Approximations and Refinements of Certain Answers via Many-Valued Logics

Marco Console
Sapienza Università di Roma

Paolo Guagliardo and **Leonid Libkin**
University of Edinburgh

Abstract

Computing certain answers is the preferred way of answering queries in scenarios involving incomplete data. This, however, is computationally expensive, so practical systems use efficient techniques based on a particular three-valued logic, although this often leads to incorrect results.

Our goal is to provide a general many-valued framework for correctly approximating certain answers. We do so by defining the semantics of many-valued answers and queries, following the principle that additional knowledge about the input must translate into additional knowledge about the output. This framework lets us compare query outputs and evaluation procedures in terms of their informativeness. For each many-valued logic with a knowledge ordering on its truth values, one can build a syntactic evaluation procedure for all first-order queries that correctly approximates certain answers; additional truth values are used to refine information about certain answers. For concrete examples, we show that a recently proposed approach fixing some of the inconsistencies of SQL query evaluation is an immediate consequence of our framework, and we also refine it by adding a fourth truth value. We show that many-valuedness is essential: no evaluation procedure based on the usual two-valued logic delivers correctness guarantees. Finally, we study the relative power of evaluation procedures based on the informativeness of the answers they produce.

1 Introduction

In a variety of data processing tasks, especially those where knowledge representation techniques play an important role, query answering is often reduced to finding those answers that hold with certainty over an incompletely described data set. Such examples include data integration (Lenzerini 2002; Halevy, Rajaraman, and Ordille 2006), data exchange (Arenas et al. 2014), inconsistent databases (Bertossi 2011), and ontology-based query answering (Calvanese et al. 2007; Kontchakov et al. 2011). In many cases, one builds a database, typically incomplete, and then answers queries over it. The latter of course is often done using traditional database query evaluation methods.

When it comes to evaluation of queries over incomplete databases, we have a well established theory, as well as practice of relational query answering; both date back more than 30 years, and somehow they are totally out of sync with each other, as extensively discussed by Libkin (2014). The standard theoretical approach to answering queries on incomplete databases is to compute *certain answers*, i.e., those answers one can be sure about for every substitution of missing values. The main drawback of this approach is its high computational complexity outside restricted classes of queries (Abiteboul, Kanellakis, and Grahne 1991).

In practice, of course, high complexity cannot be tolerated, so systems such as relational DBMSs devise efficient algorithms that are based on a *three-valued logic* (3VL) for handling incomplete data (Date and Darwen 1996). In addition to the usual *true* (**t**) and *false* (**f**), 3VL operates with the value *unknown* (**u**) to account for operations with incomplete information.

While one might expect that the approaches accepted in practice have something to do with theory, it is not so: standard relational query evaluation based on 3VL and certain answers can be completely unrelated. This was observed recently by Libkin (2016b), who also showed that a modified query evaluation procedure can provide certainty guarantees at least for all first-order (FO) queries, i.e., the backbone of relational query languages. The key idea was to *refine* the notion of certain answers, and classify possible answers not into "certainly true" and the rest, but rather into "certainly true", "certainly false", and the rest. Computing these answers precisely is an intractable task, but one can obtain an efficient evaluation procedure that gives an *approximation*.

To give a very simple example, consider the following incomplete database and query:

$$R = \{1, 2\} \quad S = \{\bot, 2\} \qquad \varphi(x) = R(x) \wedge \neg S(x)\,. \quad (1)$$

Here $\bot$ indicates a missing (currently unknown) value, and φ computes the difference of R and S. It will annotate 1 with value **u**, since, depending on the interpretation of $\bot$, it may or may not be in the query answer, and 2 will be annotated with **f**. The classical definition of certain answers (Abiteboul, Hull, and Vianu 1995; Imielinski and Lipski 1984) which accounts only for "certainly true" information will only tell us that there are no answers (missing the useful bit of information that 2 is never in the answer), but the evaluation by a relational DBMS (more precisely, of an SQL query

`SELECT * FROM R EXCEPT SELECT * FROM S`) will return value 1, which is an incorrect – false positive – result.

The goal of this paper is to provide a many-valued framework for obtaining efficient procedures that compute correct approximations of certain answers. They do so by refining the rather coarse information carried by certain answers themselves, in a way that is guided by the underlying many-valued logic. The procedure of Libkin (2016b) mentioned earlier will be just one application of such a framework, when the underlying logic is 3VL, used in commercial DBMSs. To achieve this, we need to answer the following questions:

- How do we define many-valued query answers? What is their semantics, and how can we compare them in terms of the information they contain?
- When can an evaluation scheme be viewed as an approximation for a given semantics of query answers?
- How can we obtain such evaluation schemes if we are given a many-valued logic and a semantics of query answers for it?

To understand the importance of the first question, note that queries Q on incomplete databases must satisfy the *information preservation* principle (Libkin 2014; 2016a): if a database D' has at least as much information as a database D, then the answer $Q(D')$ has at least as much information as the answer $Q(D)$. Indeed, if we are given a more informative D', then at the very least we can run Q on the less informative D; thus $Q(D')$ adds information to $Q(D)$.

Since the notion of being more informative is given by the semantics of databases, and *query answers* are (possibly incomplete) databases themselves, we need to understand their semantics, which no one has properly defined in the context of many-valued query evaluation, as practiced by commercial DBMSs. Understanding and defining such semantics, and the notion of informativeness that comes with it, is the first contribution of this paper.

To define the semantics of many-valued answers we appeal to a well known observation (Belnap 1977) that truth values can be ordered by both their degree of truth, and their *degree of knowledge*, or informativeness. The key property of the latter is that propositional connectives, such as $\wedge$, $\vee$ and $\neg$, preserve it, i.e., they return more informative results when more informative inputs are given (Arieli and Avron 1996; 1998; Ginsberg 1988). Based on this, we define many-valued semantics and informativeness ordering for query answers, with which we can proceed to answer the remaining questions.

Evaluation procedures assign truth values to tuples in such a way that the resulting answer is no more informative than the actual query answer (which may be too costly to compute). Among different evaluation procedures, we identify a particularly nice class that comes with rather strong correctness guarantees. Our next results give a technique for generating procedures in this class. This requires only very mild conditions on the many-valued logic and the query semantics. The evaluation procedures generated by this technique have low computational complexity, either the same as for FO query evaluation (AC^0), or slightly higher in terms of parallel complexity (NC^1).

We then look at concrete applications of such a technique. We show that the 3VL evaluation procedure of Libkin (2016b) that corrected SQL's shortcomings is a natural example of the general technique for 3VL. We also look at a 4-valued logic 4VL that further refines 3VL. While 4-valued logics have been considered in relational databases to handle different types of incompleteness (e.g., Gessert 1990), our idea is to provide finer information on possible answers. Consider again the example given in (1). While for possible answer 1, SQL evaluation returned the incorrect value **t**, the 3VL evaluation with correctness guarantees produced value **u**, or unknown. However, we do know something about 1, namely that depending on the interpretation of the null $\perp$, it may, or may not be in the answer. Hence, it cannot be true with certainty, nor can it be false with certainty. This is extra information compared to **u**, and it can be captured by an additional truth value in 4VL. We provide an efficient evaluation procedure for this logic, and show that it properly refines the 3VL evaluation procedure.

Finally, we look at the usual two-valued Boolean logic, and show that no evaluation procedure that respects the rules for $\wedge, \vee, \neg$ can guarantee correctness. This justifies the decision by relational DBMS designers to use 3VL, rather than Boolean logic, for handling nulls (although they did not get it quite right). We also use the information ordering to show that among all 3VL procedures that are built inductively on the structure of the query, the one our approach generates is the most informative.

Organization Basic notions and definitions are given in Section 2. Section 3 presents the main ideas of the many-valued framework, and Section 4 shows how to achieve correctness guarantees in this framework. Section 5 presents a case study for first-order logic queries based on 3VL and 4VL semantics, and also shows that the usual Boolean two-valued logic cannot provide correctness guarantees. Section 6 shows how to compare evaluation procedures and discusses their optimality. Conclusions are in Section 7.

2 Preliminaries

As was explained in the introduction, we are motivated by applications that rely on the standard database technology, and thus we follow a typical database approach to incompleteness, which we outline now. Of course there are many other representations of incompleteness that occur in AI, for instance in logic programming (Gelfond and Lifschitz 1991) or rough sets settings (Doherty et al. 2006).

Incomplete Databases We consider databases with missing values represented by *marked*, also called *naïve* or *labeled*, nulls (Abiteboul, Hull, and Vianu 1995; Imielinski and Lipski 1984). These are typical in applications such as data integration and exchange (Arenas et al. 2014; Lenzerini 2002), common in database applications (Grahne 1991; van der Meyden 1998), and furthermore they subsume null values that occur in relational DBMSs (Date and Darwen 1996); hence we use this model. Incomplete databases are populated by two types of elements: *constants* and *nulls*, that come from countably infinite sets denoted by Const and Null, respectively. We use the symbol $\bot$ to denote nulls.

A relational schema (vocabulary) is a set of relation names with associated arities. To each k-ary relation symbol S from the vocabulary, an incomplete relational instance D assigns a k-ary relation S^D over $\mathsf{Const} \cup \mathsf{Null}$, i.e., a finite subset of $(\mathsf{Const} \cup \mathsf{Null})^k$. When the instance is clear from the context we shall write S, rather than S^D, for the relation itself as well. The sets of constants and nulls that occur in D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. If $\mathsf{Null}(D)$ is empty, we refer to D as *complete*; that is, complete databases are those without nulls. The *active domain* of D is $\text{adom}(D) = \mathsf{Const}(D) \cup \mathsf{Null}(D)$.

A *valuation* $v\colon \mathsf{Null}(D) \to \mathsf{Const}$ is a map that assigns a constant value to each null in a database. The notion naturally extends to databases, by replacing each $\bot$ with $v(\bot)$ in every place where $\bot$ occurs. We denote the resulting database by $v(D)$, and use valuations to define the *semantics* of incomplete databases:

$$[\![D]\!] = \{v(D) \mid v \text{ is a valuation}\}\,. \tag{2}$$

Intuitively, $[\![D]\!]$ is the set of possible complete databases that D can represent.

Note that (2) defines a semantics where databases are not open to adding new facts. By analogy with the *closed-world* assumption (Reiter 1977), it is called the CWA semantics in database literature (Imielinski and Lipski 1984). It differs from the open-world (OWA) semantics; here we just restore the missing information already present in the database (whence "closedness"). In the study of incompleteness in databases, the closed-world semantics is more common (Abiteboul, Hull, and Vianu 1995; Abiteboul, Kanellakis, and Grahne 1991; Imielinski and Lipski 1984); it is better behaved, as query answering under the open-world semantics is undecidable for relational calculus queries even in data complexity. Various applications mix OWA and CWA semantics; for instance, both occur in integration and exchange applications, and while ontology-based data access tends to use OWA, recent studies showed the important role of CWA in that setting (Lutz, Seylan, and Wolter 2015). In this paper we look at input databases interpreted under CWA; see Section 7 for additional discussions on OWA.

Valuations are a special case of homomorphisms. For two databases D and D' of the same schema, a *homomorphism* $h\colon D \to D'$ is a map from $\text{adom}(D)$ to $\text{adom}(D')$ such that $h(c) = c$ for every $c \in \mathsf{Const}(D)$, and for every relation symbol S, if a tuple $\bar{u}$ is in the relation S in D, the tuple $h(\bar{u})$ is in the relation S in D'. By $h(D)$ we denote the image of D, i.e., the set of all tuples $S\big(h(\bar{u})\big)$ where $S(\bar{u})$ is in D. A valuation is simply a homomorphism such that $h(x)$ is a constant for every $x \in \text{adom}(D)$.

First-Order Logic (FO) As our basic query language we consider FO (often referred to as *relational calculus* in the database context), whose formulae include relational atoms $R(\bar{x})$, equality atoms $x = y$ and are closed under conjunction $\wedge$, disjunction $\vee$, negation $\neg$, existential quantifiers $\exists$ and universal quantifiers $\forall$. We write $\varphi(\bar{x})$ to indicate that $\bar{x}$ is the list of free variables of formula φ, and we write $|\bar{x}|$ for the length of $\bar{x}$. For complete databases, the notion of $D \models \varphi(\bar{a})$, where $\bar{a}$ is a tuple of elements of $\text{adom}(D)$ interpreting $\bar{x}$, is defined in the standard way. The result of the query is then the set of all tuples $\bar{a}$ over $\text{adom}(D)$ such that $D \models \varphi(\bar{a})$. If $|\bar{x}| = k$, we speak of a k-*ary* query. Over incomplete databases, there are many alternative ways of defining semantics that will be discussed in Section 3.

Certain Answers Traditionally, certain answers to a query Q on an incomplete database D are defined as $\Box(Q, D) = \bigcap\{Q(D') \mid D' \in [\![D]\!]\}$. This definition is restrictive in a variety of ways (Libkin 2016a); in particular it disallows tuples with nulls. The concrete semantics of query answering considered here will use a slight modification (Lipski 1984) that overcomes this deficiency.

Note that $\Box(Q, D)$ consists of tuples $\bar{u}$ over Const such that for every valuation v, the tuple $v(\bar{u})$ is in $Q\big(v(D)\big)$. The notion of *certain answers with nulls* (borrowing the name from Libkin 2016b) simply drops the requirement that tuples be constant. For a k-ary query Q, they are defined as

$$\Box_\bot(Q, D) \;= \Big\{\, \bar{u} \in \text{adom}(D)^k \mid v(\bar{u}) \in Q\big(v(D)\big) \text{ for every valuation } v \,\Big\}\,.$$

In particular, $\Box(Q, D)$ is the set of constant tuples in $\Box_\bot(Q, D)$. To see why this notion is better to use, consider a database D with a relation $\{(1, 2), (3, \bot)\}$ and a query Q returning that relation. Then, $\Box(Q, D)$ only keeps the tuple $(1, 2)$, losing information about a tuple with the first component 3, which is in the answer with certainty. On the other hand, $\Box_\bot(Q, D)$ returns both tuples $(1, 2)$ and $(3, \bot)$.

It is well known (Abiteboul, Kanellakis, and Grahne 1991) that data complexity of certain answers (with nulls) is CONP-complete, and thus completely out of reach, given typical sizes of databases. This explains the need for efficient approximation procedures.

Many-Valued Logics A many-valued logic $\mathcal{L}$ is given by its set of *truth values* $\mathbf{T} = \{\tau_0, \ldots, \tau_n\}$, a set Ω of propositional connectives (e.g., $\wedge, \vee, \neg$) with their arities, and an interpretation of each k-ary connective ω as a map $\omega_\mathcal{L}\colon \mathbf{T}^k \to \mathbf{T}$. We shall usually omit $\mathcal{L}$ if it is clear from the context. We shall use the convention here that τ and σ range over truth values of many-valued logics.

Logics come equipped with a *knowledge ordering* $\preccurlyeq$ on their values, indicating when a truth value carries more information (Belnap 1977; Ginsberg 1988). We shall assume that the value τ_0 is the least among those in $\mathbf{T}$, indicating 'no information whatsoever', i.e., $\tau_0 \preccurlyeq \tau_i$ for every $i \leq n$.

For the 3VL used by relational DBMS (also known as the Kleene logic), its truth tables and knowledge ordering are:

	$\neg$
t	**f**
f	**t**
u	**u**

$\wedge$	**t**	**f**	**u**
t	**t**	**f**	**u**
f	**f**	**f**	**f**
u	**u**	**f**	**u**

$\vee$	**t**	**f**	**u**
t	**t**	**t**	**t**
f	**t**	**f**	**u**
u	**t**	**u**	**u**

$$\begin{tikzcd} \mathbf{t} & & \mathbf{f} \\ & \mathbf{u} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

In terms of knowledge, rather than truth, **t** and **f** are incomparable, but **u** gives us less knowledge than both **t** and **f**. The crucial property of the knowledge ordering $\preccurlyeq$ is that the connectives of the logic are *monotone* with respect to it: indeed, the more knowledge we have about the input, the more we know about the output. It is easy to check that $\wedge, \vee, \neg$ above are monotone with respect to $\preccurlyeq$.

Information Orderings A key concept in defining the notion of certainty, and in providing the correct semantics of queries and their answers, is the notion of an *information ordering* on databases (Libkin 2016a). This, given a semantics of incomplete databases, allows us to compare them in terms of their information content. The idea is that the larger the semantics of an incomplete database, the less informative it is. Indeed, the more an incomplete database can potentially denote, the less we know about it: if we have full knowledge, the database can denote just itself, if we have no knowledge whatsoever, it can denote anything.

For input databases, the information ordering $\sqsubseteq$ is given by

$$D \sqsubseteq D' \;\Leftrightarrow\; [\![D']\!] \subseteq [\![D]\!]\,.$$

and in general this notion can be used with other semantics as well. For the CWA semantics $[\![\,]\!]$, it is known (Gheerbrant, Libkin, and Sirangelo 2014) that $D \sqsubseteq D'$ iff there is a homomorphism $h : D \to D'$ such that $h(D) = D'$, which is called a strong onto homomorphism.

3 Many-Valued Query Answering

Let us come back to the example from the introduction: the query $\varphi(x) = R(x) \wedge \neg S(x)$ on the database D with $R = \{1, 2\}$ and $S = \{\bot, 2\}$. The answer annotates each element with a truth value. We assume that the correct annotation should assign $\mathbf{u}$ to 1 and $\mathbf{f}$ to 2, although we saw that under the semantics of SQL negation, 1 is incorrectly assigned $\mathbf{t}$.

This tells us that a many-valued answer is more than just a set of tuples: it assigns truth values from $\mathbf{T}$ to tuples. The semantics of a query should tell us how to assign these values correctly, but a particular evaluation mechanism need not necessarily coincide with the semantics (in fact this rarely the case, as the correct semantics may be too expensive to compute).

This suggests that we have to clearly distinguish three distinct concepts related to queries:

- their syntax φ (in this paper, the standard FO formulae);
- their semantics, which will be denoted by $\mathsf{Sem}(\varphi)$; and
- their evaluations, which will be denoted by $\mathsf{Eval}(\varphi)$.

The main questions we need to address are:

1) How can we say when a semantics $\mathsf{Sem}(\varphi)$ is correct?
2) When do we know that an evaluation $\mathsf{Eval}(\varphi)$ is appropriate for $\mathsf{Sem}(\varphi)$?
3) How can we compare different evaluations?

We now proceed to define a framework in which we can answer these questions.

Many-Valued Query Answers With a two-valued logic, each tuple is classified into either true or false: we thus think of a query returning the set of tuples on which it evaluates to true, and the other set is its complement. With many-valued semantics, tuples can be annotated with different truth values of a logic $\mathcal{L}$ (even if only the set corresponding to $\mathbf{t}$ is given to the user). This is captured by the following definition.

Definition 1. For a many-valued logic $\mathcal{L}$ with truth values $\mathbf{T}$, a *k-ary query answer* is a map $\mathbb{A}$ that maps each truth value $\tau \in \mathbf{T}$ to a set $\mathbb{A}^\tau$ of k-ary tuples over Const and Null. Such a query answer is *complete* if no nulls occur in tuples. We also write $\mathbb{A}^{\uparrow\tau}$ for $\bigcup\{\mathbb{A}^\sigma \mid \sigma \succcurlyeq \tau\}$, i.e., the set of tuples corresponding to truth values at least as informative as τ.

Note that we do not insist on the components of the answer being disjoint. Although in many concrete semantics they will be, there are cases when they are not, for instance in many-valued logics having truth values that are combinations of others, such as inconsistency, which means both $\mathbf{t}$ and $\mathbf{f}$ at the same time (Ginsberg 1988). Also, from the technical point of view all the results are true without such a restriction, and hence we do not impose it.

Semantics and Ordering Our next goal is to define the semantics $(\!|\mathbb{A}|\!)$ of query answers, and the information ordering $\Subset$ associated with it. Note that query answers themselves are incomplete databases, and thus we need to understand how to compare them in terms of the knowledge they possess.

For this, we again appeal to example (1) from the introduction. If we deal with 3VL, then tuples are annotated with $\mathbf{t}$, $\mathbf{f}$ or $\mathbf{u}$. One possibility is $\{\mathbf{t} \mapsto \varnothing,\ \mathbf{f} \mapsto \varnothing,\ \mathbf{u} \mapsto \{1\}\}$. It provides rather little information, just that the value 1 may be considered as an answer, but we do not know whether it is in fact in the answer. A slightly better option is $\{\mathbf{t} \mapsto \varnothing,\ \mathbf{f} \mapsto \varnothing,\ \mathbf{u} \mapsto \{1, \bot\}\}$, which in addition says that some currently unknown value is a potential answer. A still better one is $\{\mathbf{t} \mapsto \varnothing, \mathbf{f} \mapsto \varnothing, \mathbf{u} \mapsto \{1, 2\}\}$, which says that a concrete value 2 is a potential answer. And an even more informative one is $\{\mathbf{t} \mapsto \varnothing, \mathbf{f} \mapsto \{2\}, \mathbf{u} \mapsto \{1\}\}$: value 2 is no longer annotated with $\mathbf{u}$, but rather with a more informative value $\mathbf{f}$, saying that 2 is never in the difference of R and S.

This suggests that the semantics $(\!|\mathbb{A}|\!)$ of a k-ary query answer $\mathbb{A}$ is a set of complete answers $\mathbb{C}$ that can improve our knowledge of $\mathbb{A}$ in three possible ways:

1) some nulls in $\bar{u} \in \mathbb{A}^\tau$ can be replaced by constants by means of a valuation v, and $v(\bar{u})$ can be placed in $\mathbb{C}^\tau$;
2) we can gain knowledge about a tuple $\bar{w} \in \mathbb{C}^\tau$ and move it to $\mathbb{C}^\sigma$ if $\tau \preccurlyeq \sigma$; and
3) we can add a new tuple to any of the $\mathbb{C}^\tau$'s, reflecting the fact that we *approximate* certain answers, hence by adding tuples we gain additional information.

All of the above ways of gaining knowledge are captured by the following semantics of k-ary query answers.

$$(\!|\mathbb{A}|\!) = \{\, \text{complete answer } \mathbb{C} \mid \exists \text{ valuation } v \text{ such that } v(\mathbb{A}^\tau) \subseteq \mathbb{C}^{\uparrow\tau} \text{ for each } \tau \in \mathbf{T} \,\}\,.$$

For the logic with $\mathbf{T} = \{\mathbf{t}, \mathbf{f}\}$ and the ordering $\mathbf{f} \preccurlyeq \mathbf{t}$, this becomes the OWA semantics of incomplete databases (for the set $\mathbb{A}^{\mathbf{t}}$), which is typically used, albeit implicitly, in the usual two-valued query answering over incomplete databases.

Every semantics of incompleteness gives us an *information ordering* (see Section 2). For input databases, the ordering $D \sqsubseteq D'$ was characterized by the existence of a homomorphism h from D to D' such that $h(D) = D'$. For query outputs, the ordering is denoted by $\Subset$, and it is given by

$$\mathbb{A} \Subset \mathbb{A}' \;\Leftrightarrow\; (\!|\mathbb{A}'|\!) \subseteq (\!|\mathbb{A}|\!)\,.$$

The ordering $\Subset$ can be characterized as follows.

Proposition 1. *For k-ary query answers, we have $\mathbb{A} \Subset \mathbb{B}$ iff there exists a mapping $h\colon$ $\mathsf{Null} \to \mathsf{Const} \cup \mathsf{Null}$ such that $h(\mathbb{A}^\tau) \subseteq \mathbb{B}^{\uparrow\tau}$ for every truth value τ.*

This is an analog of the existence of a homomorphism from $\mathbb{A}$ to $\mathbb{B}$, except that the image of a tuple in $\mathbb{A}^\tau$ may appear not only in $\mathbb{B}^\tau$ but also in any $\mathbb{B}^\sigma$ for $\sigma \succcurlyeq \tau$. When the statement of Proposition 1 holds, we say that $\mathbb{A} \Subset \mathbb{B}$ is *witnessed* by h.

Query Semantics and Evaluations We now use the notions of query answers and information orderings to explain what semantics and evaluation are.

Definition 2. A *query semantics* for a many-valued logic $\mathcal{L}$ with truth values $\mathbf{T}$ is a function Sem that, with a syntactic query $\varphi(\bar{x})$ where $|\bar{x}| = k$, and a database D, associates a k-ary query answer $\mathsf{Sem}(\varphi, D)$ such that:

1) the union of all $\mathsf{Sem}^\tau(\varphi, D)$ is $\mathrm{adom}(D)^k$, and
2) if $D \sqsubseteq D'$, then $\mathsf{Sem}(\varphi, D) \Subset \mathsf{Sem}(\varphi, D')$.

The first condition simply says that every tuple is accounted for, i.e., assigned at least one value by the semantic function. The second condition is the preservation of informativeness (Libkin 2016a): if we know more about the input, we know more about the output.

Definition 3. A $\mathbf{T}$-valued *evaluation procedure* Eval takes a syntactic query $\varphi(\bar{x})$ with $|\bar{x}| = k$, a database D and an assignment $\bar{a}$ of elements of $\mathrm{adom}(D)$ to $\bar{x}$, and returns a truth value $\mathsf{Eval}(\varphi, D, \bar{a}) \in \mathbf{T}$. The k-ary query answer produced by Eval on φ and D is denoted by $\mathsf{Eval}(\varphi, D)$, where

$$\mathsf{Eval}^\tau(\varphi, D) \;=\; \{\bar{a} \in \mathrm{adom}(D)^k \mid \mathsf{Eval}(\varphi, D, \bar{a}) = \tau\}$$

is the set of tuples on which Eval returns τ.

Observe that, for any two distinct truth values τ and σ, $\mathsf{Sem}^\tau(\varphi, D)$ and $\mathsf{Sem}^\sigma(\varphi, D)$ may have tuples in common, while $\mathsf{Eval}^\tau(\varphi, D)$ and $\mathsf{Eval}^\sigma(\varphi, D)$ are always disjoint, as an evaluation procedure assigns precisely one truth value to each tuple.

Most commonly, the evaluation output given to the user is simply $\mathsf{Eval}^{\mathbf{t}}(\varphi, D)$, serving as an approximation of certain answers. As we saw, this is often refined with information about other truth values. Moreover, other sets often need to be computed even if the goal is to just return $\mathsf{Eval}^{\mathbf{t}}(\varphi, D)$.

Information orderings on query answers suggest what can be considered as potential evaluations for a given semantics. Indeed, the least condition we should impose on evaluations is that they cannot produce more information than the semantics gives us; that is, $\mathsf{Eval}(\varphi, D) \Subset \mathsf{Sem}(\varphi, D)$ for every query φ and every database D.

This is very permissive however; for instance, evaluations satisfying this condition may unnecessarily identify nulls with constants or with each other. As a simple example, consider the ordering $\mathbf{f} \preccurlyeq \mathbf{t}$, and a query $\varphi(x)$ so that on a database with $\mathrm{adom}(D) = \{1, \bot\}$ its semantics assigns $\mathbf{f}$ to $\bot$ and $\mathbf{t}$ to 1. Now suppose an evaluation does the opposite, i.e., it assigns $\mathbf{f}$ to 1 and $\mathbf{t}$ to $\bot$. Intuitively this is wrong, as the evaluation somehow knows more about the null $\bot$ than the semantics does. However, the ordering allows this, since $\mathsf{Eval}(\varphi, D) \Subset \mathsf{Sem}(\varphi, D)$ is witnessed by a map h that sends $\bot$ to 1.

This suggests getting rid of arbitrary maps witnessing $\mathsf{Eval}(\varphi, D) \Subset \mathsf{Sem}(\varphi, D)$ and only permitting identity; in this case, evaluation simply distributes tuples among sets corresponding to different truth values.

Definition 4. We say that Eval is an *evaluation* for semantics Sem if $\mathsf{Eval}(\varphi, D) \Subset \mathsf{Sem}(\varphi, D)$ is witnessed by the identity mapping, i.e., $\mathsf{Eval}^\tau(\varphi, D) \subseteq \mathsf{Sem}^{\uparrow\tau}(\varphi, D)$ for every query φ, database D, and truth value τ.

Thus, if an evaluation Eval assigns truth value τ to a tuple, then the semantics must assign a value σ that is at least as informative as τ. The following is an immediate consequence of the definition.

Proposition 2. *Let* Eval *be an evaluation for* Sem. *If τ is a maximal truth value with respect to $\preccurlyeq$, then* $\mathsf{Eval}^\tau(\varphi, D) \subseteq \mathsf{Sem}^\tau(\varphi, D)$.

In most of the cases naturally arising in handling incompleteness, the semantics of truth value $\mathbf{t}$ is the set of certain answers (with nulls), and $\mathbf{t}$ is a maximal truth value with respect to $\preccurlyeq$. Thus, in this case the proposition implies that, in an evaluation, the set of tuples assigned $\mathbf{t}$ gives us an approximation of certain answers.

It would be desirable to extend Proposition 2 to other truth values: if Eval assigns τ to a tuple $\bar{a}$, then the semantics does the same (so far this is guaranteed only for the maximal truth values). This is too strong however: if Sem assigns each tuple to only one truth value, this simply means that Eval and Sem coincide, which usually precludes us from having a computationally efficient Eval. Ideally, we want Eval to deviate from Sem as little as possible. Thus, we only allow them to be different when Eval cannot infer any information about a tuple and assigns it the smallest truth value τ_0 (in concrete semantics that we use, it will be the unknown $\mathbf{u}$).

Definition 5. For a query φ, an evaluation procedure Eval:

- has *correctness guarantees* for semantics Sem if for every database D and every truth value $\tau \neq \tau_0$, we have $\mathsf{Eval}^\tau(\varphi, D) \subseteq \mathsf{Sem}^\tau(\varphi, D)$;
- *preserves informativeness* if $\mathsf{Eval}(\varphi, D) \Subset \mathsf{Eval}(\varphi, D')$ holds whenever $D \sqsubseteq D'$.

In the next section we give a recipe for obtaining evaluation procedures with correctness guarantees, and then study it for some concrete logics.

4 Correct and Efficient Evaluation

The previous section gave us a general framework for reasoning about many-valued query answers, as well as query evaluations and semantics. We now demonstrate that, in this framework, it is easy to generate evaluation procedures with correctness guarantees. In fact, all we need to do is define them for atomic formulae and then lift to all of FO.

Let $\mathcal{L}$ be a many-valued logic with a set Ω of propositional connectives. Each m-ary connective ω is given by its truth table $\omega_{\mathcal{L}}\colon \mathbf{T}^m \to \mathbf{T}$. We shall often omit the subscript $\mathcal{L}$ when it is clear from the context. We always assume that Ω includes the connectives $\wedge$ and $\vee$; the only condition on their interpretation is that they are associative so that we can unambiguously define $\bigwedge T$ and $\bigvee T$ when T is a tuple of truth values.

By $\mathrm{FO}_{\mathcal{L}}$ we mean the closure of atomic formulae $R(\bar{x})$ and $x = y$ under propositional connectives from Ω and quantifiers $\forall, \exists$. If $\Omega = \{\wedge, \vee, \neg\}$ this is the usual first-order logic, but in general we can be more permissive about propositional connectives.

We look at procedures that lift evaluations of atoms following the *syntax* of formulae;[1] in fact standard query evaluation procedures are such (even if they optimize a query first, then they follow the syntax). This is captured by the following definition.

Definition 6. An evaluation procedure Eval for $\mathrm{FO}_{\mathcal{L}}$ is called *syntactic* if it follows the rules below. For propositional connectives, we have

$$\begin{aligned}&\mathsf{Eval}\Big(\omega\big(\varphi_1(\bar{x}_1), \ldots, \varphi_m(\bar{x}_m)\big), D, \bar{a}\Big) = \\ &\omega_{\mathcal{L}}\Big(\mathsf{Eval}\big(\varphi_1(\bar{x}_1), D, \bar{a}_1\big), \ldots, \mathsf{Eval}\big(\varphi_m(\bar{x}_m), D, \bar{a}_m\big)\Big)\end{aligned}$$

where $\bar{a}_i$ is the subtuple of $\bar{a}$ corresponding to the variables in $\bar{x}_i$. For quantifiers, we have

$$\mathsf{Eval}\big(\exists y\, \varphi(\bar{x}, y), D, \bar{a}\big) = \bigvee_{a \in \mathrm{adom}(D)} \mathsf{Eval}\big(\varphi(\bar{x}, y), D, (\bar{a}, a)\big)\,,$$
$$\mathsf{Eval}\big(\forall y\, \varphi(\bar{x}, y), D, \bar{a}\big) = \bigwedge_{a \in \mathrm{adom}(D)} \mathsf{Eval}\big(\varphi(\bar{x}, y), D, (\bar{a}, a)\big)\,.$$

In other words, for a syntactic procedure, we need to define rules for atoms $R(\bar{x})$ and $x = y$, and then lift to all of $\mathrm{FO}_{\mathcal{L}}$ using the above rules. Our goal is then to show that correctness of such procedures can also be lifted from atoms to all of $\mathrm{FO}_{\mathcal{L}}$. For this, we need some mild conditions on the semantics.

First, we assume that each formula $\varphi(\bar{x})$ can also be viewed as a formula $\varphi(\bar{y})$, as long as all variables in $\bar{x}$ occur in $\bar{y}$, in some order. For instance, a formula $\varphi(x, y)$ with output X (which is a set of pairs) can also be viewed as a formula $\varphi(y, x, z)$ with the output $\{(b, a, c) \mid (a, b) \in X, c \in \mathrm{adom}(D)\}$. In general, $\mathsf{Sem}^{\tau}(\varphi(\bar{y}), D)$ is defined as the set of all tuples $\bar{a} \in \mathrm{adom}(D)^{|\bar{y}|}$ such that the subtuple of $\bar{a}$ corresponding to the positions of $\bar{x}$ belongs to $\mathsf{Sem}^{\tau}(\varphi(\bar{x}), D)$. This is essentially a free condition; all natural logic-based semantics will be such, and thus we assume it always holds.

The main condition is the *compatibility* of Sem with $\mathcal{L}$. To explain it informally, suppose a tuple $\bar{u}$ belongs to $\mathsf{Sem}^{\tau}(\varphi, D)$ and $\mathsf{Sem}^{\sigma}(\psi, D)$, where τ, σ, and $\tau \wedge \sigma$ are truth values carrying some information, i.e., different from τ_0. Then we would expect $\bar{u}$ to be in the set corresponding to $\tau \wedge \sigma$ in $\mathsf{Sem}(\varphi \wedge \psi, D)$, if the semantics reflects the meaning of the connectives. Note that τ_0 is essentially the 'no-information' value (i.e., the fallback position when we cannot assign any value carrying real information). Hence we do not impose any conditions on the tuple with respect to τ_0. Now we extend this idea to arbitrary propositional connectives.

Definition 7. A query semantics Sem is *compatible* with a many-valued logic $\mathcal{L}$ if for each m-ary connective $\omega \in \Omega$, truth values satisfying $\omega(\sigma_1, \ldots, \sigma_m) = \tau$ with $\tau \neq \tau_0$, formulae $\varphi_1(\bar{x}), \ldots, \varphi_m(\bar{x})$ and $\varphi(\bar{x}) = \omega(\varphi_1, \ldots, \varphi_m)$, we have

$$\bigcap_{\sigma_i \neq \tau_0} \mathsf{Sem}^{\sigma_i}(\varphi_i, D) \subseteq \mathsf{Sem}^{\tau}(\varphi, D)\,.$$

Theorem 1. *Assume that the semantics* Sem *is compatible with* $\mathcal{L}$*, and* Eval *is a syntactic evaluation procedure. If* Eval *has correctness guarantees for* Sem *for atomic formulae, then it has correctness guarantees for all* $\mathrm{FO}_{\mathcal{L}}$ *formulae.*

Most reasonable semantics, including all those in Section 5, will be compatible with the many-valued logics we use. Thus, Theorem 1 says that the key tasks for obtaining an evaluation scheme with correctness guarantees are choosing the right many-valued logic, and choosing the right evaluation for atomic formulae. The rest is then guaranteed.

Similarly, preserving informativeness can be lifted from atomic formulae to arbitrary ones. Recall that $D \sqsubseteq D'$ means that $D' = h(D)$ for some homomorphism h. Thus, if we have an atomic formula $\varphi(\bar{x})$ and a truth value $\tau = \mathsf{Eval}(\varphi, D, \bar{a})$, then we expect $h(\bar{a})$ to provide us with at least as much information in $h(D)$, that is, $\tau \preccurlyeq \mathsf{Eval}\big(\varphi, h(D), h(\bar{a})\big)$. If this happens for all $D \sqsubseteq D'$, we say that Eval *respects* $\preccurlyeq$ *for* φ.

Theorem 2. *Let* Eval *be a syntactic evaluation procedure that respects* $\preccurlyeq$ *for all atomic formulae. Then* Eval *preserves informativeness for all* $\mathrm{FO}_{\mathcal{L}}$ *formulae.*

We now address the data complexity of evaluation procedures. Normally we would expect data complexity of FO-based logics to be in AC^0. However, we imposed no condition on the many-valued interpretation of the connectives $\wedge$ and $\vee$, and quantifiers are defined by iterating those operations. Such iterations tend to increase complexity to NC^1 (Barrington, Immerman, and Straubing 1990), still a small parallel complexity class. The AC^0 complexity of FO is due to the fact that $\wedge$ and $\vee$ are idempotent, but in some many-valued logics they may not be.

Proposition 3. *If* Eval *is a syntactic evaluation procedure that has* NC^1 *data complexity for atomic formulae, then it has* NC^1 *data complexity for all* FO *formulae. If it has* AC^0 *data complexity for atomic formulae, and the connectives* $\wedge$ *and* $\vee$ *are idempotent, then* Eval *has* AC^0 *data complexity.*

The result is slightly more general in two ways. First, if the data complexity of Eval for atomic formulae is in a class $\mathcal{K}$ given by circuits that are closed under NC^1 circuit-building operations (i.e., can be incorporated into log-depth

[1]In some contexts, such procedures are called *compositional*.

bounded fan-in circuits without increase in complexity) then Eval is $\mathcal{K}$ for all queries. For instance, if we find an evaluation scheme for atomic formulae that is in PTIME, then it remains in PTIME for all queries. Second, we do not even need idempotency for the AC^0 bound: what is required is a weak idempotency, saying that there is $k > 0$ so that iterating the operation $k + 1$ times is the same as iterating it k times (for $k = 1$, this is the usual idempotency $\tau * \tau = \tau$, when $*$ is either $\wedge$ or $\vee$).

5 Applications: 3VL, 4VL, and 2VL

We now demonstrate the applicability of the framework developed in the previous section. To begin with, we show that the main result of Libkin (2016b) – an efficient and correct 3VL evaluation procedure that mends some of the most glaring issues with SQL's evaluation of nulls – is an immediate application of the framework for 3VL. As a bonus, we show that the evaluation procedure also preserves informativeness. We then provide a refinement, already hinted at in the introduction, by showing how to classify some of the unknown tuples in a way that generates additional useful information (for instance, ruling them out as certain answers). We finally show that certainty guarantees are impossible with the usual Boolean 2-valued logic, thus justifying the decision by the designers of practical query languages (SQL) to use 3VL.

3VL

We start with the usual truth values **t**, **f**, **u**. For a tuple $\bar{a}$, an incomplete database D, and a k-ary query $\varphi(\bar{x})$, **t** means that $\bar{a}$ is in the answer for all $D' \in [\![D]\!]$ and **f** means that $\bar{a}$ is not in the answer for all $D' \in [\![D]\!]$. Thus, we define the three-valued semantics $\mathsf{Sem}_{3v}(\varphi, D)$ as follows:

$$\mathsf{Sem}^{\mathbf{t}}_{3v}(\varphi, D) = \Box_\bot(\varphi, D)\,,$$
$$\mathsf{Sem}^{\mathbf{f}}_{3v}(\varphi, D) = \Box_\bot(\neg\varphi, D)\,,$$
$$\mathsf{Sem}^{\mathbf{u}}_{3v}(\varphi, D) = \mathrm{adom}(D)^k - (\Box_\bot(\varphi, D) \cup \Box_\bot(\neg\varphi, D))\,.$$

That is, the set corresponding to **t** is the set of tuples which are certainly true, the set of **f** contains tuples which are certainly false, and other tuples are classified as unknown.

Note that one still needs to show that this is a semantics in the sense of Definition 2, i.e., it classifies all the tuples in $\mathrm{adom}(D)^k$, and returns more informative results on more informative tuples. We can show this, as well as the fact that this semantics is compatible with 3VL in the sense of Definition 7.

Proposition 4. Sem_{3v} *is a query semantics that is compatible with 3VL.*

Therefore, by Theorem 1, we only need to provide a correct evaluation procedure Eval_{3v} for atomic formulae $R(\bar{x})$ and $x = y$, and then lift to full FO using the rules for syntactic evaluation procedures. Recall the notion of correctness: for each truth value $\tau \in \{\mathbf{t}, \mathbf{f}\}$, if $\mathsf{Eval}(\varphi(\bar{x}), D, \bar{a}) = \tau$, then $\bar{a} \in \mathsf{Sem}^{\tau}_{3v}(\varphi, D)$. Combining this with the definition of $\Box_\bot$, we derive evaluations for atomic formulae.

- If $\mathsf{Eval}_{3v}\big(R(\bar{x}), D, \bar{a}\big) = \mathbf{t}$, then $v(\bar{a}) \in v(R^D)$ for every valuation v, so we must require $\bar{a} \in R^D$.

$$\begin{tikzcd} \mathbf{t} & & \mathbf{f} \\ & \mathbf{s} \arrow[ul] \arrow[ur] & \\ & \mathbf{u} \arrow[u] & \end{tikzcd}$$

	$\neg$
t	**f**
f	**t**
s	**s**
u	**u**

$\wedge$	**t**	**f**	**s**	**u**
t	**t**	**f**	**s**	**u**
f	**f**	**f**	**f**	**f**
s	**s**	**f**	**u**	**u**
u	**u**	**f**	**u**	**u**

$\vee$	**t**	**f**	**s**	**u**
t	**t**	**t**	**t**	**t**
f	**t**	**f**	**s**	**u**
s	**t**	**s**	**u**	**u**
u	**t**	**u**	**u**	**u**

Figure 1: 4VL knowledge ordering and truth tables.

- If $\mathsf{Eval}_{3v}\big(R(\bar{x}), D, \bar{a}\big) = \mathbf{f}$, then $v(\bar{a}) \notin v(R^D)$ must hold for every valuation v, so we make it the condition for Eval_{3v} returning false.
- If $\mathsf{Eval}_{3v}(x = y, D, (a, b)) = \mathbf{t}$, then $v(a) = v(b)$ for every valuation v, hence we must require $a = b$.
- If $\mathsf{Eval}_{3v}(x = y, D, (a, b)) = \mathbf{f}$, then $v(a) \neq v(b)$ for every valuation v, hence we must require not only $a \neq b$ but also $a, b \in \mathsf{Const}$ to guarantee this condition.

The second of the above conditions can be expressed with the notion of *unifiable* tuples: these are tuples $\bar{a}$ and $\bar{b}$ of the same length such that $v(\bar{a}) = v(\bar{b})$ for some valuation v. We then write $\bar{a} \Uparrow \bar{b}$. This condition can be tested in linear time (Paterson and Wegman 1978).

All of the above tells us that the evaluation for atoms is:

$$\mathsf{Eval}_{3v}\big(R(\bar{x}), D, \bar{a}\big) = \begin{cases} \mathbf{t} & \bar{a} \in R^D \\ \mathbf{f} & \nexists \bar{b} \in R^D \text{ such that } \bar{a} \Uparrow \bar{b} \\ \mathbf{u} & \text{otherwise}\,, \end{cases}$$

$$\mathsf{Eval}_{3v}\big(x = y, D, (a, b)\big) = \begin{cases} \mathbf{t} & a = b \\ \mathbf{f} & a \neq b \text{ and } a, b \in \mathsf{Const} \\ \mathbf{u} & \text{otherwise}\,. \end{cases}$$

Extended to all of FO by the rules of Definition 6, this is exactly the procedure proposed by Libkin (2016b). Since in 3VL binary connectives are idempotent, our framework thus yields the main result of that paper as an immediate corollary of our framework:

Corollary 1. Eval_{3v} *has correctness guarantees with respect to the three-valued semantics, and* AC^0 *data complexity.*

In addition, we can show that the procedure Eval_{3v} returns more informative results on more informative inputs:

Proposition 5. Eval_{3v} *preserves informativeness.*

4VL

We will now show how to refine the notion of unknown (**u**) to provide additional information with query answers. So far **u** accounts for both real unknowns, and for tuples that cannot be classified as **t** or **f** for a reason. We now separate the two, and add a new value **s** (*sometimes*) whose meaning is that a tuple sometimes is in the query result, and sometimes it is not. That is, we define a 4-valued semantics Sem_{4v} with sets corresponding to truth values $\mathbf{t}, \mathbf{f}, \mathbf{s}, \mathbf{u}$. The sets $\mathsf{Sem}^{\mathbf{t}}_{4v}(\varphi, D)$ and $\mathsf{Sem}^{\mathbf{f}}_{4v}(\varphi, D)$ are defined in the same way as for 3VL, i.e., they are $\Box_\bot(\varphi, D)$ and $\Box_\bot(\neg\varphi, D)$. The set $\mathsf{Sem}^{\mathbf{s}}_{4v}(\varphi, D)$ is defined as

$$\big\{\, \bar{a} \in \text{adom}(D)^k \mid \exists \text{ valuations } v, v' \text{ such that} \\ v(D) \models \varphi\big(v(\bar{a})\big),\ v'(D) \not\models \varphi\big(v'(\bar{a})\big) \big\}$$

and the set $\mathsf{Sem}^{\mathbf{u}}_{4v}(\varphi, D)$ is the complement of the union of all the others, corresponding to real unknowns. If a tuple is in $\mathsf{Sem}^{\mathbf{s}}_{4v}(\varphi, D)$, we know with certainty that it cannot be in $\Box_\bot(\varphi, D)$ nor in $\Box_\bot(\neg\varphi, D)$, so we do gain knowledge compared to labeling such a tuple with $\mathbf{u}$.

The truth tables for the connectives and the knowledge ordering $\preccurlyeq$ for 4VL are shown in Figure 1. Note that $\mathbf{s} \vee \mathbf{s}$ is not $\mathbf{s}$ but $\mathbf{u}$. Indeed, if we have a tuple $\bar{a}$ in $\mathsf{Sem}^{\mathbf{s}}_{4v}(\varphi, D)$ and $\mathsf{Sem}^{\mathbf{s}}_{4v}(\psi, D)$, we can only conclude that there is a valuation v such that $v(D) \models \varphi \vee \psi(v(\bar{a}))$, but we cannot conclude that $v'(D) \not\models \varphi \vee \psi(v'(\bar{a}))$ for some v'. Likewise, $\mathbf{s} \wedge \mathbf{s} = \mathbf{u}$.

To explain the ordering of $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{f}$, assume that we got a value $\mathbf{s}$ for some tuple $\bar{a}$ on a database D; it means that $[\![D]\!]$ contains databases on which $\bar{a}$ is in the answer as well as databases on which it is not. If we now have a more informative database D', that is, $D \sqsubseteq D'$, then $[\![D']\!] \subseteq [\![D]\!]$, and we have three possibilities: either $\bar{a}$ is in the answer on all databases in $[\![D']\!]$ (value $\mathbf{t}$), or it is not in the answer on all such databases (value $\mathbf{f}$), or it is on some and not on others (value $\mathbf{s}$). Hence, by going to a more informative database, $\mathbf{s}$ either stays, or improves to $\mathbf{t}$ or $\mathbf{f}$. The latter in turn give us complete knowledge: going to a more informative database does not affect the answer. This leads to the ordering shown in Figure 1.

As for 3VL, we have defined a proper semantics (it preserves informativeness) that is compatible with the logic.

Proposition 6. *Sem_{4v} is a query semantics that is compatible with 4VL, and the operations $\wedge$, $\vee$ and $\neg$ are monotone with respect to $\preccurlyeq$ in 4VL.*

Thus, by Theorem 1, to get a four-valued evaluation procedure Eval_{4v}, we need to define it for atomic formulae and lift by the rules of Definition 6. The atomic rules (i.e., for $R(\bar{x})$ and $x = y$) of Eval_{4v} are the same as for Eval_{3v}, with just one difference: $\mathbf{u}$ is replaced by $\mathbf{s}$.

The correctness of Eval_{4v} can be easily shown. Furthermore, while the connectives $\wedge$ and $\vee$ are not idempotent ($\mathbf{s} \wedge \mathbf{s} = \mathbf{s} \vee \mathbf{s} = \mathbf{u}$, and this is how $\mathbf{u}$ can be produced by Eval_{4v}), they are weakly idempotent: $\mathbf{s} \vee \mathbf{s} \vee \mathbf{s} = \mathbf{s} \vee \mathbf{s}$ and likewise for $\wedge$. Therefore, we have:

Theorem 3. *Eval_{4v} has correctness guarantees with respect to Sem_{4v}, and AC^0 data complexity. Furthermore, Eval_{4v} preserves informativeness for all* FO *queries.*

Let us now return to the example (1) from the introduction. The procedure Eval_{3v} will assign $\mathbf{u}$ to 1 and $\bot$, and $\mathbf{f}$ to 2, in line with the fact that $\Box_\bot(\varphi, D) = \varnothing$, and refining this information by stating that 2 belongs to $\Box_\bot(\neg\varphi, D)$. The procedure Eval_{3v} does leave open a possibility that 1 could be a certainly true or a certainly false answer, as it assigns $\mathbf{u}$ in an attempt to provide an approximation of certain answers in a computationally efficient way. The evaluation procedure Eval_{4v} further refines this by assigning $\mathbf{s}$ to 1, thus confirming that it cannot be a certain answer.

Revisiting Two-Valued Logic

We were able to find many-valued evaluation procedures with ease. Can we perhaps adapt the techniques and obtain a two-valued procedure for approximating certain answers? To SQL designers, it appeared that they could not, hence they used 3VL. Now we formally prove that they were right.

We say that Eval is a *Boolean evaluation* if it assigns values $\mathbf{t}$,$\mathbf{f}$ to formulae, and satisfies

$$\begin{aligned} \mathsf{Eval}(\varphi \wedge \psi, D, \bar{a}) &= \mathsf{Eval}(\varphi, D, \bar{a}) \wedge \mathsf{Eval}(\psi, D, \bar{a})\,, \\ \mathsf{Eval}(\varphi \vee \psi, D, \bar{a}) &= \mathsf{Eval}(\varphi, D, \bar{a}) \vee \mathsf{Eval}(\psi, D, \bar{a})\,, \\ \mathsf{Eval}(\neg\varphi, D, \bar{a}) &= \neg\mathsf{Eval}(\varphi, D, \bar{a})\,, \end{aligned}$$

where $\wedge, \vee$ and $\neg$ are the usual Boolean connectives interpreted on $\mathbf{t}, \mathbf{f}$. In other words, we give ourselves complete freedom in defining the evaluation for atomic formulae and quantifiers, but insist on respecting the usual two-valued logic.

A *certain answers semantics* Sem is any semantics such that $\mathsf{Sem}^{\mathbf{t}}(\varphi, D) \subseteq \Box_\bot(\varphi, D)$. Then we have the following impossibility result.

Theorem 4. *No certain answers semantics admits a Boolean evaluation.*

In other words, if Eval respects the Boolean semantics of $\wedge, \vee, \neg$, then we cannot have $\mathsf{Eval}^{\mathbf{t}}(\varphi, D) \subseteq \Box_\bot(\varphi, D)$ for all FO formulae and all databases D, that is, Eval will sometimes return false positives.

To explain why this is the case, suppose that we can find a Boolean evaluation Eval with $\mathsf{Eval}^{\mathbf{t}}(\varphi, D) \subseteq \Box_\bot(\varphi, D)$. Take $D = \{R(a), S(\bot)\}$; if $\mathsf{Eval}\big(R(x), D, a\big) = \mathbf{f}$, then $\mathsf{Eval}\big(\neg R(x), D, a\big) = \mathbf{t}$, but $a \notin \Box_\bot\big(\neg R(x), D\big)$ and hence $\mathsf{Eval}\big(R(x), D, a\big) = \mathbf{t}$. Next, if $\mathsf{Eval}\big(\neg S(x), D, a\big) = \mathbf{f}$, then $\mathsf{Eval}\big(S(x), D, a\big) = \mathbf{t}$. But $a \notin \Box_\bot\big(S(x), D\big)$, which implies $\mathsf{Eval}\big(\neg S(x), D, a\big) = \mathbf{t}$. Now take $\varphi(x) = R(x) \wedge \neg S(x)$; from the above, we have $\mathsf{Eval}(\varphi, D, a\big) = \mathbf{t}$, but it is easily seen that $a \notin \Box_\bot(\varphi, D)$, since a complete database where $\bot$ is interpreted as a falsifies $\varphi(a)$. This contradiction shows that a Boolean evaluation for a certain answers semantics cannot exist.

6 Comparison and Optimality of Evaluations

The notion of information ordering $\Subset$ on many-valued outputs allows us to compare different evaluation procedures: those returning more informative outputs are preferable. Given two evaluation procedures Eval and Eval', we say that Eval is *more informative* if $\mathsf{Eval}'(\varphi, D) \Subset \mathsf{Eval}(\varphi, D)$ for every formula φ and database D.

In fact we can use this notion even if Eval and Eval' use different sets of truth values, as long as the orderings on common truth values are the same. Indeed, we simply assume that the set of answers corresponding to a missing truth value is empty. This way we can compare 3VL and 4VL evaluations.

Our first result shows that the 4VL evaluation properly refines 3VL, i.e., it adds information.

Proposition 7. *For every* FO *query φ and every database D, we have* $\mathsf{Eval}^{\tau}_{3v}(\varphi, D) = \mathsf{Eval}^{\tau}_{4v}(\varphi, D)$ *if τ is* **t** *or* **f**. *Furthermore,* $\mathsf{Eval}^{\mathbf{s}}_{4v}(\varphi, D)$ *is disjoint from both* $\Box_{\bot}(\varphi, D)$ *and* $\Box_{\bot}(\neg\varphi, D)$. *Consequently,* Eval_{4v} *is more informative than* Eval_{3v}.

This indicates that if we only care about certain answers (i.e., tuples that evaluate to **t**), and not about the additional information that the 4VL procedure provides, then Eval_{3v} is the evaluation we should use. But can it be improved? It turns out to be the best among syntactic evaluation procedures, but otherwise, improvements are possible.

We start with the optimality result saying that among syntactic procedures based on **t**, **f**, and **u**, one cannot beat Eval_{3v}.

Theorem 5. *Let 3VL′ be a three-valued logic with the same ordering on truth values* $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ *as 3VL, and let* Eval'_{3v} *be a syntactic evaluation procedure based on 3VL′ that has correctness guarantees w.r.t.* Sem_{3v}. *Then* Eval_{3v} *is more informative than* Eval'_{3v}.

For procedures that are not syntactic, we can do better than Eval_{3v}. As an extreme case, suppose we have an FO formula $\varphi(x)$ and we start the evaluation by checking if φ is equivalent to *true*. If it is, we output adom(D); otherwise we proceed with Eval_{3v}. Clearly, we get more information than from Eval_{3v} alone, but the problem is that checking whether φ is a tautology is undecidable. However, this suggests that some manipulations of formulae can help, and we now give one example that improves upon Eval_{3v}, while retaining good data complexity.

By $\mathsf{nnf}(\varphi)$ we mean the negation normal form (NNF) of φ, that is, a formula equivalent to φ which is in prenex normal form, with all the negations pushed to atomic formulae. As long as a logic has De Morgan's laws, conversion to NNF can be done purely syntactically; it is easy to verify that De Morgan's laws hold in the logics we deal with (3VL, 4VL).

Let $\bar{x} = (x_1, \ldots, x_k)$ be the tuple of free variables of φ. Given an atomic subformula ψ of φ, a database D and a tuple $\bar{a} = (a_1, \ldots, a_k)$ of elements of adom(D), we let $F_{\psi}(\varphi, \bar{a})$, or simply F_{ψ} when φ and $\bar{a}$ are clear from the context, be the set of all atomic subformulae ψ' of φ such that the result of replacing each x_i with a_i, for $i \leq k$, in ψ and ψ' gives the same formula. We say that ψ is *redundant* in φ w.r.t. D and $\bar{a}$ if

1) $F_{\psi}(\varphi, \bar{a})$ contains a formula that occurs positively and a formula that occurs negatively in φ;
2) $\mathsf{Eval}_{3v}(\psi, D, \bar{a}) = \mathbf{u}$; and
3) $\mathsf{Eval}_{3v}(\varphi[\mathbf{t}/F_{\psi}], D, \bar{a}) = \mathsf{Eval}_{3v}(\varphi[\mathbf{f}/F_{\psi}], D, \bar{a})$, where $\varphi[\mathbf{t}/F_{\psi}]$ and $\varphi[\mathbf{f}/F_{\psi}]$ are obtained by replacing all occurrences of formulae in F_{ψ} in φ by *true* and *false*, respectively.

For a quantifier-free formula φ in NNF, we let

$$\mathsf{Eval}_{\mathrm{nnf}}(\varphi, D, \bar{a}) = \max_{\preccurlyeq}\{\mathsf{Eval}_{3v}(\varphi, D, \bar{a}), \mathsf{Eval}_{3v}(\varphi', D, \bar{a})\}$$

where φ' ranges over formulae $\varphi[\mathbf{t}/F_{\psi}]$, in which ψ is redundant. It can be shown that the maximum always exists, hence the above is well-defined.

For an arbitrary formula φ, we define $\mathsf{Eval}_{\mathrm{nnf}}(\varphi, D, \bar{a})$ as $\mathsf{Eval}_{\mathrm{nnf}}\big(\mathsf{nnf}(\varphi), D, \bar{a}\big)$, where the rules for quantifiers $\exists$ and $\forall$ are the same in $\mathsf{Eval}_{\mathrm{nnf}}$ and in Eval_{3v}. Recall that, in NNF formulae, Boolean connectives occur only in the quantifier-free part, which is handled by the above rule. The conversion into NNF depends only on the size of φ, and the computation of the redundant atomic subformulae of φ, for a fixed φ, requires a constant number of calls to Eval_{3v}. Hence, $\mathsf{Eval}_{\mathrm{nnf}}$ has the same data complexity of Eval_{3v}, that is, AC^0. Moreover, we can show:

Proposition 8. *The evaluation procedure* $\mathsf{Eval}_{\mathrm{nnf}}$ *has correctness guarantees with respect to* Sem_{3v} *and it is more informative than* Eval_{3v}. *In fact*

$$\mathsf{Eval}^{\mathbf{t}}_{3v}(\varphi, D) \subseteq \mathsf{Eval}^{\mathbf{t}}_{\mathrm{nnf}}(\varphi, D) \subseteq \Box_{\bot}(\varphi, D)$$

and both inclusions can be proper.

As an example, consider the query $\varphi(x) = \exists y \, \big(R(x, y) \wedge (y = 1 \vee y \neq 1)\big)$ and a database D where $R^D = \{(1, \bot)\}$. Then, $\mathsf{Eval}_{\mathrm{nnf}}(\varphi, D, 1) = \mathbf{t}$ and $\mathsf{Eval}_{3v}(\varphi, D, 1) = \mathbf{u}$. The SQL counterpart of the query φ is `SELECT R.A FROM R WHERE R.B=1 OR R.B<>1`, which returns the empty set, whereas the certain answer is 1. Here, Eval_{3v} gives the result of the SQL evaluation, while $\mathsf{Eval}_{\mathrm{nnf}}$ gives us the certain answer.

7 Conclusion

The main message of the paper is that with the help of many-valued logics, certain answers can be efficiently approximated, and the structure of truth values in the logic lets us obtain more refined information than just certain answers themselves. Our results also bridge the general approach of Libkin (2016a) to defining and computing certain answers with many-valued procedures as practiced by commercial DBMSs. In particular, we have shown how to define the semantics of many-answers, and how to ensure that queries preserve informativeness of databases.

As for future work, we would like to apply these ideas in some of the scenarios where certain answers are the preferred query semantics, and to look at different semantics of incompleteness. For the first direction, natural candidates are applications such as data integration and exchange (Arenas et al. 2014; Lenzerini 2002), where efficient query answering is hard to achieve for full FO, and where constraints make the process even more complicated (Calì, Lembo, and Rosati 2003; Calì et al. 2004). Note that our approach, not being restricted to a fragment of FO, makes it easy to incorporate constraints.

A slightly different direction is to see if our many-valued approach could be used in knowledge-base reasoning tasks where three-valued logic was previously used to generate sound evaluations (Levesque 1998; Liu and Levesque 2003). While technically different, the approaches share the idea of using many-valuedness for efficient approximation, and Liu and Levesque (2003) further showed how database evaluation techniques could be used for reasoning with knowledge bases.

For other semantics, one obvious candidate is open-world, which is commonly used, but makes computing certain answers harder due to complexity considerations. Still there is hope of going beyond existential positive queries using many-valuedness: for instance, (Grahne, Moallemi, and Onet 2015) showed how to use Belnap's four-valued logic to compute certain answers for conjunctive queries with negated atoms. In the opposite direction, one can look at semantics that are more restrictive; e.g., semantics of uncertainty represented by a finite number of possible worlds, as in the work by Olteanu, Koch, and Antova (2008).

Acknowledgments We thank anonymous referees and Eugenia Ternovska for their comments. Part of this work was done when the first author visited University of Edinburgh. Supported by EPSRC grants J015377 and M025268.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Abiteboul, S.; Kanellakis, P.; and Grahne, G. 1991. On the representation and querying of sets of possible worlds. *Theoretical Computer Science* 78(1):158–187.

Arenas, M.; Barceló, P.; Libkin, L.; and Murlak, F. 2014. *Foundations of Data Exchange*. Cambridge Univ. Press.

Arieli, O., and Avron, A. 1996. Reasoning with logical bilattices. *J. Logic, Language and Information* 5(1):25–63.

Arieli, O., and Avron, A. 1998. The value of the four values. *Artif. Intell.* 102(1):97–141.

Barrington, D. A.; Immerman, N.; and Straubing, H. 1990. On uniformity within NC^1. *JCSS* 41(3):274–306.

Belnap, N. D. 1977. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*. D. Reidel. 8–37.

Bertossi, L. 2011. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers.

Calì, A.; Calvanese, D.; De Giacomo, G.; and Lenzerini, M. 2004. Data integration under integrity constraints. *Inf. Syst.* 29(2):147–163.

Calì, A.; Lembo, D.; and Rosati, R. 2003. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, 260–271.

Calvanese, D.; De Giacomo, G.; Lembo, D.; Lenzerini, M.; and Rosati, R. 2007. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reasoning* 39(3):385–429.

Date, C., and Darwen, H. 1996. *A Guide to the SQL Standard*. Addison-Wesley.

Doherty, P.; Lukaszewicz, W.; Skowron, A.; and Szalas, A. 2006. *Knowledge Representation Techniques - A Rough Set Approach*. Springer.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput.* 9(3/4):365–386.

Gessert, G. H. 1990. Four valued logic for relational database systems. *SIGMOD Record* 19(1):29–35.

Gheerbrant, A.; Libkin, L.; and Sirangelo, C. 2014. Naïve evaluation of queries over incomplete databases. *ACM Trans. Database Syst.* 39(4):1–31.

Ginsberg, M. L. 1988. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence* 4:265–316.

Grahne, G.; Moallemi, A.; and Onet, A. 2015. Intuitionistic data exchange. In *9th Alberto Mendelzon Workshop*.

Grahne, G. 1991. *The Problem of Incomplete Information in Relational Databases*. Springer.

Halevy, A.; Rajaraman, A.; and Ordille, J. 2006. Data integration: The teenage years. In *VLDB*, 9–16.

Imielinski, T., and Lipski, W. 1984. Incomplete information in relational databases. *Journal of the ACM* 31(4):761–791.

Kontchakov, R.; Lutz, C.; Toman, D.; Wolter, F.; and Zakharyaschev, M. 2011. The combined approach to ontology-based data access. In *IJCAI*, 2656–2661.

Lenzerini, M. 2002. Data integration: a theoretical perspective. In *ACM PODS*, 233–246.

Levesque, H. J. 1998. A completeness result for reasoning with incomplete first-order knowledge bases. In *KR*, 14–23.

Libkin, L. 2014. Incomplete information: what went wrong and how to fix it. In *PODS*, 1–13.

Libkin, L. 2016a. Certain answers as objects and knowledge. *Artificial Intelligence* 232:1–19.

Libkin, L. 2016b. SQL's three-valued logic and certain answers. *ACM Trans. Database Syst.* 41(1):1–28.

Lipski, W. 1984. On relational algebra with marked nulls. In *ACM PODS*, 201–203.

Liu, Y., and Levesque, H. J. 2003. A tractability result for reasoning with incomplete first-order knowledge bases. In *IJCAI*, 83–88.

Lutz, C.; Seylan, I.; and Wolter, F. 2015. Ontology-mediated queries with closed predicates. In *IJCAI*, 3120–3126.

Olteanu, D.; Koch, C.; and Antova, L. 2008. World-set decompositions: expressiveness and efficient algorithms. *Theoretical Computer Science* 403(2-3):265–284.

Paterson, M., and Wegman, M. N. 1978. Linear unification. *JCSS* 16(2):158–167.

Reiter, R. 1977. On closed world data bases. In *Logic and Data Bases*, 55–76.

van der Meyden, R. 1998. Logical approaches to incomplete information: A survey. In *Logics for Databases and Information Systems*, 307–356.