A Theoretical Framework for the Declarative Debugging of Datalog Programs

R. Caballero[1], Y. García-Ruiz[1], and F. Sáenz-Pérez[2,⋆]

[1] Departamento de Sistemas Informáticos y Computación,
[2] Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
rafa@sip.ucm.es, ygarciar@fdi.ucm.es, fernan@sip.ucm.es

Abstract. The logic programming language Datalog has been extensively researched as a query language for deductive databases. Although similar to Prolog, the Datalog operational mechanisms are more intricate, leading to computations quite hard to debug by traditional approaches. In this paper, we present a theoretical framework for debugging Datalog programs based on the ideas of declarative debugging. In our setting, a debugging session starts when the user detects an unexpected answer for some query, and ends with the debugger pointing to either an erroneous predicate or to a set of mutually recursive predicates as the cause of the unexpected answer. Instead of representing the computations by means of trees, as usual in declarative debugging, we propose graphs as a more convenient structure in the case of Datalog, proving formally the soundness and completeness of the debugging technique. We also present a debugging tool implemented in the publicly available deductive database system DES following this theoretical framework.

1 Introduction

Deductive databases rely on logic programming based query languages. Although not very well-known out of the academic institutions, some of their concepts are used in today relational databases to support advanced features of more recent SQL standards, and even implemented in major systems (e.g., the linear recursion provided in IBM's DB2 following the SQL-99 standard). A successful language for deductive databases has been Datalog [1], which allows users writing more expressive queries than relational databases. Relations and queries in Datalog are considered from a model-theoretic point of view, that is, thinking of relations as sets, and the language itself as a tool for manipulating sets and obtaining answer sets.

Raising the abstraction level generally implies a more complex computation mechanism acting as a black-box hidden from the user. Although this leads to more expressive programs, it also makes query debugging a very difficult

⋆ The authors have been partially supported by the Spanish National Project MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS-CAM(S-0505/TIC/0407).

process. An operational semantics oriented debugger is not helpful in this context, since the underlying computational mechanism is not directly related to the model-theoretic approach, but to implementation techniques such as *magic sets* [2] or *tabling* [3]. The few existing proposals for debugging specifically Datalog programs are usually based on “imperative” debugging, that try to follow the computation model to find bugs. These proposals are mainly based on forests of proof trees [4,5,6], which makes debugging a trace based task not so amenable to users. The first work related to the declarative debugging of Datalog programs is due to [7], but a variant of SLD resolution is used in order to look for program errors, imposing to traverse at least as many trees as particular answers are obtained for any query.

In the more general setting of *answer set programming* [8], there have been several proposals for diagnosing program errors in the last few years. In [9] a technique for detecting *conflict sets* is proposed. The paper explains how this approach can be used for detecting missing answers. Our proposal is limited to a more particular type of programs, namely stratified programs, but it can be applied for diagnosing not only missing but also *wrong* answers. In [10] the authors propose a technique that transforms programs into other programs with answer sets including debugging-relevant information about the original programs. This approach can be seen as a different, complementary view of the debugging technique described here.

In [11] we proposed a novel way of applying declarative debugging (also called *algorithmic debugging*, a term first coined in the logic programming field by E.H. Shapiro [12]), to Datalog programs. In that work, we introduced the notion of *computation graphs* (shortly *CG*s) as a suitable structure for representing and debugging Datalog computations. One of the virtues of declarative debugging is that it allows theoretical reasoning about the adequacy of the proposal. This paper addresses this task, proving formally the soundness and completeness of the debugging technique. We also present a prototype based in these ideas and included as part of a publicly available Datalog system DES [13].

The next section introduces the theoretical background needed for proving the properties of the debugger. Section 3 presents the concept of computation graph and proves several properties of *CG*s, while Section 4 includes the soundness and completeness results. Section 5 is devoted to discuss some implementation issues. Finally, Section 6 summarizes the work and presents the conclusions.

2 Datalog Programs

In this section, we introduce the syntax and semantics of Datalog programs and define the different types of errors that can occur in our setting. Although there are different proposals for this language, we will restrict our presentation to the language features included in the system DES [13]. Observe that the setting for Datalog presented here is a subsumed by the more general framework of *Answer Set Programming* [8].

2.1 Datalog Syntax

We consider (recursive) Datalog programs [14,15], i.e., normal logic programs without function symbols. In our setting, *terms* are either variables or constant symbols and *atoms* are of the form $p(t_1, \ldots, t_n)$, with p an n-ary predicate symbol and t_i terms for each $1 \leq i \leq n$. The notation $t_1, \ldots, t_n$ will be usually abbreviated as $\bar{t}_n$. A *positive* literal is an atom, and a *negative* literal is a negated atom. A negated atom is syntactically constructed as $not(A)$, where A is an atom. The atom contained in a literal L will be denoted as *atom(L)*. The set of variables of any formula F will be denoted as $var(F)$. A formula F is *ground* when $var(F) = \emptyset$.

A *rule* (or *clause* in the logic programming context) R has the form $p(\bar{t}_n) :- l_1, \ldots, l_m$ representing the first order logic formula $p(\bar{t}_n) \leftarrow l_1 \wedge \ldots \wedge l_m$, where l_i are literals for $i = 1 \ldots m$, and $m \geq 0$. The left-hand side atom $p(\bar{t}_n)$ will be referred to as the *head* of R, the right-hand side $l_1, \ldots, l_n$ as the *body* of R, and the literals l_i as *subqueries*. The variables occurring only in the body $l_1 \wedge \ldots \wedge l_m$ are assumed to be existentially quantified and the rest universally quantified. We require that $vars(H) \subseteq vars(B)$ for every program rule $H :- B$. A *fact* is a rule with empty body and ground head. The symbol :− is dropped in this case. The *definition of a relation* (or *predicate*) p in a program P consists of all the program rules with p in the head. A *query* (or *goal*) is a literal.

We consider stratified negation, a form of negation introduced in the context of deductive databases in [16]. A program P is called *stratified* if there is a partition $\{P_1, \ldots, P_n\}$ of P s.t. for $i = 1 \ldots n$:

1. If a relation symbol occurs in a positive literal of the body of any rule in P_i then its definition is contained in $\cup_{j \leq i} P_j$.
2. If a relation symbol occurs in a negative literal of the body of any rule in P_i then its definition is contained in $\cup_{j < i} P_j$.

We call each P_i a *stratum*. For instance, consider the Datalog program of Figure 1. We can check that the program is stratified by defining two strata: P_1 containing the rules for `star`, `orbits` and `intermediate`, and P_2 containing the rule for `planet`.

```
star(sun).
orbits(earth, sun).
orbits(moon, earth).
orbits(X,Y)        :- orbits(X,Z), orbits(Z,Y).
planet(X)          :- orbits(X,Y), star(Y), not(intermediate(X,Y)).
intermediate(X,Y) :- orbits(X,Y), orbits(Z,Y).
```

Fig. 1. A (buggy) Datalog Program

2.2 Program Models

We consider *Herbrand interpretations* and *Herbrand models*, i.e., Herbrand interpretations that make every Herbrand instance of the program rules logically true formulae. An *instance* of a formula is the result of applying the substitution θ to a formula F. We use the notation $F\theta$ instead of $\theta(F)$ for representing instances. The set *Subst* represents the set of all the possible substitutions. Often, we will be interested in *ground instances of a rule*, assuming implicitly that every rule is renamed with new variables each time it is selected. The composition operation between substitutions is defined in the usual way and fulfilling the property $(F\sigma)\theta = F(\sigma \cdot \theta)$ for all $\sigma, \theta \in Subst$. Two formulae φ, φ' are *variants* if $\varphi = \varphi'\theta$, where θ is a renaming, i.e., a bijection among variables. We say that $\sigma \in Subst$ is an *instance* of $\theta \in Subst$ when $\sigma = \theta\mu$, with μ some substitution. In this case, we write $\sigma \geq \theta$.

Given a Herbrand interpretation I for a the Datalog program P, we use the notation $I \models F$ to indicate that the formula F is true in I. The *meaning of a query* Q w.r.t. the interpretation I, denoted by Q_I, is the set of ground instances $Q\theta$ s.t. $I \models Q\theta$. If Q is an atom, then an equivalent definition is $Q_I = \{Q\theta \mid Q\theta \in I \text{ for some } \theta \in Subst\}$.

In logic programming without negation, the existence of a *least Herbrand model* for every program P is ensured, and it can be obtained as the least fixed point of a closure operator T_P, which is defined over any interpretation I as:

$$A \in T_P(I) \text{ iff for some rule } (H :- B) \in P, I \models B\theta \text{ and } A = H\theta$$

In these conditions, the least Herbrand model is defined as $T_P \uparrow \omega(\emptyset)$, i.e., as the fixed point obtained when iterating the operator starting at the empty interpretation. In general, however, the existence of the least Herbrand model is not ensured in programs using negation. Fortunately, due to the use of stratified programs in Datalog, the existence of a so-called *standard model*, which we will represent also as $\mathcal{M}$, is in any case ensured [14]. Given a program P stratified by the partition $\{P_1, \dots, P_k\}$, we define the sets $M_0 = \emptyset$, $M_1 = T_{P_1} \uparrow \omega(M_0)$, ..., $M_k = T_{P_k} \uparrow \omega(M_k - 1)$. Then, the standard model of P is defined as $\mathcal{M} = M_k$. The standard model verifies the following properties (the proofs can be found in [14]):

Proposition 1. *Let P be a program stratified by the partition $\{P_1, \dots, P_k\}$. Then:*

1. *$\mathcal{M}$ is a minimal model.*
2. *$\mathcal{M}$ is supported, i.e., for all $p(\bar{s}_n) \in \mathcal{M}$ there exists an* associated program rule *$(H :- B) \in P$ and an* associated substitution *$\theta \in Subst$ such that $p(\bar{s}_n) = H\theta$, $\mathcal{M} \models B\theta$ and $B\theta$ ground (due to our safety condition, $var(H) \subseteq var(B)$, which means that $H\theta$ is also ground).*
3. *Conversely, if there is some $(H :- B) \in P$, $\theta \in Subst$ s.t. $\mathcal{M} \models B\theta$, then $\mathcal{M} \models H\theta$.*
4. *$\mathcal{M}$ is independent of the stratification.*

5. *The following chain of inclusions holds:*

$$\begin{array}{llll}
\emptyset = M_0 & \subseteq T_{P_1}(M_0) & \subseteq T^2_{P_1}(M_0) & \subseteq \ldots \subseteq M_1 \\
M_1 & \subseteq T_{P_2}(M_1) & \subseteq T^2_{P_2}(M_2) & \subseteq \ldots \subseteq M_2 \\
 & & \ldots & \\
M_{k-1} & \subseteq T_{P_k}(M_{k-1}) & \subseteq T^2_{P_k}(M_{k-1}) & \subseteq \ldots \subseteq M_k = \mathcal{M}
\end{array}$$

Since functions are not allowed in Datalog, the standard model is finite and it can be actually computed. In fact, the deductive database systems such as DES are implemented to obtain the values $Q_\mathcal{M}$ for every query Q. Thus, $Q_\mathcal{M}$ will be referred to as the *answer* to Q. From now on, we assume that the Datalog system supporting the debugger verifies this condition, which is a reasonable requirement in the context of Datalog. This is different from the general setting of logic languages such as Prolog, even if we restrict to the case of Prolog programs without functions in the signature. For instance, consider the following dummy program:

```
p(X) :- q(X).     q(X) :- p(X).
```

The program is valid both in Prolog and in Datalog. However, the goal (resp. query) `p(X)` shows the difference between the two settings: In Prolog, it leads to a non-terminating computation, whereas in Datalog it succeeds with the answer $\{\}$, meaning that no ground instance of `p(X)` can be deduced from the program. Our selected system DES computes the answer to a query following a top-down approach, so that only the relevant information to obtain $Q_\mathcal{M}$ is computed in order to increase the efficiency of the computation.

The concept of standard model above is generalized by that of *stable model* [17], which can be applied also to non-stratified programs. However, in this work we restrict our semantics to stratified programs because this is a requirement of several Datalog systems.

2.3 Correct and Incorrect Programs

We use the term *intended interpretation*, denoted by $\mathcal{I}$, to denote the Herbrand model the user has in mind for the program. If $\mathcal{M} = \mathcal{I}$, we say that the program is *well-defined*, and if $\mathcal{M} \neq \mathcal{I}$ we say that the program is *buggy*. Declarative debugging assumes that the user focus on query answers for comparing the intended interpretation to the standard Herbrand model actually computed. Thus, we say that $Q_\mathcal{M}$ is an *unexpected answer* for a query Q if $Q_\mathcal{M} \neq Q_\mathcal{I}$. An unexpected answer can be either a *wrong answer*, when there is some $Q\theta \in Q_\mathcal{M}$ s.t. $Q\theta \notin Q_\mathcal{I}$, or a *missing answer*, when there is $Q\theta \in Q_\mathcal{I}$ s.t. $Q\theta \notin Q_\mathcal{M}$. In the first case, $Q\theta$ is a *wrong instance*, while in the second one $Q\theta$ is a *missing instance*. Observe that an unexpected answer can be both missing and wrong at the same time. The next proposition indicates that an unexpected answer to a positive query implies an unexpected answer to its negation.

Proposition 2. *Let P be a program containing at least one constant, $\mathcal{I}$ its intended model and Q a positive query. Then, $Q_\mathcal{M}$ is a missing answer for Q iff*

$(\neg Q)_{\mathcal{M}}$ is a wrong answer for $\neg Q$, and $Q_{\mathcal{M}}$ is a wrong answer for Q iff $(\neg Q)_{\mathcal{M}}$ is a missing answer for $\neg Q$.

Proof. Straightforward from the definition of meaning of a query w.r.t. an interpretation, since $Q_I \cap (\neg Q)_I = \emptyset$ in every interpretation I. Then, $p(\bar{t}_n) \notin Q_{\mathcal{M}}$ and $p(\bar{t}_n) \in Q_{\mathcal{I}}$, i.e., if $p(\bar{t}_n)$ is a missing instance and $Q_{\mathcal{M}}$ is a missing answer, iff $p(\bar{t}_n) \in (\neg Q)_{\mathcal{M}})$, $p(\bar{t}_n) \notin (\neg Q)_{\mathcal{I}}$, i.e., $p(\bar{t}_n)$ is a wrong instance and $(\neg Q)_{\mathcal{M}}$ is a wrong answer for $\neg Q$. Analogous for the other case. $\square$

An unexpected answer indicates that the program is erroneous, and it will be considered as the initial symptom for a user to start the debugging process. The two usual causes of errors considered in the declarative debugging of logic programs are *wrong* and *incomplete* relations:

Definition 1 (Wrong Relation). *Let P be a Datalog program. We say that $p \in P$ is a **wrong relation** w.r.t. $\mathcal{I}$ if there exist a rule variant $p(\bar{t}_n) :- l_1, \ldots, l_m$ in P and a substitution θ such that $\mathcal{I} \models l_i\theta$, $i = 1 \ldots m$ and $\mathcal{I} \not\models p(\bar{t}_n)\theta$.*

Definition 2 (Incomplete Relation). *Let P be a Datalog program. We say that $p \in P$ is an **incomplete relation** w.r.t. $\mathcal{I}$ if there exists an atom $p(\bar{s}_n)\theta$ s.t. $\mathcal{I} \models p(\bar{s}_n)\theta$ and, for each rule variant $p(\bar{t}_n) :- l_1, \ldots, l_m$ and substitution θ', either $p(\bar{t}_n)\theta' \neq p(\bar{s}_n)\theta$ or $\mathcal{I} \not\models l_i\theta'$ for some l_i, $1 \leq i \leq m$.*

In Datalog we also need to consider another possible cause of errors, namely the *incomplete set of relations*. This concept depends on the auxiliary definition of uncovered set of atoms.

Definition 3 (Uncovered Set of Atoms). *Let P be a Datalog program and $\mathcal{I}$ an intended interpretation for P. Let U be a set of atoms s.t. $\mathcal{I} \models p(\bar{s}_n)$ for each $p(\bar{s}_n) \in U$. We say that U is an **uncovered set of atoms** if for every rule $p(\bar{t}_n) :- l_1, \ldots, l_m$ in P and substitution θ s.t.:*

- *$p(\bar{t}_n)\theta \in U$,*
- *$\mathcal{I} \models l_i\theta$ for $i = 1 \ldots m$*

there is some $l_j\theta \in U$, $1 \leq j \leq m$, with l_j a positive literal.

Now, we are ready for defining the third kind of error, which generalizes the idea of incomplete relation:

Definition 4 (Incomplete Set of Relations). *Let P be a Datalog program and S a set of relations defined in P. We say that S is an **incomplete set of relations** in P iff exists an uncovered set of atoms U s.t. for each relation $p \in S$, $p(\bar{t}_n) \in U$ for some $t_1, \ldots, t_n$.*

To the best of our knowledge, this error has not been considered in the literature about Datalog debugging so far, but it is necessary for correctly diagnosing Datalog programs. Consider again the program `p(X):- q(X). q(X):-p(X).` with the intended interpretation $I = \{p(a), q(a)\}$ and the query `p(X)`. The computed answer {} is a missing answer with `p(a)` as missing instance. However, neither of the two relations is incomplete, because their rules can produce the values `p(a)`, `q(a)` by means of the instance given by the substitution $\theta = \{X \mapsto a\}$. So, $U = \{p(a), q(a)\}$ is an uncovered set of atoms and hence $S = \{p, q\}$ is an incomplete set of relations.

We say that a relation is *buggy* when it is wrong, incomplete or member of an incomplete set of relations, and that it is well-defined otherwise. Observe that, due to the use of negation, a wrong answer does not correspond always to a wrong relation. For instance, in the following program:

```
p(X) :-  r(X), not(q(X)).
% missing q(a).
r(a).
```

with intended interpretation $\mathcal{I} = \{q(a), r(a)\}$ the query `p(X)` produces the wrong answer {`p(a)`} but there is no wrong relation in the program and instead there is an incomplete relation (q).

As an example, consider the program of Figure 1. This program defines a relation `orbits` by two facts and a rule establishing the transitive closure of the relation. A relation `star` is defined by one fact and indicates that the sun is a star. The relation `intermediate` is defined in terms of `orbits`, relating two bodies `X` and `Y` whenever there is some intermediate body between them. Finally, `planet` is defined as a body `X` that orbits directly a star `Y`, without any other body in between. However, a mistake has been introduced in the program: The underlined `Y` in the rule for `intermediate` should be `Z`. As a consequence, the query `planet(X)` yields the missing answer {} (assuming that the atom `planet(earth)` is in $\mathcal{I}$). In the next section, we will show how such errors can be detected by using declarative debugging based on computation graphs.

3 Computation Graphs

In this section, we define a structure for representing Datalog computations and prove their adequacy for declarative debugging.

3.1 Graph Terminology

We consider finite *directed graphs* $G = (V, E)$, where V is a finite set of vertices and E a finite set of directed edges, $E \subseteq V \times V$. Often, we use the notation $v \in G$ instead of $v \in V$ and $(u, v) \in G$ instead of $(u, v) \in E$. Given any vertex $u \in G$ we say that $v \in G$ is a *successor* of u in G if $(u, v) \in G$, which we represent by the notation $succ_G(u, v)$.

Given $G = (V, E)$, we say that $G' = (V', E')$ is a *subgraph* of G if G' is a graph s.t. $V' \subseteq V$ and $E' \subseteq E$. A particular case of subgraph is the *subgraph generated* from a subset of vertices $V' \subseteq V$. This subgraph is of the form $G' = (V', E')$, where $E' = \{(u, v) \in G \mid u, v \in V'\}$.

In a directed graph, the *output degree* of a vertex $v \in G$ is the cardinal of the set $\{u \in G \mid (v,u) \in G\}$ and it is represented by $gr^+_G(v)$. Analogously, the value $gr^-_G(v) = \mid \{u \in G \mid (u,v) \in G\} \mid$ represents the *input degree* of v. These concepts can be naturally extended to subgraphs by defining $gr^+_G(G') = \mid \{(u,v) \in G \mid u \in G', v \notin G'\} \mid$, $gr^-_G(G') = \mid \{(v,u) \in G \mid u \in G', v \notin G'\} \mid$. We remove the subindex G in gr^+_G, gr^-_G whenever the reference to the graph considered cannot be ambiguous in the context.

A sequence of vertices $u_1, u_2, \ldots, u_n$ of G such that $(u_i, u_{i+1}) \in G$ for all $i = 1 \ldots n-1$ are called a *walk* from u_1 to u_n. A walk s.t. $u_1 = u_n$ is called a *circuit*. A walk with no repeated vertices except maybe the first and the last vertex is called a *path*. If indeed $u_1 = u_n$ the path is called a *cycle*, i.e., a cycle is a special case of circuit with exactly one vertex repeated. The notation $path_G(u,v)$ represents a path starting at u and ending in v in some graph G.

A directed graph G is called *strongly connected* if, for every pair of vertices $u, v \in G$, there is a path from u to v and a path from v to u. The strongly connected components of a directed graph are its maximal strongly connected subgraphs, and they form a partition of G.

3.2 Datalog Computation Graphs

The *computation graph* (CG in short) for a query Q w.r.t. a program P is a directed graph $G = (V, E)$ such that each vertex V is of the form $(Q', Q'_{\mathcal{M}})$, where Q' is a subquery produced during the computation, and $Q'_{\mathcal{M}}$ is the computed answer for Q'. The next definition includes the construction of a computation graph. Observe that the answers of the subqueries are not relevant for the graph structure and, therefore, they are included as part of the vertices in a last step.

Definition 5 *(***Computation Graph***). Let P be a Datalog program and Q a query either of the form $p(\bar{a}_n)$ or $not(p(\bar{a}_n))$. The computation graph for Q w.r.t. P is represented by a pair (V, E) of vertices and edges defined as follows:*

The construction of the graph uses an auxiliary set A for containing the vertices that must be expanded in order to complete the graph.

1. *Put $V = A = \{p(\bar{a}_n)\}$ and $E = \emptyset$.*

2. *While $A \neq \emptyset$ do:*
   - *(a) Select a vertex u in A with query $q(\bar{b}_n)$. $A = A \setminus \{u\}$.*
   - *(b) For each rule R defining q, $R = (q(\bar{t}_n) :\!-\; l_1, \ldots, l_m)$ with $m > 0$, such that there exists $\theta = mgu(\bar{t}_n, \bar{b}_n)$, the debugger creates a set S of new vertices. Initially, we define $S = \emptyset$ and include new vertices associated to each literal l_i, $i = 1 \ldots m$ as follows:*
     - *i. $i = 1$, a new vertex is included: $S = S \cup \{atom(l_1)\theta\}$.*
     - *ii. $i > 1$. We consider the literal l_i. For each set of substitutions $\{\sigma_1, \ldots, \sigma_i\}$ with $dom(\sigma_1 \cdot \ldots \cdot \sigma_{i-1}) \subseteq var(l_1) \cup \cdots \cup var(l_i)$ such that for every $1 < j \leq i$:*
       - *$atom(l_{j-1})(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}) \in S$, and*

– $l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_j) \in (l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}))_{\mathcal{M}}$
include a new vertex in S:

$$S = S \cup \{atom(l_i)(\sigma_1 \cdot \ldots \cdot \sigma_i)\}$$

(c) For each vertex $v \in S$, test whether there exists already a vertex $v' \in V$ such that v and v' are variants (i.e., there is a variable renaming). There are two possibilities:
  - *There is such a vertex v'. Then, $E = E \cup \{(u, v')\}$. That is, if the vertex already exists, we simply add a new edge from the selected vertex u to v'.*
  - *Otherwise, $V = V \cup \{v\}$, $A = A \cup \{v\}$, and $E = E \cup \{(u, v)\}$.*

3. Complete the vertices including the computed answer $Q_{\mathcal{M}}$ of every subquery Q.

End of Definition

We will use the notation $[Q = Q_{\mathcal{M}_A}]$ for representing the content of the vertices. The values $Q_{\mathcal{M}_A}$ included at step 3 can be obtained from the underlying deductive database system by submitting each Q. The vertex is *valid* if $Q_{\mathcal{M}_A}$ is the expected answer for Q, and *invalid* otherwise.

Figure 2 shows the *CG* for the query `planet(X)` w.r.t. the program of Figure 1. The first vertex included in the graph at step 1 corresponds to `planet(X)`.

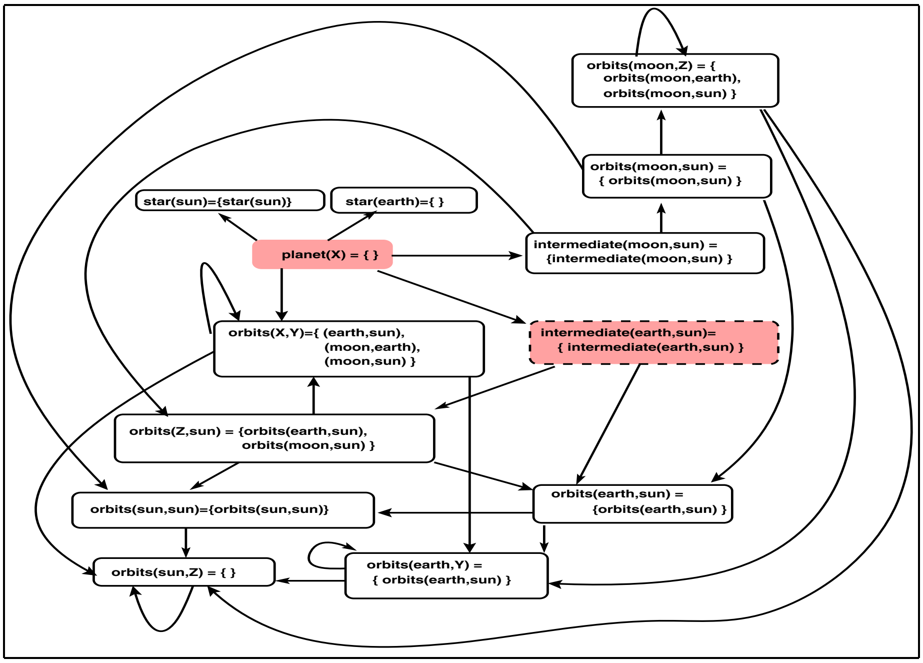
orbits(moon,Z) = { orbits(moon,earth), orbits(moon,sun) }
orbits(moon,sun) = { orbits(moon,sun) }
star(sun)={star(sun)}
star(earth)={ }
planet(X) = { }
intermediate(moon,sun) = {intermediate(moon,sun) }
orbits(X,Y)={ (earth,sun), (moon,earth), (moon,sun) }
intermediate(earth,sun)= { intermediate(earth,sun) }
orbits(Z,sun) = {orbits(earth,sun), orbits(moon,sun) }
orbits(sun,sun)={orbits(sun,sun)}
orbits(earth,sun) = {orbits(earth,sun) }
orbits(sun,Z) = { }
orbits(earth,Y) = { orbits(earth,sun) }

Fig. 2. CG for the Query `planet(X)` w.r.t. the Program of Figure 1

From this vertex and by using the only program rule for `planet`, four new vertices are added, the first one corresponding to the first literal `orbits(X,Y)`. Since two values of `Y` satisfy this subquery, namely `Y=sun` and `Y=earth`, the definition introduces two new vertices for the next literal `star(Y)`, `star(sun)` and `star(earth)`. The last one produces the empty answer, but `star(sun)` succeeds. Then, the last literal in the rule, `not(intermediate(X,Y))`, yields vertices for the two values of `X` and the only value of `Y` that satisfies the two previous literals. Observe, however, that the vertices for this literal are introduced in the graph without the negation, i.e., the *CG* will contain only subqueries for atoms. This simplifies the questions asked to the user during the navigation phase, and can be done without affecting the correctness of the technique because the validity of the positive literal implies the validity of its negation, and the other way round (although the type of associated error changes, see Proposition 2). The rest of the vertices of the example graph are built expanding the successors of `planet(X)` and repeating the process until no more vertices can be added.

The termination of the process is guaranteed because in our setting the signature is finite and the *CG* cannot have two occurrences of the same vertex due to step 2c, which introduces edges between existing vertices instead of creating new ones when possible.

The next proposition relates the elements of the computed answer stored at a vertex u with the immediate successors of u and vice versa.

Proposition 3. *Let* $u = [Q = Q_{\mathcal{M}}]$ *be a vertex in the computation graph* G *of some query w.r.t. a program* P. *Let* $p(\bar{s}_n)$ *be an instance of* Q. *Then* $p(\bar{s}_n) \in Q_{\mathcal{M}}$ *iff there exist a rule variant* $p(\bar{t}_n) :- l_1, \ldots, l_m$ *and a substitution* θ *such that among the successors of* u *in* G *there are vertices of the form* $[atom(l_i)\sigma_i = A_i]$ *with* $\theta \geq \sigma_i$ *for each* $i = 1 \ldots m$.

Proof. First, we suppose that $p(\bar{s}_n) \in Q_{\mathcal{M}}$. Let $(p(\bar{t}_n) :- l_1, \ldots, l_m) \in P$ and $\theta \in Subst$ be respectively the associated rule and the associated substitution to $p(\bar{s}_n)$, as defined in Proposition 1, item 2. Then, by this proposition, $p(\bar{s}_n) = p(\bar{t}_n)\theta$, which implies the existence of the $mgu(p(\bar{t}_n), p(\bar{s}_n))$ because we always consider rule variants and, hence, $var(p(\bar{s}_n)) \cap var(p(\bar{t}_n)) = \emptyset$. Then, the algorithm of Definition 5, item 2b, ensures that this program rule produces new vertices, successors of u in G. We check by induction on the number of literals in the body rule, m, that these vertices are of the form $[atom(l_i)\sigma_i = A_i]$ with $\theta \geq \sigma_i$ for $i = 1 \ldots m$. If $m = 0$, the result holds trivially. If $m = 1$, then there is a successor of u of the form $[atom(l_1)\theta = A_1]$ (item 2(b)i of Definition 5). For the inductive case $m > 1$, we assume that there is already a successor of u of the form $[atom(l_{m-1})\sigma = A_{m-1}]$, $\theta \geq \sigma$, i.e., $\theta = \sigma \cdot \sigma_m$ for some substitution σ_m. By the graph construction algorithm, σ must be of the form $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_{m-1}$. By Proposition 1, item 2, $M \models l_{m-1}\theta$, i.e., $l_{m-1}\theta \in A_{m-1}$ (by the same Proposition 1, $l_{m-1}\theta$ is ground, and therefore must be part of the computed answer for $l_{m-1}\sigma$). Hence, $l_{m-1}(\sigma_1 \cdot \ldots \cdot \sigma_m) \in A_{m-1}$. In these conditions, the algorithm of Definition 5 includes a new successor of u with the form $atom(l_m)(\sigma_1 \cdot \ldots \cdot \sigma_m)$.

Conversely, if there exists a program rule, a substitution, and successor vertices as the proposition indicates, then it can be proved by a similar reasoning that $M \models (l_1, \dots, l_m)\theta$, and then, Proposition 1, item 3, ensures that $p(\bar{s}_n) = p(\bar{t}_n)\theta$ verifies $\mathcal{M} \models p(\bar{s}_n)$. In particular, if $p(\bar{s}_n)$ is ground, this means that $p(\bar{s}_n) \in \mathcal{M}$. □

The relation among a vertex and its descendants also relates the validity of them, as the following proposition states:

Proposition 4. *Let G be a computation graph and $u = [p(\bar{s}_n) = A]$ be an invalid vertex of G such that p is a well-defined relation. Then, u has some invalid successor v in G.*

Proof. If the vertex u is invalid, then A is either a wrong or a missing answer for $p(\bar{s}_n)$, which means that it contains either a wrong or a missing instance.

Suppose that $p(\bar{s}_n)\theta$ is a wrong instance for some $\theta \in subst$. Since $p(\bar{s}_n)\theta \in (p(\bar{s}_n))_{\mathcal{M}}$, by Proposition 1, there exists some associated program rule $R \in P$ and substitution θ' s.t. $(R)\theta' = (p(\bar{t}_n)\ :-l_1, \dots l_m)\theta'$, with $\mathcal{M} \models l_i\theta'$ for all $i = 1 \dots m$ and $p(\bar{t}_n)\theta' = p(\bar{s}_n)\theta$. From Proposition 3, it can be deduced that there are successor vertices of u of the form $[atom(l_i)\sigma_i = A_i]$ for all $i = 1 \dots m$, with $\theta' \geq \sigma_i$. Assume that all these vertices are valid. Then, for each $i = 1 \dots m$ we can ensure the validity of $l_i\theta'$ because:

- If l_i is a positive literal, from the validity of the answer for $atom(l_i)\sigma_i$ we obtain the validity of the more particular $atom(l_i)\theta'$ (the validity of a formula entails the validity of its instances).
- If l_i is a negative literal, from the validity of the answer for $atom(l_i)\sigma_i$ we obtain the validity of the answer for $atom(l_i)\theta'$, and from this, the validity of the answer for $l_i\theta'$ (as a consequence of Proposition 2).

Then, we have that $\mathcal{M} \models (l_1, \dots, l_m)\theta'$, but $\mathcal{M} \not\models p(\bar{t}_n)\theta'$, i.e., $(R)\theta'$ is a wrong instance. But this is not possible because p is well-defined. Therefore, some of the successors of u must be invalid.

The proof is analogous in the case of a missing answer. □

3.3 Buggy Vertices and Buggy Circuits

In the traditional declarative debugging scheme [18] based on trees, program errors correspond to *buggy nodes*. In our setting, we also need the concept of buggy node, here called *buggy vertex*, but in addition our computation graphs can include *buggy circuits*:

Definition 6 (Buggy Circuit). *Let $CG = (V, A)$ be a computation graph. We define a* buggy circuit *as a circuit $W = v_1 \dots v_n$ s.t. for all $1 \leq i \leq n$:*

1. *v_i is invalid.*
2. *If $(v_i, u) \in A$ and u is invalid then $u \in W$.*

Definition 7 (Buggy Vertex). *A vertex is called buggy when it is invalid but all its successors are valid.*

The next result proves that a computation graph corresponding to an initial error symptom, i.e., including some invalid vertex, contains either a buggy circuit or a buggy vertex.

Proposition 5. *Let G be a computation graph containing an invalid vertex. Then, G contains either a buggy vertex or a buggy circuit.*

Proof. Let G be the computation graph and $u \in G$ an invalid vertex. From G, we obtain a new graph G' by including all the invalid vertices reachable from u. More formally, G' is the subgraph of G generated by the set of vertices

$$\{v \in G \mid \text{ there is a path } \Pi = path_G(u, v) \text{ and } w \text{ invalid for every } w \in \Pi\}$$

Now, we consider the set S of strongly connected components in G',

$$S = \{C \mid C \text{ is a strongly connected component of} \, G'\}$$

The cardinality of S is finite since G' is finite. Then, there must exist $C \in S$ such that $gr^{+}_{G'}(C) = 0$. Moreover, for all $u \in C$, $succ_G(u, u')$ means that either $u' \in C$ or u' is valid because $u' \notin C$, u' invalid, would imply $gr^{+}_{G'}(C) > 0$. Observe also that, by the construction of G', every $u \in C$ is invalid. Then:

– If C contains a single vertex u, then u is a buggy vertex in G.
– If C contains more than a vertex, then all its vertices form a buggy circuit in G. □

4 Soundness and Completeness

The debugging process we propose can be summarized as follows:

1. The user finds out an unexpected answer for some query Q w.r.t. some program P.
2. The debugger builds the computation graph G for Q w.r.t. P.
3. The graph is traversed, asking questions to the user about the validity of some vertices until a buggy vertex or a buggy circuit has been found.
4. If a buggy vertex is found, its associated relation is pointed out as buggy. If instead a buggy circuit is found, the set of relations involved in the circuit are shown to the user indicating that at least one of them is buggy or that the set is incomplete.

Now, we must check that the technique is reliable, i.e., that it is both sound and complete. First we need some auxiliary lemmata.

Lemma 1. *Let G be a computation graph for some query Q w.r.t. a program P, and let $C = u_1, \ldots, u_k$, with $u_k = u_1$ be a circuit in G. Then, all the u_i are of the form $[Q_i = Q_{i\mathcal{M}}]$ with Q_i associated to a positive literal in its corresponding program rule for $i = 1 \ldots k - 1$.*

Proof. It can be proved that every relation occurring in some Q_i depends recursively on itself. This means that Q_i cannot occur negatively in a clause because this would mean than P is not stratified (see Lemma 1 in [14]). $\square$

Lemma 2. *Let $v = [p(\bar{s}_n) = \dots]$ be a vertex of some CG G obtained w.r.t. some program P with standard model $\mathcal{M}$. Let $p(\bar{t}_n)$:− $l_1, \dots, l_m$ be a rule in P, and θ s.t. $p(\bar{t}_n)\theta = p(\bar{s}_n)\theta$, and that $\mathcal{M} \models l_1\theta, \dots, l_k\theta$ for some $1 \le k \le m$. Then, v has children vertices in G of the form $[atom(l_i)\theta_i = \dots]$ for $i = 1 \dots k+1$, with $\theta \ge \theta_i$.*

Proof. The proof corresponds to that of Proposition 3, but considering only the first $k+1$ literals of the program rule. $\square$

Observe that theoretically the debugger could be applied to any computation graph even if there is no initial wrong or missing answer. The following soundness result ensures that in any case it will behave correctly.

Proposition 6 (Soundness). *Let P be a Datalog program, Q be a query and G be the computation graph for Q w.r.t. P. Then:*

1. *Every buggy node in G is associated to a buggy relation.*
2. *Every buggy circuit in G contains either a vertex with an associated buggy relation or an incomplete set of relations.*

Proof

1. Suppose that G contains a buggy vertex $u \equiv [q(\bar{t}_n) = S]$. By definition of buggy vertex, all the immediate descendants of u are valid. Since vertex u is invalid, by Proposition 4, the relation q cannot be well-defined.
2. Suppose that G contains a buggy circuit $C \equiv u_1, \dots, u_n$ with $u_n = u_1$ and each u_i of the form $[A_i = S_i]$ for $i = 1 \dots n-1$. We consider two possibilities:
   (a) At least one of the vertices in the circuit contains a wrong answer. Let S be the set of the wrong atom instances contained in the circuit:
   $$S = \{B \in S_i \wedge B \notin \mathcal{I} \mid \text{ for some } 1 \le i < n\}$$
   Obviously, $S \subseteq \mathcal{M}$ and $S \cap \mathcal{I} = \emptyset$. Now, we consider a stratification $\{P_1, \dots, P_k\}$ of the program P and the sequence of Herbrand interpretations starting from $\emptyset$ and ending in $\mathcal{M}$ defined in item 5 of Proposition 1. We single out the first interpretation in this sequence including some element of S. Such interpretation must be of the form $T_{P_i}(I)$, with I the previous interpretation in the sequence and $1 \le i \le k$. Let $p(\bar{s}_n)$ be an element of $T_{P_i}(I) \cap S$. By definition of T_P, there exists a substitution θ and a program rule $(p(\bar{t}_n)$:− $l_1, \dots, l_m) \in P$ s.t. $p(\bar{s}_n) = p(\bar{t}_n)\theta$ and that $I \models l_i\theta$ for every $i = 1 \dots m$. By Proposition 3, each l_i, $i = 1 \dots m$, has some associated vertex V' successor of V in the CG with V' of the form $[atom(l_i)\sigma = \dots]$ with σ more general than θ. We distinguish two possibilities:

- V' is out of the circuit. Then, by the definition of buggy circuit V' is valid w.r.t. $\mathcal{I}$, which means all the instances of $l_i\theta$ are also valid w.r.t. $\mathcal{I}$. This is true independently of whether l_i is positive or negative because the validity of the answer for a query implies the validity of the answer for its negation in our setting.
- V' is in the circuit. Then, l_i is positive due to Lemma 1, and by construction, all the instances of $l_i\theta$ included in $\mathcal{I}$ are valid w.r.t. $\mathcal{I}$.

In any case, $\mathcal{I} \models l_i\theta$ for every $i = 1 \dots m$ but $p(\bar{t}_n)\theta \notin \mathcal{I}$ and hence p is an *incorrect relation.*

(b) If none of the vertices in the buggy circuit contains a wrong answer, then every vertex contains a missing answer.
Put

$$S = \{A_i\sigma \in \mathcal{I}, A_i\sigma \notin S_i \ \mid \ \text{for some } 1 \leq i < k\}$$

i.e., S is the set of missing instances in the circuit. Next, we check that S is an uncovered set of atoms, which means that the relations in the buggy circuit form an incomplete set of relations. Let $A_j\sigma \in S$ be an atom of S with $1 \leq j \leq k$, $(p(\bar{t}_n) \ :- l_1, \dots, l_m) \in P$ be a program rule, and $\theta \in \mathit{Subst}$ such that:
- $p(\bar{t}_n)\theta = A_j\sigma$,
- $\mathcal{I} \models l_i\theta$ for $i = 1 \dots m$

There must exist at least one $l_i\theta \notin \mathcal{M}$, $1 \leq i \leq m$, otherwise $A_j\sigma$ would be in $\mathcal{M}$. Let r be the least index, $1 \leq r \leq m$, s.t. $l_r\theta \notin \mathcal{M}$. By Lemma 2, there is a successor of $[A_j = S_j]$ in G of the form $w = [l_r\theta' = S_r]$ with $\theta \geq \theta'$. Then, $l_r\theta$ is a missing answer for w, i.e., it is an invalid vertex (it is easy to prove that, if $l\theta$ has a missing answer, then $l\theta'$ has a missing answer for every θ' s.t. $\theta \geq \theta'$). This implies that $w \in C$, and hence l_r is a positive literal (by Lemma 1), $l_i\theta \in S$, and S is uncovered. □

After the soundness result, it remains to prove that the technique is complete:

Proposition 7 (Completeness). *Let P be a Datalog program and Q be a query with answer $Q_{\mathcal{M}}$ unexpected. Then, the computation graph G for Q w.r.t. P contains either a buggy node or a buggy circuit.*

Proof. By the construction of the computation graph, G contains a vertex for $[\mathit{atom}(Q) = \mathit{atom}(Q)_{\mathcal{M}}]$. If Q is positive, then $Q = \mathit{atom}(Q)$ and the vertex is of the form $[Q = Q_{\mathcal{M}}]$. Then, by hypothesis, $Q_{\mathcal{M}}$ is unexpected, and therefore the vertex is invalid. If Q is negative and it has an unexpected answer, it is straightforward to check that $\mathit{atom}(Q)$ also produces an unexpected answer and hence $[\mathit{atom}(Q) = \mathit{atom}(Q)_{\mathcal{M}}]$ is also invalid. Then, the result is a direct consequence of Proposition 5.

5 Implementation

The theoretical ideas explained so far have been implemented in a debugger included as part of the Datalog system DES [13]. The *CG* is built after the user has detected some unexpected answer. The values $(Q, Q_{\mathcal{M}_A})$ are stored along the computation and can be accessed afterwards without repeating the computation, thus increasing the efficiency of the graph construction.

A novelty of our approach is that it allows the user to choose working either at clause level or at predicate level, depending on the grade of precision that the user needs, and its knowledge of the intended interpretation $\mathcal{I}$. At predicate level, the debugger is able to find a buggy relation or an incomplete set of relations. At clause level, the debugger can provide additional information, namely the rule which is the cause of error.

For instance, next is the debugging session at predicate level for the query `planet(X)` w.r.t. our running example:

```
DES> /debug planet(X) p

Info: Starting debugger...

Is orbits(sun,sun) = {}  valid(v)/invalid(n)/abort(a) [v]? v
Is orbits(earth,Y) = {orbits(earth,sun)}
                      valid(v)/invalid(n)/abort(a) [v]? v
Is intermediate(earth,sun) = {intermediate(earth,sun)}
                      valid(v)/invalid(n)/abort(a) [v]? n
Is orbits(sun,Y) = {}  valid(v)/invalid(n)/abort(a) [v]? v
Is orbits(X,sun) = {orbits(earth,sun),orbits(moon,sun)}
                          valid(v)/invalid(n)/abort(a) [v]? v

Error in relation: intermediate/2
Witness query:
      intermediate(earth,sun) = {intermediate(earth,sun)}
```

The first question asks whether the query `orbits(sun,sun)` is expected to fail, i.e., it yields no answer. This is the case because we do not consider the sun orbiting around itself. The answer to the second question is also `valid` because the earth orbits only the sun in our intended model. But the answer to the next question is `invalid`, since the query `intermediate(earth,sun)` should fail because the earth orbits directly the sun. The next two answers are `valid`, and with this information the debugger determines that there is a buggy node in the *CG* corresponding to the relation `intermediate/2`, which is therefore buggy. The *witness query* shows the instance that contains the unexpected instance. This information can be useful for locating the bug.

In order to minimize the number of questions asked to the user, the tool relies on a navigation strategy similar to the *divide & query* presented in [12] for deciding which vertex is selected at each step. In other paradigms it has been shown that this strategy requires an average of $\log_2 n$ questions to find the bug [19], with n the number of nodes in the computation tree. Our experiments confirms that this is also the case when the CGs are in fact trees, i.e., they do not contain cycles, which occurs very often. In the case of graphs containing cycles the results also show this tendency, although a more extensive number of experiments is still needed.

6 Conclusions and Future Work

We have applied declarative debugging to Datalog programs. The debugger detects incorrect fragments of code starting from an unexpected answer. In order to find the bug, the tool requires the help of the user as an external oracle answering questions about the validity of the results obtained for some subqueries. We have proved formally the completeness and soundness of the technique, thus proposing a solid foundations for the debugging of Datalog programs. During the theoretical study, we have found that the traditional errors considered usually in logic programming are not enough in the case of Datalog where a new kind of error, the incomplete sets of predicates, can occur.

The theoretical ideas have been set in practice by developing a declarative debugger for the Datalog system DES. The debugger allows diagnosing both missing and wrong answers, which constitute all the possible errors symptoms of a Datalog program. Although a more extensive workbench is needed, the preliminary experiments are encouraging about the usability of the tool. The debugger allows to detect readily errors which otherwise would take considerable time. This is particularly important for the DES system, which has been developed with educational purposes. By using the debugger, the students can find the errors in a program by considering only its declarative meaning and disregarding operational issues.

From the point of view of efficiency, the results are also quite satisfactory. The particular characteristics of DES make all the information necessary for producing the graph available after each computation. The answers to each subquery, therefore, are not actually computed in order to build the graph but simply pointed to. This greatly speeds up the graph construction and keeps small the size of the graph even for large computations.

As future work, we consider the possibility of allowing more elaborated answers from the user. For instance, indicating that a vertex is not only invalid but also that it contains a wrong answer. The identification of such an answer can greatly reduce the number of questions. Another task is to develop and compare different navigation strategies for minimizing the number of questions needed for finding the bug.

References

1. Ramakrishnan, R., Ullman, J.: A survey of research on Deductive Databases. The Journal of Logic Programming 23(2), 125–149 (1993)
2. Beeri, C., Ramakrishnan, R.: On the power of magic. In: Proceedings of the Sixth ACM Symposium on Principles of Database Systems, pp. 269–284 (1987)
3. Dietrich, S.W.: Extension tables: Memo relations in logic programming. In: SLP, pp. 264–272 (1987)
4. Arora, T., Ramakrishnan, R., Roth, W.G., Seshadri, P., Srivastava, D.: Explaining program execution in deductive systems. In: Deductive and Object-Oriented Databases, pp. 101–119 (1993)
5. Wieland, C.: Two explanation facilities for the deductive database management system DeDEx. In: Kangassalo, H. (ed.) ER 1990, pp. 189–203, ER Institute (1990)

6. Specht, G.: Generating explanation trees even for negations in deductive database systems. In: Proceedings of the 5th Workshop on Logic Programming Environments, Vancouver, Canada (1993)
7. Russo, F., Sancassani, M.: A declarative debugging environment for Datalog. In: Proceedings of the First Russian Conference on Logic Programming, pp. 433–441. Springer, London (1992)
8. Baral, C.: Knowledge representation, reasoning, and declarative problem solving. Cambridge University Press, Cambridge (2003)
9. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proceedings of the 11th International Workshop on Non-Monotonic Reasoning, Lake District, UK, pp. 77–84 (May 2006)
10. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP Programs by Means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
11. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A new proposal for debugging datalog programs. In: 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2007) (June 2007)
12. Shapiro, E.: Algorithmic Program Debugging. In: ACM Distiguished Dissertation. MIT Press, Cambridge (1982)
13. Sáenz-Pérez, F.: Datalog Educational System. User's Manual. Technical Report 139-04, Facultad de Informática, Universidad Complutense de Madrid (2004), `http://des.sourceforge.net/`
14. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of deductive databases and logic programming, pp. 89–148. Morgan Kaufmann Publishers Inc., San Francisco (1988)
15. Ullman, J.: Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies). Computer Science Press (1995)
16. Chandra, A.K., Harel, D.: Horn clauses queries and generalizations. J. Log. Program. 2(1), 1–15 (1985)
17. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the Fifth International Conference on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
18. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming 3 (1997)
19. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: WCFLP 2005: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming, pp. 8–13. ACM Press, New York (2005)