TR-619

Abstract Interpretation Based on OLDT Resolution

by
T. Kanamori & T. Kawamura (Mitsubishi)

February, 1991

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Abstract Interpretation Based on OLDT Resolution

Tadashi KANAMORI† Tadashi KAWAMURA‡

†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 OLDT resolution, a top-down Prolog interpreter with memo-ization. A run-time property of Prolog goals can be analyzed by executing the goals using an interpreter that is obtained by abstracting OLDT resolution according to the property. Due to the character of OLDT resolution, the execution neither enters a non-terminating computation loop nor wastes time working on goals irrelevant to the given top-level goals. In addition, the behavior of the abstract interpreter is very close to the way human programmers usually analyze the property in their mind. The correctness and termination of the abstract interpreter are discussed as well.

Keywords : Program Analysis, Abstract Interpretation, Prolog.

Contents

1 Introduction
2 OLDT Resolution
  2.1 An Example of OLDT Resolution
  2.2 A Formalization of OLDT Resolution
  2.3 Correctness of OLDT Resolution
3 Abastrct OLDT Resolution for Type Inference
  3.1 An Example of Type Inference
  3.2 A Formalization of the Type Inference
  3.3 Correctness of the Type Inference
4 Implementation of the Abstract OLDT Resolution
  4.1 A Formalization of A Modified Type Inference
  4.2 An Example of the Modified Type Inference
  4.3 Correctness of the Modified Type Inference
5 Discussion
6 Conclusions
Acknowledgements
References
Appendix Proof of the Correctness

1 Introduction

(1) What Is Abstract Interpretation?

Automatic analysis of the run-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 [11]. The information of the form of Prolog goals appearing in their successful execution enables us to eliminate unnecessary backtracking from the Prolog execution [20]. The information of modes provides the Prolog compiler with a chance of generating optimized codes [18]. Besides these properties, the functionality and the termination properties are of special importance [13,14].

But, why can we analyze such run-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 [4,5,10]. The framework of the abstract interpretation approach can be depicted schematically as below:

Figure 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 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 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 above example is trivial, it gives us some idea 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, to what data types of terms are variables L_0 and N_0 instantiated?" 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, to what data types of terms are $[X_1|L_1]$ and M_1 instantiated?"

3. Now, we first need to answer the question: "When the execution of *reverse*(L_1, N_1) succeeds, to what data type of term are L_1 and N_1 instantiated?" Because this question is identical to the original one, we would have to think forever if we repeated the same process. However, we usually proceed 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 such.

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, to what data types of terms are N_1, X_1 and M_1 instantiated?" 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, to what data types of terms are $[Y_3|N_3], X_3$ and $[Y_3|M_3]$ instantiated?"

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 in 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 in step 3, we can conclude that L_0 and N_0 are lists when the execution of *reverse*(L_0, N_0) succeeds.

Note that we needed to propagate the type information, though we have not emphasized it. For example, we needed to know in 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 in 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 analyze Prolog programs using the abstract interpretation approach? The figure below depicts the framework for type inference when the framework of Figure 1 is applied directly. Then, what interpretation should we employ for the abstract interpretation of Prolog programs?

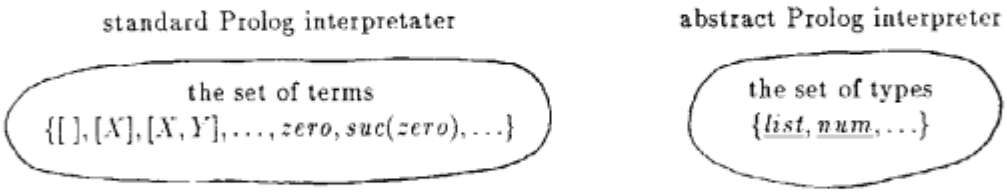


Figure 3: Framework for Type Inference by Abstract Interpretation

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 entered a non-terminating computation loop in the example just examined. For example, if we had not made the assumption in step 3, we could not have proceeded any further. (In general, due to the abstraction, the top-down execution in abstract domains is more likely to enter a non-terminating computation loop than the usual one in the domain of terms. Hence, making the assumptions in step 3 is crucial to answer the question at the beginning.)

The other Prolog interpreter, which is just as simple 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 $append(N, suc(K), suc(K))$ has not been considered in the example just examined.

Thus, the previous reflection on how we analyze Prolog programs in our mind has shown different behavior from both the top-down interpreter and the bottom-up interpreter. This suggests 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 OLDT resolution, a top-down Prolog interpreter with memo-ization. A run-time property of Prolog goals can be analyzed by executing the goals using an interpreter that is obtained by abstracting OLDT resolution according to the property. Due to the character of OLDT resolution, the execution neither enters a non-terminating computation loop nor wastes time working on goals irrelevant to the given top-level goals. In addition, the behavior of the abstract interpreter is very close to the way human programmers usually analyze the property in their mind. The correctness and termination of the abstract interpreter are discussed as well.

The rest of this paper is organized as follow: First, Section 2 introduces OLDT resolution. Next, Section 3 presents a type inference as an example of abstract interpretation. (Extracting a general framework and instantiating it to other abstract domains is immediate.) Then, Section 4 shows an implementation technique.

2 OLDT Resolution

In this section, we will first present an example of OLDT resolution [21], then formalize the notions of OLDT resolution, and last show the correspondence between OLDT resolution and the usual top-down interpretation.

2.1 An Example of OLDT Resolution

Let us first see an example of OLDT resolution. Consider the following "graph reachability" program by Tamaki and Sato [21].

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

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*" give the edges of the directed graph of Figure 4. The program is a typical *left recursive program* so that the usual top-down execution of a goal is likely to enter a non-terminating computation 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.

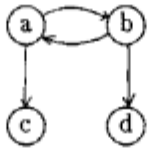


Figure 4: Graph Reachability Problem

OLDT resolution was devised by Tamaki and Sato [21] to avoid such a non-terminating computation loop. It manipulates

- a tree (representing OR search),
- 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 enable us to enjoy advantages of the both interpretations. Let us see how OLDT resolution returns solutions to the top-level goal "$reach(a, Z_0)$."

First, an initial tree consisting of 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 limited space, the goal and the substitution are arranged in two consecutive rows in the figure.) There is also generated an initial table containing a 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.

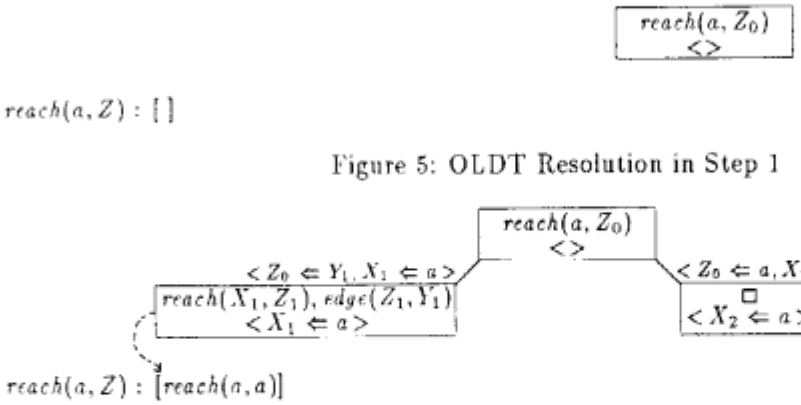


Figure 5: OLDT Resolution in Step 1

Figure 6: OLDT Resolution in Step 2

Secondly, the root node is expanded using the program to generate two child nodes in the same way as the usual 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 in step 1, 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.)

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 the 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 end of the solution list. Because $edge(a, Y_1)$, the leftmost

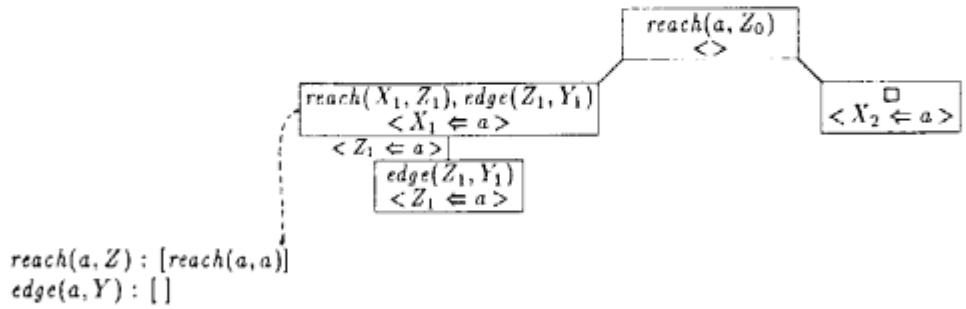


Figure 7: OLDT Resolution in Step 3

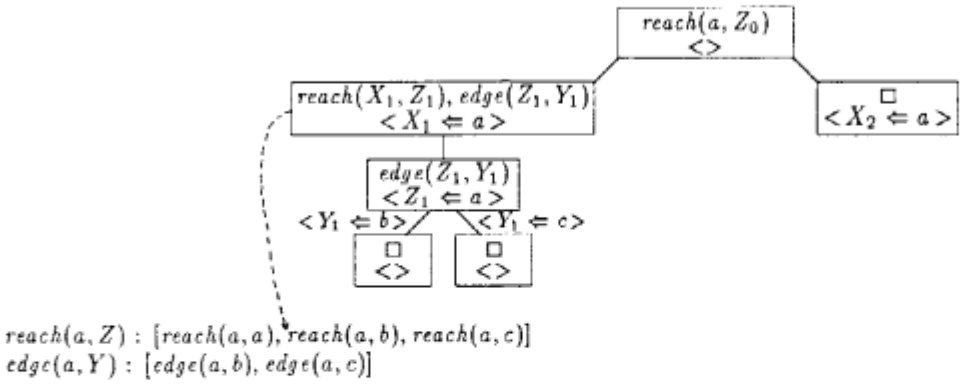


Figure 8: OLDT Resolution in Step 4

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.

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 end of the solution list of $edge(a, Y)$, and two solutions $reach(a, b)$ and $reach(a, c)$ to the end of the solution list of $reach(a, Z)$.

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 $reach(a, Z)$, therefore, the list pointed from the lookup node is not empty, that is, there exist solutions not yet utilized.

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 end of the solution list of $edge(a, Y)$, and one new solution, $reach(a, d)$, to the end of the solution list of $reach(a, Z)$.

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

OLDT resolution stops here, because no solution node is expansible and the list pointed

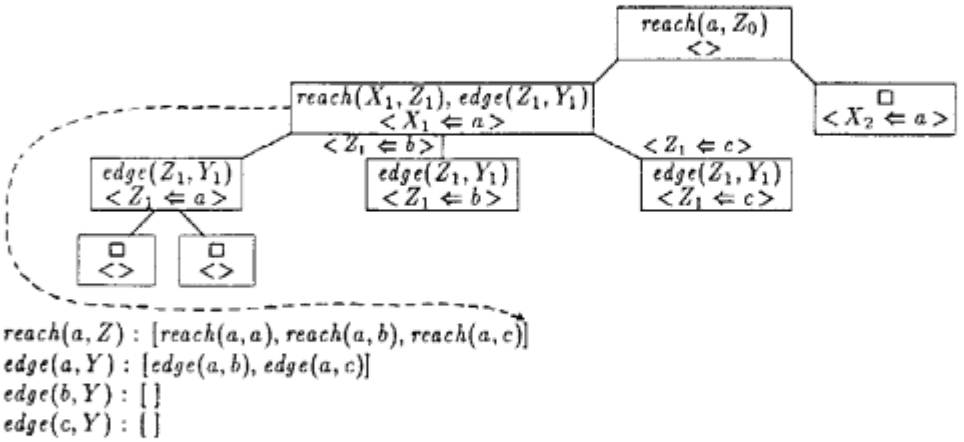


Figure 9: OLDT Resolution in Step 5

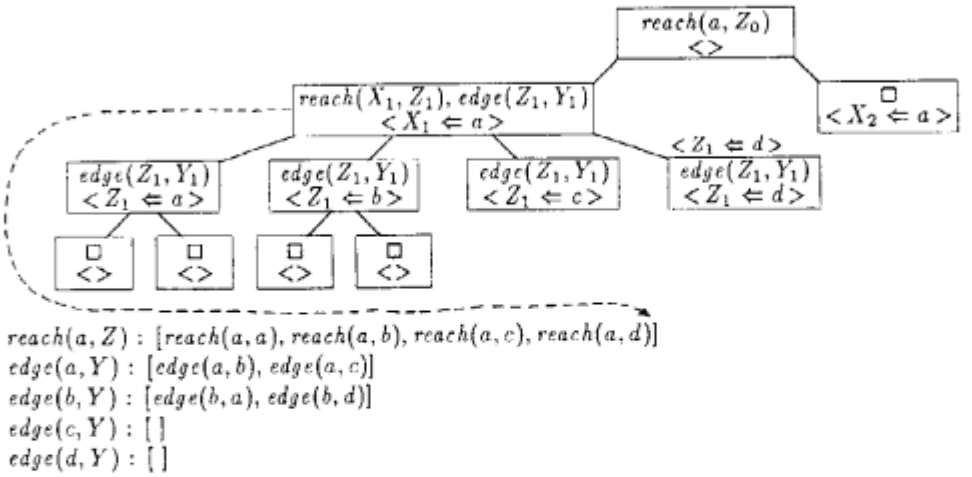


Figure 10: OLDT Resolution in Step 7

from the lookup node is empty.

2.2 A Formalization of OLDT Resolution

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 are no two assignments 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 $\sigma, \tau, \theta, \eta$. A *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 the variable X to itself when X is not in the domain variables of the substitution explicitly. Hence the empty substitution $<>$ assigns every variable to itself.

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. (When $A\sigma$ is considered, assignments in σ to the variables not 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; it is denoted by G, H. An empty goal, i.e., an empty sequence of atoms, is denoted by $\square$. $G\sigma$ 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 substitution η such that $A\sigma\eta$ and $B\tau\eta$ are identical. Then, η is called a *unifier*, and $A\sigma\eta$ and $B\tau\eta$ is called a *unification* of $A\sigma$ and $B\tau$. A unifier θ of $A\sigma$ and $B\tau$ is called a *most general unifier*, and $A\sigma\theta$ and $B\tau\theta$ is called a *most general unification* of $A\sigma$ and $B\tau$, when, for any unifier of $A\sigma$ and $B\tau$, say η, there exists a substitution ρ such that η is $\theta\rho$.

(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 $\square$. 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 no variants of this key appear (as keys) elsewhere 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), where Tr is a search tree of $G\sigma$, Tb is a solution table, and 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 leaf 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 possibly 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. (We assume that $A\sigma\theta$ is a fresh variant of the solution.) 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 case, 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* $\square\,\sigma$ with solution $\square\,\sigma$.

(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 $A_2\sigma\tau_1\tau_2$, ..., 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 node v of OLDT structure (Tr, Tb, As) is OLDT resolvable.

1. When v is a leaf 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, σ_k) to v.

2. 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 in the associated solution list with which 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. Replace the pointer from the OLDT resolved lookup node with the one connecting to the end of the associated solution list.

3. 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. If a new node is a lookup node, add a pointer from the new lookup node to the head of the solution list of the corresponding key. If 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.

4. 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 end 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 applications of immediate extensions.

Note that an immediate extension is applicable to any lookup node (so long as its associated list is non-empty), whereas it is applicable to only leaf solution nodes.

(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 OLDT Resolution

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

Theorem 1 Let P be a program and Q be an atom.

- If an atom $B\tau$ appears at the leftmost of a goal during OLD resolution of Q in P, then some extension of the initial OLDT structure of Q in P contains a node with leftmost atom $B\tau$ (Correctness of Calling Patterns).
- If an atom is solved with solution $A\tau$ during OLD resolution of Q in P, then some extension of the initial OLDT structure of Q in P contains a unit subrefutation with solution $A\tau$ (Correctness of Exiting Patterns).

Proof. See Appendix.

Though all solutions were found under the depth-first from-left-to-right extension strategy as exemplified in Section 2.1, this is not always the case, that is, some solutions 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 to the right of the lookup node might be inhibited forever. (However, when this OLDT resolution is applied to the abstract domain with finite elements, it always terminates under any strategy. See Section 3.3.)

3 Abstract OLDT Resolution for Type Inference

It is easy to see the similarity in behavior between the type inference in Section 1 and OLDT resolution in Section 2.

3.1 An Example of Type Inference

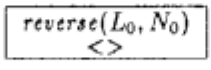
Let us first re-examine the type inference process in Section 1 using the notions similar to those in Section 2. Recall the "*reverse*" program in Section 1.

```
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).
```

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

First, an initial tree consisting of the root node labelled with a pair of "$reverse(L_0, N_0)$" and <> 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 a 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.

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



$reverse(L, N)$ <> : [ ]

Figure 11: Type Inference in Step 1

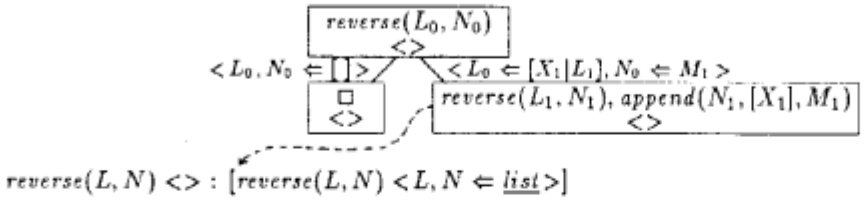


Figure 12: Type Inference in Step 2

m.g.u. When this node is generated, goal $reverse(L_0, N_0)$ has just been 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 in step 1, 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.

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 end 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.

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 solution $append(N, [X], M) < N, M \Leftarrow \underline{list} >$ to the end of the solution list of $append(N, [X], M) < N \Leftarrow \underline{list} >$. (The left node also adds a solution to the root atom, $reverse(L, N) < L, N \Leftarrow \underline{list} >$, which already appears in the solution list of $reverse(L, N) <>$; therefore, it is not added to the solution table.) The right node is a solution node so that a new entry is added to the solution table.

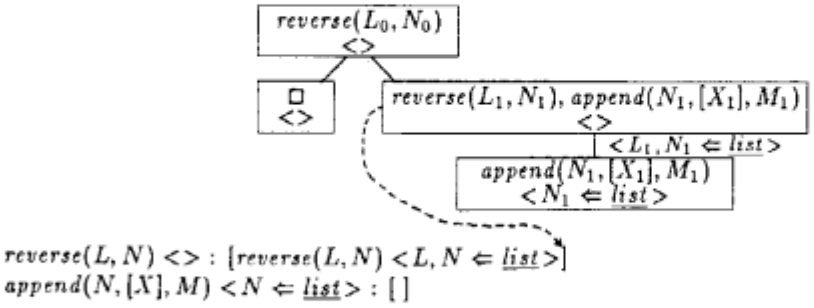


Figure 13: Type Inference in Step 3

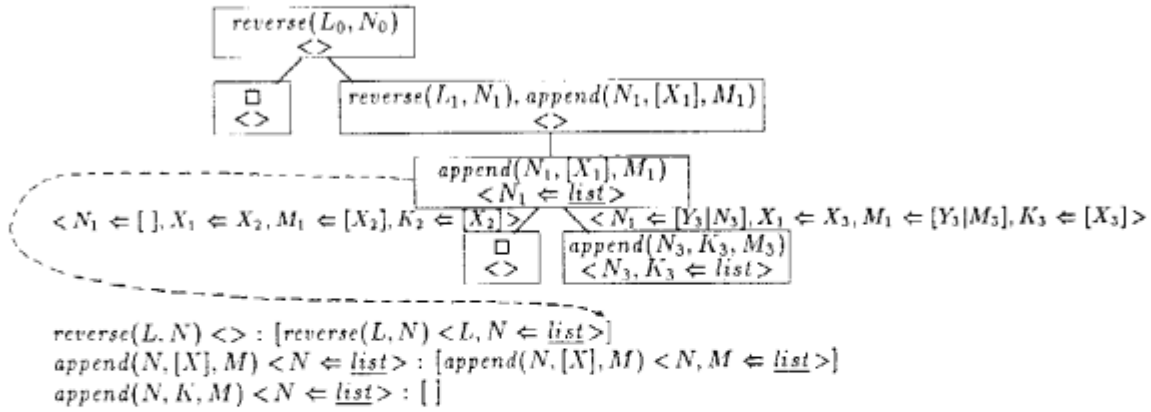


Figure 14: Type Inference in Step 4

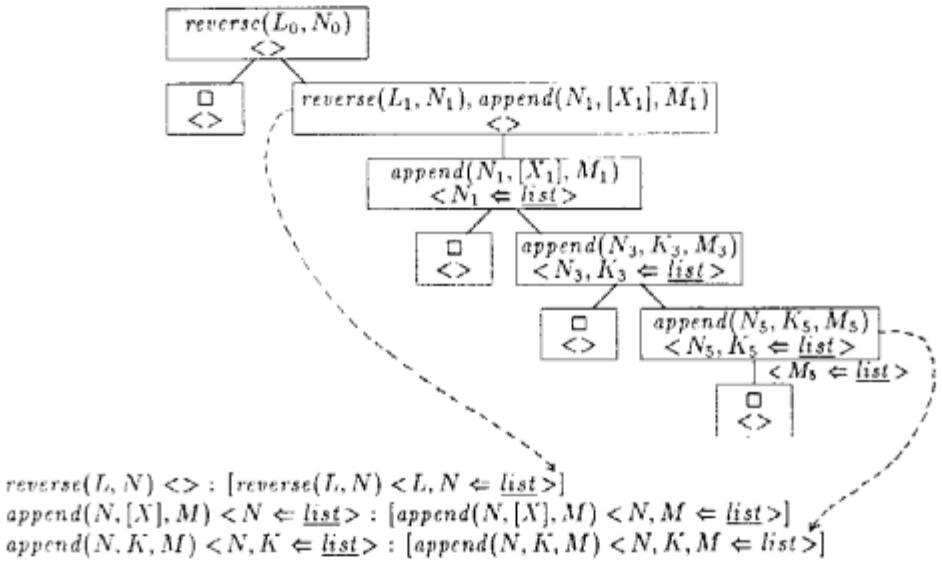


Figure 15: Type Inference in Step 6

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 end 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.

Lastly, the lookup node is expanded using the solution table to generate one child node labelled with pair $(\square, <>)$, 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 A Formalization of the Type Inference

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 $\underline{p_1}, \underline{p_2}, \ldots, \underline{p_k}$ 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 [9], 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,
$\emptyset$: the empty set.

The *instantiation ordering of types* is the ordering $\prec$ depicted on the left in Figure 16, while the *set-inclusion ordering of types* is the ordering $\subset$ depicted on the right:

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

Figure 16: Instantiation Ordering and Set Inclusion Ordering

- 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 abstract domains.)

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 are no two type assignments 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. A *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 domain variables are 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.

(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. (If two type-abstracted atoms are variants of each other, then they denote the same set of atoms, but not vice versa.)

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, *inward type propagation* and *outward 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 *inward 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 the term s itself.) It is computed as below:

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

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

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

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 *outward type propagation of* λ *to* s, denoted by s/λ. It is computed as below:

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

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 $A, X_1, X_2, \ldots, X_k, \mu, B, Y_1, Y_2, \ldots, Y_l$ 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 inward 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 inward 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 $\underline{any}$, 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 s'_j assigned to s_j using the outward type propagation of λ, hence, we can obtain a type substitution ν' of the form

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

   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 "$\mu \xrightarrow{\eta} \nu$" or "$\nu \xleftarrow{\eta} \mu$."

Note that $B(\mu \xrightarrow{\eta} \nu)$ is a superset of $A\mu \cap B\nu$, even if σ assigns two terms containing a common variable to two different variables in A. 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 $B(\mu \xrightarrow{\eta} \nu)$, 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 "$\mu \xrightarrow{\eta} \nu$," the final estimation $B(\mu \xrightarrow{\eta} \nu)$ is a superset of $A\mu \cap B\nu$, since $\underline{t_1} \supseteq \underline{t_2}$ if $t_1 \preceq t_2$.

(4) OLDT Structure for Type Inference

A *search tree*, a *solution table*, an *association* and an *OLDT structure* are defined in the same way as before except for the following points:

- The label of each node is a pair of a goal and a type substitution.
- Each edge from a lookup node is labelled with a type substitution.
- Keys and solutions in a solution table are type-abstracted atoms.

(5) 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 leaf 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. (We assume that $A(\mu \vee \lambda)$ is a fresh variant of the solution.) 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.*

(6) OLDT Refutation for Type Inference

An *OLDT subrefutation of* a type-abstracted atom and an *OLDT subrefutation of* a type-abstracted goal are defined in the same way as before except for the following points:

- A type-abstracted atom $A\mu$ is used instead of an atom $A\sigma$.
- A type-abstracted solution $A\nu$ is used instead of a usual solution $A\tau$.
- The join of type substitutions is used instead of the usual composition of substitutions.
- $\mu \xleftarrow{\eta} \nu$ is used instead of θ to specify the label of the node next to the solution node.

An *initial OLDT structure* and *extension of OLDT structure* are defined in the same way as before except that the edge from a parent node v to a child node is labelled with a type substitution of the OLDT resolution when v is a lookup node. An *OLDT refutation* is defined in the same way as before.

3.3 Correctness of the Type Inference

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 2 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 2 Let P be a program and $Q\lambda$ be a type-abstracted goal.

- If an atom $B\tau$ appears at the leftmost of a goal during OLD resolution of a goal in $Q\lambda$ using P, then some extension of the initial OLDT structure of $Q\lambda$ in P contains a node with head type-abstracted atom $B\nu$ such that $A\sigma$ is in $B\nu$ (Correctness of Calling Patterns).

- If an atom is solved with solution $A\tau$ during OLD resolution of a goal in $Q\lambda$ using P, then some extension of the initial OLDT structure of $Q\lambda$ in P contains a unit subrefutation with solution $A\nu$ such that $A\tau$ is in $A\nu$ (Correctness of Exiting Patterns).

Proof. See Appendix.

Note that any extension of the initial OLDT structure of (G, μ) in program P generates only a 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 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 a finite number of times, since the number of head type-abstracted atoms is finite. Therefore, the length of each label is bounded by

  $$|G| + \text{"the number of the extensions at solution nodes"} \times |C|_{max}$$

  where $|G|$ is the length of G and $|C|_{max}$ is the maximum length of the bodies of the clauses in P, 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 an infinite number of 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 above theorem 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 Abstract OLDT Resolution

The processing of unit subrefutations seems troublesome in OLDT resolution of Sections 2 and 3. To make the conceptual presentation of OLDT resolution 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 at all.

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

4.1 A Formalization of A Modified Type Inference

(1) Modified OLDT Structure for Type Inference

A *search tree*, a *solution table*, an *association* and an *OLDT structure* are defined in the same way as before except for the following points:

Input : a node u.
Output : a node.
Procedure : Let the label of u be of the form ("$A, \alpha_2, \ldots, \alpha_n$", μ).
Step 0 : When u is OLDT-resolved with "B :- $B_1, B_2, \ldots, B_m$" in P,

- let G_0 be a generalized goal "$B_1, B_2, \ldots, B_m, [\![A, \mu, \eta]\!], \alpha_2, \ldots, \alpha_n$," and
- let ν_0 be "$\mu \xrightarrow{\eta} <>$,"

where η is an m.g.u. of A and B. When u is OLDT-resolved with "$A(\mu \vee \lambda)$" in Tb,

- let G_0 be a generalized goal "$\alpha_2, \ldots, \alpha_n$," and
- let ν_0 be "$\mu \vee \lambda$."

Initialize i to 0.
Step 1 : If the leftmost of G_i is a call-exit marker $[\![A_{i+1}, \mu_{i+1}, \eta_{i+1}]\!]$,

- let G_{i+1} be G_i other than the leftmost call-exit marker,
- let ν_{i+1} be "$\mu_{i+1} \xleftarrow{\eta_{i+1}} \nu_i$," and
- add $A_{i+1}\nu_{i+1}$ to the end of $A_{i+1}\mu_{i+1}$'s solution list if it is not in it.

Increment i by 1. Repeat this step until the leftmost of G_i is not a call-exit marker.
Step 2 : Return a node labelled with (G_i, ν_i).

Figure 17: Modified OLDT Resolution for Type Inference

- The goal part of each node label is a *generalized goal*. A generalized goal is either $\Box$ or a sequence "$A, \alpha_2, \ldots, \alpha_n$," where α_i is either an atom or a *call-exit marker* of the form $[\![B, \nu, \eta]\!]$.
- The edges are not labelled with substitutions.

(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 leaf 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 OLDT resolvent is obtained through 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.

The precise algorithm is shown in Figure 17. The processing in the calling phase is performed in step 0, while that in the exiting phase is performed in step 1. Note that each node is labelled, say with (G, μ), in such a way that the following property holds: "the type

substitution part μ always shows the type information of atoms to the left of the leftmost call-exit marker in G." When there is a call-exit marker $[A_j, \mu_j, \eta_j]$ at the leftmost of goal part in the exiting phase, we need to update the substitution part by propagating the type substitution to μ_j through η_j in order that the above property still holds after eliminating the call-exit marker. The sequence $\nu_1, \nu_2, \ldots$ denotes the sequence of updated type substitutions. In addition, when we pass a call-exit marker $[A_j, \mu_j, \eta_j]$ in step 1 with substitution ν_j, the atom $A_j\nu_j$ denotes the solution of a unit subrefutation of $A_j\mu_j$. The solution $A_j\nu_j$ is added to the solution list of $A_j\mu_j$.

(3) Modified OLDT Refutation for Type Inference

An *initial OLDT structure* and an *extension of OLDT structure* are defined in the same way as before except that a new solution is added to the solution table at the OLDT resolution step above. An *OLDT refutation of* $G\mu$ 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. Let ν be the substitution part of the null node. Then the *solution of the refutation* is $G\nu$.

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

4.2 An Example of the Modified Type Inference

Let us show a different example. 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).
```

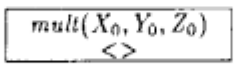
Then the type inference of $mult(X_0, Y_0, Z_0) <>$ proceeds as follows:

First, the initial OLDT structure is generated.

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.

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.

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.



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

Figure 18: Modified Type Inference in Step 1

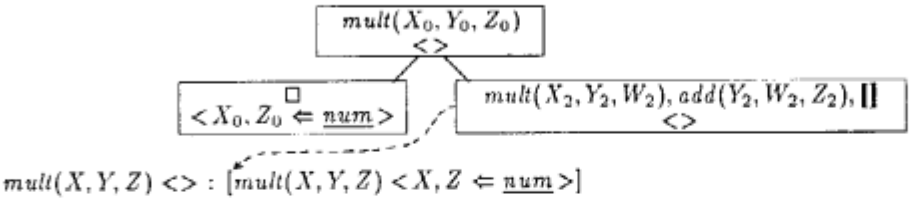


Figure 19: Modified Type Inference in Step 2

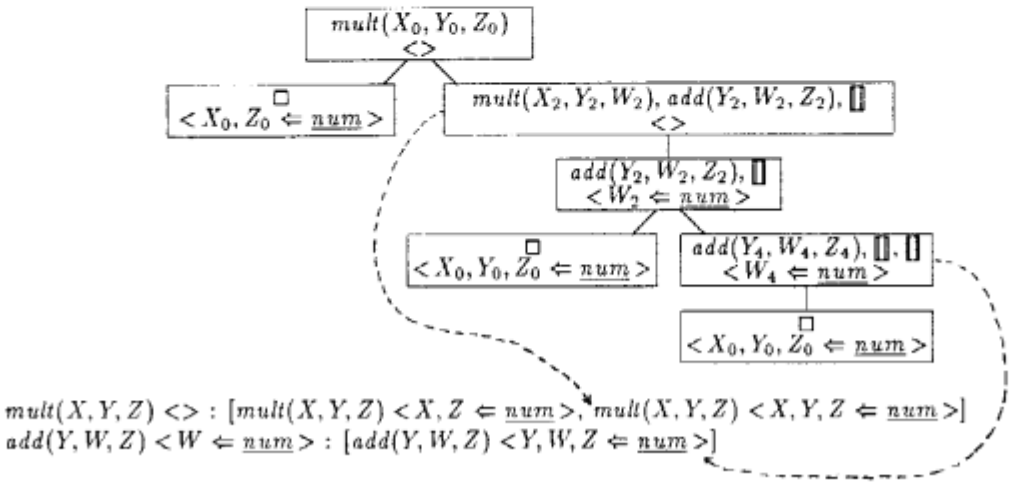


Figure 20: Modified Type Inference in Step 5

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 patterns 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.

4.3 Correctness of the Modified Type Inference

This modified type inference is a correct implementation of the type inference of Section 3, hence the same theorem as Theorem 2 holds.

Theorem 3 Let P be a program and $Q\lambda$ be a type-abstracted atom.

- If an atom $B\tau$ appears at the leftmost of a goal during OLD resolution of a goal in $Q\lambda$ using P, then some extension of the initial OLDT structure of $Q\lambda$ in P contains a node with head type-abstracted atom $B\nu$ such that $A\sigma$ is in $B\nu$ (Correctness of Calling Patterns).
- If an atom is solved with solution $A\tau$ during OLD resolution of a goal in $Q\lambda$ using P, then some extension of the initial OLDT structure of $Q\lambda$ in P contains a unit subrefutation with solution $A\nu$ such that $A\tau$ is in $A\nu$ (Correctness of Exiting Patterns).

Proof. To prove the theorem, it suffices to show that there exists an extension of an initial OLDT structure of Section 3 if and only if there exists an extension of an initial OLDT structure of Section 4 satisfying the following correspondence:

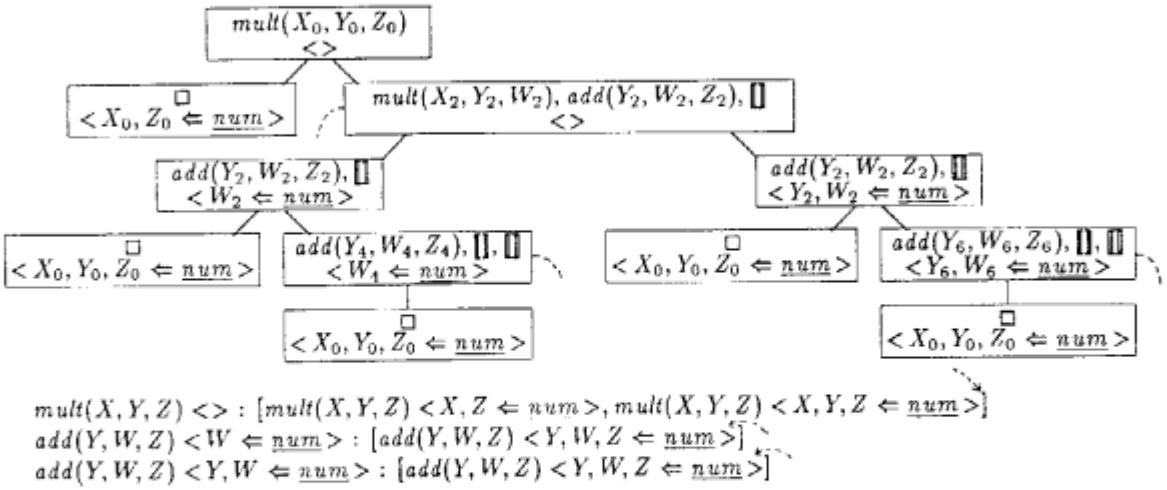


Figure 21: Modified Type Inference in Step 8

(a) The corresponding search trees have the identical form and satisfy 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 Discussion

In the abstract interpretation of Prolog programs, what we would like to analyze are the run-time properties of a given goal when it is executed using the usual top-down Prolog interpreter. As mentioned in Section 1, some operation which is *bottom-up in nature* is inevitable. 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 [11], and generalized by Marriott and Søndergard [17].

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 or the one-phase hybrid abstract interpretation.

The *two-phase abstract hybrid interpretation* was proposed by Mellish [19] to give a theoretical foundation to his practical techniques for analyzing determinacy, modes and shared structures [18]. 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 for 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 top-down execution, we can focus our attention on just the goals relevant to the top-level goal. The other is that, by solving by bottom-up approximation, we can obtain solutions without entering a non-terminating computation loop. (See Kanamori [15] for a justification of Mellish's approach.)

The *one-phase abstract hybrid interpretation* is the one we have presented in this paper. The approach differs from the two-phase approach in that it starts with OLDT resolution from the beginning. OLDT resolution can compute solutions of a given top-level goal without either entering a non-terminating computation loop (unlike the usual top-down interpretation) or wasting time working on 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.

Similar approaches have been proposed independently by several researchers. To introduce the operation bottom-up in nature, Bruynooghe [1,2,3] employed *abstract AND-OR graphs*, Mannila and Ukkonen [16] generalized the techniques of the data flow analysis of the conventional programs, and Debray and Warren [6,7] utilized *extension table* in database query processing. (Our idea of using OLDT resolution to explain abstract interpretation was propagated to [22] through [8].)

6 Conclusions

We have presented a unified framework for logic program analysis using a type inference problem as one of its examples. This approach was implemented in our system for analysis of Prolog programs "Argus/A" from April 1986 to March 1988 [12,13,14].

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 like to thank Mr. H. Seki, Ms. M. Ueno (Mitsubishi Electric Corporation) and Mr. K. Horiuchi (ICOT) for their valuable suggestions.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] Kanamori,T., Abstract Interpretation Based on Alexander Templates, ICOT Technical Report TR-549, Tokyo, March 1990.

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

[17] 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.

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

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

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

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

[22] 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.

Appendix Proof of the Correctness

The definition for *OLD tree* is the same as that for search tree of OLDT resolution, except that OLD trees contain only solution nodes. When new nodes are added, they are always solution nodes, hence the solution table and association pointers are not required. The definitions for *OLD resolution, OLD refutation* etc. carry over in the obvious way. A path in an OLD tree (or a search tree of OLDT structure) starting from a node labelled with ("A, G", σ) is called a *partial subrefutation of* $A\sigma$ when it does not contain any subrefutation of $A\sigma$ as its prefix. Using these notions, the correctness of OLDT resolution is stated as below:

Theorem 1 (Correctness of OLDT Resolution)
Let Q be a goal, T_0 be the initial OLD tree of Q, and $\mathcal{S}_0$ be the initial OLDT structure of Q.

(a) Some extension of T_0 contains a node with head atom $B\tau$, if and only if some extension of $\mathcal{S}_0$ contains a node with head atom $B\tau$. (Correctness for Calling Patterns)

(b) Some extension of T_0 contains a subrefutation with solution $A\tau$, if and only if some extension of $\mathcal{S}_0$ contains a subrefutation with solution $A\tau$. (Correctness for Exiting Patterns)

Proof. Although our standard hybrid interpretation is slightly different from the original OLDT resolution by Tamaki and Sato [19], these differences do not affect the theorem. The proof of the "if" part is by induction on the structure of OLDT structures. Due to space limit, we will omit it. The "only if" part is an immediate consequence of the following lemma:

Let T be an extension of an initial OLD tree, and let $\mathcal{S}$ be an extension of an initial OLDT strcture.

(a) If T contains a partial subrefutation of $A\sigma$ whose last node has head atom $B\tau$, and $\mathcal{S}$ contains a node with head atom $A\sigma$, then some extension of $\mathcal{S}$ contains a partial subrefutation of $A\sigma$ whose last node has head atom $B\tau$.

(b) If T contains a subrefutation of $A\sigma$ with solution $A\tau$, and $\mathcal{S}$ contains a node with head atom $A\sigma$, then some extension of $\mathcal{S}$ contains a subrefutation of $A\sigma$ with solution $A\tau$.

The proof of the lemma is almost the same as that of the lemma in Theorem 3.3. See the following proof of Theorem 3.3.

Similarly, using the notions about OLD resolutions, the correctness of the type inference is stated as below:

Theorem 2 (Correctness of the Type Inference)
Let $Q\lambda$ be a type-abstracted goal, T_0 be the initial OLD tree of a goal in $Q\lambda$, and $\mathcal{S}_0$ be the initial OLDT structure of $Q\lambda$.

(a) If some extension of T_0 contains a node with head atom $B\tau$, then some extension of $\mathcal{S}_0$ contains a node with head type-abstracted atom $B\nu$ such that $B\tau$ is in $B\nu$. (Correctness for Calling Patterns)

(b) If some extension of T_0 contains a subrefutation with solution $A\tau$, then some extension of $\mathcal{S}_0$ contains a subrefutation with solution $A\nu$ such that $A\tau$ is in $A\nu$. (Correctness for Exiting Patterns)

Proof. The theorem is an immediate consequence of the following lemma:

Let T be an extension of an initial OLD tree, and $\mathcal{S}$ be an extension of an initial OLDT structure.

(a) If T contains a partial subrefutation γ of $A\sigma$ whose last node has head atom $B\tau$, and $\mathcal{S}$ contains a node v with head type-abstracted atom $A\mu$ such that $A\sigma$ is in $A\mu$, then some extension of $\mathcal{S}$ contains a partial subrefutation of $A\mu$ whose last node has head type-abstracted atom $B\nu$ such that $B\tau$ is in $B\nu$.

(b) If T contains a subrefutation γ of $A\sigma$ with solution $A\tau$, and $\mathcal{S}$ contains a node v with head type-abstracted atom $A\mu$ such that $A\sigma$ is in $A\mu$, then some extension of $\mathcal{S}$ contains a subrefutation of $A\mu$ with solution $A\nu$ such that $A\tau$ is in $A\nu$.

The proof of the lemma 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 number of the nodes contained in the path γ ([19] pp.93–94).

Base Case : When $|\gamma| = 1$, the part(a) of the lemma is trivial, since $B\tau$ is $A\sigma$, hence $A\mu$ can be $B\nu$. The part (b) is vacantly true, since $|\gamma| > 1$ for any subrefutation.

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

Case 1 : When v is a solution node, let u' and γ' be the next 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 (partial) subrefutation γ. Then, the label of u' is ("$A_1, A_2, \ldots, A_n, \ldots$", σ'), the OLD resolvent of u and C. From the assumption, the v is also OLDT resolvable with C, and the OLDT resolvent ("$A_1, A_2, \ldots, A_n, \ldots$",$\mu'$) is such that $(A_1, A_2, \ldots, A_n)\sigma'$ is in $(A_1, A_2, \ldots, A_n)\mu'$ due to the property of "$\xrightarrow{\eta}$." Extending $\mathcal{S}$ (if necessary) by the OLDT resolution for type inference at the node v, we can get an OLDT structure $\mathcal{S}'$ in which v has a child node v' labelled with ("$A_1, A_2, \ldots, A_n, \ldots$", μ'). As for the part (a), γ' is divided into

γ_1 : subrefutation of $A_1\sigma'$ with solution $A_1\sigma'\theta_1$,

γ_2 : subrefutation of $A_2\sigma'\theta_1$ with solution $A_2\sigma'\theta_1\theta_2$,

$\vdots$

γ_k : partial subrefutation of $A_k\sigma'\theta_1 \ldots \theta_{k-1}$ such that $|\gamma_k| < |\gamma|$.

From the induction hypothesis, we have successive extensions $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ of $\mathcal{S}'$ such that each $\mathcal{S}_i$ contains a path $\delta_1\delta_2 \ldots \delta_i$ as below:

δ_1 : subrefutation of $A_1\mu'$ with solution $A_1\nu_1$ such that $A_1\sigma'\theta_1$ is in $A_1\nu_1$,

δ_2 : subrefutation of $A_2\nu_1$ with solution $A_2\nu_2$ such that $A_2\sigma'\theta_1\theta_2$ is in $A_2\nu_2$,

$\vdots$

δ_k : a partial subrefutation of $A_k\mu'$
whose last node has head type-abstracted atom $B\nu$ such that $B\tau$ is in $B\nu$.

The path in $\mathcal{S}_k$ starting from v and followed by $\delta_1, \delta_2, \ldots, \delta_k$ constitutes the required partial subrefutation of $A\mu$. The part (b) is proved similarly using the property of "$\xleftarrow{\eta}$."

Case 2 : When v is a lookup node, there is a corresponding solution node v_0 labelled with ("$A_0, \ldots$", μ_0) such that $A\mu$ is a variant of $A_0\mu_0$. From the induction hypothesis for $(\gamma, \mathcal{S}, v_0)$, we have an extension $\mathcal{S}'$ of $\mathcal{S}$ such that $\mathcal{S}'$ contains a subrefutation of $A_0\mu_0$ with solution

$A\nu$ (starting from v_0) such that $A\tau$ is in $A\nu$. By the operation in step 3 of the definition of the OLDT structure extension, the solution list of $A\mu$ in $\mathcal{S}'$ includes the solution $A\nu$. Now consider the label ("$A,\ldots$", μ) and the solution $A\nu$. Since ("$A,\ldots$", σ) and unit clause $A\tau$ have an OLD resolvent, the label ("$A,\ldots$", μ) and $A\nu$ also have an OLDT resolvent. This means that $\mathcal{S}'$ can be extended (if necessary) to $\mathcal{S}''$ by the operation in step 1 in the definition of the OLDT structure extension. Then, $\mathcal{S}$ contains a subrefutation of $A\mu$ with solution $A\nu$ (starting from v) such that $A\tau$ is in $A\nu$.