Intensional Algorithmic Debugging : Preliminary Version

Kannan Govindarajan Bharat Jayaraman
Department of Computer Science
State University of New York
Buffalo, NY 14260
E-mail : {govin-k,bharat}@cs.buffalo.edu

May 24, 1994

Abstract

Declarative error diagnosis of logic programs was introduced in [Sh82]. Other work in error diagnosis has focused on adding various kinds of control to the declarative diagnoser [Av84], or extending the debugging to a larger class of programs [Pe86]. In all this work the error that is detected is either an incorrect clause instance, which represents an incorrect implication in the intended interpretation, or there is a goal on which the program fails when it should succeed because the clauses in the program don't "cover" it. In this work we refine the notion of an error so that the exact place in the clause where the error occurs can be identified. This is made possible by reasoning with the intensional semantics of the program rather than the extensional semantics.

Key Words: Dec;arative semantics, interpretations, models, debugging, incorrectness, incompleteness, extensional semantics, intensional semantics.

1 Introduction

The need for program debugging arises in every programming paradigm. The debugging problem is particularly critical in logic programming languages, since a logic program usually undergoes non-trivial changes before it is finally executed. These compile-time transformations are necessary to improve the efficiency of the program and are quite complicated. Although the known techniques in algorithmic debugging are independent of the compile-time transformations, they only point out the clause that is causing the error and go no further. One goal of this work is to refine the techniques so that one can point to the exact place in a wrong clause where the error occurs. For example, suppose we had the following "wrong" *append* program:

$$\begin{aligned} append([\ ], X, X) &\leftarrow \\ append([X|Y], Z, A) &\leftarrow append(Y, Z, A). \end{aligned}$$

This wrong *append* program incorrectly succeeds on a goal like:

| ?- $append([1],[2,3],[2,3])$

The known error diagnosis techniques, would report

$$append([1],[2,3],[2,3]) \quad \leftarrow \quad append([],[2,3],[2,3]) \tag{1}$$

as an *incorrect clause instance* because the body of this instance of a clause in the program is in the *intended interpretation* but the head is not. We claim that one can go further and say that the *intended clause* was

$$append([1],[2,3],[1,2,3]) \quad \leftarrow \quad append([],[2,3],[2,3]) \tag{2}$$

Comparing (1) with (2) allows us to conclude that the error is in the head of the incorrect clause. The exact place of error can be found by comparing the heads.

On the other hand, if we the following "wrong" append program:

$$\begin{array}{rcl} append([\,],X,X) & \leftarrow & \\ append([X|Y],Z,[X|A]) & \leftarrow & append(Y,[Z],A). \end{array}$$

it succeeds on the following goal incorrectly.

| ?- $append([1],[2],[1,[2]])$

The known techniques would report

$$append([1],[2],[1,[2]]) \quad \leftarrow \quad append([],[[2]],[[2]]) \tag{3}$$

as an *incorrect clause instance*, but the *intended clause* was

$$append([1],[2],[1,[2]]) \quad \leftarrow \quad append([],[2],[[2]]) \tag{4}$$

Comparing (3) and (4), we can similarly conclude that the error this time is in the body of the clause. In a more general setting there could be more goals in the body, but the comparison can still be made.

Note that the intended clause in the second case is not

$$append([1],[[2]],[1,[2]]) \quad \leftarrow \quad append([],[[2]],[[2]]) \tag{5}$$

because the goal that was invoked was $append([1],[2],[1,[2]])$ and there is a clause in the *intended program* whose head unifies with the goal.

Given that $H \leftarrow B_1,\ldots,B_n$ is the incorrect clause instance, that the error diagnosis techniques produce when the goal invoked was G, it could be the case that:

1. H should not be called when a call is made to G in the *intended program*.
2. If H should be called when a call is made to G, it could be that there should be no further calls because the *intended program* has no clauses whose heads unify with H.
3. There could be clauses in the *intended program* whose heads unify with H.

If we know what the intended clause is, then the exact error in the incorrect clause can be accurately pointed out. This information is obtained by querying the user for information which requires knowledge of the ground instances of clauses in the intended program. It is not unreasonable to expect the programmer to know this information because each clause is a encodes a relation and the programmer should know ground instances of the relation.

The rest of the paper is orgranised as follows: Section 2 briefly describes algorithmic debugging of relational programs as described in [Sh82] and [Ll87]. Section 3 describes our technique to locate the error in the incorrect clause. It introduces a relation among atoms in the Herbrand Base forced by any program, which is the *supportedness* relation. If a program is incorrect with respect to some interpretation, then the relation forced by the program and the relation forced by the intended program are different. This enables us to use the supportedness relation as a means to get at the exact error in incorrect clauses. Section 4 has our conclusions and suggestions for directions for future work.

2 Algorithmic Program Debugging

This section describes the techniques used in declarative error diagnosis of relational programs. We assume that the reader is familiar with concepts like models, interpretations, declarative semantics, fixed point semantics, completion of programs, SLDNF search trees, computed answers, correct answers etc.. A good treatment of this material can be found in [Ll87].

- Let $\mathbf{P}$ be a definite clause program.
- Let $\mathbf{B_P}$ denote its Herbrand Base.
- Let $\tau_\mathbf{P}$ denote the immediate consequence operator defined by the clauses in the program $\mathbf{P}$.
- Let gfp($\tau_\mathbf{P}$) denote the greatest fixed point of the operator $\tau_\mathbf{P}$ and lfp($\tau_\mathbf{P}$), its least fixed point.
- Let $F_\mathbf{P}$ denote the finite failure set of the program $\mathbf{P}$ and
- Let comp($\mathbf{P}$) denote the Clark completion of the program.

Definition 1 *Let* $\mathbf{P}$ *be a program and* $\mathbf{I} \subseteq \mathbf{B_P}$ *an intended interpretation for* $\mathbf{P}$. *We say* $\mathbf{P}$ *is correct wrt* $\mathbf{I}$ *if* $\mathbf{I}$ *is a model for comp(*$\mathbf{P}$*), otherwise we say* $\mathbf{P}$ *is incorrect wrt* $\mathbf{I}$.

If a program $\mathbf{P}$ is *correct* wrt an interpretation $\mathbf{I}$, it does not mean that $\mathbf{P}$ computes everything in $\mathbf{I}$. For instance, $\mathbf{P}$ could be a definite clause program such that $\mathrm{lfp}(\tau_{\mathbf{P}}) \neq \mathrm{gfp}(\tau_{\mathbf{P}})$, then $\mathbf{P}$ is correct wrt $\mathrm{gfp}(\tau_{\mathbf{P}})$, but $\mathbf{P}$ does not compute all atoms in $\mathrm{gfp}(\tau_{\mathbf{P}})$.

Proposition 1 *If* $\mathbf{P}$ *is a terminating definite clause program, then* $\mathrm{lfp}(\tau_{\mathbf{P}}) = \mathrm{gfp}(\tau_{\mathbf{P}})$.

Proof If $\mathbf{P}$ is a terminating program, then given any ground goal $\leftarrow G$, either $\mathbf{P}$ succeeds on the goal $\leftarrow G$ and returns an answer θ such that $\mathbf{P} \models G\theta$, or $\mathbf{P}$ finitely fails on goal $\leftarrow G$. In other words if $\mathbf{P}$ is a terminating program then $\mathrm{lfp}(\tau_{\mathbf{P}}) \cup F_{\mathbf{P}} = \mathbf{B_P}$. Clearly $\mathrm{lfp}(\tau_{\mathbf{P}}) \cap F_{\mathbf{P}} = \phi$. Now we know that $\mathrm{gfp}(\tau_{\mathbf{P}})$ is also a model for the program $\mathbf{P}$, but it cannot contain any more ground atoms than $\mathrm{lfp}(\tau_{\mathbf{P}})$ because if $\mathbf{P}$ is terminating then $\overline{\mathrm{lfp}(\tau_{\mathbf{P}})} = F_{\mathbf{P}}$ and hence $\forall x \in \mathbf{B_P}, x \in \mathrm{gfp}(\tau_{\mathbf{P}}) \Rightarrow x \in \mathrm{lfp}(\tau_{\mathbf{p}})$. This means that $\mathrm{gfp}(\tau_{\mathbf{P}}) \subseteq \mathrm{lfp}(\tau_{\mathbf{P}})$. We already know that $\mathrm{lfp}(\tau_{\mathbf{P}}) \subseteq \mathrm{gfp}(\tau_{\mathbf{P}})$ for any definite clause program. Therefore if $\mathbf{P}$ is terminating then $\mathrm{lfp}(\tau_{\mathbf{P}}) = \mathrm{gfp}(\tau_{\mathbf{P}})$. □

An error occurs in a terminating program if one of two things happen.

1. The program computes a wrong answer, i.e. succeeds on a goal when it should fail.
2. The program misses an answer, i.e. finitely fails when it should have succeeded.

Proposition 2 *Let* $\mathbf{P}$ *be a program,* $\mathbf{G}$ *be a goal* $\leftarrow \mathbf{W}$, *and* $\mathbf{I}$ *be an interpretation for* $\mathbf{P}$.

1. *If* θ *is a computed answer for* $\mathbf{P} \cup \{\mathbf{G}\}$ *and* $\mathbf{W}\theta$ *is not true in* $\mathbf{I}$, *then* $\mathbf{P}$ *is incorrect in* $\mathbf{I}$. *An atom that belongs to* $\mathrm{lfp}(\tau_{\mathbf{P}})$ - $\mathbf{I}$ *is called an incorrectness symptom for* $\mathbf{P}$.
2. *If* $\mathbf{P} \cup \{\mathbf{G}\}$ *has a finitely failed SLDNF-tree and* $\mathbf{W}$ *is true in* $\mathbf{I}$, *then* $\mathbf{P}$ *is incorrect in* $\mathbf{I}$. *An atom that belongs to* $\mathbf{I}$ - $\mathrm{gfp}(\tau_{\mathbf{P}})$, *is called an insufficiency symptom for* $\mathbf{P}$.

Example 1 Consider the following definition of the "naive" *reverse* predicate.

$$\begin{array}{rcl}
reverse([\,],[\,]) & \leftarrow & \\
reverse([X|Y],Z) & \leftarrow & reverse(Y,T), append(T,[X],Z) \\
append([\,],X,X) & \leftarrow & \\
append([X|Y],Z,T) & \leftarrow & append(Y,Z,T)
\end{array}$$

The program is "erroneous" because it would succeed on the goal $\leftarrow reverse([u,v],[u])$
The atom $reverse([u,v],[u])$ is said to be an *incorrectness symptom* and the cause of this symptom is the instance $append([v],[u],[u]) \leftarrow append([\,],[u],[u])$ of the second clause for *append*.

Second Example Suppose instead we had the following definition for "naive" *reverse*:

$$
\begin{aligned}
reverse([\,],[\,]) &\leftarrow \\
reverse([X|Y],Z) &\leftarrow reverse(Y,T), append(T,[X],Z) \\
append([X|Y],Z,[X|T]) &\leftarrow append(Y,Z,T)
\end{aligned}
$$

No atomic formula of the form $reverse([u,v],X)$ is a logical consequence of the above program. The *intended interpretation* for *reverse* has the atomic formula $reverse([u,v],[v,u])$. The atom $reverse([u,v],[v,u])$ is said to be an *uncovered atom*

Definition 2 *Let* **P** *be a definite clause program and* **I** *an intended interpretation for* **P**. *Let A be an atom with the predicate symbol p. We say that A is an uncovered atom for* **P** *wrt* **I** *if A is true in* **I**, *and for every program statement A' ← W, in the definition of p such that A and A' unify with mgu θ, we have that Wθ is false in* **I**.

Definition 3 *Let* **P** *be a definite clause program and* **I** *an interpretation for* **P**. *We say that an instance A ← W of a program statement in* **P** *is an incorrect statement instance for* **P** *wrt* **I** *if A is false in* **I** *and W is true in* **I**.

We note that every instance of an uncovered atom is uncovered, and every instance of an incorrect statement is incorrect.

Proposition 3 **P** *is incorrect wrt* **I** *iff there is an uncovered atom for* **P** *wrt* **I** *or there is an incorrect statement instance for* **P** *wrt* **I**.

Proof Suppose that there is an incorrect statement instance for **P** wrt **I**. It is easy to see that **I** does not satisfy the *if* part

$$\forall x_1 \ldots \forall x_n \; (\mathrm{p}(x_1, \ldots, x_n) \leftarrow E_1 \vee \ldots \vee E_k)$$

of the completed definition of p, hence **P** is incorrect wrt **I**. Next suppose that there is an uncovered atom $\mathrm{p}(s_1, \ldots, s_n)$ for **P** wrt **I**. If there is no definition for p, then **P** is incorrect wrt **I**. Otherwise it does not satisfy the *only-if* part

$$\forall x_1 \ldots \forall x_n \; (\mathrm{p}(x_1, \ldots, x_n) \rightarrow E_1 \vee \ldots \vee E_k)$$

of the completed definition of p, hence **P** is incorrect wrt **I**.

Now suppose that **P** is incorrect wrt **I**. Note that any normal Herbrand interpretation for comp(**P**) is a model for the equality theory of comp(**P**) and hence **I** cannot be a model for the remainder of comp(**P**). If **I** does not satisfy a completed definition of the form

$$\forall x_1 \ldots \forall x_n \; \tilde{\mathrm{p}}(x_1, \ldots, x_n)$$

then there is an uncovered atom. If **I** does not satisfy the *only-if* part

$$\forall x_1 \ldots \forall x_n \; (\mathrm{p}(x_1, \ldots, x_n) \rightarrow E_1 \vee \ldots \vee E_k)$$

of a completed definition, then there is an uncovered atom. Finally, if **I** does not satisfy the *if* part

$$\forall x_1 \ldots \forall x_n \; (\mathrm{p}(x_1, \ldots, x_n) \leftarrow E_1 \vee \ldots \vee E_k)$$

of a completed definition, then there is an incorrect statement instance. □

2.1 Observations about declarative error diagnosis

Every clause in the completed program is a two-way implication, and if the intended interpretation is not a model for the clauses in the program, then there must be an implication for which the intended interpretation is not a model. The only way the intended interpretation cannot be a model for an implication is when the antecedent of the implication is true in the intended interpretation but the consequent is not. This is exactly what the diagnoser is checking for, for arriving at incorrect statement instances and uncovered atoms. Notice further that the diagnoser only asks if some atoms are in the intended interpretation or not. The declarative diagnoser therefore takes an extensional view of the predicates in the program. Such a diagnoser will not be able to refine the error beyond an incorrect clause because doing that requires an intensional view of the predicates.

3 Locating the Exact Error in a Clause

3.1 A More Refined Notion of Error

- Let $\mathbf{P_I}$ denote the program whose declarative semantics coincides with the intended interpretation **I**. In other words let $\mathbf{P_I}$ denote the intended program.

- Let $\tau_{\mathbf{P_I}}$ denote the immediate consequence operator of $\mathbf{P_I}$.

Definition 4 *Given a definite clause program* $\mathbf{P}$*, let ground(*$\mathbf{P}$*) denote the set of ground instances of the clauses in* $\mathbf{P}$*.*

If $\mathbf{P}$ is a definite clause program which is incorrect with respect to the intended interpretation $\mathbf{I}$, and $\mathbf{P_I}$ is the intended program, clearly $ground(\mathbf{P}) \neq ground(\mathbf{P_I})$. In particular if s = $H \leftarrow B_1, \ldots, B_n$ is an *incorrect statement instance* of $\mathbf{P}$ wrt $\mathbf{I}$, then we know s $\notin$ *ground(*$\mathbf{P}_{\underline{I}}$*)*. There are three possibilities:

1. $\exists s_1 \in ground(\mathbf{P_I})\ \ s_1 = H \leftarrow C_1, \ldots, C_n$ and $\exists i$ such that $C_i \neq B_i$. In other words, there is a clause in the intended program whose head matches H, but atleast one goal in the body is different from the corresponding goal in the body of the *incorrect statement instance*.
2. $\exists s_2 \in ground(\mathbf{P_I})\ \ s_2 = H_1 \leftarrow B_1, \ldots, B_n$ and $H \neq H_1$. In other words, the body of the incorrect clause *supports* an atom different from H in the *intended program*.
3. $\forall s_3 \in ground(\mathbf{P_I})\ \ s_3 = H_2 \leftarrow C_1, \ldots, C_m,\ \{C_1, \ldots, C_m\} \neq \{B_1, \ldots, B_n\}$ and $H \neq H_2$. In other words, every ground instance of a clause in the *intended program* is such that both the head and the body of every clause are different from the head and body of the *incorrect statement instance*.

It is clear that at least one of the three possibilities *has* to occur. However there could be some ambiguity because possibilities (1) and (2) could occur simultaneously. This could be seen in the example in the introduction. We use the notion of *supportedness* among atoms in the Herbrand Base with respect to the *intended program* to disambiguate the ambiguity. Note that this is different from the notion of supportedness of models of logic programs.

Definition 5 *An atom B is said to support an atom A wrt program* $\mathbf{P}$ *if*

- $A \leftarrow B'_1, \ldots, B, \ldots, B'_n$ *is an instance of a program clause in* $\mathbf{P}$*. In this case we say that B strongly supports A.*
- $A \leftarrow B_1, \ldots, B_n$ *is an instance of a program clause in* $\mathbf{P}$ *and B supports at least one of* $\{B_1, \ldots, B_n\}$*.*

Definition 6 *An atom A is said to depend on an atom B wrt a program* $\mathbf{P}$ *if atom B supports atom A wrt program* $\mathbf{P}$ *and an atom A is said to strongly depend on an atom B wrt a program* $\mathbf{P}$ *if atom B strongly supports atom A wrt program* $\mathbf{P}$*.*

Basically, an atom B *supports* an atom A wrt a program $\mathbf{P}$, if according to program $\mathbf{P}$ a call to A eventually results in a call to B.

Definition 7 *If an atom B supports an atom A wrt program* $\mathbf{P}$*, the sequence of atoms* $\langle A, G_1, G_2, \ldots, G_n, B\rangle$ *is said to be a support path from A to B if* $\forall i, 1 < i < n, B_{i+1}$ *supports* B_i*,* B_1 *supports* A*, and* B *supports* B_n*.*

We know that the program $\mathbf{P}$ is incorrect wrt the interpretation $\mathbf{I}$. The diagnoser when invoked with the "incorrect goal" G, finds an incorrect statement instance $H \leftarrow B_1, B_2, \ldots, B_n$. Let H be of the form $p(x_1, x_2, \ldots, x_m)$. We know that H is not in lfp($\tau_{\mathbf{P_I}}$), but $\{B_1, \ldots, B_n\} \subseteq$lfp($\tau_{\mathbf{P_I}}$). There are two possible cases:

1. H *supports* G wrt the program $\mathbf{P_I}$.
2. H does not *support* G wrt the program $\mathbf{P_I}$.

We know that H *supports* G wrt the program $\mathbf{P}$ and that the program $\mathbf{P}$ is wrong.

When H *supports* G **wrt the program** $\mathbf{P_I}$ **:-** In this case it must be that either:

1. $H \leftarrow G_1, \ldots, G_l$ must be the instance of some clause in the program $\mathbf{P_I}$ or
2. There is no clause in program $\mathbf{P_I}$ whose head unifies with H.

Case 1 But we know that H is not in lfp($\tau_{\mathbf{P_I}}$), so it must be the case that for every such instance, at least one of G_i is not in lfp($\tau_{\mathbf{P_I}}$). In other words, if H supports the goal G in the intended program too, then the error is in the body of the clause. The user program $\mathbf{P}$ incorrectly makes H depend on an atom which is in the intended interpretation, but in the intended program H depends on some other atom which is not in the intended interpretation.

Suppose for example, we had the following "wrong" append program:

$$\begin{aligned} append([\,], X, X) &\leftarrow \\ append([X|Y], Z, [X|A]) &\leftarrow \quad append(Y, [Z], A) \end{aligned}$$

The program incorrectly succeeds on a query of the form:
$|? - append([1], [2], [1, [2]])$.

The diagnoser would report $append([1], [2], [1, [2]]) \leftarrow append([\,], [[2]], [[2]])$ as an incorrect statement instance as $append([1], [2], [1, [2]])$ is not in the intended interpretation $\mathbf{I}$, but append([ ],[[2]],[[2]]) is in $\mathbf{I}$. So in this example $G = H = append([1], [2], [1, [2]])$. The goal G_1 in the body of the incorrect statement instance found by the diagnoser is $append([\,], [[2]], [[2]])$. But in the intended program $\mathbf{P_I}$, the atom $append([1], [2], [1, [2]])$ *depends* on the atom $append([\,], [2], [[2]])$ and not on $append([\,], [[2]], [[2]])$. The goal B_1 in the body of the clause in $\mathbf{P_I}$ whose head unifies with H is $append([\,], [2], [[2]])$. So by comparing B_1 and G_1, we know that the error is in the second argument in the goal in the body of the clause.

Case 2 We know that in the user program $\mathbf{P}$, H is *supported* by goals $B_1, \ldots, B_n$, so if there is no clause for in $\mathbf{P_I}$ whose head unifies with H, there are two possibilities.

1. The body of some clause in $\mathbf{P_I}$ unifies with $B_1, \ldots, B_n$ and the head of this instance of the clause is $H' \neq H$.
2. No body of any clause in $\mathbf{P_I}$ unifies with $B_1, \ldots, B_n$.

In the first case the error is in the head of the clause and in the second case both the head and the body of the clause are incorrect.
For example, if we had the following wrong *reverse* program:

$$\begin{aligned} reverse([\,],[\,]) &\leftarrow \\ reverse([X|Y], Z) &\leftarrow reverse(Y,T), append(T,[X],Z) \\ append([\,],X,X) &\leftarrow \\ append([X|Y],Z,A) &\leftarrow append(Y,Z,A) \end{aligned}$$

Then it succeeds on a query like
$|?-reverse([1,2],[1])$.

The diagnoser would come up with $append([2],[1],[1]) \leftarrow append([\,],[1],[1])$ as the incorrect statement instance. So in this example G is $reverse([1,2],[1])$ and H is $append([2],[1],[1])$. It is easily seen that in the intended program H does indeed support G. But in the intended program there is no clause for *append* whose head unifies with $append([2],[1],[1])$, but $append([\,],[1],[1])$ *supports* an atom of the form $append([X],[1],[X,1])$ in the intended program.

When $H = p(x_1, \ldots, x_m)$ **does not** *support* G **:-** In this case too there are two possibilities

1. The body of some clause for predicate p in the intended program unifies with the body of the incorrect clause in the user program.
2. None of the bodies of the clauses for p in the intended program unify with the body of the incorrect clause in the user program.

Case 1 When the body of some clause for p in the intended program unifies with the body of the incorrect clause, it means that the error is only in the head of the incorrect clause. For example, if we had the following wrong *reverse* program:

$$\begin{aligned} reverse([\,],[\,]) &\leftarrow \\ reverse([X|Y], Z) &\leftarrow reverse(Y,T), append(T,[X],Z) \\ append([\,],X,X) &\leftarrow \\ append([X|Y],Z,A) &\leftarrow append(Y,Z,A) \end{aligned}$$

Then it succeeds on a query like
$|? - reverse([1,2,3],[1])$.

The diagnoser would come up with $append([2],[1],[1]) \leftarrow append([\,],[1],[1])$ as the incorrect statement instance. So in this example G is $reverse([1,2,3],[1])$ and H is $append([2],[1],[1])$. In the intended program a call to $reverse([1,2,3],[1])$ does not result in a call to $append([2],[1],[1])$. So H does not *support* G wrt program $\mathbf{P_I}$. But there is a clause in the intended program whose body unifies with $append([\ ],[1],[1])$ and the atom it *supports* is of the form $append([X],[1],[X,1])$.

Case 2 This case means that there is no clause in the intended program whose head unifies with the head of the incorrect clause instance and there is no clause in the intended program whose body unifies with the body of the incorrect clause instance. In other words the error is both in the head and the body.

3.1.1 Soundness and Completeness of the Scheme

Soundness of Scheme

Proposition 4 *If the debugging scheme reports an error in the head (body) of a clause then the error is in the head (body) of the clause.*

Proof Suppose the top level goal on which the program $\mathbf{P}$ produced a wrong result is $\leftarrow G$ and incorrect clause instance found by the diagnoser is $H \leftarrow B_1, \ldots, B_n$. The scheme reports an error in the head of the incorrect clause if

1. H *supports* G wrt the intended program $\mathbf{P_I}$ and there is no clause in $\mathbf{P_I}$ whose head unifies with H but there is an instance of a clause in program $\mathbf{P_I}$ whose body unifies with $B_1, \ldots, B_n$ and whose head is $H' \neq H$.
2. H does not *support* G wrt program $\mathbf{P_I}$, and the body of some clause in $\mathbf{P_I}$ unifies with $B_1, \ldots, B_n$.

It is easy to see that in both these cases, there are instances of clauses in the user and the intended program such that

3.1.2 Observations about Focusing in on Error

The main idea behind the focusing in on the error is that an incorrect statement instance forces an implication of the form FALSE $\leftarrow$ TRUE in the intended interpretation. That is the body of the incorrect statement instance is TRUE in the intended interpretation but the head is not. The correct program must force the right implication. The implication could need rectification in one of three ways.

1. Only the head of the clause in the incorrect clause instance is wrong and should be changed so that the correct clause then forces an implication of the form TRUE $\leftarrow$ TRUE in the intended interpretation.
2. Only the body of the incorrect clause instance is wrong, and some goal in the body which is true in the intended interpretation should be changed so that the body is now false in the intended interpretation leading to an implication of the form FALSE $\leftarrow$ FALSE in the intended interpretation.
3. Both the head and the body are wrong and need to be changed, so that the implication now becomes TRUE $\leftarrow$ FALSE in the intended interpretation.

3.2 "Refining" Uncovered Atoms

If a program $\mathbf{P}$ is incorrect wrt an interpretation $\mathbf{I}$ because of an uncovered atom A, then we know that the atom A is in $\mathbf{I}$ but for every clause in the program $\mathbf{P}$ whose head unifies with A, it is the case that some goal in the body of the clause fails. So the interpretation is not a model for the completion of the program. To see this more clearly, suppose the clauses whose heads unified with A were the following:

$$\begin{array}{ccc} A_1 & \leftarrow & B_1 \\ \vdots & & \\ A_k & \leftarrow & B_k \end{array}$$

Then the completion of the program would force an implication of the form $A \rightarrow B_1, \vee, \ldots, \vee B_k$, but in $\mathbf{I}$, this implication is invalid as the antecedent is true and the consequent is false. When a program is incorrect because of an uncovered atom, it could be the case that a "new" clause might have to be added to "cover" the uncovered atom. Indeed this is the approach taken in [DeLe87]. In other words, A is uncovered because there should have been another applicable clause $A_{k+1} \leftarrow B_{k+1}$ where the goals in B_{k+1} are true and hence the atom A is covered by the "new" clause.

Our refinement scheme is applicable only when the cause of an uncovered atom is not the absence of a clause, but an "error" in one of the existing clauses.

Let the top level goal on which the program $\mathbf{P}$ produced an incorrect output be G, and let the debugger report A to be an uncovered atom, as the cause of the error. This means that A *supports* G wrt $\mathbf{P}$. In the intended program $\mathbf{P_I}$, there are two possibilities.

1. A does not support G wrt $\mathbf{P_I}$.
2. A supports G wrt program $\mathbf{P_I}$.

Case 1 If A does not support G wrt $\mathbf{P_I}$, then the error is in the body of the clause in which the goal A was invoked.

Case 2 If A does support G wrt $\mathbf{P_I}$, then let atoms $C_1, \ldots, C_n$ support A wrt $\mathbf{P_I}$. That is $A \leftarrow C_1, \ldots C_n$ is an instance of some clause in $\mathbf{P_I}$. We know that the bodies of all the clauses for A in $\mathbf{P}$ are different from this body for A. So we can find the exact error in the body for A in the program $\mathbf{P}$.

4 Conclusions and Directions for Future Work

In this paper, we have presented a simple extension to traditional algorithmic debugging techniques to isolate the error in incorrect clauses. This technique generalizes easily and can be used to debug programs written in any logic programming language whose operational semantics is based on an immediate consequence operator. For example, programs written in languages such as SEL [Jay92] can also be debugged using the techniques described here. We have implemented a simple debugger based on ideas presented in this paper and are using that to test out the feasibility of this approach to debugging programs.

References

[Av84] Av-Ron E., "Top-Down Diagnosis of Prolog Programs", M.Sc. Thesis, Weizmann Institute of Science, Rehovot Israel, (1984).

[DeLe87] Dershowitz N. and Lee Y., "Deductive Debugging", in *IEEE Symposium on Logic Programming*, 298-306, (1987).

[DrNaMa89] Drabent W., Nadjm-Tehrani S., Maluszyński J. , "Algorithmic Debugging with Assertions", in *Meta-Programming in Logic Programming*, Harvey Abramson and M.H. Rogers (eds.), MIT Press, 501-522, (1989).

[Fe85] Gérard Ferrand, "Error Diagnosis in Logic Programming, An Adaptation of E.Y.Shapiro's Method", Rapports de Recherche, INRIA Centre de Rocquencourt, (1985).

[Jay92] Bharat Jayaraman, "Implementation of Subset Equational Languages, *Journal of Logic Programming*, 12(4):299-324, (1992).

[Ll87] John W. Lloyd, "Foundations of Logic Programming", Springer-Verlag, (1987)

[Pe86] Pereira, L.M., "Rational Debugging in Logic Programming", '*Proc. Third Int'l Conf. on Logic Programming*, LNCS 225, Springer-Verlag, 272-280, (1986).

[Sh82] Ehud Y. Shapiro, "Algorithmic Program Debugging", Ph.D. Thesis, Yale University, New Haven, Connecticut, (1982).