

Logic, Languages, and Rules for Web Data Extraction and Reasoning over Data

Georg Gottlob¹, Christoph Koch², and Andreas Pieris³(✉)

¹ University of Oxford, Oxford, UK
`georg.gottlob@cs.ox.ac.uk`

² École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
`christoph.koch@epfl.ch`

³ University of Edinburgh, Edinburgh, Scotland
`apieris@inf.ed.ac.uk`

Abstract. This paper gives a short overview of specific logical approaches to data extraction, data management, and reasoning about data. In particular, we survey theoretical results and formalisms that have been obtained and used in the context of the Lixto Project at TU Wien, the DIADEM project at the University of Oxford, and the VADA project, which is currently being carried out jointly by the universities of Edinburgh, Manchester, and Oxford. We start with a formal approach to web data extraction rooted in monadic second order logic and monadic Datalog, which gave rise to the Lixto data extraction system. We then present some complexity results for monadic Datalog over trees and for XPath query evaluation. We further argue that for value creation and for ontological reasoning over data, we need existential quantifiers (or Skolem terms) in rule heads, and introduce the Datalog[±] family. We give an overview of important members of this family and discuss related complexity issues.

1 Introduction

“The web is the largest database” is a sentence one nowadays can hear quite frequently. However, this statement is not really true. The web, including the deep web, is certainly the largest data repository, but not a *database*. In a database, data is homogeneously formatted, and can be retrieved efficiently and uniformly via query languages. Web data, even when it is about the same type of items (say, used cars or any other consumer good) appears in a different format on many different websites. There is, moreover, no uniform query or retrieval mechanism. In order to be able to query such data, we thus have to extract it from the different web sources, recast it into a single format, and, if appropriate, store it in a single database. This process is called *web data extraction*, and the programs that extract data from the web are called *wrappers*.

The wrapping problem is often seen as a software and web-engineering task, but has also been addressed by a substantial amount of systems-oriented research work, see e.g. TSIMMIS [55], FLORID [46], DEByE [44], W4F [56], XWrap [45],

Lixto [4, 6, 29] and Diadem [22], some of which led to commercial spin-outs. Moreover, in [27], a logical theory of data extraction has been developed that has led people to consider monadic Datalog as a logical language for data extraction, which has, in turn, been the basis of a more practical logical language implemented in the Lixto system. In Sect. 2, which is a slightly shortened exposition of material from [29] (which in turn summarizes [27]), we will give a short survey of the logical approach to web data extraction.

Web documents in HTML are essentially labeled trees, where many labels correspond to formatting instructions for data presentation (such as `<table>`, `<td>`, or `<header1>`, and so on) and where the actual data items reside at the leaf level. Thus, rather than imposing a logical structure on the data, the labels in HTML take care of the display format and make sure that a web page displayed in a browser meets the eye of the beholder. However, XML¹, a well-known language quite similar to HTML, allows one to impose a tree-shaped logical structure on data. From a conceptual point of view, this generalizes the “flat” relational data format. Special query language such as XPath², XQuery³, and XSLT⁴ have been designed for XML databases. With some minor additions, monadic Datalog can be used to simulate the core fragment of XPath [26], which indicates that core XPath is not more expressive than monadic Datalog. This observation gave rise to complexity studies of XPath evaluation whose basic results will be summarized in Sect. 3.

Once data is extracted, one usually wants to combine it with other extracted data and corporate data from local databases. In addition, some cleaning, reasoning and further provisioning tasks have to be performed. All this together is called *data wrangling* [23]. Apparently, languages for data wrangling purposes should be able to perform complex data transformation, data exchange, data integration and ontological reasoning tasks. However, Datalog, let alone monadic Datalog, is not powerful enough for performing such tasks. In Sect. 4, which is based on the longer survey [10], we argue that the crucial limitation of Datalog is the fact that it is not able to infer the existence of new objects, which are not already in the extensional database. We then proceed to introduce Datalog[±], a family of logical languages that extend Datalog with key modeling features such as existential quantifiers in rule heads, which allow to infer the existence of new objects. We give an overview of important members of this family and discuss related complexity issues.

2 Logical Foundations of Web Data Extraction

2.1 Desiderata for Wrapping Languages

To allow for a foundational study of wrapping languages, we first need to establish criteria that allow us to compare such languages. In [27], four desiderata

¹ <https://www.w3.org/TR/1998/REC-xml-19980210>.

² <http://www.w3c.org/TR/xpath/>.

³ <https://www.w3.org/XML/Query/>.

⁴ <http://www.w3.org/TR/xslt>.

were proposed that a good wrapping language should satisfy. In particular, such a language should

- (i) have a solid and well-understood theoretical foundation,
- (ii) provide a good trade-off between complexity and the number of practical wrappers that can be expressed,
- (iii) be easy to use as a wrapper programming language, and
- (iv) be suitable for incorporation into visual tools.

The core notion that we base our wrapping approach on is that of an *information extraction function*, which takes a labeled unranked tree (representing a Web document) and returns a subset of its nodes. A wrapper is a program which implements one or several such functions, and thereby assigns unary predicates to document tree nodes. Based on these predicate assignments and the structure of the input tree, a new data tree can be computed as the result of the information extraction process in a natural way, along the lines of the input tree, but using the new labels and omitting nodes that have not been relabeled (by some form of tree minor computation).

Given a set of information extraction functions, one natural way to wrap an input tree t is to compute a new label for each node n (or filter out n) as a function of the predicates assigned using the information extraction functions. The output tree is computed by connecting the resulting labeled nodes using the (transitive closure of) the edge relation of t , preserving the document order of t . In other words, the output tree contains a node if a predicate corresponding to an information extraction function was computed for it, and contains an edge from node v to node w if there is a directed path from v to w in the input tree, both v and w were assigned information extraction predicates, and there is no node on the path from v to w (other than v and w) that was assigned information extraction predicates. We do not formalize this operation here; the natural way of doing this is obvious.

That way, we can take a tree, re-label its nodes, and declare some of them as irrelevant, but we cannot significantly transform its original structure. This coincides with the intuition that a wrapper may change the presentation of relevant information, its packaging or data model (which does not apply in the case of *Web wrapping*), but does not handle substantial data transformation tasks. We believe that this captures the essence of wrapping.

We assume unary queries in monadic second-order logic (MSO) over trees as the expressiveness yardstick for information extraction functions. MSO over trees is well-understood theory-wise [15, 18, 20, 58] (see also [59]) and is quite expressive. In fact, it is considered by many as the language of choice for defining expressive node-selecting queries on trees (see e.g. [27, 43, 53, 54]; [57] acknowledges the role of MSO but argues for *even stronger* languages). In our experience, when considering a wrapping system that lacks this expressive power, it is usually quite easy to find real-life wrapping problems that cannot be handled (see also the related discussion on MSO expressiveness and node-selecting queries in [43]).

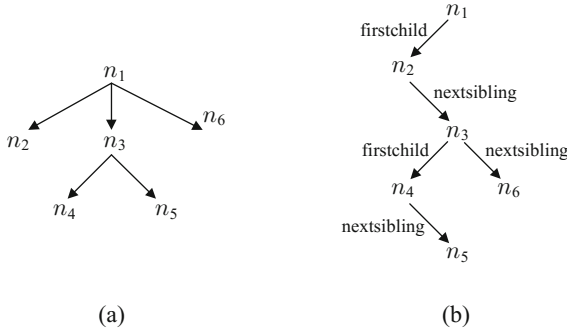


Fig. 1. (a) An unranked tree and (b) its representation using the binary relations “firstchild” (\swarrow) and “nextsibling” (\searrow).

In this section, we discuss *monadic Datalog* over trees, a simple form of the logic-based language Datalog, as a wrapper programming language. Monadic Datalog satisfies desiderata (i) to (iv) raised above. Monadic Datalog is the logical core of Elog [5], which is the internal language of the Lixto system. Elog extends monadic Datalog by features for handling the most common tasks of web navigation and visual wrapper-definition. Elog is strictly more expressive than MSO. For a detailed description of Elog, which we will not further discuss here, see [5]. For a formal study of Elog and a comparison to other wrapping languages, see [27, 28].

A monadic Datalog program can compute a *set* of unary queries (“information extraction functions”) at once. Each intensional predicate of a program selects a subset of dom and can be considered to define one information extraction function. However, in general, not all intensional predicates define information extraction functions. Some have to be declared as auxiliary.

2.2 Tree Structures

Trees are defined in the normal way and have at least one node. We assume that the children of each node are in some fixed order. Each node has a label taken from a finite nonempty set of symbols Σ , the alphabet⁵. We consider only *unranked* finite trees, which correspond closely to parsed HTML or XML documents. In an unranked tree, each node may have an arbitrary number of children. An unranked ordered tree can be considered as a structure

$$t_{ur} = \langle \text{dom}, \text{root}, \text{leaf}, (\text{label}_a)_{a \in \Sigma}, \text{firstchild}, \text{nextsibling}, \text{lastsibling} \rangle$$

where “dom” is the set of nodes in the tree, “root”, “leaf”, “lastsibling”, and the “label_a” relations are unary, while “firstchild” and “nextsibling” are binary.

⁵ In this simple model, unrestricted sets of tags as well as string and attribute values are assumed to be encoded as lists of character symbols modeled as subtrees in our document tree.

All relations are defined according to their intuitive meanings. “root” contains exactly one node, the root node. “leaf” consists of the set of all leaves. “firstchild(n_1, n_2)” is true iff n_2 is the leftmost child of n_1 ; “nextsibling(n_1, n_2)” is true iff, for some i , n_1 and n_2 are the i -th and $(i + 1)$ -th children of a common parent node, respectively, counting from the left (see also Fig. 1). $\text{label}_a(n)$ is true iff n is labeled a in the tree. Finally, “lastsibling” contains the set of rightmost children of nodes. (The root node is not a last sibling, as it has no parent.) Whenever the structure t may not be clear from the context, we state it as a subscript of the relation names (as e.g. in dom_t , root_t , \dots). By default, we will always assume trees to be represented using the schema (signature) outlined above, and will refer to them as τ_{ur} .

The *document order* relation \prec is a natural total ordering of dom used in several XML-related standards. It is defined as the order in which the opening tags of document tree nodes are first reached when reading an HTML or XML document (as a flat text file) from left to right.

2.3 Monadic Datalog

We assume the function-free logic programming syntax and semantics of the *Datalog* language and refer to [1] for a detailed survey of Datalog. *Monadic Datalog* [14, 27] is obtained from full Datalog by requiring all intensional predicates to be unary. By unary query, we denote a function that assigns a predicate to some elements of dom (or, in other words, selects a subset of dom). For monadic Datalog, one obtains a unary query by distinguishing one intensional predicate as the *query predicate*. By *signature*, we denote the (finite) set of all extensional predicates (with fixed arities) available to a Datalog program. By default, we use the signature τ_{ur} for unranked trees.⁶

Example 1. The monadic Datalog program over τ_{ur}

$$\textit{Italic}(x) \leftarrow \text{label}_i(x) \tag{1}$$

$$\textit{Italic}(x) \leftarrow \textit{Italic}(x_0), \text{firstchild}(x_0, x) \tag{2}$$

$$\textit{Italic}(x) \leftarrow \textit{Italic}(x_0), \text{nextsibling}(x_0, x) \tag{3}$$

computes, given an unranked tree (representing an HTML parse tree), all nodes whose contents are displayed in italic font (i.e., for which an ancestor node in the parse tree corresponds to a well-formed piece of HTML of the form $\langle i \rangle \dots \langle /i \rangle$ and is thus labeled “i”). The program uses the intensional predicate, *Italic*, as the query predicate.

Monadic second-order logic (MSO) extends first-order logic by quantification over set variables, i.e., variables ranging over sets of nodes, which coexist with

⁶ Note that our tree structures contain some redundancy (e.g., a leaf is a node x such that $\neg(\exists y)\text{firstchild}(x, y)$), by which (monadic) Datalog becomes as expressive as its *semipositive* generalization. Semipositive Datalog allows to use the complements of extensional relations in rule bodies.

first-order quantification of variables ranging over single nodes. A unary MSO *query* is defined by an MSO formula φ with one free first-order variable. Given a tree t , it evaluates to the set of nodes $\{x \in \text{dom} \mid t \models \varphi(x)\}$. The following holds for arbitrary finite structures:

Proposition 2 (Folklore). *Each monadic Datalog query is MSO-definable.*

Here, our main measure of query evaluation cost is *combined complexity*, i.e. where both the database and the query (or program) are considered variable. Later, we will also be interested in *data complexity*, where the query (or program) is fixed and only the database is considered variable.

Proposition 3. (see e.g. [27]) *Monadic Datalog (over arbitrary finite structures) is NP-complete w.r.t. combined complexity.*

2.4 Monadic Datalog over Trees

By restricting our structures to trees, monadic Datalog acquires a number of additional nice properties. First,

Theorem 4 [27]. *Over τ_{ur} , monadic Datalog has $O(|\mathcal{P}| \cdot |\text{dom}|)$ combined complexity (where $|\mathcal{P}|$ is the size of the program and $|\text{dom}|$ the size of the tree).*

This follows from the fact that all binary relations in τ_{ur} have bidirectional functional dependencies; for instance, each node has at most one first child and is the first child of at most one other node. Thus, given a program \mathcal{P} , an equivalent ground program can be computed in time $O(|\mathcal{P}| \cdot |\text{dom}|)$, while ground programs can be evaluated in linear time [52].

A unary query over trees is MSO-definable exactly if it is definable in monadic Datalog.

Theorem 5 [27]. *Each unary MSO-definable query over τ_{ur} is definable in monadic Datalog over τ_{ur} .*

(The other direction follows from Proposition 2.) Judging from our experience with the Lixto system, real-world wrappers written in monadic Datalog are small. Thus, in practice, we do not trade the complexity compared to MSO (for which query evaluation is known to be PSPACE-complete) for considerably expanded program sizes.

Each monadic Datalog program over trees can be efficiently rewritten into an equivalent program using only very restricted syntax. This motivates a normal form for monadic Datalog over trees.

Definition 6. A monadic Datalog program \mathcal{P} over τ_{ur} is in *Tree-Marking Normal Form* (TMNF) if each rule of \mathcal{P} is of one of the following three forms:

- (1) $p(x) \leftarrow p_0(x)$,
- (2) $p(x) \leftarrow p_0(x_0), B(x_0, x)$.
- (3) $p(x) \leftarrow p_0(x), p_1(x)$.

where the unary predicates p_0 and p_1 are either intensional or of τ_{ur} and B is either R or R^{-1} , where R is a binary predicate from τ_{ur} . \square

In the next result, the signature for unranked trees may extend τ_{ur} to include the “child” relation – likely to be the most common form of navigation in trees.

Theorem 7 [27]. *For each monadic Datalog program \mathcal{P} over $\tau_{ur} \cup \{child\}$, there is an equivalent TMNF program over τ_{ur} which can be computed in time $O(|\mathcal{P}|)$.*

From the above discussion, we conclude that monadic Datalog has the expressive power of our yardstick MSO (on trees), can be evaluated efficiently, and is a *good* (easy to use) wrapper programming language. Indeed, with respect to the desiderata listed in Subsect. 2.1, we point out that:

1. The existence of the normal form TMNF demonstrates that rules in monadic Datalog never have to be long or intricate.
2. The monotone semantics makes the wrapper programming task quite modular and intuitive. Differently from an automaton definition that usually has to be understood entirely to be certain of its correctness, adding a rule to a monadic Datalog program usually does not change its meaning completely, but *adds* to the functionality.
3. Wrappers defined in monadic Datalog are implemented as queries, whose definitions can be local and only need to consider as much context as required by the query conditions. This distinguishes them from tree automata, which, even in flavors able to define monadic queries, always recognise a language of trees and conceptually traverse the entire input tree. This makes tree automata more brittle in real-world wrapping scenarios, and causes them to require a greater effort from the programmer, much of which is directed towards accepting a large enough tree language, rather than the essence of the wrapping task at hand.

Thus, monadic Datalog over trees as a framework for Web information extraction satisfies the first three of our desiderata stated in Subsect. 2.1: efficient evaluation, appropriate expressiveness, and suitability as a practical wrapper programming language. Only the fourth desideratum – the visual specification of wrappers – is not addressed here; we refer the interested reader to [6, 29], where it is clearly explained that monadic Datalog paradigm is ideally suited for representing wrappers generated by a visual wrapper definition process using successive restriction and generalization steps.

3 The Complexity of XPath Query Evaluation

We have seen in Theorem 4 that monadic Datalog over trees defined by unary relations and the binary relations “firstchild”, “nextsibling”, and “lastsibling” can be solved in time linear in the size of the program and linear in the size

of the tree. Relations such as “child” play an important role in various query languages on trees, such as XPath (and thus, XQuery and XSLT); there, they are called *axes*.

There are two main modes of navigation in trees, horizontal and vertical. For horizontal navigation, one can distinguish between navigating among sibling nodes and among nodes – intuitively – further left or right in the tree (the “following” axis in XPath). The most natural axis relations are thus *Child*, *Child**, *Child⁺*, *Nextsibling*, *Nextsibling**, *Nextsibling⁺*, and *Following*, where

$$\textit{Following}(x, y) := \exists z_1, z_2 \textit{Child}^*(z_1, x) \wedge \textit{Nextsibling}^+(z_1, z_2) \wedge \textit{Child}^*(z_2, y).$$

Note that if we consider complexity rather than expressiveness, we do not need to deal with relations such as *FirstChild* in addition; we may assume a unary predicate *Firstsibling* such that

$$\textit{FirstChild}(x, y) \Leftrightarrow \textit{Child}(x, y) \wedge \textit{Firstsibling}(y).$$

A natural question is to ask for the complexity of monadic Datalog programs over these axes, or, to start with a more basic problem, conjunctive queries (which can be seen as Datalog programs containing only a single nonrecursive rule). Note that conjunctive queries over trees also have natural applications in computational linguistics, term rewriting, and data integration [32].

In the case that all individual rule-bodies are acyclic (conjunctive queries), it is known from [27] that monadic Datalog over arbitrary axes can be evaluated in linear time. However, in data extraction, as well as in many other practical contexts, programs with cyclic rule bodies naturally arise.

As already observed in Proposition 3, while full Datalog is EXPTIME-complete (see, e.g., [16]), monadic Datalog over arbitrary finite structures is in NP (actually, NP-complete). For a lower bound on trees, it is known [49] that already Boolean conjunctive queries over structures of the form $\langle (P_i)_i, \textit{child}, \textit{child}^* \rangle$ are NP-hard w.r.t. combined complexity.

A detailed study of the tractability frontier of conjunctive queries over trees is presented in [32]. As observed, the subset-maximal polynomial cases of axis sets are

- $\{\textit{child}^+, \textit{child}^*\}$,
- $\{\textit{child}, \textit{nextsibling}, \textit{nextsibling}^+, \textit{nextsibling}^*\}$, and
- $\{\textit{following}\}$.

That is, for each class of conjunctive queries over a subset of one of these three sets and over unary relations, the query evaluation problem is polynomial (with respect to combined complexity). We have the dichotomy that for all other cases of conjunctive queries using our axis relations (e.g. *Child* and *Child⁺*), the problem is NP-complete. Obviously, the complexity of monadic Datalog over a given set of axes is always the same as that of conjunctive queries over the same axes.

The special case that queries are acyclic is also worth studying, since the probably most important node-selecting query language on trees, XPath, is naturally tree-shaped. All XPath engines available in 2002 took exponential time in the worst case to process XPath [30]. However,

Theorem 8 [30]. *XPath 1 is in PTIME w.r.t. combined complexity.*

This result is based on a dynamic programming algorithm which, in an improved form [30], yielded the first XPath engine guaranteed to run in polynomial time.

Most people use only the most common features of XPath, so it is worthwhile to study restrictive fragments of this language. In [30], the *Core XPath* has been introduced, the navigational fragment of XPath, which includes both horizontal and vertical tree navigation with axes, node tests, and boolean combinations of condition predicates. As shown there, Core XPath can be evaluated in time linear in the size of the database and linear in the size of the query. However,

Theorem 9 [31]. *Core XPath is P-hard w.r.t. combined complexity.*

This property – shared by XPath, of which Core XPath is a strict fragment – renders it highly unlikely that query evaluation is massively parallelizable (= in the complexity class NC , c.f. [40]) or that algorithms exist that take less than a polynomial amount of space for query processing. Interestingly, if we remove negation in condition predicates, the complexity of Core XPath is reduced to LOGCFL, a parallel complexity class in NC_2 [31].

Theorem 10 [31]. *Positive Core XPath is LOGCFL-complete w.r.t. combined complexity.*

This generalizes to a very large fragment of full XPath (called pXPath), from which besides negation only few very minor features have to be removed to obtain that

Theorem 11 [31]. *pXPath is LOGCFL-complete w.r.t. combined complexity.*

Further results on the complexity of various fragments of XPath 1 can be found in [31]. Positive Core XPath queries correspond to acyclic positive queries over axis relations. Interestingly, each conjunctive query over axis relations can be mapped to an equivalent acyclic positive query, however there are no polynomial translations for doing this [32]. Thus,

Corollary 12. *For ever conjunctive query over trees, there is an equivalent positive Core XPath query.*

Of course, when talking about conjunctive queries over trees, we assume that all binary relations in the signature are relations from our set of axes.

Finally, Core XPath queries can be mapped to monadic Datalog in linear time. The slightly curious fact here is that this remains true in the presence of negation in Core XPath (for which no analogous language feature exists in Datalog.)

Theorem 13 [21]. *Each Core XPath query can be translated into an equivalent TMNF query in linear time.*

4 Datalog[±]: A Family of Logical Languages

It is generally agreed that Datalog is a powerful language with several different applications. We have already discussed that the monadic fragment of Datalog gives rise to a good wrapping language that can be used for web data extraction purposes. Moreover, Datalog has been used as an inference engine for knowledge processing within several software tools, and has gained popularity in the context of, e.g., source code querying and program analysis, and modeling distributed systems.

Although Datalog is a powerful rule-based formalism, it is not able to infer the existence of new objects that are not already in the extensional database. For a number of applications, however, it would be desirable that a Datalog extension could be able to express the existence of certain values that are not necessarily from the domain of the extensional database. This can be achieved by allowing existentially quantified variables in rule heads. Let us give a couple of brief examples of such applications.

Data Exchange. When data needs to be transposed or copied from one relational database to another one, the problem of heterogeneous schemas often arises. Imagine, for example, company ACME stores data about their employees in a relation `EmpACME` with schema $(\text{Emp\#}, \text{Name}, \text{Address}, \text{Salary})$, while the FOO corporation does not store employees' addresses, but only phone numbers, keeping their employee data in a relation `EmpFOO` having schema $(\text{Emp\#}, \text{Name}, \text{Phone}, \text{Salary})$. Imagine ACME is acquired by FOO and the ACME employee data ought to be transferred into the FOO database, although the phone numbers of the ACME employees are not (currently) known. This could be achieved by a rule of the form:

$$\text{EmpACME}(e, n, a, s) \rightarrow \exists p \text{ EmpFOO}(e, n, p, s),$$

where phone numbers are simply existentially quantified. In practice, each phone number is stored by a different (labeled) *null value*, representing a globally existentially quantified variable (i.e., a kind of Skolem constant). Advanced data management systems such as Clio [51] have been developed, that effectively manage data-exchange mappings, handle existential nulls, and allow one to query relations with nulls. In database theory, a rule of the above form is actually called a *tuple-generating dependency (TGD)*, while in the KR community is known as *existential rule*; henceforth, we adopt the term TGD. In addition to TGDs, *equality-generating dependencies (EGDs)* are often used. They cover the well-known key constraints and functional dependencies that have been studied for a long time [1]. For example, we may impose that every ACME employee has only one phone number stored. This may be expressed as a Datalog rule with an equality in the head:

$$\text{EmpFOO}(e, n, p, s), \text{EmpFOO}(e, n', p', s') \rightarrow p = p'.$$

The data exchange literature insists on *finite target relations* because it is assumed that these relations are actually stored. It is thus important in this

context to restrict our syntax to make sure that only a *finite number of different null values* will be invented.

Ontology Querying. *Description logics (DLs)* [3] are used to formalize so-called ontological knowledge about relationships between objects, entities, and classes in a certain application domain. For example, we could express that every person has exactly one father who, moreover, is himself a person, by the following DL clauses, where *Person* is a set of objects whose initial value is specified in the form of an extensional relation, called *concept*, and where *HasFather* is a binary relation, a so-called *role* in DL terminology: (i) $\text{Person} \sqsubseteq \exists \text{HasFather}$, (ii) $\exists \text{HasFather}^- \sqsubseteq \text{Person}$, (iii) (funct HasFather). In an appropriate extension of Datalog, the same can be expressed as:

$$\begin{aligned} \text{Person}(x) &\rightarrow \exists y \text{HasFather}(x, y), \\ \text{HasFather}(x, y) &\rightarrow \text{Person}(y), \\ \text{HasFather}(x, y), \text{HasFather}(x, y') &\rightarrow y = y'. \end{aligned}$$

Note that here the relation *Person*, which is supplied in the input with an initial value, is actually modified. Therefore, we no longer require (as in standard Datalog) that extensional relation symbols cannot occur in rule heads.

DLs usually rely on classical first-order (FO) semantics, and so arbitrary models (finite or infinite) are considered. In the above example, models with infinite chains of ancestors are perfectly legal. Rather than “materializing” such models, i.e., computing and storing them, we are interested in reasoning and query answering. For example, whenever the initial value of *Person* is nonempty, then the Boolean conjunctive query

$$\exists x \exists y \exists z (\text{HasFather}(x, y) \wedge \text{HasFather}(y, z))$$

will evaluate to *true*, while the query

$$\exists x \exists y (\text{HasFather}(x, y) \wedge \text{HasFather}(y, x))$$

will evaluate to *false*, because it is false in some models.

To sum up, as we have briefly tried to sketch, some applications as the ones discussed above could possibly profit from appropriate forms of Datalog extended by the possibility of using rules with existential quantifiers in their heads (TGDs), and by several additional features (such as, for example, equality, negation, disjunction, etc.).

Unfortunately, already for sets Σ of TGDs alone, most basic reasoning and query answering problems are undecidable. In particular, checking whether a Boolean conjunctive query evaluates to true w.r.t. a database D and a set Σ of TGDs is undecidable [7]. Worse than that, undecidability holds even in case both Σ and q are *fixed*, and only D is given as input [8]. It is thus important to single out large classes of formalisms for rule sets Σ that

- (i) are based on Datalog, and thus enable a modular rule-based style of knowledge representation,

- (ii) are syntactical fragments of first-order logic so that answering a Boolean query q under Σ for an input database D is equivalent to the classical entailment check $D \wedge \Sigma \models q$,
- (iii) are expressive enough for being useful in real applications in the above mentioned areas,
- (iv) have decidable query answering, and
- (v) have good query answering complexity properties in case Σ and q are fixed. This type of complexity is called *data complexity*, and is an important measure, because we can realistically assume that the extensional database D is the only really large object in the input.

In what follows we report on languages that fulfill these criteria. We dubbed the family of such languages Datalog^\pm , because, as already explained, they add features to Datalog, and on the other hand make some syntactic restrictions in order to fulfill desiderata (iv) and (v). In the rest of the paper, we focus on the key feature of existential quantification, or, in other words, on languages that are based on TGDs.

4.1 Acyclicity

Recall that for data exchange purposes, it is important to ensure that the target instance is finite since it is actually stored. However, executing an arbitrary set of TGDs on an input database, in general, we are forced to build an infinite instance due to the presence of the existentially quantified variables. Consider, for example, the set Σ of TGDs:

$$\text{Person}(x) \rightarrow \exists y \text{HasFather}(x, y) \quad \text{HasFather}(x, y) \rightarrow \text{Person}(y),$$

which states that each person has a father who is also a person. Assuming now that the input database is $D = \{\text{Person}(\text{Bob})\}$, stating that Bob is a person, after executing Σ on D we obtain an infinite instance. Indeed, from the first TGD we conclude that the atom $\text{HasFather}(\text{Bob}, z_1)$ holds, where z_1 is a (labeled) null value, while from the second TGD we obtain that $\text{Person}(z_1)$ holds. But then we can infer that also the atoms $\text{HasFather}(z_1, z_2)$ and $\text{Person}(z_2)$ hold, where z_2 is a fresh labeled null value, and it is apparent that this inference process is infinite. The inference algorithm that we have just described is known in the literature as the *chase procedure* (or simply *chase*) [1].

It is clear that a TGD-based language is suitable for data exchange purposes if, in addition to the desiderata (i)–(v) discussed above, ensures the termination of the chase. Several languages with this property have been proposed; see, e.g., [17, 19, 39, 48]. The general idea underlying all these languages is to pose an acyclicity condition on a graph that encodes how terms are propagated during the execution of the chase procedure. The two most basic formalisms in this family of languages are the classes of *acyclic* (a.k.a. non-recursive) and *weakly-acyclic* sets of TGDs.

Acyclic Sets of TGDs. The definition of this class relies on the notion of the predicate (dependency) graph, which encodes how predicates depend to each other. More precisely, the *predicate graph* of a set Σ of TGDs is a directed graph $G = (V, E)$, where V consists of all the relation symbols in Σ , and E is defined as follows: for each $\sigma \in \Sigma$, for each relation R in the body of σ , and for each relation P in the head of σ ,⁷ $(R, P) \in E$; no other edges occur in E . We say that Σ is acyclic if G is acyclic.

It is not difficult to see that the chase always terminates under acyclic sets of TGDs. This immediately implies the decidability of our main reasoning task, that is, query answering. Given a Boolean conjunctive query q , to decide whether a database D and an acyclic set Σ of TGDs entails q , we simply need to compute the chase instance C w.r.t. D and Σ , and then check whether C satisfies q . We know that:

Theorem 14 [47]. *Query answering under acyclic sets of TGDs is in AC_0 w.r.t. data complexity, and NEXPTIME-complete w.r.t. combined complexity.*⁸

Notice that to explicitly compute the chase under acyclic sets of TGDs takes polynomial time in the size of the database. Thus, to obtain the AC_0 upper bound w.r.t. the data complexity, we need a more refined approach. This is done by unfolding the given set of TGDs (using a resolution-based procedure [34]) in order to construct a (finite) union of conjunctive queries, which is then evaluated over the input database. This allows us to conclude the AC_0 upper bound stated in the above theorem.

Weakly-Acyclic Sets of TGDs. It is clear that acyclic sets of TGDs do not capture plain Datalog. Nevertheless, an acyclicity-based class exists, called weakly-acyclic sets of TGDs, that captures both acyclic sets of TGDs and Datalog. This formalism has been proposed as the main language for data exchange purposes [19]. Weak-acyclicity relies on a slightly more involved graph notion, called position (dependency) graph, which encodes how terms are propagated from one position to another during the chase. Instead of giving the rather long definition, let us explain the key idea via a simple example.

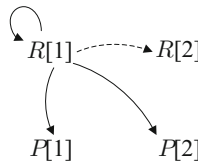


Fig. 2. Position graph.

⁷ For a TGD of the form $b \rightarrow h$, b is called the body, while h is called the head.

⁸ Here, the data complexity is calculated by fixing the set of TGDs and the query, while in the combined complexity we assume that everything is part of the input.

Example 15. Consider the set Σ consisting of the TGDs

$$R(x, y) \rightarrow \exists z R(x, z) \quad R(x, y) \rightarrow P(x, y).$$

The position graph of Σ is shown in Fig. 2. We have an edge from $R[1]$ to itself since in the first TGD the variable x is propagated from the first position of the relation R in the body to the first position of the relation R in the head. Now, observe that at the same time, during the execution of the chase, a null value will be generated at the second position of R ; this is encoded by the dashed edge, called *special*, from $R[1]$ to $R[2]$. The other two (normal) edges are present due to the second TGD.

A normal edge (π, π') keeps track of the fact that a term may propagate from π to π' during the chase. A special edge (π, π'') keeps track of the fact that propagation of a value from π to π' also creates a new value at position π'' . Thus, if there is a cycle in the dependency graph that goes via a special edge, then it is likely that the generation of a null value at certain position will cause the generation of some other null value at the same position, and thus the chase is infinite. A set Σ of TGDs is weakly-acyclic if there is no cycle in its position graph that involves a special edge. We know that:

Theorem 16 [12, 47]. *Query answering under weakly-acyclic sets of TGDs is PTIME-complete w.r.t. data complexity, and 2EXPTIME-complete w.r.t. combined complexity.*

The upper bounds are shown by simply constructing the chase instance C , and then evaluate the input query over C . Notice that the PTIME-hardness is immediately inherited from the fact that weakly-acyclic sets of TGDs capture plain Datalog.

4.2 Guardedness

Although (weakly-)acyclic sets of TGDs are good languages for data exchange, they are not suitable for modeling ontological knowledge. Even the very simple knowledge that each person has a father who is also a person goes beyond weakly-acyclic sets of TGDs. Thus, we need classes of TGDs that do not guarantee the termination of the chase, but still query answering is decidable. In other words, we need languages that allow us to develop methods for reasoning about infinite models without explicitly building them.

Guarded TGDs. A prime example of such a formalism is the class of *guarded* TGDs, inspired by the guarded-fragment of first-order logic. A TGD is called guarded if it has an atom in its body that contains all the body-variables [8]. The reason why we can answer queries under guarded TGDs, even if the chase procedure is infinite, is because the chase instance is tree-like, or, in more formal terms, has bounded tree-width. We know that:

Theorem 17 [8]. *Query answering under guarded TGDs is PTIME-complete w.r.t. data complexity, and 2EXPTIME-complete w.r.t. combined complexity.*

A core fragment of guarded TGDs, which, despite its simplicity, captures features of the most widespread tractable description logics such as DL-Lite, is the class of *linear* TGDs. A TGD is called linear if it has only one atom in its body [9]. As expected, this allows us to show that the complexity of query answering is lower:

Theorem 18 [9, 41]. *Query answering under linear TGDs is in AC_0 w.r.t. data complexity, and PSPACE-complete w.r.t. combined complexity.*

Interestingly, under some fairly weak assumptions, queries to be evaluated under linear TGDs (or corresponding DLs) can be translated into polynomially-sized Datalog programs, or even polynomially-sized first-order formulas to be evaluated directly over input databases. This is discussed in detail in [25, 33, 38].

Weakly-Guarded Sets of TGDs. As for acyclic sets of TGDs, we can define a weak version of guarded TGDs, called *weakly-guarded*, that captures both guarded TGDs and plain Datalog [8]. The key idea is to relax guardedness in such a way that a variable x in the body can be unguarded as long as, during the construction of the chase, x is unified only by constants that already appear in the input database. This seemingly mild relaxation gives rise to a highly expressive language. We know that:

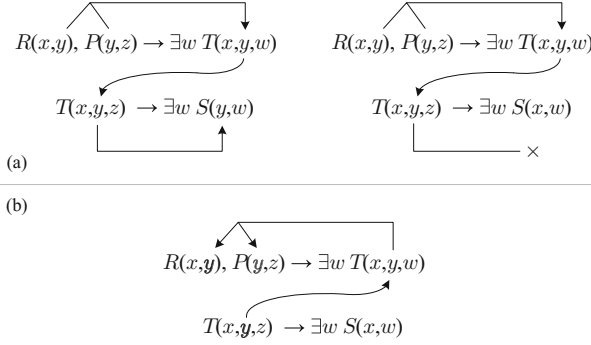
Theorem 19 [8]. *Query answering under weakly-guarded sets of TGDs is EXPTIME-complete w.r.t. data complexity, and 2EXPTIME-complete w.r.t. combined complexity.*

It is interesting, and somehow surprising, that query answering under this class of TGDs is provably intractable even w.r.t. the data complexity. What is even more interesting is the fact that by allowing negation of a very mild form, in particular, stratified negation, weakly-guarded sets of TGDs are powerful enough to capture every database property that can be checked in exponential time, even without assuming an order in the input database. In other words, every Boolean query Q that can be evaluated in exponential time in data complexity, it can be expressed as a pair (Σ, Ans) , where Σ is a weakly-guarded set of TGDs and Ans a 0-ary relation, such that the following holds: D satisfies Q iff D and Σ entails the atomic query Ans , for every database D .

Theorem 20 [37]. *Weakly-guarded sets of TGDs with stratified negation capture EXPTIME, even without assuming ordered databases.*

4.3 Stickiness

Although guardedness is a well-accepted decidability paradigm, with desirable model-theoretic and complexity properties, it is not powerful enough for cap-

**Fig. 3.** Stickiness and marking.

turing knowledge that is inherently non-tree-like. Consider, for example, the following TGDs:

$$\begin{aligned}
 \text{Elephant}(x) &\rightarrow \exists y \text{ HasAncestor}(x, y), \text{Elephant}(y), \\
 \text{Mouse}(x) &\rightarrow \exists y \text{ HasAncestor}(x, y), \text{Mouse}(y), \\
 \text{Elephant}(x), \text{Mouse}(y) &\rightarrow \text{BiggerThan}(x, y),
 \end{aligned}$$

which essentially state that elephants are bigger than mice. It is clear that the first two TGDs are guarded (in fact, linear). However, the third TGD, although it looks simple and harmless, destroys the tree-likeness of the chase instance. Indeed, due to the first two TGDs, the chase will invent infinitely many null values that represent elephants and mice; let E and M be the sets of null values that represent elephants and mice, respectively. Then, the third TGD, will force the chase to compute the cartesian product of E and M , and store it in the binary relation *BiggerThan*. Therefore, the extension of *BiggerThan* in the chase instance C stores an infinite bipartite graph, which in turn implies that the tree-width of C is infinite. This immediately implies that the above set of TGDs cannot be rewritten as a set of guarded TGDs.

Sticky Sets of TGDs. The class of *sticky* sets of TGDs, introduced in [12], is a formalism that allows us to capture non-tree-like knowledge as the one captured by the above example. The key property of stickiness can be described as follows: during the chase, terms that are unified with variables that appear more than once in the body of a TGD (i.e., join variables) are always propagated (or “stick”) to the inferred atoms. This is illustrated in Fig. 3(a); the first set of TGDs is sticky, while the second is not. The formal definition is based on an inductive marking procedure that marks the variables that may violate the semantic property of the chase described above. Roughly, during the base step of this procedure, a variable that appears in the body of a TGD σ but not in the head-atom of σ is marked. Then, the marking is inductively propagated from head to body as shown in Fig. 3(b). Finally, a finite set of TGDs Σ is *sticky* if no TGD in Σ contains two occurrences of a marked variable. We know that:

Theorem 21 [12]. *Query answering under sticky sets of TGDs is in AC_0 w.r.t. data complexity, and EXPTIME-complete w.r.t. combined complexity.*

Weakly-Sticky Sets of TGDs. As one might expect, a weak version of stickiness, which captures both sticky sets of TGDs and plain Datalog, can be defined. The principle under this more expressive language is the same as for weakly-acyclic and weakly-guarded sets of TGDs. Intuitively, we can relax the stickiness condition in such a way that variables that can be unified with finitely many null values during the construction of the chase are not taken into account. It is known that:

Theorem 22 [12]. *Query answering under weakly-sticky sets of TGDs is PTIME-complete w.r.t. data complexity, and 2EXPTIME-complete w.r.t. combined complexity.*

4.4 Further Applications

As already discussed at the beginning of the section, data exchange and ontological reasoning, are applications that could possibly profit from Datalog[±] languages that extend Datalog with existential quantifiers in rule heads. Let us conclude by briefly describing other applications that could profit from languages as the ones discussed above.

RDF and Semantic Web. Various “classical” query languages for RDF and the semantic Web are discussed in [24]. One of the distinctive features of Semantic Web data is the existence of vocabularies with predefined semantics: the *RDF Schema (RDFS)*⁹ and the *Ontology Web Language (OWL)*¹⁰, which can be used to derive logical conclusions from RDF graphs. Thus, it would be desirable to have an RDF query language equipped with reasoning capabilities to deal with these vocabularies. Besides, it has also been recognised that navigational capabilities are of fundamental importance for data models with an explicit graph structure such as RDF, and, more generally, it is also agreed that a general form of recursion is a central feature for a graph query language. Thus, it would also be desirable to have an RDF query language with such functionalities. We strongly believe that Datalog[±] languages are well-suited for this purpose. In fact, steps towards this direction have been already made in the recent works [2, 36].

Conceptual Modeling. It has been observed that graphical conceptual modeling formalisms, and in particular UML and ER diagrams, can be faithfully translated into TGDs and EGDs. In fact, core fragments of the above formalisms can be captured via guarded TGDs (with some additional features such as equality) [11, 35]. This is quite beneficial since it provides logical semantics to the above formalism, which in turn allows us to formally study relevant problems such as consistency, i.e., whether a given diagram admits at least one model.

⁹ <http://www.w3.org/TR/rdf-schema>.

¹⁰ <http://www.w3.org/TR/owl-features/>.

Object-Oriented Deductive Databases. It has been shown that formalisms introduced for object-oriented databases can be embedded into Datalog[±], which in turn allows us to exploit existing query answering algorithms. For example, F-Logic Lite, introduced in [13], is a small but expressive subset of F-Logic [42] that can be captured by weakly-guarded sets of TGDs [8].

Ontology-Based Multidimensional Contexts. Data quality assessment and data cleaning are context dependent activities, and thus, context models for the assessment of the quality of a database have been proposed. A context takes the form of a possibly virtual database or a data integration system into which the database under assessment is mapped, for additional analysis, processing, and quality data extraction. The work [50] extends contexts with dimensions, and hence, multidimensional data quality assessment becomes possible. At the core of multidimensional contexts we have ontologies that are modeled using Datalog[±], and, in particular, weakly-sticky sets of TGDs.

Acknowledgements. This work has been supported by the EPSRC Programme Grant EP/M025268/ “VADA: Value Added Data Systems – Principles and Architecture”.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
2. Arenas, M., Gottlob, G., Pieris, A.: Expressive languages for querying the semantic web. In: PODS, pp. 14–26 (2014)
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
4. Baumgartner, R., Flesca, S., Gottlob, G.: Declarative information extraction, web crawling, and recursive wrapping with *Lixto*. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 21–41. Springer, Heidelberg (2001). doi:[10.1007/3-540-45402-0_2](https://doi.org/10.1007/3-540-45402-0_2)
5. Baumgartner, R., Flesca, S., Gottlob, G.: The elog web extraction language. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 548–560. Springer, Heidelberg (2001). doi:[10.1007/3-540-45653-8_38](https://doi.org/10.1007/3-540-45653-8_38)
6. Baumgartner, R., Flesca, S., Gottlob, G.: Visual web information extraction with *lixto*. In: VLDB, pp. 119–128 (2001)
7. Beeri, C., Vardi, M.Y.: The implication problem for data dependencies. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 73–85. Springer, Heidelberg (1981). doi:[10.1007/3-540-10843-2_7](https://doi.org/10.1007/3-540-10843-2_7)
8. Cali, A., Gottlob, G., Kifer, M.: Taming the infinite chase: query answering under expressive relational constraints. J. Artif. Intell. Res. **48**, 115–174 (2013)
9. Cali, A., Gottlob, G., Lukasiewicz, T.: A general datalog-based framework for tractable query answering over ontologies. J. Web Sem. **14**, 57–83 (2012)
10. Cali, A., Gottlob, G., Lukasiewicz, T., Marnette, B., Pieris, A.: Datalog+/-: a family of logical knowledge representation and query languages for new applications. In: LICS, pp. 228–242 (2010)

11. Cali, A., Gottlob, G., Pieris, A.: Ontological query answering under expressive entity-relationship schemata. *Inf. Syst.* **37**(4), 320–335 (2012)
12. Cali, A., Gottlob, G., Pieris, A.: Towards more expressive ontology languages: the query answering problem. *Artif. Intell.* **193**, 87–128 (2012)
13. Cali, A., Kifer, M.: Containment of conjunctive object meta-queries. In: *VLDB*, pp. 942–952 (2006)
14. Cosmadakis, S.S., Gaifman, H., Kanellakis, P.C., Vardi, M.Y.: Decidable optimization problems for database logic programs (preliminary report). In: *STOC*, pp. 477–490 (1988)
15. Courcelle, B.: Graph rewriting: an algebraic and logic approach. In: van Leeuwen, J. (ed.) *Handbook of Theoretical Computer Science*, vol. 2, chap. 5, pp. 193–242. Elsevier Science Publishers B.V. (1990)
16. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. *ACM Comput. Surv.* **33**(3), 374–425 (2001)
17. Deutsch, A., Tannen, V.: Reformulation of XML queries and constraints. In: Calvanese, D., Lenzerini, M., Motwani, R. (eds.) *ICDT 2003*. LNCS, vol. 2572, pp. 225–241. Springer, Heidelberg (2003). doi:[10.1007/3-540-36285-1_15](https://doi.org/10.1007/3-540-36285-1_15)
18. Doner, J.: Tree acceptors and some of their applications. *J. Comput. Syst. Sci.* **4**(5), 406–451 (1970)
19. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. *Theor. Comput. Sci.* **336**(1), 89–124 (2005)
20. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. In: Bussche, J., Vianu, V. (eds.) *ICDT 2001*. LNCS, vol. 1973, pp. 22–38. Springer, Heidelberg (2001). doi:[10.1007/3-540-44503-X_2](https://doi.org/10.1007/3-540-44503-X_2)
21. Frick, M., Grohe, M., Koch, C.: Query evaluation on compressed trees. In: *LICS*, pp. 22–25 (2003)
22. Furche, T., Gottlob, G., Grasso, G., Guo, X., Orsi, G., Schallhart, C., Wang, C.: DIADeM: thousands of websites to a single database. *PVLDB* **7**(14), 1845–1856 (2014)
23. Furche, T., Gottlob, G., Libkin, L., Orsi, G., Paton, N.W.: Data wrangling for big data: challenges and opportunities. In: *EDBT*, pp. 473–478 (2016)
24. Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: RDF querying: language constructs and evaluation methods compared. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) *Reasoning Web 2006*. LNCS, vol. 4126, pp. 1–52. Springer, Heidelberg (2006). doi:[10.1007/11837787_1](https://doi.org/10.1007/11837787_1)
25. Gottlob, G., Kikot, S., Kontchakov, R., Podolskii, V.V., Schwentick, T., Zakharyashev, M.: The price of query rewriting in ontology-based data access. *Artif. Intell.* **213**, 42–59 (2014)
26. Gottlob, G., Koch, C.: Monadic queries over tree-structured data. In: *LICS*, pp. 189–202 (2002)
27. Gottlob, G., Koch, C.: Monadic datalog and the expressive power of languages for web information extraction. *J. ACM* **51**(1), 74–113 (2004)
28. Gottlob, G., Koch, C.: A formal comparison of visual web wrapper generators. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) *SOFSEM 2006*. LNCS, vol. 3831, pp. 30–48. Springer, Heidelberg (2006). doi:[10.1007/11611257_3](https://doi.org/10.1007/11611257_3)
29. Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Flesca, S.: The Lixto data extraction project: back and forth between theory and practice. In: *PODS*, pp. 1–12 (2004)
30. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. In: *VLDB*, pp. 95–106 (2002)

31. Gottlob, G., Koch, C., Pichler, R.: The complexity of XPath query evaluation. In: PODS, pp. 179–190 (2003)
32. Gottlob, G., Koch, C., Schulz, K.U.: Conjunctive queries over trees. In: PODS, pp. 189–200 (2004)
33. Gottlob, G., Manna, M., Pieris, A.: Polynomial rewritings for linear existential rules. In: IJCAI, pp. 2992–2998 (2015)
34. Gottlob, G., Orsi, G., Pieris, A.: Query rewriting and optimization for ontological databases. *ACM Trans. Database Syst.* **39**(3), 25:1–25:46 (2014)
35. Gottlob, G., Orsi, G., Pieris, A.: Consistency checking of re-engineered UML class diagrams via Datalog+/- . In: RuleML, pp. 35–53 (2015)
36. Gottlob, G., Pieris, A.: Beyond SPARQL under OWL 2 QL entailment regime: rules to the rescue. In: IJCAI, pp. 2999–3007 (2015)
37. Gottlob, G., Rudolph, S., Simkus, M.: Expressiveness of guarded existential rule languages. In: PODS, pp. 27–38 (2014)
38. Gottlob, G., Schwentick, T.: Rewriting ontological queries into small nonrecursive datalog programs. In: KR (2012)
39. Grau, B.C., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., Wang, Z.: Acyclicity conditions and their application to query answering in description logics. In: KR (2012)
40. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: Limits to Parallel Computation: P-Completeness Theory. Oxford University Press, Oxford (1995)
41. Johnson, D.S., Klug, A.C.: Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.* **28**(1), 167–189 (1984)
42. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. *J. ACM* **42**, 741–843 (1995)
43. Koch, C.: Efficient processing of expressive node-selecting queries on XML data in secondary storage: a tree automata-based approach. In: VLDB, pp. 249–260 (2003)
44. Laender, A.H.F., Ribeiro-Neto, B.A., da Silva, A.S.: Debye - data extraction by example. *Data Knowl. Eng.* **40**(2), 121–154 (2002)
45. Liu, L., Pu, C., Han, W.: XWRAP: An XML-enabled wrapper construction system for web information sources. In: ICDE, pp. 611–621 (2000)
46. Ludäscher, B., Himmeröder, R., Lausen, G., May, W., Schlepphorst, C.: Managing semistructured data with FLORID: a deductive object-oriented perspective. *Inf. Syst.* **23**(8), 589–613 (1998)
47. Lukasiewicz, T., Martinez, M.V., Pieris, A., Simari, G.I.: From classical to consistent query answering under existential rules. In: AAI, pp. 1546–1552 (2015)
48. Marnette, B.: Generalized schema-mappings: from termination to tractability. In: PODS, pp. 13–22 (2009)
49. Meuss, H., Schulz, K.U., Bry, F.: Towards aggregated answers for semistructured data. In: Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 346–360. Springer, Heidelberg (2001). doi:[10.1007/3-540-44503-X-22](https://doi.org/10.1007/3-540-44503-X-22)
50. Milani, M., Bertossi, L.: Ontology-based multidimensional contexts with applications to quality data specification and extraction. In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) RuleML 2015. LNCS, vol. 9202, pp. 277–293. Springer, Heidelberg (2015). doi:[10.1007/978-3-319-21542-6_18](https://doi.org/10.1007/978-3-319-21542-6_18)
51. Miller, R.J., Hernández, M.A., Haas, L.M., Yan, L., Ho, C.T.H., Fagin, R., Popa, L.: The clio project: managing heterogeneity. *SIGMOD Rec.* **30**(1), 78–83 (2001)
52. Minoux, M.: LTUR: a simplified linear-time unit resolution algorithm for horn formulae and computer implementation. *Inf. Process. Lett.* **29**(1), 1–12 (1988)
53. Neven, F., den Bussche, J.V.: Expressiveness of structured document query languages based on attribute grammars. *J. ACM* **49**(1), 56–100 (2002)

54. Neven, F., Schwentick, T.: Query automata over finite trees. *Theor. Comput. Sci.* **275**(1–2), 633–674 (2002)
55. Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., Ullman, J.: A query translation scheme for rapid implementation of wrappers. In: Ling, T.W., Mendelzon, A.O., Vieille, L. (eds.) *DOOD 1995*. LNCS, vol. 1013, pp. 161–186. Springer, Heidelberg (1995). doi:[10.1007/3-540-60608-4_40](https://doi.org/10.1007/3-540-60608-4_40)
56. Sahuguet, A., Azavant, F.: Building intelligent web applications using lightweight wrappers. *Data Knowl. Eng.* **36**(3), 283–316 (2001)
57. Seidl, H., Schwentick, T., Muscholl, A.: Numerical document queries. In: *PODS*, pp. 155–166 (2003)
58. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Syst. Theory* **2**(1), 57–81 (1968)
59. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) *Handbook of Formal Languages*, vol. 3, pp. 389–455. Springer, Heidelberg (1997). Chapter 7

Language and Automata Theory and Applications
11th International Conference, LATA 2017, Umeå,
Sweden, March 6-9, 2017, Proceedings

Drewes, F.; Martín-Vide, C.; Truthe, B. (Eds.)

2017, XXIII, 462 p. 59 illus., Softcover

ISBN: 978-3-319-53732-0