

(a1) A path with length more than 0 starting from a solution node is an *OLDT subrefutation of* a type-abstracted atom $A\mu$ with solution $A\nu$ when

- the initial node is labelled with a pair of the form ("A, G", μ), the initial edge with, say substitution η, and the last node with a pair of the form ("G", μ'),
- the node next to the initial node is labelled with a pair of the form ("$A_1, A_2, \ldots, A_n, G$", "$\mu \vee (\mu \xrightarrow{\eta} <>)$"), and the path except the initial node and the initial edge is a subrefutation of $(A_1, A_2, \ldots, A_n)(\mu \xrightarrow{\eta} <>)$ with solution $(A_1, A_2, \ldots, A_n)\nu'$ ($n \geq 0$), and
- ν is "$\mu \xleftarrow{\eta} \nu'$."

(a2) A path with length 1 starting from a lookup node is an *OLDT subrefutation of* a type-abstracted atom $A\mu$ with solution $A\nu$ when

- the initial node is labelled with a pair of the form ("A, G", μ), the initial edge with, say type substitution λ, and the last node with a pair of the form ("G", μ'), and
- ν is $\mu \vee \lambda$.

(b1) A path with length 0, i.e., a path consisting of only one node, is an *OLDT subrefutation of* $\Box\ \mu$ with solution $\Box\ \mu$.

(b2) A path with length more than 0 is an *OLDT subrefutation of* a type-abstracted goal $(A_1, A_2, \ldots, A_n)\mu$ with solution $(A_1, A_2, \ldots, A_n)\nu$ ($n > 0$) when

- the initial node is labelled with a pair of the form ("$A_1, A_2, \ldots, A_n, H$", μ), and the last node with a pair of the form ("H", μ'),
- the path is the concatination of a subrefutation of $A_1\mu$ with solution $A_1(\mu \vee \nu_1)$, a subrefutation of $A_2(\mu \vee \nu_1)$ with solution $A_2(\mu \vee \nu_1 \vee \nu_2)$, ..., a subrefutation of $A_n(\mu \vee \nu_1 \vee \nu_2 \vee \cdots \vee \nu_{n-1})$ with solution $A_n(\mu \vee \nu_1 \vee \nu_2 \vee \cdots \vee \nu_{n-1} \vee \nu_n)$, and
- ν is $\mu \vee \nu_1 \vee \nu_2 \vee \cdots \vee \nu_{n-1} \vee \nu_n$.

In particular, a subrefutation of $A\mu$ is called a *unit subrefutation of* $A\mu$.

(8) Initial OLDT Structure and Extension of OLDT Structure for Type Inference

The *initial OLDT structure* of (G, μ) is the OLDT structure (Tr_0, Tb_0, As_0), where Tr_0 is a search tree consisting of only the root solution node labelled with (G, μ), Tb_0 is the solution table consisting of only one entry whose key is the head type-abstracted atom of $G\mu$ and whose solution list is an empty list [ ], and As_0 is an 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.

1. When v is a terminal solution node, let $C_1, C_2, \ldots, C_k$ ($k \geq 1$) be all the clauses with which the node v is OLDT resolvable, and $(G_1, \mu_1), (G_2, \mu_2), \ldots, (G_k, \mu_k)$ be the respective OLDT resolvents. Then add k child nodes of v labelled with $(G_1, \mu_1), (G_2, \mu_2), \ldots, (G_k, \mu_k)$ to v. The edge from v to the node labelled with (G_i, μ_i) is labelled with η_i, where η_i is the substitution of the OLDT resolution. When v is a lookup node, let $A\nu_1, A\nu_2, \ldots, A\nu_k$ ($k \geq 1$) be all (the variants of) the solutions with which the node v is OLDT resolvable, and $(G_1, \mu_1), (G_2, \mu_2), \ldots, (G_k, \mu_k)$ be the respective OLDT resolvents. Then add k child nodes of v labelled with $(G_1, \mu_1), (G_2, \mu_2), \ldots, (G_k, \mu_k)$ to v. The edge from v to the node labelled with (G_i, μ_i) is labelled with λ_i, where λ_i is the type substitution of the OLDT resolution. A new node is a lookup node when the head type-abstracted atom is a variant of some key in Tb, and is a solution node otherwise.

2. 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.
3. When a new node is a solution node, add a new entry whose key is the head type-abstracted atom of the new node and whose solution list is the empty list. For each unit subrefutation of atom $A\mu$ (if any) starting from a solution node and ending with some of the new nodes labelled with (G_i, μ_i), add its solution $A\nu$ to the last of the solution list of $A\mu$ in Tb if $A\nu$ is not in the solution list, and update the type substitution part μ_i of the new node to $\mu_i \vee \nu$.

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.

(9) OLDT Refutation for Type Inference

An *OLDT refutation* of $G\mu$ in program P is a path in the search tree of some extension of the initial OLDT structure of $G\mu$ from the root node to a null node. The *solution of an OLDT refutation* is defined in the same way as that of an OLDT subrefutation.

3.3 Correctness of the Type Inference

Probably, the readers have already guessed that this type inference is safe, i.e., will not miss any atoms at calling time and exiting time during the top-down execution. More precisely, the correcteness is stated as Theorem 3.3 below. The proof of the theorem crucially depends on the fact mentioned before that $B(\mu \xrightarrow{\eta} \nu)$ is a superset of $A\mu \cap B\nu$.

Theorem 3.3 (Correctness of the Type Inference)

(a) Let $G_0\sigma_0$ be a goal, $G_0\mu_0$ be a type-abstracted goal such that $G_0\sigma_0$ is in $G_0\mu_0$, T be an extension of the initial OLD tree of $G_0\sigma_0$, and (Tr, Tb, As) be an extension of the initial OLDT structure of $G_0\mu_0$. If $A\sigma$ is the head atom of a node in T, then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a node with head type-abstracted atom $A\mu$ which includes $A\sigma$. (Correctness of the Type Inference for Calling Patterns)

(b) Let T be an extension of an initial OLD tree, (Tr, Tb, As) be an extension of an initial OLDT structure for type inference, $A\sigma$ be an atom and $A\mu$ be a type-abstracted atom such that $A\sigma$ is in $A\mu$. If T contains a unit subrefutation of $A\sigma$ with its solution $A\tau$, and $A\mu$ is the head atom of a node in Tr, then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a unit subrefutation of $A\mu$ with its solution $A\nu$ which includes $A\tau$. (Correctness of the Type Inference for Exiting Patterns)

Proof. The theorem is an immediate consequence of the following lemma by letting H be $\Box$ and v be the root node for the part (a), and by letting n be 1 for the part (b).

(a) Let γ be an OLD partial subrefutation of $(A_1, A_2, \ldots, A_n)\sigma$ ending with a node whose head atom is $A\tau$, $S = (Tr, Tb, As)$ be an extension of an initial OLDT structure for type inference, and v be a node in Tr labelled with ("$A_1, A_2, \ldots, A_n, H$", μ) such that $(A_1, A_2, \ldots, A_n)\sigma$ is in $(A_1, A_2, \ldots, A_n)\mu$. Then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a node with head type-abstracted atom $A\nu$ which includes $A\tau$.

(b) Let γ be an OLD subrefutation of $(A_1, A_2, \ldots, A_n)\sigma$ with its solution $(A_1, A_2, \ldots, A_n)\tau$, $\mathcal{S} = (Tr, Tb, As)$ be an extension of an initial OLDT structure for type inference, and v be a node in Tr labelled with ("$A_1, A_2, \ldots, A_n, H$", μ) such that $(A_1, A_2, \ldots, A_n)$ σ is in $(A_1, A_2, \ldots, A_n)\mu$. Then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains an OLDT subrefutation of $(A_1, A_2, \ldots, A_n)\mu$ for type inference starting from v with its solution $(A_1, A_2, \ldots, A_n)\nu$ which includes $(A_1, A_2, \ldots, A_n)\tau$.

The structure of the proof of the lemma is the same as that of Lemma 3.17 in Tamaki and Sato [27] pp.93–94, which is by induction on the trio $(\gamma, \mathcal{S}, v)$, ordered by the following well-founded ordering : $(\gamma, \mathcal{S}, v)$ precedes $(\gamma', \mathcal{S}', v')$ if and only if

- $|\gamma| < |\gamma'|$, or
- $|\gamma| = |\gamma'|$, and v is a solution node, but v' is a lookup node,

where $|\gamma|$ means the length of the path γ.

Base Case : When $|\gamma| = 1$, the lemma is trivial since $A\tau$ is $A_1\sigma$ for the part (a), and γ is a subrefutation of a null clause for the part (b).

Induction Step : When $|\gamma| > 1$, we will consider two cases depending on whether the node v is a solution node or a lookup node.

Case 1 : When v is a solution node, let u and γ' be the first node and the remaining path of the (partial) subrefutation γ, and C be the definite clause in P used in the first step of the subrefutation γ. Then γ' is a subrefutation of $(H_1, A_2, \ldots, A_n)\sigma'$, the OLD resolvent of $(A_1, A_2, \ldots, A_n)\sigma$ and C. By the assumption, the label ("$A_1, A_2, \ldots, A_n, H$", μ) of v is also OLDT resolvable by C, and the OLDT resolvent ("$H_1, A_2, \ldots, A_n, H$",μ') is such that $(H_1, A_2, \ldots, A_n)\sigma'$ is in $(H_1, A_2, \ldots, A_n)\mu'$ due to the property of "$\xrightarrow{\eta}$." Extending $\mathcal{S}$ (if necessary) by the OLDT resolution for type inference on the node v, we can get an OLDT structure $\mathcal{S}'$ in which v has a child node v' labelled with ("$H_1, A_2, \ldots, A_n, H$", μ'). The part (a) is immediate from the induction hypothesis for $(\gamma', \mathcal{S}', v')$. As for the part (b), by the induction hypothesis, we have an extension $\mathcal{S}''$ of $\mathcal{S}'$ which contains a subrefutation δ' of $(H_1, A_2, \ldots, A_n)\mu'$ with solution $(H_1, A_2, \ldots, A_n)\nu'$ starting from v' such that $(H_1, A_2, \ldots, A_n)\sigma'$ is in $(H_1, A_2, \ldots, A_n)\nu'$. The path in $\mathcal{S}''$ starting from v and followed by δ' constitutes the required subrefutation of $(A_1, A_2, \ldots, A_n)\mu$ due to the property of "$\xleftarrow{\eta}$."

$$\begin{tikzcd}
u : (\text{“}A_1, A_2, \ldots, A_n, G\text{”}, \sigma) \arrow[d, dash] & v : (\text{“}A_1, A_2, \ldots, A_n, H\text{”}, \mu) \arrow[d, dash] \\
u' : (\text{“}H_1, A_2, \ldots, A_n, G\text{”}, \sigma') \arrow[d, dash, "\gamma'"] & v' : (\text{“}H_1, A_2, \ldots, A_n, H\text{”}, \mu') \arrow[d, dash, "\delta'"] \\
(\text{“}G\text{”}, \sigma'') & (\text{“}H\text{”}, \mu'')
\end{tikzcd}$$

Figure 3.3.1 Correctness of the Type Inference (Case 1)

Case 2 : When v is a lookup node, there is a corresponding solution node v_0 in T labelled with ("A, H_0", μ_0) such that $A_1\mu$ is a variant of $A\mu_0$.

When γ does not contain a unit subrefutation of $A_1\sigma$ as its prefix, the part (a) is immediate from the induction hypothesis for $(\gamma, \mathcal{S}, v_0)$ by letting n be 1. (This is not the case for the part (b), since any OLD subrefutation of $(A_1, A_2, \ldots, A_n)\sigma$ contains a unit subrefutation of $A_1\sigma$ as its prefix.)

When γ contains a unit subrefutation of $A_1\sigma$ as its prefix, let γ be divided as concatenation of a subrefutation γ_1 and a (partial) subrefutation γ_2 so that γ_1 is a subrefutation of $A_1\sigma$ with its solution $A_1\tau_1$. Since $|\gamma_1| \leq |\gamma|$, and $A_1\sigma$ is in $A_1\mu$, hence $A_1\sigma$ is in $A\mu_0$, by the induction hypothesis, we have an extension $\mathcal{S}'$ of $\mathcal{S}$ such that $\mathcal{S}'$ contains a subrefutation of $A\mu_0$ with its solution $A\nu_1$ starting from v_0 such that $A_1\tau_1$ is in $A\nu_1$. By the operation at step 3 of the definition of the OLDT structure extension, the solution list of $A\mu'$ in $\mathcal{S}'$ includes the solution $A\nu_1$.

Now consider the label ("$A_1, A_2, \ldots, A_n, H$", μ) and the solution $A\nu_1$. Since ("$A_1, A_2, \ldots, A_n$", σ) and unit clause $A_1\tau_1$ have an OLD resolvent ("$A_2, \ldots, A_n$", σ'), the label ("$A_1, A_2, \ldots, A_n, H$", μ) and $A\nu_1$ also have an OLDT resolvent ("$A_2, \ldots, A_n, H$", μ') such that $(A_2, \ldots, A_n)\sigma'$ is in $(A_2, \ldots, A_n)\mu'$. This means that $\mathcal{S}'$ can be extended (if necessary) to $\mathcal{S}''$ by the operation at step 1 in the definition of the OLDT structure extension, so that the node v has a child node v' labelled with ("$A_2, \ldots, A_n, H$", μ').

Since γ_2 is a (partial) subrefutation of $(A_2, \ldots, A_n)\sigma\tau_1$ and $|\gamma_2| < |\gamma|$, The part (a) is immediate from the induction hypothesis for $(\gamma_2, \mathcal{S}'', v')$. As for the part (b), again by the induction hypothesis, we have an extension $\mathcal{S}'''$ of $\mathcal{S}''$ which contains an OLDT subrefutation δ_2 of $(A_2, \ldots, A_n)\mu'$ with its solution $(A_2, \ldots, A_n)\nu$ starting from v'' such that $(A_2, \ldots, A_n)\tau$ is in $(A_2, \ldots, A_n)\nu$. The path in $\mathcal{S}'''$ starting from v and followed by the subrefutation δ_2 constitutes the required subrefuation of $(A_1, A_2, \ldots, A_n)\mu$.

$$\begin{tikzcd}
u : (\text{"}A_1, A_2, \ldots, A_n, G\text{"}, \sigma) \arrow[d, dash, "\gamma_1"] & v : (\text{"}A_1, A_2, \ldots, A_n, H\text{"}, \mu) \arrow[d, dash, "\delta_1"] & v_0 : (\text{"}A, H_0\text{"}, \mu_0) \arrow[d, dash] \\
u' : (\text{"}A_2, \ldots, A_n, G\text{"}, \sigma') \arrow[d, dash, "\gamma_2"] & v' : (\text{"}A_2, \ldots, A_n, H\text{"}, \mu') \arrow[d, dash, "\delta_2"] & (\text{"}H_0\text{"}, \mu_0') \\
(\text{"}G\text{"}, \sigma'') & (\text{"}H\text{"}, \mu'') &
\end{tikzcd}$$

Figure 3.3.2 Correctness of the Type Inference (Case 2)

Note that any extension of the initial OLDT structure of (G, μ) in program P generates only finite number of nodes, because program P is assumed to be a *finite* set of definite clauses, hence the conditions of König's lemma are satisfied as follows:

- The number of type-abstracted atoms is finite, since the atom part of each type-abstracted atom must be an atom in the bodies of the definite clauses in P or an atom in G, and the number of type-substitutions for the variables in the atom is also finite. Hence, the extensions at solution nodes occur only finite times, since the number of head type-abstracted atoms is finite. Therefore, the length of each label is bounded by Max = "the number of the extensions of solution nodes" $\times$ "the maximum length of the bodies of the definite clauses in P" + "the length of G," since the extensions at lookup nodes only generate child nodes with shorter label. Thus, the length of each path is finite, since the number of solution nodes on it is finite, and there can't be infinite lookup nodes on it.
- Each solution node can be a parent node of only finite nodes, since program P is a *finite* set of definite clauses. Each lookup node can be a parent node of only finite nodes, since the number of type-abstracted atoms, hence that of solutions is finite. Thus, the number of branches at each node is finite.

Due to the finiteness, the process of extension under the depth-first from-left-to-right strategy (or any other strategy) always terminates. The theorem above implies that any maximally extended OLDT structure for type inference in finite steps covers the atoms at calling time and exiting time during the top-down execution of the goals in $G\mu$.

4. Implementation of the Standard Hybrid Interpretation

The readers might have felt that the processing of unit subrefutations is troublesome in the standard hybrid interpretation of Section 2. To make the conceptual presentation of the hybrid interpretation simpler, the details of how it is implemented have not been mentioned intentionally. In particular, it is not obvious in the "immediate extension of OLDT structure"

- how we can know whether a new node is the end of a unit subrefutation starting from some solution node, and
- how we can obtain the solution of the unit subrefutation efficiently if any.

In the actual implementation, we will use the following modified framework. (Although 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.)

4.1 Modified Standard Hybrid Interpretation of Prolog Programs

(1) Modified OLDT Structure

A *search tree* is a tree such that each node is classified into either a *solution node* or a *lookup node*, and is labelled with a pair of a *generalized* goal and a substitution. (We have said "generalized," because it might contain non-atoms called call-exit markers. In general, the goal part of each label is either □ or a sequence "$A, \alpha_2, \ldots, \alpha_n$" where α_i is either an atom or a *call-exit marker* of the form $[B, \tau]$. The edges are not labelled with substitutions any more.) A *search tree of* $G\sigma$ is a search tree whose root node is labelled with (G, σ). A *solution table* and an *association* are defined in the same way as before. An *OLDT structure* is a trio of a search tree, a solution table and an association.

(2) Modified OLDT Resolution

A node in a search tree of OLDT structure (Tr, Tb, As) labelled with ("$A, \alpha_2, \ldots, \alpha_n$", σ) is said to be *OLDT resolvable* when it satisfies either of the following conditions:

- The node is a terminal solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that $A\sigma$ and B are unifiable, say by an m.g.u. θ.
- The node is a lookup node of Tr, and for some substitution θ (for the variables in $A\sigma$), there is some variant of $A\sigma\theta$ in the associated solution list of the lookup node.

The OLDT resolvent is obtained through the following two phases, called the *calling phase* and the *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 inserted when a node is OLDT resolved using the solution table. When there is a call-exit marker at the leftmost of the goal part in the exiting phase, it means that some unit subrefutation is obtained.

1. Calling Phase: When a node labelled with ("$A, \alpha_2, \ldots, \alpha_n$", σ) is OLDT resolved, the intermediate label is generated as follows:
   - When the node is OLDT resolved using a definite clause "B :- $B_1, B_2, \ldots, B_m$" in program P and an m.g.u. θ, the intermediate goal part is "$B_1, B_2, \ldots, B_m$, $[A, \sigma]$, $\alpha_2, \ldots, \alpha_n$", and the intermediate substitution part τ_0 is θ.
   - When the node is OLDT resolved using (a variant of) $A\sigma\theta$ in the solution table, the intermediate goal part is "$\alpha_2, \ldots, \alpha_n$", and the intermediate substitution part τ_0 is $\sigma\theta$.
2. 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 goal part, the label of the new node is generated as follows:

- The goal part is obtained by eliminating all these call-exit markers. The substitution part is $\sigma_k \cdots \sigma_2\sigma_1\tau_0$.
- 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 4.1. The processing in the calling phase is performed in the first **case** statement, while that in the exiting phase is performed in the second **while** statement successively. Note that each node is labelled, say with (G, σ), in such a way that the following property holds: "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 goal 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 a unit subrefutation of $A_j\sigma_j$. The solution $A_j\tau_j$ is added to the solution list of $A_j\sigma_j$.

```
OLDT-resolve(("$A, \alpha_2, \ldots, \alpha_n$", $\sigma$) : label) : label ;
    $i$ := 0;
    case
      when a solution node is OLDT resolved with "$B$ :- $B_1, B_2, \ldots, B_m$" in $P$
        let $\theta$ be the m.g.u. of $A\sigma$ and $B$ ;
        let $G_0$ be a generalized goal "$B_1, B_2, \ldots, B_m$, $[A, \sigma]$, $\alpha_2, \ldots, \alpha_n$" ;
        let $\tau_0$ be the substitution $\theta$ ;                                        — (A)
      when a lookup node is OLDT resolved with "$A\sigma\theta$" in $Tb$
        let $G_0$ be a generalized goal "$\alpha_2, \ldots, \alpha_n$" ;
        let $\tau_0$ be the substitution $\sigma\theta$ ;                                  — (B)
    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 ;
    return $(G_i, \tau_i)$.
```

Figure 4.1 Modified OLDT Resolution for Standard Hybrid Interpretation

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

(3) Modified OLDT Refutation

The *initial OLDT structure* and *extension of OLDT structure* are defined in the same way as before. An *OLDT refutation of $G\sigma$* is a path in the search tree of some extension of the initial OLDT structure of $G\sigma$ from the root node to a null node. Let τ be the substitution part of the null node. Then the *solution of the refutation* is $G\tau$.

Note that we no longer need to keep the edges, the non-terminal solution nodes and the null nodes of search trees.

4.2 An Example of the Modified Standard Hybrid Interpretation

Let us show an example of how the modified standard hybrid interpretation works. Consider the example in Section 2.1 again. The modified standard hybrid interpretation generates the following OLDT structures of $reach(a, Z_0)$.

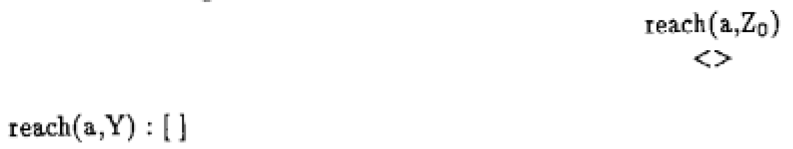


Figure 4.2.1 Modified Standard Hybrid Interpretation at Step 1

First, the initial OLDT structure above is generated.

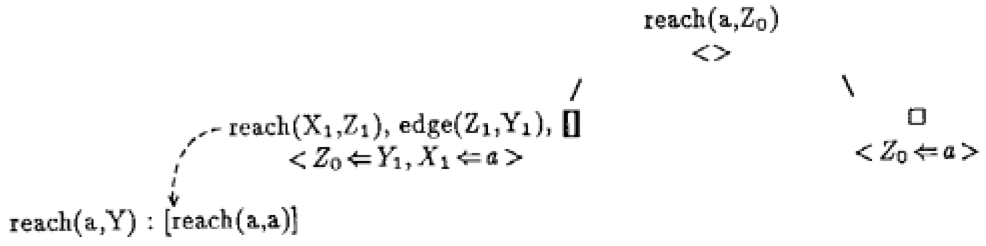


Figure 4.2.2 Modified Standard Hybrid Interpretation at Step 2

Secondly, the root node ("$reach(a, Z_0)$",<>) is OLDT resolved using the program to generate two child nodes. The intermediate label of the left child node is

("$reach(X_1, Z_1), edge(Z_1, Y_1), [reach(a, Z_0), <>]$", $< Z_0 \Leftarrow Y_1, X_1 \Leftarrow a >$).

It is the new label immediately, since its leftmost is not a call-exit marker. (Due to space limit, the call-exit markers are represented by [] in the figures hereafter.) The intermediate label of the right child node is

("$[reach(a, Z_0), <>]$", $< Z_0 \Leftarrow a, X_2 \Leftarrow a >$).

By eliminating the leftmost call-exit marker and composing the substitutions, the new label is (□, $< Z_0 \Leftarrow a >$). (We have omitted the assignments irrelevant to the top-level goal $reach(a, Z_0)$.) During the elimination of the call-exit marker, $reach(a, a)$ is added to the solution table.

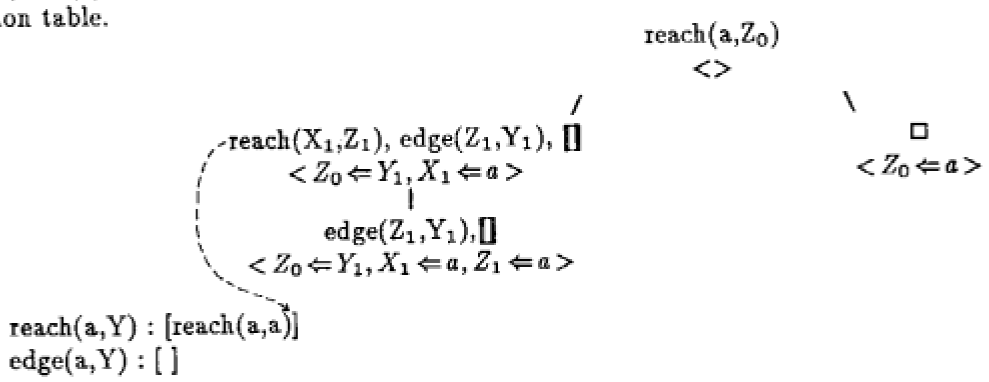

reach(a,Z₀)
<>
reach(X₁,Z₁), edge(Z₁,Y₁), []
< Z₀ ⇐ Y₁, X₁ ⇐ a >
edge(Z₁,Y₁),[]
< Z₀ ⇐ Y₁, X₁ ⇐ a, Z₁ ⇐ a >
□
< Z₀ ⇐ a >
reach(a,Y) : [reach(a,a)]
edge(a,Y) : [ ]

Figure 4.2.3 Modified Standard Hybrid Interpretation at Step 3

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

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

("$[\mathit{edge}(Z_1,Y_1), <Z_0 \Leftarrow Y_1, X_1 \Leftarrow a, Z_1 \Leftarrow a>]$, $[\mathit{reach}(a,Z_0), <>]$", $<Y_1 \Leftarrow b>$).
By eliminating the leftmost call-exit markers and composing substitutions, the new label is ($\Box$, $<Z_0 \Leftarrow b>$). 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

("$[\mathit{edge}(Z_1,Y_1), <Z_0 \Leftarrow Y_1, X_1 \Leftarrow a, Z_1 \Leftarrow a>]$, $[\mathit{reach}(a,Z_0), <>]$", $<Y_1 \Leftarrow c>$)
By eliminating the leftmost call-exit markers and composing substitutions similarly, the new label is ($\Box$, $<Z_0 \Leftarrow c>$). This time, *edge*(a,c) and *reach*(a,c) are added to the solution table during the elimination of the call-exit markers.

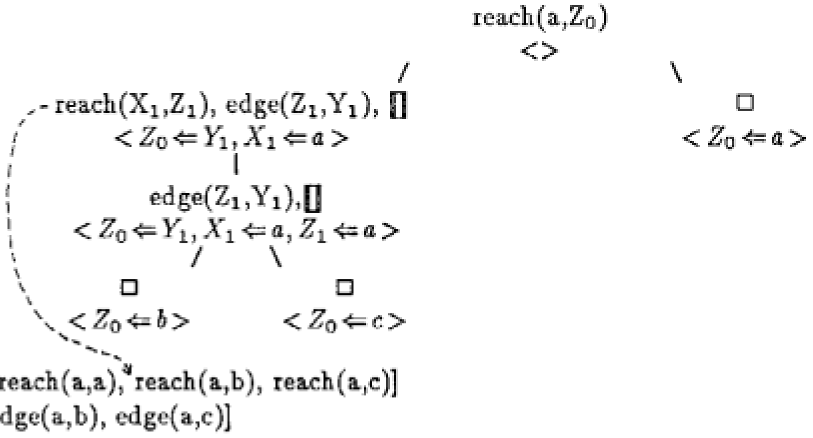

reach(a,Z0)
<>
reach(X1,Z1), edge(Z1,Y1), []
< Z0 ⇐ Y1, X1 ⇐ a >
edge(Z1,Y1),[]
< Z0 ⇐ Y1, X1 ⇐ a, Z1 ⇐ a >
< Z0 ⇐ b >
< Z0 ⇐ c >
< Z0 ⇐ a >
reach(a,Y) : [reach(a,a), reach(a,b), reach(a,c)]
edge(a,Y) : [edge(a,b), edge(a,c)]

Figure 4.2.4 Modified Standard Hybrid Interpretation at Step 4

The extension proceeds similarly to obtain all the solutions as in Section 2.1

4.3. Correctness of the Modified Standard Hybrid Interpretation

This modified standard hybrid interpreter is a correct implementation of the standard hybrid interpreter in Section 2, hence the same theorem as Theorem 2.3 holds.

Theorem 4.3 (Correctness of the Modified OLDT Resolution)

(a) Let $G_0\sigma_0$ be a goal, T be an extension of the initial OLD tree of $G_0\sigma_0$, and (Tr, Tb, As) be an extension of the initial OLDT structure of $G_0\sigma_0$. If $A\sigma$ is the head atom of a node in T, then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a node with head atom $A\sigma'$ which is a variant of $A\sigma$. (Correctness for Calling Patterns)

(b) Let T be an extension of an initial OLD tree, (Tr, Tb, As) be an extension of an initial OLDT structure, $A\sigma$ and $A\sigma'$ be atoms such that $A\sigma$ is a variant of $A\sigma'$. If T contains a unit subrefutation of $A\sigma$ with its solution $A\tau$, and $A\sigma'$ is the head atom of a node in Tr, then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a unit subrefutation of $A\sigma'$ with its solution $A\tau'$ which is a variant of $A\tau$. (Correctness for Exiting Patterns)

Proof. To prove the theorem, it suffices to show that there exists an extension of an initial OLDT structure of the standard hybrid interpreter in Section 2 if and only if there exists an extension of an initial OLDT structure of the modified standard hybrid interpreter in Section 4 satisfying the following correspondence:

(a) The corresponding search trees have the identical form and satisfies the following conditions:
  - The goal parts of the corresponding nodes are identical except for call-exit markers (if any).
  - The head atoms of the corresponding nodes are identical (although the substitution parts are not necessarily identical).
  - The computed solutions of unit subrefutations are identical.

(b) The corresponding solution tables are identical.

(c) The corresponding associations are identical.

Due to space limit, we will omit the details of the proof.

5. Implementation of the Abstract Hybrid Interpretation for Type Inference

The abstract hybrid interpretation for type inference is modified as well according to the modification of the standard hybrid interpretation.

5.1 Modified Type Inference for Prolog Programs

(1) Modified OLDT Structure for Type Inference

A *search tree*, a *solution table* and an *association for type inference* are defined in the same way as the modified standard hybrid interpretation in Section 4 except that call-exit markers are of the form $[B, \nu, \eta]$. An *OLDT structure for type inference* is a trio of a search tree, a solution table and an association.

```
OLDT-resolve(("$A, \alpha_2, \ldots, \alpha_n$", $\mu$) : label) : label ;
    $i$ := 0;
    case
      when a solution node is OLDT resolved with "$B$ :- $B_1, B_2, \ldots, B_m$" in $P$
        let $\eta$ be the m.g.u. of $A$ and $B$ ;
        let $G_0$ be a generalized goal "$B_1, B_2, \ldots, B_m, [A, \mu, \eta], \alpha_2, \ldots, \alpha_n$";
        let $\nu_0$ be "$\mu \xrightarrow{\eta} <>$" ;                                         — (A)
      when a lookup node is OLDT resolved with "$A(\mu \vee \lambda)$" in $Tb$
        let $G_0$ be a generalized goal "$\alpha_2, \ldots, \alpha_n$" ;
        let $\nu_0$ be "$\mu \vee \lambda$" ;                                                  — (B)
    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 "$\mu_{i+1} \xleftarrow{\eta_{i+1}} \nu_i$" ;                                  — (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 ;
        $i$ := $i + 1$ ;
    return ($G_i, \nu_i$).
```

Figure 5.1 Modified OLDT Resolution for Type Inference

(2) Modified OLDT Resolution for Type Inference

A node in a search tree of OLDT structure (Tr, Tb, As) labelled with ("$A, \alpha_2, \ldots, \alpha_n$", μ) is said to be *OLDT resolvable* when $\mu(X) \neq \emptyset$ for any variable X in $A, A_2, \ldots, A_n$, and $A\mu$ satisfies either of the following conditions:

- The node is a terminal solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that A and B are unifiable, say by an m.g.u. η.
- The node is a lookup node of Tr, and for some type substitution λ (for the variables in A), there is some variant of $A(\mu \vee \lambda)$ in the associated solution list of the lookup node.

The precise algorithm of OLDT resolutnion for type inference is shown in Figure 5.1. Note that only the operations at steps (A), (B) and (C) differ from those in Figure 4.1.

A node labelled with ("$A, \alpha_2, \ldots, \alpha_n$", μ) is a lookup node when a variant of $A\mu$ is a key in the solution table, and is a solution node otherwise.

(3) Modified OLDT Refutation for Type Inference

The *initial OLDT structure*, *extension of OLDT structure* and *OLDT refutation for type inference* are defined in the same way as in Section 4.

5.2 An Example of the Modified Type Inference

Let us show a different example from the one in Section 3. Consider the following program defining "*mult*" and "*add*."

```
mult(zero,Y,zero).
mult(suc(X),Y,Z) :- mult(X,Y,W), add(Y,W,Z).
add(zero,Y,Y).
add(suc(X),Y,suc(Z)) :- add(X,Y,Z).
```

Then the type inference of $mult(X_0, Y_0, Z_0) <>$ proceeds as follows:

First, the initial OLDT structure below is generated.

$$\begin{array}{c} \text{mult}(X_0,Y_0,Z_0) \\ \langle\rangle \end{array}$$

mult(X,Y,Z)<> : [ ]

Figure 5.2.1 Modified Type Inference at Step 1

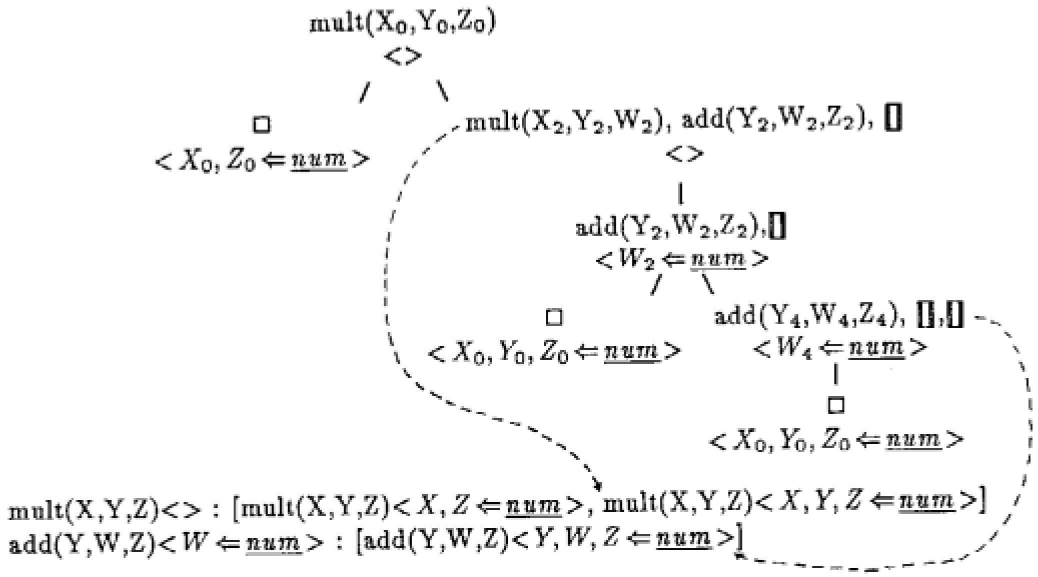
Secondly, the root node ("$mult(X_0, Y_0, Z_0)$",<>) is OLDT resolved using the program. The left child node gives a solution $mult(X_0, Y_0, Z_0) < X_0, Z_0 \Leftarrow \underline{num} >$. The right child node is a lookup node.

$$\begin{tikzcd}
& \begin{array}{c}\text{mult}(X_0,Y_0,Z_0)\\ \langle\rangle\end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{array}{c}\square\\ \langle X_0, Z_0 \Leftarrow \underline{num}\rangle\end{array} & & \begin{array}{c}\text{mult}(X_2,Y_2,W_2), \text{add}(Y_2,W_2,Z_2), []\\ \langle\rangle\end{array} \arrow[dll, dashed, bend left] \\
\text{mult}(X,Y,Z)\langle\rangle : [\mathit{mult}(X,Y,Z) \langle X, Z \Leftarrow \underline{num}\rangle] & &
\end{tikzcd}$$

Figure 5.2.2 Modified Type Inference at Step 2

Thirdly, the lookup node is OLDT resolved using the solution table. The generated child node is a solution node. Fourthly, the solution node is OLDT resolved further using the program. The left child node gives two solutions $add(Y_2, W_2, Z_2) < Y_2, W_2, Z_2 \Leftarrow \underline{num} >$

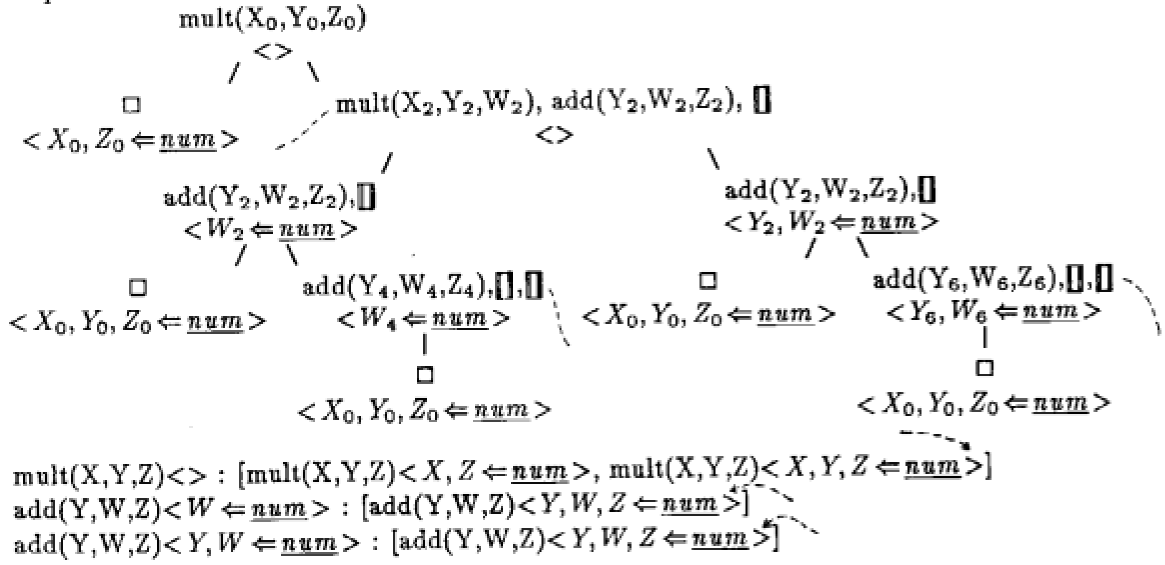
and $mult(X_0, Y_0, Z_0) < X_0, Y_0, Z_0 \Leftarrow \underline{num} >$. The right child node is a lookup node. Fifthly, the lookup node is OLDT resolved using the solution table.



mult(X₀,Y₀,Z₀) <>
□ < X₀, Z₀ ⇐ num >
mult(X₂,Y₂,W₂), add(Y₂,W₂,Z₂), [] <>
add(Y₂,W₂,Z₂),[] < W₂ ⇐ num >
□ < X₀, Y₀, Z₀ ⇐ num >
add(Y₄,W₄,Z₄), [],[] < W₄ ⇐ num >
□ < X₀, Y₀, Z₀ ⇐ num >
mult(X,Y,Z)<> : [mult(X,Y,Z)< X, Z ⇐ num >, mult(X,Y,Z)< X, Y, Z ⇐ num >]
add(Y,W,Z)< W ⇐ num > : [add(Y,W,Z)< Y, W, Z ⇐ num >]

Figure 5.2.3 Modified Type Inference at Step 5

Sixthly, the first lookup node is OLDT resolved using the new solution. The generated child node is a solution node.Seventhly, the generated solution node is OLDT resolved using the program. The left child node gives a new solution $add(Y_4, W_4, Z_4) < Y_4, W_4, Z_4 \Leftarrow \underline{num} >$. The right child node is a lookup node. Lastly, the lookup node is OLDT resolved using the solution table. Because the generated child node gives no new solution, the extension process stops.



mult(X₀,Y₀,Z₀) <>
□ < X₀, Z₀ ⇐ num >
mult(X₂,Y₂,W₂), add(Y₂,W₂,Z₂), [] <>
add(Y₂,W₂,Z₂),[] < W₂ ⇐ num >
□ < X₀, Y₀, Z₀ ⇐ num >
add(Y₄,W₄,Z₄),[],[] < W₄ ⇐ num >
□ < X₀, Y₀, Z₀ ⇐ num >
add(Y₂,W₂,Z₂),[] < Y₂, W₂ ⇐ num >
□ < X₀, Y₀, Z₀ ⇐ num >
add(Y₆,W₆,Z₆),[],[] < Y₆, W₆ ⇐ num >
□ < X₀, Y₀, Z₀ ⇐ num >
mult(X,Y,Z)<> : [mult(X,Y,Z)< X, Z ⇐ num >, mult(X,Y,Z)< X, Y, Z ⇐ num >]
add(Y,W,Z)< W ⇐ num > : [add(Y,W,Z)< Y, W, Z ⇐ num >]
add(Y,W,Z)< Y, W ⇐ num > : [add(Y,W,Z)< Y, W, Z ⇐ num >]

Figure 5.2.4 Modified Type Inference at Step 8

This problem is not so trivial as one might think at first glance. For example, Suppose that the predicate "*mult*" is defined by

```
    mult(zero,Y,zero).
    mult(suc(X),Y,Z) :- mult(X,Y,W), add(W,Y,Z).
```

by exchanging the first and the second arguments of "*add*." Then, one of the exit pattern of *mult* $(X, Y, Z) <>$ is $mult(X, Y, Z) < X \Leftarrow \underline{num} >$, hence, we can't conclude that the third argument is a number. For example, $mult(suc(zero), Y, Y)$ succeeds for any Y.

5.3. Correctness of the Modified Type Inference

This modified type inference is a correct implementation of the type inference in Section 3, hence the same theorem as Theorem 3.3 holds.

Theorem 5.3 (Correctness of the Modified Type inference)

(a) Let $G_0\sigma_0$ be a goal, $G_0\mu_0$ be a type-abstracted goal such that $G_0\sigma_0$ is in $G_0\mu_0$, T be an extension of the initial OLD tree of $G_0\sigma_0$, and (Tr, Tb, As) be an extension of the initial OLDT structure of $G_0\mu_0$. If $A\sigma$ is the head atom of a node in T, then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a node with head type-abstracted atom $A\mu$ which includes $A\sigma$. (Correctness of the Type Inference for Calling Patterns)

(b) Let T be an extension of an initial OLD tree, (Tr, Tb, As) be an extension of an initial OLDT structure for type inference, $A\sigma$ be an atom and $A\mu$ be a type-abstracted atom such that $A\sigma$ is in $A\mu$. If T contains a unit subrefutation of $A\sigma$ with its solution $A\tau$, and $A\mu$ is the head atom of a node in Tr, then there exists an extension of (Tr, Tb, As) such that the search tree of the extension contains a unit subrefutation of $A\mu$ with its solution $A\nu$ which includes $A\tau$. (Correctness of the Type Inference for Exiting Patterns)

Proof. It suffices to show that there exists an extension of an initial OLDT structure for the type inference in Section 3 if and only if there exists an extension of an initial OLDT structure for the modified type inference in Section 5 satisfying the same correspondence conditions as the proof of Theorem 4.3. Again, due to space limit, we will omit the details of the proof.

6. Depth-abstracted Term Inference for Prolog Programs

The type inference is the problem of describing supersets of terms assigned to the variables in the atoms at calling time and exiting time *in terms of data types*. In general, we call it *goal pattern inference* to describe supersets of terms assigned to the variables in the atoms at calling time and exiting time during the top-down execution of a given top-level goal. In this section, we will show the implementation of another goal pattern inference.

Suppose that a top-level goal is executed with its arguments instantiated to terms of specific form. Then, how can we know what form of arguments the atoms have when they are invoked or they succeed during the top-down execution of the top-level goal? In particular, can we say that some predicate is always invoked or succeeds with its argument instantiated to some instance of a specific term? We will reformulate the work by Sato and Tamaki [24] from the viewpoint of abstract hybrid interpretation.

(1) Depth-abstracted Term and Depth-abstracted Substitution

The *depth* of term t is defined as follows:

- When t is a variable, the depth of t is 0.

- When t is a constant c, the depth of t is 0.
- When t is a term of the form $f(t_1, t_2, \ldots, t_n)$, the depth of t is the maximum of the depths of $t_1, t_2, \ldots, t_n$, plus 1.

A *depth d abstracted term* is a term whose depth is at most d, and denotes the set of all its instances. Note that the set of all depth d abstracted terms is finite for any given d, because program P is a finite set of definite clauses, hence there appear only finite function symbols.

A *depth d abstracted substitution* is an expression of the form
$< X_1 \Leftarrow \underline{t_1}, X_2 \Leftarrow \underline{t_2}, \ldots, X_l \Leftarrow \underline{t_l} >$,
where $\underline{t_1}, \underline{t_2}, \ldots, \underline{t_l}$ are depth d abstracted terms. Depth-abstracted substitutions are denoted by μ, ν, λ in this section.

(2) Depth-abstracted Atom and Depth-abstracted Goal

Let A be an atom and μ be a depth d abstracted substitution of the form
$< X_1 \Leftarrow \underline{t_1}, X_2 \Leftarrow \underline{t_2}, \ldots, X_l \Leftarrow \underline{t_l} >$.
Then $A\mu$ (or pair (A, μ)) is called a *depth-abstracted atom*, and denotes the set of all atoms obtained by replacing each X_i in A with a term in $\underline{t_i}$. An *instance* and a *variant* are defined in the same way as usual atoms.

Similarly, $G\mu$ (or pair (G, μ)) is called a *depth-abstracted goal*, and denotes the set of goals obtained by replacing each X_i in G with a term in $\underline{t_i}$.

(3) Unification of Depth-abstracted Atoms

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 non-constant subterm of t with a newly created distinct variable is called the *depth d abstraction of t*, and denoted by $[t]_d$.

Example 6 Let t be a term $f(g(X, a), Y, b)$ and U, V be fresh variables ([24] p.642). Then

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

Note that Y and b are not replaced with new variables when $d = 1$, and neither are X and a when $d = 2$.

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
$< X_1 \Leftarrow [t_1], X_2 \Leftarrow [t_2]_d, \ldots, X_l \Leftarrow [t_l]_d >$
is called the *depth d abstraction of θ*, and denoted by $[\theta]_d$.

Two depth-abstracted atoms $A\mu$ and $B\nu$ are said to be *unifiable* when $A\mu \cap B\nu \neq \emptyset$. When $A\mu$ and $B\nu$ are unifiable, atoms $A\mu$ and $B\nu$ are unifiable in the usual sense. Let θ be an m.g.u. of $A\mu$ and $B\nu$. Then $B[\nu\theta]_d$ is a superset of $A\mu \cap B\nu$.

(4) OLDT Resolution for Depth-abstracted Term Inference

The *search tree*, *solution table*, *association* and *OLDT structure for depth-abstracted term inference* are defined in the same way as the modified type inference in Section 5.

A node in a search tree of OLDT structure (Tr, Tb, As) labelled with ("$A, \alpha_2, \ldots, \alpha_n$", μ) is said to be *OLDT resolvable* when it satisfies either of the following conditions:

- The node is a terminal solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that $A\mu$ and B are unifiable, say by an m.g.u. θ.
- The node is a lookup node of Tr, and for some substitution θ (for the variables in $A\mu$), there is some variant of $A\mu\theta$ in the associated solution list of the lookup node.

In either cases, the substitution θ is called the *substitution of the OLDT resolution.*

The precise algorithm of OLDT resolutuion for depth-abstracted pattern enumeration is shown in Figure 6. Note that only the operations at steps (A), (B) and (C) differ from those in Figure 5.1.

```
OLDT-resolve(("$A, \alpha_2, \ldots, \alpha_n$", $\mu$) : label) : label ;
    $i := 0$;
    case
      when a solution node is OLDT resolved with "$B$ :- $B_1, B_2, \ldots, B_m$" in $P$
        let $\theta$ be the m.g.u. of $A\mu$ and $B$ ;
        let $G_0$ be a generalized goal "$B_1, B_2, \ldots, B_m$, $[A, \mu]$, $\alpha_2, \ldots, \alpha_n$" ;
        let $\nu_0$ be $[\theta]_d$ ;                                                    — (A)
      when a lookup node is OLDT resolved with $A\mu\theta$ in $Tb$
        let $G_0$ be a generalized goal "$\alpha_2, \ldots, \alpha_n$" ;
        let $\nu_0$ be $\mu\theta$ ;                                                     — (B)
    while the leftmost of $G_i$ is a call-exit marker $[A_{i+1}, \mu_{i+1}]$ do
        let $G_{i+1}$ be $G_i$ other than the leftmost call-exit marker ;
        let $\nu_{i+1}$ be $[\mu_{i+1}\nu_i]_d$ ;                                        — (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 ;
        $i := i + 1$ ;
    return $(G_i, \nu_i)$.
```

Figure 6 OLDT Resolution for Depth-abstracted Term Inference

The *initial OLDT structure*, *extension of OLDT structure* and *OLDT refutation for depth-abstracted term inference* are defined in the same way as the modified type inference in Section 5.

7. Mode Inference for Prolog Programs

In this section, we will show one more goal pattern inference by our abstract hybrid interpretation. Suppose that a top-level goal "*reverse*(L_0, M_0)" is executed with its first argument L_0 instantiated to a ground term. Then, the first argument of "*reverse*" 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 "*append*" at calling time and the third argument at exiting time. How can we show it mechanically? We will reformulate the work by Mellish [21],[22] and Debray and Warren [10] from the viewpoint of abstract hybrid interpretation.

(1) Mode

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.

The *instantition ordering of modes* is the ordering $\prec$ depicted left below, while the *set-inclusion ordering of modes* is the ordering $\subset$ depicted right below:

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

Again, the instantiation ordering and the set-inclusion ordering are the reverse of each other. (To make our explanation simple, we will consider the simplest mode structure first. A little more complicated mode structure is discussed in Section 7 (5).)

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. Mode substitutions are denoted by μ, ν, λ in this section. We assume 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.

The *joined mode substitution* of two mode substitutions μ and ν, denoted by $\mu \vee \nu$, is the substitution such that $\mu \vee \nu(X)$ is the least upper bound of $\mu(X)$ and $\nu(X)$ w.r.t. the instantiation ordering.

(2) Mode-abstracted Atom and Mode-abstracted Goal

Let A be an atom and 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 $A\mu$ (or pair (A, μ)) 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 $\underline{m_i}$. An *instance* and a *variant* are defined in the same way as type-abstracted atoms.

Similarly, $G\mu$ (or pair (G, μ)) is called a *mode-abstracted goal*, and denotes the set of goal obtained by replacing each X_i in G with a term in $\underline{m_i}$.

(3) Unification of Mode-abstracted Atoms

Two mode-abstracted atoms $A\mu$ and $B\nu$ are said to be *unifiable* when $A\mu \cap B\nu \neq \emptyset$. Similarly to the type inference, it suffices to define the *inwards mode propagation* and the *outwards mode propagation*. They are defined as below:

$$s / < t \Leftarrow \underline{m} > = \underline{m}.$$

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

The *propagated mode substitution* "$\mu \xrightarrow{\eta} \nu$" (or "$\nu \xleftarrow{\eta} \mu$") is defined using the inward mode propagation and outwards mode propagation in the same way as the type inference. Again, note that $B(\mu \xrightarrow{\eta} \nu)$ is a superset of $A\mu \cap B\nu$.

(4) OLDT Resolution for Mode Inference

The *search tree*, *solution table*, *association* and *OLDT structure for mode inference* are defined in the same way as the modified type inference in Section 5.

A node in a search tree of OLDT structure (Tr, Tb, As) labeled with ("$A, \alpha_2, \dots, \alpha_n$", μ) is said to be *OLDT resolvable* when $\mu(X)$ is not $\emptyset$ for any variable X and $A\mu$ satisfies either of the following conditions:

- The node is a terminal solution node of Tr, and there is some definite clause "B :- $B_1, B_2, \dots, B_m$" ($m \geq 0$) in program P such that A and B is unifiable, say by an m.g.u. η.
- The node is a lookup node of Tr, and for some mode substitution λ (for the variables in A), there is some variant of $A(\mu \vee \lambda)$ in the associated solution list of the lookup node.

The precise algorithm of OLDT resolution for mode inference is shown in Figure 7. Note that only the operations at steps (A), (B) and (C) differ from those in Figure 5.1.

```
OLDT-resolve(("$A, \alpha_2, \dots, \alpha_n$", $\mu$) : label) : label ;
    $i := 0$ ;
    case
      when a solution node is OLDT resolved with "$B$ :- $B_1, B_2, \dots, B_m$" in $P$
        let $\eta$ be the m.g.u. of $A$ and $B$ ;
        let $G_0$ be a generalized goal "$B_1, B_2, \dots, B_m$, $[A, \mu, \eta]$, $\alpha_2, \dots, \alpha_n$" ;
        let $\nu_0$ be "$\mu \xrightarrow{\eta} <>$" ;                                    — (A)
      when a lookup node is OLDT resolved with "$A(\mu \vee \lambda)$" in $Tb$
        let $G_0$ be a generalized goal "$\alpha_2, \dots, \alpha_n$" ;
        let $\nu_0$ be "$\mu \vee \lambda$" ;                                              — (B)
    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 "$\mu_{i+1} \xleftarrow{\eta_{i+1}} \nu_i$" ;                  — (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 ;
        $i := i + 1$ ;
    return $(G_i, \nu_i)$.
```

Figure 7 OLDT Resolution for Mode Inference (3 Modes)

The *initial OLDT structure*, *extension of OLDT structure* and *OLDT refutation for mode inference* are defined in the same way as the modified type inference in Section 5.

(5) Mode Inference with 4 Modes

The mode structure with 3 modes we have considered is the simplest one. For more complicated mode structures, we are sometimes unable to ignore the possibility that two variables are bound to shared structures, as was pointed out by Debray and Warren [10].

Suppose that a *mode* is one of the following 4 sets of terms:

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

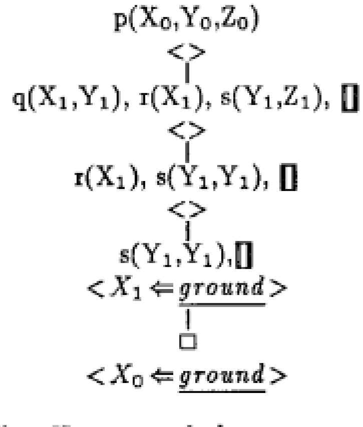
Then, the instantiation ordering and the set-inclusion ordering are as below:

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

Note that the instantiation ordering and the set-inclusion ordering are not the reverse of each other, e.g., $\underline{variable} \prec \underline{ground}$, but $\underline{variable} \not\supset \underline{ground}$. Similarly, $\underline{variable} \prec \underline{any}$, but $\underline{variable} \not\supset \underline{any}$.

A *mode substitution* is defined in the same way as before. This time, we assume that a mode substitution assigns $\underline{variable}$, 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 $\underline{variable}$ to every variable. The *joined mode substitution* of μ and ν, denoted by $\mu \vee \nu$, is defined in the same way as before.

If the propagated mode substitution were defined in the same way as before, the mode inference with 4 modes would not work safely. The reason is that we no longer enjoy the property that "if $m_1 \preceq m_2$ then $m_1 \supseteq m_2$." To make the mode inference with 4 modes safe, we need to infer the possibility that two variables are bound to terms with shared structures.



$p(X_0,Y_0,Z_0)$
<>
$q(X_1,Y_1)$, $r(X_1)$, $s(Y_1,Z_1)$, []
<>
$r(X_1)$, $s(Y_1,Y_1)$, []
<>
$s(Y_1,Y_1)$, []
$<X_1 \Leftarrow \underline{ground}>$
□
$<X_0 \Leftarrow \underline{ground}>$

p(X,Y,Z) <> : [p(X,Y,Z) $<X \Leftarrow \underline{ground}>$]
q(X,Y) <> : [q(X,Y) <>]
r(X) <> : [r(X) $<X \Leftarrow \underline{ground}>$]
s(Y,Z) <> : [s(Y,Z) <>]

Figure 7.2 Wrong Mode Inference with 4 Modes

Example 7 The following is a slight modification of the example given by Debray and Warren [10]. Let the given program P be

```
    p(X,Y,Z) :- q(X,Y), r(X), s(Y,Z).
    q(X,X).
    r(a).
    s(Y,Y).
```

If we had not the additional information about the sharing between variables, the mode inference of $p(X_0, Y_0, Z_0)$ $<>$ by our abstract hybrid interpretation would proceed as in Figure 7.2. Though Y_1 must not be a variable when the goal $s(Y_1, Z_1)$ is called, it is not correctly inferred, because the sharing of structures between the variables X_1 and Y_1 caused by the unification of $q(X_1, Y_1)$ and $q(Z, Z)$ is not considered. Similarly, Y_0 must not be a variable when the goal $p(X_0, Y_0, Z_0)$ suceeds.

A *sharing* is an equivalence relation on the set of variables $X_1, X_2, \ldots, X_l$, where $X_1, X_2, \ldots, X_l$ are distinct variables, called the *domain variables* of the sharing. Sharings are denoted by $\mathcal{M}, \mathcal{N}, \mathcal{L}$. A sharing $\mathcal{M}$ says that, if (X, Y) is not in $\mathcal{M}$, then X and Y are never bound to terms with shared structures. The *restriction of $\mathcal{M}$ to the set of variables $\mathcal{V}$* is the sharing consisting of all the pairs in both $\mathcal{M}$ and $\mathcal{V} \times \mathcal{V}$. The *identity sharing*, denoted by $\mathbf{1}$, is the identity binary relation.

The *joined sharing* of $\mathcal{M}$ and $\mathcal{N}$, denoted by $\mathcal{M} \vee \mathcal{N}$, is the transitive closure of the union of $\mathcal{M}$ and $\mathcal{N}$.

Let μ, ν be mode substitutions, and $\mathcal{M}, \mathcal{N}$ be sharings. The *join* of $\mu\mathcal{M}$ and $\nu\mathcal{N}$, denoted by $\mu\mathcal{M} \vee \nu\mathcal{N}$, is $\lambda\mathcal{L}$ such that

$$\lambda(X) = \begin{cases} \underline{any}, & \text{when } \mu(X) \text{ is } \underline{variable}\text{, and there exits } Y \text{ such that} \\ & (X, Y) \text{ is in } \mathcal{M} \vee \mathcal{N} \text{ and } (\mu \vee \nu)(Y) \text{ is } \underline{any} \text{ or } \underline{ground}; \\ (\mu \vee \nu)(X), & \text{otherwise.} \end{cases}$$

$$\mathcal{L} = \mathcal{M} \vee \mathcal{N}.$$

Let A be an atom, μ be a mode substitution of the form

$< X_1 \Leftarrow \underline{m_1}, X_2 \Leftarrow \underline{m_2}, \ldots, X_l \Leftarrow \underline{m_l} >$,

and $\mathcal{M}$ be a sharing on variables $X_1, X_2, \ldots, X_l$. Then $A\mu\mathcal{M}$ (or trio $(A, \mu, \mathcal{M})$) is called a *mode-sharing-abstracted atom*, and denotes the set of all atoms obtained by replacing each X_i in A with a term in $\underline{m_i}$ without contradicting $\mathcal{M}$. (We will consider only the restriction of μ and $\mathcal{M}$ to the variables in A when $A\mu\mathcal{M}$ is considered.) A mode-sharing-abstracted atom $A\nu\mathcal{N}$ is said to be an *instance* of a mode-sharing-abstracted atom $A\mu\mathcal{M}$ when $A\nu$ is an instance of $A\mu$ and $\mathcal{N}$ is a superset of $\mathcal{M}$ as binary relations. A mode-sharing-abstracted atom $B\nu\mathcal{N}$ is called a *variant* of a mode-sharing-abstracted atom $A\mu\mathcal{M}$ when B is a variant of A and ν and $\mathcal{N}$ is obtained from μ and $\mathcal{M}$ by renaming the variables in the domain of μ and $\mathcal{M}$ accordingly.

Similarly, $G\mu\mathcal{M}$ (or trio $(G, \mu, \mathcal{M})$) is called a *mode-sharing-abstracted goal*, and denotes the set of all goals obtained by replacing each X_i in A with a term in $\underline{m_i}$ without contradicting $\mathcal{M}$.

Two mode-sharing-abstracted atoms $A\mu\mathcal{M}$ and $B\nu\mathcal{N}$ are said to be *unifiable* when $A\mu\mathcal{M} \cap B\nu\mathcal{N} \neq \emptyset$. Let A be an atom, $X_1, X_2, \ldots, X_k$ all the variables in A, μ a mode substitution

$< X_1 \Leftarrow \underline{m_1}, X_2 \Leftarrow \underline{m_2}, \ldots, X_k \Leftarrow \underline{m_k}, \ldots >$,

$\mathcal{M}$ a sharing on $X_1, X_2, \ldots, X_k, \ldots$, B an atom, $Y_1, Y_2, \ldots, Y_l$ all the variables in B, ν a mode substitution

$<Y_1 \Leftarrow \underline{n_1}, Y_2 \Leftarrow \underline{n_2}, \ldots, Y_l \Leftarrow \underline{n_l}, \ldots>$.

$\mathcal{N}$ a sharing on $Y_1, Y_2, \ldots, Y_l, \ldots$, and η an m.g.u. of A and B. Then, the *propagated mode-sharing from $\mu\mathcal{M}$ to $\nu\mathcal{N}$ through η*, denoted by "$\mu\mathcal{M} \xrightarrow{\eta} \nu\mathcal{N}$" (or "$\nu\mathcal{N} \xleftarrow{\eta} \mu\mathcal{M}$"), is defined as follows:

1. Let ν' be the propagated mode substitution from μ to $<>$ through η computed in the same way as the mode inference with 3 modes.
2. Let $\mathcal{N}'$ be the sharing on $Y_1, Y_2, \ldots, Y_l$ such that (Y_i, Y_j) is in $\mathcal{N}'$ if and only if
   - $\eta(X_n)$ and $\eta(Y_i)$ contains an identical variable for some X_n,
   - $\eta(X_m)$ and $\eta(Y_j)$ contains an identical variable for some X_m, and
   - (X_n, X_m) is in $\mathcal{M}$.
3. Then, "$\mu\mathcal{M} \xrightarrow{\eta} \nu\mathcal{N}$" is $\nu\mathcal{N} \vee \nu'\mathcal{N}'$.

Note that $B(\mu\mathcal{M} \xrightarrow{\eta} \nu\mathcal{N})$ is a superset of $A\mu\mathcal{M} \cap B\nu\mathcal{N}$.

An *OLDT structure for mode inference (with 4 modes)* is defined in the same way as before, except that

- the label of each node in a search tree is of the form $(G, \mu\mathcal{M})$ and a call-exit marker is of the form $[B, \nu\mathcal{N}, \eta]$, and
- each key and each solution are of the form $A\mu\mathcal{M}$.

The *OLDT resolution for mode inference (with 4 modes)* is defined in the same way as before, except that the operations at steps (A), (B) and (C) are modified as follows:

- let $\nu_0\mathcal{N}_0$ be "$\mu\mathcal{M} \xrightarrow{\eta} <>1$" ;
- let $\nu_0\mathcal{N}_0$ be "$\mu\mathcal{M} \vee \lambda\mathcal{L}$" ;
- let $\nu_{i+1}\mathcal{N}_{i+1}$ be "$\mu_{i+1}\mathcal{M}_{i+1} \xleftarrow{\eta_{i+1}} \nu_i\mathcal{N}_i$" ;

Then, the *initial OLDT structure*, *extension of an OLDT structure* and *OLDT resolution* are defined in the same way as those with 3 modes.

8. Discussion

Although it had been known that seemingly different *ad hoc* methods for analyzing properties of programs can be accommodated into a single framework called *abstract interpretation* by Cousot and Cousot [7],[8], their framework was a little complicated due to the semantics of the conventional imperative programming languages on which their framework focused its attention. The research on the abstract interpretation of logic programs started at the beginning of the 1980's by several researchers taking advantages of the simple computation mechanism of logic programs [22],[14],[16],[6],[19]. We will discuss the various approaches according to two dimensions, *interpretation method* and *target property*.

(1) Interpretation Method

The first dimension of the classification is what type of interpretation the abstract interpreter is based on. As was mentioned in Section 1, some operation which is *bottom-up in nature* is inevitable, although the program properties when the top-down interpreter is employed are to be analyzed. According to how the bottom-up operation is integrated, the frameworks of the abstract interpretation are classified as follows:

The *pure bottom-up abstract interpretation* approach is based on the bottom-up interpreter, i.e., hyper-resolution. This approach was applied to type inference by Kanamori and Horiuchi [15], and generalized by Marriott and Søndergard [20].

The *hybrid abstract interpretation* approach is based on both the top-down interpreter and the bottom-up interpreter. Depending on how these two interpreters are combined, the approach is divided into the two-phase hybrid abstract interpretation and the one-phase hybrid abstract interpretation.

The *two-phase abstract hybrid interpretation* was proposed by Mellish [22] in order to give a theoretical foundation to his practical techniques for analyzing determinacy, modes and shared structures [21]. His approach derives simultaneous recurrence equations for the sets of goals at calling time and exiting time during the top-down execution of a given top-level goal, and obtains a superset of the least solution of the simultaneous recurrence equations using a bottom-up approximation. The reason of the separation into two phases, simulating the top-down execution and solving by the bottom-up approximation, is two-fold. One is that, by simulating the top-down execution, we can focus on just the goals relevant to the top-level goal. The other is that, by solving by the bottom-up approximation, we can obtain solutions without diving into infinite loop.

The *one-phase abstract hybrid interpretation* is the one we have presented in this paper (cf. [12],[30]). The approach differs from the two-phase approach in that it starts with the standard hybrid interpreter from the beginning. The standard hybrid interpreter can compute solutions of a given top-level goal without either diving into infinite loop (unlike the usual top-down interpretation) or wasting time for goals irrelevant to the top-level goal (unlike the usual bottom-up interpretation), so that the corresponding abstract hybrid interpreter achieves the same effects as Mellish's approach without the separation into two phases. (As was shown in Section 1, the behavior of our abstract interpreter is very close to the way human programmers usually analyze the properties in their mind by approximately simulating the behavior of goals, so that it has a similar flavor to the "qualitative reasoning" in artificial intelligence. In fact, the example of Figure 1.1 has been used as an introductory explanation of the qualitative reasoning.)

Similar approaches have been proposed indpendently by several researchers. To introduce the operation bottom-up in nature, Bruynooghe [4],[5],[6] employed *abstract AND-OR graphs*, Mannila and Ukkonen [19] generalized the techniques of the data flow analysis of the conventional programs, and Debray and Warren [9],[10] utilized *extension table* in database query processing.

(2) Target Property

The second dimension of the classification is what properties of Prolog programs are analyzed. The properties range from rather simple ones e.g., type inference (or type synthesis, type derivation etc.), depth-abstracted term inference (or success pattern enumeration etc.), mode inference (or mode declaration derivation etc.) and sharing inference (or possible shared structure detection, compile time garbage collection, dependency calculation etc.), to a little complicated ones, e.g., functionality (or determinacy) detection, termination detection and computational complexity analysis. Although the exact analysis of these properties are undecidable in general, even the safe analysis is useful for manipulating programs (cf. [29]).

The *type inference* for Prolog programs was investigated by Kanamori and Horiuchi [15] for the bottom-up approach, by Kanamori and Kawamura [16], by Bruynooghe, Janssens, Callebaut and Demoen [4],[5] for the one-phase hybrid approach, and by others [1],[23].

The *depth-abstracted term inference* for Prolog programs was first studied by Sato and Tamaki [24]. The connection with abstract interpretation was pointed out by Kanamori and Kawamura [16] for one-phase hybrid approach.

The *mode inference* for Prolog programs was investigated by Mellish for the two-phase hybrid approach, by Debray and Warren [10], (with some safety condition for the case with 4 modes), Kanamori and Kawamura [16], Mannila and Ukkonen [19], and Bruynooghe and Janssens [5] for the one-phase approach.

The *sharing inference* was investigated by Mellish [21], Jones and Søndergaard [14], Bruynooghe, Janssens, Callebaut and Demoen [4], Mannila and Ukkonen [19], and utilized for safe mode inference with 4 modes by Kanamori and Kawamura [16].

The *functionality detection* is the problem of detecting the possibility of returning two different solutions. The functionality detection based on abstract interpretation was investigated by Debray and Warren [9] and Kanamori, Kawamura and Horiuchi [17].

A mode-abstracted atom $A\mu$ is said to be *functional*, if, when any goal $A\sigma$ in $A\mu$ succeeds with its solution $A\tau$, 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. To detect the functionality for a given mode-abstracted atom, we need to simultaneously count the number of solutions somehow during the mode inference (with 3 modes) in Section 6. 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

for which three operations $+$, $\times$ and *max* are defined. During the mode inference, we will compute some expressions using the solution number of each mode-abstracted atom, $\times, +$ and *max*, which overestimate the numbers of each solution in the solution table.

The *determinacy detection* is the problem of detecting the possibility of backtracking. The determinacy detection based on abstract interpretation was investigated by Mellish [21]. The recent approach by Sawamura [25] is close to the abstract interpretation approach.

The *termination detection* is the problem of detecting the possibility of infinite computation. The termination detection for Prolog programs was investigated by Frances, Grumberg, Katz and Pnueli [11], Shapiro [26], Ullman and Van Gelder [28] and by Baudinet [2]. The termination detection based on abstract hybrid interpretation was investigated by Kanamori, Horiuchi and Kawamura [18]. (Cf. Boyer and Moore [3] for Lisp programs.)

Because variables in Prolog programs are freely instantiatable, we need to define the termination property of Prolog programs with the mode (or some other) information. An atom A is said to be *terminating* when there is no infinite execution path in any OLD tree of A, i.e., it succeeds or fails finitely. A mode-abstracted atom $A\mu$ is said to be *terminating* when any atom in the mode-abstracted atom is terminating.

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

- m is a mapping from the set of atoms to a well-founded set $(W, \prec)$.
- For any atom B and any substitution θ, $m(B\theta) \preceq m(B)$.
- Suppose that there is a path in the OLD tree of A starting from a node with its leftmost atom $p(t_1, t_2, \dots, t_n)$ and ending with a node with its leftmost atom $p(s_1, s_2, \dots, s_n)$. Let $\theta_1, \theta_2, \dots, \theta_i$ be the labels of the edges on the path, and θ the composed substitution $\theta_1\theta_2\cdots\theta_i$. Then $m(p(s_1, s_2, \dots, s_n)) \prec m(p(t_1, t_2, \dots, t_n)\theta)$.

An atom A is said to be *terminating by M* when M is a set of all A's measures. In particular, when an atom A is terminating by $\{\}$, it is said to be *diverging*. Then, it is not difficult to prove that an atom A is terminating if and only if there exists a measure of A.

Hence, existence of measure is a necessary and sufficient condition for guaranteeing termination. Termination of a given mode-abstracted atom can be detected by finding a measure in some class for each mode such that it always decreases in recursions.

9. Conclusions

We have presented a unified framework for logic program analysis and its applications to type inference, depth-abstracted term inference and mode inference. Functionality detection and termination detection can be done as well based on our framework by enriching the abstract domain appropriately. This approach was implemented in our system for analysis of Prolog programs "Argus/A" from April 1986 to March 1988 [16],[17],[18].

Acknowledgements

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

References

[1] Bansal, A.K. and Sterling, L., An Abstract Interpretation Scheme for Logic Programs based on Type Expression, *Proc. of Fifth Generation Computer Systems 1988* :422–429, Tokyo, November 1988.

[2] Baudinet, M., Proving Termination Properties of Prolog Programs : A Semantic Approach, *Proc. of the 3rd Annual Symposium on Logic in Computer Science* :336–347, Edinburgh, July 1988. Also Technical Report STAN-CS-1202, Computer Science Department, Stanford University, April 1988.

[3] Boyer,R. and Moore, J.S., *A Computational Logic*, Academic Press, New York, 1978.

[4] Bruynooghe, M., Janssens.G., Callebaut,A. and Demoen,B., Abstract Interpretation : Towards the Global Optimization of Prolog Programs, *Proc. of 1987 Symposium on Logic Programming* :192–204, San Francisco, August 1987.

[5] Bruynooghe, M. and Janssens.G., An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, *Proc. of Fifth International Conference and Symposium on Logic Programming* :669–683, Seattle, August 1988.

[6] Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, to appear *the Journal of Logic Programming*, 1989.

[7] Cousot,P. and Cousot,R., 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* :238–252, Los Angeles, 1977.

[8] Cousot,P. and Cousot,R., Static Determination of Dynamic Properties of Recursive Procedures, in: E.J.Neuhold (ed.), *Formal Description of Programming Concepts* :237–277, North-Holland, Amsterdam, 1978.

[9] Debray,S.K. and Warren,D.S., Detection and Optimization of Functional Computation in Prolog, *Proc. of 3rd International Conference on Logic Programming* :490–504, London, July 1986.

[10] Debray,S.K. and Warren,D.S., Automatic Mode Inference for Prolog Programs, *Proc. of 1986 Symposium on Logic Programming* :78–88, Salt Lake City, September 1986.

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

[12] Gallagher, J. and Codish,M., Specialization of Prolog and FCP Programs Using Abstract Interpretation, *Proc. of Workshop on Partial Evaluation and Mixed Computation* :125–134, October 1987.

[13] Horiuchi,K. and Kanamori,T., Polymorphic Type Inference in Prolog by Abstract Interpretation, *Proc. of Logic Programming Conference* :107–116, Tokyo, June 1987.

[14] Jones,N.D. and Søndergaard,H., A Semantics-Based Framework for the Abstract Interpretation of Prolog Programs, in: S.Abramski and C.Hankin (eds.), *Abstract Interpretation of Declarative Languages* :123–142, Ellis Horwood, 1987.

[15] Kanamori,T. and Horiuchi,K., Type Inference in Prolog and its Application, *Proc. of 9th International Joint Conference on Artificial Intelligence* :704–707, Los Angeles, August 1985.

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

[17] Kanamori,T., Kawamura,T. and Horiuchi,K., Detecting Functionality of Logic Programs based on Abstract Hybrid Interpretation, ICOT Technical Report TR-331, Tokyo, December 1987.

[18] Kanamori,T., Horiuchi,K. and Kawamura,T., Detecting Termination of Logic Programs based on Abstract Hybrid Interpretation, ICOT Technical Report TR-398, Tokyo, December 1987.

[19] Mannila,H. and Ukkonen,E., Flow Analysis of Prolog Programs, *Proc. of 1987 Symposium on Logic Programming* :205–214, San Francisco, August 1987.

[20] Marriott, K. and Søndergaard,H., Bottom-up Abstract Interpretation of Logic Programs, *Proc. of Fifth International Conference and Symposium on Logic Programming* :733–748, Seattle, August 1988.

[21] Mellish,C.S., Some Global Optimizations for A Prolog Compiler, *J. Logic Programming* 2, 1 :43–66 (1985).

[22] Mellish,C.S., Abstract Interpretation of Prolog Programs, *Proc. of 3rd International Conference on Logic Programming* :463–474, London, July 1986.

[23] Mishra, P., Towards a Theory of Types in Prolog, *Proc. of 1984 Symposium on Logic Programming* :289–298, 1984.

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

[25] Sawamura, H., Determinacy Revisited, Unpublished Memo, 1988.

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

[27] Tamaki,H. and Sato,T., OLD Resolution with Tabulation, *Proc. of 3rd International Conference on Logic Programming* :84–98, London, July 1986.

[28] Ullman, J.D. and Van Gelder,A., Efficient Tests for Top-down Termination of Logical Rules, *J. of ACM*. 35, 2 :345–373, April 1988.

[29] Warren, R., Hermenegildo,M. and Debray,S., On the Practicality of Global Flow Analysis of Logic Programs, *Proc. of Fifth International Conference and Symposium on Logic Programming* :684–699, Seattle, August 1988.

[30] Wærn, A., An Implementation Technique for the Abstract Interpretation of Prolog, *Proc. of Fifth International Conference and Symposium on Logic Programming*, :700–710, Seattle, August 1988.