ICOT Technical Report: TR-263

TR-263

Polymorphic Type Inference in Prolog by Abstract Interpretation

by
K. Horiuchi and T. Kanamori (Mitsubishi)

June, 1987

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

Polymorphic Type Inference in Prolog by Abstract Interpretation

Kenji HORIUCHI Tadashi KANAMORI

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

Abstract

A polymorphic type inference method for Prolog programs by abstract interpretation is presented. The method is an extension of our monomorphic type inference method, which is one of the examples of analyzing patterns of Prolog goals at calling time and exiting time by *abstract interpretation*. The framework is based on OLDT resolution by Tamaki and Sato, a hybrid of the top-down and bottom-up interpretations of Prolog programs. By abstracting the hybrid interpretation directly, we can compute approximately not only the type information at calling time and exiting time without infinite looping but also just the necessary and relevant information without waste. The monomorphic type inference method is extended for polymorphic types by introducing parameterized type definitions and generalizing operations for manipulating type information accordingly.

Keywords : Program Analysis, Type Inference, Prolog, Abstract Interpretation.

1. Introduction

Most programming languages for symbolic processing, like Lisp or Prolog, can work on a variety of objects without type checking. Though such flexibility is useful for programmers to construct programs with less burden, no type checking makes it more difficult to find program bugs. Inferring data types of arguments from program texts provides useful information not only to programmers on debugging but also to meta-processing systems, like a compiler.

In Prolog, types can be introduced by defining type predicates, e.g.,

```
list([ ]).
list([A|L]) :- list(L).
```

In general, when the set of all terms satisfying the definition of a type can be uniquely determined from its definition, the type is called a *monomorphic type*. For example, list is a monomorphic type, since the set of all terms satisfying the definition above, denoted by *list*, is uniquely determined. Similarly, the following is the definition of a monomorphic type *num*,

```
num(0).
num(suc(X)) :- num(X).
```

The expressive power of monomorphic type system, however, is neither flexible nor sufficient. Suppose that we would like to express a set of terms not only by *list* and *num* but also by lists of *num*, lists of *list* or lists of lists of *num* etc. If we would like to introduce a list of numbers as a type, we can define it as a new monomorphic type *numlist* by

```
numlist([ ]).
numlist([A|L]) :- num(A), numlist(L).
```

But such an overloading of type constructors [ ] and [|] causes the following problems.

- The sets *list* and *numlist* are not disjoint though they are two distinct types.
- The intersection of *list* and *numlist* is not clear from the syntax.
- Type definitions like this have less syntactical regularity.

The approach to extend monomorphic types to parametric ones, called *polymorphic type*, is investigated in [8],[10]. As for the above example, since the car part A of the list $[A|L]$ isn't mentioned at all in the definition of *list*, we can express the type *numlist* by parameterizing the type of the car part in *list* and instantiating it to *num*.

In this paper, a polymorphic type inference method for Prolog programs by abstract interpretation is presented. The method is an extension of our monomorphic type inference method [3], which is one of the examples of analyzing patterns of Prolog goals at calling time and exiting time by *abstract interpretation* [2],[7]. The framework is based on OLDT resolution by Tamaki and Sato [12], a hybrid of the top-down and bottom-up interpretations of Prolog programs. By abstracting the hybrid interpretation directly, we can compute approximately not only the type information at calling time and exiting time without infinite looping but also just the necessary and relevant information without waste. The monomorphic type inference method is extended for polymorphic types by introducing parameterized type definitions and generalizing operations for manipulating type information accordingly.

After presenting the hybrid interpretation of Prolog programs in Section 2, we will introduce a monomorphic type into Prolog and extend it to a polymorphic type in Section 3. Then we will show a polymorphic type inference method.

In the following, we assume familiarity with the basic terminology of first order logic such as term, atom (atomic formula), formula, substitution and most general unifier (m.g.u.) We follow the syntax of DEC-10 Prolog [11]. As syntactical variables, we use X, Y, Z for variables, s, t for terms and A, B for atoms, possibly with primes and subscripts. In addition we use σ, τ, μ, ν for substitutions.

2. Interpretation of Logic Programs

In this section, we will present a hybrid interpretation method of Prolog programs [12] by contrasting the well-known top-down method and bottom-up method. Then we will show an implementation of the hybrid interpretation method suitable for the basis of the abstract interpretation presented later.

2.1 Hybrid Interpretation of Logic Programs

(1) Search Tree

A *search tree* is a tree with its nodes labeled with negative or null clauses and with its edges labeled with substitutions. A *search tree of* negative clause G is a search tree whose root node is labeled with G. The relation between a node and its child nodes in a search tree is specified in various ways depending on various strategies of "resolution".

A *refutation* of negative clause G is a path in a search tree of G from the root to a node labeled with the null clause $\square$. Let $\theta_1, \theta_2, \ldots, \theta_k$ be the labels of the edges on the path. Then, the *answer substitution of the refutation* is the composed substitution $\tau = \theta_1\theta_2\cdots\theta_k$, and the *solution of the refutation* is $G\tau$.

Let $G_0, G_1, \ldots, G_k$ be a sequence of labels of the nodes and $\theta_1, \theta_2, \ldots, \theta_k$ be the labels of the edges on some path in a search tree. The path is called a *subrefutation* of H when $G_0, G_1, G_2, \ldots, G_{k-1}, G_k$ are of the form

"H, G", "$H_1, G\theta_1$", "$H_2, G\theta_1\theta_2$", ...,

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

respectively, where H, H_1, H_2, ..., H_{k-1}, G are sequences of atoms. (In particular, when H is an atom A, the path is called a *unit subrefutation* of A.) Then, the *answer substitution of the subrefutation* is the composed substitution $\tau = \theta_1\theta_2\cdots\theta_k$, and the *solution of the subrefutation* is $H\tau$.

(2) Solution Table

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

(3) Association

Let Tr be a search tree and Tb be a solution table. A node in Tr is a *lookup node* when the left most atom of the negative clause is a variant of some key in Tb, and it is a *solution node* otherwise. That is, all nodes in Tr labeled with non-null clauses are classified into either solution nodes or lookup nodes.

An *association* of Tr and Tb is a set of pointers pointing from each lookup node in Tr *into* some solution list in Tb such that the label of the lookup node and the key of the solution list are variants of each other.

(4) OLDT Structure

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

(a) Tr is a search tree of G. The relation between a node and its child nodes in a search tree is specified by the following *OLDT resolution*.

(b) Tb is a solution table. Each key in Tb is not necessarily of general form $p(X_1, X_2, \ldots, X_n)$.

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

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

(a) The node is a terminal solution node of Tr and there is some definite clause "B_0 :- $B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that A_1 and B_0 are unifiable, say by an m.g.u. θ. The negative clause (or possibly null clause) "$B_1\theta, B_2\theta, \ldots, B_m\theta, A_2\theta, \ldots, A_n\theta$" is called the *OLDT resolvent*.

(b) The node is a lookup node of Tr and there is some solution Br in the associated solution list of the lookup node such that Br is an instance of A_1, say by an instantiation θ. The negative clause (or possibly null clause) "$A_2\theta, \ldots, A_n\theta$" is called the *OLDT resolvent*.

The restriction of the substitution θ to the variables of A_1 is called the *substitution of the OLDT resolution*.

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

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

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

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

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

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

Example 2.1.1 For example, consider the following program of "graph reachability" by Tamaki and Sato [12].

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

Then, the hybrid interpretation generates the following OLDT structures of "$reach(a, Y_0)$". The underline denotes the lookup node, and the dotted line denotes the association from the lookup node.

First, the initial OLDT structure is generated. The root node of the search tree is a solution node.

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

$reach(a,Y)$. The generated right child node is the end of a unit subrefutation of $reach(a,Y_0)$. Its solution $reach(a,a)$ is added to the solution list of $reach(a,Y)$.

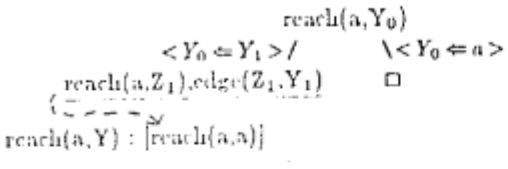


Figure 2.1.1 Step 2

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

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

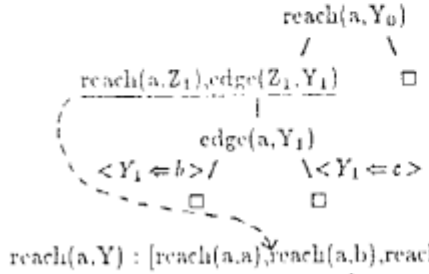


Figure 2.1.2 Step 4

Fifthly, the lookup node is OLDT resolved using the solution table, since new solutions were added to the solution list of $reach(a,Y)$ so that the associated solution list of the lookup node is not empty.

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

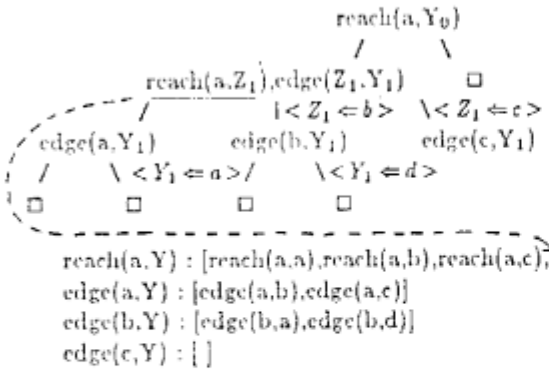


Figure 2.1.3 Step 6

Lastly, the lookup node is OLDT resolved once more using the solution table. Because the labels $edge(c,Y_1)$ and $edge(d,Y_1)$ of the solution nodes have no definite clause with unifiable heads, the extension process stops.

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

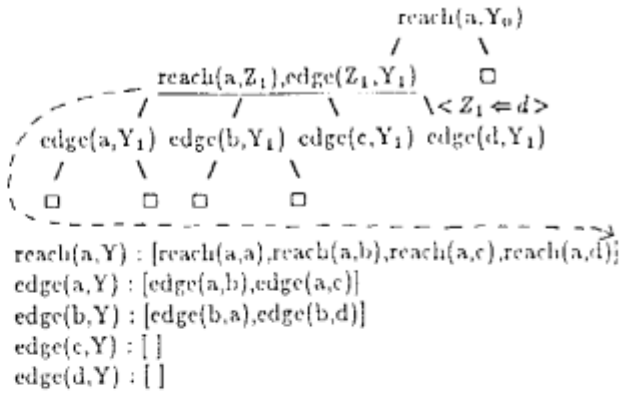


Figure 2.1.4 Last Step

lookup node might generate infinitely many child nodes so that extensions at other nodes right to the lookup node might be inhibited forever.

(5) Soundness and Completeness

Let G be a negative clause. An *OLDT refutation* of G in program P is a refutation in the search tree of some extension of OLDT structure of G. The *answer substitution of the OLDT refutation* and the *solution of the OLDT refutation* are defined in the same way as before. It is a basis of the abstract interpretation in this paper that OLDT resolution is still sound and complete.

Theorem (Soundness and Completeness)

If $G\tau$ is a solution of an OLDT refutation of G in P, its universal closure $\forall X_1 X_2 \cdots X_n\ G\tau$ is a logical consequence of P. If a universal closure $\forall Y_1 Y_2 \cdots Y_m\ G\sigma$ is a logical consequence of P, there is $G\tau$ which is a solution of an OLDT refutation of G in P and $G\sigma$ is an instance of $G\tau$.

Proof. Though our hybrid interpretation is different from the original OLDT resolution by Tamaki and Sato [12] in two respects (see [3]), these differences do not affect the proof of the soundness and the completeness. See Tamaki and Sato [12] pp.93-94.

2.2 An Implementation of the Hybrid Interpretation

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

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

(b) how we can obtain the solution of the unit subrefutation efficiently if any.

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

A *search tree* of OLDT structure in the modified framework is a tree with its nodes labeled with a pair of a negative clause and a substitution. (The edges are not labeled with substitutions.) A *search tree of* (G,σ) is a search tree whose root node is labeled with (G,σ). The clause part of each label

is a null clause $\square$, or a sequence "$\alpha_1, \alpha_2, \ldots, \alpha_n$" consisting of either atoms in the body of $P \cup \{G\}$ or *call-exit markers* of the form $[\![A, \sigma']\!]$. A *refutation of* (G, σ) is a path in a search tree of (G, σ) from the root to a node labeled with $(\square, \tau)$. The *answer substitution of the refutation* is the substitution τ, and the *solution of the refutation* is $G\tau$. A *solution table* and an *association* are defined in the same way as before.

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

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

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

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

The OLDT resolvent is obtained through the following two phases, *calling phase* and *exiting phase*. The case in which a node is OLDT resolved using solutions in the solution table is processed without generating call-exit markers in the calling phase. When there is a call-exit marker at the leftmost of the clause part in the exiting phase, it means that some unit subrefutation is obtained.

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

  a-1 When the node is OLDT resolved using a definite clause "$B_0 :- B_1, B_2, \ldots, B_m$" in program P and an m.g.u. θ, the intermediate clause part is "$B_1, B_2, \ldots, B_m, [\![\alpha_1, \sigma]\!], \alpha_2, \ldots, \alpha_n$", and the intermediate substitution part τ_0 is θ.

  a-2 When the node is OLDT resolved using a solution $B\tau$ in the solution table and an instantiation θ, the intermediate clause part is "$\alpha_2, \ldots, \alpha_n$", and the intermediate substitution part τ_0 is $\sigma\theta$.

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

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

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

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

Note that, when a node is labeled with (G, σ), the substitution part σ always shows the instantiation of atoms to the left of the leftmost call-exit marker in G. When there is a call-exit marker $[\![A_j, \sigma_j]\!]$ at the leftmost of clause part in the exiting phase, we need to update the substitution part by composing σ_j in order that the property above still holds after

```
OLDT-resolve(("α1, α2, ..., αn", σ) : label) : label ;
 i := 0;
 case
  when a solution node is OLDT resolved
        with "B0 :- B1, B2, ..., Bm" in P
    let θ be the m.g.u. of α1σ and B0 ;
    let G0 be a negative clause
     "B1, B2, ..., Bm, [α1, σ], α2, ..., αn";
    let τ0 be the substitution θ ;                    — (A)
  when a lookup node is OLDT resolved with "Bτ" in Tb
    let θ be the instantiation of α1σ to Bτ ;
    let G0 be a negative clause "α2, ... αn" ;
    τ0 be the composed substitution σθ ;              — (B)
 endcase
 while the leftmost of Gi is
       a call-exit marker [Ai+1, σi+1] do
   let Gi+1 be Gi other than the leftmost call-exit marker ;
   let τi+1 be σi+1τi ;                               — (C)
   add Ai+1τi+1 to the last of Ai+1σi+1's solution list
    if it is not in it ;
   i := i + 1;
 endwhile
 (Gnew, σnew) := (Gi, τi);
 return (Gnew, σnew).
```

Figure 2.2.1 An Implementation of OLDT Resolution

eliminating the call-exit marker. The sequence $\tau_1, \tau_2, \ldots, \tau_i$ denotes the sequence of updated substitutions. In addition, when we pass a call-exit marker $[\![A_j, \sigma_j]\!]$ in the **while** loop above with substitution τ_j, the atom $A_j\tau_j$ denotes the solution of the unit subrefutation of $A_j\sigma_j$. The solution $A_j\tau_j$ is added to the solution list of $A_j\sigma_j$.

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

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

Example 2.2.1 Consider the example in Section 2.2 again. The modified hybrid interpretation generates the following OLDT structures of $reach(a, Y_0)$.

First, the initial OLDT structure is generated, in which the root node is labeled with ("$reach(a, Y_0)$",<>).

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

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

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

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

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

$< X_1 \Leftarrow a >$.) During the elimination of the call-exit marker, $reach(a, a)$ is added to the solution table.

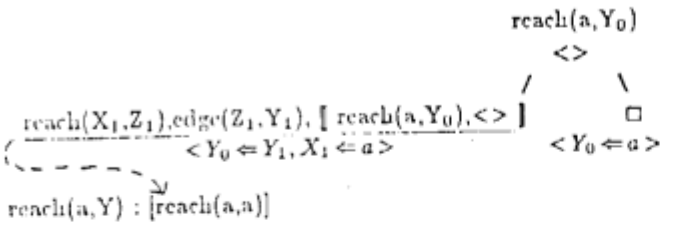


Figure 2.2.2 Step 2

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

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

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

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

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

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

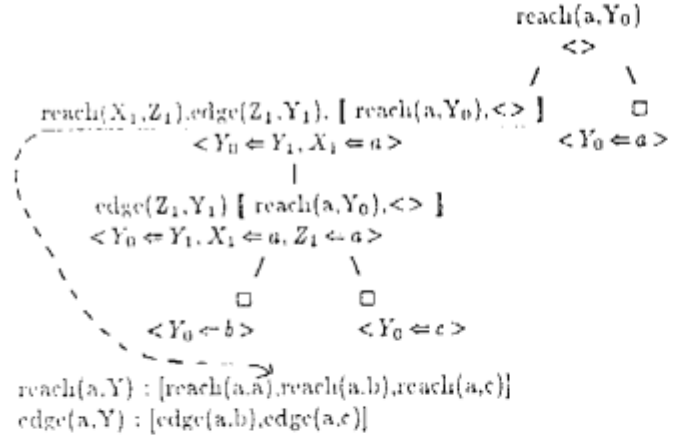


Figure 2.2.3 Step 4

The process of extension proceeds similarly to obtain all the solutions as in Example 2.1.1

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

3. Type Inference by Abstract Interpretation

Suppose that a goal "*map-plus(X,Y,Z)*" is executed with the second argument instantiated to a number and the others to any term, where *map-plus* and *plus* are defined by

```
map-plus([ ],M,[ ]).
map-plus([A|L],M,[C|N]) :-
        plus(A,M,C), map-plus(L,M,N).
plus(0,Y,0).
plus(suc(X),Y,suc(Z)) :- plus(X,Y,Z).
```

Then, the first and third arguments of *map-plus* are always instantiated to a list when the execution succeeds. But we would like to know more detailed type information if possible. In fact, these arguments are always instantiated to lists of numbers. How can we show it mechanically ?

In this section, we introduce a monomorphic type into Prolog and extend it to a polymorphic type, and show how a polymorphic type is inferred in the framework based on OLDT resolution in Section 2.

3.1 Polymorphic Type Inference
(1) Monomorphic Type

We introduce **type** construct into Prolog to separate definite clauses defining data structures from others defining procedures, e.g.,

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

type defines a unary relation by definite clauses. The head of definite clause takes a term defining a data structure as its argument, either a constant b called a *bottom element* or a term of the form $c(t_1, t_2, \ldots, t_n)$ where c is called a *constructor*. The body shows type conditions on proper subterms of the argument.

Here note that the set of terms is prescribed by type predicates. The set of all terms t such that the execution of $p(t)$ succeeds is called the *type* of p, and denoted by $\underline{p}$.

Example 3.1.1 Let the definition of a type *num* be

```
type.
  num(0).
  num(suc(X)) :- num(X).
end.
```

Then $\underline{num}$ is a set $\{0, suc(0), suc(suc(0)), \ldots\}$. Note that terms in each type is not necessarily ground, since the execution of $p(t)$ sometimes succeeds without instantiation of variables in t. For example, we include $[X]$ in *list*, since the execution of $list([X])$ succeeds without instantiation of the variable X.

Suppose there are k type predicates $p_1, p_2, \ldots, p_k$ defined using the **type** construct such that $\underline{p_1}, \underline{p_2}, \ldots, \underline{p_k}$ are disjoint. A *type* 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 type definition of p_1,
$\underline{p_2}$: the set of all terms satisfying the type definition of p_2,
:
$\underline{p_k}$: the set of all terms satisfying the type definition of p_k,
$\emptyset$: the empty set.

(2) Polymorphic Types

We extend the "**type**" construct in the above to a polymorphic one, e.g.,

```
type.
  list[α]([ ]).
  list[α]([A|L]) :- α(A), list[α](L).
end.
```

In the definition of *numlist* in Section 1, its body, i.e., num(A) and numlist(L), is its type condition. In the definition above, the *num* is parameterized to α and *numlist* is to *list*[α].

In general, we introduce polymorphic types by parameterizing the predicates of their type conditions (directly or indirectly) as follows:

$p[\alpha_1, \alpha_2, \ldots, \alpha_k](b_1).$
$\vdots$
$p[\alpha_1, \alpha_2, \ldots, \alpha_k](b_m).$
$p[\alpha_1, \alpha_2, \ldots, \alpha_k](c_1(X_{11}, X_{12}, \ldots, X_{1n_1})) :-$
$\quad p'_{11}(X_{11}), p'_{12}(X_{12}), \ldots, p'_{1n_1}(X_{1n_1}).$
$\vdots$
$p[\alpha_1, \alpha_2, \ldots, \alpha_k](c_h(X_{h1}, X_{h2}, \ldots, X_{hn_h})) :-$
$\quad p'_{h1}(X_{h1}), p'_{h2}(X_{h2}), \ldots, p'_{hn_h}(X_{hn_h}).$

where p is a new polymorphic type predicate, b_i is a new constant called *bottom element*, c_i is a new function symbol called *constructors*, p'_{ij} is either a type predicate of an arbitrary polymorphic (or monomorphic) type (including $\underline{any}$) or type parameter α_i.

A type definition obtained from $p[\alpha_1, \ldots, \alpha_k]$ by substituting type predicates $u_1, \ldots, u_k$ for parameters $\alpha_1, \ldots, \alpha_k$ is called an *instance of* $p[\alpha_1, \ldots, \alpha_k]$, and denoted by $p[u_1, \ldots, u_k]$. The corresponding set of terms is called a *type instance of* $p[\alpha_1, \ldots, \alpha_k]$, and denoted by $\underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_k}]$. The set of terms $\underline{p}[\underline{any}, \ldots, \underline{any}]$ is denoted simply by $\underline{p}$. The parameter α_i is called a *type parameter*, and the part enclosed with '[' and ']' is called the *parameter part*.

Example 3.1.2 Let *num* be defined as before. Then, we can express a list of numbers, *numlist* in Section 1, as a type instance of *list*[α], i.e., the set of all lists of *num* is denoted by $\underline{list}[num]$, where a polymorphic type definition of *list*[α] is

list[α]([ ]).
list[α]([A|L]) :- α(A), list[α](L).

list[*any*] means *list* of a monomorphic definition, and *num* can't be denoted by any other expressions. A list of lists of *num* is denoted by $\underline{list}[\underline{list}[num]]$.

Example 3.1.3 Suppose that *list* is defined in the same way as *Example 3.1.2*. A binary tree whose node is labeled with some term in *list* is defined as follows.

ltree[α](ϕ).
ltree[α](ltr(L,N,R)) :- ltree[α](L), list[α](N), ltree[α](R).

Then, a term in the type $\underline{ltree}[num]$ is a binary tree with its node labeled with a term in $\underline{list}[num]$.

In general, polymorphic types $\underline{p}[\ldots]$ and $\underline{p}[\ldots]$ are not necessarily disjoint, while $p_i[\ldots]$ and $p_j[\ldots]$ are disjoint whenever p_i and p_j are different type predicates. For example, $\underline{list}[num]$ and $\underline{list}[\underline{any}]$ are not disjoint, and $\underline{list}[num]$ and $\underline{list}[\underline{list}]$ are disjoint.

(3) Orderings of Polymorphic types

Suppose that there are k type predicates $p_1, p_2, \ldots, p_k$ such that $p_1, p_2, \ldots, p_k$ are disjoint. Then, a type is either $\underline{any}$, some type instance of $p_1[\mathcal{A}_1], p_2[\mathcal{A}_2], \ldots, p_k[\mathcal{A}_k]$ or $\emptyset$, where $\underline{any}$ is the set of all terms, $\mathcal{A}_i$ is a sequence of type parameters, and $\emptyset$ is the empty set.

Polymorphic types are ordered in two different ways. One is the instantiation ordering. A type instance $\underline{p}[\underline{u_1}, \ldots, \underline{u_m}]$ is said to be *smaller than* a type instance $\underline{q}[\underline{v_1}, \ldots, \underline{v_n}]$ *w.r.t. the instantiation ordering* iff any terms in $\underline{p}[\underline{u_1}, \ldots, \underline{u_m}]$ are instances of terms in $\underline{q}[\underline{v_1}, \ldots, \underline{v_n}]$, and denoted by $\underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \preceq \underline{q}[\underline{v_1}, \ldots, \underline{v_n}]$. The instantiation ordering between polymorphic types is defined as follows:

(a) $\underline{any} \preceq \underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_m}]$
(b) $\underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_m}] \preceq \emptyset$
(c) $\underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_m}]$ and $\underline{q}[\underline{v_1}, \underline{v_2}, \ldots, \underline{v_n}]$ are incomparable when p and q are different type predicates,
(d) when p and q are the same type predicate,
  (i) $\underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_m}] \preceq \underline{q}[\underline{v_1}, \underline{v_2}, \ldots, \underline{v_n}]$, when $\underline{u_j}$ and $\underline{v_j}$ are the same type or $\underline{u_j} \preceq \underline{v_j}$ for any $\underline{u_j}$ and $\underline{v_j}$,
  (ii) $\underline{q}[\underline{v_1}, \underline{v_2}, \ldots, \underline{v_n}] \preceq \underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_m}]$ when $\underline{u_j}$ and $\underline{v_j}$ are the same type or $\underline{v_j} \preceq \underline{u_j}$ for any $\underline{u_j}$ and $\underline{v_j}$,
  (iii) otherwise, $\underline{p}[\underline{u_1}, \underline{u_2}, \ldots, \underline{u_m}]$ and $\underline{q}[\underline{v_1}, \underline{v_2}, \ldots, \underline{v_n}]$ are incomparable.

Example 3.1.4 Let *num* and *list* be defined as before. Then,
$\underline{list} \preceq \underline{list}[num]$
$\underline{list}[\underline{list}] \preceq \underline{list}[\underline{list}[num]]$.
$\underline{list}[\underline{list}]$ and $\underline{list}[\underline{num}]$ are incomparable.

Example 3.1.5 Suppose that the type definition of *dtree* is

dtree[α_1, α_2](dlf(A,B)) :- α_1(A), α_2(B).
dtree[α_1, α_2](dtr(L,A,B,R)) :-
  dtree[α_1, α_2](L), α_1(A), α_2(B), dtree[α_1, α_2](R).

Then, $\underline{dtree}[\underline{list}, \underline{num}]$ and $\underline{dtree}[num, \underline{list}]$ are incomparable.

The other ordering is the set inclusion ordering. A type instance $\underline{p}[\underline{u_1}, \ldots, \underline{u_m}]$ is said to be *smaller than* a type instance $\underline{q}[\underline{v_1}, \ldots, \underline{v_n}]$ *w.r.t. the set inclusion ordering* iff $\underline{p}[\underline{u_1}, \ldots, \underline{u_m}]$ is included in $\underline{q}[\underline{v_1}, \ldots, \underline{v_n}]$, and denoted by $\underline{p}[u_1, \ldots, u_m] \subseteq \underline{q}[v_1, \ldots, v_n]$.

Example 3.1.6 Let *num* and *list* be defined as before. Then,
$\underline{list}[num] \subseteq \underline{list}$.
$\underline{list}[\underline{list}[num]] \subseteq \underline{list}[\underline{list}]$.
$\underline{list}[\underline{list}]$ and $\underline{list}[num]$ are disjoint.

(4) Join and Union of Polymorphic Types

We define the join and union operations of polymorphic types reformulating the monomorphic one. The join operation w.r.t. the instantiation ordering, denoted by $\vee$, is defined as follows:

$$\underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \vee \underline{q}[\underline{v_1}, \ldots, \underline{v_n}] =$$

$$\begin{cases} \underline{p}[\underline{u_1}, \ldots, \underline{u_m}], & \text{when } \underline{q}[\underline{v_1}, \ldots, \underline{v_n}] \text{ is } \underline{any}; \\ \underline{q}[\underline{v_1}, \ldots, \underline{v_n}], & \text{when } \underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \text{ is } \underline{any}; \\ \emptyset, & \text{when } p \text{ and } q \text{ are different, or} \\ & \text{either } \underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \text{ or} \\ & \underline{q}[\underline{v_1}, \ldots, \underline{v_n}] \text{ is } \emptyset; \\ \underline{p}[\ldots, \underline{u_i} \vee \underline{v_i}, \ldots], & \text{when } p \text{ and } q \text{ are identical.} \end{cases}$$

Similarly,the union operation w.r.t. the set inclusion ordering, denoted by $\cup$, is defined as follows:

$$\underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \cup \underline{q}[\underline{v_1}, \ldots, \underline{v_n}] =$$

$$\begin{cases} \underline{p}[\underline{u_1}, \ldots, \underline{u_m}], & \text{when } \underline{q}[\underline{v_1}, \ldots, \underline{v_n}] \text{ is } \emptyset; \\ \underline{q}[\underline{v_1}, \ldots, \underline{v_n}], & \text{when } \underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \text{ is } \emptyset; \\ \underline{any}, & \text{when } p \text{ and } q \text{ are different, or} \\ & \text{either } \underline{p}[\underline{u_1}, \ldots, \underline{u_m}] \text{ or} \\ & \underline{q}[\underline{v_1}, \ldots, \underline{v_n}] \text{ is } \underline{any}; \\ \underline{p}[\ldots, \underline{u_i} \cup \underline{v_i}, \ldots], & \text{when } p \text{ and } q \text{ are identical.} \end{cases}$$

Example 3.1.7 Let *num* and *list* be defined as before. Then,
$\underline{list}[num] \vee \underline{list}[\underline{any}] = \underline{list}[num]$
$\underline{list}[\underline{list}] \vee \underline{list}[num] = \emptyset$
$\underline{list}[\underline{list}] \cup \underline{list}[num] = \underline{list}[\underline{any}]$.

Example 3.1.8 Let $dtree[\alpha_1, \alpha_2]$ be defined as before.
$dtree[\underline{any}, num] \vee dtree[num, \underline{any}] = dtree[num, num]$,
$dtree[\underline{any}, num] \cup dtree[num, \underline{any}] = dtree[\underline{any}, \underline{any}]$.

(5) Type Inference

A *type substitution* μ is an expression of the form
$< X_1 \Leftarrow p_1, X_2 \Leftarrow p_2, \ldots, X_l \Leftarrow p_l >$,
where $p_1, p_2, \ldots, p_l$ are types. The type assigned to variable X by type substitution μ is denoted by $\mu(X)$. We stipulate 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 of the type substitution explicitly. Hence the empty type substitution $<>$ assigns $\underline{any}$ to every variable.

Let A be an atom in the body of some clause in $P \cup \{G\}$ and μ be a type substitution of the form
$< X_1 \Leftarrow p_1, X_2 \Leftarrow p_2, \ldots, X_l \Leftarrow p_l >$.
Then $A\mu$ 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 p_i. Two type-abstracted atoms $A\mu$ and $B\nu$ are said to be *unifiable* when $A\mu \cap B\nu \neq \emptyset$. A list of type-abstracted atoms $[A_1\mu_1, A_2\mu_2, \ldots, A_n\mu_n]$ denotes the set union $\cup_{i=1}^{n} A_i\mu_i$. Similarly, $G\mu$ is called a *type-abstracted negative clause*, and denotes the set of negative clauses obtained by replacing each X_i in G with a term in p_i.

The conventional top-down Prolog interpreter is modeled by OLD resolution. The relation between a node and its child nodes in OLD trees is specified by OLD resolution.

Let G be a negative clause of the form "$A_1, A_2, \ldots, A_n$" ($n \geq 1$), and C be a definite clause in program P of the form "$B_0 :- B_1, B_2, \ldots, B_m$" ($m \geq 0$). A node of OLD tree labeled with negative clause G is said to be *OLD resolvable* with C if A_1 and B_0 are unifiable, say by an m.g.u. θ. Without loss of generality, we assume that the m.g.u. θ substitutes a term consisting of fresh variables for every variable in A_1 and C. The negative clause (or possibly null clause) "$B_1\theta, B_2\theta, \ldots, B_m\theta, A_2\theta, \ldots, A_n\theta$" is called the *OLD resolvent* of G with C. The restriction of the substitution θ to the variables of A_1 is called the *substitution of the OLD resolution*.

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

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

Let $G\mu$ be a type-abstracted negative clause, $\mathcal{C}(G\mu)$ be the set of all calling patterns of negative clauses in $G\mu$ and $\mathcal{E}(G\mu)$ be the set of all exiting patterns of negative clauses in $G\mu$. The *type inference* w.r.t. $G\mu$ is the problem to compute
(a) some list of type-abstracted atoms which is a superset of $\mathcal{C}(G\mu)$,
(b) some list of type-abstracted atoms which is a superset of $\mathcal{E}(G\mu)$.

3.2. Type Inference by Hybrid Interpretation

3.2.1 OLDT Structure for Type Inference

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

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

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

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

3.2.2 Overestimation of Data Types

We need to specify the corresponding operations at step (A), (B) and (C) in Figure 2.2.1 for type inference. In order to specify these operations, we need to consider the following problems. Let A be an atom, $X_1, X_2, \ldots, X_k$ all the variables in A, μ a type substitution of the form
$< X_1 \Leftarrow p_1, X_2 \Leftarrow p_2, \ldots, X_k \Leftarrow p_k >$,
B an atom, $Y_1, Y_2, \ldots, Y_l$ all the variables in B, and ν a type substitution of the form
$< Y_1 \Leftarrow q_1, Y_2 \Leftarrow q_2, \ldots, Y_l \Leftarrow q_l >$.
Then
(a) How can we know whether there is an atom in $A\sigma = B\tau$ in $A\mu \cap B\nu$?
(b) If there is such an atom, what terms are expected to be assigned to $Y_1, Y_2, \ldots, Y_l$ by τ ?

Example 3.2.1 The following two type-abstracted atoms
$p(L_1, suc(N_1)) < L_1 \Leftarrow \underline{list}[\underline{num}],\ N_1 \Leftarrow \underline{num} >$,
$p([X_2|L_2], N_2) < X_2 \Leftarrow \underline{any},\ L_2 \Leftarrow \underline{any},\ N_2 \Leftarrow \underline{any} >$
are unifiable. Then, the common atom is of the form $p([X|L], suc(N))$, and terms in $\underline{num}$, $\underline{list}[num]$ and $\underline{num}$ must be assigned to variables X, L and N.

(1) Overestimation of Unifiability

When two type-abstracted atoms $A\mu$ and $B\nu$ are unifiable, two atoms A and B must be unifiable in the usual sense. Let η be an m.g.u. of A and B of the form
$< X_1 \Leftarrow t_1, X_2 \Leftarrow t_2, \ldots, X_k \Leftarrow t_k,$
$Y_1 \Leftarrow s_1, Y_2 \Leftarrow s_2, \ldots, Y_l \Leftarrow s_l >$.
If we can overestimate the type assigned to each occurrence of Z in t_i from the type substitution μ and that of Z in s_j from the type substitution ν, we can overestimate the type

assigned to the variable Z by taking the join $\vee$ w.r.t. the instantiation ordering for all occurrences of Z. If it is the emptyset $\emptyset$ for some variable, we can't expect that there exist an atom $A\sigma = B\tau$ in $A\mu \cap B\nu$.

When a term t containing an occurrence of variable Z is instantiated to a term in p, we compute a type set containing all instances of the occurrence of Z as follows and denote it by $Z/ < t \Leftarrow \underline{p} >$.

$$Z/ < t \Leftarrow \underline{p} > =$$

$$\begin{cases} \underline{p}, & \text{when } t \text{ is } Z; \\ \underline{any}, & \text{when } p \text{ is } any; \\ \underline{Z/ < t_i \Leftarrow p_i >}, & \text{when } t \text{ is of the form } c(t_1, \ldots, t_n), \\ & Z \text{ is in } t_i, \\ & p \text{ is a type } \underline{p}[\underline{u_1}, \ldots, \underline{u_m}], \\ & c \text{ is a constructor of } p[\alpha_1, \ldots, \alpha_m] \text{ and} \\ & p_i \text{ is a type assigned to } t_i; \\ \emptyset & \text{otherwise.} \end{cases}$$

Example 3.2.2 Let *num* and *list* be defined as before. Then,

$A/ < [A|L] \Leftarrow \underline{list}[\underline{num}] > = \underline{num}$,
$L/ < [A|L] \Leftarrow \underline{list}[\underline{num}] > = \underline{list}[num]$.

If we would like to check the unifiability of type-abstracted atoms $A\mu$ and $B\nu$ exactly, i.e., would like a procedure returning true *if and only if* they are unifiable, we can check it using the estimation $Z/ < t[Z] \Leftarrow \underline{p} >$. The exact unifiability check, however, takes more computational time because it can't be reduced to the unifiability of terms. But, if we would like just to overestimate the unifiability, i.e., would like a procedure returning true *if* they are unifiable, we may use the unifiability check of A and B instead of the more time-consuming one.

Example 3.2.3 We can check the unifiability of $p(X) < X \Leftarrow \underline{list} >$ and $p(suc(Y)) < Y \Leftarrow \underline{list} >$ by computing $Z/ < suc(Z) \Leftarrow \underline{list} >= \emptyset$ and $Z/ < Z \Leftarrow \underline{list} >= \underline{list}$, because an m.g.u. of $p(X)$ and $p(suc(Y))$ is $< X \Leftarrow suc(Z), Y \Leftarrow Z >$. If we use the unifiability check of $p(X)$ and $p(suc(Y))$ instead of the exact one, we would consider these type-abstracted atoms unifiable.

(2) One Way Propagation of Type Substitutions

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

As we have just shown, we can overestimate the type assigned to the variable Z appearing in the m.g.u. η of A and B due to the type substitution μ. By collecting these type assignments for all variables, we can overestimate the type substitution λ for the variables in $t_1, t_2, \ldots, t_k$ substituted by the m.g.u. η. Then, if we can overestimate the type assigned to s_j, from the type substitution λ obtained above, we can obtain the type substitution ν'

$$< Y_1 \Leftarrow \underline{q_1'}, Y_2 \Leftarrow \underline{q_2'}, \ldots, Y_l \Leftarrow \underline{q_l'} >$$

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

When each variable Z in term s is instantiated to a term in $\lambda(Z)$, we compute a type set containing all instances of s as follows and denote it by s/λ.

$$s/\lambda =$$

$$\begin{cases} \emptyset, & \text{when } \lambda(Z) \text{ is } \emptyset \text{ for some } Z \text{ in } s; \\ \lambda(Z), & \text{when } s \text{ is a variable } Z; \\ q[\emptyset, \ldots, \emptyset], & \text{when } s \text{ is a bottom element of} \\ & q[\alpha_1, \alpha_2, \ldots, \alpha_m]; \\ \underline{q}[\underline{u_1}, \ldots, \underline{u_m}] & \text{when } s \text{ is of the form } c(s_1, \ldots, s_n), \\ & c \text{ is a constructor of } q[\alpha_1, \ldots, \alpha_m] \text{ and} \\ & s_i/\lambda \subseteq \underline{q_i}, \text{ for any } s_i \ (1 \leq i \leq n), \\ & \text{where } \underline{q_i} \text{ is the type condition of} \\ & \text{the } i\text{-th argument of } c \text{ in } \underline{q}[\underline{any}, \ldots, \underline{any}], \\ & \underline{u_j} \text{ is a type obtained as below;} \\ \underline{any}, & \text{otherwise.} \end{cases}$$

Let $q_i'(1 \leq i \leq n)$ be the type condition of the i-th argument of c in $p[\alpha_1, \alpha_2, \ldots, \alpha_m]$. Then, for all occurrences of a parameter α_j in all q_i''s, $\underline{u_j}(1 \leq j \leq m)$ is a union w.r.t. the set inclusion ordering of the types. Each of these types is s_i/λ when an occurrence appears as a predicate of q_i', the corresponding type in a parameter part of s_i/λ when an occurrence appears in a parameter part of q_i'.

Example 3.2.4 Let *num* and *list* be defined as before. Then,

$[\,]/\lambda = \underline{list}[\emptyset]$, for any type substitution λ,
$[A|L]/ < A \Leftarrow \underline{num}, L \Leftarrow \underline{list}[\underline{num}] > = \underline{list}[num]$,
$[A|L]/ < A \Leftarrow \underline{num}, L \Leftarrow \underline{list}[\underline{any}] > = \underline{list}[\underline{any}]$.

Let λ be a type substitution

$$< A \Leftarrow \underline{num}, B \Leftarrow \underline{list}, L \Leftarrow \underline{list}[\emptyset] >.$$

Then, $[A|L]/\lambda$ is $\underline{list}[\underline{num}]$ and $[B|L]/\lambda$ is $\underline{list}[\underline{list}]$, but $[A, B|L]/\lambda$ and $[A|B]/\lambda$ are $\underline{list}[\underline{any}]$.

Example 3.2.5 Let $\underline{tree}$ be defined as follows:

tree[α](ψ).
tree[α](tr(L,N,R)) :- tree[α](L), α(N), tree[α](R).

and λ be a type substitution

$$< A \Leftarrow \underline{tree}[num], B \Leftarrow \underline{list}, C \Leftarrow \underline{tree}[\underline{list}] >.$$

And let $\alpha^{(1)}, \alpha^{(2)}$ and $\alpha^{(3)}$ be occurrences of the parameter α at tree[α](L), α(N) and tree[α](R) respectively. Suppose that we compute $tr(\psi, B, C)/\lambda$. Then, the type obtained from $\alpha^{(1)}$ is $\emptyset$, because $\alpha^{(1)}$ appears in a parameter part of the type condition of the 1st argument, i.e., tree[$\alpha^{(1)}$], and ψ/λ is $\underline{tree}[\emptyset]$. The type obtained from $\alpha^{(3)}$ is $\underline{list}$, because $\alpha^{(3)}$ appears in that of the 3rd argument, i.e., tree[α_2], and C/λ is $\underline{tree}[\underline{list}]$. The type obtained from $\alpha^{(2)}$ is B/λ, i.e., $\underline{list}$, because $\alpha^{(2)}$ appears as the predicate of the type condition of the 2nd argument. The union of these types is $\underline{list}$, therefore

$tr(\psi,B,C)/\lambda = \underline{tree}[\underline{list}]$.

Now suppose that we compute $tr(A, B, C)/\lambda$. The types obtained from $\alpha^{(1)}, \alpha^{(2)}, \alpha^{(3)}$ are $\underline{num}$, $\underline{list}$, $\underline{list}$ respectively because A/λ, B/λ, C/λ are $\underline{tree}[\underline{num}]$, $\underline{list}$, $\underline{tree}[\underline{list}]$ respectively. Then, because the union of these types is $\underline{any}$,

$tr(A, B, C)/\lambda = \underline{tree}[\underline{any}]$.

The type substitution obtained from μ and η using $Z/ < t[Z] \Leftarrow \underline{t} >$ and s/λ above is denoted by $propagate(\mu, \eta)$. (Note that ν' depends on just μ and η.)

(3) Overestimation of Type Substitutions

As for the operation at step (A) for type inference, we can adopt the one-way propagation

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

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

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

i.e., elementwise join of the type assigned by the previous type substitution and the one by the one-way propagation.

3.2.3 OLDT Resolution for Type Inference

The relation between a node and its child nodes of a search tree is specified by *OLDT resolution for type inference* as follows.

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

(a) The node is a terminal solution node of Tr and there is some definite clause "$B_0 :- B_1, B_2, \ldots, B_m$" ($m \geq 0$) in program P such that α_1 and B_0 are unifiable, say by an m.g.u. η.

(b) The node is a lookup node of Tr and there is some type-abstracted atom $B\nu$ in the associated solution list of the lookup node. Let η be the renaming of B to α_1.

The precise algorithm of OLDT resolution for type inference is as below. Note that only the operations at steps (A), (B) and (C) are modified.

```
OLDT-resolve(("α1,α2,...,αn", μ) : label) : label ;
 i := 0;
 case
  when a solution node is OLDT resolved
        with "B0 :- B1, B2, ..., Bm" in P
    let η be the m.g.u. of α1 and B0 ;
    let G0 be a negative clause
      "B1, B2, ..., Bm, [α1, μ, η], α2, ..., αn";
    let ν0 be propagate(μ, η)                    — (A)
  when a lookup node is OLDT resolved with "Bν" in Tb
    let η be the renaming of B to α1 ;
    let G0 be a negative clause "α2, ..., αn" ;
    let ν0 be μ ∨ propagate(ν, η) ;              — (B)
 endcase
 while the leftmost of Gi is
        a call-exit marker [Ai+1, μi+1, ηi+1] do
    let Gi+1 be Gi other than the leftmost call-exit marker ;
    let νi+1 be μi+1 ∨ propagate(νi, ηi+1) ;     — (C)
    add Ai+1νi+1 to the last of Ai+1μi+1's solution list
    if it is not in it ;
    i := i + 1 ;
 endwhile
 (Gnew, μnew) := (Gi, νi);
 return (Gnew, μnew).
```

Figure 3.2.1 OLDT Resolution for Type Inference

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

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

3.3 An Example of the Polymorphic Type Inference

We show a simple example of the polymorphic type inference. Recall the following definition of *map-plus* and *plus*.

```
map-plus([ ],M,[ ]).
map-plus([A|L],M,[C|N]) :-
      plus(A,M,C), map-plus(L,M,N).
plus(0,Y,0).
plus(suc(X),Y,suc(Z)) :- plus(X,Y,Z).
```

Then the polymorphic type inference of *map-plus*(X_0, Y_0, Z_0) proceeds as follows, when it is executed with the second argument instantiated to a number.

First, the initial OLDT structure below is generated. The root node is a solution node labeled with ("*map-plus*(L_0, M_0, N_0)", $< M \Leftarrow \underline{num} >$).

Secondly, the root node ("*map-plus*(L_0, M_0, N_0)", $< M \Leftarrow \underline{num} >$) is OLDT resolved using the program. The left child node gives a solution *map-plus*$(L_0, M_0, N_0) < M_0 \Leftarrow \underline{num}, L_0, N_0 \Leftarrow \underline{list}[\emptyset] >$. The right child node is a solution node labeled with ("*plus*(A_2, M_2, C_2),*map-plus*(L_2, M_2, N_2)", $< M_2 \Leftarrow \underline{num} >$). (From now on, the quantities inside call-exit markers are omitted due to space limit so that they are depicted simply by [].)

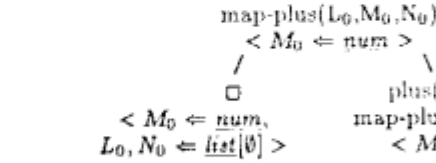


Figure 3.3.1 at Step 2

Thirdly, the solution node is OLDT resolved using the program. The left child node is a lookup node labeled with ("*map-plus*(L_2, M_2, N_2)", $< M_2 \Leftarrow \underline{num} >$), which gives a solution *plus*$(A_2, M_2, C_2) < A_2, M_2, C_2 \Leftarrow \underline{num} >$. The right child node also is a lookup node labeled with ("*plus*(A_3, M_3, C_3),*map-plus*(L_2, M_2, N_2)", $< M_3 \Leftarrow \underline{num} >$).

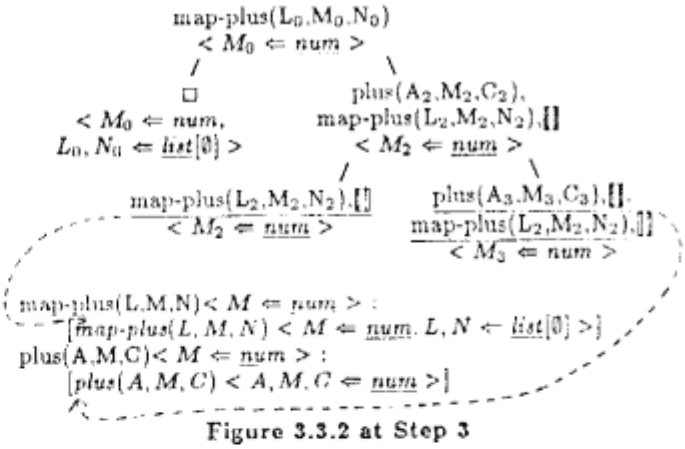
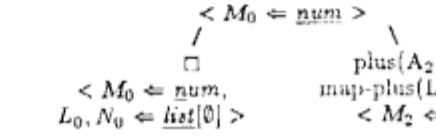


Figure 3.3.2 at Step 3

Fourthly, The left lookup node is OLDT resolved further using the solution table. The child node gives a new solution *map-plus*$(L_0, M_0, N_0) < M_0 \Leftarrow \underline{num}, L_0, N_0 \Leftarrow \underline{list}[num] >$. Fifthly, The right lookup node is OLDT resolved using the solution table. The child is a lookup node labeled with ("*map-plus*(L_2, M_2, N_2)", $< M_2 \Leftarrow \underline{num} >$), which gives a solution *plus*$(A_3, M_3, C_3) < A_3, M_3, C_3 \Leftarrow num >$.

Lastly, the lookup node is OLDT resolved using the solution table. Because the generated child node gives no new solution, the execution process stops.



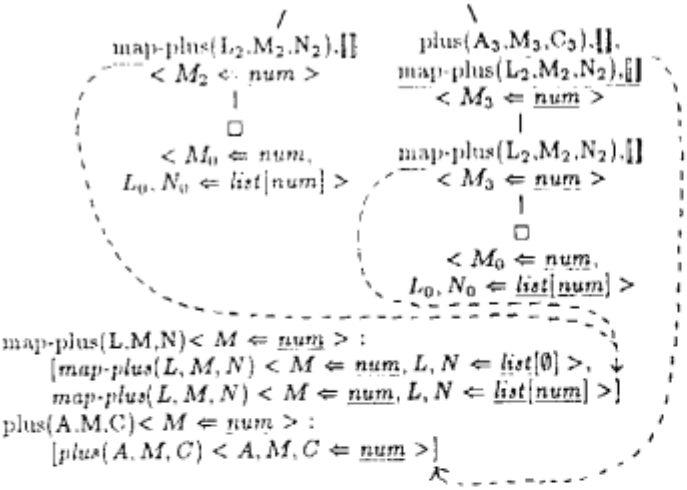


Figure 3.3.3 at Step 6

4. Discussion

Type inference for functional programs has been studied by several researchers, while a few attempts to introduce types into Prolog are done [1],[9],[10].

Bruynooghe [1] discussed the addition of type and mode information to Prolog in order to increase the reliability and readability of the programs. They gave a method to check the consistency using data flow analysis. Mycroft et al. [10] showed a type system for Prolog based on Milner's type polymorphism [8]. In their system, type declarations are considered as the syntactical restrictions on arguments of predicates, and the consistency between type declarations and programs is checked. Both of their systems are type checking systems, not type inference systems. Their slogan is "Well-typed programs do not go wrong." Note that their notion of "wrong" is independent of success, failure or looping. For example [10]. $plus(suc(0), suc(suc(0)), suc(0))$ is well-typed but fails, while $eqnum(foo, foo)$ is ill-typed but succeeds, where *plus* and *eqnum* are defined with type declarations by

```
type-declaration plus(num,num,num).
    plus(0,Y,0).
    plus(suc(X),Y,suc(Z)) :- plus(X,Y,Z).
type-declaration eqnum(num,num).
    eqnum(X,X).
```

In contrast, Mishra [9] considered types as sets of terms which were described by some regular expressions, and gave an algorithm for inferring types from Prolog programs. He didn't require either type declarations or any other type annotations. He claimed that no atom can succeed when its arguments are not the terms described by the types of its predicate. That is, his slogan is "Ill-typed program can't succeed." But his approach is monomorphic and it is not clear whether his idea can be easily extended to polymorphic cases.

In our approach, types are considered as the set of all terms satisfying the definition of type predicates. Although we need to define type predicates, we don't require type declarations. Our system infers types of the arguments from the definition of the predicates using abstract interpretation. Hence, our approach is close to Mishra's one rather than Bruynooghe's or Mycroft's. Moreover we extended monomorphic types to polymorphic ones, and gave a precise algorithm for inferring polymorphic types.

Our approach is new in the following respects :

(1) The meaning of types (including polymorphic types) is clear due to the introduction of the explicit definition of type predicates.
(2) Our system can infer polymorphic types.
(3) Our system can work on goals of any forms i.e. the goals whose types are inferred are not necessarily in general form $p(X_1, X_2, \ldots, X_n)$.

5. Conclusions

We have shown a polymorphic type inference method by abstracted interpretation. This method is an element of our system for analysis of Prolog programs Argus/A under development [3],[4],[5],[6].

Acknowledgments

Our analysis system Argus/A under development is a subproject of the Fifth Generation Computer System(FGCS) "Intelligent Programming System". The authors would like to thank Dr.K.Fuchi (Director of ICOT) for the opportunity of doing this research and Dr.K.Furukawa (Vice Director of ICOT), Dr.T.Yokoi (Vice Director of ICOT) and Dr.H.Ito (Chief of ICOT 3rd Laboratory) for their advice and encouragement.

References

[1] Bruynooghe,M., "Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs", Proc. of 1st International Logic Programming Conference, pp. 129-133, 1982.

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

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

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

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

[6] Maeji,M. and T.Kanamori, "Top-down Zooming Diagnosis of Logic Programs," in preparation, 1987.

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

[8] Milner,R. "A Theory of Type Polymorphism in Programming", J. Computer and Systems Science, Vol. 17, pp. 348-375, 1978.

[9] Mishra,P., "Towards a Theory of Types in Prolog", Proc. of 1984 International Symposium on Logic Programming, pp. 289-298, 1984.

[10] Mycroft,A. and R.A.O'Keefe, "A Polymorphic Type System for Prolog," Artificial Intelligence, Vol. 23, pp.295–307, 1984.

[11] Pereira,L.M.,F.C.N.Pereira and D.H.D.Warren, "User's Guide to DECsystem-10 Prolog", Occasional Paper 15, Dept. of Artificial Intelligence, Edinburgh, 1979.

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