Monadic Datalog and the Expressive Power of Languages for Web Information Extraction*

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

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

ABSTRACT

Research on information extraction from Web pages (wrapping) has seen much activity in recent times (particularly systems implementations), but little work has been done on formally studying the expressiveness of the formalisms proposed or on the theoretical foundations of wrapping.

In this paper, we first study monadic datalog as a wrapping language (over ranked or unranked tree structures). Using previous work by Neven and Schwentick, we show that this simple language is equivalent to full monadic second order logic (MSO) in its ability to specify wrappers. We believe that MSO has the right expressiveness required for Web information extraction and thus propose MSO as a yardstick for evaluating and comparing wrappers.

Using the above result, we study the kernel fragment Elog$^-$ of the Elog wrapping language used in the Lixto system (a visual wrapper generator). The striking fact here is that Elog$^-$ exactly captures MSO, yet is easier to use. Indeed, programs in this language can be entirely visually specified. We also formally compare Elog to other wrapping languages proposed in the literature.

1. INTRODUCTION

The Web wrapping problem, i.e., the problem of extracting structured information from HTML documents, is one of high practical importance and has spurred a great amount of work, including theoretical research (e.g., [5]) as well as systems. Previous work can be classified into two categories, depending on whether the HTML input is regarded as a sequential character string (e.g., TSIMMIS [27], Editor [5], FLORID [21], and DEByE [18]) or a pre-parsed document tree (for instance, W4F [28], XWrap [20], and Lixto[1] [8, 7]). The latter category of work thus assumes that systems may make use of an existing HTML parser as a front end.

Taking a practical perspective, robust wrappers are easier to build over pre-parsed documents, as the handling of the intricacies of HTML is left to the parser and does not need to be programmed from scratch into each wrapper being created. This allows the wrapper implementor to focus on the essentials of each wrapping task. Even from the standpoint of theory, many practical problems are presumably simpler to solve over the parse trees of documents rather than over the documents themselves (that is, as strings). [2]

It is understood in the literature that the scope of wrapping is a conceptually limited one. 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.

One may thus want to look for a wrapping language over document trees that (i) has a solid and well understood 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. This paper exhibits and studies such languages.

The core notion that we base our wrapping approach on is the one of an *information extraction function*. An information extraction function takes a labeled unranked tree (representing a Web document) and returns a subset of its nodes or, viewed differently, subtrees rooted by these nodes. In the context of the present paper, a wrapper is a program which implements one or several such functions. 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 exactly the essence of wrapping.

We propose unary queries in monadic second-order logic (MSO) as an expressiveness yardstick for information ex-

*This work was supported by the Austrian Science Fund (FWF) under project No. Z29-INF.

[1]See `http://www.lixto.com`.

[2]In fact, it is known that a word language is context-free iff it is the yield of a regular tree language [16], where the yield of a tree is the sequence of labels of its leaf nodes extracted depth-first from left to right.

traction functions. MSO over trees is well-understood theory-wise (see e.g. [30, 25]) and quite expressive. Moreover, unary MSO queries can be evaluated in *linear time* w.r.t. the sizes of the input trees [15, 11]. Unfortunately, MSO does not satisfy requirements (iii) and (iv): It is neither easy to use as a wrapping language nor does it lend itself to visual specification.

The main contributions of the paper are the following.

- We study monadic datalog (over labeled trees) and show that it is equivalent to monadic second-order logic (MSO) in its ability to express (unary) queries for tree nodes. This is true both for ranked and unranked trees. Proofs are partially based on results for query automata (by Neven and Schwentick [25]).

  Monadic datalog is a very simple programming language and much better suited as a wrapping language than MSO. Consequently, it satisfies the first three of our requirements.

- We then go on to present a simple Web wrapping language equivalent to MSO, which we call Elog^-. It is obtained by slightly restricting the syntax of monadic datalog and is a simplified version of the core wrapping language of the Lixto system, Elog. Rules of this language can be completely visually specified, without requiring the wrapper implementor to deal with Elog^- programs directly or to know datalog. We also give a brief overview of this visual specification process. Thus, this language satisfies all of our four desiderata for tree-based wrapping languages.

  In Elog^-, a wrapper is a function that assigns unary predicates to document tree nodes. Based on these predicate assignments and the structure of the input 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 trees but using the new labels and omitting nodes that have not been relabeled.

- The capability to produce a hierarchically structured result is essential to tree wrapping. We define the language Elog_2^- in order to be able to make the creation of complex nested structures explicit and to improve flexibility. Elog_2^- is obtained by enhancing Elog^- with binary predicates in a restricted form, which allow to represent hierarchical dependencies between selected nodes in the fixpoint computation of an Elog^- program. Note that such a binary dependency relation can be quadratic in the size of the input tree. Notwithstanding, we are able to show that the expressiveness of Elog_2^- and Elog^- with respect to unary queries is the same, and the edge relation defined by the binary predicates is again MSO-definable. Elog_2^- is an actual fragment of the wrapping language Elog used internally in the Lixto system [7], a commercial visual wrapper generator.

- Finally, we take a closer look at two other tree-based approaches to wrapping HTML documents. The first is the language of regular path queries (e.g., [1, 2]) with nesting. Regular path queries are considered essential to Web query languages [1], and by extending the language of regular path queries by capabilities for producing nested output (and for restricting queries by additional conditions), one obtains a useful wrapping language. We show that this formalism is strictly less expressive than Elog_2^-.

- The second formalism is HEL [28], the wrapping language of the commercially available W4F framework, which is the only tree-based wrapping formalism besides Elog of which a formal specification has been published. Again, we are able to show that HEL is strictly less expressive than Elog_2^-.

This is – to the best of our knowledge – the first work providing a theoretical study of advanced tree-based wrapping tools and languages. In summary, we present a thorough theoretical analysis of expressiveness aspects of tree-based information extraction based on the expressiveness of MSO as an intuitively justifiable yardstick for languages attacking this problem.

The paper is structured as follows. We start with preliminaries regarding tree languages and MSO in Section 2 and introduce monadic datalog (over trees) in Section 3. Section 4 presents our equivalence results between MSO and monadic datalog. In Section 5 we proceed to the wrapping problem and show that Elog^- captures MSO (Section 5.3). The modification from Elog^- to Elog_2^- is studied in Section 5.4. Finally, we compare our family of wrapping languages to other languages in Section 6.

2. PRELIMINARIES

Throughout this paper, only *finite* trees will be considered. 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 alphabet. We consider both ranked and unranked trees. Ranked trees have a ranked alphabet, i.e., each symbol in Σ has some fixed arity or rank $k \leq K$ (where K is some constant integer). Thus, in ranked trees, each node with a label a of rank k has exactly k children. Nodes with labels of rank 0 are leaves. We may partition Σ into sets $\Sigma_0, \ldots, \Sigma_K$ of symbols of equal rank. Each ranked tree can be considered as a relational structure[3]

$$t_r = \langle \text{dom},\ \text{root},\ \text{leaf},\ (\text{child}_k)_{k \leq K},\ (\text{label}_a)_{a \in \Sigma} \rangle,$$

and each unranked tree as a structure

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

where dom is the set of nodes in the tree and the relations are defined according to their intuitive meanings. For instance, child_k is binary and denotes the k-th direct child relation and $\text{nextsibling}(n_1, n_2)$ is true iff n_1 and n_2 are the i-th and $(i+1)$-th children of a common parent node, respectively. $label_a(n)$ is true iff n is labeled a in the tree.

Monadic second-order logic (MSO) over trees is a second-order logical language consisting of (1) variables (with lowercase names $x, y, \ldots$) ranging over nodes, (2) set variables (written using upper-case names $P, Q, \ldots$) 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 model-theoretic 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.

[3]Of course, equally well-suited structures are obtained by adding predicates definable over the structure in some formalism or removing others that are redundant (w.r.t. definability).

Π_1-MSO refers to MSO sentences of the form

$$\forall P_1, \ldots, P_n : \psi(P_1, \ldots, P_n)$$

where the P_i are set variables and ψ is a first-order formula. It is easy to define a natural total ordering $\prec$ of dom in MSO (obtained by depth-first left-to-right traversal of the tree, where, say, parents precede children), which is also called the *document order* in the context of wrapping HTML documents (see e.g. [32]). A unary MSO *query* is a unary predicate definable in MSO (i.e., by a formula with one free first-order variable). A tree language $\mathcal{L}$ is definable in MSO iff there is a closed MSO formula φ over tree structures t such that $\mathcal{L} = \{t \mid t \vDash_{MSO} \varphi\}$.

The regular tree languages are precisely those recognizable by a number of finite automata, such as nondeterministic descending (or *top-down*) tree automata (NDTA), both nondeterministic (NATA) and deterministic (DATA) ascending (or *bottom-up*) tree automata [9], and deterministic (2DTA) as well as nondeterministic (2NTA) two-way tree automata [9, 25]. This is true for ranked as well as for unranked alphabets. We provide definitions of deterministic bottom-up automata in both the ranked and the unranked case.

DEFINITION 2.1. A deterministic bottom-up *ranked* tree automaton is a tuple $\mathcal{A} = \langle Q, \Sigma, \delta_0, \ldots, \delta_n, F\rangle$ where Q is a finite set of states, $\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_n$ is a ranked alphabet, δ_k, for each $0 \leq k \leq n$, is the (partial) transition function $(Q^k \times \Sigma_k \to Q)$, and $F \subseteq Q$ is a set of final states.

The semantics of $\mathcal{A}$ (by the notion of a *run*) on a tree t, $\delta^*(t)$, is defined inductively as follows: If t consists of only a leaf node labeled a, then $\delta^*(t) = \delta_0(a)$ (if $\delta_0(a)$ is defined). If t is labeled a and has the children $t_1, \ldots, t_m$, then $\delta^*(t) = \delta_m(\delta^*(t_1), \ldots, \delta^*(t_m), a)$ (if defined). A run is called successful for some tree t (that is, the tree is *accepted*) iff $\delta^*(t) \in F$. The set of Σ-trees accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$. □

DEFINITION 2.2. A deterministic bottom-up *unranked* tree automaton is a tuple $\mathcal{A} = \langle Q, \Sigma, \delta, F\rangle$ where Q is a finite set of states, Σ is an (unranked) alphabet, δ is a (partial) transition function $Q \times \Sigma \to 2^{Q^*}$ s.t. $\delta(q, a)$ is a regular language over states in Q, and $F \subseteq Q$ is a set of final states.

The semantics of $\mathcal{A}$ on a tree t, $\delta^*(t)$, is defined inductively as follows: If t consists of only a leaf node labeled a, and $\delta(q, a) = \epsilon$ for some q, then $\delta^*(t) = q$. If t is labeled a and has the children $t_1, \ldots, t_m$ with $\delta^*(t_1) = q_1, \ldots, \delta^*(t_m) = q_m$, and there is some q such that $q_1 \cdots q_m \in \mathcal{L}(\delta(q, a))$, then $\delta^*(t) = q$. As before, a run is called successful for some tree t (that is, the tree is *accepted*) iff $\delta^*(t) \in F$. The set of Σ-trees accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$. □

DEFINITION 2.3. A ranked (resp., unranked) tree language is regular iff is is accepted by some deterministic bottom-up ranked (resp., unranked) tree automaton. □

The following is a classical result for ranked trees, which has been shown in [25] to hold for unranked trees as well (see also [9]).

PROPOSITION 2.4. *A tree language is regular iff it is definable in MSO.*

REMARK 2.5. In the context of wrapping HTML documents, it is worthwile 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 $\langle \text{dom}, R_1, R_2, R_3, \ldots\rangle$ consisting of a domain dom and a countable (but possibly *infinite*) set of relations, of which only a finite number is nonempty. Even though all results cited or shown in this paper (such as Proposition 2.4) were proven for finite alphabets, it is trivial to see that they also hold for infinite alphabets in case the symbols of the alphabet (i.e., the node labels) are not part of the domain, and labels of domain elements are expressed via predicates (such as the label_a) only. Given these requirements, it is impossible to quantify over symbols of Σ and any query in whatever language can only refer to a finite number of symbols of Σ. (See the related discussion in the preliminaries of [24].) In this paper, we avoid this problem by assuming a finite set Σ. Attribute assignments can be encoded, for instance, as lists of character symbols modeled as subtrees in our document tree. □

3. MONADIC DATALOG OVER TREES

In the following sections, we will use a syntactically restricted fragment of standard datalog [31, 10]. We adhere to the usual minimal model (= least fixpoint) semantics, which can be defined using, say, the immediate consequence operator $\mathcal{T}_\mathcal{P}$. By *signature* we denote a finite set of built-in predicates.

DEFINITION 3.1. Let signature τ denote a tree structure. A monadic datalog program is a set of datalog rules in which all extensional predicates are from τ and all intensional predicates are unary. □

By default, we will always use the signature

$$\tau_r = \langle \text{root}, \text{leaf}, (\text{child}_k)_{k \leq K}, (\text{label}_a)_{a \in \Sigma}\rangle$$

for ranked trees and

$$\tau_u = \langle \text{root},\ \text{leaf},\ (\text{label}_a)_{a \in \Sigma},\ \text{firstchild},\ \text{lastchild},\ \text{nextsibling}\rangle$$

for unranked trees. We use t_r and t_u to denote the extensions of τ_r and τ_u, respectively.

In order to be able to compare MSO with monadic datalog, we make a few assumptions. By (unary) *query*, for monadic datalog as for MSO, we denote a function that assigns a predicate to some nodes of a tree t (or, in other words, selects a subset of $\text{dom}(t)$).

The following result is part of the database folklore:

PROPOSITION 3.2. *Over arbitrary finite structures, each monadic datalog query is Π_1-MSO-definable.*

We say that a monadic datalog program with some dedicated intensional predicate (say, "accept") accepts a tree t iff $\text{accept}(r) \in T^\omega_\mathcal{P}(t)$ (i.e., the root node is in the inferred extension of "accept"). A monadic datalog program $\mathcal{P}$ recognizes the tree language $L = \{t \mid \mathcal{P} \text{ accepts } t\}$.

The following result is similar to the folklore result that monadic fixpoint logic over trees expresses MSO (w.r.t. tree language acceptance), and can be shown by a straightforward simulation of bottom-up tree automata (a proof will be given in the long version of this paper).

THEOREM 3.3. *A tree language is regular iff it is definable in monadic datalog.*

We conclude this section by providing a new complexity result which characterizes the hardness of executing monadic datalog programs.

THEOREM 3.4. *For ranked and unranked trees, monadic datalog has $\mathcal{O}(|\mathcal{P}| * |dom(t)|)$ combined complexity (where $|\mathcal{P}|$ is the size of the input program and $|dom(t)|$ is the size of the tree), linear-time data complexity, and its program complexity is complete for linear time.*

Proof Sketch We assume the signature τ_r for ranked trees and τ_u for unranked trees. Observe that all of the binary predicates in τ_r and τ_u (that is, $(\text{child}_k)_{k\leq K}$, firstchild, lastchild, and nextsibling) have both a functional dependency from their first column to the second and one from the second column to the first.

Given a monadic datalog program $\mathcal{P}$, let us apply the following transformation. For each rule, we split off connected parts that do not contain the head variable, create a rule with a propositional head predicate for them, and add the propositional predicates to the original rule as replacements of the removed parts. For instance, $p(X) \leftarrow p_1(X),\ p_2(Y).$ is rewritten into $p(X) \leftarrow p_1(X),\ b.$ and $b \leftarrow p_2(Y)$. It is easy to see that this transformation is linear in the size of the program.

Now, for each rule in the transformed program $\mathcal{P}'$, each variable functionally determines all others, as the variables are connected via binary predicates that are one-to-one only. (That is, in a graph of functional dependencies, each variable is reachable from each other variable.) Consequently, there is only a linear number of relevant ground instances (in the size of the data), which can be computed in linear time. It is known from [13, 22] (see also [12]) that the fixpoint of a ground program can be computed in time linear in the size of the program. Thus, the composition of these steps requires $\mathcal{O}(|\mathcal{P}| * |\text{dom}(t)|)$ time. This is an upper bound for the combined complexity of the problem, and we have both linear time data and program complexities.

Finally, the problem is P-hard w.r.t. combined resp. program complexity, as it generalizes the propositional Horn-SAT problem, which is P-complete (e.g., [26]). □

The data complexity results of Theorem 3.4 also follow from the fact that the data complexity of MSO *queries* over finite structures of bounded tree-width is in linear time [15] and the facts that ranked labeled trees (where $|\Sigma|$ is the size of the labeling alphabet) are of tree-width $\leq 1 + |\Sigma|$ and unranked labeled trees are of tree-width $\leq 2+|\Sigma|$. However, the other results of Theorem 3.4 are new.

4. MONADIC DATALOG QUERIES

In this section, we show that MSO and monadic datalog are equivalent in their power to define unary queries. One direction was already stated in Proposition 3.2. For the proof of the other, we need some additional machinery.

DEFINITION 4.1. [25] A ranked query automaton (QA^r) – that is, a two-way deterministic ranked tree automaton with a selection function – is a tuple

$$\mathcal{A} = \langle Q, \Sigma, F, s, \delta_\uparrow, \delta_\downarrow, \delta_{root}, \delta_{leaf}, \lambda \rangle$$

where Q is a finite set of states, $F \subseteq Q$ is the (nonempty) set of final states, $s \in Q$ is the start state, Σ is a ranked alphabet, the δ's are transition functions, and λ is the so-called *selection function*. Let there be a partition of $Q \times \Sigma$ into two disjoint sets U and D. (1) $\delta_\uparrow : U^{\leq K} \to Q$ is the transition function for *up transitions*. (2) $\delta_\downarrow : D \times \{1, \ldots, K\} \to Q^*$ is the transition function for *down transitions*. For each $i \leq K$, $\delta_\downarrow(q, a, i)$ is a string of states of length i. (3) $\delta_{root} : U \to Q$ is the transition function for *root transitions*. (4) $\delta_{leaf} : D \to Q$ is the transition function for *leaf transitions*.

Let t be a ranked tree. A cut is a subset of $\text{dom}(t)$ which contains exactly one node of each path from the root to a leaf. A configuration of $\mathcal{A}$ on t is a mapping $c : C \to Q$ from a cut C of t to the set of states Q of $\mathcal{A}$.

The automaton $\mathcal{A}$ makes a transition between two configurations $c_1 : C_1 \to Q$ and $c_2 : C_2 \to Q$, denoted by $c_1 \to c_2$, if it makes an up, down, root, or leaf transition:

1. $\mathcal{A}$ makes an up transition from c_1 to c_2 if there is a node n s.t. (a) the children of n, say, $n_1, \ldots, n_m$, are in C_1, (b) $C_2 = (C_1 - \{n_1, \ldots, n_m\}) \cup \{n\}$, (c) $c_2(n) = \delta_\uparrow(\langle c_1(n_1), label(n_1)\rangle,\ \ldots,\ \langle c_1(n_m), label(n_m)\rangle)$, and (d) c_2 is identical to c_1 on $C_1 \cap C_2$.

2. $\mathcal{A}$ makes a down transition from c_1 to c_2 if there is a node n s.t. (a) $n \in C_1$, (b) $C_2 = (C_1 - \{n\}) \cup \{n_1, \ldots, n_m\}$, where $\{n_1, \ldots, n_m\}$ is the set of children of n, (c) $c_2(n_1) \cdots c_2(n_m) = \delta_\downarrow(c_1(n), label(n), arity(n))$, and (d) c_2 is identical to c_1 on $C_1 \cap C_2$.

3. $\mathcal{A}$ makes a root transition from c_1 to c_2 if (a) $C_1 = C_2 = \{root(t)\}$, where $root(t)$ denotes the root node of t, and (b) $c_2(root(t)) = \delta_{root}(c_1(root(t)), label(root(t)))$.

4. $\mathcal{A}$ makes a leaf transition from c_1 to c_2 if there is a (leaf) node n s.t. (a) $n \in C_1$, (b) $C_2 = C_1$, (c) $c_2(n) = \delta_{leaf}(c_1(n), label(n))$, and (d) c_2 is identical to c_1 on $C_1 - \{n\}$.

The start configuration $c : C \to Q$ has $C = \{n\}$ (where n is the root node of t) and $c(n) = s$. Any configuration with $c(root(t)) \in F$ is an accepting configuration. (That is, a 2DTA^r starts at the root and terminates there.) A run is a sequence of configurations $c_1, \ldots, c_m$ s.t. $c_1 \to \cdots \to c_m$ and c_1 is the start configuration. A run is accepting if c_m is an accepting configuration and there does not exist a c_{m+1} s.t. $c_m \to c_{m+1}$.

Since often a number of transitions can be made in parallel, there are usually many different *sequences* of transitions that are possible. However, because of the disjointness of U and D, given a node n with some label and a ("current") state q, at most one (kind of) transition involving n is possible at any point in time, and for all nodes, the sequence of states in which they are visited is the same in all these runs. Thus we can consider this type of automaton as *deterministic* and refer to *the* run of $\mathcal{A}$ rather than a run of $\mathcal{A}$. Even though an automaton of the kind specified can run forever on an input tree, we can restrict ourselves to automata that always terminate. (This is a decidable property [25].)

The selection function $\lambda : Q \times \Sigma \to \{\bot, 1\}$ is defined as follows. A query automaton $\mathcal{A}$ selects a node n in configuration $c : C \to Q$ if $n \in C$ and $\lambda(c(n), label(n)) = 1$. $\mathcal{A}$ selects n if the run $c_1, \ldots, c_m$ is accepting and if there is an $1 \leq i \leq m$ s.t. n is selected by $\mathcal{A}$ in c_i. □

Thus, a query automaton computes the set of nodes selected at any time during the run, not just in the terminating configuration (which, by our definition, only contains the root node in its cut).

PROPOSITION 4.2. **[25]** *A unary query over ranked trees is MSO-definable iff there is a ranked query automaton which computes it.*

For use in the following lemma, let

$$Q_{n,X} = \{q \in Q \mid \langle q, label(n)\rangle \in X \text{ and } \exists i : q = c_i(n)\}$$

where X is either U or D.

LEMMA 4.3. *Let $\mathcal{A}$ be a ranked query automaton, t be a (ranked) tree, and $n, n' \in dom(t)$ a pair of nodes where n' is the k-th child of n.*

We can define a (partial) function $f_{n,k} : Q_{n,D} \to Q_{n',U}$ s.t. for all $i < j$ with $n \in C_i$, $n \notin (C_{i+1} \cup \cdots \cup C_j)$, $n' \in C_j$, and $c_i(n) \in Q_{n,D}$, $c_j(n') \in Q_{n',U}$,

$$c_j(n') = f_{n,k}(c_i(n)).$$

That is, the state of a node n uniquely determines the states each of its child nodes can have directly before an up transition back to n is made. We can define a function $f_{n,k}$ by which the state (before an up transition) of the k-th child of n only depends on the state of n but not on the configuration to which this state assignment belongs.

Proof Sketch This lemma can easily be shown by a simple induction over trees (bottom-up). However, such an induction is not even necessary: The fact that $c_j(n')$ functionally depends on $c_i(n)$ (for $i < j$, $n \in C_i$, $n \notin (C_{i+1} \cup \cdots \cup C_j)$, $n' \in C_j$) is a direct consequence of determinism as required in Definition 4.1.

Furthermore, let $q = c_i(n)$ and $q' = c_j(n')$. Then, by definition (in particular of down transitions, which overwrite child states without reading them first), all the configurations $c_{i+1}, \ldots, c_j$ (i.e., in which transitions occur below n in the tree) only depend on q and on the tree. Thus, whenever a configuration contains state q for n (and, after all, $\langle q, label(n)\rangle \in D$), the subsequent configurations again lead to the state assignment q' for n' ($\langle q', label(n')\rangle \in U$). □

Now we can state our result for ranked queries.

THEOREM 4.4. *For each unary MSO-definable query there exists a monadic datalog query (over τ_r) s.t. these two queries return the same result for all* ranked *trees.*

Proof By virtue of Proposition 3.2 and Proposition 4.2, all we need to show is an encoding that maps each ranked query automaton $\mathcal{A}$ to a monadic datalog program $\mathcal{P}$ such that $\mathcal{A}$ and $\mathcal{P}$ compute the same query.

In our encoding, predicate names are pairs (of state names) in $(Q \cup \{r\}) \times Q$. Intuitively, $\langle q_1, q_2\rangle(n)$ means that n is in state q_2 and its parent is in state q_1. r is a dummy state which we will assign to the imaginary parent of the root node. As we will see, it is only to correctly handle up transitions that parent states need to be managed via predicates.

The encoding $\mathcal{P}$ of $\mathcal{A}$ is the following set of rules. For all $q, q', q_1, \ldots, q_m \in Q$ and for all $a, a_1, \ldots, a_m \in \Sigma$,

1. (*Start state*) we add the single rule

   $$\langle r, s\rangle(n) \leftarrow \text{root}(n).$$

   where s is the start state of $\mathcal{A}$;

2. (*Up transition*)

   if $\delta_\uparrow(\langle q_1, a_1\rangle, \ldots, \langle q_m, a_m\rangle) = q'$, we add the rules

   $$\begin{aligned}\langle x, q'\rangle(n) \leftarrow\ & \langle x, q\rangle(n),\\ & \text{child}_1(n, n_1),\ \ldots,\ \text{child}_m(n, n_m),\\ & \langle q, q_1\rangle(n_1),\ \ldots,\ \langle q, q_m\rangle(n_m),\\ & \text{label}_{a_1}(n_1),\ \ldots,\ \text{label}_{a_m}(n_m).\end{aligned}$$

   for all $x \in (Q \cup \{r\})$;

3. (*Down transition*) if $\delta_\downarrow(q, a, m) = q_1 \cdots q_m$, we add

   $$\langle q, q_i\rangle(n_i) \leftarrow \langle x, q\rangle(n),\ \text{child}_i(n, n_i),\ \text{label}_a(n).$$

   for all $1 \le i \le m$, $x \in (Q \cup \{r\})$;

4. (*Root transition*) if $\delta_{root}(q, a) = q'$, we add

   $$\langle r, q'\rangle(n) \leftarrow \langle r, q\rangle(n),\ \text{label}_a(n),\ \text{root}(n).$$

5. (*Leaf transition*) finally, if $\delta_{leaf}(q, a) = q'$, we add

   $$\langle x, q'\rangle(n) \leftarrow \langle x, q\rangle(n),\ \text{label}_a(n),\ \text{leaf}(n).$$

   for all $x \in (Q \cup \{r\})$;

Note that all datalog variables n, n_i in our encoding range over nodes in $\text{dom}(t)$.

The encoding aims at computing exactly *all* the state assignments made during the run of $\mathcal{A}$, formalized as

$$H = \{\langle q, n\rangle \mid n \in \text{dom}(t),\ c_i(n) = q \text{ for some } i\}$$

(the "history" of $\mathcal{A}$), while we do not try to model configurations. We abbreviate $\{\langle q, n\rangle \mid \langle q_0, q\rangle(n) \in X\}$ (for a set $X \subseteq \mathcal{T}_\mathcal{P}^\omega$) as $\pi(X)$.

It is easy to see from $\mathcal{P}$ and Definition 4.1 that the state assignments in our fixpoint $\mathcal{T}_\mathcal{P}^\omega$ are certain to subsume those in H, i.e., $\pi(\mathcal{T}_\mathcal{P}^\omega) \supseteq H$. The other direction (i.e., soundness of $\mathcal{T}_\mathcal{P}^\omega$) can be shown by induction over the computation of $\mathcal{T}_\mathcal{P}^\omega$.

- Initially, we obtain $\mathcal{T}_\mathcal{P}^1 = \{\langle r, s\rangle(root(t))\}$ by applying the start state rule. (Clearly, $\pi(\mathcal{T}_\mathcal{P}^1) \subseteq H$.)
- Let X (with $\pi(X) \subseteq H$) be the set of facts obtained so far in the fixpoint computation. Rules in $\mathcal{P}$ which correspond to root, leaf, and down transitions have only a single state assignment premise in their bodies. If the premise is true w.r.t. X (and thus H), the state assignment $\langle q_0, q\rangle(n)$ inferred by such a rule must again be in some configuration of the run of $\mathcal{A}$ and thus be sound (that is, $\langle q, n\rangle \in H$).
- Let X (with $\pi(X) \subseteq H$) be the set of facts obtained so far in the fixpoint computation, and let an up transition rule r of $\mathcal{P}$ infer $\langle x, q'\rangle(n)$ from

  $$\langle q, q_1\rangle(n_1), \ldots, \langle q, q_m\rangle(n_m) \in X$$

  (where nodes $n_1, \ldots, n_m$ are the children of node n and $\langle q_1, label(n_1)\rangle, \ldots, \langle q_m, label(n_m)\rangle \in U$).

  By Lemma 4.3, we know that for each $q \in Q$ and each node n_k, there is at most one $q_k \in Q$ s.t. $\langle q, q_k\rangle(n) \in X$ and $\langle q_k, label(n_k)\rangle \in U$.

  Since for each $\langle q, q_k\rangle(n_k)$, q_k only depends on q and on t, the $\langle q_k, n_k\rangle$ are all computed independently by $\mathcal{A}$ after some down transition has been made from n to the n_k, and state assignments of this form remain in the current

configuration of $\mathcal{A}$ until a common up transition occurs, $\langle q_1, n_1\rangle, \ldots, \langle q_m, n_m\rangle$ can be assumed to be in the same configuration of $\mathcal{A}$ at some point. Consequently, the new state assignment inferred by r is again sound. (That is, $\langle q', n\rangle \in H$.)

Thus, $\pi(\mathcal{T}^{\omega}_{\mathcal{P}}) = H$. The definition of the selection function for a query automaton nicely coincides with the monotonic semantics of monadic datalog. We obtain the program $\mathcal{P}'$ from $\mathcal{P}$ by adding, for each $q \in Q$, $a \in \Sigma$, with $\lambda(q, a) = 1$, a rule

$$query(n) \leftarrow \langle x, q\rangle(n),\ \mathrm{label}_a(n).$$

for each $x \in (Q \cup \{r\})$. Then,

$$\begin{aligned} query(\mathcal{A}) &= \{n \mid \langle q, n\rangle \in H \text{ and } \lambda(q, label(n)) = 1\} \\ &\equiv \{n \mid query(n) \in \mathcal{T}^{\omega}_{\mathcal{P}'}\}. \end{aligned}$$

□

Next we characterize queries in monadic datalog over unranked trees. Analogously to query automata for ranked trees, we define the class of *strong query automata* over unranked trees as a tool for proving the above theorem. Let two-way deterministic finite (string) automata (2DFA) be defined in the normal way (e.g., [17]).

DEFINITION 4.5. [25] A *strong unranked query automaton* (SQA^u) is a tuple

$$\mathcal{A} = \langle Q, \Sigma, F, s, \delta_{\uparrow}, \delta_{\downarrow}, \delta_{\_}, \delta_{root}, \delta_{leaf}, \lambda\rangle,$$

where Q, F, s, U, D, δ_{leaf}, δ_{root} and λ are as in Definition 4.1. Let U_{up} and U_{stay} be two disjoint regular subsets of U^*. The transition function for up transitions is now of the form $\delta_{\uparrow} : U_{up} \to Q$, and the transition function for down transitions is of the form $\delta_{\downarrow} : D \times N \to Q^*$ (where N is the set of natural numbers). For each $\langle q, a\rangle \in D$, $L_{\downarrow}(q, a) := \{\delta_{\downarrow}(q, a, i) \mid i \in N\}$ is regular; for each $j \in N$, $\delta_{\downarrow}(q, a, j)$ must be a string of length j; and for each $q \in Q$, the language $L_{\uparrow}(q) := \{w \in U^* \mid \delta_{\uparrow}(w) = q\}$ must be regular. To assure determinism, we require that $L_{\uparrow}(q) \cap L_{\downarrow}(q') = \emptyset$ for all $q \neq q'$.

$\delta_{\_} : U_{stay} \to Q^*$ is the transition function for so-called *stay transitions*. We require this function to be computed by a 2DFA $\mathcal{B} = \langle S, \Sigma_{\mathcal{B}} = Q \times \Sigma, s_0, \delta_{\mathcal{B}}, F_{\mathcal{B}}, L, R\rangle$ over the string $\langle c_1(n_1), label(n_1)\rangle, \ldots, \langle c_1(n_m), label(n_m)\rangle$ with a selection function $\lambda_{\mathcal{B}} : S \times \Sigma_{\mathcal{B}} \to Q \cup \{\bot\}$ that – anytime during its run – maps nodes to states s.t., upon the termination of $\mathcal{B}$, each node has been assigned exactly one state in Q. $\mathcal{A}$ makes a stay transition at a node n (whose children are $n_1, \ldots, n_m$) from a configuration $c_1 : C_1 \to Q$ to $c_2 : C_2 \to Q$ if

(a) $n_1, \ldots, n_m \in C_1$,
(b) $C_2 = C_1$,
(c) $\delta_{\_}(\langle c_1(n_1), label(n_1)\rangle, \ldots, \langle c_1(n_m), label(n_m)\rangle) = c_2(n_1) \cdots c_2(n_m)$, and
(d) c_2 is identical to c_1 on $C_1 - \{n_1, \ldots, n_m\}$.

We require that at each node, at most one stay transition is made (this is a decidable property [25] for a given SQA^u).

The definitions of configurations, leaf, root, up and down transitions, run, and accepting run carry over from Definition 4.1. The query computed by $\mathcal{A}$ and the tree language defined by $\mathcal{A}$ are defined analogously to Definition 4.1. □

PROPOSITION 4.6. **[25]** *A unary query over unranked trees is MSO-definable iff there is an SQA^u that computes it.*

We can now show the following theorem by a proof similar to the one of Theorem 4.4.

THEOREM 4.7. *For each unary MSO-definable query there is a monadic datalog query (over τ_u) s.t. for all* unranked *trees, these two queries return the same result.*

Proof (Sketch) The proof works analogously to the one for the case of ranked queries[4], with the following changes to the encoding of the automaton (which now is an SQA^u) in monadic datalog.

1. Down transitions: It is clear from Definition 4.5 that the regular language $L_{\downarrow}(q, a)$ must be of density 1 (i.e. for each i, $|L_{\downarrow}(q, a) \cap \Sigma^i| \leq 1$).

   As a special case of an interesting result for regular languages of polynomial density [29, 33], we have that if a regular language $\mathcal{L}$ over Σ has constant density, $\mathcal{L}$ can be denoted by a finite union of regular expressions of the form uv^*w (where the u, v, w are words over Σ).

   Let $\bigcup_i u_i v_i^* w_i$ be such a union expression for $L_{\downarrow}(q, a)$ (that is, the u_i, v_i, and w_i are words over an alphabet consisting of the states of the query automaton).

   Intuitively, we proceed as follows, for each i in parallel. First, we use temporary predicates to match the word u_i with the first $|u_i|$ child nodes of n with respect to the nextsibling relation, starting from the left (rules 1 and 2 of the encoding provided below). Next, we match w_i with the $|w_i|$ rightmost children of n (rules 3 and 4). We furthermore mark all nodes before those with $\neg w_i$ (rules 5 and 6; This is not needed if we use monadic datalog with stratified negation). Next we match the $(|u_i|+1)$-th node up to the rightmost node marked $\neg w_i$ with v_i^* (rules 7 to 9; let y be the length of this sequence of nodes). If this matching is exact ($y = |v_i| = 0$ or $y \bmod |v_i| = 0$), we create state assignments out of the temporary facts computed so far for i (rules 10 to 13). Since $\mathcal{L}(\bigcup_i u_i v_i^* w_i)$ has density one, such assignments are made for at most one i.

   The monadic datalog encoding for down transitions is as follows.

$$\begin{aligned} \mathrm{tmp}_{q,u_i,1}(n_1) &\leftarrow \langle x, q\rangle(n), \\ &\quad \mathrm{firstchild}(n, n_1),\ \mathrm{label}_a(n). \\ \mathrm{tmp}_{q,u_i,k+1}(n_{k+1}) &\leftarrow \mathrm{tmp}_{q,u_i,k}(n_k), \\ &\quad \mathrm{nextsibling}(n_k, n_{k+1}). \end{aligned}$$

$$\begin{aligned} \mathrm{tmp}_{q,w_i,|w_i|}(n') &\leftarrow \langle x, q\rangle(n),\ \mathrm{lastchild}(n, n'). \\ \mathrm{tmp}_{q,w_i,l-1}(n') &\leftarrow \mathrm{tmp}_{q,w_i,l}(n), \\ &\quad \mathrm{nextsibling}(n', n). \end{aligned}$$

$$\begin{aligned} \mathrm{tmp}_{q,\neg w_i}(n') &\leftarrow \mathrm{tmp}_{q,w_i,|1|}(n),\ \mathrm{nextsibling}(n', n). \\ \mathrm{tmp}_{q,\neg w_i}(n') &\leftarrow \mathrm{tmp}_{q,\neg w_i}(n),\ \mathrm{nextsibling}(n', n). \end{aligned}$$

[4]Clearly, an analogous result to Lemma 4.3 can be stated for unranked trees.

$$\begin{aligned}
\mathrm{tmp}_{q,v_i,1}(n') &\leftarrow \mathrm{tmp}_{q,u_i,|u|}(n),\\
&\quad \mathrm{nextsibling}(n,n'),\ \mathrm{tmp}_{q,\neg w_i}(n').\\
\mathrm{tmp}_{q,v_i,m+1}(n') &\leftarrow \mathrm{tmp}_{q,v_i,m}(n),\\
&\quad \mathrm{nextsibling}(n,n'),\ \mathrm{tmp}_{q,\neg w_i}(n').\\
\mathrm{tmp}_{q,v_i,1}(n') &\leftarrow \mathrm{tmp}_{q,v_i,|v_i|}(n),\\
&\quad \mathrm{nextsibling}(n,n'),\ \mathrm{tmp}_{q,\neg w_i}(n').\\
\mathrm{succ}_{q,i}(n') &\leftarrow \mathrm{tmp}_{q,u_i,|u_i|}(n'),\\
&\quad \mathrm{nextsibling}(n',n),\ \mathrm{tmp}_{q,w_i,1}(n).\\
\mathrm{succ}_{q,i}(n') &\leftarrow \mathrm{tmp}_{q,v_i,|v|}(n'),\\
&\quad \mathrm{nextsibling}(n',n),\ \mathrm{tmp}_{q,w_i,1}(n).\\
\mathrm{succ}_{q,i}(n') &\leftarrow \mathrm{succ}_{q,i}(n),\ \mathrm{nextsibling}(n,n').\\
\mathrm{succ}_{q,i}(n') &\leftarrow \mathrm{succ}_{q,i}(n),\ \mathrm{nextsibling}(n',n).\\
\langle q,\alpha_{i,j}\rangle(n) &\leftarrow \mathrm{succ}_{q,i}(n),\ \mathrm{tmp}_{q,\alpha_i,j}(n).
\end{aligned}$$

for all $x \in Q \cup \{r\}$, i, j, $1 \leq k < |u_i|$, $1 \leq l < |w_i|$, $1 \leq m < |v_i|$, and where α_i is either u_i, v_i, or w_i and $\alpha_{i,j}$ is the j-th symbol in α_i.

2. Up transitions: Let $\mathcal{B} = \langle Q, s_0, \delta, F\rangle$ be a deterministic finite automaton for $\mathcal{L}_\uparrow(q_0, a_0)$ (that is, its alphabet is U). For each $x \in (Q \cup \{r\}), y \in Q$, we create rules as follows.

   (a) For each $\delta(s_0, \langle q, a\rangle) = s'$,

   $$\mathrm{tmp}_{y,s'}(n) \leftarrow \langle y,q\rangle(n),\ \mathrm{firstchild}(n_0,n),\ \mathrm{label}_a(n).$$

   (b) For each $\delta(s, \langle q, a\rangle) = s'$,

   $$\mathrm{tmp}_{y,s'}(n') \leftarrow \mathrm{tmp}_{y,s}(n),\ \mathrm{nextsibling}(n,n'),\ \langle y,q\rangle(n'),\ \mathrm{label}_a(n').$$

   (c) For each $s \in F$,

   $$\langle x,q_0\rangle(n_0) \leftarrow \mathrm{tmp}_{y,s}(n),\ \langle x,y\rangle(n_0),\ \mathrm{lastchild}(n_0,n),\ \mathrm{label}_{a_0}(n_0).$$

   That is, we traverse siblings from a first to a last (from left to right) to check whether their state-and-label pairs constitute a word of language $\mathcal{L}_\uparrow(q_0, a_0)$ before we make our up transition.

3. Stay transitions: Since each tree node may only be involved in a stay transition once, we may consider the simulation of the 2DFA taken out of the context of our proof, by itself.

   The encoding of a 2DFA with a selection function λ is straightforward. Each transition only depends on a single state assignment. As discussed for the case of query automata for ranked *trees* earlier, this condition entails that the computation of the union of all the configurations run through by the 2DFA as a fixpoint of our monadic datalog program and the application of a selection function λ to this set is sound. □

5. TREE WRAPPING

In this section, we make a bridging step from the topic of the previous sections – monadic datalog over trees – to extracting information from parse trees of HTML documents.

DEFINITION 5.1. A *document* is a node-labeled unranked tree in which the root node has a special label "document" not used by any other node. □

We assume that HTML attributes are simply represented as nodes of the parse tree and can be accessed through the child relation of the document tree. [5]

DEFINITION 5.2. An information extraction function is a function that maps each unranked labeled tree t to a subset $S \subseteq \mathrm{dom}(t)$ of its nodes. □

Clearly, each unary predicate of a monadic datalog program can be considered to define one information extraction function. Now consider the following method of wrapping a Web document using information extraction functions: [6]

REMARK 5.3. Given a tree t and a number of information extraction functions $f_1, \ldots, f_m$, there is a natural way of extracting a wrapped tree t' from t using $f_1, \ldots, f_m$. Let $A_n = \{\text{`}f_i\text{'} \mid n \in f_i(t),\ 1 \leq i \leq m\}$, for each node $n \in \mathrm{dom}(t)$, and let compute_label be a function that computes a label from a set of predicate assignments A_n of a node n. Tree t' is obtained as follows:

- $n \in \mathrm{dom}(t)$ is a node of t' iff $A_n \neq \emptyset$ or n is the root of t (then, it is also the root of t'). Let the label of $n \in \mathrm{dom}(t')$ for tree t' be compute_label(A_n).
- Let $n_0, n \in \mathrm{dom}(t')$. n is a child of n_0 iff (1) n_0 is an ancestor of n in t and (2) there is no node $n' \in \mathrm{dom}(t')$ s.t. n_0 is an ancestor of n' in t and n' is an ancestor of n in t.
- The ordering of siblings in t' is coherent with the document order in t. □

5.1 Elog⁻

In this section, we introduce Elog⁻, a simplified fragment of the wrapping language Elog presented in [8, 7]. Subsequently, we refer to monadic intensional predicates as *pattern predicates*.

DEFINITION 5.4. The language Elog⁻ is a fragment of monadic datalog over the signature

$$\tau_{Elog^-} = \langle (\mathrm{subelem}_\pi)_{\pi\in\Pi},\ (\mathrm{contains}_\pi)_{\pi\in\Pi},\ (\mathrm{before}_\pi)_{\pi\in\Pi},\ (\mathrm{after}_\pi)_{\pi\in\Pi},\ \mathrm{firstson},\ \mathrm{lastson}\rangle,$$

where Π is the set of star-free (and disjunction-free) regular paths over Σ and rules are restricted to the form

$$p(X) \leftarrow p_0(X_0),\ \mathrm{subelem}_{path}(X_0,X),\ C,\ R.$$

s.t. p is a pattern predicate, p_0 – the so-called "parent pattern" – is either a pattern predicate or "root", R ("pattern references") is a set of atoms over pattern predicates, and C is a set of "condition atoms" (using the predicates "contains", "before", "after", "firstson", and "lastson").

The predicate $\mathrm{subelem}_{path}$ is defined inductively[7] in terms of τ_u and the predicate "child" (which is easily definable in

[5] See also Remark 2.5.

[6] However, note that this method is not central to our work and may be replaced by a different one.

[7] Note that the definition of subelem is *nonrecursive* and for each path π, $\mathrm{subelem}_\pi$ is defined through a fixed conjunction of child and label atoms.

MSO over τ_u) as

$$\begin{aligned}
\text{subelem}_\epsilon(X,Y) &:= X = Y.\\
\text{subelem}_{_.path}(X,Y) &:= \text{child}(X,Z),\\
&\quad\ \text{subelem}_{path}(Z,Y).\\
\text{subelem}_{a.path}(X,Y) &:= \text{child}(X,Z),\ \text{label}_a(Z),\\
&\quad\ \text{subelem}_{path}(Z,Y).
\end{aligned}$$

where '_' is a wild card matching *any* symbol, and the remaining (condition) predicates are defined as

$$\begin{aligned}
\text{contains}_{path}(X,Y) &:= \text{subelem}_{path}(X,Y).\\
\text{before}_{path}(X_0,X,Y) &:= \text{subelem}_{path}(X_0,Y),\\
&\quad\ \text{nextsibling}(X,Y).\\
\text{after}_{path}(X_0,X,Y) &:= \text{subelem}_{path}(X_0,Y),\\
&\quad\ \text{nextsibling}(Y,X).\\
\text{firstson}(X_0,X) &:= \text{firstchild}(X_0,X).\\
\text{lastson}(X_0,X) &:= \text{lastchild}(X_0,X).
\end{aligned}$$

ϵ-paths must not be used in condition atoms. □

We may write rules of the form

$$p(X) \leftarrow p_0(X_0),\ \text{subelem}_\epsilon(X_0,X),\ C,\ R.$$

as

$$p(X) \leftarrow p_0(X),\ C,\ R.$$

and call such rules *specialization rules*.

5.2 Visual Wrapper Specification

A main strength of Elog$^-$ (and Elog) is that programs can be completely visually specified (see [8]). The above mentioned "patterns" are a useful metaphor for the building blocks of wrappers. Given an example document to be wrapped, a user may be guided in the graphical specification of an Elog$^-$ rule

$$p(X) \leftarrow p_0(S),\ \text{subelem}_\pi(S,X),\ C,\ R.$$

as follows[8].

- First, a destination pattern p is named (which may be new) and a parent pattern p_0 is selected from among the patterns defined so far. Initially, the only pattern available is the "root" pattern, which matches an entire document.
- 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 basic rule $p(X) \leftarrow p_0(S),\ \text{subelem}_\pi(S,X).$ can then be refined by generalizing the path or adding conditions or references of other patterns. Conditions that involve a path (such as "contains", "before" and "after") are defined similarly to the process just described. It is easy to imagine an appropriate graphical user interface for the other manipulations.

[8]The process outlined is used in the Lixto system and is described in more detail in [8].

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.

5.3 Elog$^-$ Captures MSO

As stated next, *Elog*$^-$ – some surrogate for a completely visual wrapper specification process – has *exactly* the wrapping power of MSO (and equally, monadic datalog) over unranked trees.

THEOREM 5.5. *A wrapper (over documents) is definable in monadic datalog over τ_u iff it is definable in Elog$^-$.*

Proof Sketch A full proof of Theorem 5.5 is quite easy to obtain and will appear in the long version of the paper; here, we give a rough sketch. The fact that each query in Elog$^-$ can be expressed in monadic datalog over τ_u follows directly from the definition of Elog$^-$ and its consequence that each query in Elog$^-$ can be expressed in MSO. For the other direction, one can basically observe in the proof of Theorem 4.7 that only a very limited fragment of monadic datalog is required to simulate (strong) query automata. Programs of that form can be decomposed into a small number of rule templates; for instance, one such template is

$$p_2(X_2) \leftarrow p_1(X_1),\ \text{nextsibling}(X_1,X_2).$$

for some predicates p_1 and p_2. Each of the rules of the simplified program can now easily be simulated in Elog$^-$. For example, the above template rewrites into

$$\begin{aligned}
p_2(X_2) \leftarrow\ & \text{dom}(X_0),\ \text{subelem}_{_}(X_0,X_2),\\
& \text{after}_{_}(X_0,X_2,X_1),\ p_1(X_1).
\end{aligned}$$

□

Note at this point that the full Elog language of [7] is strictly more expressive than MSO. For example, Elog supports so-called distance tolerances in before and after predicates. Let Elog$^-_\Delta$ be the new language obtained from Elog$^-$ by extending its "before" predicate by a distance tolerance, which is a pair of percentage values s.t. whenever S refers to a node with k children, $\text{before}_{a\in\Sigma,x\%-y\%}(S,X,Y)$ requires that among the children of S, X is at least $\frac{k*100}{x}$ and at most $\frac{k*100}{y}$ before Y.

THEOREM 5.6. *The Elog$^-_\Delta$ language is strictly more expressive than unary MSO queries over unranked trees.*

Proof The Elog$^-_\Delta$ program $\mathcal{P}$

a_0(X) ← parent(X_0), subelem$_a$(X_0, X), notafter$_a$(X_0, X).
b_0(X) ← parent(X_0), subelem$_b$(X_0, X), notafter$_b$(X_0, X).
a^nb^n(X) ← parent(X),
contains$_a$(X, Y), a_0(Y),
before$_{b,50\%-50\%}$(X, Y, Z), b_0(Z),
notbefore$_a$(X, Z).

over $\Sigma = \{a, b\}$ classifies a node as a^nb^n if and only if its list of children is of the same form. However, the tree language $\{t \mid a^nb^n(\text{root}(t)) \in \mathcal{T}^\omega_\mathcal{P}\}$ is not regular. □

5.4 Binary Pattern Predicates

In this section, we step out of our framework of information extraction functions to address the *limited* form of binary pattern predicates that the Elog language supports. We enhance Elog$^-$ by such predicates and obtain a new language which we call Elog$_2^-$. In the long version of this paper, we discuss the generalization of the subelem predicate to support paths with the Kleene star (and ranges). Such an extended subelem predicate remains MSO-definable, but the size of our binary pattern relations can become quadratic in the size of the input tree. Binary intensional predicates allow to explicitly represent the parent-child relationship of the tree computed as a result of the wrapping process, and extend the expressiveness of the wrapping formalism when subelem paths make use of the Kleene star.

DEFINITION 5.7. Let Elog$_2^-$ be a language such as Elog$^-$, but where all pattern predicates are binary and all rules are of the form

$$p(X_0, X) \leftarrow p_0(\_, X_0),\ \text{subelem}_{path}(X_0, X),\ C,\ R.$$

where C is again a set of condition atoms as for Elog$^-$ and R is a set of pattern atoms of the form $p_i(\_, X_i)$. In accordance with Elog$^-$, the predicate "root" is also pro-forma binary and can be substituted as a pattern predicate, although built-in. □

There is a close correspondence between the semantics of Elog$^-$ and Elog$_2^-$.

LEMMA 5.8. *Let $\mathcal{P}$ be an Elog$_2^-$ program and $\mathcal{P}'$ be the Elog$^-$ program obtained by rewriting the i-th rule*

$$\begin{aligned} p(X_0, X) \quad \leftarrow \quad & p_0(\_, X_0),\ \mathit{subelem}_{path_i}(X_0, X),\ C, \\ & p_1(\_, X_1),\ \ldots,\ p_n(\_, X_n). \end{aligned}$$

of $\mathcal{P}$ (where C is a number of condition atoms) into

$$\begin{aligned} p(X) \quad &\leftarrow \quad r_i(X). \\ r_i(X) \quad &\leftarrow \quad p_0(X_0),\ \mathit{subelem}_{path_i}(X_0, X),\ C, \\ & \qquad p_1(X_1),\ \ldots,\ p_n(X_n). \end{aligned}$$

in $\mathcal{P}'$ (where r_i is a new predicate). Then,

1. *$p(v) \in \mathcal{T}^\omega_{\mathcal{P}'}$ iff $p(v, w) \in \mathcal{T}^\omega_{\mathcal{P}}$ for some node w.*
2. *$p(v_1, v_2) \in \mathcal{T}^\omega_{\mathcal{P}}$ iff there is a rule with head predicate p, parent predicate p_0, and, say, index i, in $\mathcal{P}$ s.t. $p_0(v_1), r_i(v_2) \in \mathcal{T}^\omega_{\mathcal{P}'}$ and $\mathit{subelem}_{path_i}(v_1, v_2)$.*

In the above lemma, we have introduced the intermediate step through r_i predicates in order to make the mapping of rules (and their subelem atoms) to head predicates unique.

It immediately follows from Lemma 5.8 and previous theorems that Elog$_2^-$ and Elog$^-$ characterize the same tree languages, can define the same (monadic) queries, and have the same data complexity.

THEOREM 5.9. *(1) A monadic query resp. tree language over documents is definable in Elog$_2^-$ iff it is definable in MSO and (2) the data complexity of Elog$_2^-$ is in linear time.*

Note that Elog$_2^-$ is equally well-suited for visual specification as is Elog$^-$.

The rationale of supporting binary pattern predicates in Elog is to build a child relation for an output XML graph during the wrapping process.

DEFINITION 5.10. The output language of Elog$_2^-$ is defined as follows. An Elog$_2^-$ program $\mathcal{P}$ is a function mapping each document t to a node-labeled directed graph

$$G = \langle \text{dom}(t), E = \{\langle n_1, n_2\rangle \mid p_i(n_1, n_2) \in \mathcal{T}^\omega_{\mathcal{P}}\}, (Q_p)_{p\in P}\rangle$$

where $Q_p = \{n \mid \exists n' : p(n', n) \in \mathcal{T}^\omega_{\mathcal{P}}\}$ and P is the set of pattern predicate names occurring in $\mathcal{P}$. □

The edge relation E constitutes a partial order of the nodes. The graph is acyclic except for the reflexive loops of the form $\langle n, n\rangle \in E$.

THEOREM 5.11. *The relations of G are MSO-definable.*

Proof Let $\mathcal{P}$ be an Elog$_2^-$ program. The case of the Q_p relations was covered in Proposition 3.2, so we only need to show the MSO-definability of E, which is defined as

$$E(x, y) \equiv_{def} \forall P_1, \ldots, P_n : SAT(P_1, \ldots, P_n) \rightarrow \Phi$$

where SAT is based on a version $\mathcal{P}'$ of $\mathcal{P}$ where the first columns of all IDB predicates have been projected out and Φ is a disjunction of formulae B (one for each rule $H \leftarrow B$ in $\mathcal{P}'$) where the variable appearing in H has been replaced by y and the variable appearing in the parent pattern atom of B has been replaced by x. (We assume that x and y are new variables that do not appear in $\mathcal{P}'$). □

6. OTHER WRAPPING LANGUAGES

In this section, we compare the expressiveness of two more wrapping languages, namely regular path queries with nesting and HEL, the wrapping language of the W4F framework [28], to Elog$_2^-$. For space reasons, we have to be extremely brief in this section.

Other previously proposed wrapping languages were evaluated as well. The majority of previous work is string-based (e.g., TSIMMIS [27], EDITOR [5], FLORID [21], DEByE [18], and Stalker [23]) and artificially restricting them in some way to work on trees would not be true to their motivation. Thus, we decided not to include them in this discussion. For some other systems (such as XWrap [20], which is essentially tree-based like W4F or Lixto), no formal specifications have been published which can be made subject to expressiveness evaluations.

Web query languages were also evaluated, but some (e.g., WebSQL [4], WebLOG [19]) are unsuited for wrapping because they cannot access the structure of Web documents, and others (e.g., WebOQL [3]; for a survey of further Web query languages see [14]) are highly expressive query languages that permit data transformations not in the spirit of wrapping.

6.1 Regular Path Queries with Nesting

The first language we compare to Elog$_2^-$ is obtained by combining regular path queries [2] with nesting to create complex structures. This new language – which we will call RPN (**R**egular **P**ath queries with **N**esting) – on one hand is simple yet appropriate for defining practical wrappers, and on the other hand serves to prepare some machinery for comparing further wrapping languages later on. The creation of complex objects is a characteristic common to many wrapping approaches (e.g., [28, 18]), but on which we have insufficiently shed light up to this point.

DEFINITION 6.1. The syntax of *RPN* is defined by the grammar

rpn:	*path*.txt | *path* '(' *rpn* '#' $\cdots$ '#' *rpn* ')'
path:	*patom* '.' $\cdots$ '.' *patom*
patom:	$patom_0$ | $patom_0$ *conds*
$patom_0$:	regexp | regexp '[' range ';' $\cdots$ ';' range ']'
conds:	'{' *cond* 'and' $\cdots$ 'and' *cond* '}'
cond:	*path*'.txt' = string

where a range is either '*', i, or $i-j$ (where i and j are integers), "regexp" denotes the regular expressions over HTML tag names, and "string" the set of strings. □

DEFINITION 6.2. Denotational semantics of RPN. In the following, ρ denotes a regular expression over HTML tags, α a range, s a string, and n, n' denote tree nodes. Without loss of generality, we assume that every *patom* has a range[9]. We have

$$\begin{aligned} \mathbb{E}[\![.\rho[\alpha]X]\!]n &:= \mathbb{E}[\![X]\!]\{n' \mid n.\rho[\alpha] = n'\} \\ \mathbb{E}[\![X_1\#\ldots\#X_n]\!]n &:= \langle \mathbb{E}[\![X_1]\!]n,\ \ldots,\ \mathbb{E}[\![X_1]\!]n\rangle \\ \mathbb{E}[\![.\text{txt}]\!]n &:= \{n.\text{txt}\} \end{aligned}$$

$n.\rho[\alpha]$ denotes the tree nodes reachable from n through the path ρ which satisfy the range α in depth-first left-to-right traversal (where nodes are marked before going down in the depth-first traversal); that is, in document order. Furthermore, we have

$$\begin{aligned} &\mathbb{E}[\![.\rho[\alpha]\{Y_1 \text{ and } \ldots \text{ and } Y_n\}X]\!]n := \\ &\quad \mathbb{E}[\![.\rho[\alpha]X]\!]n \cap \\ &\quad \{n' \mid \mathbb{E}[\![Y_1]\!]n' \neq \emptyset\} \cap \cdots \cap \{n' \mid \mathbb{E}[\![Y_n]\!]n' \neq \emptyset\} \\ &\mathbb{E}[\![.\text{txt} = s]\!]n \ := \ \textbf{if } n.\text{txt} = s \textbf{ then } \{true\} \textbf{ else } \emptyset \end{aligned}$$

for conditions. □

Intuitively, in RPN, all entries in tuples are set-valued and may remain empty. Whenever a condition $\{\text{path.txt} = s\}$ is tested for a node n, there must be at least one node n' reachable from n through path "path" s.t. $n'.txt = s$.

EXAMPLE 6.3. Note that the semantics of paths and conditions in RPN is similar to those of the fragment obtained from XPath [32] by removing the functional aspect of the language. Using the syntax of Definition 6.1, the XPath query

/html/body/table/tr[td[1] = "item"]/td[2];

is written as

html.body.table.tr{td[0].txt = "item"}.td[1];

A path of the form $\cdots//a/\cdots$ in XPath corresponds to $\cdots .\bar{a}^*.a.\cdots$ in RPN.

However, apart from nesting and a more general class of path expressions that can be used in RPN, there is another difference. While XPath selects nodes of a document tree – and the HTML tree of a document is usually irrelevant to a wrapping result per se – RPN extracts text below nodes rather that selecting the nodes themselves. □

THEOREM 6.4. *For each wrapper expressible in RPN, there is an equivalent wrapper in* $Elog_2^-$.

[9]We can always add a range [*] to a *patom* without a range without changing the semantics.

Proof Sketch We use $Elog_2^-$ extended as follows. We use the predicates $\text{subelem}^{(b,f)}_{path,range}$ and $\text{contains}^{(b,f)}_{path,range}$ and allow arbitrary numbers of binary predicates as in monadic datalog. It is easy to verify that this language is not more expressive than $Elog_2^-$.

Ranges in RPN are regular and can be encoded using the $\text{subelem}^{(b,f)}_{path,range}$ and $\text{contains}^{(b,f)}_{path,range}$ predicates. We make a shortcut and assume that each range in RPN is directly understood by the $Elog_2^-$ predicates. Let W be an RPN wrapper in which every *patom* has a range.

We create the $Elog_2^-$ program $\mathcal{P} = f(\text{root}, \text{root}, \underline{W})$ from the query tree of W using the function f defined as follows.

$$\begin{aligned} &f(p', p, \underline{.\rho[\alpha]X}) = f(p'_1, p_1, \underline{X}) \cup \\ &\quad \{\ p'_1(X_0, X) \leftarrow p'(\_, X_0),\ \text{subelem}_{\rho[\alpha]}(X_0, X). \\ &\qquad p(X_0, X) \leftarrow p'(X_0, X),\ \text{contains}_{\rho[\alpha]}(X, Y),\ p_1(\_, Y).\ \} \end{aligned}$$

if $p \neq p'$. Otherwise,

$$\begin{aligned} &f(p, p, \underline{.\rho[\alpha]X}) = f(p_1, p_1, \underline{X}) \cup \\ &\quad \{\ p_1(X_0, X) \leftarrow p(\_, X_0),\ \text{subelem}_{\rho[\alpha]}(X_0, X).\ \} \end{aligned}$$

$$f(p, p, \underline{.txt}) = \emptyset$$

$$\begin{aligned} &f(p', p, \underline{.txt = s}) = \\ &\quad \{\ p(X_0, X) \leftarrow p'(X_0, X),\ \text{contains}_{s'.nil}(X, Y).\ \} \end{aligned}$$

$$\begin{aligned} &f(p', p, \underline{.\rho[\alpha]X\{X_1 \text{ and } \ldots \text{ and } X_n\}Y}) = \\ &\quad f(p_0, p, \underline{Y}) \cup f(p', p_1, \underline{X_1}) \cup \cdots \cup f(p', p_n, \underline{X_n}) \cup \\ &\quad \{\ p_0(X_0, X) \leftarrow p'(X_0, X),\ \textstyle\bigwedge_i p_i(\_, X).\ \} \end{aligned}$$

$$f(p, p, \underline{(X_1\#\ldots\#X_n)}) = f(p, p, \underline{X_1}) \cup \cdots \cup f(p, p, \underline{X_n})$$

where the p, p_0, p_1, p', p'_1 are new predicates, the ρ are regular expressions over HTML tags, the α are ranges, s is a string, and s' is the same string represented as a path of character labels (e.g., "string" becomes "s.t.r.i.n.g").

Now, let M be the fixpoint of $\mathcal{P}$. M is interpreted as follows to obtain the semantics of RPN for W. First we remove all atoms over auxiliary predicates (using prime $'$) and predicates used to verify conditions from M. Then, for each sequence of predicates $p_1, \ldots, p_n$ s.t. $p_1 \ldots p_n$ cover exactly the *patoms* of a *path*[10] (from left to right), we compute $p_n := p_1 \bowtie \cdots \bowtie p_n$ and eliminate all atoms with predicates $p_1, \ldots, p_{n-1}$ from M. Now, we can interpret M as complex objects by assuming each n nested in n_0 if there is an atom $p(n_0, n)$ in M.

The ordering of tuple elements is implicit in the naming of the predicates chosen. Note that in the Lixto system, by convention, computed edges are ordered (in the XML output) in the same way rules in the Elog program are. □

THEOREM 6.5. *There is an* $Elog_2^-$ *wrapper for which no equivalent RPN wrapper exists.*

Proof For trees of depth one, all RPN queries are first-order. We thus cannot check whether, say, the root node has an even number of children (which we can do easily in MSO and thus $Elog_2^-$). □

[10]That is, the p_1 is invented while computing $f(p, p, \underline{.\rho[\alpha]X})$.

6.2 HEL

In this section, we compare the expressive power of the HEL wrapping language of the World Wide Web Wrapper Factory (W4F) with the expressiveness of Elog_2^-. For an introduction to and a formal specification of HEL please refer to [28].

We consider a fragment of HEL called HEL^- which is obtained by taking HEL without string extraction using *match* and *split* expressions (although we support strings in conditions as essential to the philosophy of HEL) and without the getNumberOf and getAttr functions. Note that this is done to compare this language in the framework we have built based on the languages Elog^- and Elog_2^-. Full Elog again supports string extraction in the way HEL does. Using the getNumberOf function, one may require that the number of nodes (in the document tree) reachable through a given path starting from some node is equal to some constant number, which is easy to define in MSO. The getAttr function of HEL extracts HTML attributes, which we manage as tree nodes. Given the assumptions we have made, the function is redundant with those for accessing nodes.

DEFINITION 6.6. The syntax of the language HEL^- is defined by the following grammar.

HEL$^-$:	*cc* | *cc* 'where' *conds*
cc:	*path*.txt | *path* '(' *cc* '#' $\cdots$ '#' *cc* ')'
path:	*patom* (('.'|'$\rightarrow$')*patom*)*
patom:	tag | tag'['*vrange*']'
vrange:	*ranges* | var':' *ranges* | var
ranges:	*range* ';' $\cdots$ ';' *range*
conds:	*cond* ['!'] 'and' $\cdots$ 'and' *cond* ['!']

where "var" is a set of index variable names, "int" is the set of integers, "tag" the set of HTML tag names, "string" the set of strings, and *range* and *cond* are defined as in RPN.

Each index variable used in a wrapper occurs *exactly once* in its *cc* construct. Each cond construct *c* in the where clause of a wrapper is constrained in the way that the smallest prefix of *c* that contains all ranges with index variables has to match a prefix of a path (in terms of both tags and index variables appearing in ranges) that can be constructed by concatenating a path in the cc construct starting from the left and always choosing one element of a record while going to the right. □

Defining the semantics of HEL (and HEL^-) is a tedious task. Here, we will define an alternative variable-free language (called HEL_{vf}^- and described below) which has the desirable property that the semantics of HEL^- and HEL_{vf}^- entail a one-to-one relationship between wrappers in the two languages [6]. This variable-free syntax is possible because of the very special and restricted way in which index variables may be used in HEL.

DEFINITION 6.7. The language HEL_{vf}^- (that is, variable-free HEL^-) mainly inherits the syntax of RPN, except that *patoms* are restricted to the form ('.'|'$\rightarrow$')*t*, conditions may be marked with a prolog-like cut '!', and conditions may not be nested inside conditions.

The semantics of HEL_{vf}^- differs from RPN as follows.

- '$\rightarrow$' denotes reachability of nodes in the tree.
- Condition paths must be single-valued.
- Ranges and the Cut apply relative to document order traversal and only to those nodes for which all given conditions hold (i.e., intuitively, conditions are evaluated "first"). The cut causes the evaluation of a path to stop if a condition marked with the cut is false.

Formally, we can denote this as

$$\mathbb{E}[\![\pi[\alpha]\{\bigwedge_k \pi_k.txt = s_k\}X]\!]n = \mathbb{E}[\![X]\!]Y$$

where π is either $.t$ or $\rightarrow t$ (t is a tag), the π_k are paths without conditions, and $y \in Y \leftrightarrow y \in Z \wedge R_\alpha(y) \wedge \neg!(y)$ s.t. Z is the largest set for which

$$\forall z \in Z : \text{subelem}_\pi(n, z) \wedge C(z)$$

with

$$C(n) \leftrightarrow \bigwedge_k \left(|\mathbb{E}[\![\pi_k]\!]n| = 1 \wedge \mathbb{E}[\![\pi_k]\!]n.txt = s_k\right)$$

and $R_\alpha(y)$ is stated relative to Y; e.g.,

$$R_i(y_i) \leftrightarrow \exists y_0, \ldots, y_i \in Y : \neg\exists y_{-1} \in Y : y_{-1} \prec y_0 \wedge \bigwedge_{0 \le k < i}(y_k \prec y_{k+1} \wedge \neg\exists y' : y_k \prec y' \prec y_{k+1}),$$

and $!(y) \leftrightarrow \exists x : x \preceq y \wedge \text{subelem}_\pi(n, x) \wedge \neg C(x)$ and a condition for n has the cut. □

LEMMA 6.8. **[6]** *A wrapper is expressible in* HEL^- *iff it is expressible in* HEL_{vf}^-.

Proof Sketch A HEL^- wrapper can be easily transformed into HEL_{vf}^- by simply removing its conditions one by one and merging them into the construction part of the wrapper (everything up to the where clause). Starting from the left, each condition is deleted up to the rightmost of its variables, and the remaining condition is nested into the construction part of the wrapper at the position of that variable. For example, the HEL wrapper

```
html.body.table(tr[0].td[0].txt
          #  tr[i:*].td[1].txt)
where html.body.table.tr[i].td[0].txt = "item";
```

can be written as

```
html.body.table(tr[0].td[0].txt
          #  tr[*]{td[0].txt = "item"}.td[1].txt);
```

in HEL_{vf}^-. □

THEOREM 6.9. *For each wrapper expressible in the* HEL_{vf}^- *language, there is an equivalent wrapper in* Elog_2^-.

Proof Sketch By Definition 6.7, which is essentially already stated in MSO. □

Finally, Theorem 6.10 can be justified by the same argument used previously for showing Theorem 6.5.

THEOREM 6.10. *There is an* Elog_2^- *wrapper for which no equivalent* HEL^- *wrapper exists.*

Acknowledgments

We thank Fabien Azavant, Martin Grohe, Frank Neven, and Thomas Schwentick for insightful discussions.

7. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann Publishers, 2000.

[2] S. Abiteboul and V. Vianu. "Regular Path Queries with Constraints". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 11–15, 1997, Tucson, AZ USA*, 1997.

[3] G. Arocena and A. Mendelzon. "WebOQL: Restructuring Documents, Databases, and Webs". In *Proc. ICDE'98*, Orlando, Florida, Feb. 1998.

[4] G. Arocena, A. Mendelzon, and G. Mihaila. "Applications of a Web Query Language". In *Proceedings of the 6th International WWW Conference*, Santa Clara, California, Apr. 1997.

[5] P. Atzeni and G. Mecca. "Cut and Paste". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 11–15, 1997, Tucson, AZ USA*, 1997.

[6] F. Azavant. Personal communication, Oct. 2001.

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

[8] R. Baumgartner, S. Flesca, and G. Gottlob. "Visual Web Information Extraction with Lixto". In *Proc. VLDB'01*, 2001.

[9] A. Brüggemann-Klein and D. Wood. "Regular Tree Languages over Non-ranked Alphabets". Unpublished manuscript, 1998.

[10] S. Ceri, G. Gottlob, and L. Tanca. *"Logic Programming and Databases"*. Springer-Verlag, Berlin, 1990.

[11] 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.

[12] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. "Complexity and Expressive Power of Logic Programming". To appear in *ACM Computing Surveys*.

[13] W. F. Dowling and J. H. Gallier. "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae". *Journal of Logic Programming*, **1**(3):267–284, 1984.

[14] M. Fernandez, J. Siméon, P. Wadler (eds.), S. Cluet, A. Deutsch, D. F. A. Levy, D. Maier, J. M. J. Robie, D. Suciu, and J. Widom. "XML Query Languages: Experiences and Exemplars", 1999. http://www-db.research.bell-labs.com/user/simeon/xquery.html.

[15] J. Flum, M. Frick, and M. Grohe. "Query Evaluation via Tree-Decompositions". In *Proc. of the International Conference on Database Theory*, 2001.

[16] F. Gécseg and M. Steinby. "Tree Languages". In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer Verlag, 1997.

[17] J. E. Hopcroft and J. D. Ullman. *"Introduction to Automata Theory, Languages, and Computation"*. Addison-Wesley Publishing Company, Reading, MA USA, 1979.

[18] 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.

[19] L. V. Lakshmanan, F. Sadri, and I. N. Subramanian. "A Declarative Language for Querying and Restructuring the World-Wide-Web". In *Workshop on Research Issues in Data Engineering (RIDE-NDS'96)*, New Orleans, Feb. 1996.

[20] L. Liu, C. Pu, and W. Han. "XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources". In *Proceedings of the 16th International Conference on Data Engineering*, 1998.

[21] 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.

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

[23] I. Muslea, S. Minton, and C. Knoblock. "STALKER: Learning Extraction Rules for Semistructured, Web-based Information Sources", 1998.

[24] F. Neven and T. Schwentick. "Expressive and Efficient Pattern Languages for Tree-structured Data". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS) 2000*, 2000.

[25] F. Neven and T. Schwentick. "Query Automata on Finite Trees". *Theoretical Computer Science (to appear)*, 2001.

[26] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[27] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. "A Query Translation Scheme for Rapid Implementation of Wrappers". In *Proc. International Conference on Deductive and Object-oriented Databases (DOOD'95)*, pages 97–107, Aug. 1995.

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

[29] A. Szilard, S. Yu, K. Zhang, and J. Shallit. "Characterizing Regular Languages with Polynomial Densities". In *Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science, LNCS 629*, pages 494–503. Springer Verlag, Berlin, 1992.

[30] 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.

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

[32] World Wide Web Consortium. http://www.w3c.org/tr/xpath/.

[33] S. Yu. "Regular Languages". In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 2. Springer-Verlag, 1997.