Declarative Debugging

Gérard Ferrand and Alexandre Tessier

Université d'Orléans and INRIA (LOCO project)
LIFO, BP 6759, F-45067 Orléans Cedex 2, France
{ferrand|tessier}@lifo.univ-orleans.fr

1 Introduction

In program development, bugs localization is unavoidable. For high level languages such as logic programming languages, traditional tracing techniques become difficult to use because of the complexity of the computational behaviour. Moreover it would be incoherent to use only low level debugging tools whereas for theses languages the emphasis is on declarative semantics (as opposed to operational semantics).

From an intuitive viewpoint, we call *symptom* the appearance of an anomaly during the execution of a program. An anomaly is relative to *expected properties* of the program i.e. to some *intended semantics*. For example in logic programming a symptom can be a "wrong answer" but, because of the relational nature of logic programming, it can be also a "missing answer". Symptoms can be produced by testing.

Symptoms are caused by *errors* in the program. An error is a piece of code. Strictly speaking, error localization, when a symptom is given, is *error diagnosis* and it can be seen as a first step in *debugging*, a second step being error correction.

Declarative Debugging was introduced in Logic Programming by Shapiro (and called *Algorithmic Program Debugging* [39]). *Declarative* means that the user has no need to consider the computational behaviour of the logic programming system, he needs only a declarative knowledge of the expected properties of the program.

Unlike Shapiro, we do not consider infinite loops, which need operational notions. A survey on debugging methods in Logic Programming from a procedural view point is in [15, 16].

For a *declarative diagnoser system*, the input must include at least (1) the actual *program*, (2) the *symptom* and (3) some information on the *expected* semantics of the program. This information on the expected semantics can be given by the programmer during the diagnosis session or it can be specified by other means but, in any case, from a conceptual viewpoint, this information is given by an *oracle*.

2 Abstract Symptoms and Errors

Symptom and errors can be defined in an abstract scheme ([18, 32]). They have been first defined for definite programs by Shapiro ([39]) using essentially an inductive framework: on the one hand least fixpoint, i.e. *induction*, for the relation between incorrectness symptom ("wrong answer") and incorrectness (a kind of error), on the other hand greatest fixpoint, i.e. *co-induction*, for the relation between insufficiency symptom ("missing answer") and insufficiency (another kind of error).

In an abstract inductive framework, the notions of symptom and error, and the relation between them, come straightforwardly from fixpoint theory:

Let H be a set and $T : 2^H \to 2^H$ a *monotonic* operator. So $lfp(T)$ (the *least fixpoint* of T) is defined, and it is the least $I \subseteq H$ such that $T(I) \subseteq I$. So $T(I) \subseteq I \Rightarrow lfp(T) \subseteq I$ (it is the basis of the *proof method by induction*).

Let $S \subseteq H$ and let us suppose that we are expecting that $lfp(T) \subseteq S$ but that actually we see an $h \in lfp(T)$ such that $h \notin S$. Such an h shows that something goes wrong in T. h is called a *symptom* (for T wrt S). But $lfp(T) \not\subseteq S$ so $T(S) \not\subseteq S$.

Now we have to define a convenient notion of *error*. Let us suppose that T is the *operator defined by a set* $\mathcal{R}$ of *rules*, i.e. $T(I)$ is defined by: for any $h \in H$, $h \in T(I)$ iff there is a rule in $\mathcal{R}$ the head of which is h and the body is a subset of I (a rule is a pair denoted by $h \leftarrow B$ where $h \in H$ and $B \subseteq H$. h is the *head* and B is the *body* of the rule).

$T(S) \not\subseteq S$ means that there is a rule $h' \leftarrow B'$ in $\mathcal{R}$ with $B' \subseteq S$ and $h' \notin S$. The existence of such rules $h' \leftarrow B'$ explains that $lfp(T) \not\subseteq S$ i.e. the appearance of symptoms. So such a rule $h' \leftarrow B'$ is called an *error* (of $\mathcal{R}$ wrt S).

Moreover, the set of rules defines a notion of *proof tree* and the members of $lfp(T)$ are exactly the roots of proof trees. In particular if the rules are *finitary* (i.e. their bodies are finite sets) a *proof tree* for $\mathcal{R}$ is merely a *finite* tree where each node is (an occurrence of) a member of H and such that, for each node h, if its children are $b_1, \cdots, b_n$ then $h \leftarrow \{b_1, \cdots, b_n\}$ is a rule of $\mathcal{R}$; such a rule is said to be *in the proof tree*. (In particular if h is a *leaf* of the proof tree then $h \leftarrow \emptyset$ is in $\mathcal{R}$).

A relation of *"causality"* between an error and a symptom can be expressed with the notion of *proof tree*. It is easy to see that each symptom is the root of a proof tree in which there is a rule which is an error. This error can be seen as a "cause" of the symptom.

Fixpoint theory has also a *dual* aspect: $gfp(T)$ (the *greatest fixpoint* of T) is defined also, and it is the greatest $I \subseteq H$ such that $I \subseteq T(I)$. So $I \subseteq T(I) \Rightarrow I \subseteq gfp(T)$ (it is the basis of the *proof method by co-induction*).

Let $S \subseteq H$ and let us suppose that we are expecting that $S \subseteq gfp(T)$ but that actually we see an $h \in S$ such that $h \notin gfp(T)$. Such an h shows that something goes wrong in T. h is called a *co-symptom* (for T wrt S). But $S \not\subseteq gfp(T)$ so $S \not\subseteq T(S)$.

Let us suppose again that T is the *operator defined by a set* $\mathcal{R}$ of *rules*. $S \not\subseteq T(S)$ means that there is a $h' \in S$ such that in $\mathcal{R}$ there is no rule $h' \leftarrow B'$ with $B' \subseteq S$. Such a h' is called an *co-error* (of $\mathcal{R}$ wrt S).

There is again a relation between a co-error and a co-symptom that can be expressed with a notion of "partial proof tree" which cannot be "completed".

Lastly we can note that *no error and no co-error* means S is a *fixpoint* of T, and this implies $lfp(T) \subseteq S \subseteq gfp(T)$ (the converse is not valid).

3 Symptom and Error for Logic Programs

Obviously we have a first application of this scheme if we consider a *definite program* P, the Herbrand base H, i.e. the set of the ground atoms, and the rules which are the ground instances of the clauses of P (see [25, 3]). $lfp(T)$ is the least Herbrand model of P and it is also the set of all the ground atoms which are logical consequences of P. It is a formalization of its (ground, positive) *declarative semantics*.

S is a set-theoretical formalization of an expected property. Here a symptom for T wrt S is an atom which is a formalization of a ground *wrong answer* and it is called *incorrectness symptom* of P wrt S. An error is called *incorrectness* of P wrt S. So *no incorrectness* means S is a Herbrand model of P.

Note that the same scheme can also be applied if we consider another fixpoint formalization of the semantics of a program, for example with variables as [17, 12] or the *s-semantics* ([6]).

Of course in practice such a set S is not effectively available, it is only a set-theoretical formalization for the concept of *oracle* (Shapiro [39]) i.e. the way by which a diagnoser system can get knowledge on the expected properties formalized by S.

Moreover let us emphasize that the expected properties, which are here formalized by S, are not necessarily a complete specification for P. In other terms it is only expected that $lfp(T) \subseteq S$, the equality is not necessarily expected. So for example the expected properties may be about only the form of atoms, typing ... as [31, 37].

The dual notions of co-symptom and co-error occur with a formalization of the (ground) *negative* declarative semantics of the program P, because of the *finite failure set* of P which is included in $H - gfp(T)$. Here a co-symptom for T wrt S is called *insufficiency symptom* of P wrt S, it is an abstract notion which can be applied to "effective" (ground) *missing answer* i.e. when an atom is expected (in S) but is in the *finite failure set* of P.

A co-error is called *insufficiency* (or *uncovered atom*) of P wrt S.

The word *insufficiency* ([39]) (not *incompleteness*) stresses the difference between $S \subseteq gfp(T)$ and $S \subseteq lfp(T)$.

It is interesting to see why *insufficiency* is a *right notion of error* i.e. it explains the symptom in order to correct the program. In particular, another notion which explain only why there is a *finite failure* would not be a right notion of error because *finite failure* is a property of the program P alone, whereas the right notions of *symptom* and *error* must also depend on the expected properties of P, i.e. also on S.

The convenience of this theoretical approach is that the notions of symptom and error and the relation between them are clear in this inductive framework, and that this approach can be applied to symptoms which are effectively computed by SLD-resolution or finite failure.

In this framework with $lfp(T)$ on the one hand and $gfp(T)$ on the other hand, we can consider *positive* and *negative* information together. This framework can be generalized to logic programs with *negation* (*normal programs*) ([18, 4]). Fitting semantics ([19]) gives a logical view for this framework. Again each symptom is linked to some error but now any interaction is possible through negations between symptom and error, so *wrong* (resp. *missing*) answers are not necessary linked to *incorrectness* (resp. *insufficiency*).

In this framework we can also recover Lloyd's approach ([25, 24]) which is based on the logical semantics of Clark's completion, SLDNF resolution and a notion of *intended interpretation* I such that *no error* means *I is a model of the completed program.*

Another notion of error is used ([37, 14]): *not completely covered* atom (or *weak insufficiency*). It is an atom possibly with variables which has an instance which is an uncovered atom (i.e. an insufficiency). So this notion of error is weaker than insufficiency but it occurs in diagnosis algorithms (see below) in which the interaction with the oracle is less complex.

4 Diagnosis

Let us consider a proof tree rooted by an atom formalizing a wrong answer. So, the root is a member of $lfp(T)$ but is not expected; it is an incorrectness symptom of P with respect to the intended semantics S.

The diagnoser queries the oracle on the proof tree nodes: the oracle tells the diagnoser if nodes are members of S. It is straightforward to show inductively that some node is not expected while all its children are expected: the root is unexpected and the tree is finite. The rule linked to this node is an error of T wrt S. It corresponds to an incorrect clause instance, i.e. an incorrectness of P wrt S. The diagnoser can return the first error founded but also each error which occurs in the proof tree.

Each strategy in order to locate an incorrectness can be considered, for example the top-down strategy [39, 17, 24, 32]. Usually the search terminates when the first incorrectness is found, so changing the strategy can optimize the number of queries to the oracle (the programmer). Strategies better than the top-down (or bottom-up) are for example the divide-and-query strategy [39, 5], or some heuristics based strategies [38, 35].

Another way to reduce the number of queries to the programmer is to store its previous answers, then it is no more invoked to answer to a query subsumed by a previous answer [39, 28, 7].

A partial specification of the intended semantics can be given to the diagnoser in order to reduce again the number of queries [9, 14]. Drabent et al. [14] suggest to use assertions, which can be attached to the program predicates, as partial specification. In [13] an executable specification is used to generate test cases, locate bug and guide correction, using deductive and inductive inference techniques.

The advantage of declarative diagnosis wrt tracing techniques is now clear. First, the user does not need to understand the operational behaviour of the system (proof trees are intrinsic to the program. i.e. do not depend on the resolution strategy). Second, it follows the straight path from the symptom toward the error. Third, it displays only relevant informations, avoiding useless exploration due to the backtracking. Fourth, it helps the programmer to ask itself the relevant queries. Fifth, the output of the diagnoser is a faulty clause but also the condition of its incorrectness, formalized by an incorrect clause instance. Then it makes easier program repairing.

For the missing answer case, two kinds of diagnosis methods can be considered. First kind search for an

insufficiency (an uncovered atom) [39, 17, 24] while the second kinds search for a weak insufficiency (a non completely covered atom) [37, 14, 30].

The input of the first kind of diagnosers is an insufficiency symptom, i.e. an expected atom formalizing a missing answer.

Basically, the diagnoser try to build a proof tree rooted by the insufficiency symptom. Obviously, the attempt must fail (if the construction is fair). Assume we have build a partial proof tree. The diagnoser choose an hypothesis of the proof tree (a leaf which is not a fact) and ask to the oracle a clause instance whose head is the hypothesis in order to graft the rule to the leaf. When no clause instance is founded, the hypothesis is an uncovered atom.

A good strategy is for example to build the partial proof tree level by level. Several strategies could be used, guided by some heuristics, in order to choose the leaf and minimize the number of queries. Note that interaction with the oracle is quite complicated: it must provide some substitutions.

The input of the second kind of diagnosers is an incompleteness symptom, i.e an atom (possibly with variables) which has an expected instance formalizing a missing answer.

Interaction with the oracle is easier than interaction of the first kind of diagnosers. Indeed, it does not have to provide substitutions. It just answers yes or no to incompleteness questions [37] (clarified in [14]). The idea is to check if each expected instance of an atom is covered by the set of the answers to the atom.

But the diagnoser is more complicated. Moreover, generally, it assumes an SLD-tree without coroutining [30, 41] (or a standard SLD-tree [14]) from the incompleteness symptom (in order to have finite incompleteness questions). Naish in [30] discusses possible ways to remove the condition of non coroutining.

Rational Debugging [37] relies on term dependencies and makes cooperative use of the declarative and operational semantics using a notion of inadmissible calls.

The diagnosers described above have been implemented for several logic programming systems. The diagnosers are often meta-programs, and their implementation raise some problems of object program representation [21]. From this view point, Gödel have some facilities to handle programs at an object level. Binks developed a declarative diagnoser (GRADE) [5] for Gödel, written in Gödel, with support for abstract data types and coroutining. A component of the debugging system (NUDE) for NU-Prolog, implemented by Naish [34], is a declarative diagnoser. A declarative diagnoser using heuristics (HyperTracer environment) has been implemented for C-Prolog by Calejo [7].

5 Extensions

Declarative Debugging has been extended to some non declarative properties. For example, in a context of concurrent logic programming, Abstract Algorithmic Debugging ([23]) reduces the complexity of oracle queries using abstractions.

More recently Abstract Diagnosis [42, 8, 10, 9] extend declarative diagnosis to a method combining the s-semantics approach [6] and abstract interpretation techniques [11]. The method is based on the comparison of the specification S with $T(S)$ for a suitable operator T. For some observable properties (for example the set of computed answers) it gives a symptom-independent incorrectness diagnosis method. It gives also an incompleteness error diagnosis method for a large class of programs including acceptable programs [2] (taking advantage of the uniqueness of the fixpoint of T). The diagnosis is usually not effective because S is infinite, it becomes effective if a suitable finite abstract semantics is considered.

Declarative Debugging has been adapted for other kinds of programming languages: logico-functional languages such as NUE-Prolog ([33]), Escher ([26]). A typical feature of the Escher debugging framework is its simplicity, mainly because the computations are not described explicitly by a search tree but by equations and a single computation path. Other kinds of languages are considered: lazy functional languages ([36, 29]); even imperative languages including a subset of Pascal ([20]). In the scheme of [32], Naish describes a declarative debugger for an object oriented logic language.

Recent works extend declarative diagnosis to constraint logic programming. The main difficulties is that Herbrand interpretations do not represent program semantics any more. Moreover, few constraint domains have the Independence of Negated Constraints [27].

Naish announce in [32] a prototype implemented for CLP($\mathcal{R}$) based on its scheme. [41] provide a formal inductive framework based on the grammatical view [12] in order to extend declarative diagnosis to CLP. [22] (wrong answers) abstracts the constraint interpretation by a reject criterion in order to take into account incompleteness of the constraint solver. [40] (missing answers) extends the reject criterion to a cover relation and defines insufficiency and weak insufficiency in a unique inductive framework.

Another point is the "presentation problem" [26] which is concerned with finding ways of presenting large and complex oracle queries in such a way the programmer is able to answer correctly. This point is at least as significant in constraint logic programming as in pure logic programming.

6 Conclusion

Declarative Debugging is an attractive approach to debugging, but its practical success in program development depends on the declarativity level of the language. The drawbacks coming from the non-declarative nature of many features of traditional PROLOG-like language must be weakened in Constraint Logic Programming, mainly because of the added expressivity of CLP languages.

Declarative Debugging is one of the basic paradigms of the Long Term Research Project DiSCiPl ([1]), where it is combined with assertion-based methods and graphic tools.

References

[1] Debugging Systems for Constraint Programming. Long Term Research Project DiSCiPL, Reactive Scheme, Number 22532, 1996.

[2] K. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 106(1):109–157, 1993.

[3] K. R. Apt. *Handbook of Theoretical Computer Science*, volume 2, chapter Logic Programming, pages 493–574. Elsevier, 1990.

[4] M. Bergère. *Approche déclarative du diagnostic d'erreur pour la programmation en logique avec négation.* PhD thesis, Université d'Orléans, 1991.

[5] D. F. J. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.

[6] A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The S-Semantics Approach: Theory and Applications. *The Journal of Logic Programming*, 19&20:149–198, 1994.

[7] M. C. C. Calejo. *A Framework for Declarative Prolog Debugging.* PhD thesis, Universidade Nova de Lisboa, 1992.

[8] M. Comini, G. Levi, and G. Vitiello. Abstract Debugging of Logic Programs. In L. Fribourg and F. Turini, editors, *Logic Program Synthesis and Transformation and Metaprogramming*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, 1994.

[9] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J. Lloyd, editor, *International Logic Programming Symposium*, Logic Programming, pages 275–287. MIT Press, 1995.

[10] M. Comini, G. Levi, and G. Vitiello. Efficient Detection of Incompleteness Errors in the Abstract Debugging of Logic Programs. In M. Ducassé, editor, *Automated and Algorithmic Debugging*, pages 159–174. IRISA-CNRS, 1995.

[11] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[12] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.

[13] N. Dershowitz and Y.-J. Lee. Logical Debugging. *Journal of Symbolic Computation*, 15:745–773, 1993.

[14] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[15] M. Ducassé. A Pragmatic Survey of Automated Debugging. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1993.

[16] M. Ducassé and J. Noyé. Logic Programming Environments : Dynamic Program Analysis an Debugging. Technical Report 2618, INRIA, 1995.

[17] G. Ferrand. Error Diagnosis in Logic Programming: an adaptation of E. Y. Shapiro's method. *Journal of Logic Programming*, 4:177–198, 1987.

[18] G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.

[19] M. Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 2:295–312, 1985.

[20] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmeri. Generalized Algorithmic Debugging and Testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, 1992.

[21] P. M. Hill and J. W. Lloyd. Analysis of Meta-Programs. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*. MIT Press, 1989.

[22] F. Le Berre and A. Tessier. Declarative Incorrectness Diagnosis in Constraint Logic Programming. In P. Lucio, M. Martelli, and M. Navarro, editors, *Joint Conference on Declarative Programming*, pages 379–391, 1996.

[23] Y. Lichtenstein and E. Y. Shapiro. Abstract Algorithmic Debugging. In *Joint International Conference and Symposium on Logic Programming*, pages 512–530. MIT Press, 1988.

[24] J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154,, 1987.

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

[26] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.

[27] M. J. Maher. A Logic Programming view of CLP. In Warren, editor, *International Conference on Logic Programming*, pages 737–753. MIT Press, 1993.

[28] S. Nadjm-Tehrani. *Contributions to the Declarative Approach to Debugging Prolog Programs*. PhD thesis, Linkoping University, 1989.

[29] L. Naish. Declarative Debugging of Lazy Functional Programs. Technical Report 92/6, Department of Computer Science, University of Melbourne, 1992.

[30] L. Naish. Declarative Diagnosis of Missing Answers. *New Generation Computing*, 10(3):255–285, 1992. (Also available as Melbourne University Technical Report 88/9).

[31] L. Naish. *Types in Logic Programming*, chapter Types and the Intended Meaning of Logic Programs, pages 189–216. Logic Programming Series. MIT Press, 1992.

[32] L. Naish. A declarative debugging scheme. Technical Report 95/1, Department of Computer Science, University of Melbourne, 1995.

[33] L. Naish and T. Barbour. A Declarative Debugger for a Logical-Functional Language. Technical Report 94/30, Department of Computer Science, University of Melbourne, 1994.

[34] L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog Debugging Environment. In G. Levi and M. Martelli, editors, *International Conference on Logic Programming*, pages 521–536. MIT Press, 1989.

[35] A. E. Nicholson. Declarative Debugging of the Parallel Logic Programming Language GHC. In *Australian Computer Science Conference*, pages 225–236, 1988.

[36] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[37] L. M. Pereira. Rational Debugging in Logic Programming. In E. Y. Shapiro, editor, *International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.

[38] L. M. Pereira and M. Calejo. A Framework for Prolog Debugging. In *International Conference and Symposium on Logic Programming*, pages 481–495. MIT Press, 1988.

[39] E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1982.

[40] A. Tessier. Declarative Debugging in Constraint Logic Programming. In J. Jaffar, editor, *Asian Computing Science Conference*, volume 1023 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[41] A. Tessier. *A "Skeleton View" of Constraint Logic Programming and Declarative Error Diagnosis.* PhD thesis, University of Orléans, 1996 (to appear).

[42] G. Vitiello. *On the Abstract Diagnosis of Logic Programs.* PhD thesis, University of Salerno, 1996.