A Generic Declarative Diagnoser for Normal Logic Programs

Lunjin Lu

The University of Birmingham, Birmingham B15 2TT, U.K.

Abstract. In this paper we develop a generic declarative diagnoser for normal logic programs that is based on tree search. The soundness and the completeness of the diagnoser are proved. The diagnoser is generic in that it can be used with different search strategies such as the bottom-up, top-down, top-down zooming and divide-and-query strategies in the literature. The user can specialise the diagnoser by choosing their own search strategy. The diagnoser also has a smaller search space than diagnosers reported in the literature. This is achieved by using the acquired part of the intended interpretation of the program to prune the search space before it is searched.

1 Introduction

An error that makes a program exhibit an unexpected behaviour is called a bug while the unexpected behaviour that a bug causes is called a bug symptom. After a bug symptom has been found, the programmer has to locate and identify the bug that causes the bug symptom and correct the bug in order to obtain the expected program behaviour. The process of locating and identifying the bug that causes a bug symptom is called program diagnosis. A software tool that supports such a process is called a diagnoser. Declarative program diagnosis is an interactive process whereby 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. There has been much research into declarative logic program diagnosis [4, 8, 11, 13, 18, 19, 22].

A buggy logic program may exhibit many kinds of bug symptom. It may produce a wrong answer, fail to produce a correct answer, fall into looping, call procedures with wrong types of arguments, or violate the safe rule of negation as failure, etc. This paper is concerned with the first two kinds of bug symptom.

We assume that readers are familiar with the terminology of logic programming [13]. Let ϵ be the identity substitution and P a logic program. We define the success set of P as

$$SS(P) = \{A \mid A \textit{ is an atom and } \epsilon \textit{ is a computed answer for } P \cup \{\leftarrow A\}\}$$

$SS(P)$ possibly contains non-ground atoms and describes the operational behaviour of logic programs more precisely than the success set defined in terms of ground atoms [21]. A model-theoretic counterpart of $SS(P)$ for definite programs is given in [7].

Definition 1 *Let P be a logic program and I an interpretation.*

(1) An atom A is an inconsistency symptom of P w.r.t. I if A is invalid in I and $A \in SS(P)$.
(2) An atom A is an insufficiency symptom of P w.r.t. I if A is satisfiable in I and P finitely fails on A.

Definition 2 *Let P be a logic program and I an interpretation.*

(1) A clause instance $A \leftarrow W$ is an inconsistent clause instance of P w.r.t. I if $A \leftarrow W$ is an instance of a clause of P and $A \leftarrow W$ is invalid in I.
(2) An atom A is an uncovered atom of P w.r.t. I if A is valid in I and, for every clause $A' \leftarrow W$ of P s.t. A and A' unify with m.g.u. θ, $W\theta$ is unsatisfiable in I. An atom A is an incompletely covered atom of P w.r.t. I if an instance of A is an uncovered atom of P w.r.t. I.

If there is an inconsistency or insufficiency symptom of P w.r.t. I then there is an inconsistent clause instance of P w.r.t. I or an incompletely covered atom of P w.r.t. I [8, 12, 13, 22]. Therefore, given an inconsistency or insufficiency symptom of P w.r.t. I, a declarative diagnoser for logic programs searches for an inconsistent clause instance of P w.r.t. I or an incompletely covered atom of P w.r.t. I. Many declarative diagnosers for logic programs have been developed. Shapiro [18, 19] developed the algorithmic debugging method[1] and exemplified the method through pure Prolog. Ferrand [8] adapted the algorithmic debugging method for definite logic programs. Lloyd [12, 13] 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 [12, 13] obtained a top-down diagnoser by adding control information. Yan [22] improved the top-down diagnoser by reorganising its control information.

Each of these declarative diagnosers for logic programs has an inconsistency diagnosis procedure and an insufficiency diagnosis procedure. The inconsistency diagnosis procedure is called with an inconsistency symptom of P w.r.t. I as its input and the insufficiency diagnosis procedure is called with an insufficiency symptom of P w.r.t. I as its input.

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 about is the intended interpretation of the program. The quantity of queries may be large and reducing the quantity of queries has been the main objective of much research into declarative diagnosis [3, 4, 6, 16, 17]. The quantity of queries is dependent on the size of the search space and the search strategy [13, 19].

In this paper we present a declarative diagnoser for normal programs. A normal program consists of a set of normal clauses of the form $A \leftarrow L_1, \cdots, L_m$

[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.

where A is an atom and each literal L_i is either an atom or the negation of an atom. The SLDNF resolution procedure is used to implement normal programs. To ensure the soundness of the SLDNF resolution procedure, a safe computation rule or a weak safe computation rule must be used. A safe computation rule always selects a positive literal or a ground negative literal. A weak safe computation rule always selects a positive literal or a negative literal $\neg A_i$ s.t. A_i will not be instantiated if P succeeds on A_i. The diagnosis of unsafe uses of negation as failure is beyond of the scope of this paper. We assume that no unsafe use of negation as failure arises during the execution of an inconsistency or insufficiency symptom of P w.r.t. I.

Our diagnoser has a smaller search space than the diagnosers reported in the literature. The intended interpretation I of a logic program P consists of the set of atomic formulae that should be proved by P and the set of the atomic formulae that should be disproved by P. Whilst program P is being debugged, the oracle incrementally provides the debugging system with the intended interpretation I of P. Therefore, at some stage of debugging, the debugging system has already acquired part I' of I. Our declarative diagnoser uses I' to reduce the size of the search space.

Given an inconsistency symptom A of P w.r.t. I, our diagnoser first constructs a tree called an I-congruent partial proof tree (cpp) for P and A and then diagnoses P by searching this tree. An I-*cpp* for P and A is similar to a proof tree for P and A [1] except that a leaf of an I-*cpp* for P and A is an atom that is valid in I while a leaf of a proof tree for P and A is the atom *true*. Because I' is a part of I, an I'-*cpp* for P and A is also an I-*cpp* for P and A. Therefore, an I'-*cpp* for P and A can be used where an I-*cpp* for P and A is required. Whilst an I'-*cpp* tree for P and A is being constructed it is not necessary to query the oracle because I' has already been known to the debugging system. An I'-*cpp* for P and A is in size smaller than or equal to a proof tree for P and A because an I'-*cpp* for P and A may contain only one node for an atom that is valid in I' while a proof tree for P and A has a subtree for the same atom. When I' is empty, an I'-*cpp* for P and A is a proof tree for P and A. As I' increases during debugging, our diagnoser is able to construct a smaller and smaller I'-*cpp* for P and A.

Given an insufficiency symptom A of P w.r.t. I, our diagnoser first constructs a tree called an I-complete partial SLDNF tree (cps) for $P \cup \{\leftarrow A\}$ and then diagnoses by searching this tree. An I-*cps* for $P \cup \{\leftarrow A\}$ is similar to an SLDNF tree for $P \cup \{\leftarrow A\}$ [13] except that a node $\leftarrow W$ in an SLDNF tree for $P \cup \{\leftarrow A\}$ has a child node for each goal $\leftarrow W'$ that is derived from $\leftarrow W$ and P, while a node $\leftarrow W$ in an I-*cps* for $P \cup \{\leftarrow A\}$ only needs to have a child node for a goal $\leftarrow W'$ that is derived from $\leftarrow W$ and P s.t. W' is satisfiable in I. In other words, if W' is unsatisfiable in I then node $\leftarrow W'$ and the subtree rooted at $\leftarrow W'$ can be removed from an I-*cps* for $P \cup \{\leftarrow A\}$. Because I' is a part of I, an I'-*cps* for $P \cup \{\leftarrow A\}$ is also an I-*cps* for $P \cup \{\leftarrow A\}$. Hence, an I'-*cps* for $P \cup \{\leftarrow A\}$ can be used where an I-*cps* for $P \cup \{\leftarrow A\}$ is needed. The construction of an I'-*cps* for $P \cup \{\leftarrow A\}$ does not need to query the oracle because I' has already been known to the debugging system. An I'-*cps* for $P \cup \{\leftarrow A\}$ is smaller in size than an SLDNF tree for $P \cup \{\leftarrow A\}$ because a node $\leftarrow W$ in an SLDNF tree for $P \cup \{\leftarrow A\}$ usually has more child nodes than in an I'-*cps* for $P \cup \{\leftarrow A\}$. When I' is empty, an I'-*cps* for $P \cup \{\leftarrow A\}$ is an SLDNF tree for $P \cup \{\leftarrow A\}$. As I' increases during debugging, our diagnoser is able to construct a smaller and smaller I'-*cps* for $P \cup \{\leftarrow A\}$.

Our diagnoser is generic in that different tree search strategies can be used with the diagnoser and the user can specialise the diagnoser by specifying the search strategy to be used.

Section 2 formally introduces the concepts of I-*cpp* and I-*cps* and establishes that they are sufficient for the purpose of diagnosis, that is, an inconsistency symptom A of P w.r.t. I can be diagnosed by searching an I-*cpp* for P and A, and an insufficiency symptom A of P w.r.t. I can be diagnosed by searching an I-*cps* for $P \cup \{\leftarrow A\}$. Section 3 presents the diagnoser and proves its soundness and completeness. In section 4, we show the generality of our declarative diagnoser and compares the diagnoser with the declarative diagnosers in the literature with respect to the size of the search space. Section 5 concludes the paper and points to some further work on our declarative diagnoser.

2 Search Space

This section formally introduces the notions of I-*cpp* and I-*cps* and shows that an inconsistency symptom A of P w.r.t. I can be diagnosed by searching an I-*cpp* for P and A and an insufficiency symptom A of P w.r.t. I can be diagnosed by searching an I-*cps* for $P \cup \{\leftarrow A\}$.

Let T be a tree, r the root of T, and v a node of T. v is a branch node if v is neither the root of T nor a leaf of T. The height of T, written as $h(T)$, is the length of the longest path of T. $n(T)$ denotes the number of nodes of T. The level of v in T, denoted by $l(v,T)$, is the length of the path from r to v. T_v denotes the sub-tree of T that is rooted at v. Notice that $T = T_r$.

Definition 3 An ordered tree T is a literal-labelled tree if each node of T is a literal. Let L be a node of 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 $RI(T,L)$.

Definition 4 Let P be a normal program, I an interpretation and A an atom.

(1) A partial proof tree T for P and A is a literal-labelled tree satisfying the following two conditions. (i) The root of T is A. (ii) For each non-leaf node L' of T, either $RI(T,L')$ is an instance of a clause of P or $RI(T,L') = (\neg A' \leftarrow true)$ where A' is an atom on which P finitely fails.
(2) T is a proof tree for P and A if T is a partial proof tree for P and A, and every leaf node of T is *true*.
(3) A partial proof tree T for P and A is an I-*cpp* for P and A if every leaf node of T is a literal that is valid in I.

A proof tree for P and A is an I-*cpp* for P and A because *true* is valid in I. This definition of a proof tree is similar to that of [1]. The leaves of a proof tree defined in [1] are instances of unit clauses of P while they are *true* according to the above definition that treats a unit clause as having body *true*.

Example 1 Let P be the following buggy quick sort program. P has a bug that is 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).
```

Let the intended interpretation I be as usual. $qs([2,3,1],[2,1,3])$ is an inconsistency symptom of P w.r.t. I. Suppose that the acquired part I' of I consists of the knowledge of built-in predicates. Then the following is an I'-*cpp* for P and $qs([2,3,1],[2,1,3])$ and hence an I-*cpp* for P and $qs([2,3,1],[2,1,3])$.

```
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)
```

The above tree is smaller than a proof tree for P and $qs([2,3,1],[2,1,3])$ because in a proof tree for P and $qs([2,3,1],[2,1,3])$ each of node (3) (labelled with $2 =< 3$) and node (5) (labelled with $2 > 1$) has a child labelled with *true*.

Example 2 Let P and I be the same as in example 1. Suppose that at some stage during debugging, the acquired part I' of I consists of the knowledge of built-in predicates and the knowledge that $qs([X],[X])$ is valid in I for all possible X. Then removing nodes 9-17 and 19-27 from the tree in example 1 will result in an I'-*cpp* for P and $qs([2,3,1],[2,1,3])$ and hence an I-*cpp* for P and $qs([2,3,1],[2,1,3])$. This tree and the tree in example 1 are both I-*cpp* for P and $qs([2,3,1],[2,1,3])$. But the former is much smaller than the latter. This example shows that, given the same inconsistency symptom A of P w.r.t. I, our diagnoser is able to construct a smaller and smaller I-*cpp* for P and A as I' increases during debugging.

Lemma 1 *Let P be a normal program, I an interpretation, A an atom that is invalid in I, and T an I-cpp for P and A. Then there is a node L' of T s.t. $RI(T,L')$ is invalid I. Furthermore, either $RI(T,L')$ is an inconsistent clause instance of P w.r.t. I, or $RI(T,L') = (\neg A' \leftarrow \mathrm{true})$ and A' is an insufficiency symptom of P w.r.t. I.* ∎

Lemma 1 states that an inconsistency symptom A of P w.r.t. I can be diagnosed by searching an I-*cpp* T for P and A to find a node L' of T s.t. $C = RI(T,L')$ is invalid in I. Either C is an inconsistent clause instance of P w.r.t. I, or $C = (\neg A' \leftarrow true)$ s.t. A' is an insufficiency symptom of P w.r.t. I.

Definition 5 An ordered tree T is a goal-labelled tree if each node of T is a goal.

Definition 6 Let P be a normal program, R a computation rule, G and G' normal goals, and I an interpretation.

(1) We say that G' is derived from G and P via R and write $G \xrightarrow{P,R} G'$ if $G = \leftarrow L_1, \cdots, L_i, \cdots, L_m$, L_i is the selected literal of G by R, and either (i) L_i is positive, there is a clause $A \leftarrow W$ of P s.t. L_i and A unify with a m.g.u. θ, and $G' = \leftarrow (L_1, \cdots, L_{i-1}, W, L_{i+1}, \cdots, L_m)\theta$, or (ii) L_i is negative with $L_i = \neg A_i$, P finitely fails on A_i, and $G' = \leftarrow (L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_m)$.
(2) $\mathcal{C}(G,P,R)$ is the set of all the goals that are derived from G and P via R, that is, $\mathcal{C}(G,P,R) = \{G' | G \xrightarrow{P,R} G'\}$.
(3) $\mathcal{E}(G,P,R,I)$ is the subset of $\mathcal{C}(G,P,R)$ s.t. for each goal $G' = \leftarrow W'$ in $\mathcal{E}(G,P,R,I)$, W' is satisfiable in I.

$$\mathcal{E}(G,P,R,I) = \{\leftarrow W' \mid (\leftarrow W' \in \mathcal{C}(G,P,R)) \textit{ and } W' \textit{ is satisfiable in } I\}$$

Definition 7 Let P be a normal program, R a computation rule, G a normal goal, T a goal-labelled tree s.t. the root of T is G, and I an interpretation.

(1) T is a partial SLDNF tree for $P \cup \{G\}$ via R if any two nodes G^1 and G^2 of T s.t. G^2 is a child of G^1 satisfy $G^2 \in \mathcal{C}(G^1,P,R)$.
(2) T is an SLDNF tree for $P \cup \{G\}$ via R if T is a partial SLDNF tree for $P \cup \{G\}$ via R s.t. if G^1 is a node of T, then each G^2 in $\mathcal{C}(G^1,P,R)$ is also a node of T and G^2 is a child of G^1.

(3) T is an *I-cps* for $P \cup \{G\}$ via R if T is a partial SLDNF tree for $P \cup \{G\}$ via R s.t. if G^1 is a node of T, then each G^2 in $\mathcal{E}(G^1, P, R, I)$ is also a node of T and G^2 is a child of G^1.

The definition of an SLDNF tree is equivalent to that given in the literature such as [13]. The notion of an *I-cps* T for $P \cup \{G\}$ via R captures the idea that if P is correct w.r.t. I, then any successful derivation of $P \cup \{G\}$ via R corresponds to a path from the root of T to a leaf of T which is $\Box$. It follows from definition 7 that an SLDNF tree for $P \cup \{G\}$ via R is an *I-cps* for $P \cup \{G\}$ via R for any I. If A is an insufficiency symptom of P w.r.t. I, then an I-cps for $P \cup \{\leftarrow A\}$ via a fair computation rule R is a finite tree, and none of its leaves is $\Box$ because any derivation of $P \cup \{\leftarrow A\}$ terminated with $\Box$ corresponds to a proof tree for P and $A\theta$ for some θ.

Example 3 Let P be 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).          m(X,[X|Xs]).
d(X,Ys,Zs):- m(X,Zs),\+ m(X,Zs).          m(X,[Y|Ys]):- m(X,Ys).
                           % \+ m(X,Ys)
```

$d(3, [1, 2, 4], [2, 3])$ is an insufficiency symptom of P w.r.t. I. Suppose that the acquired part I' of I is empty. Then the following is an *I-cps* for $P \cup \{\leftarrow d(3, [1, 2, 4], [2, 3])\}$ via the left-to-right computation rule. $\leftarrow$ is written as '?'. This tree is also an SLDNF tree for $P \cup \{\leftarrow d(3, [1, 2, 4], [2, 3])\}$ via the left-to-right computation rule.

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

Example 4 Let P and I be the same as in example 3. $Q \wedge \neg Q$ is unsatisfiable in I' for any Q and any consistent I'. So, removing nodes 6-9 from the tree in example 3 will result in an I'-*cps* for $P \cup \{\leftarrow d(3, [1, 2, 4], [2, 3])\}$ via the left-to-right computation rule. This tree and the tree in example 3 are both *I-cps* for $P \cup \{\leftarrow d(3, [1, 2, 4], [2, 3])\}$ via the left-to-right computation rule. But the former is much smaller than the latter. This example shows that, given the same insufficiency symptom A of P w.r.t. I, our diagnoser is able to constructs a smaller and smaller *I-cps* for $P \cup \{\leftarrow A\}$ via a fixed computation rule as I' increases during debugging.

Definition 8 Let I be an interpretation, T a goal-labelled tree and $\leftarrow W$ a node of T. We say $\leftarrow W$ is a critical node of T w.r.t. I if either W is satisfiable in I and $\leftarrow W$ is a leaf of T, or W is satisfiable in I and for each child $\leftarrow W'$ of $\leftarrow W$, W' is unsatisfiable in I.

Lemma 2 *Let P be a normal program, G^1 a normal goal, R a computation rule, I an interpretation, T an I-cps for $P \cup \{G^1\}$ via R, then there is a critical node G of T w.r.t. I. Furthermore, if $G = \leftarrow W = \leftarrow L_1, \cdots, L_{i-1}, L_i, L_{i+1}, \cdots, L_m$ with L_i being the selected literal of G by R then either (1) L_i is a positive literal and L_i is an incompletely covered atom of P w.r.t. I or (2) $L_i = \neg A_i$ where A_i is an atom and A_i is an inconsistency symptom of P w.r.t. I.* ∎

Lemma 2 states that an insufficiency symptom A of P w.r.t. I can be diagnosed by searching an *I-cps* T for $P \cup \{\leftarrow A\}$ via a fair computation rule R to find a critical node G of T w.r.t. I. $G \neq \Box$ since P finitely fails on A. Let L be the selected literal L of G by R. Either L is an incompletely covered atom of P w.r.t. I, or $L = \neg A'$ and A' is an inconsistency symptom of P w.r.t. I.

3 A generic diagnoser

This section presents our declarative diagnoser for normal programs and proves its soundness and completeness. This declarative diagnoser will be referred to as π and in presented in Edinburgh Prolog [9].

Let T be a tree and v a node in T. We use $T - T_v$ to denote the tree resulting from deleting T_v, the sub-tree rooted at v, from T and $T \backslash T_v$ to denote the tree resulting from replacing T_v of T with a node v.

The inconsistency diagnosis procedure of π is $inconsistency(+A, -D)$. When called with an inconsistency symptom A of P w.r.t. I, $inconsistency/2$ succeeds with D being either an inconsistent clause instance of P w.r.t. I or an incompletely covered atom of P w.r.t. I. $inconsistency/2$ first calls $cpp/2$ to construct an *I-cpp* T for P and A. Then $inconsistency/2$ calls $invalid\_impl/2$ to find a node L' of T s.t. $C = RI(T, L')$ is invalid. If C is an instance of a clause of P, $inconsistency/2$ returns with C as its output. Otherwise $C = (\neg A' \leftarrow true)$ with A' being an insufficiency symptom of P w.r.t. I. In this case, $inconsistency/2$ calls $insufficiency/2$ to diagnose the insufficiency symptom A' of P w.r.t. I. The specification for $cpp(+A, -T)$ is that T is an *I-cpp* for P and A.

```
inconsistency(A,D) :-          |     invalid_impl(T,C) :-
  cpp(A,T),                    |       height(T,1),
  !,                           |       root_impl(T,C).
  invalid_impl(T,C),           |     invalid_impl(T,C) :-
  (                            |       branch_node(T,L),
     C=(\+A1:-true)            |       !,
  -> insufficiency(A1,D)       |       (
  ;  D = C                     |          valid(L)
  ).                           |       -> '\'(T,L,T1)
                               |       ; '%'(T,L,T1)
                               |       ),
                               |       invalid_impl(T1,C).
```

invalid_impl$(+T, -C)$ succeeds with $C = RI(T, L')$ being invalid in I for some node L' of T if T is a literal-labelled tree whose root is invalid in I and each of whose leaves is valid in I. height$(+T, -H)$ succeeds with $h(T) = H$. branch_node$(+T, -L)$ succeeds with L being a branch node of T. *root_impl*$(+T, -C)$ succeeds with C being the root implication of T. valid$(+L)$ succeeds iff L is valid in I. $'\%'(+T, +L, -T1)$ succeeds with $T1 = T_L$ if L is a node of T. $'\backslash'(+T, +L, -T1)$ succeeds with $T1 = T\backslash T_L$ if L is a node of T.

The insufficiency diagnosis procedure of π is $insufficiency(+A, -D)$. When called with an insufficiency symptom A P w.r.t. I, $insufficiency/2$ succeeds with D being either an inconsistent clause instance of P w.r.t. I or an incompletely covered atom of P w.r.t. I. $insufficiency/2$ first calls $cps/2$ to construct an *I-cps* T for $P \cup \{\leftarrow A\}$. Then $insufficiency/2$ calls $critical/2$ to find a G node of T s.t. G is a critical node of T w.r.t. I. Let L be the selected literal of G by R. If L is positive, $insufficiency/2$ returns with L as its output. Otherwise, $L = \neg A'$ with A' being an inconsistency symptom of P w.r.t. I. In this case $insufficiency/2$ calls $inconsistency/2$ to diagnose inconsistency symptom A' of P w.r.t. I. $cps(+G, -T)$ succeeds with T being an *I-cps* for $P \cup \{G\}$ via a fixed fair computation rule R. $selected(+G, -L)$ succeeds with L is the selected literal of G by R.

```
insufficiency(A,D) :-
  cps('?'(A),T),
  !,
  critical(T,G),
  selected(G,L),
  (
     L=\+A1
  -> inconsistency(A1,D)
  ;  D = L
  ).
```

```
critical(T,G) :-
  height(T,0),
  root(T,G).
critical(T,G) :-
  \+ height(T,0),
  non_root(T,'?'(W)),
  !,
  (
     satisfiable(W)
  -> '%'(T,'?'(W),T1)
  ;  '-'(T,'?'(W),T1)
  ),
  critical(T1,G).
```

$critical(+T, -G)$ succeeds with G being a critical node of T w.r.t. I if T is a goal-labelled tree s.t. W is satisfiable in I where $\leftarrow W$ is the root of T. $root(+T, -G)$ succeeds with G being the root of T. $non\_root(+T, -G)$ succeeds with G being a node of T other than the root of T. '-'$(+T, +G, -T1)$ succeeds with $T1 = T - T_G$ if G is a node T. $satisfiable(+W)$ succeeds iff W is satisfiable in I.

At a certain stage of diagnosis, I' embodies the knowledge about the intended interpretation I that have been acquired by the diagnoser. When the diagnoser constructs an *I-cpp* (or an *I-cps*), it uses I' to decide if a literal L is *known* to be valid in I (or if a conjunction W of literals is *known* to be unsatisfiable in I). These judgements are made by a set of rules based on I' and other knowledge that the diagnoser have acquired. One example rule is that A is valid in I if A is an instance of another atom that is valid in I.

It is possible that the validity of a literal L in I or the unsatisfiability of a conjunction W of literals in I may not be detected by the set of rules at a certain stage either becuase I' does not have enough information or because the set of rules are not complete. This does not affect the soundness and the completeness of π that are given later.

Example 5 We now show a session of using diagnoser π. The program is a buggy quick sort program. When a query is imposed on the oracle by *satisfiable*/1 or *valid*/1, variables in the atom concerned are replaced with generated mnemonic names that should be understood to be local to the query. The top-down zooming strategy [14] is used to search both *I-cpp* and *I-cps* . Before the diagnosis session begins, the acquired part I' of the intended interpretation I contains the following.

- calls to $ap/3$ or built-in procedures do not result in any bug symptom.
- $qs([X],[X])$ is valid in I.

The definitions for $qs/2$ and $ap/3$ are the same as those in example 1 and $pt/4$ is defined as follows.

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

```
| ?- qs([2,1],L).
L = [2,1]
yes
| ?- inconsistency(qs([2,1],[2,1]),D).
Is qs([],[]) valid? y.       Is pt([1],2,[],[1]) valid? n.
Is pt([],2,[],[]) valid? y.
D = (pt([1],2,[],[1]) :- pt([],2,[],[]))
yes
| ?- % the user corrects the bug in the 1st clause for pt/4
| ?- qs([2,1],L).
no
| ?- insufficiency(qs([2,1],L),D).
Is pt([1],2,A,B) satisfiable? y.       A = ? [1].       B = ? [].
D = pt([1],2,_12938,_12939)
yes
| ?- % the user corrects the bug in the 2nd clause for pt/4
| ?- qs([2,1],L).
L = [2,1]
yes
| ?- inconsistency(qs([2,1],[2,1]),D).
D = (qs([2,1],[2,1]) :-
     pt([1],2,[1],[]),qs([1],[1]),qs([],[]),ap([2,1],[],[2,1]))
yes
```

```
| ?- % the user corrects the bug in the 1st clause for qs/2.
| ?- qs([2,1],L).
L = [1,2]
yes
| ?-
```

The following theorems establish the soundness and the completeness of π.

Theorem 1 (Soundness of π) *Let P be a normal program, A an atom and I an interpretation.*

(1) If A is an inconsistency symptom of P w.r.t. I and $inconsistency(A, D) \in SS(\pi)$, then D is either an inconsistent clause instance of P w.r.t. I, or an incompletely covered atom of P w.r.t. I.
(2) If A is an insufficiency symptom of P w.r.t. I and $insufficiency(A, D) \in SS(\pi)$, then D is either an inconsistent clause instance of P w.r.t. I, or an incompletely covered atom of P w.r.t. I. ∎

Theorem 2 (Completeness of π) *Let P be a normal program, A an atom and I an interpretation.*

(1) If A is an inconsistency symptom of P w.r.t. I, then there is some D s.t. D is either an inconsistent clause instance of P w.r.t. I, or an incompletely covered atom D of P w.r.t. I and $inconsistency(A, D) \in SS(\pi)$.
(2) If A is an insufficiency symptom of P w.r.t. I, then there is some D s.t. D is either an inconsistent clause instance of P w.r.t. I, or an incompletely covered atom D of P w.r.t. I and $insufficiency(A, D) \in SS(\pi)$. ∎

4 Related work

This section compares our diagnoser with the diagnosers reported in the literature w.r.t. the search strategy and the search space, two major factors that affect the quantity of queries.

4.1 Generality of π

Since procedure *invalid_impl*/2 uses only the first branch node of a literal-labelled tree enumerated by *branch_node*/2, we can implement *branch_node*/2 as the following without compromising the soundness and the completeness of π.

```
branch_node(T,N):- branch_node_1(T,N).
```

where the specification for *branch_node_1*/2 is that *branch_node_1*(T, N) succeeds once and only once with N being a branch node of T. The different implementations of *branch_node_1*/2 will result in different performances of the inconsistency diagnosis procedure *inconsistency*/2 in terms of the quantity of queries.

A top-down inconsistency diagnosis procedure based on tree search [4, 14, 20] can be obtained by using an implementation of $branch\_node\_1/2$ s.t. $branch\_node\_1(T, N)$ succeeds with N being a child of the root of T. A bottom-up inconsistency diagnosis procedure based on tree search [20] can be obtained by using an implementation of $branch\_node\_1/2$ s.t. $branch\_node\_1(T, N)$ succeeds with N being the parent node of a leaf node of T. The divide-and-query inconsistency diagnosis procedure [18, 19] can be obtained through an implementation of $branch\_node\_1/2$ s.t. $branch\_node\_1(T, N)$ succeeds with N being a node of T s.t. $|n(T_N) - n(T)/2| \leq |n(T_{N'}) - n(T)/2|$ for any other node N' of T. The top-down zooming inconsistency diagnosis procedure [14] can be obtained by using an implementation of $branch\_node\_1/2$ s.t. $branch\_node\_1(T, N)$ succeeds with N being a node of T satisfying either (1) N is a node of T other than the root of T, and (2) N and the root of T have the same predicate name, and (3) N is not subordinate to any node of T that satisfies (1) and (2), or N is a child of the root of T when no node of T satisfies (1) and (2).

Similarly, we can implement $non\_root/2$ as the following without affecting the soundness and the completeness of π.

```
non_root(T,N):- non_root_1(T,N).
```

where the specification for $non\_root\_1/2$ is that $non\_root\_1(T, N)$ succeeds once and only once s.t. N is a node of T other than the root node of T. The different implementations of $non\_root\_1/2$ will result in different performances of the insufficiency diagnosis procedure $insufficiency/2$ in terms of the quantity of queries.

A top-down insufficiency diagnosis procedure can be obtained by using an implementation of $non\_root\_1/2$ s.t. $non\_root\_1(T, N)$ succeeds with N being a child of the root of T. A bottom-up insufficiency diagnosis procedure can be obtained by using an implementation of $non\_root\_1/2$ s.t. $non\_root\_1(T, N)$ succeeds with N being a leaf node of T.

$insufficiency/2$ can be specialised resulting in a divide-and-query insufficiency diagnosis procedure through an implementation of $non\_root\_1/2$ s.t. $non\_root\_1(T, N)$ succeeds with N being a node of T s.t. $|n(T_N) - n(T)/2| \leq |n(T_{N'}) - n(T)/2|$ for any other node N' of T.

The formulation of π not only enables standard tree search strategies such as top-down, bottom-up and divide-and-query to be used, but also allows more flexible strategies to be used as long as these strategies conform to the specifications for $branch\_node\_1/2$ and $non\_root\_1(T, N)$. This provides us with a platform for evaluating various strategies as well as tailoring the declarative diagnoser to a user who prefers a particular search strategy.

4.2 Search space

The search space of a declarative diagnoser is one of the major factors that affect the quantity of queries. We briefly compare the search space of our declarative diagnoser π with the search spaces of the declarative diagnosers in the literature.

Suppose that A is the inconsistency symptom of P w.r.t. I to be diagnosed. An inconsistency diagnosis procedure based on tree search [4, 14, 18, 19, 20], including the divide-and-query diagnoser [18, 19], is a specialised version of $inconsistency/2$. These inconsistency diagnosis procedures search for an inconsistent clause instance of P w.r.t. I in the set of all the clause instances that are used in one successful derivation of $P \cup \{\leftarrow A\}$, that is, the clause instances that are used in one proof tree for P and A. The search space of $inconsistency/2$ is the set of all the clause instances that are used in one *I-cpp* for P and A. Because an *I-cpp* for P and A is smaller than a proof tree for P and A, $inconsistency/2$ has a smaller search space.

There are inconsistency diagnosis procedures that are not based on tree search [8, 12, 13, 18, 19, 22]. Such inconsistency diagnosis procedures search a larger space than $inconsistency/2$. We exemplify this through the single stepping inconsistency diagnosis procedure [18, 19]. The single stepping inconsistency diagnosis procedure simulates Prolog's execution of A. Whenever a call A' has been executed successfully with a computed answer θ, the oracle is asked if $A'\theta$ is valid in I. If $A'\theta$ is valid in I, the single stepping inconsistency diagnosis procedure continues to simulate Prolog's execution of the remaining calls. Otherwise, $A'\theta$ is invalid in I. Let $C = H' \leftarrow L'_1, L'_2, \cdots, L'_m$ be the clause instance used to solve A'. $(L'_1, L'_2, \cdots, L'_m)\theta$ is already known to be valid in I. Therefore, $C\theta$ is an inconsistent clause instance of P w.r.t. I. Let R' be the left-to-right computation rule. The single stepping inconsistency diagnosis procedure searches for an inconsistent clause instance of P w.r.t. I in the set of all the clause instances used in the first successful SLDNF derivation of $P \cup \{\leftarrow A\}$ via R' and all the clause instances used in all the unsuccessful SLDNF derivations of $P \cup \{\leftarrow A\}$ via R' that are previous to the successful SLDNF derivation of $P \cup \{\leftarrow A\}$. The other inconsistency diagnosis procedures that are not based on tree search can be shown to have larger spaces than $inconsistency/2$ as well.

Suppose that A is the insufficiency symptom of P w.r.t. I to be diagnosed. $insufficiency/2$ searches for an incompletely covered atom of P w.r.t. I in the set of all the selected atoms of all the nodes of an *I-cps* for $P \cup \{\leftarrow A\}$ via a fixed computation rule R. The search space of the insufficiency diagnosis procedures presented in [4, 14, 16, 17] is the set of all the selected atoms of all the nodes of a SLDNF tree for $P \cup \{\leftarrow A\}$. Because an *I-cps* for $P \cup \{\leftarrow A\}$ is smaller than a SLDNF tree for $P \cup \{\leftarrow A\}$, $insufficiency/2$ has a smaller search space than these insufficiency diagnosis procedures.

The insufficiency diagnosis procedures presented in [8, 12, 13, 22] have larger search spaces than $insufficiency/2$ because, given an insufficiency symptom A of P w.r.t. I, their search spaces are larger than the set of all the selected atoms of all the nodes of a SLDNF tree for $P \cup \{\leftarrow A\}$. See [15] for a detailed analysis of the search spaces of the insufficiency diagnosis procedures in the literature.

4.3 Search space pruning versus oracle automation

The objective of reducing the quantity of queries has also been pursued by fully or partly automating the oracle. Diagnosers in [2, 6] use a full specification. A full specification makes it possible to completely avoid querying the user because any query about the intended interpretation I can be answered by using the specification. Diagnosers in [3, 4, 10] use assertions about the intended interpretation I to answer queries. Whenever a query is necessary, these diagnosers will try to answer the query by using only the assertions. Those queries that cannot be answered this way are imposed on the oracle. The assertions are descriptions of the acquired part I' of the intended interpretation I.

Given an inconsistency symptom A of P w.r.t. I, our diagnoser constructs an *I-cpp* for P and A using I'. The effect is equivalent to pruning a proof tree for P and A before it is searched. Similarly, given an insufficiency symptom A of P w.r.t. I, our diagnoser constructs an I-cps for $P \cup \{\leftarrow A\}$ using I'. The effect is equivalent to pruning a SLDNF tree for $P \cup \{\leftarrow A\}$ before it is searched.

Using I' to prune the search space before it is searched rather than to answer queries makes sense. Firstly, a smaller search space means a smaller upper bound for the quantity of queries. See [13] for a detailed analysis for inconsistency diagnosis procedures. Secondly, given an inconsistency symptom A of P w.r.t. I, if a proof tree for P and A is searched, then a search strategy may select a node L and query the oracle about the validity of L if the validity of L cannot be decided by using I'. If L is subordinate to another node L' in the proof tree s.t. L' is valid in I', then an *I-cpp* for P and A that is constructed using I' will not contain node L because L' is valid in I'. Therefore, this query can be spared. A similar argument applies to insufficiency diagnosis. This does not apply to the top-down search strategy. However, the top-down strategy may not be either the optimal strategy for the diagnosis problem at hand or the strategy preferred by the user. We share with [5] the opinion that the user should be allowed to choose their own strategy.

5 Conclusion

We have presented the generic declarative diagnoser π for normal logic programs and established its soundness and its completeness. π is generic in the sense that it can be used with various tree search strategies. π has a smaller search space than the declarative diagnosers in the literature when diagnosing an inconsistency or insufficiency symptom of P w.r.t. I.

References

1. P. Deransart. Proofs of Declarative Properties of Logic Programs. In J.Diaz and F.Orejas, editors, *Proceedings of International Joint Conference on TAPSOFT'89*, pages 207–226, Barcelona, Spain, March 1989.
2. 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.
3. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. The Use of Assertions in Algorithmic Debugging. In ICOT, editor, *The Proceedings of the International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

4. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic Debugging with Assertions. In Harvey Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 502–521. The MIT Press, 1989.
5. M. Ducassé. Opium$^+$, a Meta-Debugger for Prolog. In Y. Kodratoff, editor, *Proceedings of the eighth ECAI*, pages 272–277, Münich, August 1-5 1988. Pitman.
6. 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.
7. M. Falaschi, G. Levi, and C. Palamidessi. Declarative Modelling of the Operational Behavior of Logic Programs. *Theoretical Computer Science*, 69:289–318, 1989.
8. 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.
9. A.M.J. Hutching, D.L. Bowen, L. Byrd, P.W.H. Chung, F.C.N. Pereira, L.M. Pereira, R.Rae, and D.H.D. Warren. Edinburgh Prolog (the new implementation) user's manual. AI Applications Institute, University of Edinburgh, 8 October 1986.
10. T. Kanamori, T. Kawamura, M. Maeji, and K.Horiuchi. Logical Program Diagnosis from Specifications. ICOT Technical Report TR-447, March 1989.
11. Y. Lichtenstein and E. Shapiro. Abstract Algorithm Debugging. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 512–531. The MIT Press, 1988.
12. J.W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
13. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
14. M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Logic Programs. ICOT Technical Report TR-290, August 1987.
15. L. Naish. Declarative Diagnosis of Missing Answers. Technical Report 88/9 (Revised May 1991), Department of computer science, The University of Melbourne, May 1991.
16. 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.
17. 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.
18. E. Shapiro. Algorithmic Program Diagnosis. In *ACM Conference Record of the ninth annual ACM Symposium on Principles of Programming Languages*, pages 299–308, Albuquerque, New Mexico, Jan. 25-27 1982.
19. E. Shapiro. *Algorithmic Debugging*. The MIT Press, 1983.
20. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.
21. M.H. van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Artificial Intelligence*, 23(10):733–742, 1976.
22. S.Y. Yan. Foundations of Declarative Debugging in Arbitrary Logic Programming. *International Journal of Man Machine Studies*, 32:215–232, 1990.

This article was processed using the LaTeX macro package with LLNCS style