# Use of Correctness Assertions in Declarative Diagnosis

Lunjin Lu
Department Of Computer Science and Engineering
Oakland University
Rochester, Michigan 48309
lunjin@acm.org

## ABSTRACT

We use assertions to reduce the quantity of queries in declarative diagnosis of logic programs. We first present a declarative diagnoser for normal logic programs. Given a bug symptom, the diagnoser first constructs a tree that models the execution of the bug symptom and then searches the tree for the bug that causes the bug symptom. We then incorporate into the diagnoser three tree transformations that prune the tree before it is searched. These transformations make use of two kinds of assertion about the correctness of the program and maintain the soundness and completeness of the diagnoser. These transformations reduce the size of the tree and thus reduce the quantity of queries imposed on the oracle.

**Keywords**: Declarative diagnosis; Correctness assertions; Logic programs

## 1. INTRODUCTION

Declarative program diagnosis is an interactive process where a declarative diagnoser obtains the intended interpretation of the program from an oracle, usually the programmer, and compares the intended interpretation with the actual interpretation of the program. The declarative diagnosis was proposed for logic programs by Shapiro [17, 18] and has been since adapted for other programming paradigms [6, 13, 12, 11, 16].

A buggy logic program may exhibit many kinds of bug symptom. It may produce a wrong answer, fail to produce a correct answer, or loop and so on. This paper is concerned with the first two kinds of bug symptom. Many declarative diagnosers for logic programs have been developed. Shapiro [17, 18] developed the algorithmic debugging method[1] and exemplified the method through pure Prolog. Ferrand [5] adapted the algorithmic debugging method for definite logic programs. Lloyd [8, 9] presented a declarative diagnoser for arbitrary logic programs. The diagnoser is a meta-program which makes it easy to improve its performance by adding control information as meta-calls. Lloyd [8, 9] obtained a top-down diagnoser by adding control information. Yan [19] improved the top-down diagnoser by reorganising its control information.

The main advantage of using a declarative diagnoser is that the oracle does not need to know anything about the operational aspect of the program. All that they need to know is the intended interpretation of the program. The quantity of the queries imposed on the oracle may be large and reducing the quantity of the queries has been the main objective of much research into declarative diagnosis [4, 2, 3, 14, 15]. The quantity of the queries is dependent on the size of the search space and the search strategy [18, 9].

This paper formalises the programmer's practice of using assertions about the correctness of the program and uses two kinds of correctness assertion to reduce the size of the search space of a declarative diagnoser for normal logic programs [10]. We first recall the original declarative diagnoser and then present an improved declarative diagnoser that makes use of correctness assertions to reduce the quantity of queries.

The remaider of the paper is organised as follows. Section 2 defines bugs and bug symptoms and establishes the connection between a bug symptom and a bug by means of a tree that models the execution of the bug symptom. Section 3 presents the original diagnoser that constructs and searches the tree. In section 4, we formalise the two kinds of correctness assertion and present the improved diagnoser. The improved diagnoser reduces the quantity of queries by reducing the size of the search space. It does so by applying three tree pruning transformations that make use of the two kinds of correctness assertion. Section 5 concludes the paper and compares our work with other work on the use of assertions in declarative logic program diagnosis.

Let $T$ be a tree and $v$ a node of $T$. The height of $T$, written as $h(T)$, is the length of the longest path of $T$. $T_v$ denotes the sub-tree of $T$ that is rooted at $v$. We use $T_{/v}$ to denote the tree resulting from replacing $T_v$ of $T$ with a single node $v$. Let $v_1, \cdots, v_m$ be the children of $v$, we write

[1]We use the term diagnosis instead of debugging because debugging is a process which involves bug detection and bug correction as well as bug diagnosis.



SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.

$T_v = t(v, \{T_{v_1}, \cdots, T_{v_m}\})$. Let $T_1$ be a sub-tree of $T$ and $T_2$ a tree. We write $T[T_1 \mapsto T_2]$ to denote the tree resulting from substituting $T_2$ for $T_1$.

## 2. SYMPTOMS, BUGS AND EXECUTION

We assume that readers are familiar with the terminology of logic programming [9]. To simplify presentation, we use the left-to-right computation rule. Adaptation to other computation rules is straightforward.

### 2.1 Symptoms and Bugs

Let $P$ be the program to diagnose and $I$ its intended interpretation. An atom $A$ is a wrong answer if $A$ is invalid in $I$ and $A \in SS(P)$ where $SS(P)$ is the success set of $P$. An atom $A$ is a missing answer if $A$ is satisfiable in $I$ and $P$ finitely fails on $A$. A clause instance $A \leftarrow W$ is inconsistent if $A \leftarrow W$ is invalid in $I$. An atom $A$ is uncovered if $A$ is valid in $I$ and, for every clause $A' \leftarrow W$ of $P$ such that $A$ and $A'$ unify with `mgu` $\theta$, $W\theta$ is unsatisfiable in $I$. An atom $A$ is incompletely covered if an instance of $A$ is uncovered. If there is a wrong or missing answer then there is an inconsistent clause instance or an incompletely covered atom [9, 8, 5, 19]. Therefore, given a wrong or missing answer, a declarative diagnoser for logic programs is to search for an inconsistent clause instance or an incompletely covered atom.

### 2.2 Execution Trees

We now establish the relation between a symptom and a bug by means of an execution tree. Consider wrong answers first. Let $L$ be a node of an ordered tree $T$ and $L_1, L_2, \cdots, L_k$ the children of $L$ in that order. We say that $L \leftarrow L_1, L_2, \cdots, L_k$ is the root implication of $T_L$ and write it as $RI(T, L)$. A partial proof tree for an atom $A$ is rooted at $A$ and, for each non-leaf node $L'$, either $RI(T, L')$ is an instance of a clause or $RI(T, L') = (\neg A' \leftarrow \texttt{true})$ and $A'$ is an atom on which $P$ finitely fails. A partial proof tree is congruent if each of its leaves is valid in $I$. A proof tree is a partial proof tree whose leaves are all labelled with `true`. Observe that A proof tree is a congruent partial proof tree (*CPP* for short) since `true` is valid in $I$.

Let $P$ be the following quicksort program that has a bug indicated by a comment.

```
qs([X|L],L0) :- pt(L,X,L1,L2), qs(L1,L3), qs(L2,L4),
                ap([X|L3],L4,L0). %ap(L3,[X|L4],L0)
qs([],[]).

pt([X|L],Y,L1,[X|L2]) :- Y=<X, pt(L,Y,L1,L2).
pt([X|L],Y,[X|L1],L2) :- Y>X,  pt(L,Y,L1,L2).
pt([],X,[],[]).

ap([X|L1],L2,[X|L3]) :- ap(L1,L2,L3).
ap([],L,L).
```

The atom `qs`$([2,3,1],[2,1,3])$ is a wrong answer. Figure 1 illustrates a *CPP* for $qs([2,3,1],[2,1,3])$. It is a part of a proof tree for $qs([2,3,1],[2,1,3])$ because in a proof tree for $qs([2,3,1],[2,1,3])$ each of node (3) (labelled with $2 =< 3$) and node (5) (labelled with $2 > 1$) has a child labelled with `true`.

```
qs([2,3,1],[2,1,3])       (1)
|--------------------------|
|                          |
|-pt([3,1],2,[1],[3])  (2)  |-qs([3],[3])          (18)
| |--2=<3              (3)  | |-pt([],3,[],[])     (19)
| |--pt([1],2,[1],[])  (4)  | | |--true            (20)
|    |--2>1            (5)  | |-qs([],[])          (21)
|    |--pt([],2,[],[]) (6)  | | |--true            (22)
|       |--true        (7)  | |-qs([],[])          (23)
|-qs([1],[1])          (8)  | | |--true            (24)
  |--pt([],1,[],[])    (9)  | |-ap([3],[],[3])     (25)
  |  |--true           (10) |   |--ap([],[],[])    (26)
  |--qs([],[])         (11) |      |--true         (27)
  |  |--true           (12) |-ap([2,1],[3],[2,1,3])(28)
  |--qs([],[])         (13)   |-ap([1],[3],[1,3])  (29)
  |  |--true           (14)     |--ap([],[3],[3])  (30)
  |--ap([1],[],[1])    (15)        |--true         (31)
     |--ap([],[],[])    (16)
        |--true        (17)
```

**Figure 1: A *CPP* for `qs`$([2,3,1],[2,1,3])$**

It is shown in [10] that a wrong answer $A$ can be diagnosed by searching a *CPP* $T$ for $A$ to find a node $L'$ of $T$ such that $C = RI(T, L')$ is invalid in $I$. Either $C$ is an inconsistent clause instance, or $C = (\neg A' \leftarrow \texttt{true})$ such that $A'$ is a missing answer.

We now consider missing answers. A goal $W'$ is derived from another $W$, denoted $W \Longrightarrow W'$ if $W = L_1, L_2, \cdots, L_m$, and either (i) $L_1$ is positive, there is a clause $A \leftarrow W''$ such that $L_1$ and $A$ unify with a `mgu` $\theta$, and $W' = (W'', L_2, \cdots, L_m)\theta$, or (ii) $L_1$ is negative with $L_1 = \neg A_1$, $P$ finitely fails on $A_1$, and $W' = (L_2, \cdots, L_m)$.

Let $W$ be a goal, and $T$ a tree rooted at $W$.

(1) An SLD tree for $W$ is a tree rooted at $W$ such that $W''$ is a child of $W'$ in $T$ iff $W''$ derived from $W'$.

(2) A partial SLD tree for $W$ is a tree rooted at $W$ such that $W''$ is a child of $W'$ in $T$ only if $W' \Longrightarrow W''$.

(3) A partial SLD tree $T$ for $W$ is complete (*CPS* for short) if $W''$ is a child of $W'$ in $T$ for each $W''$ derived from $W'$ such that is satisfiable in $I$.

The notion of a *CPS* $T$ for $W$ captures the idea that if $P$ is correct *wrt* $I$, then any successful derivation of $W$ corresponds to a path from the root of $T$ to a leaf of $T$ which is $\Box$. It follows that an SLD tree for $W$ is a *CPS* for $W$ . If $A$ is a missing answer, then a *CPS* for $A$ (via a fair computation rule) is a finite tree, and none of its leaves is $\Box$ because any derivation of $A$ terminated with $\Box$ corresponds to a proof tree for $A\theta$ for some $\theta$.

Consider the following buggy program. The intended interpretation for `d`(X,Ys,Zs) is that either $X$ is in list Ys but not in list $Zs$ or $X$ is in list $Zs$ but not in list $Ys$. The intended interpretation for `m`(X,L) is that $X$ is in list $L$.

```
d(X,Ys,Zs) :-  m(X,Ys), \+ m(X,Zs).
d(X,Ys,Zs) :-  m(X,Zs), \+ m(X,Zs). % \+ m(X,Ys)
```

```
m(X,[X|Xs]).
m(X,[Y|Ys]) :- m(X,Ys).
```

The atom $\mathtt{d}(3,[1,2,4],[2,3])$ is a missing answer with this *CPS* .

```
(d(3,[1,2,4],[2,3])) (1)
|
|-(m(3,[1,2,4]),\+ m(3,[2,3])) (2)
| |-(m(3,[2,4]),\+ m(3,[2,3])) (3)
|   |-(m(3,[4]),\+ m(3,[2,3])) (4)
|     |-(m(3,[]),\+ m(3,[2,3]))(5)
|
|--(m(3,[2,3]),\+ m(3,[2,3])) (6)
   |-(m(3,[3]),\+ m(3,[2,3])) (7)
     |-(\+ m(3,[2,3]))        (8)
     |-(m(3,[]),\+ m(3,[2,3]))(9)
```

Let $T$ be a *CPS* . A node $W$ of $T$ is critical if $W$ is satisfiable in $I$ and each child of $W$ is unsatisfiable in $I$. It is shown in [10] that a missing answer $A$ can be diagnosed by searching a *CPS* $T$ for $A$ to find a critical node $W$ of $T$. $W \neq \Box$ since $P$ finitely fails on $A$. Let $L$ be the selected literal $L$ of $W$. Either $L$ is an incompletely covered atom, or $L = \neg A'$ and $A'$ is a wrong answer.

# 3. A DIAGNOSER

This section recalls a declarative diagnoser $\pi$ for normal logic programs that is presented in details in [10]. The top level procedures of $\pi$ are defined.

```
wrong(A,D) :-
    cpp(A,T), !, invalid_impl(T,C),
    (  C = (\+A1:-true) -> missing(A1,D) ; D = C ).

missing(A,D) :-
    cps((A),T), !, critical(T,W), selected(W,L),
    (  L = \+A1 -> wrong(A1,D) ; D = L ).
```

The auxiliary procedure $\mathtt{cpp}(A,T)$ succeeds with $T$ being a *CPP* for $A$. Likewise, $\mathtt{cps}(W,T)$ succeeds with $T$ being a *CPS* for $W$. The procedure $\mathtt{selected}(W,A)$ succeeds with $A$ the selected atom of $W$.The procedure $\mathtt{invalid\_impl}(T,C)$ succeeds with $C$ being an invalid root implication of a node in $T$; and the procedure $\mathtt{critical}(T,W)$ succeeds with $W$ being a critical node of $T$.

To diagnose a wrong answer $A$, $\mathtt{wrong}/2$ first calls $\mathtt{cpp}/2$ to construct a *CPP* $T$ for $A$. Then it calls $\mathtt{invalid\_impl}/2$ to find a node $L'$ of $T$ such that $C = RI(T,L')$ is invalid. If $C$ is an instance of a clause of $P$, $\mathtt{wrong}/2$ returns with $C$ as its output. Otherwise $C = (\neg A' \leftarrow \mathit{true})$ with $A'$ being a missing answer and $\mathtt{wrong}/2$ calls $\mathtt{missing}/2$ to diagnose the missing answer $A'$. To diagnose a missing answer $A$, $\mathtt{missing}/2$ first calls $\mathtt{cps}/2$ to construct a *CPS* $T$ for $\leftarrow A$. Then it calls $\mathtt{critical}/2$ to find a $W$ node of $T$ such that $W$ is a critical node of $T$. Let $L$ be the selected literal of $W$. If $L$ is positive, $\mathtt{missing}/2$ returns with $L$ as its output. Otherwise, $L = \neg A'$ with $A'$ being a wrong answer and $\mathtt{missing}/2$ calls $\mathtt{wrong}/2$ to diagnose the wrong answer $A'$.

# 4. USE OF CORRECTNESS ASSERTIONS

Given a symptom, diagnoser $\pi$ first constructs a tree modelling the execution of the symptom and then searches the tree for the bug that causes the symptom. The quantity of the queries on the oracle is dependent on the size of the tree. Therefore it is desirable to reduce size of the tree before it is searched.

We present three tree transformations that reduces the size of the tree modelling the execution of the symptom before it is searched. Each of these tree transformations makes use of assertions about the correctness of the program and these assertions can be accumulated during the process of debugging.

## 4.1 Correctness Assertions I

At some stage of debugging, the programmer may know that the result of a call of a certain pattern is always correct. Such knowledge is expressed as an assertion $\mathtt{faithful}(p(\vec{t}))$. The meaning of $\mathtt{faithful}(p(\vec{t}))$ is that $p(\vec{t})\theta \in SS(P)$ iff $p(\vec{t})\theta$ is valid in $I$ for any substitution $\theta$. In other words, $P$ gives correct answers when called with an instance of $p(\vec{t})$. This applies naturally to all the built-in-predicates and all the predefined predicates. This also applies to a procedure that has been thoroughly tested, especially when all procedures that might be called during the execution of the procedure have been thoroughly tested. Let $\Gamma_1$ denote the set of the assertions of the form $\mathtt{faithful}(p(\vec{t}))$.

Let $A$ be a wrong answer, $T$ a *CPP* for $A$, and $B$ a node of $T$. If $\mathtt{faithful}(p(\vec{t})) \in SS(\Gamma_1)$ then $B$ is valid in $I$, whence, $T_B$ can be replaced by a single node $B$. This is formalised as a transformation (relation) $\rightarrowtriangle_1$: $T \rightarrowtriangle_1 T_{/B}$ iff $B$ is a non-leaf node in $T$ such that $\mathtt{faithful}(B) \in SS(\Gamma_1)$. Observe that $T \not\rightarrowtriangle_1 T_{/A}$ since $A$ is a wrong answer and $\mathtt{faithful}(A) \notin SS(\Gamma_1)$.

LEMMA 1. *Let $A$ be an atom, and $T$ a CPP for $A$. If $T \rightarrowtriangle_1 T'$ then $T'$ is a CPP for $A$.* $\Box$

We thus can diagnose a wrong answer $A$ by constructing a *CPP* for $A$, repeatedly applying $\rightarrowtriangle_1$ to the *CPP* until it can not be applied further, and then invoke $\mathtt{invalid\_impl}/2$ to search the *CPP* . Each time $\rightarrowtriangle_1$ is applied, it reduces the size of the *CPP* at least by one. For an example, let $\mathtt{faithful}(\mathtt{ap}(L1,L2,L3)) \in \Gamma_1$. Applying $\rightarrowtriangle_1$ repeatedly to *CPP* in figure 1 will remove nodes (16), (17), (26), (27), (29), (30) and (31).

Let $(L,W)$ be a node of a *CPS* $T$ for $G$. For each computed answer $\theta$ for $L$, either $W\theta$ is a node of $T_{L,W}$ or there is an unsatisfiable goal $W'$ such that $(L,W) \Longrightarrow^* W' \Longrightarrow^* W\theta$. If $\mathtt{faithful}(B) \in SS(\Gamma_1)$ then it does not compromise the soundness and completeness of $\pi$ to substitute $t((L,W),Ts)$ for $T_{L,W}$ where $Ts = \{T_{W\theta} \mid W\theta \text{ is a node of } T_{L,W}\}$. This is formalised as a transformation $\rightarrowtriangle_2$ defined

$$T \rightarrowtriangle_2 T[T_{(L,W)}, t((L,W), \{T_{W\theta} \mid W\theta \text{ is a node of } T_{(L,W)}\}]$$

iff $\mathtt{faithful}(B) \in SS(\Gamma_1)$.

LEMMA 2. *Let $G$ be a goal, $T$ a CPS for $G$ , and $T \rightarrowtriangle_2 T'$. If $G_1$ is a critical node of $T'$ then $G_1$ is a critical node of $T$. Furthermore, if $T$ has a critical node then $T'$ has one.* □

We therefore can diagnose a missing answer $A$ by first constructing a *CPS* for $A$, then repeatedly applying transformation $\rightarrowtriangle_2$ to the *CPS* until it cannot be further pruned and finally invoke `critical`/2 to search the *CPS* . For instance, if `faithful`(`m`$(X, L)) \in \Gamma_1$ then repeatedly applying $\rightarrowtriangle_2$ to the *CPS* for `d`$(3, [1, 2, 4], [2, 3])$ in section 2 will result in the following *CPS* . The effect is that nodes (3),(4),(5),(7) and (9) are removed and node (8) becomes a child of node (6).

```
(d(3,[1,2,4],[2,3]))                (1)
|--(m(3,[1,2,4]),\+ m(3,[2,3]))     (2)
|--(m(3,[2,3]),\+ m(3,[2,3]))       (6)
   |--(\+ m(3,[2,3]))               (8)
```

## 4.2 Correctness Assertions II

During the process of debugging, the programmer may also know that a particular procedure when called with inputs of a certain pattern does not cause any symptom. We represent such knowledge by means of an assertion of the form `correct`$(p(\vec{t}))$. The semantics of `correct`$(p(\vec{t}))$ is:

(1) $p(\vec{t})\theta$ is not an incompletely covered atom for any $\theta$, and

(2) if $A \leftarrow W$ is a clause of $P$ with $A$ unifying with $p(\vec{t})$ with `mgu` $\theta$ then $(A \leftarrow W)\theta$ is not an inconsistent clause instance.

(1) is equivalent to that if $p(\vec{t})\theta$ is satisfiable in $I$ and $P$ finitely fails on $p(\vec{t})\theta$ then there is a clause $A \leftarrow W$ of $P$ such that $A$ and $p(\vec{t})\theta$ unify with `mgu` $\eta$ and $W\eta$ is satisfiable in $I$. (2) is equivalent to that if $p(\vec{t})\theta$ is invalid in $I$ and $p(\vec{t})\theta \in SS(P)$ then, for each a clause $A \leftarrow W$ of $P$ such that $A$ and $p(\vec{t})\theta$ unify with `mgu` $\eta$, $W\eta$ is invalid in $I$. (1) and (2) together implies that when called with an instance of $p(\vec{t})$, procedure $p/m$ does not cause any symptom and it only propagates symptoms. We now introduce a transformation $\rightarrowtriangle_3$ on a *CPS* .

Let `children`$(G)$ be the set of children of $G$ and $G^p$ be the parent of $G$. Let $T$ be a *CPS* and $G$ a non-root node of $T$ such that $G = (L, W)$. If `correct`$(L) \in SS(\Gamma_2)$ then it is safe to remove $G$ by linking its children to its parent. This is formalised as follows.

$$T \rightarrowtriangle_3 T[T_{G^p}, t(G^p, \texttt{children}(G^p) \setminus G \cup \texttt{children}(G))]$$

iff $G = (L, W)$ and `correct`$(L) \in SS(\Gamma_2)$. The transformation $\rightarrowtriangle_3$ removes a node whose selected literal satisfies a correctness assertion in $\Gamma_2$ and promotes its children.

LEMMA 3. *Let $T$ a CPS for $G$ , and $T \rightarrowtriangle_3 T'$. If $G_1$ is a critical node of $T'$ then $G_1$ is a critical node of $T$. Moreover, if $T$ has a critical node then $T'$ has one.* □

By lemma 3, given a missing answer $A$ and a *CPS* for $A$, we can diagnose by repeatedly applying transformation $\rightarrowtriangle_3$ to the *CPS* and then invoking `critical`/2 to search the resulting tree. For instance, if `correct`$(\texttt{m}(X, L)) \in \Gamma_2$ then repeatedly applying $\rightarrowtriangle_3$ to the *CPS* in the previous section will result in the following *CPS* . The effect is that nodes (2) and (6) are removed and node (8) becomes a child of node (1).

```
(d(3,[1,2,4],[2,3]))                (1)
|--(\+ m(3,[2,3]))                  (8)
```

## 4.3 An Improved Diagnoser

We now present an improved declarative diagnoser $\pi'$ for normal logic programs. It applies $\rightarrowtriangle_1$ to reduce the size of a *CPP* or $\rightarrowtriangle_2$ and $\rightarrowtriangle_3$ to reduce the size of a *CPS* before it is searched. `transform_1`$(T_1, T_2)$ is true iff $T_1 \rightarrowtriangle_1^* T_2$ and $T_2 \rightarrowtriangle_1 T_2$ where $\rightarrowtriangle_1^*$ is the reflexive and transitive closure of $\rightarrowtriangle_1$, `transform_2`$(T_1, T_2)$ is true iff $T_1 \rightarrowtriangle_2^* T_2$ and $T_2 \rightarrowtriangle_2 T_2$, and `transform_3`$(T_1, T_2)$ is true iff $T_1 \rightarrowtriangle_3^* T_2$ and $T_2 \rightarrowtriangle_3 T_2$. $\pi'$ consists of all the predicates in $\Gamma_1$ and $\Gamma_2$, predicates `transform_1`/2, `transform_2`/2 and `transform_3`/2, all the predicates of $\pi$ except *wrong*/2 and `missing`/2 which are redefined as follows.

```
wrong(A,D) :-
    cpp(A,T), transform_1(T,T1) !, invalid_impl(T1,C),
    ( C = (\+A1:-true) -> missing(A1,D) ; D = C ).

missing(A,D) :-
    cps((A),T), transform_2(T,T1), transform_3(T1,T2),
    !, critical(T2,G), selected(G,L),
    ( L = \+A1 -> wrong(A1,D) ; D = L ).
```

The improved diagnoser has been implemented in SWI Prolog. The following is a session with the diagnoser.

```
debug % load buggyqs.pl
debug % faithful
   procedure or goal? pt/4.
debug % faithful
   procedure or goal? ap/3.

debug % exec qs([8,3,8,4,6,2,7],[2,3,4,6,7,8,8]).
  Is qs([8,3,8,4,6,2,7], [2,3,4,6,7,8,8]) valid? y
  qs([8,3,8,4,6,2,7], [2,3,4,6,7,8,8])) is missing.
debug % diagnose
  qs([8,3,8,4,6,2,7], [2,3,4,6,7,8,8]) is missing.
  Is qs([3,8,4,6,2,7], [3,2,8,4,6,7]) valid? n
  qs([3,8,4,6,2,7], [3,2,8,4,6,7]) is wrong.
  Is qs([2], [2]) valid? y
  Is qs([8,4,6,7], [8,4,6,7]) valid? n
  Is qs([4,6,7], [4,6,7]) valid? y
  Is qs([], []) valid? y

  This is an inconsistent clause.
    qs([8, 4, 6, 7], [8, 4, 6, 7]) :-
      pt([4, 6, 7], 8, [4, 6, 7], []),
      qs([4, 6, 7], [4, 6, 7]),  qs([], []),
      ap([8, 4, 6, 7], [], [8, 4, 6, 7]).
  It is an instance of
    qs([X|L], L0) :-
      pt(L, X, L1, L2),
      qs(L1, L3),  qs(L2, L4),  ap([X|L3], L4, L0).
  with
      L  = [4,6,7]  L1 = [4,6,7]  L2 = []  X = 8
      L3 = [4,6,7]  L4 = []       L0 = [8,4,6,7]
  discard/modify/none?
```

In the session, the programmer first loads the buggy quicksort program from the file named "buggyqs.pl". Then he informs the diagnoser that calls to procedures $pt/4$ and $ap/2$ do not result in any bug symptom. After that, he tests the program with the call $qs([8,3,8,4,6,2,7],[2,3,4,6,7,8,8])$ which is a missing answer and then instructs the diagnoser to find the bug. The diagnoser queries the programmer with five questions before it reports an inconsistent clause. No questions about procedures $pt/4$ and $ap/2$ are asked.

The session uses the top-down zooming strategy to search both *CPP* and *CPS* trees. Diagnoser supports various other search strategies.

# 5. CONCLUSION

We have presented a declarative diagnoser that models the execution of a symptom as a tree and searches the tree for the bug that causes the symptom. We have presented three tree transformations that reduce the size of the tree and maintain the soundness and the completeness of the diagnoser. These three tree transformations use two kinds of correctness assertion.

The diagnoser presented in [17, 18] stores the oracle's answer to each query as an assertion about the intended interpretation of the program. For instance, if an atom $A$ is judged by the oracle to be valid in the intended interpretation then assertion `fact`$(A, valid)$ is stored. An assertion stored by the diagnoser is simply a unit clause of procedure `fact`/2. When a query is to be made later, the diagnoser first tries to use the stored assertions to answer the query. Only when the query cannot be answered this way will the query be imposed on the oracle.

The diagnosers presented in [4], [1] and [7] use a full specification for the program therefore the oracle can be fully automated. However, a full infallible and executable specification is a strong requirement.

The diagnoser presented in [2, 3] allows the programmer to partially specify the intended interpretation of the program. An assertion is a clause of one of the four procedures that are used to specify what is known valid, what is known false, what is known satisfiable and what is known unsatisfiable in the intended interpretation of the program. When a query is to be made, the diagnoser first tries to answer the query using assertions and imposed the query on the oracle only if the query cannot be answered this way.

There are two main differences between the use of assertions in the above diagnosers and that in our diagnoser. All the above diagnosers use assertions to answer queries in an automated or partially automated manner. Our diagnoser uses assertions to reduce the tree modelling the execution of the bug symptom instead of to answer queries. These diagnosers use assertions about the intended interpretation of the program while our diagnoser uses correctness assertions. The correctness assertion are much more abstract than assertions about the intended interpretation. They are also easier for the programmer to specify. The assertions about intended interpretation can be easily incorporated into our declarative diagnoser. They can be used to answer queries as in other diagnosers or they can be used during the construction of *CPP* and *CPS* trees.


## *Acknowledgements*

This work was supported, in part, by NSF grants CCR-0131862 and INT-0327760.


# 6. REFERENCES


[1] N. Dershowitz and Y.-J. Lee. Deductive debugging. In *Proceedings of 1987 Symposium of Logic Programming*, pages 298–306. The IEEE Computer Society Press, 1987.

[2] W. Drabent, S. Nadjm-Tehrani, and J. Małuszynski. The use of assertions in algorithmic debugging. In *The Proceedings of the International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

[3] W. Drabent, S. Nadjm-Tehrani, and J. Małuszynski. Algorithmic debugging with assertions. In Harvey Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 502–521. The MIT Press, 1989.

[4] A. Edman and S.-Å. Tärnlund. Mechanization of an oracle in a debugging system. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, volume 2, pages 553–555, Karlsruhe, West Germany, August 1983.

[5] G. Ferrand. Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. *The Journal of Logic Programming*, 4(3):177–198, 1987.

[6] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri. Generalized algorithmic debugging and testing. *SIGPLAN Notices*, 26(6):317–326, 1991.

[7] T. Kanamori, T. Kawamura, M. Maeji, and K.Horiuchi. Logical program diagnosis from specifications. ICOT Technical Report TR-447, March 1989.

[8] J.W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[9] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[10] L. Lu. A generic declarative diagnoser for normal logic programs. In *Logic Programming and Automated Reasoning, 5th International Conference*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 290–304. Springer, 1994.

[11] L. Naish. Declarative debugging of lazy functional programs. *Australian Computer Science Communications*, 15(1):287–294, 1993.

[12] L. Naish and T. Barbour. A declarative debugger for a logical-functional language. In G. Forsyth and M. Ali, editors, *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert*, volume 2, pages 91–99, Melbourne, June 1995.

[13] L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.

[14] L.M. Pereira. Rational debugging in logic programming. In E. Shapiro, editor, *Proceedings of the 3rd International Logic Programming Conference*, pages 203–210. Springer Verlag, 1986. Lecture Notes in Computer Science no. 225.

[15] L.M. Pereira and M. Calejo. A framework for Prolog debugging. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 481–495. The MIT Press, 1988.

[16] G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in CLP. In Michael J. Maher and Jean-Francois Puget, editors, *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, page 472. Springer, 1998.

[17] E. Shapiro. Algorithmic program diagnosis. In *ACM Conference Record of the ninth annual ACM Symposium on Principles of Programming Languages*, pages 299–308. The ACM Press, 1982.

[18] E. Shapiro. *Algorithmic Debugging*. The MIT Press, 1983.

[19] S.Y. Yan. Foundations of declarative debugging in arbitrary logic programming. *International Journal of Man Machine Studies*, 32:215–232, 1990.