A Simple Top-Down Query Answering Procedure for Many-Valued Logic Programming

Umberto Straccia
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa ITALY
April 7, 2006

Abstract
We present a simple, yet general top-down query answering procedure for many-valued logic programming, which allow to deal with imprecision and some forms of uncertainty. The main features of the logic are: (i) the truth values are taken from a complete lattice; (ii) computable functions may appear in the rule bodies to manipulate truth values. To answer queries, we provide a novel and simple tabling-like top-down procedure.

Keywords: deductive databases, logic programming, many-valued logic, imprecision, uncertainty

Category: F.4.1: Mathematical Logic and Formal Languages: Mathematical Logic: [Logic and constraint programming]

Category: I.2.3: Artificial Intelligence: Deduction and Theorem Proving: [Logic programming]

Terms: Theory
1 Introduction

The management of uncertainty and/or imprecision within deduction systems is an important issue whenever the real world information to be represented is of imperfect nature. In logic programming (and deductive databases, in particular), the problem has attracted the attention of many researchers and numerous frameworks have been proposed. Essentially, they differ in the underlying notion of uncertainty theory and imprecision theory and how uncertainty/implication values, associated to rules and facts, are managed. Below a list of references and the underlying imprecision and uncertainty theory:

**Probability theory:** [8, 4, 5, 17, 29, 31, 27, 28, 30, 48, 56, 57, 68, 66, 81, 82, 83, 84, 85, 86, 93, 100, 104, 105, 106, 107, 110, 125, 131];

**Fuzzy set theory:** [92, 6, 7, 12, 14, 42, 54, 55, 62, 51, 50, 92, 102, 101, 109, 111, 116, 117, 118, 87, 124, 127, 126, 128, 129, 132];

**Multi-valued logic:** [13, 18, 19, 20, 25, 21, 22, 23, 24, 33, 32, 36, 34, 35, 43, 44, 45, 46, 47, 52, 58, 59, 60, 61, 64, 65, 67, 69, 72, 73, 74, 75, 76, 77, 79, 80, 88, 90, 89, 91, 94, 95, 98, 96, 97, 99, 112, 113, 114, 115, 120, 119, 121, 122];

**Possibilistic logic:** [2, 3, 1, 15, 40, 108].

We recall that under uncertainty theory fall all those approaches in which statements rather than being either true or false, are true or false to some probability or possibility/necessity, while under imprecision theory fall all those approaches in which statements are true to some degree which is taken from a truth space (see [41] for a clarification between the notions of uncertainty and imprecision).

In this work we deal with imprecision and, thus, statements have a degree of truth. However, as we will see later on, in some cases we can simulate also some forms of uncertainty (e.g. possibilistic logic programs and probabilistic logic programs under the event independence assumption).

Current frameworks for managing imprecision in logic programming can roughly be classified into annotation based (AB) and implication based (IB).

In the AB approach (e.g. [60, 61, 103, 104]), a rule is of the form

\[ A : f(\beta_1, \ldots, \beta_n) \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n \]
which asserts “the value of atom $A$ is at least (or is in) $f(\beta_1, \ldots, \beta_n)$, whenever the value of atom $B_i$ is at least (or is in) $\beta_i$, $1 \leq i \leq n$”. Here $f$ is an $n$-ary computable function and $\beta_i$ is either a constant or a variable ranging over an appropriate truth domain.

In the IB approach, (e.g. [18, 24, 68, 69, 97, 124, 126] a rule is of the form

$$A \leftarrow B_1, \ldots, B_n$$

which says that the value associated with the implication $B_1 \land \ldots \land B_n \rightarrow A$ is $\alpha$. Computationally, given an assignment $I$ of values to the $B_i$, the value of $A$ is computed by taking the “conjunction” of the values $I(B_i)$ and then somehow “propagating” it to the rule head. The values the atoms may have are taken from a lattice. More recently, [18, 63, 69, 126] show that most of the frameworks dealing with imprecision and logic programming can be embedded into the IB framework.

To accommodate most frameworks, we have presented a general framework for logic programs with many-valued semantics (see, [120, 119, 121]) where the truth space is a complete lattice (a bilattice [49] is used in case we have to deal with default negation). Sorts, as used in [18, 24], can be simulated by using the join of lattices.

Rules and facts have the very general form

$$A \leftarrow f(B_1, \ldots, B_n),$$

where $f$ is an $n$-ary computable function over lattices and $B_i$ are atoms. Each rule may have a different $f$. Computationally, given an assignment (interpretation) $I$ of values to the $B_i$, the value of $A$ is computed by stating that $A$ is at least as true as $f(I(B_1), \ldots, I(B_n))$ (and, thus, follows the IB approach). The form of the rules is sufficiently expressive to encompass most approaches to many-valued normal logic programming.

**Contribution.** In this paper we present an simple top-down procedure to answer queries within our formalism. What makes our proposal different from all other ones ([19, 26, 61, 69, 119, 120, 121, 126], – see related work) is that we present a tabulation procedure, which allows us not just to compute one answer to a query, but rather all answers to a query, where an answer is a query instance together with its degree of truth. Being able to compute all answers to a query is especially very important as ultimately we are interested to rank all answers with respect to their degrees. As illustrative example suppose a user may have the following information need:
“Find cheap hotels near to the conference location.”

Here, cheap and near are fuzzy predicates. Unlike the classical case, tuples satisfy now these predicates to a score (usually in [0, 1]). In the former case the score may depend, e.g., on the price, while in the latter case it may depend e.g. on the distance between the hotel location and the conference location. Therefore, a major problem we have to face with in such cases is that now we want rather the set of all tuples ranked according to their score, instead of just one answer.

Related work. As seen above, there are many works dealing with imprecision with logic programming with or without negation, either using the AB approach or the IB approach. We recall that it is not difficult to see that the use of arbitrary computable truth combination functions in the body is sufficiently expressive to subsume all those works. Also, it is worth mentioning that very few works address non-monotonic reasoning, as [22, 43, 44, 72, 73, 74, 78, 94, 119, 121]

Additionally, in most frameworks, in order to answer to a query, we have to compute the whole intended model (e.g., by a bottom-up fixed-point computation) and then answer with the evaluation of the query in this model. This always requires the computation of a whole model, even if not all the atom’s truth is required to determine the answer. Some works that also present top-down procedures are [19, 26, 61, 69, 120, 126], but in none of them non-monotonic negation is considered. The only exception is [119, 121], which deals with normal logic programs over bilattices.

A common characteristics of all these top-down procedures is that they are some variant to the many-valued case of the usual classical SLD resolution [71]. Therefore, the are tailored to compute one answer only. The computation of all answers may led to a well-known non-termination issue in case recursive rules are allowed such as the typical “path” example

\[ \text{path}(x, y) \leftarrow \text{path}(x, z) \land \text{edge}(z, y). \]

Essentially, any goal containing \( \text{path}(x, y) \) will unify with the above rule over and over. We avoid this problem by applying a variant of so-called memoing techniques (tabling/magic sets) developed for classical logic programming (see, e.g. [130] for an overview). Essentially, the basic idea of our procedure is
to collect, during the computation, all correct answers incrementally together in a similar way as it is done for classical Datalog [123].

We proceed as follows. In the next section we present some basic definitions about the logic programming formalism.

2 Preliminaries

Truth spaces. The truth spaces we consider are complete lattices. A truth space is a structure \( L = \langle L, \preceq \rangle \) where \( L \) is a non-empty set and \( \preceq \) is a partial ordering giving \( L \) the structure of a complete lattice. Meet (or greatest lower bound) and join (or least upper bound) under \( \preceq \) are denoted \( \land \) and \( \lor \). Top and bottom under \( \preceq \) are denoted \( \top \) and \( \bot \).

The main idea is that an statement \( P(a) \), rather than being interpreted as either true or false, will be mapped into a truth value \( c \) in \( L \). Examples of typical truth spaces are:

- Classical 0-1: \( L_{\{0,1\}} \) corresponds to the classical truth-space, where 0 stands for ‘false’, while 1 stands for ‘true’.

- Fuzzy: \( L_{[0,1]} \), which relies on the rationals in the unit real interval, is quite frequently used as truth space.

- Four-valued: another frequent truth space is Belnap’s \( \mathcal{FOUR} \) [9], where \( L \) is \( \{f, t, u, i\} \) with \( f \preceq u \preceq t \) and \( f \preceq i \preceq t \). Here, \( u \) stands for ‘unknown’, whereas \( i \) stands for inconsistency. We denote the lattice as \( L_B \).

- Many-valued: \( L = \langle \{0, \frac{1}{n-1}, \ldots, \frac{n-2}{n-1}, 1\}, \preceq \rangle \), \( n \) positive integer.

- Belief-Doubt: a further popular lattice allows us to reason about belief and doubt. Indeed, the idea is to take any lattice \( L \), and to consider the cartesian product \( L \times L \). For any pair \( (b, d) \in L \times L \), \( b \) indicates the degree of belief a reasoning agent has about a sentence \( s \), while \( d \) indicates the degree of doubt the agent has about \( s \). The order on \( L \times L \) is determined by \( (b, d) \preceq (b', d') \) iff \( b \preceq b' \) and \( d' \preceq d \), i.e. belief goes up, while doubt goes down. The minimal element is \( (\bot, \top) \) (no

---

1See also, http://tinman.cs.gsu.edu/~raj/8710/f03/datalog-eval.html

2See [66] for more examples.
belief, maximal doubt), while the maximal element is \((\top, \bot)\) (maximal belief, no doubt). We indicate this lattice with \(\bar{\mathcal{L}}\).

- Interval: another lattice allows us to reason about intervals [43]. Indeed, the idea is to take any lattice \(\mathcal{L}\), to consider the cartesian product \(\mathcal{L} \times \mathcal{L}\) and for any pair \((b, d) \in \mathcal{L} \times \mathcal{L}\), \(b\) indicates the lower bound truth degree of a sentence \(s\), while \(d\) indicates the upper bound degree of \(s\). That is, \((b, d)\) denotes the interval of truth values \(\{c | b \preceq t c \preceq t d\}\). The order on \(\mathcal{L} \times \mathcal{L}\) is determined by \((b, d) \preceq (b', d')\) iff \(b \preceq b'\) and \(d \preceq d'\). The minimal element is \((\bot, \top)\) (no constraint), while the maximal element is \((\top, \bot)\) (maximal inconsistency).

We also provide a family \(\mathcal{F}\) of \(\preceq\)-monotone \(n\)-ary functions over \(\mathcal{L}\) to manipulate truth values.

**Generalized logic programs.** Fix a truth space \(\mathcal{L} = \langle L, \preceq \rangle\). We extend logic programs [71] to the case where computable functions \(f \in \mathcal{F}\) are allowed to manipulate truth values (see [119, 121]). That is, we allow any \(f \in \mathcal{F}\) to appear in the body of a rule to be used to combine the truth of the atoms appearing in the body. The language is sufficiently expressive to accommodate almost all frameworks on many-valued logic programming [119, 121].

A term, \(t\), is either a variable or a constant symbol. An atom, \(A\), is an expression of the form \(p(t_1, \ldots, t_n)\), where \(p\) is an \(n\)-ary predicate symbol and all \(t_i\) are terms. A formula, \(\varphi\), is an expression built up from the atoms, the truth values \(b \in L\) of the truth space and the functions \(f \in \mathcal{F}\). The members of the truth space may appear in a formula, as well as functions \(f \in \mathcal{F}\). For instance, e.g. in \(\mathcal{L}_{[0,1]}\), the expression

\[
\min(p, q) \cdot \max(r, 0.7) + v
\]

is a formula \(\varphi\), where \(p, q, r\) and \(v\) are atoms. The intuition here is that the truth value of the formula is obtained by determining the truth value of \(p, q, r\) and \(v\) and then to apply the arithmetic functions to determine the final value of \(\varphi\).

A rule is of the form

\[
A \leftarrow \varphi
\]

where \(A\) is an atom and \(\varphi\) is a formula.

\(^3\)With computable we mean that for any input, the value of \(f\) can be determined in finite time.
Example 1 For instance,

\[ p \leftarrow \max(0, q + r - 1) \]

is a rule dictating that \( p \) is at least as true as the conjunction of \( q \) and \( r \) with respect to the Lukasiewicz t-norm \( x \wedge y = \max(0, x + y - 1) \) \[53\]. On the other hand,

\[ P(x) \leftarrow A(x) + B(x) - A(x) \cdot B(x) \]

dictates that the truth of \( P(x) \) is determined by the algebraic sum of \( A(x) \) and \( B(X) \) and may be used to simulate reasoning in logic programming under the probabilistic event independence assumption.

We note that from a practical point of view, we may have introduced typed or sorted terms and predicates in which each argument has a type or sort (as in e.g. \[126\]). This is useful in practice, but from a theoretical point of view their management is straightforward, so we leave it out for the ease of presentation.

A generalized normal logic program, or simply logic program, \( \mathcal{P} \), is a finite set of rules. The notions of Herbrand universe \( H_{\mathcal{P}} \) of \( \mathcal{P} \) and Herbrand base (as the set of all ground atoms) \( B_{\mathcal{P}} \) of \( \mathcal{P} \) are as usual. Additionally, given \( \mathcal{P} \), the generalized normal logic program \( \mathcal{P}^* \) derived from grounding \( \mathcal{P} \) is constructed as follows:

1. set \( \mathcal{P}^* \) to the set of all ground instantiations of rules in \( \mathcal{P} \) \(^4\);
2. replace several rules in \( \mathcal{P}^* \) having same head, \( A \leftarrow \varphi_1, A \leftarrow \varphi_2, \ldots \) with \( A \leftarrow \varphi_1 \lor \varphi_2 \lor \ldots \) (recall that \( \lor \) is the join operator of the truth space); and
3. if an atom \( A \) is not head of any rule in \( \mathcal{P}^* \), then add the rule \( A \leftarrow f \) to \( \mathcal{P}^* \) (it is a standard practice in logic programming to consider such atoms as \textit{false}).

Note that in \( \mathcal{P}^* \), each atom appears in the head of \textit{exactly one} rule and that \( \mathcal{P}^* \) is \textit{finite}.

We next recall the usual semantics of logic programs over truth spaces (cf. \[119, 120\]).

\(^4\)Note that typed terms may reduce the size of \( \mathcal{P}^* \).
Interpretations. An interpretation $I$ on the truth space $\mathcal{L} = \langle L, \preceq \rangle$ is a mapping from atoms to members of $L$. $I$ is extended from atoms to formulae in the usual way: (i) for $b \in L$, $I(b) = b$; (ii) for formulae $\varphi$ and $\varphi'$, $I(\varphi \land \varphi') = I(\varphi) \land I(\varphi')$, and similarly for $\lor$; and (iii) for formulae $f(A)$, $I(f(A)) = f(I(A))$, and similarly for $n$-ary functions. $\preceq$ is extended from $L$ to the set $I(L)$ of all interpretations point-wise: (i) $I_1 \preceq I_2$ iff $I_1(A) \preceq I_2(A)$, for every ground atom $A$.

With $I_\bot$ we denote the bottom interpretation (it maps any atom into $\bot$). Of course, $\langle I(L), \preceq \rangle$ is a complete lattice as well.

Models. $I$ is a model of $\mathcal{P}$, denoted $I \models \mathcal{P}$, iff for all $A \leftarrow \varphi \in \mathcal{P}^*$, $I(A) = I(\varphi)$ holds. Note that usually a model has to satisfy $I(\varphi) \preceq I(A)$ only, i.e. $A \leftarrow \varphi \in \mathcal{P}^*$ specifies the necessary condition on $A$, “$A$ is at least as true as $\varphi$”. But, as $A \leftarrow \varphi \in \mathcal{P}^*$ is the unique rule with head $A$, the constraint becomes also sufficient (see also e.g. [44, 77, 78, 121]).

Among all the models, one model plays a special role: namely the $\preceq$-least model $M_\mathcal{P}$ of $\mathcal{P}$. Furthermore, for the sake of this paper, we note that the existence and uniqueness of $M_\mathcal{P}$ is guaranteed by the fixed-point characterization based on the $\preceq$-monotone function $\Phi_\mathcal{P}$: for an interpretation $I$, for any ground atom $A$ with (unique) $A \leftarrow \varphi \in \mathcal{P}^*$,

$$
\Phi_\mathcal{P}(I)(A) = I(\varphi).
$$

Then all models of $\mathcal{P}$ are fixed-points of $\Phi_\mathcal{P}$ and vice-versa, and $M_\mathcal{P}$ can be computed in the usual way by iterating $\Phi_\mathcal{P}$ over $I_\bot$.

In the following, we recall some examples, which both might help informally the reader to get confidence with the formalism and show how our formalism may capture different approaches to the management of imprecision (and some forms of uncertainty) in logic programming (some examples are taken from [69]).

Consider the following logic program with the four rules $r_i$,

$$
\begin{align*}
    r_1 & : A \leftarrow f_1(\alpha_1, B) \\
    r_2 & : A \leftarrow f_2(\alpha_2, C) \\
    r_3 & : B \leftarrow \alpha_3 \\
    r_4 & : C \leftarrow \alpha_4
\end{align*}
$$

where $A, B, C$ are ground atoms and $\alpha_i \in [0, 1]_\mathbb{Q}$.
Example 2 (Classical case, [69]) Consider $L_{[0,1]}$ and $\alpha_i = 1$, for $1 \leq i \leq 4$. Suppose $f_i$ is min. Then, $P$ is a program in the standard logic programming framework.

Example 3 ([40, 69]) Consider $L_{[0,1]}$. Suppose $\alpha_1 = 0.8$, $\alpha_2 = \alpha_3 = 0.7$, and $\alpha_4 = 0.8$ are possibility/necessity degrees associated with the implications. Suppose $f_i$ is min. Then $P$ is a program in the framework proposed by Dubois et al. [40], which is founded on Zadeh’s possibility theory [134]. In a fixed-point evaluation of $P$, the possibility/necessity degrees derived for $A$, $B$, $C$ are 0.7, 0.7, 0.8, respectively.

Example 4 ([124, 69]) Consider $L_{[0,1]}$ and suppose $\alpha_i$’s are defined as in Example 3. But, suppose $f_i$ is multiplication ($\cdot$). Then $P$ is a program in van Emden’s framework [124], which is mathematically founded on the theory of fuzzy sets proposed by Zadeh [133]. In a fixed-point evaluation of $P$, the values derived for $A$, $B$, $C$ are 0.56, 0.7, 0.8, respectively.

Example 5 (MYCIN [11, 69]) Consider $L_{[0,1]}$, and suppose $\alpha_i$’s are probabilities defined as in the previous example. Suppose $f_i$ is ($\cdot$). However, in order to simulate a probabilistic setting, in particular related to the atom $A$, with independent events, we write the program above as:

\[
\begin{align*}
  r_0 & : A \leftarrow f_s(A', A'') \\
  r_1' & : A' \leftarrow f_1(0.8, B) \\
  r_2' & : A'' \leftarrow f_2(0.7, C) \\
  r_3 & : B \leftarrow 0.7 \\
  r_4 & : C \leftarrow 0.8
\end{align*}
\]

where we use two new atoms $A'$ and $A''$ to indicate that $A$ is head of two rules and use the algebraic sum $f_s(\alpha, \beta) = \alpha + \beta - \alpha \cdot \beta$ to sum up the probabilities of deriving $A$. Viewing an atom as an event, $f_s$ returns the probability of the occurrence, of any one of two independent events, in the probabilistic sense. Note that $f_s$ is the disjunction function used in MYCIN [11]. Let us consider a fixed-point evaluation of $P$. In the first step, we derive $B$ and $C$ with probabilities 0.7 and 0.8, respectively. In step 2, applying $r_1'$ and $r_2'$, we obtain two derivations of $A$ (namely for $A'$ and $A''$), the probability of each of which is 0.56. The probability of $A$ is then defined as $f_s(0.56, 0.56) = 0.8064$, which is indeed the probability that $A$ occurs.
From Example 5 above, it is easy to see that more generally, in order to accommodate independent probabilities, a logic program \( \mathcal{P} \) has to be transformed: rules with same head \( A \leftarrow \varphi_1, A \leftarrow \varphi_2, \ldots \) rather being transformed into \( A \leftarrow \varphi_1 \lor \varphi_2 \lor \ldots \), are transformed into \( A \leftarrow f_s(\ldots f_s(\varphi_1, \varphi_2) \ldots) \).

In a similar way, we can manage \([69]\).

**Example 6 (PDDU \([69, 74]\))** In \([69]\), a Parametric Approach to Deductive Databases with Uncertainty (PDDU) is proposed, where rules have the form

\[
\begin{align*}
\alpha_r & : A \leftarrow B_1, \ldots, B_n; (f_d, f_p, f_c) \\
\end{align*}
\]

\(f_d\) is the disjunction function associated with \(A\) and, \(f_c\) and \(f_p\) are respectively the conjunction and propagation functions associated with the rule \(r\). \(\alpha_r\) is the weight of the rule. Roughly, this functions are mappings from \(L \times L\) to \(L\) and are continuous w.r.t. each one of its arguments and satisfying some constraints such that they behave as conjunction and disjunction functions (see, \([69]\)). The intuition behind a rule is as follows. Ground the program and evaluate each atom \(B_i\). Combine their truth using the conjunction function \(f_c\), i.e. let \(c_1 = f_c(B_1, \ldots, B_n)\) (for instance, \(c_1 = \text{min}(B_1, \ldots, B_n)\)). Then propagate the truth value \(c_1\) to the head using the weight of the rule \(\alpha_r\) and the propagation function \(f_p\), i.e. let \(c'_1 = f_p(\alpha_r, c_1)\) (for instance, \(c'_1 = \alpha_r \cdot c_2\)). Repeat this operation for rules heaving \(A\) in the head. If \(c'_1, \ldots, c'_k\) are all this values, combine them using the disjunction function \(f_d\), \(c_A = f_d(c'_1, \ldots, c'_k)\) (for instance, \(c_A = \text{max}(c'_1, \ldots, c'_k)\)). Informally, a logic program \(\mathcal{P}\) in the sense of \([69]\) can be represented in our framework by grounding \(\mathcal{P}\) and then transforming rule of the form \(r : A \leftarrow B_1, \ldots, B_n; (f_d, f_p, f_c)\) into

\[
A \leftarrow f_p(\alpha_r, f_c(B_1, \ldots, B_n))
\]

Afterwards, all rules with same head \(A \leftarrow \varphi_1, A \leftarrow \varphi_2, \ldots\) are transformed into \(A \leftarrow f_d(\ldots f_d(\varphi_1, \varphi_2) \ldots)\).

\([74]\) is as \([69]\), but additionally non-monotonic negation is considered as well. We can encode \([74]\) into our framework in the same way as for \([69]\).

**Example 7 (Fuzzy Logic Programming \([126]\))** In \([126]\), Fuzzy Logic Programming is proposed, where rules have the form

\[
A \leftarrow f(B_1, \ldots, B_n)
\]
for some specific \( f \) and the truth space is \( \mathcal{L}_{[0,1]} \). \cite{126} is just a special case of our framework. As an illustrative example consider the following scenario. Assume that we have the following facts, represented in the tables below. There are hotels and conferences, their locations and the distance among locations.

<table>
<thead>
<tr>
<th>HasLocationH</th>
<th>HasLocationC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HotelID</td>
<td>LocationH</td>
</tr>
<tr>
<td>h1</td>
<td>h1</td>
</tr>
<tr>
<td>h2</td>
<td>h2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasLocationH</td>
</tr>
<tr>
<td>h1</td>
</tr>
<tr>
<td>h1</td>
</tr>
<tr>
<td>h2</td>
</tr>
<tr>
<td>h2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Now, suppose that our query is to find hotels close to the conference venue, labeled \( c1 \). We may formulate our query as the rule:

\[
\text{Query}(h) \leftarrow \min(\text{HasLocationH}(h, hl), \text{HasLocationC}(c1, cl), \text{Distance}(hl, cl, d), \text{Close}(d))
\]

where \( \text{Close}(x) \) is defined as

\[
\text{Close}(x) = \max(0, 1 - \frac{x}{1000})
\]

As a result to that query we get a ranked list of hotels as shown in the table below.

<table>
<thead>
<tr>
<th>Result List</th>
</tr>
</thead>
<tbody>
<tr>
<td>HotelID</td>
</tr>
<tr>
<td>h1</td>
</tr>
<tr>
<td>h2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Example 8 ([74]) Consider $\mathcal{L}_{[0,1]_Q}$, where $\land = \min$ and $\lor = \max$. Consider an insurance company, which has information about its customers used to determine the risk coefficient of each customer. Suppose a value of the risk coefficient is already known, but has to be re-evaluated (the client is a new client and his risk coefficient is given by his precedent insurance company). The company may have: (i) data grouped into a set of facts

$$\begin{align*}
\text{Experience}(\text{john}) & \leftarrow 0.7 \\
\text{Risk}(\text{john}) & \leftarrow 0.5 \\
\text{Sport\_car}(\text{john}) & \leftarrow 0.8 ;
\end{align*}$$

and (ii) a set of rules, which after grounding are:

$$\begin{align*}
\text{Good\_driver}(\text{john}) & \leftarrow \text{Experience}(\text{john}) \land (0.5 \cdot \text{Risk}(\text{john})) \\
\text{Risk}(\text{john}) & \leftarrow 0.8 \cdot \text{Young}(\text{john}) \\
\text{Risk}(\text{john}) & \leftarrow 0.8 \cdot \text{Sport\_car}(\text{john}) \\
\text{Risk}(\text{john}) & \leftarrow \text{Experience}(\text{john}) \land (0.5 \cdot \text{Good\_driver}(\text{john}))
\end{align*}$$

It turns out that by a bottom-up computation (iterating $\Phi_P$ over $I_\perp$) the minimal model is $M_P$, where (for ease, we use first letters only)

$$\begin{align*}
M_P(R(j)) &= 0.64 \\
M_P(S(j)) &= 0.8 \\
M_P(Y(j)) &= 0 \\
M_P(G(j)) &= 0.32 \\
M_P(E(j)) &= 0.7 .
\end{align*}$$

The following examples shows that $\omega$ steps may not be sufficient to compute the minimal model.

Example 9 Consider $\mathcal{L}_{[0,1]_Q}$, the function $f(x) = \frac{x+a}{2}$ ($0 < a \leq 1, a \in \mathbb{Q}$), and $P_1 = \{A \leftarrow f(A)\}$. Then the minimal model is attained after $\omega$ steps of $\Phi_P$ iterations starting from $I_\perp(A) = 0$ and is $M_P(A) = a$.

Now, consider the function $g(x) = 0$ if $x < a$, and $g(x) = 1$ otherwise, and the logic program $P_2$ with rules

$$\begin{align*}
A & \leftarrow f(A) \\
B & \leftarrow g(A) .
\end{align*}$$

Then the minimal model is attained after $\omega + 1$ steps of $\Phi_P$ iterations starting from $I_\perp$ and is $M_P(A) = a, M_P(B) = 1$. 

12
However, it is easy not difficult to show that if all functions appearing in $P$ are continuous, then at most $\omega$ steps are necessary to compute the minimal model.

**Top-down query answering.** A query is an expression of the form $?A$ (*query atom*), intended as a question about the truth of the atom $A$ in the minimal model of $P$. We also allow a query to be a set $?{A_1, \ldots, A_n}$ of query atoms. In that latter case we ask about the truth of all the atoms $A_i$ in the minimal model.

We next recall a basic procedure *Solve* for top-down query answering. Though the procedure and variants of it are reported elsewhere [119, 120, 121], we report it here not only for the sake of completeness of the paper, but also because

- our novel procedure for computing all answers, described in the next section, is inspired on *Solve* and makes its explanation much easier;
- the computational complexity analysis and termination results we are describing here apply to the novel procedure as well.

We anticipate that the main reason why the procedure *Solve* is not suitable to be used for computing all answers to a query $?A$, given $P$, is that (i) it basically relies on its grounded version $P^*$, which may be rather huge (exponential with respect to $|P|$, in general) in applications with many facts; and (ii) it is rather tailored towards ground queries, only.

The top-down procedure [119, 120, 121] is based on a transformation of a program into a system of equations of monotonic functions over complete lattices for which we can compute the least fixed-point in a top-down style. The idea is the following. Consider a complete lattice $L = \langle L, \preceq \rangle$, a logic program $P$, its Herbrand base $B_P = \{A_1, \ldots, A_n\}$ and $P^*$. Let us associate to each atom $A_i \in B_P$ a variable $x_i$, which will take a value in the domain $L$ (sometimes, we will refer to that variable with $x_A$ as well). An interpretation $I$ may be seen as an assignment of truth values to the variables $x_1, \ldots, x_n$. For the immediate consequence operator $\Phi_P$, a fixed-point is such that $I = \Phi_P(I)$, i.e. for all atoms $A_i \in B_P$, $I(A_i) = \Phi_P(I)(A_i)$. Therefore, we may identify the fixed-points of $\Phi_P$ as the solutions over $L$ of the system
of equations of the following form:

\[
\begin{align*}
  x_1 &= f_1(x_{11}, \ldots, x_{1a_1}), \\
  & \vdots \\
  x_n &= f_n(x_{n1}, \ldots, x_{nan}),
\end{align*}
\]

(1)

where for \(1 \leq i \leq n\), \(1 \leq k \leq a_i\), we have \(1 \leq i_k \leq n\). Each variable \(x_{ik}\) will take a value in the domain \(L\), each (monotone) function \(f_i\) determines the value of \(x_i\) (i.e. \(A_i\)) given an assignment \(I(A_{ik})\) to each of the \(a_i\) variables \(x_{ik}\). The function \(f_i\) implements \(\Phi_P(I(A_i))\). The models of \(P\) are bijectively related to the solutions of the system (1) and the \(\preceq\)-least solution corresponds to the \(\preceq\)-least model of \(P\), i.e. \(M_P\).

In the general case, we assume that each function \(f_i : L^{a_i} \mapsto L\) in Equation (1) is \(\preceq\)-monotone. We also use \(f_x\) in place of \(f_i\), for \(x = x_i\). We refer to the monotone system as in Equation (1) as the tuple \(S = (\mathcal{L}, V, \mathbf{f})\), where \(\mathcal{L}\) is a lattice, \(V = \{x_1, ..., x_n\}\) are the variables and \(\mathbf{f} = (f_1, ..., f_n)\) is the tuple of functions.

As it is well known, a monotonic equation system as (1) has a \(\preceq\)-least solution, \(\text{lfp}(\mathbf{f})\), the \(\preceq\)-least fixed-point of \(\mathbf{f}\) is given as the least upper bound of the \(\preceq\)-monotone sequence, \(y_0, \ldots, y_i, \ldots\), where (of course, \(\mathbf{f}(y) = (f_1(y), ..., f_n(y))\))

\[
\begin{align*}
  y_0 &= \perp \\
  y_{i+1} &= \mathbf{f}(y_i)
\end{align*}
\]

(we point out that in the frequent case of \(\mathcal{L}_{[0,1]}\) in which each \(f_i\) is a linear function, we may apply directly linear algebra to solve the equational system (1)).

**Example 10** Consider Example 8. Given \(P\), we consider \(P^*\) and represent it as an equational system over \([0,1]_{\mathbb{Q}}\) as follows:

\[
\begin{align*}
  x_{E(j)} &= 0.7 \\
  x_{S(j)} &= 0.8 \\
  x_{Y(j)} &= 0 \\
  x_{G(j)} &= \min(x_{E(j)}, 0.5 \cdot x_{R(j)}) \\
  x_{R(j)} &= \max(0.5, 0.8 \cdot x_{Y(j)}, 0.8 \cdot x_{S(j)}, \min(x_{E(j)}, 0.5 \cdot x_{G(j)}))
\end{align*}
\]
It is easily verified that the fixed-points of the $\Phi_P$ operator, i.e. the models of $P$, are the solutions of the system of equations above and, thus, the least fixed-point, corresponds to the minimal model $M_P$ of $P$. The bottom-up least fixed-point computation is (the tuples represent $\langle x_{E(j)}, x_{S(j)}, x_{Y(j)}, x_{G(j)}, x_{R(j)} \rangle$)

\[
\begin{align*}
y_0 &= \langle 0, 0, 0, 0, 0 \rangle \\
y_1 &= \langle 0.7, 0.8, 0, 0, 0.5 \rangle \\
y_2 &= \langle 0.7, 0.8, 0, 0.25, 0.64 \rangle \\
y_3 &= \langle 0.7, 0.8, 0, 0.32, 0.64 \rangle \\
y_4 &= y_3,
\end{align*}
\]

which corresponds to the minimal model of the program, as expected.

Informally, the algorithm works as follows (see Table 1). Assume, we are interested in the value of $x_0$ in the fixed-point $\text{lfp}(f)$ of the system. We call the procedure with $\text{Solve}(S, \{x_0\})$. We associate to each variable $x_i$ a marking $v(x_i)$ denoting the current value of $x_i$ (the mapping $v$ contains the current value associated to the variables). Initially, $v(x_i)$ is 0. We start with putting $x_0$ in the active list of variables $A$, for which we evaluate whether the current value of the variable is identical to whatever its right-hand side evaluates to. When evaluating a right-hand side it might of course turn out that we do indeed need a better value of some sons, which will assumed to have the value 0 and put them on the list of active nodes to be examined. In doing so we keep track of the dependencies between variables, and whenever it turns out that a variable changes its value all variables that might depend on this variable are put in the active set to be examined. At some point (even if cyclic definitions are present) the active list will become empty and we have actually found part of the fixed-point, sufficient to determine the value of the query $x_0$. \footnote{The attentive reader will notice that the $\text{Solve}$ has commonalities with the so-called \textit{tabulation} procedures, like [16, 19].}

$\text{Solve}(S, Q)$ uses some auxiliary functions and data structures: given the equational system (1),

- $s(x)$ denotes the set of sons of $x$, i.e. $s(x_i) = \{x_{i_1}, \ldots, x_{i_a} \}$ (the set of variables appearing in the right hand side of the definition of $x_i$);

- $p(x)$ denotes the set of parents of $x$, i.e. the set $p(x) = \{x_i : x \in s(x_i) \}$ (the set of variables depending on the value of $x$).
Procedure \textit{Solve}$(S, Q)$

\textbf{Input:} \(\preceq\)-monotonic system \(S = (L,V,f)\), where \(Q \subseteq V\) is the set of query variables

\textbf{Output:} A set \(B \subseteq V\), with \(Q \subseteq B\) such that the mapping \(v\) restricted to \(B\), equals to \(\text{lfp}(f)\).

1. \(A := Q, \; \text{dg} := Q, \; \text{in} := \emptyset, \; \text{for all} \; x \in V \; \text{do} \; v(x) = 0, \; \exp(x) = \text{false}\)
2. \(\text{while } A \neq \emptyset \; \text{do}\)
3. \(\text{select } x_i \in A, \; A := A \setminus \{x_i\}, \; \text{dg} := \text{dg} \cup \mathcal{s}(x_i)\)
4. \(r := f_i(v(x_{i_1}),...,v(x_{i_n}))\)
5. \(\text{if } r \succ v(x_i) \; \text{then } v(x_i) := r, \; A := A \cup (p(x_i) \cap \text{dg}) \; \text{fi}\)
6. \(\text{if not } \exp(x_i) \; \text{then } \exp(x_i) = \text{true}, \; A := A \cup (s(x_i) \setminus \text{in}), \; \text{in} := \text{in} \cup s(x_i) \; \text{fi}\)

\textbf{Table 1:} General top-down algorithm. Grounded version.

- the variable \text{dg} collects the variables that may influence the value of the query variables;

- the array variable \text{exp} traces the equations that has been “expanded” (the body variables are put into the active list);

- the variable \text{in} keeps track of the variables that have been put into the active list so far due to an expansion (to avoid, to put the same variable multiple times in the active list due to function body expansion).

\textbf{Example 11} Consider Example 8 and query variable \(x_{R(j)}\) (we ask for the risk coefficient of John). In Table 2 we report a sequence of \text{Solve}$(S, \{x_{R(j)}\})$ computation. Each line is a sequence of steps in the ‘while loop’. What is left unchanged is not reported.

The fact that only a part of the model is computed becomes evident, as the computation does not change if we add any program \(P'\) to \(P\) not containing atoms of \(P\), while a bottom-up computation will consider \(P'\) as well.

Given a system \(S = (L,V,f)\), where \(L = (L, \preceq)\), let \(h(L)\) be the \textit{height} of the truth-value set \(L\), i.e. the length of the longest strictly \(\preceq\)-increasing chain in \(L\) minus 1, where the length of a chain \(v_1,...,v_\alpha,...\) is the cardinal \(|\{v_1,...,v_\alpha,...\}|\). The \textit{cardinal} of a set \(X\) is the least ordinal \(\alpha\) such that \(\alpha\) and \(X\) are \textit{equipollent}, i.e. there is a bijection from \(\alpha\) to \(X\). For instance, \(h(\text{FOUR}) = 2\) w.r.t. \(\preceq_k\) as well as w.r.t. \(\preceq_t\), while \(h([0,1]_Q) = \omega\). It can be shown that the above algorithm behaves correctly.
1. \( A := \{ x_{R(j)} \}, x_i := x_{R(j)}, A := \emptyset, d_g := \{ x_{R(j)}, x_{Y(j)}, x_{S(j)}, x_{E(j)}, x_{G(j)} \}, r := 0.5, v(x_{R(j)}) := 0.5, \\
A := \{ x_{G(j)} \}, \exp(x_{R(j)}) := 1, A := \{ x_{Y(j)}, x_{S(j)}, x_{E(j)}, x_{G(j)} \}, \text{in} := \{ x_{Y(j)}, x_{S(j)}, x_{E(j)}, x_{G(j)} \} \\
2. \( x_i := x_{Y(j)}, A := \{ x_{S(j)}, x_{E(j)}, x_{G(j)} \}, r := 0, \exp(x_{Y(j)}) := 1 \\
3. \( x_i := x_{S(j)}, A := \{ x_{E(j)}, x_{G(j)} \}, r := 0.8, v(x_{S(j)}) := 0.8, A := \{ x_{E(j)}, x_{G(j)}, x_{R(j)} \}, \exp(x_{S(j)}) := 1 \\
4. \( x_i := x_{E(j)}, A := \{ x_{G(j)}, x_{R(j)} \}, r := 0.7, v(x_{E(j)}) := 0.7, \exp(x_{E(j)}) := 1 \\
5. \( x_i := x_{G(j)}, A := \{ x_{R(j)} \}, r := 0.25, v(x_{G(j)}) := 0.25, \exp(x_{G(j)}) := 1, \\
\text{in} := \{ x_{Y(j)}, x_{S(j)}, x_{E(j)}, x_{G(j)}, x_{R(j)} \} \\
6. \( x_i := x_{R(j)}, A := \emptyset, r := 0.64, v(x_{R(j)}) := 0.64, A := \{ x_{G(j)} \} \\
7. \( x_i := x_{G(j)}, A := \emptyset, r := 0.32, v(x_{G(j)}) := 0.32, A := \{ x_{R(j)} \} \\
8. \( x_i := x_{G(j)}, A := \emptyset, r := 0.64 \\
9. \text{stop. return v} \) (in particular, \( v(x_{R(j)}) = 0.64 \))

Table 2: Top-down computation related to Example 11.

**Proposition 1 ([119, 120])** Given a monotone system of equations \( S = (L, V, f) \), then there is a limit ordinal \( \lambda \) such that after \( |\lambda| \) steps \( \text{Solve}(S, Q) \) determines a set \( B \subseteq V \), with \( Q \subseteq B \) such that the mapping \( v \) equals \( \text{lfp}(f) \) on \( B \), i.e. \( v|_B = \text{lfp}(f)|_B \).

**Computational complexity.** From a computational point of view, by means of appropriate data structures, the operations on \( A, v, d_g, \text{in}, \exp, p \) and \( s \) can be performed in constant time. Therefore, step 1 is \( O(|V|) \), all other steps, except step 2 and step 4 are \( O(1) \). Let \( c(f_x) \) be the maximal cost of evaluating function \( f_x \) on its arguments, so step 4 is \( O(c(f_x)) \). It remains to determine the number of loops of step 2. In case the height \( h(L) \) of the lattice \( L \) is finite, observe that any variable is increasing in the \( \preceq \) order as it enters in the \( A \) list (step 5), except it enters due to step 6, which may happen one time only. Therefore, each variable \( x_i \) will appear in \( A \) at most \( a_i \cdot h(L) + 1 \) times, where \( a_i \) is the arity of \( f_i \), as a variable is only re-entered into \( A \) if one of its sons gets an increased value (which for each son only can happen \( h(L) \) times), plus the additional entry due to step 6. As a consequence, the worst-case complexity is
\[ O\left( \sum_{x_i \in V} \left( c(f_i) \cdot (a_i \cdot h(L) + 1) \right) \right). \]

Therefore:

**Proposition 2 ([119, 120])** Given a monotone system of equations \( S = \langle \mathcal{L}, V, f \rangle \). If the computing cost of each function in \( f \) is bounded by \( c \), the arity bounded by \( a \), and the height is bounded by \( h \), then the worst-case complexity of the algorithm Solve is \( O(|V|cah) \).

In case the height of a lattice is not finite, the computation may not terminate after a finite number of steps (see Example 9). Fortunately, under reasonable assumptions on the functions, we may guarantee the termination of Solve. We exploit two of such conditions. Consider a monotonic equational system \( S = \langle \mathcal{L}, V, f \rangle \). Consider a function \( f : \mathcal{L} \rightarrow \mathcal{L} \), where \( \langle \mathcal{L}, \preceq \rangle \) is a lattice. Let \([\bot]_f\) be the \( f \)-closure of \( \{\bot\} \), i.e. the smallest set that contains \( \{\bot\} \) and is closed under \( f \). We say that \( f \) has a finite generation (see also [10] for more on this issue) iff \([\bot]_f\) is finite. For instance, it can be verified that the functions \( \land, \lor \) have a finite generation on any finite set \( X \subseteq \mathcal{L} \). More concretely, over \( \mathcal{L}_{[0,1]} \), \( \min, \max \) and \( \max(x + y - 1, 0), \min(x + y, 1) \) have a finite generation, while e.g. the product \( x \cdot y \) and the algebraic sum \( x + y - x \cdot y \) have not. Note also that if \( f, g \) have a finite generation over \( X \) then so has \( f \circ g \). Therefore, given an equational system \( S = \langle \mathcal{L}, V, f \rangle \). If \( f \) has a finite generation, then \([\bot]_f\) is finite. That is, \( \{\bot, f(\bot), f^2(\bot), \ldots\} \) is finite. In particular, on induction on the computation of the \( \preceq \)-least fixed-point of \( S \) it can be shown that at each step of the bottom-up computation of the \( \preceq \)-least fixed-point, the values of the variables are in \([\bot]_f\). Therefore, the height of \([\bot]_f, h([\bot]_f) \), is finite. On the other hand, it can easily be seen that Solve terminates if the sequence, \( \bot, f(\bot), f^2(\bot), \ldots \) converges after a finite number of steps. Therefore:

**Proposition 3 ([119, 120])** Given a monotone system of equations \( S = \langle \mathcal{L}, V, f \rangle \). Then Solve terminates iff \( f \) has a finite generation. If the cost of computing each of the functions in \( f \) is bounded by \( c \) and the arity bounded by \( a \) then the worst-case complexity of the algorithm Solve is \( O(|V|cah) \), where \( h \) is the height of \([\bot]_f\).

The second condition, which guarantees the termination of Solve, is inspired directly by [18] and is a special case of above. On bilattices, we
say that a function \( f : \mathcal{L}^n \rightarrow \mathcal{L} \) is bounded iff \( f(x_1, \ldots, x_n) \preceq \bigwedge_i x_i \). Now, consider a monotone system of equations \( S = \langle \mathcal{L}, V, f \rangle \). We say that \( f \) is bounded iff each \( f_i \) is a composition of functions, each of which is either bounded, or a constant in \( \mathcal{L} \) or one of \( \vee, \wedge \). For instance, the function in Example 9 is not bounded, while \( f_i(x, y) = \max(0, x + y - 1) \wedge 0.3 \) over \( \mathcal{L}_{[0,1]} \) is. The idea is to prevent the existence of an infinite ascending chain of the form \( \perp \prec f(\perp) \prec \ldots \prec f^m(\perp) \prec \ldots \). In fact, roughly, consider a \( \preceq \)-monotone function \( f = g \circ h \), where \( g \) is a bounded function, while \( h \) is the composition of constants in \( \mathcal{L} \) or functions among \( \vee, \wedge \). Then \( \perp \preceq f(\perp) = g \circ h(\perp) = g(h(\perp)) \preceq h(\perp) \). But \( h \) has a finite generation and, thus, so has \( f \). The argument for \( f = h \circ g \) is similar. Therefore:

**Proposition 4 ([119, 120])** Given a monotone system of equations \( S = \langle \mathcal{L}, V, f \rangle \), where \( f \) is bounded. Then \( \text{Solve} \) terminates.

Note that for bounded functions \( f = g \circ h \), the height of \([\perp]_f \) is given by the height of \([\perp]_h \). This latter height is bounded by the number \( n = |V| \) as \( h^n(\perp) = h^{n+1}(\perp) \) (this is compatible with [18]). This implies that the worst-case complexity of the algorithm \( \text{Solve} \) is \( O(|V|^2ca) \) in that case.

We are ready now to finalize the query answering procedure. Let \( P \) be a logic program and consider \( P^* \). As already pointed out, each atom appears exactly once in the head of a rule in \( P^* \). The system of equations that we build from \( P^* \) is straightforward. Assign to each atom \( A \) a variable \( x_A \) and substitute in \( P^* \) each occurrence of \( A \) with \( x_A \). Finally, substitute each occurrence of \( \leftarrow \) with \( = \) and let \( \mathcal{S}(P) = \langle \mathcal{L}, V, f_P \rangle \) be the resulting equational system. Of course, \( |V| = |B_P| \), \( |\mathcal{S}(P)| \) can be computed in time \( O(|P|) \) and all functions in \( \mathcal{S}(P) \) are \( \preceq \)-monotone. As \( f_P \) is one to one related to \( \Phi_P \), it follows that the \( \preceq \)-least fixed-point of \( \mathcal{S}(P) \) corresponds to the minimal model of \( P \). The algorithm \( \text{Answer}(\mathcal{L}, P, ?A) \), first computes \( \mathcal{S}(P) \) and then calls \( \text{Solve}(\mathcal{S}(P), \{x_A\}) \) and returns the output \( v \) on the query variable, where \( v \) is the output of the call to \( \text{Solve} \). \( \text{Answer} \) behaves correctly (see Example 11).

**Proposition 5 ([119, 120])** Let \( \mathcal{L}, P \) and \(?A \) be a be a truth space, a logic program and a query, respectively. Then \( M_P(\mathcal{L}) = \text{Answer}(\mathcal{L}, P, \{?A\}) \) \(^6\).

\(^6\)The extension to a set of query atoms is straightforward.
From a computational point of view, we can avoid the cost of translating \( P \) into \( S(P) \) as we can directly operate on \( P \). So the cost \( O(|P|) \) can be avoided. In case the height of the lattice is finite, from Proposition 2 it follows immediately that the worst-case complexity for top-down query answering under the minimal model semantics of a logic program \( P \) is \( O(|BP|cah) \).

Furthermore, often the cost of computing each of the functions of \( f_P \) is in \( O(1) \). By observing that \( |BP|a \) is in \( O(|P|) \) we immediately have that in this case the complexity is \( O(|P|h) \).

It follows that over the lattice \( FOUR \) (\( h = 2 \)) the top-down algorithm works in linear time as in case the height is a fixed parameter, i.e. a constant. We can conclude that the additional expressive power of logic programs over lattices (with functions with constant cost, and lattices with fixed height) does not increase the computational complexity of classical propositional logic programs, which is linear. The computational complexity of the case where the height of the lattice is not finite is determined by Proposition 3 and Proposition 4. In general, the continuity of the functions in \( S(P) \) guarantees the termination after at most \( \omega \) steps.

### 3 Computing all answers

Often a query atom \(?Q\) over a logic program \( P \) has associated a rule in \( P \) of the form

\[
Q(x) \leftarrow \varphi(x, y)
\]

in which we ask for all substitutions \( \theta \) of the variables \( x \) and truth degrees \( v \), i.e. answers \( \langle \theta, v \rangle \), such that \( MP(Q(x)\theta) = v \) holds. That is, by substituting the variables in \( x \) using \( \theta \), the evaluation of the query in the minimal model is \( v \). We say that the answer \( \langle \theta, v \rangle \) is correct iff \( MP(Q(x)\theta) = v \).

By relying on the method of the previous section, one immediate way to compute all answers is as follows. We consider \( P^* \), which contains all instantiations of the query rule. That is, \( P^* \) contains a rule

\[
Q(c) \leftarrow \bigvee_{c'} \varphi(c, c')
\]

for each tuple \( c \) which can be formed from the constants of the Herbrand universe \( HP \) of \( P \) (similarly, \( c' \) is a tuple of constants in \( HP \)). As a a consequence, in order to find all answers \( \langle \theta, v \rangle \) to \(?Q\), we have to call the procedure
Answer with the list of query atoms $S_Q$ defined as

$$S_Q := \bigcup_c \{Q(c)\}.$$

Answer($S(P), S_Q$) returns then the degree of truth $v_c$ of all instantiations $Q(c)$ of $Q(x)$. All the answer tuples $\langle \theta_c, v_c \rangle$, where $\theta_c$ is the substitution $\theta_c = \{x/c\}$, are the answers we are looking for.

A drawback of query answering based on grounded logic programs is that that both the size of $S_Q$, $|S_Q|$, and that of $P^*$ may be large. In particular, $P^*$ may have exponentially more rules than $P$. In the following we modify the Answer algorithm in such a way, which allows us to find all answers $\langle \theta, v \rangle$ to query $?Q$, without grounding neither the program nor the query.

For the rest of the paper we assume that $P$ is made out of an extensional database (EDB), $P_E$, and an intentional database (IDB), $P_I$. The extensional database is a set of facts of the form

$$R(c_1, \ldots, c_n) \leftarrow b,$$

where $R(c_1, \ldots, c_n)$ is a ground atom and $b$ is a truth value. The intentional database is a set of rules for the form

$$P(x) \leftarrow \varphi(x, y)$$

in which the predicates occurring in the extensional database (called extensional predicates) do not occur in the head of rules of the intentional database. Essentially, we do not allow that the fact predicates occurring in $P_E$ can be redefined by $P_I$. We also assume that the intentional predicate symbol $P$ occurs in the head of at most one rule in the intentional database. Due to the expressiveness of rule bodies, it is not difficult to see that by using an equality predicate, logic programs can be put into this form.

For convenience, for each $n$-ary extensional predicate $R$, we represent the facts $R(c_1, \ldots, c_n) \leftarrow b$ in $P$ by means of a relational $n + 1$-ary table $tab_R$, containing the records $(c_1, \ldots, c_n, b)$. Thus, the table contains all the instances of $R$ together with their degree.

An answer set is a set of answers. Given two answers $\delta_1 = \langle \theta_1, v_1 \rangle$ and $\delta_2 = \langle \theta_2, v_2 \rangle$, we define $\delta_1 \preceq \delta_2$ iff $\theta_1 = \theta_2$ and $v_1 \leq v_2$. We write $\delta_1 \prec \delta_2$ iff $\theta_1 = \theta_2$ and $v_1 < v_2$. The intuition is that $\delta_2$ represents a “better” answer than $\delta_1$. If $\Delta_1$ and $\Delta_2$ are two sets of answers, we write $\Delta_1 \preceq \Delta_2$ iff for all
\( \delta_1 \in \Delta_1 \) there is \( \delta_2 \in \Delta_2 \) such that \( \delta_1 \preceq \delta_2 \). We write \( \Delta_1 < \Delta_2 \) iff \( \Delta_1 \preceq \Delta_2 \) and there is \( \delta_2 \in \Delta \) such that for no \( \delta_1 \in \Delta \), \( \delta_2 \preceq \delta_1 \) holds. Analogously, the intuition is that \( \Delta_2 \) represents a “better” set of answers than \( \Delta_1 \). For a given \( n \)-ary predicate \( P \) and a set of answers \( \Delta_P \) of \( P \), i.e. a set of answers \( \langle \theta, v \rangle \) in which \( \theta = \{ x_1/c_1, \ldots, x_n/c_n \} \) is a substitution of the \( n \) variables with constants (\( P\theta \) is ground), for convenience we represent \( \Delta_P \) as an \( n + 1 \)-ary table \( \text{tab}_{\Delta_P} \), containing the records \( \langle c_1, \ldots, c_n, v \rangle \).

For the sake of illustrative purposes, we consider the following well-known example in which we define a path a weighted graph.

**Example 12** Consider the lattice \( \mathcal{L}_{[0, 1]} \), and the following rules:

\[
\begin{align*}
\text{path}(x, y) & \leftarrow \text{edge}(x, y) \\
\text{path}(x, y) & \leftarrow \text{path}(x, z) \land \text{edge}(z, y)
\end{align*}
\]

These rules have to be transformed into a form in which the \text{path} predicate appears in one rule only:

\[
\text{path}(x, y) \leftarrow \text{edge}(x, y) \lor (\text{path}(x, z) \land \text{edge}(z, y)) \quad (4)
\]

The logic program \( \mathcal{P} \) contains the intentional database containing rule (4) and the extensional database of edges shown in the relational table \( \text{tab}_{\text{edge}} \) below (the omitted edges have degree 0):

<table>
<thead>
<tr>
<th>\text{tab}_{\text{edge}}</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( b )</td>
</tr>
<tr>
<td>( b )</td>
<td>( a )</td>
</tr>
<tr>
<td>( a )</td>
<td>( c )</td>
</tr>
<tr>
<td>( c )</td>
<td>( b )</td>
</tr>
</tbody>
</table>

Of course, \( \text{tab}_{\text{edge}} \) is also the set \( \text{tab}_{\Delta_{\text{edge}}} \) of correct answers of predicate \( \text{edge} \), while it can be verified that the set of correct answers of predicate \( \text{path} \) is
given by the relational table:

<table>
<thead>
<tr>
<th>tab_{\Delta_{\text{path}}}</th>
<th>a</th>
<th>a</th>
<th>0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>c</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>a</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>c</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>a</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>b</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>0.4</td>
</tr>
</tbody>
</table>

One possibility to compute a correct answer is to follow a SLD-refutation like procedure \[71\] adapted to the many-valued case, as described for instance in \[25, 126\]. To compute all answers corresponds to compute all SLD-refutations. The downside of this approach is that in case of recursive definitions such as in Example 12 above, we may lead to an infinite loop (due to the recursive definition of \text{path}).

One way to overcome to this difficulty is to use so-called memoing techniques (\textit{tabling/magic sets}) developed for classical logic programming (see, e.g. \[130\] for an overview).

In this work, we present a tabling like procedure applied to our top-down query answering procedure presented for grounded logic programs. The basic idea of our procedure is similar to the \textit{Solve} algorithm, but now, we try to collect, during the computation, all correct answers incrementally.

At first, consider a general rule of the form (3), i.e. \( p(x) \leftarrow \varphi(x, y) \). We note that \( \varphi(x, y) \) depends on a computable function \( f \) and the predicates \( p_1, \ldots, p_k \), which occur in the rule body \( \varphi(x, y) \). Assume that \( \Delta_{p_1}, \ldots, \Delta_{p_k} \) are the answers collected so far for the predicates \( p_1, \ldots, p_k \). Let us consider a procedure \( \text{eval}(p, \Delta_{p_1}, \ldots, \Delta_{p_k}) \), which computes the set of answers \( \{\{x/c\}, v\} \) of \( p \), by evaluating the body \( \varphi(x, y) \) over the data provided by \( \Delta_{p_1}, \ldots, \Delta_{p_k} \). Formally, let \( I \) be an interpretation restricted to the predicates \( p_1, \ldots, p_k \) and tuples in such that for all \( n_i \)-ary predicates \( p_i \), \( I(p_i(c_i)) = v_i \) if \( (c_i, v_i) \in \Delta_{p_i} \) and \( I(p_i(c_i)) = 0 \) otherwise. Then

\[
\text{eval}(p, \Delta_{p_1}, \ldots, \Delta_{p_k}) = \{\{\{x/c\}, v\} \mid v = \max_{c'} I(\varphi(c, c'))\},
\]
where $c'$ is a tuple of constants occurring in $\bigcup_i \Delta_{p_i}$. For instance, consider Example 12. Assume that both $\Delta_{edge}$ and $\Delta_{path}$ are given by $tab_{edge}$. Then $\text{eval}(\text{path}, \Delta_{edge}, \Delta_{path})$ returns the set of answers $\Delta'_{path}$ shown below:

<table>
<thead>
<tr>
<th>$\Delta'_{path}$</th>
<th>a</th>
<th>a</th>
<th>0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>c</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>a</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>c</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>a</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>b</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Essentially, $\text{eval}(\text{path}, \Delta_{edge}, \Delta_{path})$ requires to compute for each tuple satisfying the body of rule (4) its truth degree with respect to the tables $tab_{edge}$ and $tab_{path}$. The tuples satisfying the body can be obtained using relational algebra:

$$\pi_{1,2}(\text{tab}_{\Delta_{edge}}) \cup \pi_{1,5}(\text{tab}_{\Delta_{edge}} \bowtie_{a=a} \text{tab}_{\Delta_{path}}).$$

In substance $\text{eval}$ revises the set of answers for $\text{path}$. Note that with respect to Example 12, $\text{tab}_{\Delta'_{path}}$ does not have the record $\langle c, c, 0.4 \rangle$, and the truth of $\langle a, a \rangle$ and $\langle b, b \rangle$ are smaller. Furthermore, we obtain the correct answers after reiterating the evaluation step once more. That is, it is easily verified that $\text{eval}(\text{path}, \Delta_{edge}, \Delta'_{path})$ returns all correct answers of $\text{path}$.

In the following, we are not going to detail the $\text{eval}(p, \Delta_{p_1}, \ldots, \Delta_{p_k})$ procedure, though it has to be carefully be written to minimize the table lookups and joins. It can be obtained by of means of a combination of SQL statements over the tables (similarly to how it works for Datalog) and the application of the functions occurring in the rule body of $p$. We point out that $\text{eval}(p, \Delta_{p_1}, \ldots, \Delta_{p_k})$ can also be seen as a query to a database made out by the relations $\text{tab}_{\Delta_{p_1}}, \ldots, \text{tab}_{\Delta_{p_k}}$ and that any successive evaluation step corresponds to the execution of the same query over an updated database. We refer the reader to e.g. [37, 38, 39, 70] concerning the problem of repeatedly evaluating the same query to a database that is being updated between successive query requests. In this situation, it may be possible to use the difference between successive database states and the answer to the query in one state to reduce the cost of evaluating the query in the next state.
We describe now our query answering procedure. Informally, our procedure works as Solve except that now the variables are the predicate symbols and the variable $v$ holds all the answers collected so far for the predicates. At each iteration step we select a new predicate $p$ from the queue $A$ and evaluate it using the $eval$ function with respect to the answers gathered so far. If the evaluation leads to a better answer set for $p$, we update the current answer set $v(p)$ and add all predicates $p'$, whose rule body contains $p$ (the parents of $p$), to the queue $A$.

The procedure is describe in Table 3. Some explanations of the procedure are in order:

- for predicate symbol $p_i$, $s(p_i)$ is the set of predicate symbols occurring in the rule body of $p_i$, i.e. the successors of $p_i$;
- for predicate symbol $p_i$, $p(p_i) = \{p_j : p_i \in s(p_j)\}$, i.e. the predecessors of $p_i$;
- in step 5, $p_{i_1}, \ldots, p_{i_{k_i}}$ are all predicate symbols occurring in the rule body of $p_i$, i.e. $s(p_i) = \{p_{i_1}, \ldots, p_{i_{k_i}}\}$;
- the variables $dg$, $exp$ and $in$ work as for the Solve procedure.

**Example 13** Consider Example 12. Let us compute all correct answers of predicate path. So, let $Q = \{\text{path}\}$. The execution of $Answer(L_{[0,1]} , P, Q)$ is shown in Table 4. Table 4 also reports $\Delta_{p_i}$ and $v(p_i)$ at each iteration $i$.  

---

**Table 3: General top-down algorithm. Non-grounded version.**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>$Answer(L, P, Q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>Complete lattice $L$, logic program $P$, set of query predicate symbols $Q$</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td>A mapping $v$ such that it contains all correct answers of predicates in $Q$.</td>
</tr>
</tbody>
</table>

1. $A := Q, \, dg := Q, \, in := \emptyset$, for all predicate symbols $p$ in $P$ do $v(p) = \emptyset, \, exp(p) = false$
2. while $A \neq \emptyset$ do
3. select $p_i \in A$, $A := A \setminus \{p_i\}$, $dg := dg \cup s(p_i)$
4. if ($p_i$ extensional predicate) $\land$ ($v(p_i) = \emptyset$) then $v(p_i) := \Delta_{p_i}$
5. if ($p_i$ intentional predicate) then $\Delta_{p_i} := eval(p_i, v(p_{i_1}), ..., v(p_{i_{k_i}}))$
6. if $\Delta_{p_i} \succ v(p_i)$ then $v(p_i) := \Delta_{p_i}$, $A := A \cup (p(p_i) \cap dg)$
7. if not $exp(p_i)$ then $exp(p_i) = true$, $A := A \cup (s(p_i) \setminus in)$, $in := in \cup s(p_i)$
8. od
1. \( A := \{\text{path}\}, p_i := \text{path}, A := \emptyset, \text{dg} := \{\text{path}, \text{edge}\}, \Delta_{\text{path}} := \emptyset \)
   \( \exp(\text{path}) := 1, A := \{\text{path}, \text{edge}\}, \text{in} := \{\text{path}, \text{edge}\} \)

2. \( p_i := \text{path}, A := \{\text{edge}\}, \Delta_{\text{path}} := \emptyset \)

3. \( p_i := \text{edge}, A := \emptyset, \Delta_{\text{edge}} \rightarrow \text{v(edge)}, \text{v(edge)} := \Delta_{\text{edge}}, A := \{\text{path}\}, \exp(\text{edge}) := 1 \)

4. \( p_i := \text{path}, A := \emptyset, \Delta_{\text{path}} \rightarrow \text{v(path)}, v(\text{path}) := \Delta_{\text{path}}, A := \{\text{path}\} \)

5. \( p_i := \text{path}, A := \emptyset, \Delta_{\text{path}} \rightarrow \text{v(path)}, v(\text{path}) := \Delta_{\text{path}}, A := \{\text{path}\} \)

6. \( p_i := \text{path}, A := \emptyset, \Delta_{\text{path}} \rightarrow \text{v(path)}, v(\text{path}) := \Delta_{\text{path}}, A := \{\text{path}\} \)

7. \( p_i := \text{path}, A := \emptyset, \Delta_{\text{path}} = \text{v(path)} \)

8. \text{stop. return } v(\text{path})

<table>
<thead>
<tr>
<th>Iter i</th>
<th>( \Delta_{p_i} )</th>
<th>v(p_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>-</td>
<td>\text{v(edge)} = v(\text{path}) = \emptyset</td>
</tr>
<tr>
<td>1.</td>
<td>( \Delta_{\text{path}} = \emptyset )</td>
<td>-</td>
</tr>
<tr>
<td>2.</td>
<td>( \Delta_{\text{path}} = \emptyset )</td>
<td>-</td>
</tr>
<tr>
<td>3.</td>
<td>( \Delta_{\text{edge}} = {\langle a, b, 0.3 \rangle, \langle b, a, 0.4 \rangle, \langle a, c, 0.5 \rangle, \langle c, b, 0.6 \rangle} )</td>
<td>\text{v(edge)} = \Delta_{\text{edge}}</td>
</tr>
<tr>
<td>4.</td>
<td>( \Delta_{\text{path}} = {\langle a, b, 0.3 \rangle, \langle b, a, 0.4 \rangle, \langle a, c, 0.5 \rangle, \langle c, b, 0.6 \rangle} )</td>
<td>\text{v(path)} = \Delta_{\text{path}}</td>
</tr>
<tr>
<td>5.</td>
<td>( \Delta_{\text{path}} = {\langle a, a, 0.3 \rangle, \langle a, b, 0.5 \rangle, \langle a, c, 0.5 \rangle, \langle b, a, 0.4 \rangle, \langle b, b, 0.3 \rangle, \langle b, c, 0.4 \rangle, \langle c, a, 0.4 \rangle, \langle c, b, 0.6 \rangle} )</td>
<td>\text{v(path)} = \Delta_{\text{path}}</td>
</tr>
<tr>
<td>6.</td>
<td>( \Delta_{\text{path}} = {\langle a, a, 0.4 \rangle, \langle a, b, 0.5 \rangle, \langle a, c, 0.5 \rangle, \langle b, a, 0.4 \rangle, \langle b, b, 0.4 \rangle, \langle b, c, 0.4 \rangle, \langle c, a, 0.4 \rangle, \langle c, b, 0.6 \rangle, \langle c, c, 0.4 \rangle} )</td>
<td>\text{v(path)} = \Delta_{\text{path}}</td>
</tr>
<tr>
<td>7.</td>
<td>( \Delta_{\text{path}} = {\langle a, a, 0.4 \rangle, \langle a, b, 0.5 \rangle, \langle a, c, 0.5 \rangle, \langle b, a, 0.4 \rangle, \langle b, b, 0.4 \rangle, \langle b, c, 0.4 \rangle, \langle c, a, 0.4 \rangle, \langle c, b, 0.6 \rangle, \langle c, c, 0.4 \rangle} )</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4: Top-down computation related to Example 13.
For completeness, we report the computation for Example 8.

**Example 14** Let us consider a similar example to Example 8, in which the intentional database is written as

\[
\begin{align*}
\text{Good\_driver}(x) & \leftarrow \text{Experience}(x) \land (0.5 \cdot \text{Risk}(x)) \\
\text{Risk}(x) & \leftarrow (0.8 \cdot \text{Young}(x)) \lor \\
& \quad (0.8 \cdot \text{Sport\_car}(x)) \lor \\
& \quad (\text{Experience}(x) \land (0.5 \cdot \text{Good\_driver}(x))) \lor \\
& \quad \text{PriorRisk}(x)
\end{align*}
\]

and the extensional database is

<table>
<thead>
<tr>
<th>tab_{\Delta_{\text{Experience}}}</th>
<th>tab_{\Delta_{\text{PriorRisk}}}</th>
<th>tab_{\Delta_{\text{Sport_car}}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>john 0.7</td>
<td>john 0.5</td>
<td>john 0.8</td>
</tr>
<tr>
<td>tim 0.6</td>
<td>tim 0.4</td>
<td>tim 0.1</td>
</tr>
<tr>
<td>elisa 0.3</td>
<td>elisa 0.1</td>
<td>elisa 0.6</td>
</tr>
</tbody>
</table>

Suppose we are interested in knowing all risk factors. So, let \( Q = \{\text{Risk}\} \). The execution of \( \text{Answer}(\mathcal{L}_{[0,1]Q}, \mathcal{P}, Q) \) is shown in Table 5. Table 5 also reports \( \Delta_{p_i} \) and \( v(p_i) \) at each iteration \( i \) (for ease, we use first letters of the predicates only).

Similarly to the grounded case, we have that:

**Proposition 6** Let \( \mathcal{L}, \mathcal{P} \) and \( Q \) be a truth space, a logic program and a query, respectively. Then there is a limit ordinal \( \lambda \) such that after \( |\lambda| \) steps \( \text{Answer}(\mathcal{L}, \mathcal{P}, Q) \) returns the set all of correct answers of \( \mathcal{P} \) with respect to the predicates in \( Q \).

From a computational point of view, the analysis is the same as for the *Solve* algorithm. It is worth noting that, as the propositional Example 9 shows, the computation may not terminate after a finite number of steps.
1. \( A := \{ R \}, p_i := R, A := \emptyset, dg := \{ R, Y, S, E, P, G \}, \Delta_R := \emptyset \)  
   \( \exp(R) := 1, A := \{ Y, S, E, P, G \}, \text{in} := \{ Y, S, E, P, G \} \)

2. \( p_i := Y, A := \{ S, E, P, G \}, \Delta_Y := \emptyset, \exp(Y) := 1 \)

3. \( p_i := S, A := \{ E, P, G \}, \Delta_S \succ v(S), v(S) := \Delta_S, A := \{ E, P, G, R \}, \exp(S) := 1 \)

4. \( p_i := E, A := \{ P, G, R \}, \Delta_E \succ v(E), v(E) := \Delta_E, \exp(E) := 1 \)

5. \( p_i := P, A := \{ G, R \}, \Delta_P \succ v(P), v(P) := \Delta_P, \exp(P) := 1 \)

6. \( p_i := G, A := \{ R \}, \Delta_G = \emptyset, \exp(G) := 1, \text{in} := \{ Y, S, E, P, G, R \} \)

7. \( p_i := R, A := \emptyset, \Delta_R \succ v(R), v(R) := \Delta_R, A := \{ G \} \)

8. \( p_i := G, A := \emptyset, \Delta_G \succ v(G), v(G) := \Delta_G, A := \{ R \} \)

9. \( p_i := R, A := \emptyset, \Delta_R = v(R) \)

10. \textbf{stop. return} \( v(R) \)

<table>
<thead>
<tr>
<th>Iter i</th>
<th>( \Delta_{p_i} )</th>
<th>( v(p_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>( \Delta_R = \emptyset )</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>( \Delta_Y = \emptyset )</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>( \Delta_S = {(j, 0.8), (t, 0.1), (e, 0.6)} )</td>
<td>( v(S) = \Delta_S )</td>
</tr>
<tr>
<td>4</td>
<td>( \Delta_E = {(j, 0.7), (t, 0.6), (e, 0.3)} )</td>
<td>( v(E) = \Delta_E )</td>
</tr>
<tr>
<td>5</td>
<td>( \Delta_P = {(j, 0.5), (t, 0.4), (e, 0.1)} )</td>
<td>( v(P) = \Delta_P )</td>
</tr>
<tr>
<td>6</td>
<td>( \Delta_G = \emptyset )</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>( \Delta_R = {(j, 0.64), (t, 0.4), (e, 0.48)} )</td>
<td>( v(R) = \Delta_R )</td>
</tr>
<tr>
<td>8</td>
<td>( \Delta_G = {(j, 0.32), (t, 0.2), (e, 0.24)} )</td>
<td>( v(G) = \Delta_G )</td>
</tr>
<tr>
<td>9</td>
<td>( \Delta_R = {(j, 0.64), (t, 0.4), (e, 0.48)} )</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5: Top-down computation related to Example 14.
4 Conclusions

We have presented a simple, general, yet effective top-down algorithm to retrieve all correct answers to queries for logic programs over lattices with arbitrary continuous functions in the body to manipulate truth values. We believe that its interest relies on its easiness for an effective implementation and on avoiding the non-termination problem of typical SLD-based resolution methods.

As immediate future activity we try to extend the algorithm to deal with non-monotonicity. We plan to rely on [120] as starting point, which is based on an extension of the Solve algorithm.

Disclaimer

The list of references above is by no means intended to be all-inclusive. The authors of this overview apologizes both with the authors and with the readers for all the relevant works which are not cited here.

References


Alex Dekhtyar, Michael I. Dekhtyar, and V. S. Subrahmanian. Temporal probabilistic logic programs. In Danny De Schreye, editor, Logic


34


[90] Zoran Majkic. Truth and knowledge fixpoint semantics for many-valued logic programming. In *19th Workshop on (Constraint) Logic Programming ((W(C)LP-05)*, number 76-87, Ulm, Germany, 2005.


