On logic programming and locating errors in programs

Włodzimierz Drabent

Institute of Computer Science, Polish Academy of Sciences (IPI PAN);
IDA, Linköpings universitet, Sweden

SaS seminar 2019-11-08
Version 1.0, compiled November 15, 2019

Outline

- Introduction to Logic Programming (LP)
- On proving program correctness (and completeness),
  i.e. how to reason about our programs
- Approximate specifications
- Declarative Diagnosis (DD)
  Why abandoned; a cure
  Inadequacy of Prolog debuggers
- Summary

Outline

Logic Programming (LP) is declarative

We can do declarative programming in Prolog

Debugging should be declarative too

Methods exist:

Declarative Diagnosis (DD), a.k.a. algorithmic debugging
[Shapiro'83,Pereira'86,Naish,...]

Tools do not $\ddot{\frown}$

We discuss the (possibly) main reason for non-acceptance of DD

Declarative programming

WHAT to compute

Program – a description of the problem

not a description of computer actions

Logic Programming

Program – a set of axioms
Results – its logical consequences
Computation – proof construction

Main programming language – Prolog

Logic Programming (LP). The core part

Program – a set of axioms (of the form $A_0 \leftarrow A_1, \ldots, A_n$
A_i – atoms (atomic formulae)).

Computation – search for logical consequences of the program.

Query Q (of the form $A_1, \ldots, A_n$).

Answers $Q\theta$ such that $P \models Q\theta$

(P – the program, θ – substitution).

Any answer Q' computed for P is a logical consequence of P, $P \models Q'$.

And conversely
(if $P \models Q\theta$ then $Q\theta$ is an instance of a computed answer for Q).

Note: untyped logic

LP, example, puzzle

Build a sequence out of three 1's, three 2's, . . . , three 9's, so that between each consecutive occurrences of i there are exactly i elements.

[1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7]

[1,8,1,9,1,5,2,6,7,2,8,5,2,9,6,4,7,5,3,8,4,6,3,9,7,4,3]
[1,9,1,6,1,8,2,5,7,2,6,9,2,5,8,4,7,6,3,5,4,9,3,8,7,4,3]
[3,4,7,8,3,9,4,5,3,6,7,4,8,5,2,9,6,2,7,5,2,8,1,6,1,9,1]
[3,4,7,9,3,6,4,8,3,5,7,4,6,9,2,5,8,2,7,6,2,5,1,9,1,8,1]
[7,5,3,8,6,9,3,5,7,4,3,6,8,5,4,9,7,2,6,4,2,8,1,2,1,9,1]

Notation

Variables in programs – begin with upper case

_ – anonymous variable (each occurrence of _ – a distinct variable)

$[a_1, \ldots, a_n]$ – list, its elements $a_1, \ldots, a_n$ $(n \geq 0)$

$[\,]$ – empty list

$[h|t]$ – the list with head h and tail t

$[h_1, h_2|t]$ – the list with head h_1 and tail $[h_2|t]$, i.e. $[h_1|[h_2|t]]$

Notation

Variables in programs – begin with upper case

_ – anonymous variable (each occurrence of _ – a distinct variable)

$[a_1, \ldots, a_n]$ – list, its elements $a_1, \ldots, a_n$ $(n \geq 0)$

$[\,]$ – empty list

$[h|t]$ – the list with head h and tail t

$[h_1, h_2|t]$ – the list with head h_1 and tail $[h_2|t]$, i.e. $[h_1|[h_2|t]]$

LP, example, puzzle

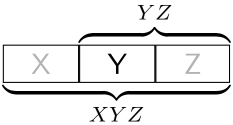
$$
\begin{array}{l}
solution(\,S\,) \leftarrow \\
\quad sequence27(\,S\,), \\
\quad sublist(\,[1, \_, 1, \_, 1], S\,), \\
\quad sublist(\,[2, \_, \_, 2, \_, \_, 2], S\,), \\
\quad sublist(\,[3, \_, \_, \_, 3, \_, \_, \_, 3], S\,), \\
\quad sublist(\,[4, \_, \_, \_, \_, 4, \_, \_, \_, \_, 4], S\,), \\
\quad sublist(\,[5, \_, \_, \_, \_, \_, 5, \_, \_, \_, \_, \_, 5], S\,), \\
\quad sublist(\,[6, \_, \_, \_, \_, \_, \_, 6, \_, \_, \_, \_, \_, \_, 6], S\,), \\
\quad sublist(\,[7, \_, \_, \_, \_, \_, \_, \_, 7, \_, \_, \_, \_, \_, \_, \_, 7], S\,), \\
\quad sublist(\,[8, \_, \_, \_, \_, \_, \_, \_, \_, 8, \_, \_, \_, \_, \_, \_, \_, \_, 8], S\,), \\
\quad sublist(\,[9, \_, \_, \_, \_, \_, \_, \_, \_, \_, 9, \_, \_, \_, \_, \_, \_, \_, \_, \_, 9], S\,).
\end{array}
$$

$$sublist(\,Y, XYZ\,) \leftarrow app(\,\_, YZ, XYZ\,), app(\,Y, \_, YZ\,).$$

$$sequence27([\_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_]).$$

$$
\begin{array}{l}
app(\,[\,], L, L\,). \\
app(\,[H|K], L, [H|M]\,) \leftarrow app(\,K, L, M\,).
\end{array}
$$

LP, example, puzzle



$$
\begin{array}{l}
solution(\,S\,) \leftarrow \\
\quad sequence27(\,S\,), \\
\quad sublist(\,[1,\_,1,\_,1], S\,), \\
\quad sublist(\,[2,\_,\_,2,\_,\_,2], S\,), \\
\quad sublist(\,[3,\_,\_,\_,3,\_,\_,\_,3], S\,), \\
\quad sublist(\,[4,\_,\_,\_,\_,4,\_,\_,\_,\_,4], S\,), \\
\quad sublist(\,[5,\_,\_,\_,\_,\_,5,\_,\_,\_,\_,\_,5], S\,), \\
\quad sublist(\,[6,\_,\_,\_,\_,\_,\_,6,\_,\_,\_,\_,\_,\_,6], S\,), \\
\quad sublist(\,[7,\_,\_,\_,\_,\_,\_,\_,7,\_,\_,\_,\_,\_,\_,\_,7], S\,), \\
\quad sublist(\,[8,\_,\_,\_,\_,\_,\_,\_,\_,8,\_,\_,\_,\_,\_,\_,\_,\_,8], S\,), \\
\quad sublist(\,[9,\_,\_,\_,\_,\_,\_,\_,\_,\_,9,\_,\_,\_,\_,\_,\_,\_,\_,\_,9], S\,).
\end{array}
$$

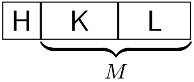
$$sublist(\,Y, XYZ\,) \leftarrow app(\,\_, YZ, XYZ\,), app(\,Y, \_, YZ\,).$$

$$sequence27([\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_]).$$

$$
\begin{array}{l}
app(\,[\,], L, L\,). \\
app(\,[H|K], L, [H|M]\,) \leftarrow app(\,K, L, M\,).
\end{array}
$$

LP, example, puzzle

$$
\begin{aligned}
&solution(\,S\,) \leftarrow \\
&\quad sequence27(\,S\,), \\
&\quad sublist(\,[1,\_,1,\_,1], S\,), \\
&\quad sublist(\,[2,\_,\_,2,\_,\_,2], S\,), \\
&\quad sublist(\,[3,\_,\_,\_,3,\_,\_,\_,3], S\,), \\
&\quad sublist(\,[4,\_,\_,\_,\_,4,\_,\_,\_,\_,4], S\,), \\
&\quad sublist(\,[5,\_,\_,\_,\_,\_,5,\_,\_,\_,\_,\_,5], S\,), \\
&\quad sublist(\,[6,\_,\_,\_,\_,\_,\_,6,\_,\_,\_,\_,\_,\_,6], S\,), \\
&\quad sublist(\,[7,\_,\_,\_,\_,\_,\_,\_,7,\_,\_,\_,\_,\_,\_,\_,7], S\,), \\
&\quad sublist(\,[8,\_,\_,\_,\_,\_,\_,\_,\_,8,\_,\_,\_,\_,\_,\_,\_,\_,8], S\,), \\
&\quad sublist(\,[9,\_,\_,\_,\_,\_,\_,\_,\_,\_,9,\_,\_,\_,\_,\_,\_,\_,\_,\_,9], S\,).
\end{aligned}
$$



$$sublist(\,Y, XYZ\,) \leftarrow app(\,\_, YZ, XYZ\,), app(\,Y, \_, YZ\,).$$

$$sequence27([\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_]).$$

$$
\begin{aligned}
&app(\,[\,], L, L\,). \\
&app(\,[H|K], L, [H|M]\,) \leftarrow app(\,K, L, M\,).
\end{aligned}
$$

LP. Two levels of reading a program

declarative – a set of axioms,
operational – a description of computations.

ALGORITHM = LOGIC + CONTROL

[Robert Kowalski, 1974]

Operational level (prog. lang. Prolog): control information
(the ordering within the program, some special constructs).

Important:

The two levels can be considered separately.
☞ Program correctness is a property of the declarative level.

LP. Two levels of reading a program

declarative – a set of axioms,
operational – a description of computations.

ALGORITHM = LOGIC + CONTROL

[Robert Kowalski, 1974]

Operational level (prog. lang. Prolog): control information
(the ordering within the program, some special constructs).

Important:

The two levels can be considered separately.

☞ Program correctness is a property of the declarative level.

We do not need to reason in terms of von Neumann machine.
J.Backus, *Can programming be liberated from the von Neumann style?* CACM, 1978

(One may also program operationally, neglecting the 1st level.)

LP. Two levels of reading a program

declarative – a set of axioms,
operational – a description of computations.

ALGORITHM = LOGIC + CONTROL

[Robert Kowalski, 1974]

Operational level (prog. lang. Prolog): control information
(the ordering within the program, some special constructs).

Important, often neglected:

The two levels can be considered separately.

☞ Program correctness is a property of the declarative level.

We do not need to reason in terms of von Neumann machine.
J.Backus, *Can programming be liberated from the von Neumann style?* CACM, 1978

(One may also program operationally, neglecting the 1st level.)

Program correctness

How to reason about program results ?

Imperative programming: partial correctness + termination

Program correctness

How to reason about program results ?

Imperative programming: partial correctness + termination

Correctness –
the program answers compatible with the specification

Completeness – all the required answers will be produced
(by the specification)

Program correctness

How to reason about program results ?

Imperative programming:	partial correctness	+	termination
	↙ ↘		
LP :	correctness completeness		

Correctness –
the program answers compatible with the specification

Completeness – all the required answers will be produced
(by the specification)

Program correctness

How to reason about program results ?

Imperative programming:	partial correctness	+	termination
	↙ ↘		
LP :	correctness completeness		
	full correctness (?)		

Correctness –

the program answers compatible with the specification

Completeness – all the required answers will be produced (by the specification)

Reasoning about program correctness

Specification – a set S of ground atoms (a Herbrand interpretation)

Correctness (of P) – each ground answer (of P) $\in S$: $\boxed{\mathbf{M}_P \subseteq S}$

Correctness proving method:

$$\boxed{S \models P \quad \Rightarrow \quad P \text{ correct w.r.t. } S.}$$

$\uparrow$

For each ground instance $H \leftarrow B_1, \ldots, B_n$ of a clause from P,
if $B_1, \ldots, B_n \in S$ then $H \in S$.

(Out of atoms $\in S$, the rules of P produce only atoms $\in S$)

The method has been already informally applied at this presentation.

Reasoning about program correctness

Specification – a set S of ground atoms (a Herbrand interpretation)

Correctness (of P) – each ground answer (of P) $\in S$: $\boxed{\mathbf{M}_P \subseteq S}$

Correctness proving method:

$$\boxed{S \models P \quad \Rightarrow \quad P \text{ correct w.r.t. } S.}$$

$\uparrow$

For each ground instance $H \leftarrow B_1, \ldots, B_n$ of a clause from P, if $B_1, \ldots, B_n \in S$ then $H \in S$.

(Out of atoms $\in S$, the rules of P produce only atoms $\in S$)

The method has been already informally applied at this presentation.

Reasoning about program completeness

Completeness (of P w.r.t. S) – each atom $\in S$ is an answer of P

$$S \subseteq \mathbf{M}_P$$

Completeness proving method

Main part of the sufficient condition – reverse of that for correctness

If $H \in S$ then
(∗) there exists a ground instance $H \leftarrow B_1, \ldots, B_n$ of a clause from P
s.that $B_1, \ldots, B_n \in S$.

(Each atom of S can be produced by a rule of P from atoms of S.)

The two methods much simpler than those for proving correctness of imperative programs!

Reasoning about program completeness

Completeness (of P w.r.t. S) – each atom $\in S$ is an answer of P

$$S \subseteq \mathbf{M}_P$$

Completeness proving method

Main part of the sufficient condition – reverse of that for correctness

If $H \in S$ then
$(*)$ there exists a ground instance $H \leftarrow B_1, \ldots, B_n$ of a clause from P
s.that $B_1, \ldots, B_n \in S$.

(Each atom of S can be produced by a rule of P from atoms of S.)

The two methods much simpler than those for proving correctness of imperative programs!

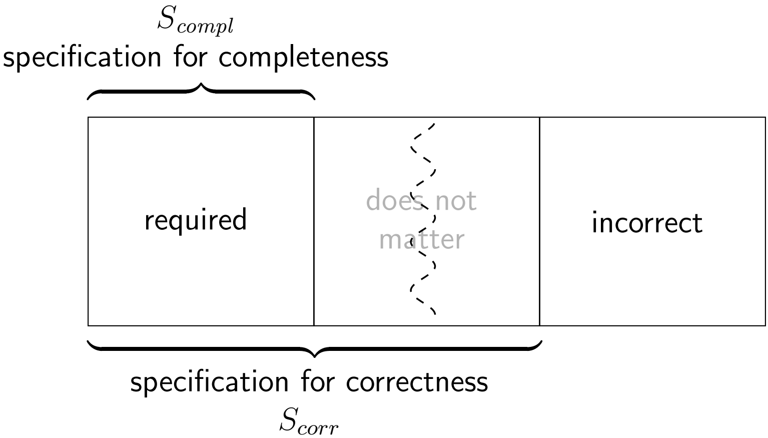
Important feature

Exact specification – often not known. E.g.

- `member`(e, t) for a non-list t,
- `append`(l, t, t') for non-lists t, t',
- `insert`(e, l, y) in insertion sort, for unsorted l,
- a predicate may have distinct semantics in distinct versions of a program under development!

(see Howe&King SAT solver in [D...,TPLP2018])

Approximate specifications



Approximate specifications, example

Ex.: specification for `member`/2:

$S_{corr} = S_{compl} \cup \{\mathtt{member}(e,t) \mid t \text{ not a list}\}$,

S_{compl} – the list membership relation, i.e.

$$S_{compl} = \{\mathtt{member}(t_i, [t_1, \ldots, t_n]) \mid 1 \leq i \leq n\}.$$

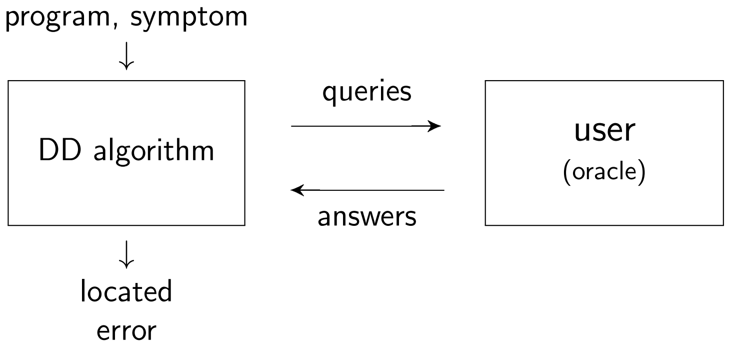
Declarative diagnosis (DD) a.k.a. algorithmic debugging

Methods of locating errors in programs,
based solely on the declarative semantics.

[Shapiro'83,Pereira'86,Naish,...]
[S.Nadjm-Tehrani,W.Drabent,J.Małuszyński,
H.Nilsson,N.Shahmehri,M.Kamkar,P.Fritzson,
R.Westman,P.Bunus,M.Sjölund]

The methods exist, but are abandoned.

DD (Declarative Diagnosis)



Queries – about the intended declarative semantics of the program

User can locate the error without looking at the program
solely in terms of declarative semantics

Examples – DD of incorrectness

Diagnosis sessions, to be shown after the first two items of the next slide

* A buggy insertion sort program [Shapiro'83]

* An actual bug in a rather big student program (from TDDD08, lab)

Reasons for DD being neglected

- No freedom: Fixed order or queries to answer
- The user cannot change her mind
- ...
- Exact specification (*intended model*) required from the user ☜
  But often she does not know it (and it does not matter)
  - member(e,t) for a non-list t,
  - append(l,t,t') for non-lists t,t',
  - insert(e,l,y) in insertion sort, for unsorted l,
  - a predicate may have distinct semantics in distinct versions of a program under development!

(see Howe&King SAT solver in [D...,TPLP2018])

Reasons for DD being neglected

- No freedom: Fixed order or queries to answer
- The user cannot change her mind
- ...
- Exact specification (*intended model*) required from the user ☞
  But often she does not know it (and it does not matter)
  - $\mathtt{member}(e, t)$ for a non-list t,
  - $\mathtt{append}(l, t, t')$ for non-lists t, t',
  - $\mathtt{insert}(e, l, y)$ in insertion sort, for unsorted l,
  - a predicate may have distinct semantics in distinct versions of a program under development!

    (see Howe&King SAT solver in [D...,TPLP2018])

Instead of “the intended model” the user knows

- its certain superset S_{corr} – what may be computed
- and a subset S_{compl} – what must be computed

i.e. an approximate specification

The program should be correct w.r.t. S_{corr} and complete w.r.t. S_{compl}:

$$S_{compl} \subseteq \mathbf{M}_P \subseteq S_{corr}$$

The standard Declarative Diagnosis works!

when instead of the intended model we use

- S_{corr} for incorrectness diagnosis
- S_{compl} for incompleteness diagnosis

Apparently, this simple fact has been unnoticed

Instead of "the intended model" the user knows

- its certain superset S_{corr} – what may be computed
- and a subset S_{compl} – what must be computed

i.e. an approximate specification

The program should be correct w.r.t. S_{corr} and complete w.r.t. S_{compl}:

$$S_{compl} \subseteq \mathbf{M}_P \subseteq S_{corr}$$

The standard Declarative Diagnosis works!

when instead of the intended model we use

- S_{corr} for incorrectness diagnosis
- S_{compl} for incompleteness diagnosis

Apparently, this simple fact has been unnoticed

Prolog debuggers

Prolog debugging tools – based solely on operational semantics

Worse, they are "declarative-programmer-unfriendly" $\ddot{\frown}$

Difficult to obtain info about e.g.

Which answers to a query A have been obtained?

What is the proof tree for a given obtained answer?
(i.e. which "local" answers contributed to a given "top level" answer?)

We need tools for DD for Prolog.

A basic tool for DD of incorrectness

Not an implementation of a DD algorithm,
but a proof tree browser.

A simple prototype.

(Used in the example diagnosis sessions.)

Summary. This work dealt with some basic issues of LP

- Simple method for proving correctness (old [Clark'79], but neglected)
- Proving completeness. (Hardly anybody has dealt with this previously)
- The usefulness of approximate specifications
- Explaining & solving the main (?) problem with DD
- A study when least Herbrand models exactly characterize programs, a sufficient and necessary condition.

* W. Drabent. "Logic + control: On program construction and verification." TPLP, 2018
* W. Drabent. "Correctness and Completeness of Logic Programs." ACM TOCL, 2016
* W. Drabent. "On definite program answers and least Herbrand models." TPLP, 2016

Conclusions

Declarative programming in Prolog possible;

reasoning about correctness / completeness
error diagnosis

can be dealt with declaratively (abstracting from operational semantics)

Proof methods for correctness/completeness can be used
more or less formally by programmers

At the informal end
they show how to reason about our programs
in a systematic / orderly way.

To be applied in everyday programming

Conclusions

Declarative programming in Prolog possible;

reasoning about correctness / completeness
error diagnosis

can be dealt with declaratively (abstracting from operational semantics)

Proof methods for correctness/completeness can be used
more or less formally by programmers

At the informal end
they show how to reason about our programs
in a systematic / orderly way.

To be applied in everyday programming