Logic-based Web Information Extraction*

Georg Gottlob and Christoph Koch

Database and Artificial Intelligence Group,
Technische Universität Wien,
A-1040 Vienna, Austria.
{gottlob, koch}@dbai.tuwien.ac.at

1 Introduction

The Web wrapping problem, i.e., the problem of extracting structured information from HTML documents, is one of great practical importance. The often-observed *information overload* that users of the Web experience witnesses the lack of intelligent and encompassing Web services that provide high-quality collected and value-added information. The Web wrapping problem has been addressed by a significant amount of research work. Previous work can be classified into two categories, depending on whether the HTML input is regarded as a sequential character string (e.g., [34, 27, 24, 30, 23]) or a pre-parsed document tree (for instance, [35, 25, 22, 29, 3, 2, 26]). The latter category of work thus assumes that systems may make use of an existing HTML parser as a front end.

Robust wrappers are *easier to program* using a wrapper programming language that models documents as pre-parsed document trees rather than as text strings. Writing a fully standards-compliant HTML parser is a substantial task, which should not have to be redone from scratch for each wrapper being created. The use of an existing parser allows the wrapper implementor to focus on the essentials of each wrapping task and to work on a higher, more user-friendly level.

Simplifications such as this one are important, since Web service designers often face the task of wrapping a large number of Web sites. In order to provide a useful Web service, the information from a significant number of source sites relevant to the domain of the service has to be integrated and made accessible in a uniform manner. Otherwise, a Web service may fail to attract the acceptance of the users it is intended for.

Moreover, Web page layouts may be subject to frequent change. This is often intentional – to discourage screen-scraping wrapper access and to force humans to personally visit the sites. In such a case wrapper programs may need to be adapted.

These are just two reasons for which wrapping tools need to assist humans to render the creation of wrappers a more manageable task. Two ways to approach this requirement have been proposed: the use of machine learning techniques to create wrappers automatically from annotated examples (e.g. [23, 30]) and the *visual specification* of wrappers. The first approach currently suffers from the need to provide machine learning algorithms with too many example instances – which have to be wrapped manually – and from negative theoretical results that put a bound on the expressive power of learnable wrappers.[1]

By the second candidate[2] for a substantial productivity leap, the *visual specification* of wrappers, we ideally mean the process of interactively defining a wrapper from one (or few) example document(s) using mainly "mouse clicks", supported by a strong and intuitive design metaphor. During this visual process, the wrapper program should be automatically generated and should not actually require the human designer to know the wrapper programming language. Visual wrapping is now a reality supported by several implemented systems (cf. XWrap [25], W4F [35], and Lixto [2]), however with varying thoroughness.

One may thus want to look for a wrapping language over HTML document trees that

(i) has a solid theoretical foundation,
(ii) provides a good trade-off between complexity and the number of practical wrappers that can be expressed,
(iii) is easy to use as a wrapper programming language, and
(iv) is suitable for being incorporated into visual tools, since ideally all constructs of a wrapping language can be realized through corresponding visual primitives.

* Database Principles Column. Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

[1] For example, it is known that even regular string languages cannot be learned from positive examples only [12].

[2] A promising direction of future work on Web wrapping may be to combine visual specification with machine learning techniques, in order to make the visual specification process more "intelligent" and at the same time guide the learning process to reduce the number of examples needed to learn a wrapper.

Clearly, languages which do not have the right expressive power and computational properties cannot be considered satisfactory, even if wrappers are easy to define. A few words on the "right expressiveness" of a wrapper programming language are in order here.

Throughout the literature, the scope of wrapping is a conceptually limited one (see e.g. [11, 20]). Information systems architectures that employ wrapping usually consist of at least two layers, a lower one that is restricted to extracting *relevant* data from data sources and making them available in a coherent representation using the data model supported by the higher layer, and a higher layer in which data transformation and integration tasks are performed which are necessary to fuse syntactically coherent data from distinct sources in a semantically coherent manner. With the term wrapping we refer to the lower, syntactic integration layer. The higher, semantic integration layer will not be further considered in this article. Therefore, a wrapper is assumed to extract relevant data from a possibly poorly structured source and to put it into the desired representation formalism by applying a number of transformational changes close to the minimum possible. A wrapping language that permits arbitrary data transformations may be considered overkill.

In this article, we provide a survey of the (logic-based) approch to visual wrapping that has been introduced in [2, 3, 16] and has been implemented in a commercial software product, the Lixto Visual Wrapper [26]. The core notion that this wrapping approach is based 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 document viewed as a tree, a new 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. (See Figure 1 for an example of such a transformation.) 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 our expressiveness yardstick for information extraction functions. MSO over trees is well-understood theory-wise [38, 7, 5, 8] (see also [39]) and quite expressive. In fact, it is considered by many as the language of choice for defining expressive

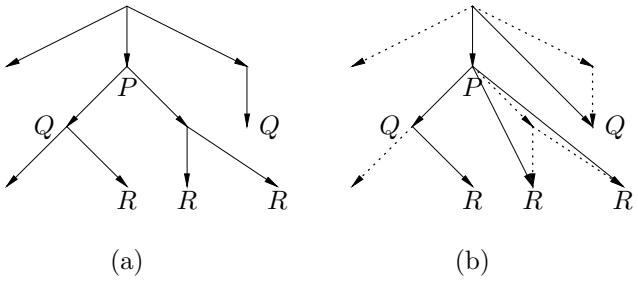

Figure 1: Tree annotated with predicates P, Q, and R defined by information extraction functions (a), and wrapping result (b). (Original node labels of the input tree are not shown.)

node-selecting queries on trees (see e.g. [33, 32, 16, 21]; [36] 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 [21]).

In this article, we discuss *monadic datalog* over trees, a simple form of the logic-based language datalog, as a wrapper programming language. As argued for in detail in [16], monadic datalog is the first and currently only language which satisfies desiderata (i) to (iv) raised above.

The article is structured as follows. We start with preliminaries regarding trees and MSO in Section 2 and introduce monadic datalog in Section 3. In Section 4, we discuss the complexity of monadic datalog over trees and its expressive power. Interestingly, monadic datalog is equivalent to MSO in its ability to express unary queries on trees. Monadic datalog can be evaluated in linear time both in the size of the data and the query, given that tree structures are appropriately represented. We also define a simple *normal form* for monadic datalog over trees, TMNF, to which any monadic datalog program over trees can be mapped in linear time. In Section 5, we discuss monadic datalog as a wrapper programming language. Finally, in Section 6, we look at the specification of wrappers based on monadic datalog without requiring the wrapper implementor to deal with or know datalog, by an entirely visual process that works on example Web documents. We conclude with Section 7.

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[3] nonempty set of symbols Σ, the alphabet. We consider only *unranked* finite trees, which cor-

[3]The finite alphabet choice is discussed in more detail below, in Remark 2.1.

respond 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, and "firstchild" and "nextsibling" are binary. 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 Figure 2). $\text{label}_a(n)$ 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, ...).

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 (see e.g. [42]). 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.

Remark 2.1 In the context of wrapping HTML documents, it is worthwhile to consider an *infinite* alphabet Σ, which allows to merge both HTML tags and attribute assignments into labels. This requires a generalized notion of relational structures $\langle \text{dom}, R_1, R_2, R_3, \dots\rangle$ consisting of a countable (but possibly *infinite*) set of relations, of which only a finite number is nonempty. All results in this article also hold for infinite alphabets in case the symbols of the alphabet (i.e., the node labels) are not part of the domain, labels of domain elements are expressed via predicates such as label_a only (rather than, say, a binary relation label $\subseteq \text{dom} \times \Sigma$), and for each predicate label_a we can also use its complement $\overline{\text{label}_a}$ (in the finite-alphabet case such a complement can be obtained by the union $\bigcup_{l\in(\Sigma-\{a\})} \text{label}_l$). Given these requirements, it is impossible to quantify over symbols of Σ and any query in finitary logical languages can only refer to a finite number of symbols of the alphabet Σ.

Another way to cope with composite tags and attribute values is to encode such values as lists of character symbols modeled as subtrees in our document tree. Whatever way is preferred, it should be clear that the assumption of a finite alphabet Σ made in this article is not a true limitation for representing real-world documents. □

3 Monadic Datalog

We define the function-free logic programming syntax and semantics of datalog in brief (cf. [1, 40] for detailed surveys of datalog).

A datalog program is a set of datalog rules. A datalog rule is of the form

$$h \leftarrow b_1,\ \dots,\ b_n.$$

where $h,\ b_1,\ \dots,\ b_n$ are called atoms, h is called the rule head, and $b_1,\ \dots,\ b_n$ (understood as a conjunction of atoms) is called the body. Each atom is of the form $p(x_1, \dots, x_m)$, where p is a predicate and $x_1, \dots, x_m$ are variables and constants (from a finite domain dom). Variable-free atoms, rules, or programs are called *ground*. Rules are required to be *safe*, i.e., all variables appearing in the head also have to appear in the body. Predicates that appear in the head of some rule of a program are called *intensional*, all others are called *extensional*. An *extension* is a set of ground atoms that are assumed to be true. We assume that for each extensional predicate, a (possibly empty) extension is given as input data. By *signature*, we denote the (finite) set of all extensional predicates (with fixed arities) available to the program. By default, we use the signature τ_{ur} for unranked trees.[4]

Let r be a datalog rule. By $\mathit{Vars}(r)$ we denote the set of variables occurring in r and by $\mathit{Body}(r)$ we denote the set of body atoms of r.

A *valuation* is a function $\phi : (\mathit{Vars}(r) \cup \text{dom}) \to \text{dom}$ which maps each variable to an element of dom and is the identity on dom. Given an atom $p(x_1, \dots, x_m)$, let $\phi(p(x_1, \dots, x_m)) := p(\phi(x_1), \dots, \phi(x_n))$.

We define the semantics of datalog as the fixpoint $\mathcal{T}_\mathcal{P}^\omega$ of the immediate consequence operator $\mathcal{T}_\mathcal{P}$.

Definition 3.1 Let $\mathcal{P}$ be a datalog program and $\mathcal{B}$ the (finite) set of all ground atoms over the domain dom and a given signature. The *immediate consequence operator* $\mathcal{T}_\mathcal{P} : 2^\mathcal{B} \to 2^\mathcal{B}$ is defined as

$$\mathcal{T}_\mathcal{P}(X) := X \cup \{\phi(h) \mid \text{ex. a rule } h \leftarrow b_1, \dots, b_n. \text{ in } \mathcal{P} \text{ and a valuation } \phi \text{ on the rule s.t. } \phi(b_1), \dots, \phi(b_n) \in X\}.$$

Let $\mathcal{T}_\mathcal{P}^0 := X$ and $\mathcal{T}_\mathcal{P}^{i+1} := \mathcal{T}_\mathcal{P}(\mathcal{T}_\mathcal{P}^i)$ for each $i \geq 0$, where X is the database given as a set of ground atoms. The *fixpoint* $\mathcal{T}_\mathcal{P}^n = \mathcal{T}_\mathcal{P}^{n+1}$ of the sequence $\mathcal{T}_\mathcal{P}^0, \mathcal{T}_\mathcal{P}^1, \mathcal{T}_\mathcal{P}^2, \dots$ is denoted by $\mathcal{T}_\mathcal{P}^\omega$. □

[4] 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.

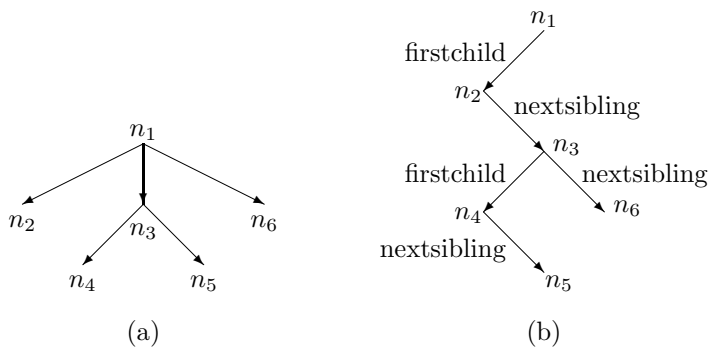

Figure 2: (a) An unranked tree and (b) its representation using the binary relations "firstchild" (↙) and "nextsibling" (↘).

It is clear that $\mathcal{T}_\mathcal{P}$ eventually reaches a fixpoint because it ranges over a finite universe dom given with the database and the sequence $\mathcal{T}^0_\mathcal{P}, \mathcal{T}^1_\mathcal{P}, \mathcal{T}^2_\mathcal{P}, \ldots$ is strictly (because $\mathcal{T}_\mathcal{P}$ is deterministic) monotonically increasing until the fixpoint is reached. The semantics of $\mathcal{P}$ on X is defined as $\mathcal{T}^\omega_\mathcal{P}$.

Monadic datalog 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*. In the remainder of this article, when talking about a monadic datalog query, we will always refer to a unary query specified as a monadic datalog program with a distinguished query predicate.

Example 3.2 We construct a monadic datalog program over τ_{ur} which, given an unranked tree (representing an HTML parse tree), computes all those 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 $\langle i\rangle \ldots \langle /i\rangle$ and is thus labeled "i"). The program uses a single intentional predicate, *Italic*, which computes this unary query. It consists of the rules

$$\textit{Italic}(x) \leftarrow \text{label}_i(x). \quad (1)$$
$$\textit{Italic}(x) \leftarrow \textit{Italic}(x_0),\ \text{firstchild}(x_0, x). \quad (2)$$
$$\textit{Italic}(x) \leftarrow \textit{Italic}(x_0),\ \text{nextsibling}(x_0, x). \quad (3)$$

The computation of fixpoint $\mathcal{T}^\omega_\mathcal{P}$ for this program proceeds as follows. $\mathcal{T}^0_\mathcal{P}$ consists of the nodes labeled "i", $\mathcal{T}^1_\mathcal{P}$ of the first children and neighboring right siblings of those nodes in $\mathcal{T}^0_\mathcal{P}$, if they exist, and $\mathcal{T}^{k+1}_\mathcal{P}$ contains a node $v \in$ dom if there is a node v_0 in $\mathcal{T}^k_\mathcal{P}$ that is $k+1$ layers above v in the binary tree defined by the firstchild and nextsibling relations (see Figure 2 for an example of such a tree). If d is the depth of the HTML tree, this fixpoint computation terminates after at most $d+2$ steps ($\mathcal{T}^{d+1}_\mathcal{P} = \mathcal{T}^{d+2}_\mathcal{P}$). □

Monadic second-order logic (MSO) over trees is a second-order logical language consisting of (1) individual variables ranging, (2) variables ranging over sets of nodes, (3) parentheses, (4) boolean connectives $\vee$ and $\neg$, (5) quantifiers $\forall$ and $\exists$ over both node and set variables, (6) the relation symbols of the tree structure in consideration, $=$ (equality of node variables), and, as syntactic sugaring, possibly (7) the boolean operations $\wedge$, $\rightarrow$, and $\leftrightarrow$ and the relation symbols $=$ and $\subseteq$ between sets. 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 $\{x \in \text{dom} \mid t \vDash \varphi(x)\}$.

The following holds for arbitrary finite structures:

Proposition 3.3 (Folklore) *Each monadic datalog query is MSO-definable.*

Throughout the article, our main measure of query evaluation cost is *combined complexity*, i.e. where both the database and the query (or program) are considered variable.

Proposition 3.4 ([16]) *Monadic datalog (over arbitrary finite structures) is NP-complete w.r.t. combined complexity.*

This is is stark contrast to full datalog, which is known to be EXPTIME-complete w.r.t. combined complexity (implicit in [41, 19]; cf. [6]). Monadic datalog has some other nice properties which have been studied; in particular, the containment and boundedness properties – which are both considered relevant to query optimization – for monadic datalog are decidable [4], while they are undecidable for full datalog [37, 10].

4 Monadic Datalog over Trees

By restricting our structures to trees, monadic datalog acquires a number of additional nice properties. First, the query evaluation complexity is linear in the size of the data and of the program:

Theorem 4.1 ([16]) *Over τ_{ur}, monadic datalog has $O(|\mathcal{P}| * |dom|)$ combined complexity (where $|\mathcal{P}|$ is the size of the program and $|dom|$ 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 $O(|\mathcal{P}| * |\text{dom}|)$. Ground programs can be evaluated in linear time [28].

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

Theorem 4.2 ([16]) *Each unary MSO-definable query over τ_{ur} is definable in monadic datalog over τ_{ur}.*

(The other direction follows from Proposition 3.3.) Interestingly, 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 the query evaluation problem is known to be PSPACE-complete) for considerably expanded program sizes.

Theorem 4.2 also asserts that monadic datalog programs can define "universal" properties over trees, such as that a certain fact holds everywhere or nowhere in a tree. This may seem somewhat unintuitive because monadic datalog does not feature negation. Still, it follows from the fact that the relations defining the tree in τ_{ur} allow us to traverse the tree (starting from its "ends" such as the leaves or the root) using a recursive program and to compute universal properties along the way.

Example 4.3 Consider the problem of selecting all those nodes from an HTML tree which are italic (using the program from Example 3.2) and do *not* contain an HTML "table" in their subtrees. We can define a monadic datalog program for this (with query predicate Q) as follows.

$$
\begin{aligned}
\mathit{NoTableBelow}(x) &\leftarrow \text{leaf}(x).\\
\mathit{NoTableBelow}(x) &\leftarrow \text{firstchild}(x,y), \mathit{NoTable}(y).\\
\mathit{NoTableRight}(x) &\leftarrow \text{lastsibling}(x).\\
\mathit{NoTableRight}(x) &\leftarrow \text{nextsibling}(x,y), \mathit{NoTable}(y).\\
\mathit{NoTable}(x) &\leftarrow \overline{\text{label}_{table}}(x),\\
&\qquad \mathit{NoTableBelow}(x),\\
&\qquad \mathit{NoTableRight}(x).\\
Q(x) &\leftarrow \mathit{Italic}(x), \overline{\text{label}_{table}}(x),\\
&\qquad \mathit{NoTableBelow}(x).
\end{aligned}
$$

Here we may either assume that there is a predicate $\overline{\text{label}_{table}}$ true for those nodes not labeled "table" (this predicate then needs to be added to τ_{ur}) or we can define $\overline{\text{label}_{table}}$ in monadic datalog by the rules $\{\overline{\text{label}_{table}}(x) \leftarrow \text{label}_l(x). \mid l \in \Sigma, l \neq \text{"table"}\}$. (See Remark 2.1 for a related discussion.)

Note that both $\overline{\text{label}_{table}}$ and *NoTableBelow* are true for a node iff it does not contain a "table" in its subtree, while *NoTable* is true for a node iff the same holds (i.e., it does not contain a "table" in its subtree) in the binary-tree model of Figure 2 (b). □

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 4.4 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 four forms:

$$
\begin{aligned}
&(1) \quad p(x) \leftarrow p_0(x).\\
&(2) \quad p(x) \leftarrow p_0(x_0),\ R(x_0, x).\\
&(3) \quad p(x) \leftarrow p_0(x_0),\ R(x, x_0).\\
&(4) \quad p(x) \leftarrow p_0(x),\ p_1(x).
\end{aligned}
$$

where the unary predicates p_0 and p_1 are either intensional or from τ_{ur} and R is a binary predicate from τ_{ur}. □

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

Theorem 4.5 ([16]) *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 $O(|\mathcal{P}|)$.*

5 Monadic Datalog as a Wrapper Programming Language

We now make a bridging step from the main topic of this article so far, monadic datalog over trees, to extracting information from parse trees of Web documents. In our framework, a wrapper is defined via a set of unary queries, "information extraction functions", that select tree nodes. A monadic datalog program can compute a *set* of such queries 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 are auxiliary.)

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.

In the previous section, we have shown 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,

- The existence of the normal form TMNF demonstrates that rules in monadic datalog never have to be long or intricate. [5]
- 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.
- Wrappers defined in monadic datalog only need to specify queries, rather than the full source trees on which they run. This is very important to practical wrapping, because this way changes in parts of documents not immediately relevant to the objects to be extracted do not break the wrapper.

Thus, monadic datalog over trees as a framework for Web information extraction satisfies the first three of our desiderata from the introduction (efficient evaluation, appropriate expressiveness, and suitability as a practical wrapper programming language). Only the fourth desideratum remains to be addressed, the visual specification of wrappers.

6 Visual Wrapper Specification

In this section, we introduce the core visual specification procedure used in the Lixto wrapper generator [2, 3]. Lixto uses a wrapping language called Elog. Elog programs can be completely visually specified and are actually very similar to monadic datalog; the core of the language (called Elog$^-$ in [16] and studied there in detail) is monadic datalog as discussed before with a few minor syntactic restrictions which do not lower its expressiveness. Thus, the property that unary queries can be entirely visually specified is also inherited by MSO.

[5] The simple structure of TMNF rules is actually essential to visual wrapper specification. Compared to MSO with its first- and second-order quantifiers, few syntactical features remain for which viable visual design metaphors have to be developed.

As discussed in the introduction, by visual wrapper specification, we refer to the process of interactively defining a wrapper from few example documents using ideally mainly "mouse clicks".

The visual wrapping process in systems such as Lixto heavily relies on one main operation performed by users: By marking a region of an example Web document displayed on screen using an input device such as a mouse, the node in the document tree best matching the selected region can be robustly determined. By selecting a reference region followed by a second region inside the former, it is possible to define a fixed path π in an example document.

Let $\text{subelem}_{a_1 \ldots a_n}(x, y)$, where $a_1 \ldots a_n \in \Sigma^*$ is a word from the labeling alphabet interpreted as a directed path in the tree, be true if, for each $1 \leq i \leq n$, the i-th node in the path from node x to y excluding x is labeled a_i. Note that "subelem" can be expressed by a fixed conjunction of child and label atoms, so we will consider it as a shortcut rather than a new built-in predicate. (Theorem 4.5 provides a method to eliminate child atoms to obtain programs strictly over τ_{ur}.) For example, $\text{subelem}_{a.b}(x, y)$ is a shortcut for $\text{child}(x, z)$, $\text{label}_a(z)$, $\text{child}(z, y)$, $\text{label}_b(y)$, where z is a new variable.

To provide a useful metaphor for the building blocks of wrappers, Lixto calls the visual counterparts of monadic intensional predicates *patterns* and those of rules *filters*.

Given an example document representative for a family of documents to be wrapped, a user may be guided in the visual specification of a rule as follows.

- First, a destination pattern p is selected from those existing or newly created and a parent pattern p_0 is selected from among the patterns defined so far. Initially, the only pattern available is the "root" pattern.

  The "root" pattern corresponds to the extensional predicate root of τ_{ur} and is the only exception to the correspondence of patterns and intensional predicates.
- The system can then display the document and highlight those regions in it which correspond to nodes in its parse tree that are classified p_0 using the wrapper program specified so far.
- A new rule is defined by selecting – by a few mouse clicks over the example document – a subregion of one of those highlighted. The system can automatically decide which path π relative to the highlighted region best describes the region selected by the user.
- The rule $p(x) \leftarrow p_0(x_0), \text{subelem}_\pi(x_0, x)$. obtained in this way can be refined by generalizing the path π (dropping "label" atoms) or adding conditions. These tasks can be carried out visually as well (see [2]).

To obtain the expressiveness of MSO, little power has to be added via conditions; one only has to be able to refer to root, leaf, and leftmost sibling nodes of the tree and to patterns via unary atoms; moreover, one has to be able to specify "nextsibling" atoms [16]. TMNF rules such as $p(x) \leftarrow p_0(x_0), \mathit{firstchild}(x_0, x)$ can then be specified by selecting a child node (say) labeled a of an instance of pattern p_0 in an example document, selecting p as destination pattern (this produces the rule $p(x) \leftarrow p_0(x_0), \mathit{subelem}_a(x_0, x)$), generalizing from the specified path a (the result is $p(x) \leftarrow p_0(x_0), \mathit{subelem}\_(x_0, x)$), and adding the condition that x has no left sibling (= is a first sibling). The Elog$^-$ fragment of Elog, discussed in detail in [16], has precisely the expressive power of MSO.

Very few example documents are needed for defining a wrapper program: It is only required that for each rule to be specified, there exists a document in which an instance of the parent pattern can be recognized and an instance of the destination pattern relates to it in the desired manner.

The process outlined is used in the Lixto system and is described in more detail in [3, 2], where many examples and screenshots are dedicated to the visual specification process. Note that the full Elog language discussed there supports Web crawling, stratified (datalog) negation, and navigation via certain forms of regular paths (optionally with so-called *distance tolerances*). Presenting these features in detail is beyond the scope of this article. Many features only serve as shortcuts to simplify the wrapper specification process and the improve productivity, but some actually render the full Elog language of [2] strictly more expressive than MSO [16].

7 Conclusions

We have presented a significant new application of logic (programming) to information systems. The database programming language *datalog*, which has received considerable attention from the database theory community over many years (see e.g. [1]) but has ultimately failed to attract a large following in database practice, would deserve to experience a "rebirth" in the context of trees and the Web. Indeed, for datalog as a framework for selecting nodes from trees, the situation is substantially different from the general case of full datalog on arbitrary databases. Monadic datalog over trees has very low evaluation complexity, programs have a simple normal form, so rules never have to be long or intricate, and various automata-theoretic, language-theoretic, and logical techniques exist (cf. [39, 31, 4, 16, 18]) for evaluating programs or optimizing them which are not available for full datalog.

We have omitted the presentation of a number of results showing that Elog is actually currently the only language of its kind with the expressive power of MSO; in particular, it is shown in [13, 15] that the only other visual, tree-based wrapping system in the literature for which a formal specification exists, W4F [35], is based on a strictly weaker wrapping language.

As a final remark, monadic datalog also has applications in querying XML and checking the conformance of XML documents to DTD's and regular tree languages. For example, Core XPath [17], the logical core fragment of the popular XPath language, can be mapped efficiently to monadic datalog [14, 9] and thus inherits its very favorable worst-case evaluation complexity bounds.

Acknowledgments

This work was supported by the Austrian Science Fund (FWF) under project No. Z29-N04 and the GAMES Network of Excellence of the European Union.

References

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

[2] R. Baumgartner, S. Flesca, and G. Gottlob. "Declarative Information Extraction, Web Crawling, and Recursive Wrapping with Lixto". In *Proc. LPNMR'01*, Vienna, Austria, 2001.

[3] R. Baumgartner, S. Flesca, and G. Gottlob. "Visual Web Information Extraction with Lixto". In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, 2001.

[4] S. Cosmadakis, H. Gaifman, P. Kanellakis, and M. Vardi. "Decidable Optimization Problems for Database Logic Programs". In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 477–490, Chicago, Illinois, USA, 1988. ACM Press, New York, NY, USA.

[5] B. Courcelle. "Graph Rewriting: An Algebraic and Logic Approach". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 5, pages 193–242. Elsevier Science Publishers B.V., 1990.

[6] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. "Complexity and Expressive Power of Logic Programming". *ACM Computing Surveys*, **33**(3):374–425, Sept. 2001.

[7] J. Doner. "Tree Acceptors and some of their Applications". *Journal of Computer and System Sciences*, **4**:406–451, 1970.

[8] J. Flum, M. Frick, and M. Grohe. "Query Evaluation via Tree-Decompositions". *Journal of the ACM*, **49**(6):716–752, 2002.

[9] M. Frick, M. Grohe, and C. Koch. "Query Evaluation on Compressed Trees". In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Canada, June 2003.

[10] H. Gaifman, H. G. Mairson, Y. Sagiv, and M. Y. Vardi. "Undecidable Optimization Problems for Database Logic Programs". In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 106–115, 1987.

[11] M. R. Genesereth and S. P. Ketchpel. "Software Agents". *Commun. ACM*, **37**(7):48–53, 1994.

[12] E. Gold. "Language Identification in the Limit". *Inform. Control*, 10:447–474, 1967.

[13] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages". In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 17–28, Madison, Wisconsin, 2002.

[14] G. Gottlob and C. Koch. "Monadic Queries over Tree-Structured Data". In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 189–202, Copenhagen, Denmark, July 2002.

[15] G. Gottlob and C. Koch. "A Formal Comparison of Visual Web Wrapper Generators". Technical Report cs.DB/0310012, CoRR, 2003. http://arxiv.org/abs/cs.DB/0310012.

[16] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages". *Journal of the ACM*, **51**(1):74–113, 2004.

[17] G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[18] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'04)*, 2004.

[19] N. Immerman. "Relational Queries Computable in Polynomial Time". *Information and Control*, **68**(1–3):86–104, 1986.

[20] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer-Verlag, 2000.

[21] C. Koch. "Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 249–260, 2003.

[22] R. Kosala, H. Blockeel, M. Bruynooghe, and J. V. den Bussche. "Information Extraction from Web Documents based on Local Unranked Tree Automaton Inference". In *Proc. IJCAI*, 2003.

[23] N. Kushmerick, D. Weld, and R. Doorenbos. "Wrapper Induction for Information Extraction". In *Proc. IJCAI*, 1997.

[24] A. H. F. Laender, B. Ribeiro-Neto, and A. S. da Silva. "DEByE – Data Extraction By Example". *Data and Knowledge Engineering*, **40**(2):121–154, Feb. 2002.

[25] L. Liu, C. Pu, and W. Han. "XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources". In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE)*, pages 611–621, San Diego, USA, 2000.

[26] http://www.lixto.com.

[27] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. "Managing Semistructured Data with Florid: A Deductive Object-oriented Perspective". *Information Systems*, **23**(8):1–25, 1998.

[28] M. Minoux. "LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation". *Information Processing Letters*, **29**(1):1–12, 1988.

[29] Mostrare project. http://www.grappa.univ-lille3.fr/mostrare/.

[30] I. Muslea, S. Minton, and C. Knoblock. "A Hierarchical Approach to Wrapper Induction". In *Proc. 3rd Intern. Conf. on Autonomous Agents*, 1999.

[31] F. Neven. "Automata Theory for XML Researchers". *SIGMOD Record*, **31**(3), Sept. 2002.

[32] F. Neven and T. Schwentick. "Query Automata on Finite Trees". *Theoretical Computer Science*, **275**:633–674, 2002.

[33] F. Neven and J. van den Bussche. "Expressiveness of Structured Document Query Languages Based on Attribute Grammars". *Journal of the ACM*, **49**(1):56–100, Jan. 2002.

[34] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. "A Query Translation Scheme for Rapid Implementation of Wrappers". In *Proc. 4th International Conference on Deductive and Object-oriented Databases (DOOD'95)*, pages 161–186, Singapore, 1995. Springer.

[35] A. Sahuguet and F. Azavant. "Building Intelligent Web Applications Using Lightweight Wrappers". *Data and Knowledge Engineering*, **36**(3):283–316, 2001.

[36] H. Seidl, T. Schwentick, and A. Muscholl. "Numerical Document Queries". In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'03)*, pages 155–166, San Diego, California, 2003.

[37] O. Shmueli. "Decidability and Expressiveness Aspects of Logic Queries". In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*, pages 237–249, 1987.

[38] J. Thatcher and J. Wright. "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-order Logic". *Mathematical Systems Theory*, **2**(1):57–81, 1968.

[39] W. Thomas. "Languages, Automata, and Logic". In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7, pages 389–455. Springer Verlag, 1997.

[40] J. D. Ullman. *Principles of Database & Knowledge-Base Systems Vol. 1.* Computer Science Press, 1988.

[41] M. Y. Vardi. "The Complexity of Relational Query Languages". In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 137–146, San Francisco, CA USA, May 1982.

[42] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.