数理解析研究所講究録
第655巻 1988年 147-166

Top-down Zooming Diagnosis of Logic Programs

Machi MAEJI　Tadashi KANAMORI
前地 真知　金森 直

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

Abstract

This paper presents a new diagnosis algorithm for Prolog programs. Bugs are located by tracing either proof trees or search trees in a top-down manner. Human programmers just need to answer "Yes" or "No" for queries issued during the top-down tracing. Moreover, queries about atoms with the same predicates are issued continually so that not only segments containing bugs are identified more quickly but also queries are easier for human programmers to answer. An outline of an implementation of the diagnosis algorithm is shown as well.

Keywords : Program Diagnosis, Debugging, Prolog, Program Analysis.

Contents

1. Introduction

Though it is said that the programming language Prolog is a much higher level language so that writing programs in Prolog is much easier than the conventional languages, still it remains as an important task to debug Prolog programs. Several conventional debugging tools, e.g., "trace" and "spy" commands, are provided in DEC-10 Prolog. In addition, several more advanced debugging tools have been studied by taking advantages of the characteristics of logic programs, e.g., "algorithmic debugging" by Shapiro [10], "declarative debugging" by Lloyd [6], and "rational debugging" by Pereira [8]. In these approaches, they all assume a device, called "*oracle*", which always answers correctly for queries issued during the diagnosis. If the device is a human programmer, not only should the diagnoser be efficient in the query number complexity but also should the queries be easy to answer for human programmers. Attention should be paid to both of these points when an efficient debugging tool for human programmers is aimed at.

This paper presents a new diagnosis algorithm for Prolog programs. Bugs are located by tracing either proof trees or search trees in a top-down manner. Human programmers just need to answer "Yes" or "No" for queries issued during the top-down tracing. Moreover, queries about atoms with the same predicates are issued continually so that not only segments containing bugs are identified more quickly but also queries are easier for human programmers to answer.

This paper is organized as follows: First in Section 2, we will present two kinds of program's bugs, and two DEC-10 Prolog commands, "trace" and "spy". Next in Section 3, we will show a top-down diagnosis algorithm in the "trace" manner. Then we will improve the diagnosis algorithm into the one in the "spy" manner. Last in Section 4, we will show an implementation with efficiency consideration.

The following sections assume familiarity with the basic terminologies of first order logic such as term, atom (atomic formula), clause (definite clause), substitution, most general unifier (m.g.u.) and so on. Knowledge of the semantics of Prolog such as Herbrand interpretations, least Herbrand models and transformation T_P associated with program P is also assumed. The syntax of DEC-10 Prolog is followed. Syntactical variables are X, Y, Z for variables, A, B for atoms, L, G for atom sequences, and θ for substitution, possibly with primes and subscripts. A *program* is a finite set of *definite clauses* of the form "A :- $B_1, B_2, \ldots, B_k$" ($k \geq 0$), where A, B_1, B_2, ..., B_k are atoms. The atom A and the atom sequence B_1, B_2, ..., B_k are called the *head* and the *body* of this clause, respectively. The empty atom sequence is denoted by $\Box$.

2. Diagnosis of Logic Programs

2.1 Bugs of Prolog Programs

Even if a very higher level programming language like Prolog is used, we are likely to write buggy programs.

Example 2.1.1 The following is a program of *quick sort*. It contains a wrong clause in the last line. Its correct clause is shown at the right of the symbol "%".

```
qsort([ ], [ ]).
qsort([X|L], L0) :- partition(L, X, L1, L2), qsort(L1, L3),
                    qsort(L2, L4), append(L3, [X|L4], L0).
partition([X|L], Y, L1, [X|L2]) :- Y≤X, partition(L, Y, L1, L2).
partition([X|L], Y, [X|L1], L2) :- X≤Y, partition(L, Y, L1, L2).
partition([ ], X, [ ], [ ]).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
append([ ], L, [ ]).                          % append([ ], L, L).
```

Example 2.1.2 The following is a program of *permutation*. It misses a recursive clause with the predicate "insert", which is shown at the right of the symbol "%".

```
perm([ ], [ ]).
perm([X|L], N) :- perm(L, M), insert(X, M, N).
insert(X, L, [X|L]).
                          % insert(X, [Y|M], [Y|N]) :- insert(X, M, N).
```

When a Prolog program is buggy, we experience differences between the actual behavior when executed and the intended model in our mind.

The algorithm for *executing* pure Prolog program is the usual ordered linear, that always selects the leftmost atom from atom sequences to be resolved. When $G\theta$ is obtained by executing an atom sequence G in a program P, the instance $G\theta$ is called a *computed solution* (or a *solution*, for simplicity) of G in P. A program is called a *terminating program* when the execution of any atom in the program terminates finitely. In this paper, the program considered are restricted to terminating one.

Let G be an atom sequence and M be an intended Herbrand interpretation in our mind. G is said to be *valid* in M if all ground instances of G are true in M. G is said to be *invalid* in M if some ground instance of G is false in M.

What computed solutions should be returned when an atom sequence is executed in a program that is correct w.r.t. an intended interpretation M? An atom sequence is called an *intended solution* of G with respect to M when it is an instance of G and valid in M. A ground atom sequence is called a *missed solution* of G in P w.r.t. M, when it is an intended solution of G w.r.t. M but not an instance of computed solution of G in P. Then, we say that *the execution-result of G in P is correct* w.r.t. M when

(a) computed solutions of G in P are all intended solutions w.r.t. M, and
(b) there is no missed solution of G in P w.r.t. M.

Otherwise, we say that *the execution-result of G in P is incorrect* w.r.t. M. When the execution-results of all atom sequences in a program are correct w.r.t. M, we say that this program is *correct* w.r.t. M. Otherwise we say that this program is *incorrect* w.r.t. M.

When a program is incorrect w.r.t. an intended interpretation, what kinds of bugs are there in the program? We will define two kinds of bugs in incorrect programs following Shapiro [10] and Lloyd [6].

Definition 2.1.1 — *wrong clause instance* —

Let P be a program and M be an intended interpretation. An instance "A:-L" of a clause in P is called a *wrong clause instance* in P w.r.t. M when

(a) A is invalid in M, and
(b) L is valid in M.

Example 2.1.3 In the program of Example 2.1.1, "*append*([ ], [1], [ ])" is a wrong clause instance, because "*append*([ ], [1], [ ])" is invalid w.r.t. our intention.

Definition 2.1.2 — *uncovered atom* —

Let P be a program and M be an intended interpretation. An atom A is called an *uncovered atom* in P w.r.t. M, when there exists some ground instance $A\theta$ such that

(a) $A\theta$ is true in M, and

(b) for any ground instance "$A\theta$:-L" of a definite clause in P, the body L is false in M.

Example 2.1.4 In the program of Example 2.1.2, "*insert*$(2, [1], X)$" is an uncovered atom, because "*insert*$(2, [1], [1, 2])$" is true in M, and there is no clause in the program whose head is unifiable with the atom.

Then, the following theorem ensures that we can attribute incorrectness of programs to either a wrong clause instance or an uncovered atom (cf. Shapiro [10], Lloyd [6]).

Theorem 2.1

Let P be a terminating program and M be an intended interpretation. Then P is incorrect w.r.t. M if and only if either there is a wrong clause instance in P w.r.t. M or there is an uncovered atom in P w.r.t. M.

Proof: First we will show the "if" part. Suppose that the program P is correct.

If there is a wrong clause instance "A:-L" in P w.r.t. M, the atom A is invalid in M and the atom sequence L is valid in M. Because the program P is correct, L is an computed solution of itself in P. By using the clause instance "A:-L", A is an computed solution of itself in P, so that A is valid in M due to the correctness of P. This fact contradicts the fact that A is invalid in M.

If there is an uncovered atom A for P w.r.t. M, there is a ground instance $A\theta$ such that $A\theta$ is true in M, and for any ground instance "$A\theta$:-L" of a definite clause in P, L is false in M. Then, such L has no computed solution due to the correctness of P. Hence $A\theta$ has no computed solution, but $A\theta$ is true in M, which means that $A\theta$ is a missed solution of A. This fact contradicts the fact that P is correct.

Next, we will show the contrapositive of the "only if" part. Suppose that there is neither a wrong clause instance nor an uncovered atom. For every ground atom A in $T_P(M)$, there is a ground instance "A:-L" of some clause in P such that L is valid in M. Because "A:-L" is not a wrong clause instance, A is valid in M, so that $A \in M$. Hence $T_P(M) \subseteq M$ holds. For every ground atom A in M, because A is not an uncovered atom, there is a ground instance "A:-L" of some clause in P such that L is true in M, so that $A \in T_P(M)$. Hence $T_P(M) \supseteq M$ holds. Therefore $T_P(M) = M$, that is, M is a fixpoint of T_P. (See [5] for the transformation T_P associated with program P.) Because P is terminating, and the finite failure set must be included in the complement of the greatest fixpoint [5], there is just one fixpoint of T_P, i.e., the least Herbrand model of P. Hence, M is the least Herbrand model of P, which obviously means that P is correct w.r.t. M.

2.2 "trace" and "spy" Commands in DEC-10 Prolog

When we experience differences between the program behavior and its intended model, we often trace and examine the execution using a "tracer". Let us trace the execution of an atom "*qsort*$([1, 2], X)$" in the program of Example 2.1.1 using the "trace" command in DEC-10 Prolog. The numbers preceded by the underline " _ " are the inner variables generated by the Prolog system.

```
| ?- trace, qsort([2,1], X).
Debug mode switched on.
    (1)  0 Call  : qsort([2,1], _40)
      (2)  1 Call  : partition([1], 2, _105, _106)
        (3)  2 Call  : 2≤1
        (3)  2 Fail  : 2≤1
        (4)  2 Call  : 1≤2
        (4)  2 Exit  : 1≤2
        (5)  2 Call  : partition([ ], 2, _120, _106)
        (5)  2 Exit  : partition([ ], 2, [ ], [ ])
      (2)  1 Exit  : partition([1], 2, [1], [ ])
      (6)  1 Call  : qsort([1], _107)
        (7)  2 Call  : partition([ ], 1, _149, _150)
        (7)  2 Exit  : partition([ ], 1, [ ], [ ])
        (8)  2 Call  : qsort([ ], _151)
        (8)  2 Exit  : qsort([ ], [ ])
        (9)  2 Call  : qsort([ ], _152)
        (9)  2 Exit  : qsort([ ], [ ])
       (10)  2 Call  : append([ ], [1], _107)
       (10)  2 Exit  : append([ ], [1], [ ])
      (6)  1 Exit  : qsort([1], [ ])
     (11)  1 Call  : qsort([ ], _108)
     (11)  1 Exit  : qsort([ ], [ ])
     (12)  1 Call  : append([ ], [2], _40)
     (12)  1 Exit  : append([ ], [2], [ ])
    (1)  0 Exit  : qsort([2,1], [ ])
X = [ ] ;
    (1)  0 Redo : qsort([2,1], [ ])
      (12)  1 Redo : append([ ], [2], [ ])
      (12)  1 Fail  : append([ ], [2], _40)
      (11)  1 Redo : qsort([ ], [ ])
      (11)  1 Fail  : qsort([ ], _108)
       (6)  1 Redo : qsort([1], [ ])
        (10)  2 Redo : append([ ], [1], [ ])
        (10)  2 Fail  : append([ ], [1], _107)
         (9)  2 Redo : qsort([ ], [ ])
         (9)  2 Fail  : qsort([ ], _152)
         (8)  2 Redo : qsort([ ], [ ])
         (8)  2 Fail  : qsort([ ], _151)
         (7)  2 Redo : partition([ ], 1, [ ], [ ])
         (7)  2 Fail  : partition([ ], 1, _149, _150)
       (6)  1 Fail  : qsort([1], _107)
       (2)  1 Redo : partition([1], 2, [1], [ ])
         (5)  2 Redo : partition([ ], 2, [ ], [ ])
         (5)  2 Fail  : partition([ ], 2, _120, _106)
         (4)  2 Redo : 1≤2
         (4)  2 Fail  : 1≤2
       (2)  1 Fail  : partition([1], 2, _105, _106)
    (1)  0 Fail  : qsort([2,1], _40)
no
```

Figure 2.2.1 Example of "trace"

When it is too messy to examine all the trace list step by step, we often focus our attention on the behavior of specific predicates. Let us put a spy point on the predicate "*qsort/2*" in the program of Example 2.1.1 and trace the execution of an atom "$qsort([2,1], X)$" using the "spy" command in DEC-10 Prolog.

```
| ?- spy(qsort/2), qsort([2,1], X).
Spy-point placed on qsort/2.
Debug mode switched on.
   ** (1)  0 Call  : qsort([2,1], _40)
     ** (6)  1 Call  : qsort([1], _107)
       ** (8)  2 Call  : qsort([ ], _151)
       ** (8)  2 Exit  : qsort([ ], [ ])
       ** (9)  2 Call  : qsort([ ], _152)
       ** (9)  2 Exit  : qsort([ ], [ ])
     ** (6)  1 Exit  : qsort([1], [ ])
     ** (11)  1 Call  : qsort([ ], _108)
     ** (11)  1 Exit  : qsort([ ], [ ])
   ** (1)  0 Exit  : qsort([2,1], [ ])
X = [ ] ;
   ** (1)  0 Redo  : qsort([2,1], [ ])
     ** (11)  1 Redo  : qsort([ ], [ ])
     ** (11)  1 Fail  : qsort([ ], _108)
     ** (6)  1 Redo  : qsort([1], [ ])
       ** (9)  2 Redo  : qsort([ ], [ ])
       ** (9)  2 Fail  : qsort([ ], _152)
       ** (8)  2 Redo  : qsort([ ], [ ])
       ** (8)  2 Fail  : qsort([ ], _151)
     ** (6)  1 Fail  : qsort([1], _107)
   ** (1)  0 Fail  : qsort([2,1], _40)
no
```

Figure 2.2.2 Example of "spy"

In Section 3, first, a top-down diagnosis algorithm in the "trace" manner is going to be developed for systematizing the process of examining trace lists. Then, a top-down diagnosis algorithm in the "spy" manner is going to be developed for systematizing the process of examining specific predicates.

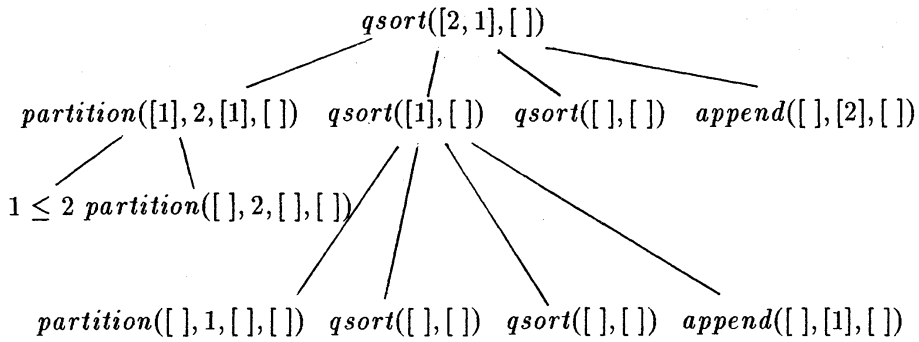
2.3 Proof Tree and Search Tree

We can show a diagnosis algorithm using the terminology of "trace list". However, it is difficult to formally discuss the properties of the diagnosis algorithm, e.g., soundness and completeness, using the terminology. So we will give the diagnosis algorithm using the terminology of "proof tree" and "search tree".

(1) Proof Tree

A *proof tree* of an atom A in a program P is a tree T whose nodes are labelled with atoms as follows: T is a proof tree of A when T has immediate subtrees T_1, T_2, ..., T_n ($n \geq 0$) with their root labels A_1, A_2, ..., A_n satisfying the following conditions. The root label of T is A, "A :- $A_1, A_2, \ldots, A_n$" is an instance of some clause in P, and T_1, T_2, ..., T_n are proof trees of A_1, A_2, ..., A_n in P. The clause "A :- A_1, A_2, ..., A_n" is said to be *used at the root* of the proof tree T, and the atoms A_1, A_2, ..., A_n are called *child atoms* of A in T.

Example 2.3.1 An atom "$qsort([2,1],[\,])$" is a computed solution of atom "$qsort([2,1],X)$" in the program of Example 2.1.1. Its proof tree is as below:

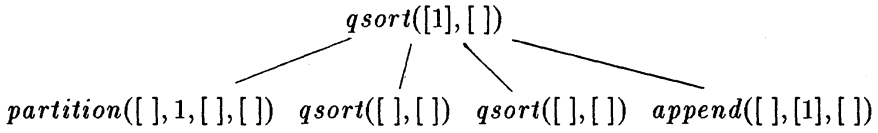


The child atoms of "$qsort([2,1],[\,])$" in this proof tree are
$partition([1],2,[1],[\,])$, $qsort([1],[\,])$, $qsort([\,],[\,])$ and $append([\,],[2],[\,])$.
The clause used at the root in this proof tree is
$qsort([2,1],[\,])$:- $partition([1],2,[1],[\,]), qsort([1],[\,]), qsort([\,],[\,]), append([\,],[2],[\,])$.

(2) Proof Subtree

A subtree of a proof tree T is called a *proof subtree* of T. In particular, a proof subtree is called an *immediate proof subtree* of T when
(a) it is properly contained in T, and
(b) it is not properly contained in any proof subtree satisfying (a).
The root labels of the immediate proof subtrees of T are child atoms of the root label of T.

Example 2.3.2 The following is an immediate proof subtree of the proof tree of Example 2.3.1.



(3) Search Tree

A *search tree* of an atom sequence G in a program P is a tree T whose nodes are labelled with atom sequences and whose edges are labelled with substitutions as follows:

(a) If G is an empty atom sequence, T is a search tree of G when it is a tree consisting of only one node labelled with $\square$.
(b) If G is a non-empty atom sequence "A, L", let "A_1 :- L_1", "A_2 :- L_2", ..., "A_n :- L_n" be all the clauses whose heads are unifiable with A, say by m.g.u.'s $\theta_1, \theta_2, \ldots, \theta_n$. Let $T_1, T_2, \ldots, T_n$ $(n \geq 0)$ be all immediate subtrees of T, and $G_1, G_2, \ldots, G_n$ be their root labels. T is a search tree of G when the following conditions are satisfied.
  b-1 G_i is of the form "$(L_i, L)\theta_i$". The atom sequences $G_1, G_2, \ldots, G_n$ are called *child atom sequences* of G in T. The clause "$(A_i$:- $L_i)\theta_i$" is said to be *used at the root* of the search tree T.
  b-2 θ_i is the label of the edge from the root node of T to the root node of T_i.
  b-3 T_i is a search tree of G_i in P.

A path in a search tree from the root to a node labelled with $\square$ is called a *success path.*

A *search tree* of an atom A in a program P is a search tree of the atom sequence consisting of only one atom A. A success path in a search tree of A corresponds to some proof tree of A.

Example 2.3.3 When an atom "$perm([2,1],X)$" is executed in the program of Example 2.1.2, it returns only one computed solution "$perm([2,1],[2,1])$". Its search tree is as below:

$$perm([2,1],X)$$
$$\mid \ \{\}$$
$$perm([1],Y),insert(2,Y,X)$$
$$\mid \ \{\}$$
$$perm([\,],Z),insert(1,Z,Y),insert(2,Y,X)$$
$$\mid \ \{Z \Leftarrow [\,]\}$$
$$insert(1,[\,],Y),insert(2,Y,X)$$
$$\mid \ \{Y \Leftarrow [1]\}$$
$$insert(2,[1],X)$$
$$\mid \ \{X \Leftarrow [2,1]\}$$
$$\square$$

The child atom sequence of "$perm([2,1],X)$" in the search tree is only
"$perm([1],X),insert(2,Y,X)$".
The clause used at the root in this search tree is
"$perm([2,1],X)$:- $perm([1],Y)$, $insert(2,Y,X)$."
(In this example, because the second clause for *insert* is missed, the search tree is a tree without multiple branching, i.e., a path. But, this is not a case in general.)

(4) Search Subtree

Let T be a search tree, and ν_1 be a node in T labelled with non-empty atom sequence. Consider a path U from the node ν_1 to a node ν_2 in T such that, for every node ν on the path other than ν_2,

$$length(\nu) \geq length(\nu_1)$$

holds. ($length(\nu)$ denotes the number of atoms in the label of ν.) Then, the path, which is constructed by neglecting last $length(\nu_1) - 1$ atoms in the label of every node on the path U, is called a *subdeduction* at the node ν_1 in T.

Let ν be a node in a search tree T, and U_1, U_2, ..., U_h be all subdeductions at ν in T, that are not properly contained in any subdeduction at ν in T. Then, the tree, which is constructed by putting the paths U_1, U_2, ..., U_h together, is called a *search subtree* of T at ν. Let T_1 and T_2 be search subtrees of T at u_1 and u_2, respectively. Then, T_2 is said to be *properly contained in* T_1 when T_2 is a search subtree of T_1 at some node u other than the root node in T_1, and the node u corresponds to u_2 in T. Note that the root label of a search subtree is always one atom, and a search subtree with root label "A" is a search tree of "A".

In particular, a search subtree of T is called an *immediate search subtree* of T when
(a) it is properly contained in T, and
(b) it is not properly contained in any search subtree satisfying (a).

Example 2.3.4 The tree below is an immediate search subtree of the search tree of Example 2.3.3. (Because the original search tree does not have a multiple branching, neither does this immediate search subtree. But, this is not a case in general, either.)

$$perm([1], Y)$$

$$| \quad \{\}$$

$$perm([\,], Z), insert(1, Z, Y)$$

$$| \quad \{Z \Leftarrow [\,]\}$$

$$insert(1, [\,], Y)$$

$$| \quad \{Y \Leftarrow [1]\}$$

$$\square$$

The tree below is the part of the search tree which consists of the nodes corresponding to those in the search subtree above.

$$perm([1], Y), insert(2, Y, X)$$

$$| \quad \{\}$$

$$perm([\,], Z), insert(1, Z, Y), insert(2, Y, X)$$

$$| \quad \{Z \Leftarrow [\,]\}$$

$$insert(1, [\,], Y), insert(2, Y, X)$$

$$| \quad \{Y \Leftarrow [1]\}$$

$$insert(2, [1], X)$$

3. Top-down Zooming Diagnosis of Logic Programs

In this section, we will first present two kinds of unexpected execution-results. Next, we will present a top-down diagnosis algorithm in the "trace" manner. Then, we will improve the top-down diagnosis algorithm in the "spy" manner.

3.1 Unexpected Success and Unexpected Failure

(1) Unexpected Success (Incorrect Solution)

Suppose that the execution of an atom A has succeeded with computed solution $A\theta$. If $A\theta$ is invalid in our intended interpretation M, $A\theta$ is said to have *succeeded unexpectedly* w.r.t. M.

Example 3.1.1 An atom "$qsort([2, 1], X)$" in the program of Example 2.1.1 has a computed solution $qsort([2, 1], [\,])$. But it is invalid w.r.t. our intention. Hence, the success of atom $qsort([2, 1], [\,])$ is an unexpected one.

(2) Query for Invalid Instances

To examine whether an atom A has succeeded unexpectedly or not, our diagnoser issues a query as follows :

"Is some instance of A false?"

The answer for this query is either "Yes" or "No".

"Yes": The atom A is invalid, hence, it has succeeded unexpectedly.

"No": The atom A is valid, hence, it is an intended solution.

Example 3.1.2 In Example 3.1.1, our diagnoser asks a query as follows:

"Is some instance of $qsort([2,1],[\,])$ false?"

The human programmer (or oracle) answers "Yes", hence the success of $qsort([2,1],[\,])$ is an unexpected one.

(3) Unexpected Failure (Missing Solution)

Suppose that the execution of an atom A has failed after returning several (possibly zero) computed solutions that are all valid in our intended interpretation M. If the atom A has some missed solution, the atom A is said to have *failed unexpectedly* w.r.t. M.

Example 3.1.3 The execution of atom "$perm([2,1],X)$" in the program of Example 2.1.2 fails after returning only one computed solution "$perm([2,1],[2,1])$", which is valid in our intention. But the atom has a missed solution "$perm([2,1],[1,2])$". Hence, the last failure of atom "$perm([2,1],X)$" is an unexpected one.

(4) Query for Valid Instances

Suppose that the execution of an atom A has failed after returning several (possibly zero) computed solutions, which has been already confirmed to be valid in our intended interpretation. To examine whether the execution has failed unexpectedly or not after obtaining these computed solutions, our diagnoser issues a query as follows:

"Is some other instance of A true?"

The answer for these queries is either "Yes" or "No".

"Yes": The atom A has some missed solution, hence, it has failed unexpectedly.

"No": The atom A has all intended solutions.

Example 3.1.4 Suppose that all computed solutions of an atom $perm([2,1],X)$ in the program of Example 2.1.2 have been confirmed to be correct w.r.t. our intention. Then our diagnoser asks a query as follows:

"Is some other instance of $perm([2,1],X)$ true?"

The human programmer (or oracle) answers "Yes". Hence the last failure of $perm([2,1],X)$ (with only one computed solution $perm([2,1],[2,1])$) is an unexpected one.

3.2 Top-down Diagnosis Algorithm

The top-down diagnosis algorithm "*diagnose*0" receives a tree (either a proof tree or a search tree of an atom), and returns a definite clause instance, an atom, or a message "no bug is found".

diagnose0(T : tree) : bug-message ;
when T is a proof tree with root label A
issue a query "Is some instance of A false?"
if the answer is "No"
then return "no bug is found"
else let PT_1, PT_2, ..., PT_n be the immediate proof subtrees of T;
if the application of "*diagnose*0" to some PT_j returns a bug
then return it
else return the clause used at the root of T as a bug;
when T is a search tree with root label A
let T_1, T_2, ..., T_k be the proof trees corresponding to success paths in T;
if the application of "*diagnose*0" to some T_i returns a bug
then return it
else issue a query "Is some other instance of A true?"
if the answer is "No"
then return "no bug is found"
else let ST_1, ST_2, ..., ST_n be the immediate search subtrees of T;
if the application of "*diagnose*0" to some ST_j returns a bug
then return it
else return the atom A as a bug

Figure 3.2 Top-down Diagnosis Algorithm Using Subtrees

The following theorem holds for this algorithm. (See [7] for this proof.)

Theorem 3.2 *(soundness and completeness of the top-down diagnosis algorithm)*
Let P be a terminating program, M be an intended interpretation, and T be a tree (either a proof tree or a search tree) of an atom A. The execution-result of the atom A in P is incorrect w.r.t. M, if and only if "diagnose0" applied to T returns either a wrong clause instance or an uncovered atom w.r.t. M.

In the following examples, an "answer database" accumulates answers to previous queries in order to partly mechanize the oracle answers. A new query is first posed to the "answer database". Only if the "answer database" fails to answer it, a query is issued to the programmer (or an oracle), and the answer is added to the "answer database". (See Shapiro [10].)

Example 3.2.1 Let us diagnose an atom $qsort([2,1], X)$ in the program of Example 2.1.1. In the following, "*diagnose*0" takes the root label of each tree as an argument instead of the tree itself. (See section 4.1 for its implementation.)

|?- diagnose0($qsort([2,1], X)$).
Is some instance of $qsort([2,1],[\,])$ false? yes.
Is some instance of $partition([1], 2, [1], [\,])$ false? no.
Is some instance of $qsort([1],[\,])$ false? yes.
Is some instance of $partition([\,], 1, [\,], [\,])$ false? no.
Is some instance of $qsort([\,],[\,])$ false? no.
Is some instance of $append([\,], [1], [\,])$ false? yes.

%%% Wrong Clause Instance %%% $append([\,], [1], [\,])$

yes

Example 3.2.2 Let us diagnose an atom $perm([2,1],X)$ in the program of Example 2.1.2.

|?- diagnose0($perm([2,1],X)$).
Is some instance of $perm([2,1],[2,1])$ false? no.
Is some other instance of $perm([2,1],\_47)$ true? yes.
Is some instance of $perm([1],[1])$ false? no.
Is some other instance of $perm([1],\_319)$ true? no.
Is some instance of $insert(2,[1],[2,1])$ false? no.
Is some other instance of $insert(2,[1],\_47)$ true? yes.

%%% Uncovered Atom %%% $insert(2,[1],\_47)$

X = _47

yes

3.3 Top-down Zooming Diagnosis Algorithm

During the the top-down diagnosis, our diagnoser issues several queries for human programmers (or oracles). All these queries are issued about instances of atoms, whose predicate may change query by query. The human programmers must change his attention according to the predicates of atoms. Instead, we can change the order of queries to issue the queries about atoms with the same predicate continually so that the burden of human programmers is lightened. This change of the order also makes it possible to quickly narrow down the location containing a bug.

(1) Immediate Recursion Subtree in Proof Trees

Let T be a proof tree in a program P whose root label is an atom A with predicate p. A proof subtree in T is called a *recursion proof subtree* of T when its root label is with predicate p. In particular, a proof subtree in T is called an *immediate recursion proof subtree* when

(a) it is properly contained in T,
(b) the root label of the proof subtree is with predicate p, and
(c) it is not properly contained in any proof subtree satisfying (a) and (b).

Example 3.3.1 In the proof tree of Example 2.3.1, the atom $qsort([2,1],[\,])$ has only two immediate recursion proof subtrees with roots $qsort([1],[\,])$ and $qsort([\,],[\,])$.

(2) Immediate Recursion Subtree in Search Trees

Let T be a search tree in a program P whose root label is an atom A with predicate p. A search subtree in T is called a *recursion search subtree* of T when its root label is with predicate p. In particular, a search subtree in T is called an *immediate recursion search subtree* when

(a) it is properly contained in T,
(b) the root label is with predicate p, and
(c) it is not properly contained in any search subtree satisfying (a) and (b).

Example 3.3.2 In the search tree of Example 2.3.3, the atom "$perm([2,1],X)$" has only one immediate recursion search subtree with root $perm([1],Y)$.

(3) A Top-down Zooming Diagnosis Algorithm Using Recursion Subtrees

The top-down zooming diagnosis algorithm using recursion subtrees is almost the same as the previous top-down diagnosis algorithm "*diagnose*0" except that it works with aids of a subprocedure "*zooming*", which receives a tree and returns a subtree for diagnosis by searching for recursion subtrees.

```
diagnose(T : tree) : bug-message ;
    if the application of "zooming" to T does not return a tree
    then return "no bug is found"
    else let T' be the tree returned by "zooming"
        when T' is a proof tree with root label A
          let PT_1, PT_2, ..., PT_n be the immediate proof subtrees of T';
          if the application of "diagnose" to some PT_j returns a tree
          then return it
          else return the clause used at the root of T' as a bug;
        when T' is a search tree with root label A
          let ST_1, ST_2, ..., ST_n be the immediate search subtrees of T';
          if the application of "diagnose" to some ST_j returns a bug
          then return it
          else return the atom A as a bug

zooming(T : tree) : tree ;
    when T is a proof tree with root label A
      issue a query "Is some instance of A false?"
      if the answer is "No"
      then return "no tree is found"
      else let PT_1, PT_2, ..., PT_n be the immediate recursion proof subtrees of T;
          if the application of "zooming" to some PT_j returns a tree
          then return it
          else return T
    when T is a search tree with root label A
      let T_1, T_2, ..., T_k be the proof trees corresponding to success paths in T;
      if the application of "zooming" to some T_i returns a tree
      then return it
      else issue a query "Is some other instance of A true?"
          if the answer is "No"
          then return "no tree is found"
          else let ST_1, ST_2, ..., ST_n be the immediate recursion search subtrees of T;
              if the application of "zooming" to some ST_j returns a tree
              then return it
              else return T
```

Figure 3.3 Top-down Zooming Diagnosis Algorithm Using Recursion Subtrees

Roughly speaking, the top-down zooming diagnosis algorithm identifies the subtrees containing a bug by changing its attention in the following two different ways by turns.

(a) One way is to narrow down to immediate recursion subtrees quickly by changing "*diagnose*0" to possibly leap the intermediate subtrees. "*zooming*" above detects a

recursion subtree such that the execution-result of its root label is incorrect but the execution-result of root label of every immediate recursion subtree is correct w.r.t. M.

(b) The other way is to narrow down to the immediate subtrees slowly in the same way as "*diagnose*0". "*diagnose*" above proceeds in the same way as "*diagnose*0" after making use of "*zooming*" first.

The following theorem holds for this algorithm. (See [7] for this proof.)

Theorem 3.3 *(soundness and completeness of the top-down zooming diagnosis algorithm)* *Let P be a terminating program, M be an intended interpretation, and T be a tree (either a proof tree or a search tree) of an atom A. When the execution-result of the atom A in P is incorrect w.r.t. M, if and only if "diagnose" applied to T returns either a wrong clause instance or an uncovered atom w.r.t. M.*

Example 3.3.3 Let us diagnose the atom $qsort([2,1],X)$ in the program of Example 2.1.1 by this top-down zooming diagnosis.

|?- diagnose($qsort([2,1],X)$).
Is some instance of $qsort([2,1],[\,])$ false? yes.
Is some instance of $qsort([1],[\,])$ false? yes.
Is some instance of $qsort([\,],[\,])$ false? no.
Is some instance of $partition([\,],1,[\,],[\,])$ false? no.
Is some instance of $append([\,],[1],[\,])$ false? yes.

%%% Wrong Clause Instance %%% $append([\,],[1],[\,])$

yes

Example 3.3.4 Let us diagnose the atom $perm([2,1],X)$ in the program of Example 2.1.2 by this top-down zooming diagnosis.

|?- diagnose($perm([2,1],X)$).
Is some instance of $perm([2,1],[2,1])$ false? no.
Is some other instance of $perm([2,1],\_47)$ true? yes.
Is some instance of $perm([1],[1])$ false? no.
Is some other instance of $perm([1],\_319)$ true? no.
Is some instance of $insert(2,[1],[2,1])$ false? no.
Is some other instance of $insert(2,[1],\_47)$ true? yes.

%%% Uncovered Atom %%% $insert(2,[1],\_47)$

X = _47

yes

4. Implementation of the Top-down Zooming Diagnosis Algorithm

In this section, we will show a brief outline of an implementation of the top-down zooming diagnosis algorithm.

4.1 Consideration on Space Efficiency

We assume that all the trees used in the diagnosis of Section 3.3 are recorded in a "tree database". They are used somehow for processing proof trees and search trees, which are passed as arguments of "*diagnose*" and "*zooming*". However, recording all the trees is very space-consuming. Recall how the trees are used in the diagnosis. They are used only when

(a) "*diagnose*" recurses with an immediate subtree,
(b) "*zooming*" recurses with some tree, either an immediate recursion subtree or a proof tree corresponding to success path, or
(c) "*zooming*" issues a query about the root label of the tree.

So, if we can obtain the root labels of any immediate subtree, any immediate recursion subtree, and any proof tree corresponding to success path somehow, it is enough for the diagnosis. Let A be a root label of a tree.

The root label of its immediate subtree are obtained from A by repeating only the top-level of the computation using the root labels of trees as follows:

(a) When A is a root label of a proof tree PT, let A_1, A_2, ..., A_n be the root labels of all immediate proof subtrees of PT. Then there is a clause "B:-$B_1, B_2, \ldots, B_n$" in P and a substitution θ such that $A_i \equiv B_i\theta (1 \leq i \leq n)$ and $A \equiv B\theta$ hold. On the other hand, if there is a clause "B:-$B_1, B_2, \ldots, B_n$" in P and a substitution θ such that $A_1 \equiv B_1\theta$, $A_2 \equiv B_2\theta$, ..., $A_n \equiv B_n\theta$ hold for some root labels A_1, A_2, ..., A_n of proof trees in the "tree database", then $B\theta$ should succeed in P using the clause instance "$(B$:-$B_1, B_2, \ldots, B_n)\theta$" in P. Hence, we may conclude that there is a proof tree for $B\theta$ in the "tree database" such that the root labels of all its immediate proof subtrees are the atoms A_1, A_2, ..., A_n.
(b) When A is a root label of a search tree ST, let A' be a root label of an immediate search subtree of ST. Then there is a clause instance "H:-$A_1, A_2, \ldots, A_n$" such that the head is unifiable with A, say by σ, and $A_i\sigma\theta$ is equal to A' for some computed solution $(A_1, A_2, \ldots, A_{i-1})\sigma\theta$ of "$(A_1, A_2, \ldots, A_{i-1})\sigma$" $(1 \leq i \leq n)$. On the other hand, all the atoms constructed by such a way are the root labels of immediate search subtrees of ST.

The root labels of immediate recursion subtrees are obtained by constructing the root labels of immediate subtrees recursively.

The root labels of proof trees corresponding to success paths of a search tree for A are obtained with less time (in compensation for the space for recording), if we record them in the structure which associates A to the root labels of such proof trees.

Hence, the root labels of subtrees are all obtained by using the clauses in P and the recorded root labels of trees.

Now, we do not need all the information in the trees. The information we record in the "tree database" is only

(a) the root labels of proof trees, and
(b) the pairs of the root label of a search tree, and the sequence of the root labels of its proof trees corresponding to success paths.

(Hence, it is more appropriate to call it a "label database".) Due to this implementation method, the arguments of "*diagnose*" and "*zooming*" are now not trees but root labels of the trees.

4.2 Consideration on Time Efficiency

Even if we have employed a space-efficient representation above, the space required for recording them is still large. We can reduce the necessary space at each instance by recording only some root labels of the tree which is immediately necessary for the present, and by recomputing another root labels which will become necessary later. In the top-down zooming diagnosis, we do not need to immediately search root labels of trees other than recursion subtrees so that the diagnoser needs to record only the root labels of the recursion

subtrees relevant for the present. The computation of the root labels of trees other than recursion subtrees is done afterwards if necessary.

If we have employed such an implementation method for reducing space at each instance, it may require much more time due to the recomputation, i.e., some goals might have to be re-executed again and again during the diagnosis. However, note that the atoms appearing in the re-execution are only those appearing in the execution before. Hence, we can improve the time-efficiency by utilizing the root labels of proof trees, that are used before for the diagnosis, during each re-execution to avoid some of the recomputation.

4.3 Consideration on Query

As was already used in the previous examples, an "answer database" accumulates answers to previous queries in order to partly mechanize the oracle answers. A new query is first posed to the "answer database". Only if the "answer database" fails to answer it, the query is asked to a programmer (or an oracle), and the answer is added to the "answer database". (See Shapiro [10].)

(1) Collective Queries

The queries can be improved to be more natural for human programmers in several points. For example, it is more natural to ask

"Is some instance of A true?"

instead of "Is some other instance of A true?" when A has failed without returning any computed solution. It is also more natural to ask

"Is A true?"

"Is A false?"

when the atom A in queries are ground.

In addition to such easy improvements, it is more comfortable for human programmer to answer several related questions at one stroke. Such queries reduce the number of answers the programmer must type in. For example, in the diagnosis for unexpected success, the queries for the root labels of every immediate recursion subtrees (or every immediate subtrees) can be issued together. Similarly, in the diagnosis for unexpected failure, the queries for the root labels of proof trees corresponding to success paths can be issued together.

Example 4.3.1 Let us diagnose the atom $qsort([2,1],X)$ in the program of Example 2.1.1 by issuing several queries together.

```
|?- diagnose(qsort([2,1],X)).
Is some instance of the following atoms false?
     1 : qsort([2,1],[ ])                    Which? 1.
     1 : qsort([1],[ ])
     2 : qsort([ ],[ ])                      Which? 1.
     1 : qsort([ ],[ ])                      Which? no.
     1 : partition([ ],1,[ ],[ ])
     2 : append([ ],[1],[ ])                 Which? 2.
%%% Wrong Clause Instance %%%   append([ ],[1],[ ])
yes
```

Example 4.3.2 Let us diagnose the atom $perm([2,1],X)$ in the program of Example 2.1.2 in the same way.

|?- diagnose($perm([2,1],X)$).
Is some instance of the following atoms false?

1 : $perm([2,1],[2,1])$ Which? no.
Is some other instance of $perm([2,1],\_47)$ true? yes.

1 : $perm([1],[1])$ Which? no.
Is some other instance of $perm([1],\_319)$ true? no.

1 : $insert(2,[1],[2,1])$ Which? no.
Is some other instance of $insert(2,[1],\_47)$ true? yes.

%%% Uncovered Atom %%% $insert(2,[1],\_47)$

X = _47

yes

(2) Query for Instances of Missed Solutions

As was adopted by Shapiro [10] and Lloyd [6], we can enjoy both the time efficiency and the space efficiency, if an oracle can give a suitable instantiation of variables to the diagnoser. Suppose that the diagnoser is modified to ask the oracle to give a missed solution when an oracle has given an answer "Yes" for a query "Is some other instance of A true?". If such a missed solution is given, the number of queries decreases in some cases, because the number of immediate search subtrees to be diagnoses decreases.

Example 4.3.3 Let *sort* be a predicate defined by

```
sort(L,N) :- perm(L,N), ordered(N).
perm([ ],[ ]).
perm([X|L],N) :- perm(L,M), insert(X,M,N).
insert(X,M,[X|M]).
insert(X,[Y|M],[Y|N]) :- insert(X,M,N).
```

It misses the program of the predicate *ordered*. The diagnosis proceeds as below if the previous algorithm is used.

|?- diagnose($sort([2,3,1],X)$).
Is some instance of $sort([2,3,1],\_55)$ true? yes.
Is some instance of $perm([2,3,1],[2,3,1])$ false? no.
Is some instance of $perm([2,3,1],[3,2,1])$ false? no.
Is some instance of $perm([2,3,1],[3,1,2])$ false? no.
Is some instance of $perm([2,3,1],[2,1,3])$ false? no.
Is some instance of $perm([2,3,1],[1,2,3])$ false? no.
Is some instance of $perm([2,3,1],[1,3,2])$ false? no.
Is some other instance of $perm([2,3,1],\_55)$ true? no.
Is some instance of $ordered([2,3,1])$ true? no.
Is some instance of $ordered([3,2,1])$ true? no.
Is some instance of $ordered([3,1,2])$ true? no.
Is some instance of $ordered([2,1,3])$ true? no.
Is some instance of $ordered([1,2,3])$ true? yes.

%%% Uncovered Atom %%% $ordered([1,2,3])$

X = [1,2,3]

yes

If the diagnoser is modified as given in this section, the diagnosis process is as shown below.

|?- diagnose($sort([2,3,1],X)$).
Is some instance of $sort([2,3,1],\_55)$ true? yes.
What is the correct instance of _55 ? [1,2,3].
Is some instance of $perm([2,3,1],[1,2,3])$ false? no.
Is some instance of $ordered([1,2,3])$ true? yes.

%%% Uncovered Atom %%% $ordered([1,2,3])$

X = [1,2,3]

yes

When a programmer knows that there are some missed solutions, he/she probably knows some of the missed solutions. Of course, he/she may refuse to give such an instance if he/she thinks it troublesome to give a concrete true instance. We are not sure, however, which imposes less burden on the programmers in general. It depends on the characteristics of programs.

(3) Oracle Answer "Unknown"

So far, we have assumed that the oracle returns a definite answer "Yes" or "No". When the oracle is a human programmer, however, he/she may want to give an answer "Unknown" occasionally. We can modify the diagnoser so as to accept such an answer.

Recall the differences between the top-down diagnosis and the top-down zooming diagnosis. The latter was in the same way as the former except that the recursion subtrees are diagnosed in the preference to the top-down manner by *zooming*.

Suppose that the answer "Unknown" is returned. We may continue the diagnosis by preference for the other subtrees in the same way as *zooming*. If the definite answer of the root node of this tree is needed finally, the diagnoser asks again to return the reserved answer.

5. Discussion

Debugging of logic programs has been studied by several researchers intensively. Shapiro [10] said that program debugging is composed of *program diagnosis*, the process of finding a bug, and *program correction*, the process of fixing the bug. In this paper we have discussed the program diagnosis.

We have attributed incorrectness of programs to two bugs, *wrong clause instance* and *uncovered atom* in the same way as Shapiro [10], Lloyd [6], et al. However, *wrong clause instances* and *uncovered atoms* in our definitions are slightly different from those in Shapiro [10] or Lloyd's definitions [6]. For example, in the definition of a *wrong clause instance* by Lloyd [6], the condition (a) in Definition 2.1.1 is replaced with the following:
"(a) the atom A is unsatisfiable in M, i.e., all ground instances of A are false in M, and"
In the definition of an *uncovered atom* by Lloyd [6], an atom A is called an *uncovered atom* when
(a) A is valid in M, and

(b) for any clause "B :- L" in P whose head B is unifiable with A, say by an m.g.u. θ, $L\theta$ is unsatisfiable in M, i.e., all ground instance of $L\theta$ are false in M.

Theorem 2.1 is the same as Proposition 3 in Lloyd [6] p.6. These differences do not affect the proof of this theorem.

Our definition of *correctness* is stronger than that of Lloyd [6]. Our definition implies that a program P is correct w.r.t. an intended Herbrand model M if and only if M is the least Herbrand model of completion P^*, while his definition is that a program P is correct w.r.t. an intended model M if and only if M is a model of completion P^*. Hence, for proving Theorem 2.1, we needed an additional condition "terminating".

In addition to these subtle differences of definitions, our diagnoses algorithm is different from theirs in several respects.

(1) Our diagnosis algorithm is basically top-down.

Shapiro's algorithm [10] for unexpected success (incorrect solution) is in the bottom-up manner, while that for unexpected failure (missing solution) is in the top-down manner. There is no inherent reason to stick to the bottom-up manner. In fact, Lloyd [6] showed the top-down algorithm for unexpected success.

Similarly to Lloyd [6], our diagnosis algorithm is basically top-down. For non-recursive predicates, our approach issues queries in the usual top-down manner so that the programmers can locate bugs more quickly than the single stepping bottom-up diagnoser in general.

(2) Our diagnosis algorithm just needs answer "Yes" or "No".

Though our algorithm can accelerates the diagnosis by answering concrete instances (see Section 4.3), it needs just answer "Yes" or "No" in general due to the utilization of the previous result of program execution. In both Shapiro [10] and Lloyd [6], the diagnosis requests an oracle to instantiate variables to suitable forms, because of their definitions of wrong clause instances and uncovered atoms detected by their diagnosis algorithms.

Of course, recording all the trees is very space-consuming. We can reduce the necessary space at each instance by recording only the parts of the trees which are immediately necessary for the present, and by recomputing the parts which will be necessary later. Moreover we do not need all the information in such parts of the trees (see Section 4.1). For example, in the top-down zooming diagnosis, we do not need to immediately search atoms other than recursions so that the diagnoser needs to record only the relevant recursions, though such reduction of space may be inferior in the time efficiency due to the overhead for recomputation.

(3) Our diagnosis algorithm issues queries for the same predicate continually.

In general, it is easier and more natural for human programmers to answer queries for the same predicates continually. For recursive predicates, our approach jumps and omits some intermediate atoms with different predicates so that the queries for atoms with the same predicate are issued to the programmers continually. (Our concern is close to that of Eisenstadt [1].)

Moreover, such queries sometimes identify the segment containing bugs more quickly. Plaisted [9] showed a bug location algorithm more efficient than Shapiro's original algorithm.

His method selects nodes of trees (either proof trees or search trees), called *cutoffs*, in such a way that the execution time of each node distributes as uniformly w.r.t. the computation time as possible with some average interval, and roughly identify subcomputation containing a bug first, then apply his methods recursively to the subcomputation. Though our approach is not eager in uniformly distributing nodes for queries, the similar effect is obtained by leaping to immediate recursion subtrees.

Several problems still remain. One is that we have restricted our target program to terminating one. (See Kanamori[4] for detection of Prolog program termination.) Another problem is that we have restricted our target programming language to *pure* Prolog. The extra-logical control symbols like cut(!) and the predicates with side-effects like "assert" and "retract" are neglected. (These restrictions can be relaxed to a certain extent.)

6. Conclusions

We have shown a framework for top-down zooming diagnosis of logic programs. This method is an element of our system for analysis of Prolog programs Argus/A under development [2],[3],[4].

Acknowledgements

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

References

[1] Eisenstadt,E., "Retrospective Zooming : A Knowledge Based Tracing and Debugging Methodology for Logic Programming", Proc. of 9th International Joint Conference on Artificial Intelligence, pp.717–719, Los Angeles 1985.

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

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

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

[5] Lloyd, J. W., "Foundation of Logic Programming", Springer-Verlag, 1984.

[6] Lloyd, J. W., "Declarative Program Diagnosis", Technical Report 86/3, Department of Computer Science, University of Melbourne, 1986.

[7] Maeji,M. and T.Kanamori, "Top-down Zooming Diagnosis of Logic Programs", ICOT Technical Report TR-290, 1987.

[8] Pereira,L.M., "Rational Debugging in Logic Programming", Proc. of 3rd International Conference on Logic Programming, pp.203–210, 1986.

[9] Plaisted,D., "An Efficient Bug Location Algorithm", Proc. of 2nd International Logic Programming Conference, pp.151–157, 1984.

[10] Shapiro,E.Y., "Algorithmic Program Diagnosis", Conf. Rec. of the 9th ACM Symposium on Principles of Programming Languages, pp.299–308, 1984.