ICOT Technical Report: TR-290

TR-290

Top-down Zooming Diagnosis of Logic Programs

by
M. Maeji and T. Kanamori (Mitsubishi)

August, 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

Top-down Zooming Diagnosis of Logic Programs

Machi MAEJI Tadashi KANAMORI

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 examining the record of execution trace in some systematic manner, which corresponds to 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 [9], "declarative debugging" by Lloyd [6], and "rational debugging" by Pereira [7]. 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 examining the record of the execution trace in some systematic manner, which corresponds to 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 in Section 4, we will improve the diagnosis algorithm into the one in the "spy" manner. Last in Section 5, 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. Preliminaries

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 [9] 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 [9], 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" Command 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 Example of "trace"

Each line in a trace list is of the form

`(c) l Gate : A ,`

where `c` is the call number, `l` is the level number, `Gate` is "Call", "Redo", "Exit" or "Fail", and `A` is an atom. We call each line a *Call line*, a *Redo line*, an *Exit line* or a *Fail line* according to the `Gate` of the line. Their meanings are as follows:

(a) Call line

`(c) 1 Call : A` What is the first computed solution of A?

(b) Redo line

`(c) 1 Redo : Aθ` What is the computed solution next to $A\theta$?

(c) Exit line

`(c) 1 Exit : Aθ` The computed solution is $A\theta$.

(d) Fail line

`(c) 1 Fail : A` There is no other computed solution of A.

Let us collect all the lines with the same call number in a trace list. There are two types of such sequences.

Type 1 : A has the solutions $A\theta_1, A\theta_2, \ldots, A\theta_n$, and may have another solution ($n \geq 1$).

`(c) 1 Call : A`	What is the first computed solution of A?
`(c) 1 Exit : Aθ1`	The computed solution is $A\theta_1$.
`(c) 1 Redo : Aθ1`	What is the computed solution next to $A\theta_1$?
`(c) 1 Exit : Aθ2`	The computed solution is $A\theta_2$.
⋮	
`(c) 1 Redo : Aθn-1`	What is the computed solution next to $A\theta_{n-1}$?
`(c) 1 Exit : Aθn`	The computed solution is $A\theta_n$.

Type 2 : A has the solutions $A\theta_1, A\theta_2, \ldots, A\theta_n$, and they are the all solutions ($n \geq 0$).

`(c) 1 Call : A`	What is the first computed solution of A?
`(c) 1 Exit : Aθ1`	The computed solution is $A\theta_1$.
`(c) 1 Redo : Aθ1`	What is the computed solution next to $A\theta_1$?
`(c) 1 Exit : Aθ2`	The computed solution is $A\theta_2$.
⋮	
`(c) 1 Redo : Aθn`	What is the computed solution next to $A\theta_n$?
`(c) 1 Fail : A`	There is no other computed solution of A.

Each two consecutive lines in such a sequence is a pair of question and answer in its meaning. Call or Redo lines are questions, while Exit or Fail lines are answers. We call the question-answer pair a "`Call-Exit`" pair, a "`Redo-Exit`" pair, a "`Call-Fail`" pair, or a "`Redo-Fail`" pair according to the `Gates` of the lines composing the pair.

Definition 2.2.1 — *unit block* —

A subsequence of lines in a trace list is called a *unit block* with call number `c` and level number `l` when it consists of consecutive lines enclosed between the first line and the second line of a question-answer pair with call number `c` and level number `l` (including the first line and the second line themselves) in the trace list.

Definition 2.2.2 — *success block* —

A subsequence of lines in a trace list is called a *success block for* $A\theta$ when

(a) the sequence consists of all the unit blocks with call number `c` and level number `l` between a "Call" line and an "Exit" line such that these two lines have the same call number `c` and the level number `l`, and

(b) the atom of the "Exit" line is $A\theta$.

`c`, `l` and $A\theta$ are called the *call number*, the *level number* and the *label* of the success block, respectively.

Definition 2.2.3 — *failure block* —

A subsequence of lines in a trace list is called a *failure block for* A when

(a) the sequence consists of all the unit blocks with call number c and level number l between a "Call" line and a "Fail" line such that these two lines have the same call number c and the level number l, and

(b) the atom of the "Fail" line is A.

c, l and A are called the *call number*, the *level number* and the *label* of the failure block, respectively.

Definition 2.2.4 — *immediate success subblock* —

Let SB be a success block. A success block is called an *immediate success subblock* of SB when

(a) it is properly contained in SB, and

(b) it is not properly contained in any success block or any failure block satisfying (a).

Note that, when the level number of SB is l, the level number of the immediate success subblock of SB is $l+1$.

Definition 2.2.5 — *immediate failure subblock* —

Let FB be a failure block. A failure block is called an *immediate failure subblock* of FB when

(a) it is properly contained in FB, and

(b) it is not properly contained in any failure block satisfying (a).

Note that, when the level number of FB is l, the level number of the immediate failure subblock of FB is $l+1$.

Definition 2.2.6 — *composing success block of failure block* —

Let FB be a failure block with level number l. A success block is called a *composing success block* of FB when

(a) it is properly contained in FB, and

(b) the level number of the success block is l.

Definition 2.2.7 — *clause used for success block* —

Let SB be a success block with label A, and SB_1, SB_2, ..., SB_n be all the immediate success subblocks of SB with labels A_1, A_2, ..., A_n, respectively. Then, there exists a clause "B :- $B_1, B_2, \ldots, B_n$" in P such that $A \equiv B\theta_1\theta_2\cdots\theta_n$, $A_1 \equiv B_1\theta_1$, $A_2 \equiv B_2\theta_1\theta_2$, ..., $A_n \equiv B_n\theta_1\theta_2\cdots\theta_n$. The definite clause instance $(B$:- $B_1, B_2, \ldots, B_n)\theta_1\theta_2\cdots\theta_n$ is called the definite clause used for the block SB.

When the execution of an atom succeeds, there is a success block for the atom in its trace list. When the execution of an atom fails (after obtaining all computed solutions), there is a failure block for the atom in its trace list.

For Type 1 and Type 2 sequences, when A has some unintended computed solution $A\theta_i$ w.r.t. the intended interpretation, bugs should be found in the success block for $A\theta_i$. For Type 2 sequences, when A has some missed solutions w.r.t. the intended interpretation, that is, A should have a computed solution other than the computed solutions $A\theta_1, A\theta_2, \ldots, A\theta_n$, bugs should be found in the failure block for A. If there is no problem in both cases, the execution-result of A is correct so that it is unnecessary to check the trace list of the execution.

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

2.3 "spy" Command in DEC-10 Prolog

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 2.3 Example of "spy"

In Section 4, a top-down diagnosis algorithm in the "spy" manner is going to be developed for systematizing the process of examining specific predicates.

3. Top-down Diagnosis of Logic Programs

In this section, we will first present a top-down diagnosis algorithm using the terminology of trace lists. Next, we will re-present the top-down diagnosis algorithm using the notions of "proof tree" and "search tree". Last, we will show soundness and completeness of the top-down diagnosis algorithm.

3.1 A Top-down Diagnosis Algorithm Using Traces

(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.

(5) A Top-down Diagnosis Algorithm

We will show a top-down diagnosis algorithm using the terminology of trace lists. The top-down diagnosis algorithm "*diagnose*0" receives a block (either a success block or a failure block), and returns a definite clause instance, an atom, or a message "no bug is found". We assume here that a trace list including the block is completely recorded in a "trace database".

diagnose0(BL : block) : bug-message ;
  when BL is a success block with label A
    issue a query "Is some instance of A false?"
    if the answer is "No"
    then return "no bug is found"
    else let SB_1, SB_2, ..., SB_n be the immediate success subblocks of BL;
      if the application of "*diagnose*0" to some SB_j returns a bug
      then return it
      else return the definite clause used for BL as a bug
  when BL is a failure block with label A
    let BL_1, BL_2, ..., BL_k be the composing success blocks of BL;
    if the application of "*diagnose*0" to some BL_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 FB_1, FB_2, ..., FB_n be the immediate failure subblocks of BL;
        if the application of "*diagnose*0" to some FB_j returns a bug
        then return it
        else return the atom A as a bug

Figure 3.1 Top-down Diagnosis Algorithm Using Subblocks

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 [9].)

Example 3.1.5 Let us diagnose an atom $qsort([2,1],X)$ in the program of Example 2.1.1. In the following, "*diagnose*0" takes the label of each block as an argument instead of the block itself. (See section 5.1 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 3.1.6 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
```

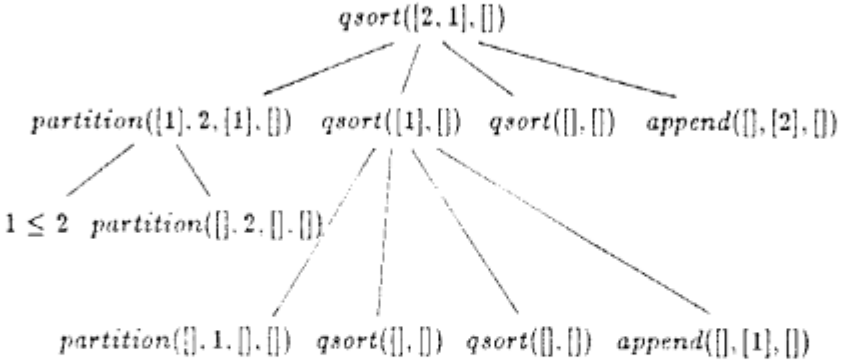
3.2 Top-down Diagnosis Algorithm Using Trees

A top-down diagnosis algorithm has been given in the previous section 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. In this section, we will give the top-down diagnosis algorithm over again 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 3.2.1 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

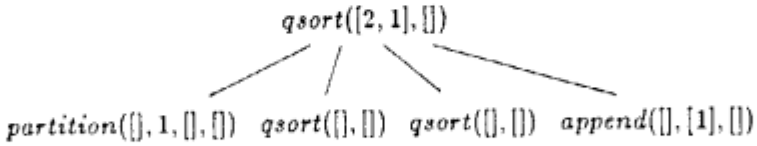
A subtree of a proof tree T is called a *proof subtree* of T. In particular, a proof subtree is called an *immediate proof subtree* of T when

(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 3.2.2 The following is an immediate proof subtree of the proof tree of Example 3.2.1.



(3) Correspondence to Success Block.

Let SB be a success block for A, and T be a proof tree of A. Then immediate success subblocks of SB corresponds to immediate proof subtrees of T, though immediate success subblocks might contain lines irrelevant to the finally constructed proof tree, that is, the lines corresponding to search which has turned out unnecessary after backtracking.

Example 3.2.3 In Figure 2.2.1, there is a success block for "$qsort([2,1],[])$" as below:

(1) 0 Call : qsort([2,1],X)

⋮

(1) 0 Exit : qsort([2,1],[ ]).

The proof tree of "$qsort([2,1],[])$" shown in Example 3.2.1 is constructed from the "Exit" lines of success subblocks in it.

(4) 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 3.2.4 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:

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

$$| \quad \{\}$$

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

$$| \quad \{\}$$

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

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

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

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

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

$$| \quad \{X \Leftarrow [2,1]\}$$

$$\square$$

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.)

(5) 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 $U_1, U_2, \ldots, U_h$ 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 $U_1, U_2, \ldots, U_h$ 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 3.2.5 The tree below is an immediate search subtree of the search tree of Example 3.2.3. (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.

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

$$| \quad \{\}$$

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

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

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

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

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

(6) Correspondence to Failure Block

Let FB be a failure block for an atom A, and T be a search tree of A. Then immediate failure subblocks of FB corresponds to immediate search subtrees of T, while the composing success blocks of FB corresponds to success paths in T.

Example 3.2.6 When $perm([2,1], X)$ is executed in the program of Example 2.1.2, there is a failure block for "$perm([2,1], X)$" as below:

```
(1) 0 Call : perm([2,1],X)
  :
(1) 0 Exit : perm([2,1],[2,1])
(1) 0 Redo : perm([2,1],[2,1])
  :
(1) 0 Fail : perm([2,1],X).
```

The search tree of "$perm([2,1], X)$" is shown in Example 3.2.3.

(7) The Top-down Diagnosis Algorithm Using Subtrees

The top-down diagnosis algorithm using proof trees and search trees receives a tree (either a proof tree or a search tree of an atom), and returns a bug message as before.

```
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 "diagnose0" 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 "diagnose0" 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 "diagnose0" 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

3.3 Soundness and Completeness of the Top-down Diagnosis Algorithm

In this section, we will prove soundness and completeness of the top-down diagnosis algorithm using the notions "proof subtree" and "search subtree".

(1) Completeness of the Top-down Diagnosis Algorithm

To show completeness of the top-down diagnosis algorithm, we need the following lemma.

Lemma 3.3.1 *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 an atom A in P is correct w.r.t. M, "diagnose0" applied to T returns no bug.*

Proof: We will trace the behavior of "*diagnose*0" for each case.

When the tree T is a proof tree, the diagnosis proceeds as follows: First, the query "Is some instance of A false?" is issued. Because the execution-result of A is correct w.r.t. M, and A is a computed solution of itself, the atom A is valid in M, so that the answer for this query is "No". Hence, "*diagnose*0" applied to T returns no bug, and stops.

When the tree T is a search tree, let $T_1, T_2, \ldots, T_k$ be the proof trees corresponding to success paths in T with root nodes $A_1, A_2, \ldots, A_k$ respectively. First, for every i $(1 \leq i \leq k)$, the query "Is some instance of A_i false?" is issued. Because the execution-result of A is correct w.r.t. M, and $A_1, A_2, \ldots, A_k$ are the computed solutions of A, the atoms $A_1, A_2, \ldots, A_k$ are valid in M, so that all the answers for these queries are "No". Hence, "*diagnose*0"

applied to any T_i returns no bug. Next, the query "Is some other instance of A true?" is issued. Because the execution-result of A is correct w.r.t. M, A has no missed solution, so that the answer is also "No". Hence, "*diagnose*0" of A returns no bug, and stops.

First, we will show completeness of the top-down diagnosis algorithm for proof trees.

Theorem 3.3.1 *(completeness of the top-down diagnosis algorithm for proof tree)*
Let P be a terminating program, M be an intended interpretation, and T be a proof tree in P. When the execution-result of the root label of T in P is incorrect w.r.t. M, diagnosis0 applied to T returns a definite clause as a bug.

Proof: Because P is a terminating program, the length of every branch in T is finite. The depth of T is defined by the length of the longest branch in T, and denoted by $depth(T)$. We will prove the theorem by induction on $depth(T)$.

Let A be the root label of T. Because the execution-result of A in P is incorrect, and A is a computed solution of itself, the answer for a query "Is some instance of A false?" is "Yes" . (A never has a missed solution.) Let $PT_1, PT_2, \ldots, PT_n$ be all the immediate proof subtrees of T.

If the execution-result of root label of some PT_j $(1 \leq j \leq n)$ is incorrect w.r.t. M, because of $depth(PT_j) < depth(T)$ and induction hypothesis, "*diagnose*0" of PT_j returns a definite clause as a bug.

If the execution-result of root label of each PT_j $(1 \leq j \leq n)$ is correct w.r.t. M, "*diagnose*0" applied to each PT_j does not returns a bug due to Lemma 3.3.1. Hence, "*diagnose*0" applied to PT returns a definite clause as a bug.

Next, we will show completeness of the top-down diagnosis algorithm for search trees.

Theorem 3.3.2 *(completeness of the top-down diagnosis algorithm for search tree)*
Let P be a terminating program, M be an intended interpretation, and T be a search tree of an atom A in P. When the execution-result of A in P is incorrect w.r.t. M, "diagnose0" applied to T returns either a definite clause or an atom as a bug.

Proof: Because P is a terminating program, the length of every branch in T is finite. We will prove the theorem by induction on the depth of T $depth(T)$.

Let $T_1, T_2, \ldots, T_k$ be the proof trees corresponding to success paths in T with root labels $A_1, A_2, \ldots, A_k$ respectively. Then the atoms $A_1, A_2, \ldots, A_k$ are all the computed solutions of A in P.

If the execution-result of some A_i $(1 \leq i \leq k)$ in P is incorrect, "*diagnose*0" applied to some T_i returns a definite clause as a bug due to Theorem 3.3.1.

If the execution-result of any A_i in P is correct w.r.t. M, "*diagnose*0" applied to each T_i returns no bug due to Lemma 3.3.1. Because the execution-result of A in P is incorrect, A has some missed solution, so that the answer for a query "Is some other instance of A true?" is "Yes". Let $ST_1, ST_2, \ldots, ST_n$ be all the immediate search subtrees of T. If the execution-result of root label of some ST_j $(1 \leq j \leq n)$ is incorrect, because of $depth(ST_j) < depth(T)$ and induction hypothesis, "*diagnose*0" applied to ST_j returns either a definite clause or an atom as a bug. If the execution-result of root label of each ST_j $(1 \leq j \leq n)$ is correct, "*diagnose*0" applied to each ST_j returns no bug due to Lemma 3.3.1. Hence "*diagnose*0" applied to T returns the atom A as a bug.

(2) Soundness of the Top-down Diagnosis Algorithm

By taking the contrapositive of Lemma 3.3.1, it is obvious that, when "*diagnose*0" applied to a tree (either a proof tree or a search tree) of an atom A returns a bug, the execution-result of the atom A in P is incorrect w.r.t. M. However, what we would like to show is that the bug returned by "*diagnose*0" is indeed either a wrong clause instance or an uncovered atom.

First, we will show soundness of the top-down diagnosis algorithm for proof trees. To show this, we need the following lemma.

Lemma 3.3.2 *Let P be a program, M be an intended interpretation, A be an atom which has succeeded unexpectedly w.r.t. M, and T be the proof tree of the atom A in P. If the execution-results of root labels of all the immediate proof subtree are correct w.r.t. M, the clause used at the root of T is a wrong clause instance w.r.t. M.*

Proof: Suppose that the clause used at the root is not a wrong clause instance. Because A has succeeded unexpectedly, the atom A is invalid in M. Let A_1, A_2, ..., A_n be the root labels of all the immediate proof subtrees. Because the execution-results of A_1, A_2, ..., A_n are correct w.r.t. M, and A_1, A_2, ..., A_n are computed solutions of themselves, all the atoms A_1, A_2, ..., A_n are valid in M. Since A_1, A_2, ..., A_n are child atoms of A, this fact contradicts the fact that "A :- A_1, A_2, ..., A_n" is not a wrong clause instance. (cf. Shapiro [9], Lloyd [6]).

Theorem 3.3.3 *(soundness of the top-down diagnosis algorithm for proof tree)*
Let P be a program, M be an intended interpretation, A be an atom which has succeeded unexpectedly w.r.t. M, and T be the proof tree of the atom A in P. When "diagnose0" applied to T returns a definite clause as a bug, the definite clause is a wrong clause instance w.r.t. M.

Proof: When "*diagnose*0" returns a definite clause, there must exist a proof tree T such that the root label of T has succeeded unexpectedly, and "*diagnose*0" applied to the immediate proof subtrees return no bug. Because the execution-result of an atom is correct w.r.t M if and only if "*diagnose*0" applied to the atom returns no bug from Lemma 3.3.1 and Theorem 3.3.1, the clause used at the root of T is a wrong clause instance w.r.t. M due to Lemma 3.3.2.

Next, we will show completeness of the top-down diagnosis algorithm for search trees. To show this, we need the following lemma.

Lemma 3.3.3 *Let P be a terminating program, M be an intended interpretation, A be an atom which has failed unexpectedly w.r.t. M, and T be a search tree of the atom A in P. If the execution-results of root labels of all the immediate search subtrees in T are correct w.r.t. M, the atom A is an uncovered atom w.r.t. M.*

Proof: Suppose that the atom A is not an uncovered atom w.r.t. M. Because A has failed unexpectedly, there is some missed solution $A\theta$, which is a ground instance of A and true in M. Because A is not an uncovered atom, there is a ground instance of a definite clause "B :- $B_1, B_2, \ldots, B_n$" in P by a substitution θ' such that $B\theta'$ is identical to $A\theta$ and $(B_1, B_2, \ldots, B_n)\theta'$ is true in M. Moreover, because B is unifiable with A, let σ_0 be its m.g.u. Then there is a substitution λ such that $\sigma_0\lambda = \theta'$ holds.

Let ν_0 be the root node of T. From the definition of search trees, there is an immediate subtree of T with root ν_1 labelled with $(B_1, B_2, \ldots, B_n)\sigma_0$. Let ST_1 be the search subtree of T at ν_1 with root label $B_1\sigma_0$. Because there is no node between ν_0 and ν_1 in T, and the

search subtree of ν_0 is T itself, ST_1 is an immediate search subtree of T. Because $B_1\theta'$ is true in M and the execution-result of $B_1\sigma_0$ is correct, $B_1\theta' = B_1\sigma_0\lambda$ is an instance of a computed solution of $B_1\sigma_0$ in P. Hence, there is a success path in ST_1 corresponding to such a solution, say $B_1\sigma_1$. The node, say ν_2, in T corresponding to the node labelled with the empty atom sequence $\square$ in ST_1 is labelled with $(B_2, B_3, \ldots, B_n)\sigma_1$. For any node ν on the path between ν_0 and ν_2 other than ν_0 and ν_2 in T, $length(\nu) \geq length(\nu_1) = n > n - 1 = length(\nu_2)$ holds. Hence, there is no search subtree at ν properly containing the search tree at ν_2, i.e., the search subtree at ν_2 is an immediate search subtree of T.

Similarly, the following holds for every i $(1 \leq i \leq n)$: Suppose that the atom sequence $(B_i, B_{i+1}, \ldots, B_n)\theta'$ is an instance of $(B_i, B_{i+1}, \ldots, B_n)\sigma_{i-1}$, which is the label of the node ν_i corresponding to the root node of an immediate search subtree of T. Then the atom $B_i\theta'$ is an instance of a computed solution, say $B_i\sigma_i$, of $B_i\sigma_{i-1}$ in P, and the atom sequence $(B_{i+1}, B_{i+2}, \ldots, B_n)\theta'$ is an instance of $(B_{i+1}, B_{i+2}, \ldots, B_n)\sigma_i$, which is again the label of the node ν_{i+1} corresponding to the root node of an immediate search subtree of T.

Hence $(B_1, B_2, \ldots, B_n)\theta'$ is an instance of a computed solution of $(B_1, B_2, \ldots, B_n)\sigma_0$. That is, $A\theta$ is an instance of a computed solution of A in P. This fact contradicts the fact that $A\theta$ is a missed solution of A.

Theorem 3.3.4 *(soundness of the top-down diagnosis algorithm for search tree)*
Let P be a program, M be an intended interpretation, A be an atom which has failed unexpectedly w.r.t. M, and T be the search tree of the atom A in P. When "diagnose0" applied to T returns either a definite clause or an atom as a bug, the definite clause is a wrong clause instance w.r.t. M, and the atom is an uncovered atom w.r.t. M.

Proof: We will prove the theorem by case analysis.

When "*diagnose*0" returns a definite clause, it must be returned by some "*diagnose*0" applied to a proof tree. Hence, the clause is a wrong clause instance w.r.t. M due to Theorem 3.3.3.

When "*diagnose*0" returns an atom A, there must exist a search tree T such that the root label of T has failed unexpectedly, and "*diagnose*0" applied to the immediate search subtrees return no bug. Because the execution-result of A is correct w.r.t M if and only if "*diagnose*0" applied to A returns no bug from Lemma 3.3.1 and Theorem 3.3.2, the root label of T is an uncovered atom w.r.t. M due to Lemma 3.3.3.

(3) Soundness and Completeness of the Top-down Diagnosis Algorithm

The following theorem is an immediate consequence of Lemma 3.3.1 and Theorem 3.3.1, 3.3.2, 3.3.3 and 3.3.4.

Corollary 3.3 *(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.

4. Top-down Zooming Diagnosis of Logic Programs

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.

4.1 A Top-down Zooming Diagnosis Algorithm Using Traces

Let SB be a success block whose label is with predicate p. A success block in T is called a *recursion success subblock* in SB when its label is with predicate p.

Definition 4.1.1 — *immediate recursion success subblock* —
Let SB be a success block whose label is with predicate p. A success block is called an *immediate recursion success subblock* of SB when
(a) it is properly contained in SB,
(b) the label of the success block is with predicate p, and
(c) it is not properly contained in any success block satisfying (a) and (b) or any failure block satisfying (a).

Let FB be a failure block whose label is with predicate p. A failure block in FB is called a *recursion failure subblock* in FB when its label is with predicate p.

Definition 4.1.2 — *immediate recursion failure subblock* —
Let FB be a failure block whose label is with predicate p. A failure block is called an *immediate recursion failure subblock* of FB when
(a) it is properly contained in FB,
(b) the label of the failure block is with predicate p, and
(c) it is not properly contained in any failure block satisfying (a) and (b).

The top-down zooming diagnosis algorithm 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 block and returns a subblock for diagnosis by searching for recursion subblocks. Again, we assume here that a trace list including the block is completely recorded in a "trace database".

```
diagnose(BL : block) : bug-message ;
    if the application of "zooming" to BL does not return a tree
    then return "no bug is found"
    else let BL' be the returned block;
        when BL' is a success block with label A
          let SB_1, SB_2, ..., SB_n be the immediate success subblocks of BL';
          if the application of "diagnose" to some SB_j returns a bug
          then return it
          else return the definite clause used for BL' as a bug
        when BL' is a failure block with label A
          let FB_1, FB_2, ..., FB_n be the immediate failure subblocks of BL';
          if the application of "diagnose" to some FB_j returns a bug
          then return it
          else return the atom A as a bug

zooming(BL : block) : block ;
    when BL is a success block with label A
      issue a query "Is some instance of A false?"
```

if the answer is "No"
then return "no block is found"
else let SB_1, SB_2, ..., SB_n be the immediate recursion success subblocks of BL;
if the application of "*zooming*" to some SB_j returns a block
then return it
else return BL

when BL is a failure block with label A
let BL_1, BL_2, ..., BL_k be the composing success blocks of BL;
if the application of "*zooming*" to some BL_i returns a block
then return it
else issue a query "Is some other instance of A true?"
if the answer is "No"
then return "no block is found"
else let FB_1, FB_2, ..., FB_n be the immediate recursion failure subblocks of BL;
if the application of "*zooming*" to some FB_j returns a block
then return it
else return BL

Figure 4.1 Top-down Zooming Diagnosis Algorithm Using Recursion Subblocks

Roughly speaking, the top-down zooming diagnosis algorithm identifies the subblocks 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 subblocks quickly by changing "*diagnose*0" to possibly leap the intermediate subblocks. "*zooming*" above detects a recursion subblock such that the execution-result of its label is incorrect but the execution-result of label of every immediate recursion subblock is correct w.r.t. M.

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

Example 4.1.1 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 4.1.2 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

4.2 Top-down Zooming Diagnosis Algorithm Using Trees

Similarly to Section 3.2, we will give the top-down zooming diagnosis algorithm over again using the terminology of "proof tree" and "search tree".

(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 4.2.1 In the proof tree of Example 3.2.1, 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 4.2.2 In the search tree of Example 3.2.3, 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 receives a tree (either a proof tree or a search tree of an atom), and returns a bug message as before.

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 $PT_1, PT_2, \ldots, PT_n$ 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 4.2 Top-down Zooming Diagnosis Algorithm Using Recursion Subtrees

4.3 Soundness and Completeness of the Top-down Zooming Diagnosis Algorithm

In this section, we will prove soundness and completeness of the top-down zooming diagnosis algorithm using the notions of "immediate proof subtree" and "immediate search subtree".

(1) Soundness and Completeness of Zooming

Note that the algorithm "*zooming*" is almost the same as the top-down diagnosis algorithm "*diagnose*0" except only the following two points:

(a) "*zooming*" is applied recursively to "immediate recursion (proof or search) subtree", while "*diagnose*0" is applied recursively to "immediate (proof or search) subtree".

(b) "*zooming*" returns a tree, while "*diagnose*0" returns a bug message.

Hence, similarly to Corollary 3.3, the following lemma holds.

Lemma 4.3.1 *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 is incorrect w.r.t. M, if and only if "zooming" applied to the tree T returns a tree*

such that the execution-result of its root label is incorrect w.r.t. M, and the execution-result of any root label of its immediate recursion subtrees is correct w.r.t. M.

(2) Completeness of the Top-down Zooming Diagnosis Algorithm

"*diagnose*" is similar to "*diagnose*()" as well, except that the "··· **if** ··· **then** ··· **else**" part is omitted from the first **when** statement, and "··· **if** ··· **then** ··· **else** ··· **if** ··· **then** ··· **else**" part is omitted from the second **when** statement. This modification is superficial just for saving cost of computation common to "*zooming*" and "*diagnose*", because T' in "*diagnose*" is returned by "*zooming*".

(a) When T' is a proof tree with root label A, the answer for a query "Is some instance of A false?" is always "Yes".

(b) When T' is a search tree with root label A, let $T_1, T_2, \ldots, T_k$ be the proof trees corresponding to success paths in T. Then the application of "*diagnose*" (hence the application of "*zooming*" inside it) to any T_i returns "no bug is found". Moreover the answer for a query "Is some instance of A true?" is always "Yes".

Lemma 4.3.2 *Let P be a terminating program and M be an intended interpretation. When the execution-result of the atom A in P is correct w.r.t. M, "diagnose" applied to a tree (either a proof tree or a search tree) of an atom A returns no bug.*

Proof: Suppose that the execution-result of the atom A in P is correct w.r.t. M. Then the application of "*zooming*" to T returns "no tree is found" because of soundness of "*zooming*". Hence, "*diagnose*" for the atom A returns "no bug is found".

Theorem 4.3.1 *(completeness of the top-down zooming diagnosis algorithm for proof tree)* *Let P be a terminating program, M be an intended interpretation, and T be a proof tree in P. When the execution-result of the root label of T in P is incorrect w.r.t. M, the top-down zooming diagnosis of T returns a definite clause as a bug.*

Proof: Because P is a terminating program, the length of every branch in T is finite. We will prove the theorem by induction on $depth(T)$.

Because the execution-result of the root label of T is incorrect, "*zooming*" returns a tree due to Lemma 4.3.1. Let T' be the tree. It is a proof subtree of T. Let $PT_1, PT_2, \ldots, PT_n$ be all the immediate proof subtrees of T'.

If the execution-result of root label of some PT_j $(1 \leq j \leq n)$ is incorrect w.r.t. M, because of $depth(PT_j) < depth(T') \leq depth(T)$ and induction hypothesis, "*diagnose*" applied to PT_j returns a definite clause as a bug.

If the execution-result of root label of each PT_j $(1 \leq j \leq n)$ is correct w.r.t. M, "*diagnose*" applied to each PT_j returns no bug due to Lemma 4.3.2. Hence, "*diagnose*" applied to T returns a definite clause as a bug.

Theorem 4.3.2 *(completeness of the top-down zooming diagnosis algorithm for search tree)* *Let P be a terminating program, M be an intended interpretation, and T be a search tree of an atom A in P. When the execution-result of the root label of T in P is incorrect w.r.t. M, "diagnose" applied to T returns either a definite clause or an atom as a bug.*

Proof: Because P is a terminating program, the length of every branch in T is finite. We will prove the theorem by induction on the depth of T $depth(T)$.

Because the execution-result of root label of T in P is incorrect, "*zooming*" returns a tree due to Lemma 4.3.1. Let T' be the tree. Since "*zooming*" applied to T' also returns T', "*diagnose*" applied to T is the same as that to T'.

If T' is a proof tree, its root label has succeeded unexpectedly so that "*diagnose*" applied to T' returns a definite clause as a bug.

If T' is a search tree, T' is a search subtree of T. Let ST_1, ST_2, ..., ST_n be the immediate search subtrees of T'. If the execution-result of root label of some ST_j $(1 \leq j \leq n)$ is incorrect w.r.t. M, because of $depth(ST_j) < depth(T') \leq depth(T)$ and induction hypothesis, "*diagnose*" applied to ST_j returns either a definite clause or an atom as a bug. If the execution-result of root label of each ST_j $(1 \leq j \leq n)$ is correct w.r.t. M, "*diagnose*" applied to each ST_j returns no bug due to Lemma 4.3.2. Hence, "*diagnose*" applied to T' returns the root label as a bug.

(3) Soundness of the Top-down Zooming Diagnosis Algorithm

Soundness of the top-down zooming diagnosis algorithm is proved similarly to the previous top-down diagnosis algorithm without zooming.

Theorem 4.3.3 *(soundness of the top-down zooming diagnosis algorithm for proof tree)*
Let P be a program, M be an intended interpretation, A be an atom which has succeeded unexpectedly w.r.t. M, and T be the proof tree of the atom A in P. When "diagnose" applied to T returns a definite clause as a bug, the definite clause is a wrong clause instance w.r.t. M.

Proof: It is proved in the completely same way as Theorem 3.3.3.

Theorem 4.3.4 *(soundness of the top-down zooming diagnosis algorithm for search tree)*
Let P be a program, M be an intended interpretation, A be an atom which has failed unexpectedly w.r.t. M, and T be the search tree of the atom A in P. When "diagnose" applied to T returns either a definite clause or an atom as a bug, the definite clause is a wrong clause instance w.r.t. M, and the atom is an uncovered atom w.r.t. M.

Proof: It is proved in the completely same way as Theorem 3.3.4.

(4) Soundness and Completeness of the Top-down Zooming Diagnosis Algorithm

The following theorem is an immediate consequence of Lemma 4.3.2 Theorem 4.3.1, 4.3.2, 4.3.3 and 4.3.4.

Corollary 4.3 *(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.

5. 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.

5.1 Consideration on Space Efficiency

In Section 4.1, we have assumed that all the trace lists of the execution are recorded in a "trace database", and are used somehow for processing success blocks and failure blocks, which are passed as arguments of "*diagnose*" and "*zooming*". However, recording all the

trace lists is very space-consuming. Recall how the blocks are used in the diagnosis. They are used only when

(a) "*diagnose*" recurses with an immediate subblock,
(b) "*zooming*" recurses with some block, either an immediate recursion subblock or an composing block, or
(c) "*zooming*" issues a query about the label of the block.

So, if we can obtain the labels of any immediate subblock, any immediate recursion subblock, and any composing block somehow, it is enough for the diagnosis. Let A be a label of a block.

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

(a) When A is a label of a success block SB, let A_1, A_2, ..., A_n be the labels of all immediate success subblocks of SB. Then there is a clause "B:-$B_1, B_2, \ldots, B_n$" in P such that $A_1 \equiv B_1\theta_1$, $A_2 \equiv B_2\theta_1\theta_2$, ..., $A_n \equiv B_n\theta_1\theta_2\cdots\theta_n$, and $A \equiv B\theta_1\theta_2\cdots\theta_n$ hold. On the other hand, if there is a clause "B:-$B_1, B_2, \ldots, B_n$" in P such that $A_1 \equiv B_1\theta_1$, $A_2 \equiv B_2\theta_1\theta_2$, ..., $A_n \equiv B_n\theta_1\theta_2\cdots\theta_n$ hold for some labels A_1, A_2, ..., A_n of success blocks in the "trace database", then $B\theta_1\theta_2\cdots\theta_n$ should succeed in P using the clause instance "$(B$:-$B_1, B_2, \ldots, B_n)\theta_1\theta_2\cdots\theta_n$" in P. Hence, we may conclude that there is a success block for $B\theta_1\theta_2\cdots\theta_n$ in the "trace database" such that the labels of all its immediate success subblocks are the atoms A_1, A_2, ..., A_n.
(b) When A is a label of a failure block FB, let A' be a label of an immediate failure subblock of FB. 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 "$(A_1, A_2, \ldots, A_{i-1})\sigma$" $(1 \leq i < n)$. On the other hand, all the atoms constructed by such a way are the labels of immediate failure subblocks of FB.

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

The labels of composing success blocks of a failure block 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 labels of composing success blocks.

Hence, the labels of subblocks are all obtained by using the clauses in P and the recorded labels of blocks.

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

(a) the labels of success blocks, and
(b) the pairs of the label of an failure block, and the sequence of the labels of its composing success blocks.

(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 blocks but labels of the blocks.

5.2 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 labels of the trace list which is immediately necessary for the present, and by recomputing another labels which will become necessary later. In the top-down zooming diagnosis, we do not need to immediately search labels of blocks other than recursion subblocks

so that the diagnoser needs to record only the labels of the recursion subblocks relevant for the present. The computation of the labels of blocks other than recursion subblocks 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 labels of success blocks, that are used before for the diagnosis, during each re-execution to avoid some of the recomputation.

5.3 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 a oracle), and the answer is added to the "answer database". (See Shapiro [9].)

(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 labels of every immediate recursion subblocks (or every immediate subblocks) can be issued together. Similarly, in the diagnosis for unexpected failure, the queries for the labels of composing success blocks can be issued together.

Example 5.3.1 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 5.3.2 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 [9] 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 5.3.3 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.

Suppose that the answer "Unknown" is returned. First, the diagnoser interprets this answer as "Yes", and continues the following diagnosis as usual. If no bug is found there, the diagnoser informs the programmer of it, and asks again to return the reserved answer. If the answer is still "Unknown", nevertheless, this point is recorded as a candidate for a bug of the unexpected result. Next, the diagnoser interprets this answer as "No", and continues the following diagnosis as usual. A bug is selected out of the bug found there and the recorded candidates.

6. Discussion

Debugging of logic programs has been studied by several researchers intensively. Shapiro [9] 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 [9], Lloyd [6], et al. However, *wrong clause instances* and *uncovered atoms* in our definitions are slightly different from those in Shapiro [9] 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

(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 Herbraud 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 [9] 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 5.3), it needs just answer "Yes" or "No" in general due to the utilization of the previous result of program execution. In both Shapiro [9] 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 trace lists is very space-consuming. We can reduce the necessary space at each instance by recording only the parts of the trace lists 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 trace lists (see Section 5.1). 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 [8] showed a bug location algorithm more efficient than Shapiro's original algorithm. His methods 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.)

7. 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 (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., K.Horiuchi and T.Kawamura, "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] Pereira,L.M., "Rational Debugging in Logic Programming", Proc. of 3rd International Conference on Logic Programming, pp.203–210, 1986.

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

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