On Feasibility of Declarative Diagnosis

Włodzimierz Drabent
Institute of Computer Science, Polish Academy of Sciences
drabent *at* ipipan *dot* waw *dot* pl

The programming language Prolog makes declarative programming possible, at least to a substantial extent. Programs may be written and reasoned about in terms of their declarative semantics. All the advantages of declarative programming are however lost when it comes to program debugging. This is because the Prolog debugger is based solely on the operational semantics. Declarative methods of diagnosis (i.e. locating errors in programs) exist, but are neglected. This paper discusses their possibly main weaknesses and shows how to overcome them. We argue that useful ways of declarative diagnosis of logic programs exist, and should be usable in actual programming.

Keywords: declarative diagnosis / algorithmic debugging, Prolog, coroutining, program correctness, program completeness

1 Introduction

The main concept of logic programming is that a program is a set of formulae, and computation produces its logical consequences. Such program can be understood declaratively – not as a description of any computation, but rather as a description of a problem to solve. Its answers depend solely on its "logic", i.e. on the formulae of which it consists. So logic programming is a declarative programming paradigm. The programmer can construct programs and reason about them at a higher level of logic, abstracting from their computations. Whole reasoning about program correctness (more precisely, correctness and completeness) can be done at this level. This is an important advantage of the paradigm. One needs to resort to the operational semantics (the "control") only to deal with termination and efficiency. Modifying the control does not change (the set of) the answers of the program. What is changed is the way they are computed; the logic is separated from the control [Kow79]. The programming language Prolog was introduced as an implementation of logic programming. It is still the major logic programming language. It makes declarative programming possible, at least to a substantial extent.

On the other hand, a Prolog program can be looked at from a point of view of its operational semantics. A program is a precise description of computations; one can program in Prolog imperatively. This is often necessary, for instance when a program has to interact with its environment. Dealing with the operational semantics is often difficult, due to among others the non-straightforward nature of backtracking, coroutining and tabulation.

An important activity in programming is program debugging. And when it comes to debugging of Prolog programs, all the declarativeness is lost. Declarative methods, known as declarative diagnosis (DD) or algorithmic debugging, exist but are neglected. Basically, the only tool available for a programmer is the Prolog debugger, which is based solely on the operational semantics. So the tool is incompatible with declarative programming (cf. e.g. [Llo87, DN94, Dra19]). To use the debugger the programmer has to go into details of the operational semantics, abandoning the declarative thinking. This makes the debugging difficult, and encourages programmers to resign from the declarative view of programs. The difficulties are even more serious when more sophisticated control is involved, like coroutining or tabulation.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, L. Serafini (Eds.): ICLP2023
EPTCS ??, 2023, pp. 22–29, doi:10.4204/EPTCS.??.??

As the Prolog debugger is a rather powerful tool, one may expect that a "declarative programmer" can obtain from it the information she needs. An example of such information is which premises have been used to derive a given answer, displayed at an `exit` port. Obtaining such information is possible, but surprisingly tedious [Dra19].

In the next section we briefly introduce declarative diagnosis (DD), present a unifying view of incorrectness and incompleteness diagnosis, and suggest a kind of suitable DD tools. Then (Section 3) we identify the intended model problem, which has possibly been the main obstacle for acceptance of DD, and show how to overcome it. In Section 4 we discuss applying DD to Prolog with tabulation and delays.

Preliminaries. The presentation is rather informal, in most cases references are given to a more precise treatment. In some places the terminology of logic is used [Apt97], instead of that of Prolog manuals. So by an *atom* we mean an atomic formula, not a Prolog constant. We consider SLD-resolution dealing with *queries* (conjunctions of atoms), instead of goals (negated queries). From a programmer's point of view, the selected atom in a query in an SLD-derivation is a *procedure call*. By a (*computed*) *answer* of a program we mean the result of applying a (computed) answer substitution to the initial query. The set of clauses (in a program) beginning with a predicate symbol p will often be called *procedure* p. The Herbrand base will be denoted by $\mathscr{HB}$, and the least Herbrand model of a program P by $\mathscr{M}_P$.

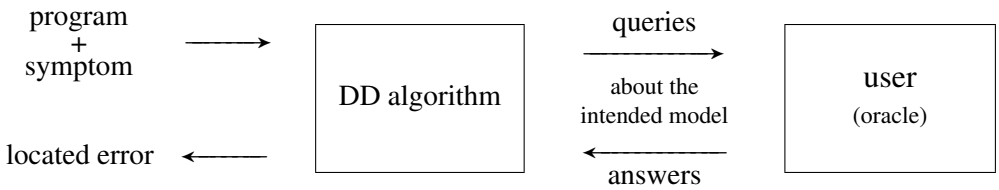
In imperative and functional programming, program correctness consists of program termination and partial correctness. The latter means that the program produces required results provided it terminates. In logic programming, due to its nondeterministic nature, partial correctness splits into *correctness* and *completeness* [Hog84, DM05, Dra16]. Correctness means that each answer of the program is compatible with the specification. Completeness means that all the answers required by the specification are produced by the program. Obviously, termination is a property of the operational semantics, in contrast to correctness and completeness. Debugging consists of program *diagnosis*, i.e. locating errors in programs, and program correction. In this paper we are interested in diagnosing incorrectness and incompleteness for logic / Prolog programs.

2 Declarative diagnosis

The idea of declarative diagnosis was introduced by Shapiro in his seminal thesis [Sha83], under the name algorithmic debugging. Shapiro gave algorithms for diagnosing incorrectness and incompleteness (and also non-termination). His ideas were followed and developed by further work, dealing mainly with logic programming, but also with other paradigms (see e.g. [Per86, DNTM89, Nai92, CRS17]).

Specifications, symptoms, errors. Diagnosis starts with a *symptom* of program incorrectness or incompleteness. The purpose is to locate in a program an *error*, i.e. a fragment responsible for the symptom. In incorrectness diagnosis a symptom is an incorrect answer of the program. In diagnosing incompleteness it is, generally speaking, a missing answer. Here by an *incompleteness symptom* we mean an atomic query for which the program terminates, but it does not produce some required answers.

Diagnosis must be based on a specification. In DD the specification is the *intended model* of the program. This is the least Herbrand model of the target program. A DD algorithm is informed about the intended model by an *oracle*, which answers queries about the model. Usually the oracle is the user.



In incorrectness diagnosis, an *incorrectness error* is an incorrect clause. In such clause the body atoms are true, and the head is not true in the intended specification. In logical terms $M \not\models C$, where M is the intended model and C the clause.

By an *incompleteness error* we will mean an atomic query A whose answer (required by the specification) cannot be produced by any clause of the program (out of atomic queries required by the specification). Logically, we have an instance $A\theta \in M$ such that the program does not contain a clause with an instance $A\theta \leftarrow \vec{B}$ such that $M \models \vec{B}$, where M is the intended model. So an incompleteness error points at a procedure responsible for the error (and presents an atom the procedure should additionally produce).

It can be shown that a more precise notion of an error is impossible. For instance, an incorrect clause can be corrected in various ways, leaving various its fragments unchanged. So we cannot consistently name such a fragments as correct, or incorrect.

DD algorithms, a unifying view. For incompleteness, we prefer Pereira style diagnosing [Nai92], as it is more convenient than that of [Sha83]. In such context, both incorrectness and incompleteness diagnosis can be treated as instances of a single algorithm. (See [Nai97, CRS17] for a similar observation.)

Let us first describe the oracle queries. A query in incorrectness diagnosis is an atom A (which was obtained as an answer in the computation). In incompleteness diagnosis it is an atom A (a procedure call from the computation) together with the computed answers $A\theta_1, \ldots, A\theta_n$ ($n \geq 0$) obtained for A. The oracle has to answer whether the query is a symptom (of correctness or completeness, respectively).

The algorithm inspects an erroneous computation of the program, and builds builds a *DD search tree* of oracle queries. The root is the symptom we begin with. The tree contains a *target*, i.e. a node which is a symptom and its children are not. The algorithm employs the oracle to search the tree for a target. Now we give further details of the tree and explain how a target determines an error.

In incorrectness diagnosis the tree is a proof tree (the root is a wrong answer, each node together with its children is an instance of a program clause). Thus a target with its children is an incorrectness error.

For incompleteness diagnosis, each node $\mathscr{A}$ (which is an atom A with its computed answers) has children that are the top-level queries (together with the answers) used in the computation for A. (By top-level we mean one that is an instance of a body atom of a clause used to resolve A.) If $\mathscr{A}$ is a target and its children are not, then $\mathscr{A}$ contains an incompleteness error (which is the procedure call from $\mathscr{A}$). Informally, as we found no incompleteness related to the body atoms of the applicable clauses, the clauses are responsible (for the initial symptom).

It is important that with a DD algorithm the user can locate the error without looking at the program. Also, all the diagnosis is performed solely in terms of the declarative semantics. So we do not need to understand the operational semantics.

Non-algorithmic declarative diagnosis. Practice shows some inconveniences of using DD algorithms, at least in their basic form (cf. e.g. [DNTM89]). An inconvenient and counterproductive feature is that the order of queries is determined by the algorithm. The effort (and time) to answer various queries may differ substantially,[1] so it is desirable to first deal with queries considered by the user to be easier. As a result, some difficult queries many eventually not be asked. Also, it should be possible to modify one's previous answer, as errors in answering are possible. Such modifications are also useful in making and withdrawing temporary assumptions ("what if this were correct").

Such inconveniences disappear when the search is performed by the user. It is sufficient that a DD tool extracts from the computation the DD search tree, and gives the user convenient means to explore the tree. The author's experience with such prototype tools shows that they are more convenient to use than

[1] So it is not realistic to judge the efficiency of DD algorithms by means of query complexity [Sha83, CRS17], which deals solely with the number of queries.

(prototype implementations of) DD algorithms. The main burden – with understanding and answering queries – is the same (only the answers are not typed in). Additional human effort, due to performing the search, is minor. Being relieved of the restraints of DD algorithms is a substantial convenience. The user may look at various queries and choose which to answer first. Some answers may be postponed, the user may inspect the tree freely. An already visited part of the tree may be re-inspected, to correct former decisions (if erroneous, or assumed temporarily).

What we propose is, in a sense, a declarative counterpart of the Prolog debugger. The debugger gives the user access to the operational semantics of a program; we show how to give access to the declarative semantics.

3 The intended model problem

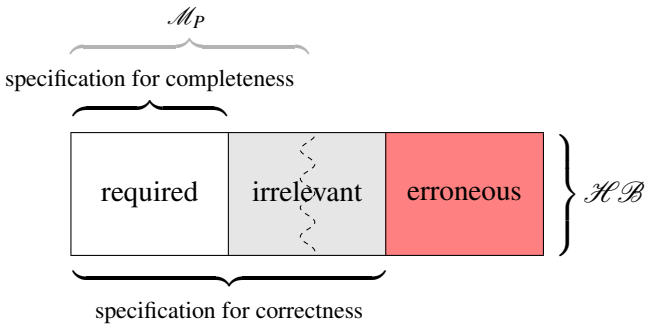
In this section, based on [Dra16, Section 7], we discuss what possibly is the main reason for lack of acceptance of DD in practice. Namely a fundamental notion of DD is that of the intended model. This means the the programmer has to know exactly the intended least Herbrand (2-valued) model of her program; in other words, to know exactly the relation to be defined by each predicate. This is usually not realistic.

As an example [Dra16] consider insertion sort and predicate insert/3, dealing with inserting an element into a sorted list (to obtain a sorted list as a result). The user does not (and should not) know how the predicate should behave on unsorted lists. Hence she is unable to answer an oracle query like "Is atom $B = \mathit{insert}(2,[3,1],[2,3,1])$ correct"? Let us call this phenomenon the *intended model problem*.

Both treating B as correct, or incorrect may be reasonable, and result in different debugged programs. For instance treating all such atoms as incorrect leads to a program in which insert/3 checks that its second argument is a list. Such program is inefficient. So choosing the intended model may then force certain design decisions, possibly undesirable ones. Often while developing a program it is impossible to know in advance its exact semantics (i.e. its intended model). Apparently the semantics of insert/3 (in a standard insertion sort program [SS94, Program 3.21]) is a result of making the program simple and efficient, and has not been decided in advance. See [Dra18] for a larger example.

In the author's opinion the intended model problem makes DD inapplicable in many, if not in most of, practical cases.

Approximate specifications. Such examples show a common feature of specifications in logic programming (both formal, and informal ones which programmers have in their minds): often such specifications have a 3-valued flavour. Some atoms are clearly true, some false, and the remaining ones are irrelevant. They may be accepted as answers, but should not be required to be answers of the program.



So instead of specifying a single intended model, we should provide two specifications, S^0 for completeness and S for correctness (where $S^0 \subseteq S$). The irrelevant atoms, like B in our example are in $S \setminus S^0$. Let us call such pair S^0, S of specifications an *approximate specification*.

A program P is correct w.r.t. an approximate specification S^0, S if $S \models A$ for each answer A of P (i.e. $P \models A$ implies $S \models A$); P is complete w.r.t. it if $S^0 \models A$ implies that A is an answer of P ($S^0 \models A$ implies $P \models A$).

A reasonable approximate specification for insert/3 of insertion sort is:

$$\text{for completeness:}\quad S^0_{insert} = \left\{ insert(n,l_1,l_2) \in \mathscr{HB} \;\middle|\; \begin{array}{l} l_1, l_2 \text{ are sorted lists of integers,} \\ elms(l_2) = \{n\} \cup elms(l_1) \end{array} \right\},$$

where $elms(l)$ – the multiset of elements of l,

$$\text{for correctness:}\quad S_{insert} = \left\{ insert(n,l_1,l_2) \in \mathscr{HB} \;\middle|\; \begin{array}{l} \text{if } n \text{ is an integer and} \\ l_1 \text{ is a sorted list of integers,} \\ \text{then } insert(n,l_1,l_2) \in S^0_{insert} \end{array} \right\}.$$

Note that neither S^0_{insert}, nor S_{insert} corresponds to the semantics of the actual corrected insertion sort program (i.e. neither of them is the set of atoms of the form $insert(\vec{t})$ from its least Herbrand model).

Obviously, diagnosis of incorrectness and that of incompleteness should be done employing the respective specification. With such approach to specifications, the standard DD methods (based on intended, 2-valued interpretations) are sufficient.

The intended model problem has been noticed, among others, by [Per86, DNTM89, Nai00, Fer93]. Pereira [Per86] introduces inadmissible atoms, which should not appear in the computation. In this way the debugging is no more declarative as it refers also to the intended operational semantics of the program.

Naish [Nai00] introduces a rather sophisticated 3-valued approach to DD, and two kinds of errors (e-bug and i-bug). The atoms from $S_{insert} \setminus S^0_{insert}$ above have in the 3-valued intended model the third logical value **i**. (Those from S^0_{insert} are **t**, and those from $\mathscr{HB} \setminus S_{insert}$ are **f**.) It seems strange, that both incorrectness and incompleteness diagnosis treat in the same way the **i** atoms; the discussion above shows that this should not be the case.

Any such complications are not necessary when we have separate specifications for correctness and completeness. Introducing such specifications solves the intended model problem. In the author's opinion this is a main step to make DD practical.

Negation and partial specifications. When we deal with logic programs with negation, a specification should also describe the negative results of programs; roughly, which ground atoms fail. In other words, which negated literals are program consequences under the chosen semantics.

As in the discussion above, a specification often has to distinguish between those atoms that have to fail (specifying completeness) from those that may fail (specifying correctness). To specify both positive and negative program answers, it is natural to use an approximate specification S^0, S as introduced above, and interpret it as follows [DM05]. The atoms that are not allowed to succeed have to fail. The atoms that are required to succeed cannot fail. Thus the set of atoms that have to fail is $\mathscr{HB} \setminus S$ (the "erroneous" in the diagram). Similarly, the set of atoms that may fail is $\mathscr{HB} \setminus S^0$ (the "irrelevant" and "erroneous" in the diagram). In a correct program, if $\neg A$ is its answer then $S^0 \models \neg A$. In a complete program, if $S \models \neg A$. then $\neg A$ is an answer. So the description of correctness for positive answers describes completeness of negative ones, and vice versa.

We do not discuss here DD of programs with negation [Llo87, DNTM89, NT90]. We only mention that finding $\neg A$ as a target in incorrectness diagnosis leads to incompleteness diagnosis for A (and vice versa).

4 Extracting the DD search tree.

This section considers implementing declarative diagnosis for Prolog with tabulation and coroutining. It turns out that only incompleteness diagnosis for coroutining poses problems. We discus two ways of

dealing with them.

DD requires extracting the necessary information (i.e. the DD search tree) from a computation of the program. For basic Prolog (without tabulation and coroutining) we know how to do this, see e.g. [Sha83, DNTM89, Nai92]; we skip further details. For incorrectness diagnosis this means extracting the proof tree. (Most of) the methods to do this for basic Prolog seem easy to generalize for coroutining and tabulation. Such generalization is far from obvious for incompleteness diagnosis.

For incompleteness diagnosis (in Pereira-style) one needs to collect from the computation the procedure calls, and for each of them the obtained computed answers. For basic Prolog, a simplest way to do this seems top-level meta-interpretation for the considered procedure call (cf. [DNTM89], where however the DD search tree is not made explicit). So for a call $p(\vec{t})$, all the clauses $p(\vec{s}) \leftarrow B_1, \ldots, B_n$ are meta-interpreted, however each encountered procedure call $B_i\theta_j$ is executed by the Prolog system. All such calls (together with their answers) are children in the DD search tree of the node $p(\vec{t})$ (with its answers).

Other ways of obtaining the search tree for incompleteness DD may be suitable. In particular, obtaining the whole tree from a single computation may be preferable (as this mimics the actual side effects of a not purely declarative program).

Transferring such ideas to Prolog with tabulation should not create problems. The task is however difficult for programs with coroutining. This is because the answers for a given procedure call may actually not be computed (during the computation for the considered initial query). The query instance returned by Prolog as an answer (e.g. that appearing at the Exit port of Prolog debugger) will be called here a *pseudo-answer*. See [Dra23] for a formal description of coroutining.

Let us explain. During a computation for a sub-query A it may happen that (i) unblocking of a formerly delayed sub-query results in a pseudo-answer being an instance of an actual computed answer, or (ii) delaying of a sub-query results in a pseudo-answer being more general than an actual one (or an actual answer may not exist). When (i) and (ii) coincide, nothing can be concluded.

For sound incompleteness diagnosis, a DD query should contain actual computed answers for a given sub-query A, but what we can obtain from the computation are pseudo-answers. This difficulty was discussed in [Nai92], without a general solution. Executing A without coroutining is of no help in general, as it may result in non-termination. It is useful to do this under a suitable time limit; if A terminates, the problem is solved. For a general case consider the following. From a computation trace one can find if cases (i) or (ii) have been involved in producing a given pseudo answer for A. If not, the pseudo-answer is an answer. If (i) has not occurred then the pseudo-answer is more general than the actual answer (or the latter does not exist). If (ii) has not occurred, the pseudo-answer is an instance of an actual one. A pseudo-answer together with such information can be included in a DD query, instead of the unknown actual answer; the information makes it possible in some cases to answer the query (formally, to state whether A with the actual answers is an incompleteness symptom).[2]

This approach may fail to find that, despite of the involved coroutining, a pseudo-answer is an answer. We propose another method of checking this, and to find the actual answer in some cases.

From the computation producing a pseudo-answer $A\theta$ for A (w.r.t. a program P) a proof tree may be collected as in incorrectness diagnosis. Due to delays, some subtrees of an actual proof tree may be missing, so the obtained tree will be called a *pseudo-proof tree*. Such tree T, with root $A\theta$, is inspected to find if it is (a) a proof tree or (b) not. For (a), $A\theta$ is an answer of P, but it may not be a computed

[2] If case (i) has not occurred for any pseudo-answer for A, and some required answer is not an instance of any displayed (pseudo-)answer, then A (with its possibly unknown answers) is surely a symptom. If case (ii) has not occurred for any pseudo-answer for A, and each answer required by the specification is an instance of some displayed one, then A (with answers) is surely not a symptom.

answer for A (but its instance, case (i)). Out of the tree we can obtain the clauses of P used to compute $A\theta$. Applying the clauses, we can obtain a computed answer $A\sigma$ for A, so that T is an instance of the proof tree T' for $A\sigma$ (and $A\sigma$ is more general than $A\theta$). Now $A\sigma$ can be used in a DD query,

The same procedure can be applied to T in case (b), it results in a pseudo-proof tree T' with root $A\sigma$, more general than $A\theta$. The tree corresponds to executing A alone (in a context without any pending delayed sub-queries). Formally, T' corresponds to a prefix D of an SLD-derivation for A; the last query Q of D consists of the delayed sub-queries. So any computed answer for A obtained by completing the derivation is an instance of $A\sigma$. Thus $A\sigma$ can be used instead of the unknown answer in a DD query, and be treated like a pseudo-answer for which (i) has not occurred (cf. footnote 2).

As previously, it makes sense to execute Q under a time limit. If it terminates, then all the derivations with D as a prefix have been found. They provide computed answers for A, to be used in a DD query (instead of the pseudo-answer $A\theta$).

To summarize, generalizing DD to take care of tabulation and coroutining seems rather obvious, except for diagnosing incompleteness for coroutining. In this case some DD queries cannot be extracted from the computation of the program. We discussed how to deal with this problem. For most of cases we showed a way to obtain a sound answer for such an undetermined DD query. There is one exception (in case (b) above, when the delayed Q does not terminate under a given time limit, one cannot determine that the DD query is not a symptom). We do not pursue here another possible approach, based on detailed monitoring of variable bindings during the computation.

5 Conclusion

Comments. Due to lack of space, we do not discuss various issues related to pragmatics of DD (declarative diagnosis), and to Prolog features apart basic ones. The importance of a user friendly interface is obvious (it should be possible to view big terms effectively, to conveniently inspect DD search trees, etc). Experience from actual debugging is needed to evaluate diagnosis approaches, and to better understand which features a useful diagnosis tool should have. The author is interested in buggy programs that could be used in such experiments, preferably with challenging or otherwise interesting bugs.

DD diagnosis turns out to be applicable also to programs with non-declarative fragments. Of course this concerns only issues related to the program answers (and not e.g. the sequences of input/output actions). For instance, prototype "non-algorithmic" DD tools (cf. Section 2) for incorrectness and incompleteness have been used to debug themselves. Locating errors with these tools seems substantially simpler than with the Prolog debugger.

Debugging for ASP is a separate issue, not dealt with here (as the role of answers in ASP is different from that in standard logic programming). An interesting challenge is to generalize DD so that more symptoms (of possibly a single error) can be used to locate the error more efficiently.

Summary. This note deals with DD of logic programs. We treat incorrectness diagnosing and incompleteness diagnosing (Pereira style) as instances of the same algorithm. We point out the intended model problem as possibly the main reason for lack of acceptance of DD in practice. We also consider DD for programs with tabulation and coroutining, and propose how to cope with the difficulties that arise. We advocate that a tool which allows the user to inspect the DD search tree is more suitable to perform diagnosis than an implementation of a whole DD algorithm.

References

[Apt97] K. R. Apt (1997): *From Logic Programming to Prolog*. International Series in Computer Science, Prentice-Hall.

[CRS17] R. Caballero, A. Riesco & J. Silva (2017): *A Survey of Algorithmic Debugging*. *ACM Comput. Surv.* 50(4), pp. 60:1–60:35, doi:10.1145/3106740.

[DM05] W. Drabent & M. Miłkowska (2005): *Proving correctness and completeness of normal programs – a declarative approach*. *TPLP* 5(6), pp. 669–711, doi:10.1017/S147106840500253X.

[DN94] Mireille Ducassé & Jacques Noyé (1994): *Logic Programming Environments: Dynamic Program Analysis and Debugging*. *J. Log. Program.* 19/20, pp. 351–384, doi:10.1016/0743-1066(94)90030-2.

[DNTM89] W. Drabent, S. Nadjm-Tehrani & J. Małuszyński (1989): *Algorithmic Debugging with Assertions*. In H. Abramson & M. H. Rogers, editors: *Meta-Programming in Logic Programming*, The MIT Press, pp. 501–522.

[Dra16] W. Drabent (2016): *Correctness and Completeness of Logic Programs*. *ACM Trans. Comput. Log.* 17(3), pp. 18:1–18:32, doi:10.1145/2898434.

[Dra18] W. Drabent (2018): *Logic + control: On program construction and verification*. *Theory and Practice of Logic Programming* 18(1), pp. 1–29, doi:10.1017/S1471068417000047.

[Dra19] W. Drabent (2019): *The Prolog Debugger and Declarative Programming*. In M. Gabbrielli, editor: *LOPSTR 2019, Revised Selected Papers*, *Lecture Notes in Computer Science* 12042, Springer, pp. 193–208, doi:10.1007/978-3-030-45260-5_12.

[Dra23] W. Drabent (2023): *Implementing backjumping by means of exception handling*. *arXiv*. Available at `https://doi.org/10.48550/arXiv.2305.16137`.

[Fer93] G. Ferrand (1993): *The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs*. In P. Fritszon, editor: *AADEBUG'93, Proceedings*, *LNCS* 749, Springer, pp. 40–57, doi:10.1007/BFb0019399.

[Hog84] C. J. Hogger (1984): *Introduction to Logic Programming*. Academic Press, London.

[Kow79] Robert A. Kowalski (1979): *Algorithm = Logic + Control*. *Commun. ACM* 22(7), pp. 424–436. Available at `http://doi.acm.org/10.1145/359131.359136`.

[Llo87] J. W. Lloyd (1987): *Declarative error diagnosis*. *New Gener Comput* 5, pp. 133–154. Available at `https://doi.org/10.1007/BF03037396`.

[Nai92] L. Naish (1992): *Declarative Diagnosis of Missing Answers*. *New Generation Comput.* 10(3), pp. 255–286. Available at `http://dx.doi.org/10.1007/BF03037939`.

[Nai97] L. Naish (1997): *A Declarative Debugging Scheme*. *J. Funct. Log. Program.* 1997(3). Available at `http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1997/A97-03/A97-03.html`.

[Nai00] L. Naish (2000): *A Three-Valued Declarative Debugging Scheme*. In: *23rd Australasian Computer Science Conference (ACSC 2000)*, IEEE Computer Society, pp. 166–173. Available at `http://doi.ieeecomputersociety.org/10.1109/ACSC.2000.824398`.

[NT90] S. Nadjm-Tehrani (1990): *Debugging Prolog Programs Declaratively*. In: *META90: 2nd Workshop on Meta-Programming in Logic Programming*, Leuven.

[Per86] L. M. Pereira (1986): *Rational Debugging in Logic Programming*. In E. Y. Shapiro, editor: *ICLP*, *Lecture Notes in Computer Science* 225, Springer, pp. 203–210. Available at `http://dx.doi.org/10.1007/3-540-16492-8_76`. Extended version at `https://userweb.fct.unl.pt/~lmp/`.

[Sha83] E. Shapiro (1983): *Algorithmic Program Debugging*. The MIT Press.

[SS94] L. Sterling & E. Shapiro (1994): *The Art of Prolog*, 2 edition. The MIT Press. Available at `https://mitpress.mit.edu/books/art-prolog-second-edition`.