Justifications for Logic Programming

C. V. Damásio[1] and A. Analyti[2] and G. Antoniou[3]

[1] CENTRIA, Departamento de Informática Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa, 2829-516 Caparica, Portugal (cd@fct.unl.pt)
[2] Institute of Computer Science, FORTH-ICS, Crete, Greece (analyti@ics.forth.gr)
[3] Department of Informatics, University of Huddersfield, Huddersfield, UK, and Institute of Computer Science, FORTH-ICS, Crete, Greece (G.Antoniou@hud.ac.uk)

Abstract. Understanding why and how a given answer to a query is generated from a deductive or relational database is fundamental to obtain justifications, assess trust, and detect dependencies on contradictions. Propagating provenance information is a major technique that evolved in the database literature to address the problem, using annotated relations with values from a semiring. The case of positive programs/relational algebra is well-understood but handling negation (or set difference in relational algebra) has not been addressed in its full generality or has deficiencies. The approach defined in this work provides full provenance information for logic programs under the least model, well-founded semantics and answer set semantics, and is related to the major existing notions of justifications for all these logic programming semantics.

1 Introduction

An essential problem that users of logic programming systems face is the understanding of why a given query is true or false in a model of a program, under a given particular semantics. This problem has received attention for quite a long time, and has been addressed for the case of definite programs under least model semantics in [20, 15], stratified negation in [20, 19], for the case of well-founded semantics in [16, 19, 17], and for the answer set semantics in [17]. Most of the approaches resort to the non-deterministic construction of complex structures, usually graph based in order to obtain justifications for programs [19, 15, 17], or provide algorithmic approaches [20, 2]. In this paper it is presented a fully declarative and logical approach to the problem, by constructing provenance formulae from which justifications can be extracted. Even though the problem of finding justifications is related to the debugging of logic programming theories [16, 2, 1, 5], the exact relationship will be deferred to a subsequent work.

In the current work is defined a declarative logical approach able to extract provenance information for logic programs. Using values of the Lindenbaum-Tarski algebra as annotation tags for atoms, we are able to specify why-provenance both for definite and normal logic programs under well-founded semantics, and

[1] Partially supported by FCT Project ERRO PTDC/EIACCO/121823/2010.

relate it to abduction and calculation of prime implicants. The approach is subsequently generalised to the case of answer set programming. Moreover, why-provenance for the case of well-founded semantics is novel, extending the results known for the case of relational algebra, since the case of bag semantics with positive recursion has been addressed in [10], but to the best of our knowledge no work in the literature considers the case of recursion over negation under set (or bag) semantics. We will use these provenance formulae to obtain justifications for literals true in a given model, extending the approaches of evidence graphs [15] and offline justifications [17].

The background is introduced in the next section. Section 3 specifies the why-provenance approach for the case of definite programs, covering the case of positive relational algebra, introducing also why-provenance for the case of an atom not belonging to the model. The technique is generalized afterwards to the well-founded semantics in Section 4. Why-provenance for the answer set semantics can be obtained from the provenance formulas for the well-founded model. We discuss our results and summarize the main conclusions in the last section. It is assumed that the reader is acquainted with the major semantics for logic programs: least model [21], well-founded [7] and answer set semantics [8].

2 Preliminaries and background

Normal logic programs are sets of rules. A rule r has the following syntax: $A_1 \text{ :- } A_2, \ldots, A_m, {\sim}A_{m+1}, \ldots, {\sim}A_n (n \geq m \geq 0)$, where each A_i is a logical atom without occurrence of function symbols. As usual, define $Head(r) = A_1$, $Body^+(r) = \{A_2, \ldots, A_m\}$, $Body^-(r) = \{A_{m+1}, \ldots, A_n\}$, and $Body(r) = \{A_2, \ldots, A_m, {\sim}A_{m+1}, \ldots, {\sim}A_n\}$. A program is definite (or positive) if there are no occurrences of weakly negated atoms. Without loss of generality, it is assumed that programs are ground (no variables in the rules). Given a set of literals J let ${\sim}J = \{a \mid {\sim}a \in J\} \cup \{{\sim}a \mid a \in J \wedge \forall_b\ a \neq {\sim}b\}$. The Herbrand Base $\mathbb{H}_P$ of a program P is formed by the set of atoms occurring in it. A two-valued interpretation is a subset of $\mathbb{H}_P$ specifying the true atoms, and a partial interpretation is a subset of $\mathbb{H}_P \cup {\sim}\mathbb{H}_P$. A two-valued interpretation I corresponds to the partial interpretation $I \cup {\sim}(\mathbb{H}_P \setminus I)$. The least model $least(P)$ of a definite program P is the least fixpoint of operator $T_P(I) = \{Head(r) \mid r \in P \wedge Body(r) \subseteq I\}$. The answer sets of normal logic program P are the fixpoints of $\Gamma(I) = least(P^I)$, where $P^I = \{Head(r) \leftarrow Body^+(r) \mid r \in P, Body^-(r) \cap I = \emptyset\}$. The well-founded model WFM_P of P is $T \cup {\sim}F$ where T and F are interpretations such that $T \cap F = \emptyset$, T is the least fixpoint $T = \Gamma(\Gamma(T))$ and $F = \mathbb{H}_P \setminus \Gamma(T)$.

Example 1. Consider the following logic program

$$a \text{ :- } c, {\sim}b. \qquad b \text{ :- } {\sim}a. \qquad d \text{ :- } {\sim}c, {\sim}d. \qquad c \text{ :- } {\sim}e. \qquad e \text{ :- } f. \qquad f \text{ :- } e.$$

This program has two answer sets: $M_1 = \{a, c\}$ and $M_2 = \{b, c\}$ (absent atoms are false). Its well-founded model is $WFM = \{c, {\sim}d, {\sim}e, {\sim}f\}$ (absent literals are undefined). A program may not have answer sets: by adding the fact e to the previous program, c becomes false, but then d can only be true iff it is false.

Determining why a given atom belongs to the model of a program is not a trivial task, due to the mutual dependencies that can occur in programs. This is fundamental to users to be able to debug or to check their programs. In [15] is defined the notion of evidence for tabled definite logic programs, extending the previous work in [19], where it is assumed a left-to-right execution order of goals. This approach has been generalised in [17] to the case of normal logic programs by introducing offline-justifications.

Definition 1 (Offline explanation graph). *Let P be a program, J a partial interpretation, U a set of atoms, and $b^{\pm}$ an element of $\mathbb{H}_P^+ \cup \mathbb{H}_P^-$, where $\mathbb{H}_P^+ = \{a^+ \mid a \in \mathbb{H}_P\}$ and $\mathbb{H}_P^- = \{a^- \mid a \in \mathbb{H}_P\}$. An offline explanation graph G of b with respect to J and U is a labeled, directed graph $G = (N, E)$, where $N \subseteq \mathbb{H}_P^+ \cup \mathbb{H}_P^- \cup \{assume, \top, \bot\}$ and $E \subseteq N \times N \times \{+, -\}$, satisfying:*

1. *Element $b^{\pm}$ belongs to N and every node of G is reachable from $b^{\pm}$;*
2. *The only sinks in the graph are: $assume$, $\top$, and $\bot$;*
3. *For every $h^+ \in N$ then $h \in J$, and there is $h :- b_1, \ldots, b_m, \sim c_1, \ldots, \sim c_n$ in P such that $\{b_1, \ldots, b_m\} \subseteq J$ and $\{\sim c_1, \ldots, \sim c_n\} \subseteq J \cup \sim U$.*
   Additionally, there is an arc $(h^+, b_i^+, +)$ for each $1 \leq i \leq m$, and an arc $(h^+, c_j^-, -)$ for each $1 \leq j \leq n$; if the rule has an empty body (it is a fact), then h^+ is $(h^+, \top, +) \in E$. No more arcs have source h^+.
4. *If $h^- \in N$ and $h \in U$ then $(h^-, assume, -)$ is the only arc with source h^-;*
5. *For every $h^- \in N$ such that $h \notin U$ then $\sim h \in J$, and the successors of h^- is the least set of arcs such that for every rule $h :- b_1, \ldots, b_m, \sim c_1, \ldots, \sim c_n$:*
   - *$\exists_{1 \leq i \leq m}$ such that $(h^-, b_i^-, +)$ and $\sim b_i \in J$ or $b_i \in U$, or*
   - *$\exists_{1 \leq j \leq n}$ such that $(h^-, c_j^+, -)$ and $c_j \in J$.*

   If there are no rules for h then the only arc with source h^- is $(h^-, \bot, +)$;
6. *There are no cycles in the subgraph involving nodes in $\mathbb{H}_P^+$ and containing arcs labeled only with $+$;*

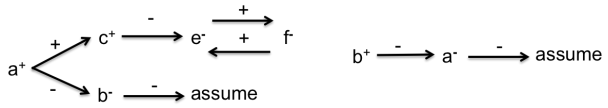

Fig. 1. Offline justification graphs for the program of Example 1

The labels of the arcs in an offline explanation graph indicate if the dependency is positive or negative. There are special nodes used to denote true facts ($\top$), and atoms without rules ($\bot$). The set of assumptions U captures the literals which are assumed false, and there is a special sink node for those ($assume$). In order to make an atom true, a rule with a true body must exist (true positive atoms and false weakly negated literals). We have a dual reason for a literal being false: all rules must have a falsified positive atom b_i (thus b_i^- must belong to the graph) or a falsified negated literal c_j (thus c_j^+ must belong to G).

An offline justification of an answer set M of a program P is an offline explanation graph for M using the set of assumptions U containing the false atoms in M that are undefined in the well-founded model WFM_P. Moreover, this graph does not have negative cycles. It is also shown in [17] that every non-undefined atom in WFM_P can be justified without any assumptions.

Example 2. The offline justification for a in the answer set $\{a, c\}$ of Example 1 can be found in the left hand side of Figure 1. So, a is true because c is true and b is assumed false. Atom c is true because e is false due to a positive mutual dependency between e and f. The offline justification for b in the answer set $\{b, c\}$ is simpler, and rests solely in the assumption that a is false. Intuitively, an offline justification graph represents a conjunction of literals true in the model supporting the conclusion plus dependency information.

Relational algebra has expressive power equivalent to acyclic datalog programs, and the translation of relational algebra queries into datalog is immediate. There is an extensive work on provenance for the case of relational algebra, that is summarised here to motivate the use of the proposed algebraic structure to represent why-provenance for logic programs, and simultaneously relate to this area of research. A general data model for annotated relations has been introduced in [10], for positive relational algebra (i.e., excluding the difference operator). These annotations can be used to check derivability of a tuple, lineage, and provenance, and perform query evaluation over incomplete/probabilistic databases. The main concept is the notion of $\mathcal{K}$-relation where tuples are annotated with values (tags) of a commutative semiring $\mathcal{K}$, while positive relational algebra operators semantics are extended and captured by corresponding compositional operations over $\mathcal{K}$. A commutative semiring is an algebraic structure $\mathcal{K} = (\mathbb{K}, \oplus, \otimes, 0, 1)$ where $(\mathbb{K}, \oplus, 0)$ is a commutative monoid ($\oplus$ is associative and commutative, to capture union of relations) with identity element 0, $(\mathbb{K}, \otimes, 1)$ is a commutative monoid with identity element 1, to encode natural join, and thus operation $\otimes$ distributes over $\oplus$, and 0 is an annihilating element of $\otimes$. The obtained algebra on $\mathcal{K}$-relations is expressive enough to capture different kinds of annotations with set or bag semantics, and it is shown that the semiring of polynomials with integer coefficients is the most general semiring.

Example 3. A photo sharing site allows users to register, and users can upload photos of users. Of course, all registered users are users. Users are represented in a relation `u(Name)`, registered users in `r(Name)`, guest users in `g(Name)`, and `p(U1,U2)` stores the information that user `U1` has uploaded a photo of user `U2`. Ann, Bob and David are registered users, Ann uploaded a photo of David, and Bob uploaded a photo of himself. Consider $\mathcal{K}$-relations $\mathtt{r} = \{a : \mathtt{t}_1, b : \mathtt{t}_2, d : \mathtt{t}_3\}$, and $\mathtt{p} = \{(a, d) : \mathtt{p}_1, (b, b) : \mathtt{p}_2\}$, where $\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{p}_1$ and $\mathtt{p}_2$ are tuple identifiers. Let $u = r \cup g$ be a view and query $\mathit{\Pi}\,[\rho_{x \leftarrow \mathtt{Name}}(u) \bowtie \rho_{x \leftarrow \mathtt{U1}, y \leftarrow \mathtt{U2}}(p) \bowtie \rho_{y \leftarrow \mathtt{Name}}(r)]$ to check if there is an user that uploaded a photo of a registered user. In the most general semiring of polynomials with integer coefficients, the provenance for the query under bag semantics is $\mathtt{p}_1 \times \mathtt{t}_1 \times \mathtt{t}_3 + \mathtt{p}_2 \times {\mathtt{t}_2}^2$, showing that a join of the tuples identified by $\mathtt{p}_1$, $\mathtt{t}_1$ and $\mathtt{t}_3$, or the join of the tuple annotated of $\mathtt{p}_2$ with $\mathtt{t}_2$ (two times) are the ways to construct a solution to the

query. Why-provenance returns the annotation $\{\{\mathtt{p}_1, \mathtt{t}_1, \mathtt{t}_3\}, \{\mathtt{p}_2, \mathtt{t}_2\}\}$, or equivalently, the boolean formula $(\mathtt{p}_1 \wedge \mathtt{t}_1 \wedge \mathtt{t}_3) \vee (\mathtt{p}_2 \wedge \mathtt{t}_2)$ showing which tuples have been used for obtaining each answer under set semantics; using lineage semiring one gets simply $\{\mathtt{p}_1, \mathtt{p}_2, \mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3\}$, i.e. the tuples supporting the query. This situation can be encoded in datalog by the set of facts (extensional part) $\{r(a), r(b), r(d), p(a,d), p(b,b)\}$ and intensional rules: $c :- u(X), p(X,Y), r(Y).$; $u(X) :- r(X).$; $u(X) :- g(X)$. Because we have facts, $r(a), r(d), p(a,d)$, or $r(b)$ and $p(b,b)$ we can conclude that c holds. This corresponds exactly to what is obtained with the why-provenance semiring of Boolean formulas (see [10]).

To be able to capture relational difference, i.e. negation, the authors in [6] assume the $\mathcal{K}$ semiring is naturally ordered (i.e. binary relation $x \preceq y$ is a partial order, where $x \preceq y$ iff there exists a $z \in \mathbb{K}$ such that $x \oplus z = y$), and require additionally that for every pair x and y there is a least z such that $x \preceq y \oplus z$, defining in this way $x \ominus y$ to be such smallest z. A $\mathcal{K}$ semiring with such monus $\ominus$ operator is designated by m-semiring. An important m-semiring is obtained from the above Boolean formulas semiring, being powerful enough to represent why-provenance for relational algebra under set semantics. The shortcomings of the database approach is that it can only handle negation over acyclic programs, and it is not able to indicate which rules have been used to derive an answer. In Example 3, c holds because the ground program contains rule $u(b) : -r(b)$ and the above mentioned facts (or similarly, for the cases of a and c).

3 Provenance for Definite Logic Programming

Provenance in Logic Programming is captured by adapting the Boolean formulas m-semiring for tackling full why-provenance information, i.e. both negative and positive. This is achieved by annotating every literal in the language by an identifier, as well as every rule with an identifier $\mathtt{r}_\mathtt{i}$, where i is the rule number in some ordering of the program. Our setting is restricted to the case of finite logic programs, and is inspired in the debugging transformation presented in [16].

Definition 2. *Given a logic program P over the Herbrand Base $\mathbb{H}_P$, let B_P be the free Boolean algebra generated by the propositional variables $\mathbb{H}_P \cup not(\mathbb{H}_P) \cup \{r_i | 1 \leq i \leq \mid P \mid\}$, i.e. the Lindenbaum-Tarski algebra of the propositional language $\mathbb{H}_P \cup not(\mathbb{H}_P) \cup \{r_i | 1 \leq i \leq \mid P \mid\}$. The elements of B_P are the equivalence classes of propositional formulas under logical equivalence. Meet ($\wedge$), join ($\vee$), and complementation ($'$) are defined by: $[\phi] \vee [\psi] = [\phi \vee \psi]$, $[\phi] \wedge [\psi] = [\phi \wedge \psi]$, $[\phi]' = [\neg\phi]$. The bottom element is $0 = [\phi \wedge \neg\phi] = [\mathbf{f}]$, and the top element $1 = [\phi \vee \neg\phi] = [\mathbf{t}]$. The partial ordering of B_P is entailment: $[\phi] \preceq [\psi]$ iff $\phi \models \psi$. The $\mathcal{K}_{WhyNot}$ provenance m-semiring is obtained from B_P by letting $[\phi] \oplus [\psi] = [\phi] \vee [\psi] = [\phi \vee \psi]$, $[\phi] \otimes [\psi] = [\phi] \wedge [\psi] = [\phi \wedge \psi]$ and $[\phi] - [\psi] = [\phi \wedge \neg\psi]$.*

Notice that in the above definition propositional variables of the form $\mathtt{at}$ and $\mathtt{not(at)}$ are introduced for every atom at in the Herbrand Base. There is no relationship between $\mathtt{at}$ and $\mathtt{not(at)}$ since they are different propositional symbols; thus $[\neg\mathtt{not(at)}]$ and $[\mathtt{at}]$ are distinct elements of $\mathcal{K}_{WhyNot}$. An identifier

is also introduced for each rule in the program. A provenance formula is simply an arbitrary Boolean formula over the atoms, default negation of atoms and rule identifiers. We could use as annotation of literals any set of identifiers just taking care that identifiers are in one-to-one correspondence with $\mathbb{H}_P \cup not(\mathbb{H}_P)$.

In order to extract provenance information for definite logic programs, we introduce why-not provenance programs:

Definition 3. *A why-not provenance program is a finite set of why-not provenance rules of the form* $A \Leftarrow [J] \otimes B_1 \otimes \ldots \otimes B_m$ *where* A, $B_1, \ldots, B_m$ $(m \geq 0)$ *are ground atoms, and* $[J]$ *is an element of* $\mathcal{K}_{WhyNot}$ *.*

Why-provenance programs have rules which are monotonic, and thus the standard results of multivalued logic programming apply [4], namely the existence of a least model that can be obtained by iterating a modified T_P operator starting from the interpretation that maps every atom to 0.

Definition 4. *Consider a logic program* P *and a why-not provenance program* $\mathfrak{P}$ *over* $\mathbb{H}_P$*. An interpretation* I *for why-not provenance program* $\mathfrak{P}$ *is a mapping* $I : \mathbb{H}_P \rightarrow B_P$*. The set of all interpretations is a lattice with pointwise ordering: given any interpretations* I_1 *and* I_2 *we say that* $I_1 \preceq I_2$ *iff for every* $A \in \mathbb{H}_P$ *it is the case that* $I_1(A) \preceq I_2(A)$*, i.e.* $I_1(A) \models I_2(A)$*.*

An interpretation I *satisfies a rule* $A \Leftarrow [J] \otimes B_1 \otimes \ldots \otimes B_m$ *of why-not program* $\mathfrak{P}$ *iff* $I(A) \succeq [J] \otimes I(B_1) \otimes \ldots \otimes I(B_m)$ *iff* $J \wedge I(B_1) \wedge \ldots \wedge I(B_m) \models I(A)$*. Intepretation* I *is a model of* $\mathfrak{P}$ *iff* I *satisfies all the rules of* $\mathfrak{P}$*.*

Lemma 1. *Consider a why-not provenance program* $\mathfrak{P}$*. Program* $\mathfrak{P}$ *has a least model* $M_{\mathfrak{P}}$ *which can be obtained by iterating through the following operator starting from the least interpretation* I_0*, which maps every atom to* 0*:*

$$\begin{aligned} T_{\mathfrak{P}}(I)(A) &= \bigoplus \{J \wedge I(B_1) \wedge \ldots \wedge I(B_m) \mid A \Leftarrow [J] \otimes B_1 \otimes \ldots \otimes B_m \in \mathfrak{P}\} \\ &= \left[\bigvee_{A \Leftarrow [J] \otimes B_1 \otimes \ldots \otimes B_m \in \mathfrak{P}} J \wedge I(B_1) \wedge \ldots \wedge I(B_m)\right] \end{aligned}$$

Determining why-not provenance for logic programs rests in the following techniques. First, every non-factual rule is annotated with the formula $[\mathtt{r_i}]$. Second, for every atom A if there is a fact for it in the program, a rule $A \Leftarrow [\mathtt{A}]$ will be introduced in the why-provenance program, otherwise the rule $A \Leftarrow [\neg\mathtt{not(A)}]$ is added. We could have used a rule $A \Leftarrow [\mathtt{r_i}]$ for annotating facts, but the outcome intepretation would be less readable.

Definition 5. *Let* P *be a definite logic program and* $\mathfrak{P}(P)$ *the why-not provenance program constructed as follows:*

- *For the* i^{th} *rule* $A :- B_1, \ldots, B_m$ *that is not a fact in* P *(i.e.,* $m \geq 1$*) add the why-not provenance rule* $A \Leftarrow [r_i] \otimes B_1 \otimes \ldots \otimes B_m$ *to* $\mathfrak{P}(P)$*;*
- *For every atom* A *in* $\mathbb{H}_P$ *if there is a fact* A *in* P *then add* $A \Leftarrow [\mathtt{A}]$ *to* $\mathfrak{P}(P)$*, otherwise add* $A \Leftarrow [\neg\mathtt{not(A)}]$ *to* $\mathfrak{P}(P)$*.*

The why-not provenance information $Why(A)$ *for an atom* A *is given by* $Why_P(A) = M_{\mathfrak{P}(P)}(A)$*, and for literal* $\sim A$ *is given by* $Why_P(\sim A) = [\neg M_{\mathfrak{P}(P)}(A)]$*.*

The rationale is: if there is a fact A and possibly rules $r_i, \ldots, r_j$ for it, the why-provenance formula for A has the shape $[(\mathtt{r_i} \wedge Why_i) \vee \ldots \vee (\mathtt{r_j} \wedge Why_j) \vee \mathtt{A}]$; otherwise, if there is no fact for A it has the form $[(\mathtt{r_i} \wedge Why_i) \vee \ldots \vee (\mathtt{r_j} \wedge Why_j) \vee \neg\mathtt{not(A)}]$. In the former, one justification for A being true is $[\mathtt{A}]$ meaning that there is a fact for A, other justifications are obtained by using a given rule r_k and justifying why the body is true. The latter case is better understood if we look at the justification for $\sim A$ having why-provenance formula $[\neg(\mathtt{r_i} \wedge Why_i) \wedge \ldots \wedge \neg(\mathtt{r_j} \wedge Why_j) \wedge \mathtt{not(A)}]$, expressing that all bodies must be falsified and that $[\mathtt{not(A)}]$ holds (there is no fact for A).

Definition 6. *Let P be a logic program, and C be a conjunction of literals of $\mathcal{K}_{WhyNot}$. Define the following sets of facts and rules of program P:*

$KeepFacts(C) = \{A. \mid \mathtt{A} \in C\}$ $\quad$ $RemoveFacts(C) = \{A. \mid \neg\mathtt{A} \in C\}$
$MissingFacts(C) = \{A. \mid \neg\mathtt{not(A)} \in C\}$ $\quad$ $NoFacts(C) = \{A. \mid \mathtt{not(A)} \in C\}$
$KeepRules(C) = \{A :- Body \mid \mathtt{r_i} \in C$ *and* $A :- Body$ *is the* i^{th} *rule of* $P\}$
$RemoveRules(C) = \{A :- Body \mid \neg\mathtt{r_i} \in C$ *and* $A :- Body$ *is the* i^{th} *rule of* $P\}$

The first major result relates why-not provenance information with changes to the original program:

Theorem 1. *Let P be a definite logic program, A an arbitrary atom, G a set of facts not in P, F a subset of facts of P and R a subset of rules of P. Then:*

- *Atom A belongs to the least model of $P \setminus (F \cup R) \cup G$ iff there is a conjunction $C = \mathtt{A_1} \wedge \ldots \wedge \mathtt{A_m} \wedge \mathtt{r_{i_1}} \wedge \ldots \wedge \mathtt{r_{i_k}} \wedge \neg\mathtt{notQ_1} \wedge \ldots \wedge \neg\mathtt{notQ_n}$ such that $C \models Why_P(A)$, $MissingFacts(C) \subseteq G$, $KeepFacts(C) \cap F = \emptyset$ and $KeepRules(C) \cap R = \emptyset$. A conjunction of this form is said to be a truth-support for A.*
- *Atom A does not belong to the least model of $P \setminus (F \cup R) \cup G$ iff there is a conjunction $C = \neg\mathtt{A_1} \wedge \ldots \wedge \neg\mathtt{A_m} \wedge \neg\mathtt{r_{i_1}} \wedge \ldots \neg\mathtt{r_{i_k}} \wedge \mathtt{notQ_1} \wedge \ldots \wedge \mathtt{notQ_n}$ such that $C \models Why_P(\sim A)$, $RemoveFacts(C) \subseteq F$, $RemoveRules(C) \subseteq R$ and $NoFacts(C) \cap G = \emptyset$. Conjunction C is said to be a falsity-support for A.*

By letting $F = R = G = \{\}$ no changes to the program are permitted, and thus justifications for the literals true in the least model of P are:

Corollary 1. *Let P be a definite logic program and M its least model. Then:*

- *An atom A belongs to M (A is true) iff there is a conjunction of propositional variables $C = \mathtt{A_1} \wedge \ldots \wedge \mathtt{A_m} \wedge \mathtt{r_{i_1}} \wedge \ldots \wedge \mathtt{r_{i_k}}$ such that $C \models Why_P(A)$;*
- *An atom A does not belong to M (A is false) iff there is a conjunction of propositional variables $C = \mathtt{notQ_1} \wedge \ldots \wedge \mathtt{notQ_n}$ such that $C \models Why_P(\sim A)$.*

The results in the above corollary are very interesting, meaning that the minimal justification for literals in the least model of a logic program are the prime implicants[4] containing no negations of the corresponding why-not provenance formula. The fundamental use of prime implicates/implicants as justifications goes back to Assumption-Truth Maintenance Systems [18], and has been recently used to capture the notion of causality in databases [14].

[4] A prime implicant of F is a minimal conjunction of literals C such that $C \models F$.

Example 4. Consider the logic program P, where rules are numbered:

(1) $a :- b.\quad a.$ (2) $b :- a.$ (3) $c :- b, d.$ (4) $c :- e, f.\quad d.$ (5) $e :- f.\quad f.$

All atoms hold in the least model of P. Why-not provenance program $\mathfrak{P}(P)$ is:

$a \Leftarrow [\mathtt{r}_1] \otimes b.\quad b \Leftarrow [\mathtt{r}_2] \otimes a.\quad c \Leftarrow [\mathtt{r}_3] \otimes b \otimes d.\quad d \Leftarrow [\mathtt{d}].\quad e \Leftarrow [\mathtt{r}_5] \otimes f$
$a \Leftarrow [\mathtt{a}].\quad b \Leftarrow [\neg\mathtt{not(b)}].\quad c \Leftarrow [\mathtt{r}_4] \otimes e \otimes f.\quad e \Leftarrow [\neg\mathtt{not(e)}].$
$c \Leftarrow [\neg\mathtt{not(c)}].\quad f \Leftarrow [\mathtt{f}].$

$$
\begin{array}{ll}
Why(a) = [(\mathtt{r}_1 \wedge \neg\mathtt{not(b)}) \vee \mathtt{a}] & Why(\sim a) = [\neg((\mathtt{r}_1 \wedge \neg\mathtt{not(b)}) \vee \mathtt{a})] \\
 & \quad = [(\neg\mathtt{a} \wedge \neg\mathtt{r}_1) \vee (\neg\mathtt{a} \wedge \mathtt{not(b)})] \\
Why(b) = [(\mathtt{r}_2 \wedge \mathtt{a}) \vee \neg\mathtt{not(b)}] & Why(\sim b) = [\neg((\mathtt{r}_2 \wedge \mathtt{a}) \vee \neg\mathtt{not(b)})] \\
 & \quad = [(\neg\mathtt{r}_2 \wedge \mathtt{not(b)}) \vee (\neg\mathtt{a} \wedge \mathtt{not(b)})] \\
Why(d) = [\mathtt{d}] & Why(\sim d) = [\neg\mathtt{d}] \\
Why(e) = [(\mathtt{r}_5 \wedge \mathtt{f}) \vee \neg\mathtt{not(e)}] & Why(\sim e) = [(\neg\mathtt{f} \wedge \mathtt{not(e)}) \vee (\neg\mathtt{r}_5 \wedge \mathtt{not(e)})] \\
Why(f) = [\mathtt{f}] & Why(\sim f) = [\neg\mathtt{f}]
\end{array}
$$

$Why(c) = [(\mathtt{r}_3 \wedge ((\mathtt{r}_2 \wedge \mathtt{a}) \vee \neg\mathtt{not(b)}) \wedge \mathtt{d}) \vee (\mathtt{r}_4 \wedge ((\mathtt{r}_5 \wedge \mathtt{f}) \vee \neg\mathtt{not(e)}) \wedge \mathtt{f}) \vee \neg\mathtt{not(c)}]$
$= [(\mathtt{r}_2 \wedge \mathtt{r}_3 \wedge \mathtt{a} \wedge \mathtt{d}) \vee (\mathtt{r}_3 \wedge \mathtt{d} \wedge \neg\mathtt{not(b)}) \vee (\mathtt{r}_4 \wedge \mathtt{r}_5 \wedge \mathtt{f}) \vee (\mathtt{r}_4 \wedge \mathtt{f} \wedge \neg\mathtt{not(e)}) \vee \neg\mathtt{not(c)}]$

We can conclude that d and f are true, because they are stated as facts, and thus to make them false is required their removal. Regarding e, it can be concluded that e is true because of rule 5 and that the fact f is true; anyway, we can make it true by adding the fact e. In order to make e false, it cannot be added as a fact, and fact f or rule r_5 should be removed (or both). Atom a is true because there is a fact for it, or if the rule r_1 is kept and a fact for b is added (a may be removed). To make a false we always have to remove the fact for a, and additionally remove rule r_1 or not introduce a fact for b. The situation for c is rather complex, but it holds because rules r_2 and r_3 are present, as well as facts a and d, or because rules r_4 and r_5 are in the program as well as the fact for f. There are many ways of making c false, for instance by removing facts d and f.

Example 5. Consider the following definite logic program where all atoms are false in the least model: $\{(1)\ a :- b.\quad (2)\ b :- a, c.\quad (3)\ b :- d.\}$. It can be checked that $\mathtt{not(a)} \wedge \mathtt{not(b)} \wedge \mathtt{not(d)} \models Why(\sim a), \mathtt{not(a)} \wedge \mathtt{not(b)} \wedge \mathtt{not(d)} \models Why(\sim b)$ and $\mathtt{not(b)} \wedge \mathtt{not(c)} \wedge \mathtt{not(d)} \models Why(\sim b)$. If we do not allow changes to the program, this provenance information is interpreted as follows: a is false because we do not have a fact for a, b and d; while b is false because we do not have a fact for b, d and a or c. Thus, one needs to add a, b, or d to make a true, while it is required to add b, d, or a and c to turn b true. This can be confirmed from $Why(a) = [(\mathtt{r}_1 \wedge \mathtt{r}_3 \wedge \neg\mathtt{not(d)}) \vee (\mathtt{r}_1 \wedge \neg\mathtt{not(b)} \vee \neg\mathtt{not(a)})]$ and $Why(b) = [(\mathtt{r}_2 \wedge \neg\mathtt{not(a)} \wedge \neg\mathtt{not(c)}) \vee (\mathtt{r}_3 \wedge \neg\mathtt{not(d)} \vee \neg\mathtt{not(b)})]$.

The relationship to evidence graphs (see [15]) is stated in the next theorem:

Theorem 2. *Consider a definite logic program P. For every evidence graph for A (resp. $\sim A$) it is possible to construct a truth-support (resp. falsity-support) formula C such that $C \models Why_P(A)$ (resp. $C \models Why_P(\sim A)$). For every, prime implicant truth-support $C = A_1 \wedge \ldots \wedge A_m \wedge r_{i_1} \wedge \ldots \wedge r_{i_k}$ of $Why_P(A)$ it is possible to construct an evidence graph for A.*

Our approach allows us to capture all the evidence according to [15], but this work lacks some of our justifications mostly for the case of false atoms. This is expected since evidence graphs are constructed for the negative case just by looking at the first false atom in the body of rules, ignoring possibly other false atoms. The comparison to offline justifications is deferred to the next since our results will cover both definite and normal logic programs, generalizing Th. 2.

4 Provenance for Well-founded Semantics

By mimicking the iteration of Γ^2 operator, we can obtain the provenance information for logic programs under well-founded semantics by defining a corresponding Gelfond-Lifschitz like operator. The correctness and extra motivation for this approach can be found in [3].

Definition 7. *Let P be a logic program and I a why-not provenance interpretation. Construct provenance program $\frac{\mathfrak{P}}{I}$ as follows:*

- *For the i^{th} rule $A :- B_1, \ldots, B_m, \sim C_1, \ldots, \sim C_n$ $(m + n \geq 1)$ in P add provenance rule $A \Leftarrow [r_i \wedge \neg I(C_1) \wedge \ldots \neg I(C_n)] \otimes B_1 \otimes \ldots \otimes B_m$ to $\frac{\mathfrak{P}}{I}$;*
- *For every atom A in $\mathbb{H}_P$ if there is a fact A in P then add $A \Leftarrow [\mathtt{A}]$ to $\frac{\mathfrak{P}}{I}$, otherwise add $A \Leftarrow [\neg\mathtt{not(A)}]$ to $\frac{\mathfrak{P}}{I}$.*

Operator $\mathfrak{G}_P(I) = M_{\frac{\mathfrak{P}}{I}}$ returns the least model of why-not program $\frac{\mathfrak{P}}{I}$.

It is clear that the operator $\mathfrak{G}_P$ is anti-monotonic, and therefore $\mathfrak{G}_P^2$ is monotonic having a least model $\mathfrak{T}_P$, corresponding to provenance information for what is true in the well-founded model, while $\mathfrak{TU}_P = \mathfrak{G}_P(\mathfrak{T}_P)$ contains the why-not provenance of what is true or undefined in the well-founded model of P.

Definition 8. *Let P be a normal logic program, and $\mathfrak{T}_P$ the least model of $\mathfrak{G}_P^2$, and $\mathfrak{TU}_P = \mathfrak{G}_P(\mathfrak{T}_P)$, and A an atom. The why-not provenance information under the well-founded semantics is defined as follows: $Why_P(A) = [\mathfrak{T}_P(A)]$; $Why_P(\sim A) = [\neg\mathfrak{TU}_P(A)]$; and $Why_P(\mathtt{undef}\ A) = [\neg\mathfrak{T}_P(A) \wedge \mathfrak{TU}_P(A)]$.*

As usual, for the case of stratified programs (no cycles through negations) we obtain a model on which for every atom A we have $Why_P(A) = [\mathfrak{T}_P(A)] = [\mathfrak{TU}_P(A)] = [\neg Why_P(\sim A)]$, and thus $Why_P(\mathtt{undef}\ A) = 0$. The why-not provenance for undefined literals is obtained from the why-not provenance for truth or undefinedness minus the why-not provenance of truth.

Theorem 3. *Let P be a normal logic program, G a set of facts not in P, F a subset of facts of P, and R a subset of rules of P. A literal L belongs to the WFM of $(P \setminus (F \cup R)) \cup G$ iff there is a conjunction of literals $C \models Why_P(L)$, such that $RemoveFacts(C) \subseteq F$, $KeepFacts(C) \cap F = \emptyset$, $RemoveRules(C) \subseteq R$, $KeepRules(C) \cap R = \emptyset$, $MissingFacts(C) \subseteq G$, and $NoFacts(C) \cap G = \emptyset$.*

The above result, generalizing Th. 2, is a fundamental new contribution to the literature of provenance in logic programming, and in particular for datalog. Since any relational algebra query can be translated into an acyclic logic program, thus it is possible to extract for the first time complete provenance information for a more expressive extension of full relational algebra.

Example 6. Consider the logic program $P = \{(1)\ a :- \sim a, b.\quad (2)\ b :- \sim c.\}$. In the well-founded model of P we have a undefined, b true, and c false. Thus:

$$\begin{array}{ll} \mathfrak{T}_P(a) = [\neg \mathtt{not(a)}] & \mathfrak{TU}_P(a) = [(\mathtt{r_1} \wedge \mathtt{r_2} \wedge \mathtt{not(c)}) \vee (\mathtt{r_1} \wedge \neg\mathtt{not(b)} \vee \neg\mathtt{not(a)})] \\ \mathfrak{T}_P(b) = [(\mathtt{r_2} \wedge \mathtt{not(c)}) \vee \neg\mathtt{not(b)}] & \mathfrak{TU}_P(b) = [(\mathtt{r_2} \wedge \mathtt{not(c)}) \vee \neg\mathtt{not(b)}] \\ \mathfrak{T}_P(c) = [\neg\mathtt{not(c)}] & \mathfrak{TU}_P(c) = [\neg\mathtt{not(c)}] \end{array}$$

The why-not provenance information for negated literals is:

$$\begin{array}{l} Why_P(\sim a) = [(\mathtt{not(a)} \wedge \mathtt{not(b)} \wedge \neg\mathtt{not(c)}) \vee (\neg\mathtt{r_2} \wedge \mathtt{not(a)} \wedge \mathtt{not(b)}) \vee (\neg\mathtt{r_1} \wedge \mathtt{not(a)})] \\ Why_P(\sim b) = [(\mathtt{not(b)} \wedge \neg\mathtt{not(c)}) \vee (\neg\mathtt{r_2} \wedge \mathtt{not(b)})] \\ Why_P(\sim c) = [\mathtt{not(c)}] \end{array}$$

Thus, $Why_P(\mathtt{undef}\ a) = [(\mathtt{r_1} \wedge \mathtt{r_2} \wedge \mathtt{not(a)} \wedge \mathtt{not(c)}) \vee (\mathtt{r_1} \wedge \mathtt{not(a)} \wedge \neg\mathtt{not(b)})]$, $Why_P(\mathtt{undef}\ c) = 0$ and $Why_P(\mathtt{undef}\ b) = 0$. The interpretation of these results is now clear. Atom a is undefined since there is no positive prime implicant both for $Why_P(a)$ and $Why_P(\sim a)$; the justification for a being undefined is that there is no fact for a and for c and both rules are in the program, or if we keep only rule r_1 then a fact for b must be added, as can be extracted from $Why_P(\mathtt{undef}\ a)$. Moreover, in order to make a true we need to add fact a, while to make it false one solution is to make c true or remove the rule for b in order to make b false, and not adding facts for a and b; alternatively, we remove the rule for a and do not add a fact for it. There is no way of making b and c undefined.

Theorem 4. *Let $G = (N, E)$ be an offline justification for a literal L true in the well-founded model M of a program P, with respect to M and empty set of assumptions. Let $C = \bigwedge_{(\mathtt{h}^+,\top,+)\in\mathtt{E}} \mathtt{h} \wedge \bigwedge_{\mathtt{h}^-\in\mathtt{N}} \mathtt{not(h)} \wedge \bigwedge_{\mathtt{h}^+\in\mathtt{N}} \mathtt{r_{i_h}}$, where r_{i_h} is the identifier of a rule for h satisfied by G, then $C \models Why_P(L)$.*

The converse direction does not hold, since we have more justifications for a literal being true. An important particular example is a program containing rules $a :- \sim b$ and $a :- b$. We have $Why_P(A) = [(\mathtt{r_1} \wedge \neg\mathtt{not(b)}) \vee (\mathtt{r_2} \wedge \mathtt{not(b)})]$, but $\mathtt{r_1} \wedge \mathtt{r_2} \models Why_P(A)$, thus if both rules are kept, a holds independently of any changes to the program. This cannot be obtained from offline-justifications.

5 Provenance for Answer Set Semantics

The extension of our approach to answer set programming is now straightforward due to the previous results.

Definition 9. *Let P be a logic program, and L a literal. The answer set why-not provenance for L is $AnsWhy_P(L) = Why_P(L) \wedge \bigwedge_{A\in H_P} \neg Why_P(\mathtt{undef} A)$.*

We combine the provenance of the literal obtained under well-founded semantics, and impose that all literals cannot be undefined. Note that this contrasts with traditional approaches where the justification is *local* in the sense that only a subset of the dependency graph is included in the justification. This is required since a literal can occur in the body of a constraint, or the program may be inconsistent because of odd-loops, and is formally supported by the results in [11] showing that there is no modular transformation of answer set semantics into propositional theories (this requirement is illustrated in a subsequent example).

Theorem 5. *Let P be a program, M an answer set of P, and L a literal true in M. Then there is a conjunction $C \models AnsWhy_P(L)$ that does not contain any negative literals (obtained as in Th. 4) for literals true in the WFM_P . For every atom $\sim A \in M$ that is undefined in WFM_P, C includes $\mathtt{not(A)} \wedge \neg \mathtt{r}_{i_1} \wedge \ldots \wedge \neg \mathtt{r}_{i_k}$ where $\{r_{i_1}, \ldots, r_{i_k}\}$ is the set of identifiers of all rules for A.*

The above theorem follows from the result in [17] stating that there is an offline justification with respect to M and the set of assumptions containing the literals false in M that are undefined in the well-founded model of P. Moreover, this justification does not have cycles. Therefore, by representing these assumptions as a conjunction of literals, we can then construct such a C. In order to assume a literal false we cannot add a fact for it ($\mathtt{not(A)}$), and must remove all existing rules for it in the program ($\neg \mathtt{r}_{i_1} \wedge \ldots \wedge \neg \mathtt{r}_{i_k}$).

Example 7. Consider the simplified version of the program in Example 1.

$$(1)\ a :- c, \sim b. \qquad (2)\ b :- \sim a. \qquad (3)\ d :- \sim c, \sim d. \qquad c.$$

It has two stable models $\{a, c\}$ and $\{b, c\}$, and its why-not provenance is:

$$\begin{aligned}
AnsWhy(a) &= [(\mathtt{c} \wedge \neg\mathtt{not(a)}) \vee (\mathtt{r}_1 \wedge \neg \mathtt{r}_2 \wedge \mathtt{c} \wedge \mathtt{not(b)}) \vee (\neg\mathtt{not(a)} \wedge \neg\mathtt{not(d)})] \\
AnsWhy(\sim a) &= [\mathtt{not(a)} \wedge (\neg \mathtt{r}_1 \vee (\neg \mathtt{c} \wedge \neg\mathtt{not(d)}) \vee (\mathtt{c} \wedge \neg\mathtt{not(b)}) \vee (\neg\mathtt{not(b)} \wedge \neg\mathtt{not(d)}))] \\
AnsWhy(b) &= [\mathtt{r}_2 \wedge \mathtt{not(a)} \wedge (\neg \mathtt{r}_1 \vee \neg \mathtt{c} \vee \neg\mathtt{not(d)}) \wedge \neg(\mathtt{r}_1 \wedge \neg \mathtt{c} \wedge \mathtt{not(d)})] \\
AnsWhy(\sim b) &= [\mathtt{not(b)} \wedge (\neg \mathtt{r}_2 \vee \neg\mathtt{not(a)}) \wedge \neg(\mathtt{r}_1 \wedge \neg \mathtt{c} \wedge \mathtt{not(d)})] \\
AnsWhy(c) &= [(\mathtt{c} \wedge \neg(\mathtt{r}_1 \wedge \mathtt{r}_2 \wedge \mathtt{not(a)} \wedge \mathtt{not(b)})] \\
AnsWhy(\sim c) &= [(\neg \mathtt{c} \wedge \neg\mathtt{not(d)}) \vee (\neg \mathtt{c} \wedge \neg \mathtt{r}_1)] \\
AnsWhy(d) &= [(\neg \mathtt{c} \wedge \neg\mathtt{not(d)}) \vee (\neg\mathtt{not(d)} \wedge \neg(\mathtt{r}_1 \wedge \mathtt{r}_2 \wedge \mathtt{not(a)} \wedge \mathtt{not(b)}))] \\
AnsWhy(\sim d) &= [(((\mathtt{c} \wedge \mathtt{not(d)}) \vee (\neg \mathtt{r}_1 \wedge \mathtt{not(d)})) \wedge \neg(\mathtt{r}_1 \wedge \mathtt{r}_2 \wedge \mathtt{not(a)} \wedge \mathtt{not(b)})]
\end{aligned}$$

The results are expected and intuitive. Regarding c, it holds because we have a fact for it. The formula $\neg(\mathtt{r}_1 \wedge \mathtt{r}_2 \wedge \mathtt{not(a)} \wedge \mathtt{not(b)})$ guarantees that a and b are not undefined: we have to remove at least one of the first two rules, or to add the fact a or b. More interestingly is $AnsWhy(\sim c)$: it expresses that to make c false, it is necessary of course to remove the fact for c, and remove the rule for d or introduce a fact for d, otherwise an odd-loop would appear. By making c false, a becomes false and b true, and thus it is no longer required to guarantee that they are not undefined. The justifications for d and $\sim d$ are similar. The truth of a in the first model is justified by the fact c, and by assuming b to be false as captured by the conjunction $\mathtt{r}_1 \wedge \neg \mathtt{r}_2 \wedge \mathtt{c} \wedge \mathtt{not(b)}$; if c is kept then a can

be added to make it true. If c is removed, then a can be made true by adding it, as well as the fact for d in order to avoid inconsistency. Regarding the truth of b in the second answer set, it is required to assume a false as encoded by $\texttt{not}(\texttt{a}) \wedge \neg \texttt{r}_1$ in the conjunction $\texttt{r}_2 \wedge \texttt{not}(\texttt{a}) \wedge \neg \texttt{r}_1$, and corresponds exactly to the offline justification for b in Fig. 1.

6 Discussion and Conclusions

The applications of why-provenance for logic programs are manifold. The first, and our main motivation, is to understand how a literal is derived from the program. This extends to the case of queries with arbitrary relational algebra operators. Other important application, is the study of why-provenance in the Semantic Web: our approach can detect monotonic and non-monotonic dependencies in SPARQL queries over arbitrary graphs.

It is defined for the first-time a complete provenance model for the major semantics of logic programs. However, there are some important questions remaining to be addressed. In particular, the size of the formulas generated is not guaranteed to be polynomial on the size of the program. This is not a surprise since for some programs it may be required an exponential number of propositional formulas to capture the answer set semantics [12]. However, there are also known polynomial encodings by introducing extra symbols in the translation [13, 11], and for the case of programs without positive loops we can construct a polynomial formula and thus obtaining a polynomial representation for full relational algebra. Additionally, we need only a linear number of iterations of the immediate consequences operator to reach the fixpoint in the case of definite programs, and a polynomial time number of iterations for the case of the well-founded semantics. Our approach contrasts with evidence graphs [15] and offline-justifications [17] by being more informative. In fact, our provenance formulas are a declarative representation of all such justifications, which can be exponential in number since the number of prime implicants is exponential in the size of the formula, and the problem of generating them is co-NP-hard. Additionally, the complexity of conjunctive query answering over K-relations most of the times is NP-complete (see [9]).

Why-not provenance formulas resort to a program transformation previously defined for declarative debugging of logic programs [16]. The debugging of Answer Set Programs has been addressed in the literature by several authors, and the most effective approaches resort to meta-transformations to address the several forms of anomalies that can be found in programs [1, 5]. In fact, our approach is capable of providing the corrections (adding or removing facts, and removing rules) in order to obtain what is desired. The approaches presented in [1, 5] are more fine grained, and are designed to detect errors in programs. Nevertheless, we conjecture the approaches to be related and this is left for future work. For instance, justifications for violation of integrity constraints can be obtained by putting the head $false$ in all ICs, and determine why-not justification for $false$. In general, a direction to explore, consists of translating why-provenance into

ASP, apply the debugging transformations and extract the corresponding propositional theories. The generalization to the first-order case is an open research issue, but first-order abduction or constructive negation may be necessary.

References

1. M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. Debugging ASP programs by means of ASP. In *LPNMR'07*, pages 31–43. Springer, 2007.
2. M. Brain and M. D. Vos. Debugging logic programs under the answer set semantics. In *Proc. of ASP'05 Workshop*, volume 142 of *CEUR Workshop Proceedings*, 2005.
3. C. V. Damásio and L. M. Pereira. Antitonic logic programs. In *Proc. of LPNMR'01*, volume 2173 of *LNCS*, pages 379–392. Springer, Sept. 2001.
4. C. V. Damásio and L. M. Pereira. Monotonic and residuated logic programs. In *Proc. of ECSQARU'0*, volume 2143 of *LNCS*, pages 748–759. Springer, Sept. 2001.
5. M. Gebser, J. Pührer, T. Schaub, and H. Tompits. A meta-programming technique for debugging answer-set programs. In *AAAI'08*, pages 448–453. AAAI Press, 2008.
6. F. Geerts and A. Poggi. On database query languages for K-relations. *J. Applied Logic*, 8(2):173–185, 2010.
7. A. V. Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080. MIT Press, 1988.
9. T. J. Green. Containment of conjunctive queries on annotated relations. In *Proc. of Database Theory - ICDT 2009*, volume 361, pages 296–309, 2009.
10. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. of PODS '07*, pages 31–40, New York, NY, USA, 2007. ACM.
11. T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.
12. V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Trans. Comput. Logic*, 7(2):261–268, Apr. 2006.
13. F. Lin and J. Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proc. of IJCAI'03*, pages 853–858. Morgan Kaufmann Publishers Inc., 2003.
14. A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 33(3):59–67, 2010.
15. G. Pemmasani, H.-F. Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *FLOPS*, pages 24–38, 2004.
16. L. M. Pereira, C. V. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In *Proc. of LPNMR'93*, pages 316–330, 1993.
17. E. Pontelli, T. C. Son, and O. El-Khatib. Justifications for logic programs under answer set semantics. *TPLP*, 9(1):1–56, 2009.
18. R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proc. of AAAI'87*, pages 183–189, 1987.
19. A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Proc. of PPDP*, pages 178–189, 2000.
20. G. Specht. Generating explanation trees even for negations in deductive database systems. In *Proc. of LPE*, pages 8–13, 1993.
21. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.