Feasibility of
Declarative Diagnosis (DD)
a.k.a. Algorithmic Debugging

Włodek Drabent

Insitute of Computer Science, Polish Academy of Sciences

Prolog makes declarative programming possible

(at least to a substantial extend)

A Prolog programmer can focus on “logic”

(“control” of secondary importance)

☹ But: Declarativeness **broken**

when it comes to debugging

The Prolog debugger – purely operational,

forces us to abandon declarative thinking

Declarative methods exist, but are neglected.

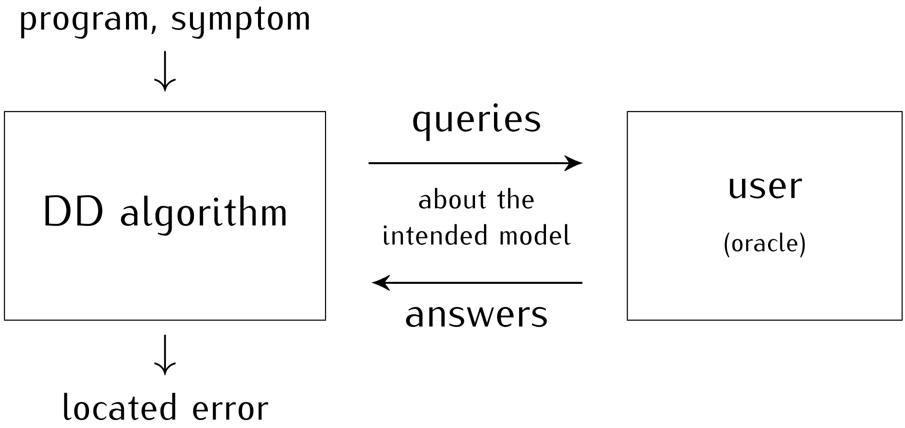
We show **why**

We show **how** to overcome the problems.

Debugging – diagnosis (locating errors)
+ error correction

Declarative diagnosis (DD) / algorithmic debugging

[Shapiro83]



Symptom – a wrong result of the program
Error – the/a reason for the symptom

Intended model M – the specification
(the least Herbrand model of the intended program)

Incorrectness

symptom – wrong atomic answer A ($M \not\models A$)
error – incorrect clause C ($M \not\models C$), instance of a program clause
DD query – "is A correct?" (does $M \models A$? i.e. is A a non-symptom?)

Incompleteness

symptom – atomic query A which terminates with missing answers
($\exists\theta M \models A\theta$, but $A\theta$ is not an instance of any computed answer for A)

:

error – A for which some required answer $A\theta$ cannot be produced by any clause of the program out of M
(no program clause instance $A\theta \leftarrow \vec{B}$ where$M \models \vec{B}$)

Note: This is what (Pereira style) diagnosers find, although one may like to consider $A\theta$ as an error

DD query – "is A with answers $A\theta_1, \ldots, A\theta_n$ a non-symptom?"

DD algorithms

1. Extract from the computation a DD search tree
(of oracle queries, the root is the initial symptom).
2. Search it for a target
(a symptom with all children not being symptoms).

Incorrectness diagnosis

DD search tree = proof tree (each node with its children – an instance of a program clause)

A target with its children - incorrectness error.

Incompleteness diagnosis (Pereira style)

DD search tree:

node – a procedure call + its computed answers from the computation

children of a node $\mathcal{A}$ – the top level procedure calls used to evaluate $\mathcal{A}$ with their answers

Incompleteness error – the procedure call in the target

Intended model problem – main obstacle to DD

☞ **The intended model often not known!** ☜

Ex. insert/3 of insertion sort

DD query: Is B correct?

$$B = insert(2, [3, 1], [2, 3, 1]) \quad - \quad ??$$
$$B_1 = insert(2, [1, 3], [1, 2, 3]) \quad - \quad \text{YES}$$
$$B_2 = insert(2, [1, 3], [1, 3, 2]) \quad - \quad \text{NO}$$

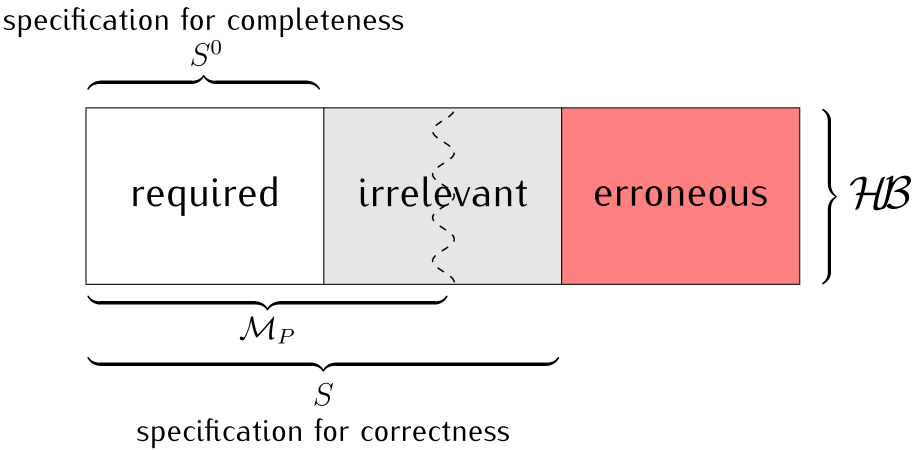
The user does not (and should not) know how insert/3 should behave on unsorted lists

(Various versions possible, lead to different debugged programs)

The intended model problem makes DD inapplicable in many practical cases.

Intended model problem – solution

Realize that our (formal/informal) specifications are approximate



Approximate specification – S^0, S,

a pair of Herbrand interpretations

P correct w.r.t. S^0, S if $\mathcal{M}_P \subseteq S$

P complete w.r.t. S^0, S if $S^0 \subseteq \mathcal{M}_P$

Ex. Approximate specification S^0_{insert}, S_{insert} for insert/3

For completeness $S^0_{insert} =$

$$\left\{ insert(n, l_1, l_2) \in \mathcal{HB} \;\middle|\; \begin{array}{l} l_1, l_2 \text{ are sorted lists of integers,} \\ elms(l_2) = \{n\} \cup elms(l_1) \end{array} \right\},$$

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

For correctness $S_{insert} =$

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

This solves the intended model problem:

Perform α diagnosis using the specification for α as the intended model
where $\alpha \in \{\text{incorrectness}, \text{incompleteness}\}$

Standard DD algorithms sufficient.
No additional sophistication needed
(like inadmissible atoms [Pereira'86],
3-valued DD with 2 kinds of bugs [Naish'00], ...)

Inconveniences of DD algorithms

The order of DD queries imposed by the algorithm

The user cannot

- postpone difficult queries
- correct her buggy answers
- make assumptions (what if this were correct?)

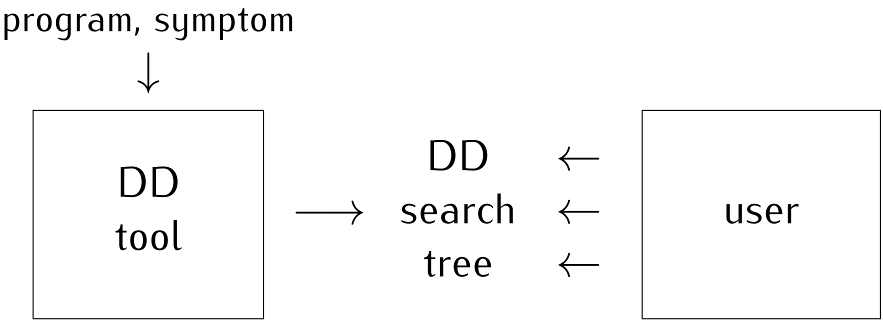
Solution: non-algorithmic DD

Searching of DD search trees

– simple and can be done by the user

What is really needed:

a debugger working in terms of declarative semantics



Experience with prototypes
(for incorrectness, for incompleteness) :

More convenient than DD algorithms
than Prolog debugger

Prolog debugger – a powerful tool

Surprisingly, not useful for obtaining the nodes of a DD search tree [Dra19] (especially for incorrectness diagnosis)

Extracting DD search trees

out of a Prolog execution

Rather obvious, except for

Coroutining + incompleteness diagnosis

We need

all computed answers for a given procedure call A

They actually may not be produced

due to unfrozen/frozen calls between A and its success

Let *pseudo-answer* – answer in presence of coroutining

(i) A call unfrozen –

a pseudo-answer an instance of a computed answer

(ii) A call delayed –

a pseudo-answer more general than a computed answer

(or the latter does not exist)

(i) + (ii) – a combination of the above

(Note: E an instance of E)

Solutions (partial)

1. Re-executing A alone

   May not terminate; apply a time limit

2. If no (i) occurred,

   A with its pseudo-answers is a symptom

   $\Rightarrow$ A with $\underset{\text{unknown}}{\text{computed answers}}$ is a symptom.

3. If no (ii) occurred,

   A with its pseudo-answers is not a symptom

   $\Rightarrow$ A with $\underset{\text{unknown}}{\text{computed answers}}$ is not a symptom.

4. Construct a computed answer

   out of the clauses used to obtain a pseudo-answer

OK if no (ii) occurred.
In case (ii) frozen calls are to be executed
(may not terminate, apply a time limit)
Otherwise we obtain $A\theta$ more general than a (possible) computed answer; solution 2 applies

5. Something else ?

Could we find a DD method, which uses DD queries $(A, A\theta_1, \ldots, A\theta_n)$
where A is an instance of an actual procedure call A_0
(and $A\theta_1, \ldots, A\theta_n$ are the computed answers for A, and
the queries can somehow be obtained from the actual computation) ?

Comments

* DD applicable to "real" Prolog programs
  (i.e. to their "declarative aspects")
  E.g. my prototype tools (≈ 400+200 lines without comments)
  used to debug themselves

* Experiments needed to evaluate diagnosis approaches.
  Problem: How to make them realistic?
  What makes useful bug examples?
  The author solicits interesting buggy programs. ☜

* DD for ASP – separate issue,
  as the role of answers is substantially different

  Known how to do:

  DD for NAFF (negation as finite failure)

  approximate specifications for NAFF

* Various pragmatic issues not discussed here.
  Effective user interfaces (…presenting big trees, big terms)
  choice of implementation approaches, dealing with built-ins,
  debugging guide, …

* Query complexity of DD algorithms – impractical

Summary, main points

DD of logic programs

Intended model problem – possibly the main reason for non-acceptance of DD

A tool to inspect DD search trees
more suitable than a DD algorithm

Difficult case
– incompleteness diagnosis + coroutining
A partial solution given

We often teach a programming language instead of teaching programming.
Even if we teach programming, we often do not teach debugging.
[Mireille Ducassé]