Uncertainty Management in Logic Programming: Simple and Effective Top-Down Query Answering

Umberto Straccia
ISTI - CNR, Via G. Moruzzi, 1 I-56124 Pisa (PI) ITALY

Abstract. We present a simple, yet general top-down query answering procedure for logic programs managing uncertainty. The main features are: (i) the certainty values are taken from a certainty lattice; (ii) computable functions may appear in the rule bodies to manipulate certainties; and (iii) we solve the problem by a reduction to an equational systems, for which we device a top-down procedure.

1 Introduction

The management of uncertainty in deduction systems is an important issue whenever the real world information to be represented is of imperfect nature (which is likely the rule rather than an exception). Classical logic programming, with its advantage of modularity and its powerful top-down and bottom-up query processing techniques, has attracted the attention of researchers and numerous frameworks have been proposed towards the management of uncertainty. Essentially, they differ in the underlying notion of uncertainty [1] (e.g. probability theory [10, 11], fuzzy set theory [13], multi-valued logic [3, 6, 7, 9], possibilistic logic [5]) and how uncertainty values, associated to rules and facts, are managed. Roughly, these frameworks can be classified into *annotation based* (AB) and *implication based* (IB). In the AB approach (e.g. [6, 11]), a rule is of the form $A\colon f(\beta_1, \ldots, \beta_n) \leftarrow B_1\colon \beta_1, \ldots, B_n\colon \beta_n$, which asserts "the certainty of atom A is at least (or is in) $f(\beta_1, \ldots, \beta_n)$, whenever the certainty of atom B_i is at least (or is in) β_i, $1 \leq i \leq n$". Here f is an n-ary computable function and β_i is either a constant or a variable ranging over an appropriate certainty domain. In the IB approach (see [3, 7] for a more detailed comparison between the two approaches), a rule is of the form $A \stackrel{\alpha}{\leftarrow} B_1, ..., B_n$, which says that the certainty associated with the implication $B_1 \wedge ... \wedge B_n \rightarrow A$ is α. Computationally, given an assignment v of certainties to the B_i, the certainty of A is computed by taking the "conjunction" of the certainties $v(B_i)$ and then somehow "propagating" it to the rule head. The truth-values are taken from a certainty lattice. More recently, [3, 7, 13] show that most of the frameworks can be embedded into the IB framework (some exceptions deal with probability theory). Usually, in order to answer to a query in such frameworks, we have to compute the whole intended model by a bottom-up fixed-point computation and then answer with the evaluation of the query in this model. This always requires to compute a whole model, even if not all the atoms truth is required to determine the answer. To the best of our knowledge the only work presenting top-down procedures are [4, 6, 7, 13].

In this paper we present a general, simple and effective top-down query answering procedure for logic programs over lattices in the IB framework, which generalizes the above cited works. The main features are: (i) the certainty values are taken from a certainty lattice; (ii) computable functions may appear in the rule bodies to manage these certainties values; and (iii) we solve the problem by a reduction to an equational systems over lattices, for which we device a top-down procedure, which to the best of our knowledge is novel.

[1] See e.g. [12] for an extensive list of references

We proceed as follows. In the next section, we will briefly recall some preliminary definitions. Section 3 is the main part of this work, where we present our top-down query procedure and the computational complexity analysis, while Section 4 concludes.

2 Preliminaries

Certainty lattice. A *certainty lattice* is a complete lattice $\mathcal{L} = \langle L, \preceq \rangle$, with L a countable set of certainty values, bottom $\bot$, top element $\top$, meet $\wedge$ and join $\vee$. The main idea is that an statement $P(a)$, rather than being interpreted as either true or false, will be mapped into a certainty value c in L. The intended meaning is that c indicates to which extend (how *certain* it is that) $P(a)$ is true. Typical certainty lattices are the following. (i) Classical 0-1: $\mathcal{L}_{\{0,1\}}$ corresponds to the classical truth-space, where 0 stands for 'false', while 1 stands for 'true'. (ii) Fuzzy: $\mathcal{L}_{[0,1]_\mathbb{Q}}$, which relies on the unit real interval, is quite frequently used as certainty lattice. (iii) Four-valued: another frequent certainty lattice is Belnap's $\mathcal{FOUR}$ [1], 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 $\mathcal{L}_B$. (iv) Many-valued: $L = \langle \{0, \frac{1}{n-1}, \ldots \frac{n-2}{n-1}, 1\}, \leq \rangle$, n positive integer. A special case is $\mathcal{L}_4$, where L is $\{f, lf, lt, t\}$ with $f \preceq lf \preceq lt \preceq t$. Here, lf stands for 'likely false', whereas lt stands for 'likely true'. (v) Belief-Doubt: a further popular lattice allows us to reason about *belief and doubt*. Indeed, the idea is to take any lattice $\mathcal{L}$, and to consider the cartesian product $\mathcal{L} \times \mathcal{L}$. For any pair $(b, d) \in \mathcal{L} \times \mathcal{L}$, 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 $\mathcal{L} \times \mathcal{L}$ 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 (f, t) (no belief, maximal doubt), while the maximal element is (t, f) (maximal belief, no doubt). We indicate this lattice with $\bar{\mathcal{L}}$.

In a complete lattice $\mathcal{L} = \langle L, \preceq \rangle$, a function $f\colon L \to L$ is *monotone*, if $\forall x, y \in L$, $x \preceq y$ implies $f(x) \preceq f(y)$. A *fixed-point* of f is an element $x \in L$ such that $f(x) = x$. The basic tool for studying fixed-points of functions on lattices is the well-known Knaster-Tarski theorem. Let f be a monotone function on a complete lattice $\langle L, \preceq \rangle$. Then f has a fixed-point, the set of fixed-points of f is a complete lattice and, thus, f has a *least* fixed-point. The *least* fixed-point can be obtained by iterating f over $\bot$, i.e. is the limit of the non-decreasing sequence $y_0, \ldots, y_i, y_{i+1}, \ldots, y_\lambda, \ldots$, where for a successor ordinal $i \geq 0$, $y_0 = \bot$, $y_{i+1} = f(y_i)$, while for a limit ordinal λ, $y_\lambda = lub_\preceq\{y_i \colon i < \lambda\}$. We denote the least fixed-point by $\mathrm{lfp}(f)$. For ease, we will specify the initial condition y_0 and the next iteration step y_{i+1} only, while the condition on the limit is implicit.

Logic programs. Fix a lattice $\mathcal{L} = \langle L, \preceq \rangle$. We extend standard logic programs [8] to the case where *arbitrary computable functions* $f \in \mathcal{F}$ are allowed in rule bodies to manipulate the certainty values. In this paper we assume that $\mathcal{F}$ is a family of continuous n-ary functions $f\colon \mathcal{L}^n \to \mathcal{L}$. That is, for any monotone chain $x_0, x_1, \ldots$ of values in L, $f(\vee_i x_i) = \vee_i f(x_i)$. The n-ary case $n > 1$ is similar. We assume that the standard functions $\wedge$ and $\vee$ belong to $\mathcal{F}$. Notably, $\wedge$ and $\vee$ are both continuous. For reasons of space, we limit our attention to propositional logic programs. The first order case can be handled by grounding. There exists free software (e.g. Lparse), which transforms a logic program with variables into one with propositional variables only. So, consider an alphabet of propositional letters. An *atom*, denoted A is a propositional letter. A *formula*, φ, is an expression built up from the atoms, the certainty values $c \in L$ of the lattice and the functions $f \in \mathcal{F}$. Note that members of the lattice may appear in a formula, as well as functions: e.g. in $\mathcal{L}_{[0,1]_\mathbb{Q}}$, $\varphi = \min(p, q) \cdot \max(r, 0.7) + v$ is a formula, where p, q, r and v are atoms. The

intuition here is that the truth value of the formula $\min(p,q)\cdot\max(r,0.7)+v$ is obtained by determining the truth value of p, q, r and v and then to apply the arithmetic functions to determine the value of φ. A *rule* is of the form $A \leftarrow \varphi$, where A is an atom and φ is a formula. The atom A is called the *head*, and the formula φ is called the *body*. A *logic program*, denoted with $\mathcal{P}$, is a finite set of rules. The *Herbrand base* of $\mathcal{P}$ (denoted $B_\mathcal{P}$) is the set of atoms occurring in $\mathcal{P}$. Given $\mathcal{P}$, the set $\mathcal{P}^*$ is constructed as follows; (i) if an atom A is not head of any rule in $\mathcal{P}^*$, then add the rule $A \leftarrow \bot$ to $\mathcal{P}^*$; [2] and (ii) replace several rules in $\mathcal{P}^*$ having same head, $A \leftarrow \varphi_1$, $A \leftarrow \varphi_2$, ... with $A \leftarrow \varphi_1 \vee \varphi_2 \vee \ldots$. Note that in $\mathcal{P}^*$, each atom appears in the head of *exactly one* rule.

Example 1 ([9]) *Consider $\mathcal{L}_{[0,1]_\mathbb{Q}}$, where $\wedge = \min$ and $\vee = \max$. 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 $\{(\texttt{Experience(john)} \leftarrow 0.7, (\texttt{Risk(john)} \leftarrow 0.5, (\texttt{Sport\_car(john)} \leftarrow 0.8\}$; and (ii) a set of rules, which after grounding are:*

$$\begin{array}{ll}
\texttt{Good_driver(john)} & \leftarrow \texttt{Experience(john)} \wedge (0.5 \cdot \texttt{Risk(john)}) \\
\texttt{Risk(john)} & \leftarrow 0.8 \cdot \texttt{Young(john)} \\
\texttt{Risk(john)} & \leftarrow 0.8 \cdot \texttt{Sport_car(john)} \\
\texttt{Risk(john)} & \leftarrow \texttt{Experience(john)} \wedge (0.5 \cdot \texttt{Good_driver(john)})
\end{array}$$

Interpretations. An *interpretation I of a logic program* on the lattice $\mathcal{L} = \langle L, \preceq \rangle$ is a mapping from atoms to members of L. I is extended from atoms to formulae as follows: (i) for $c \in L$, $I(c) = c$; (ii) for formulae φ and φ', $I(\varphi \wedge \varphi') = I(\varphi) \wedge I(\varphi')$, and similarly for $\vee$; and (iii) for formulae $f(\varphi)$, $I(f(\varphi)) = f(I(\varphi))$, and similarly for n-ary functions. The ordering $\preceq$ is extended from $\mathcal{L}$ to the set $\mathcal{I}(\mathcal{L})$ of all interpretations point-wise: (i) $I_1 \preceq I_2$ iff $I_1(A) \preceq I_2(A)$, for every ground atom A. We define $(I_1 \wedge I_2)(A) = I_1(A) \wedge I_2(A)$ and similarly for $\vee$. With $\mathtt{I}_\bot$ we denote the bottom interpretation under $\preceq$ (it maps any atom into $\bot$). It is easy to see that $\langle \mathcal{I}(\mathcal{L}), \preceq \rangle$ is a complete lattice as well.

Models. An interpretation I is a *model* of a logic program $\mathcal{P}$, denoted by $I \models \mathcal{P}$, iff for all $A \leftarrow \varphi \in \mathcal{P}^*$, $I(\varphi) \preceq I(A)$ holds.

Query. A *query*, denoted q, 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 $\mathcal{P}$ (see below). 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 of $\mathcal{P}$.

Semantics of logic programs. The semantics of a logic program $\mathcal{P}$ is determined by the least model of $\mathcal{P}$, $M_\mathcal{P} = \min\{I : I \models \mathcal{P}\}$. The *existence and uniqueness* of $M_\mathcal{P}$ is guaranteed by the fixed-point characterization, by means of the *immediate consequence operator* $\Phi_\mathcal{P}$. For an interpretation I, for any ground atom A, $\Phi_\mathcal{P}(I)(A) = I(\varphi)$, where $A \leftarrow \varphi \in \mathcal{P}^*$. We can show that the function $\Phi_\mathcal{P}$ is continuous over $\mathcal{I}(\mathcal{L})$, the set of fixed-points of $\Phi_\mathcal{P}$ is a complete lattice under $\preceq$ and, thus, $\Phi_\mathcal{P}$ has a least fixed-point and I is a model of a program $\mathcal{P}$ iff I is a fixed-point of $\Phi_\mathcal{P}$. Therefore, the minimal model of $\mathcal{P}$ coincides with the least fixed-point of $\Phi_\mathcal{P}$, which can be computed in the usual way by iterating $\Phi_\mathcal{P}$ over $\mathtt{I}_\bot$ and is attained after at most ω (the least limit ordinal) iterations.

Example 2 *Consider $\mathcal{L}_{[0,1]_\mathbb{Q}}$, the function $f(x) = \frac{x+a}{2}$ ($0 < a \leq 1, a \in \mathbb{Q}$), and $\mathcal{P} = \{A \leftarrow f(A)\}$. Then the minimal model is attained after ω steps of $\Phi_\mathcal{P}$ iterations starting from $\mathtt{I}_\bot(A) = 0$ and is $M_\mathcal{P}(A) = a$.*

[2] It is a standard practice in logic programming to consider such atoms as *false*.

Example 3 *Consider Example 1. It turns out that by a bottom-up computation the minimal mode is $M_{\mathcal{P}}$, where (for ease, we use first letters only)* $M_{\mathcal{P}}(\mathtt{R(j)}) = 0.64$, $M_{\mathcal{P}}(\mathtt{S(j)}) = 0.8$, $M_{\mathcal{P}}(\mathtt{Y(j)}) = 0$, $M_{\mathcal{P}}(\mathtt{G(j)}) = 0.32$, $M_{\mathcal{P}}(\mathtt{E(j)}) = 0.7$.

3 Top-down query answering

Given a logic program $\mathcal{P}$, one way to answer to a query $?A$ is to compute the minimal model $M_{\mathcal{P}}$ of $\mathcal{P}$ by a bottom-up fixed-point computation and then answer with $M_{\mathcal{P}}(A)$. This always requires to compute a whole model, even if in order to determine $M_{\mathcal{P}}(A)$, not all the atom's truth is required. Our goal is to present a general, simple, yet effective top-down method, which relies on the computation of just a part of the minimal model. Essentially, we will try to determine the value of a single atom by investigating only a part of the program $\mathcal{P}$. Our method is based on a transformation of a program into a system of equations of monotonic functions over lattices for which we compute the least fixed-point in a top-down style. The idea is the following. Let $\mathcal{L} = \langle L, \preceq \rangle$ be a lattice and let $\mathcal{P}$ be a logic program. Consider the Herbrand base $B_{\mathcal{P}} = \{A_1, \ldots, A_n\}$ of $\mathcal{P}$ and consider $\mathcal{P}^*$. Let us associate to each atom $A_i \in B_{\mathcal{P}}$ 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 lattice values to the variables $x_1, ..., x_n$. For the immediate consequence operator $\Phi_{\mathcal{P}}$, a fixed-point is such that $I = \Phi_{\mathcal{P}}(I)$, i.e. for all atoms $A_i \in B_{\mathcal{P}}$, $I(A_i) = \Phi_{\mathcal{P}}(I)(A_i)$. Therefore, we may identify the fixed-points of $\Phi_{\mathcal{P}}$ as the solutions over $\mathcal{L}$ of the system of equations of the following form:

$$\begin{aligned} x_1 &= f_1(x_{1_1}, \ldots, x_{1_{a_1}}) \,, \\ &\vdots \\ x_n &= f_n(x_{n_1}, \ldots, x_{n_{a_n}}) \,, \end{aligned} \tag{1}$$

where for $1 \leq i \leq n$, $1 \leq k \leq a_i$, we have $1 \leq i_k \leq n$. Each variable x_{i_k} will take a value in the domain L, each (continuous) function f_i determines the value of x_i (i.e. A_i) given an assignment $I(A_{i_k})$ to each of the a_i variables x_{i_k}. The function f_i implements $\Phi_{\mathcal{P}}(I)(A_i)$. For instance, by considering the logic program in Example 1, the fixed-points of the $\Phi_{\mathcal{P}}$ operator are the solutions over a lattice of the system of equations

$$\begin{aligned} &x_{\mathtt{E(j)}} = 0.7 \,,\; x_{\mathtt{S(j)}} = 0.8 \,,\; x_{\mathtt{Y(j)}} = 0 \,,\; x_{\mathtt{G(j)}} = \min\{x_{\mathtt{E(j)}}, 0.5 \cdot x_{\mathtt{R(j)}}\}, \\ &x_{\mathtt{R(j)}} = \max\{0.5, 0.8 \cdot x_{\mathtt{Y(j)}}, 0.8 \cdot x_{\mathtt{S(j)}}, \min\{x_{\mathtt{E(j)}}, 0.5 \cdot x_{\mathtt{G(j)}}\}\} \,. \end{aligned} \tag{2}$$

It is easily verified that the least solution corresponds to the minimal model of $\mathcal{P}$. Therefore, our general approach for query answering is as follows: given a logic program $\mathcal{P}$, translate it into an equational system as (1) and then compute the answer in a top-down manner. Formally, let $\mathcal{P}$ be a logic program and consider $\mathcal{P}^*$. As already pointed out, each atom appears exactly once in the head of a rule in $\mathcal{P}^*$. The system of equations that we build from $\mathcal{P}^*$ is straightforward. Assign to each atom A a variable x_A and substitute in $\mathcal{P}^*$ each occurrence of A with x_A. Finally, substitute each occurrence of $\leftarrow$ with $=$ and let $\mathcal{S}(\mathcal{P})$ be the resulting equational system (see Equation 2). The answer of a query variable $?A$ w.r.t. a logic program $\mathcal{P}$ is computed by the algorithm $Solve(\mathcal{P}, ?A)$. It first computes $\mathcal{S}(\mathcal{P})$ and then calls $Solve(\mathcal{S}(\mathcal{P}), \{x_A\})$, which will solve the equational system answering with the value for x_A. Therefore, query answering in logic programs reduces to query answering in equational monotone systems of the form (1), which we address next. We refer to the monotone system as in Equation (1) as the tuple $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$, where $\mathcal{L}$ is a lattice, $V = \{x_1, ..., x_n\}$ are the variables and $\boldsymbol{f} = \langle f_1, ..., f_n \rangle$ is the tuple of functions. As it is well known, a monotonic equation system as (1) has a least solution, $\mathrm{lfp}(\boldsymbol{f})$,

which can be computed by a bottom-up evaluation. Indeed, the least fixed-point of $\boldsymbol{f}$ is given as the least upper bound of the monotone sequence, $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_i, \ldots$, where $\boldsymbol{y}_0 = \bot$ and $\boldsymbol{y}_{i+1} = \boldsymbol{f}(\boldsymbol{y}_i)$.

Our top-down procedure needs some auxiliary functions. $\mathtt{s}(x)$ denotes the set of *sons* of x, i.e. $\mathtt{s}(x_i) = \{x_{i_1}, \ldots, x_{i_{a_i}}\}$ (the set of variables appearing in the right hand side of the definition of x_i). $\mathtt{p}(x)$ denotes the set of *parents* of x, i.e. the set $\mathtt{p}(x) = \{x_i \colon x \in \mathtt{s}(x_i)\}$ (the set of variables depending on the value of x). In the general case, we assume that each function $f_i \colon L^{a_i} \mapsto L$ in Equation (1) is monotone. We also use f_x in place of f_i, for $x = x_i$. Informally our algorithm works as follows. Assume we are interested in the value of x_0 in the least fixed-point of the system. We associate to each variable x_i a marking $\mathtt{v}(x_i)$ denoting the current value of x_i (the mapping $\mathtt{v}$ contains the current value associated to the variables). Initially, $\mathtt{v}(x_i)$ is $\bot$. We start with putting x_0 in the *active* list of variables $\mathtt{A}$, 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 $\bot$ 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 (actually, it can only increase) 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. The algorithm is given below.

Procedure $Solve(\mathcal{S}, Q)$
Input: monotonic system $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$, where $Q \subseteq V$ is the set of query variables;
Output: A set $B \subseteq V$, with $Q \subseteq B$ such that the mapping $\mathtt{v}$ equals $\mathrm{lfp}(f)$ on B.

1. $\mathtt{A} := Q$, $\mathtt{dg} := Q$, $\mathtt{in} := \emptyset$, **for all** $x \in V$ **do** $\mathtt{v}(x) = \bot$, $\mathtt{exp}(x) = 0$
2. **while** $\mathtt{A} \neq \emptyset$ **do**
3. **select** $x_i \in \mathtt{A}$, $\mathtt{A} := \mathtt{A} \setminus \{x_i\}$, $\mathtt{dg} := \mathtt{dg} \cup \mathtt{s}(x_i)$
4. $r := f_i(\mathtt{v}(x_{i_1}), ..., \mathtt{v}(x_{i_{a_i}}))$
5. **if** $r \succ \mathtt{v}(x_i)$ **then** $\mathtt{v}(x_i) := r$, $\mathtt{A} := \mathtt{A} \cup (\mathtt{p}(x_i) \cap \mathtt{dg})$ **fi**
6. **if not** $\mathtt{exp}(x_i)$ **then** $\mathtt{exp}(x_i) = 1$, $\mathtt{A} := \mathtt{A} \cup (\mathtt{s}(x_i) \setminus \mathtt{in})$, $\mathtt{in} := \mathtt{in} \cup \mathtt{s}(x_i)$ **fi**

od

The variable dg collects the variables that may influence the value of the query variables, the array variable exp traces the equations that has been "expanded" (body variables are put into the active list), while 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). The attentive reader will notice that the $Solve$ procedure is related to the so-called *tabulation* procedures, like [2, 4]. Indeed, it is a generalization of it to arbitrary monotone equational systems over lattices.

Example 4 *Consider Example 1 and query variable $x_{R(j)}$ (we ask for the risk coefficient of John). Below is a sequence of $Solve(\mathcal{S}, \{x_{R(j)}\})$ computation. Each line is a sequence of steps in the 'while loop'. What is left unchanged is not reported.*

1. $\mathtt{A} := \{x_{\mathtt{R(j)}}\}, x_i := x_{\mathtt{R(j)}}, \mathtt{A} := \emptyset, \mathtt{dg} := \{x_{\mathtt{R(j)}}, x_{\mathtt{Y(j)}}, x_{\mathtt{S(j)}}, x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}\}, r := 0.5, \mathtt{v}(x_{\mathtt{R(j)}}) := 0.5,$
   $\mathtt{A} := \{x_{\mathtt{G(j)}}\}, \mathtt{exp}(x_{\mathtt{R(j)}}) := 1, \mathtt{A} := \{x_{\mathtt{Y(j)}}, x_{\mathtt{S(j)}}, x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}\}, \mathtt{in} := \{x_{\mathtt{Y(j)}}, x_{\mathtt{S(j)}}, x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}\}$
2. $x_i := x_{\mathtt{Y(j)}}, \mathtt{A} := \{x_{\mathtt{S(j)}}, x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}\}, r := 0, \mathtt{exp}(x_{\mathtt{Y(j)}}) := 1$
3. $x_i := x_{\mathtt{S(j)}}, \mathtt{A} := \{x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}\}, r := 0.8, \mathtt{v}(x_{\mathtt{S(j)}}) := 0.8, \mathtt{A} := \{x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}, x_{\mathtt{R(j)}}\}, \mathtt{exp}(x_{\mathtt{S(j)}}) := 1$
4. $x_i := x_{\mathtt{E(j)}}, \mathtt{A} := \{x_{\mathtt{G(j)}}, x_{\mathtt{R(j)}}\}, r := 0.7, \mathtt{v}(x_{\mathtt{E(j)}}) := 0.7, \mathtt{exp}(x_{\mathtt{E(j)}}) := 1$
5. $x_i := x_{\mathtt{G(j)}}, \mathtt{A} := \{x_{\mathtt{R(j)}}\}, r := 0.25, \mathtt{v}(x_{\mathtt{G(j)}}) := 0.25, \mathtt{exp}(x_{\mathtt{G(j)}}) := 1,$
   $\mathtt{in} := \{x_{\mathtt{Y(j)}}, x_{\mathtt{S(j)}}, x_{\mathtt{E(j)}}, x_{\mathtt{G(j)}}, x_{\mathtt{R(j)}}\}$

6. $x_i := x_{\mathtt{R(j)}}, \mathtt{A} := \emptyset, r := 0.64, \mathtt{v}(x_{\mathtt{R(j)}}) := 0.64, \mathtt{A} := \{x_{\mathtt{G(j)}}\}$
7. $x_i := x_{\mathtt{G(j)}}, \mathtt{A} := \emptyset, r := 0.32, \mathtt{v}(x_{\mathtt{G(j)}}) := 0.32, \mathtt{A} := \{x_{\mathtt{R(j)}}\}$
8. $x_i := x_{\mathtt{G(j)}}, \mathtt{A} := \emptyset, r := 0.64$
10. `stop`. `return` v(*in particular*, $\mathtt{v}(x_{\mathtt{R(j)}}) = 0.64$)

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

Given $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$, where $\mathcal{L} = \langle L, \preceq \rangle$, let $h(\mathcal{L})$ be the *height* of the truth-value set L, i.e. the length of the longest strictly increasing chain in L minus 1, where the length of a chain $v_1, ..., v_\alpha, ...$ is the cardinal $|\{v_1, ..., v_\alpha, ...\}|$. The *cardinal* of a countable set X is the least ordinal α such that α and X are *equipollent*, i.e. there is a bijection from α to X. For instance, $h(\mathcal{FOUR}) = 2$, while $h(\mathcal{L}_{[0,1]_\mathbb{Q}}) = \omega$. It can be shown that the above algorithms answer correctly.

Proposition 5 *Given monotone $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$, then there is a limit ordinal λ such that after $|\lambda|$ steps $Solve(\mathcal{S}, Q)$ determines a set $B \subseteq V$, with $Q \subseteq B$ such that the mapping* v *equals lfp($\boldsymbol{f}$) on B, i.e. $\mathtt{v}_{|B} = lfp(\boldsymbol{f})_{|B}$. As a consequence, let $\mathcal{P}$ and $?A$ be a logic program and a query, respectively. Then $M_\mathcal{P}(A) = Solve(\mathcal{P}, \{?A\})$* [3].

From a computational point of view, by means of appropriate data structures, the operations on `A`, `v`, `dg`, `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 $h(\mathcal{L})$ of the lattice $\mathcal{L}$ 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 $a_i \cdot h(\mathcal{L}) + 1$ times, where a_i is the arity of f_i, as a variable is only re-entered into `A` if one of its son gets an increased value (which for each son only can happen $h(\mathcal{L})$ times), plus the additional entry due to Step 6. As a consequence, the worst-case complexity is $O(\sum_{x_i \in V}(c(f_i) \cdot (a_i \cdot h(\mathcal{L}) + 1))$. Therefore:

Proposition 6 *Given monotone $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$, where the computing cost of each function in $\boldsymbol{f}$ 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 2). Fortunately, under reasonable assumptions on the functions, we may guarantee the termination of $Solve$ (see [12]). For instance, a condition that guarantees the termination of $Solve$ is inspired directly by [3]. On lattices, we say that a function $f\colon L^n \to L$ is *bounded* iff $f(x_1, \ldots, x_n) \preceq \wedge_i x_i$. Now, consider a monotone system of equations $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$. We say that $\boldsymbol{f}$ is *bounded* iff each f_i is a composition of functions, each of which is either bounded, or a constant in L or one of $\vee$ and $\wedge$. For instance, the function in Example 2 is not bounded, while $f(x, y) = \max(0, x+y-1) \wedge 0.3$ over $\mathcal{L}_{[0,1]_\mathbb{Q}}$ is. It can be shown that

Proposition 7 *Given monotone $\mathcal{S} = \langle \mathcal{L}, V, \boldsymbol{f} \rangle$ with $\boldsymbol{f}$ bounded, then $Solve$ terminates.*

Concerning the special case were the equational system is directly obtained from the translation of a logic program, we can avoid the cost of translating $\mathcal{P}$ into $\mathcal{S}(\mathcal{P})$ as we can

[3] The extension to a set of query atoms is straightforward.

directly operate on $\mathcal{P}$. So the cost $O(|\mathcal{P}|)$ can be avoided. In case the height of the lattice is finite, from Proposition 6 it follows immediately that the worst-case complexity for top-down query answering is $O(|B_{\mathcal{P}}|cah)$. Furthermore, often the cost of computing each of the functions of $\boldsymbol{f}_{\mathcal{P}}$ is in $O(1)$. By observing that $|B_{\mathcal{P}}|a$ is in $O(|\mathcal{P}|)$ we immediately have that in this case the complexity is $O(|\mathcal{P}|h)$. It follows that over the lattice $\mathcal{FOUR}$ ($h = 2$) the top-down algorithm works in *linear time*. Moreover, if 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) 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 7. In general, the continuity of the functions in $\mathcal{S}(\mathcal{P})$ guarantees the termination after at most ω steps.

4 Conclusions

We have presented a simple, general, yet effective top-down algorithm to answer queries for logic programs over lattices with arbitrary continues functions in the body to manipulate uncertainty values. We believe that its interest relies on its easiness for an effective implementation and the fact that many approaches to uncertainty management in logic programming are based on lattices, respectively.

References

1. N. D. Belnap. A useful four-valued logic. In G. Epstein and J. Michael Dunn, editors, *Modern uses of multiple-valued logic*, pages 5–37. Reidel, Dordrecht, NL, 1977.
2. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
3. C. Viegas Damásio, J. Medina, and M. Ojeda Aciego. Sorted multi-adjoint logic programs: Termination results and applications. In LNCS 3229, pages 252–265. Springer Verlag, 2004.
4. C. Viegas Damásio, J. Medina, and M. Ojeda Aciego. A tabulation proof procedure for residuated logic programming. In *Proc. of European Conf. on Artificial Intelligence (ECAI-04)*, 2004.
5. D. Dubois, J. Lang, and H. Prade. Towards possibilistic logic programming. In *Proc. of the 8th Int. Conf. on Logic Programming (ICLP-91)*, pages 581–595. The MIT Press, 1991.
6. M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.
7. L. V.S. Lakshmanan and N. Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001.
8. J. W. Lloyd. *Foundations of Logic Programming*. Springer, Heidelberg, RG, 1987.
9. Y. Loyer and U. Straccia. The approximate well-founded semantics for logic programs with uncertainty. In LNCS 2747, pages 541–550, 2003. Springer-Verlag.
10. T. Lukasiewicz. Probabilistic logic programming. In *Proc. of the 13th European Conf. on Artificial Intelligence (ECAI-98)*, pages 388–392, 1998.
11. R. Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1993.
12. U. Straccia. Top-down query answering for logic programs over bilattices. Technical Report 2004-TR-62, Istituto di Scienza e Tecnologie dell'Informazione, CNR, Pisa, Italy, 2004.
13. P. Vojtáš. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124:361–370, 2004.