Highly Expressive Query Languages for Unordered Data Trees

Serge Abiteboul · Pierre Bourhis · Victor Vianu

Abstract We study highly expressive query languages for unordered data trees, using as formal vehicles Active XML and extensions of languages in the *while* family. All languages may be seen as adding some form of control on top of a set of basic pattern queries. The results highlight the impact and interplay of different factors: the expressive power of basic queries, the embedding of computation into data (as in Active XML), and the use of deterministic vs. nondeterministic control. All languages are Turing complete, but not necessarily query complete in the sense of Chandra and Harel. Indeed, we show that some combinations of features yield serious limitations, analogous to FO^k definability in the relational context. On the other hand, the limitations come with benefits such as the existence of powerful normal forms providing opportunities for optimization. Other languages are "almost" complete, but fall short because of subtle limitations reminiscent of the copy elimination problem in object databases.

S. Abiteboul
INRIA & ENS Cachan, 94235 CACHAN Cedex, France
e-mail: Serge.Abiteboul@inria.fr

P. Bourhis
CNRS LIFL & Université Lille 1 & INRIA Lille, 59650 Villeneuve d'Ascq, France
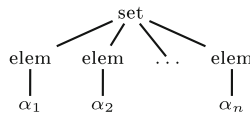e-mail: pierre.bourhis@univ-lille1.fr

V. Vianu (✉)
U. C. San Diego & INRIA-Saclay, La Jolla, CA 92093, USA
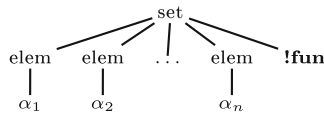e-mail: vianu@cs.ucsd.edu

1 Introduction

In recent years, there has been much interest in query languages on trees, motivated by the ubiquity of XML. Most formal studies have focused on languages of limited expressiveness, with an eye towards efficient evaluation and tractable static analysis. In this paper, we consider the other end of the spectrum: highly expressive query languages for trees with data. Moreover, we focus on *unordered* trees, motivated by considerations familiar from classical databases, including opportunities for optimization provided by set-oriented processing. Our languages use simple tree pattern queries as basic building blocks, and various forms of control to build complex programs. As in the relational case, it is easy to obtain languages that are Turing complete, that is, providing full computational power. However, such languages are not necessarily *query complete*. Indeed, some of them fail to express very simple queries (such as the parity of the number of data values in the document).

To study such languages, we use as a convenient vehicle the language Active XML (AXML) [1]. The language is appealing for such a study because it provides a powerful, elegant mechanism for overcoming limitations such as above and controlling expressiveness. We illustrate this aspect informally next.
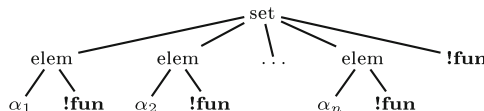
Example 1.1 Consider the tree

This document represents a set of data elements $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Consider the *parity* query, asking whether n is even or odd. This query is notoriously difficult to express in classical relational languages. Now consider AXML, which allows embedding function calls inside trees. Computation in AXML is based on making calls to these functions, which evaluate tree pattern queries. For example, one possibility is to add a function **!fun** under the root:

It turns out that with this placement of the function (referred to as *isolated*), no AXML program using basic tree pattern queries can compute the parity of the set $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Intuitively, similarly to the relational case, the problem lies in the inability to break the symmetry between the set elements in the course of the computation. Now consider instead the following embedding, placing additional functions **!fun** next to every data value:

With this embedding (referred to as *dense*), an AXML program using basic tree pattern queries can easily compute the parity of the data domain. Intuitively, this is due to the ability to break symmetry using calls to functions attached to each element, in nondeterministically chosen order.   □

Our investigation focuses on the impact on expressiveness of code embedding in conjunction with the power of the basic tree pattern queries. We also consider extensions to trees of highly expressive relational languages of the *while* family, and establish tight connections with the AXML languages. The results highlight the interplay of various language features on expressiveness. They provide insight into the specificity of unordered data trees, while also showing some interesting extensions of classical results. In particular, we show how the notion of FO^k definability can be lifted to the context of data trees, yielding a powerful tool for understanding the expressiveness of various languages. We also encounter a new incarnation of the well-known copy elimination problem [5], arising in expressive relational and object-oriented languages.

The AXML model has proven useful in many scenarios. While our focus here is on its ability to define queries, understanding its expressiveness is of interest beyond querying itself. For example, AXML has been proposed as a high-level specification framework for data-centric workflows [2, 6], because it is particularly well suited to describe workflows whose stages correspond to an evolving document. In this context, it is of interest to understand the connection between starting and final states of the workflow. For instance, this transformation underlies the notion of *dominance* [11], introduced as a basic way to compare the expressiveness of workflow formalisms, and is also useful when performing abstraction in hierarchical workflows, by replacing a sub-workflow with a signature specifying the connection between its inputs and outputs. Static analysis can also benefit from information on the expressiveness of AXML fragments, primarily for proving negative results.

We briefly describe the abstraction of AXML used here, based on the GAXML variant of [6]. An instance consists of a forest of unordered, unranked trees whose internal nodes are labeled by tags from a finite alphabet, and whose leaves are labeled by tags, data values from an infinite alphabet, or function symbols. The activation of functions, as well as their return, are controlled by *guards*, which are Boolean combinations of tree pattern queries. Trees evolve under two types of actions: function calls and function returns. A function call creates a fresh workspace initialized by a simple tree-pattern-based query on the current instance. The workspace may in turn contain function calls, and workspaces can thus be created recursively. The answer to a function call consists of a forest which is the answer to a query applied to the final state of its workspace. AXML typically adopts a nondeterministic control semantics, by which transitions are caused by the call or return of a single arbitrarily chosen function whose corresponding guard is true. Alternatively, one can adopt a natural deterministic semantics under which *all* calls and returns whose guards are true are fired simultaneously (analogously to Datalog rules). We can view AXML as a query language whose input is an initial instance and whose output is a tree produced under a designated root (say *Out*). We refer to GAXML viewed as a query language as QAXML thereby stressing that its main role is as a query language.

The main contribution of our work is to highlight new fundamental aspects of querying unordered data trees. Our investigation of the expressibility of query languages for such trees can be viewed as a continuation of works on relational languages (see, e.g., [4]) and object-oriented languages [5]. As discussed earlier, we pay special attention to the impact on expressiveness of the embedding of functions into data, in combinations with restrictions on the tree patterns used by functions, and deterministic or nondeterministic semantics.

A first group of results focuses on the case when the functions are isolated from the data (by disallowing all but trivial embeddings, as in the first embedding in Example 1.1), and the queries used by functions manipulate only data values rather than full subtrees. We show that the resulting expressiveness is analogous to relational languages in the spirit of embedded SQL, consisting of a Turing complete programming language interacting with an underlying database by first-order (FO) queries. In the relational case, such languages are formalized by the *relational machine*, or equivalently, languages of the *while* family augmented with integers [7]. Recall that despite their Turing completeness, these languages are far from query complete; in fact, they are definable in $L^{\omega}_{\infty\omega}$ (infinitary logic with bounded number of variables), they have a 0–1 law, and cannot compute even "simple" queries such as the parity of the domain. We define analogous languages (and nondeterministic variants) for trees and show that QAXML with isolated functions is equivalent to the tree variant of *while* with integers. This allows proving limitations in expressive power analogous to the relational case, but also yields similarly powerful normal forms. For example, every such QAXML query with isolated functions can be evaluated in three phases: (i) a PTIME pre-processing data analysis phase on the trees; (ii) a computation with no data; and (iii) the construction of the final answer in PTIME (with respect to the answer). The normal from is a powerful technical tool. It also suggests opportunities for optimization, since the outcome of the first phase may be much smaller than the original input. In particular, Boolean queries require only phases (i) and (ii), so can be computed by first eliminating data by a PTIME computation, then carrying out the remaining of the computation on a potentially much smaller instance with no data values. This may be seen as an adaptation to trees of similar normal forms that hold in the relational case, where the first pre-processing phase can be defined by a *fixpoint* query [4, 16]. The normal form is also a key technique in understanding the relative expressiveness of various languages and showing sometimes surprising equivalences. Thus, it is instrumental in proving the equivalence of QAXML with isolated functions and tree variants of *while* with integers. It is also key in showing that the nondeterminism does not increase the ability of QAXML with isolated functions to express *deterministic* queries (compared to the deterministic semantics).

The limited expressive power of QAXML with isolated functions is alleviated by allowing arbitrary embedding of functions, yielding QAXML with *dense* functions (as in the second embedding in Example 1.1). In this case, QAXML with nondeterministic semantics allows expressing any computable query over trees, i.e., QAXML is query complete. Intuitively, this is because function embedding allows some form of *data nondeterminism*, i.e., the possibility to nondeterministically choose a data value in a set. This allows to nondeterministically compute an ordering of the data

values. With this ordering, the first phase of the computation permits to fully identify the input, thereby yielding query completeness.

We next consider a deterministic semantics. Rather surprisingly, QAXML with dense functions and deterministic semantics is *not* query complete, so in this case nondeterminism *does* allow expressing more deterministic queries. In fact, we encounter a phenomenon that has already been observed for languages with value invention, namely the well-known *copy elimination problem* [5], precluding completeness even for inputs and outputs of bounded depth. Intuitively, one can obtain *several copies* of the result, but the language does not permit retaining only one final copy.

In the bulk of our study, variables in queries denote atomic data values. We also consider variables denoting subtrees. The use of tree variables provides queries with the ability to perform complex subtree manipulations. As a result, the expressive power is substantially increased. In particular, deterministic QAXML becomes query complete even with isolated functions. Interestingly, the nondeterministic variant falls slightly short of completeness – it expresses a subclass of queries called *weakly nondeterministic*, corresponding intuitively to nondeterminism arising from control rather than choice of data. To render the language fully complete for nondeterministic queries, we need to go beyond isolated functions, although full density is not required. As a side effect of the first result, we obtain a powerful normal form for deterministic QAXML queries with tree variables: embedding of functions can be entirely eliminated with no loss of expressiveness. In the nondeterministic case, embedded functions can be eliminated from the input but must be allowed in intermediate instances produced by function calls.

As previously, we exhibit close connections between QAXML and languages of the *while* flavor, allowing subtree manipulations. The *while* languages are simpler than the previous variants, since integers and other constructs are no longer needed. The results also yield a normal form for the nondeterministic variant of the *while* language, confining all nondeterminism to the last step in the computation.

Related work Our investigation of AXML leverages various techniques of the classical theory of query languages, including expressiveness of FO with a bounded number variables, normal forms, 0-1 laws, and highly expressive languages. This background is reviewed in the next section.

Query and transformation languages on trees have been widely investigated in the context of XML, focusing on abstractions of fragments of XQuery, XPath, and XSLT (see the surveys [17, 18] and [14]). Many of these studies have focused on trees without data (i.e., over a finite alphabet). More recently, trees with data have been studied. Much of this work is geared towards static analysis, so aims to capture computations of limited expressiveness for which questions such as emptiness remain decidable [10, 19, 20]. There is little work on highly expressive languages on trees with data, and it usually adopts a model of *ordered* unranked trees (siblings are ordered) [9, 12, 13, 15]. In contrast, we consider a model of *unordered* trees. This is in the spirit of the relational model where the order of tuples in relations is immaterial. The intuition is that we focus on the essence of the information rather than on aspects of its representation such as an ordering of data elements. The absence of order is also a source of opportunities for optimization and set-oriented parallel processing, and

presents advantages for static analysis. This difference in focus renders our results incomparable to the cited work.

Organization After some preliminaries, Section 3 introduces QAXML query languages. QAXML with isolated functions is studied in Section 4 and with dense functions in Section 5. The impact of tree variables (deep equality and tree copying) is discussed in Section 6.

2 Preliminaries

We briefly recall some background on relational query languages. See [4, 16] for formal and detailed presentations. We assume an infinite set **dom** of data values, and an infinite set of *variables*, disjoint from **dom**. A relational schema σ is a finite set of relation symbols with associated arities. An instance over σ provides a finite relation of appropriate arity over **dom** for each symbol in σ. First-order (FO) queries over σ are defined as follows. An atom is $R(x_1, \ldots, x_m)$ or $x_1 = x_2$, where R is a relation in σ of arity m and each x_i is a variable or data value (always interpreted by the identity). Formulas are obtained by closing the set of atoms under $\wedge$, $\vee$, $\neg$, $\forall$, and $\exists$, in the usual way. We use the standard active domain semantics, which limits the ranges of variables to the data values occurring in the current instance or in the query.

A query language is *query complete* if it expresses all computable queries. In the classical relational context, it is generally assumed that queries produce answers using only data values from the input (perhaps augmented with a finite set of values explicitly mentioned in the query) and that queries are deterministic. Nondeterministic variants of query completeness have also been defined, some allowing new values in answers to queries.

FO is not query complete and in fact cannot express simple queries such as the transitive closure of a graph. This can be partly alleviated by augmenting FO with a recursion mechanism. Many extensions of FO with recursion converge around two robust classes of queries: *fixpoint* and *while*. We recall two imperative languages expressing these classes. The language *while* (homonymous with the class) extends FO with (i) relation variables to which FO queries can be assigned (with destructive semantics), and (ii) a looping construct of the form *while $R \neq \emptyset$ do*. The *while* queries are those expressed in this language. The *fixpoint* queries are expressed by *while$^+$*, an inflationary variant of *while* obtained by giving *cumulative* semantics to assignments and replacing the looping construct with *while change do*. Note that because of the cumulative assignment, the contents of relation variables is increasing. The loop stops when two consecutive iterations produce no change to the contents of the relation variables (i.e. a fixpoint is reached). Clearly, every query in *while$^+$* is in PTIME with respect to the size of the input (for fixed schema), and every query in *while* is in PSPACE. To break the PSPACE barrier, one possibility is to make *while* Turing complete by augmenting it with integer variables, increment and decrement instructions, and looping of the form *while $i > 0$ do*. Indeed, this allows simulating counters machines, which are computationally complete. The extended language is denoted *while$_\mathbb{N}$*. It partially achieves the goal of increased expressiveness by being query complete on *ordered* databases. However, there remain very simple queries

that are not expressible in the absence of order, such as the parity of the domain. A measure of the expressiveness limitations of $while_\mathbb{N}$ is that it has a 0-1 law, i.e. the probability that a program with Boolean answer in this language returns *true* for instances of size n converges to zero or to one when n goes to infinity.

The expressiveness of $while_\mathbb{N}$ and variants of this language is illuminated by a powerful normal form allowing to reduce in PTIME the evaluation of any such program to a computation on integers. Intuitively, the integers correspond to equivalence classes of tuples that are manipulated together by the program. More precisely, consider a $while_\mathbb{N}$ program that refers to some finite set C of data values, and whose FO queries use at most k variables. It is easy to see that every relation constructed in a specific execution of the program is definable by composing FO^k formulas of the program (yielding another FO^k formula). Consider an instance I, the set C of constants, and let $\equiv_{I,k,C}$ be the equivalence relation on tuples of arity $l \leq k$ defined as follows: for every $\varphi \in FO^k$ mentioning data values in C and having l free variables, $\bar{a} \in \varphi(I)$ iff $\bar{b} \in \varphi(I)$. The following key fact holds (see [16]). There exists a *fixpoint* query Φ (mentioning data values in C) that, on input I, computes the following:

- the equivalence classes of $\equiv_{I,k,C}$;
- a total order on the above equivalence classes.

By definition, all relations constructed from I by FO^k formulas are unions of classes of $\equiv_{I,k,C}$. Since the classes are ordered, they can be viewed as integers, and each relation as above as the set of integers corresponding to the equivalence classes it contains. To show the normal form, one needs to be able to evaluate an FO^k formula directly on the integer representation, without recourse to the actual equivalence classes. To do so, we need sufficient information on the action of such formulas on the equivalence classes. One can show that there exists a finite set F^k of conjunctive queries with at most k variables such that every FO^k formula over a given schema can be evaluated by applying queries in F^k, together with union and negation. For each $q \in F^k$, let $a(q)$ be the number of atoms in q. It can be shown that there exists a *fixpoint* query Ψ which computes, for each $q \in F^k$, a relation $Action_q$ providing, for each $a(q)$-tuple of equivalence classes of $\equiv_{I,k,C}$, the result of applying q to that tuple. Clearly, the instance $Action(I, k, C) = \{Action_q \mid q \in F^k\}$ provides the needed information for evaluating FO^k queries directly on the integers representing the equivalence classes of $\equiv_{I,k,C}$. The normal form for $while_\mathbb{N}$ then follows.

As a useful application of the normal form technique, consider the extension of $while_\mathbb{N}$ allowing to store integers mixed together with data in relational variables, denoted $while_\mathbb{N}^*$. More precisely, this is done by an assignment instruction $X := \langle i \rangle$ where X is a unary relational variable and i an integer variable. It turns out that this seemingly more powerful language remains equivalent to $while_\mathbb{N}$. This is shown by extending the normal form to $while_\mathbb{N}^*$, by considering "slices" of relations sharing the same integer components, and showing that their data portions remain definable in FO^k. As a consequence, all properties (and queries producing only data values) remain definable in $while_\mathbb{N}$ [8].

One way to obtain a query complete language is to extend *while* with the ability to introduce new data values throughout the computation. This is done by an instruction $X := new(Y)$, where X, Y are relational variables and $arity(X) = arity(Y) + 1$.

This inserts in X all tuples of Y extended with an additional coordinate containing a distinct new data value for each tuple (akin to a nondeterministically chosen tuple identifier). It turns out that this language, denoted $while_{new}$, is query complete for queries whose answers do not contain invented values. Interestingly, the language is *not* complete when invented values are allowed in the answer, due to the notorious *copy elimination problem* [5].

Trees The data trees we consider are labeled, unranked and unordered. We assume given the following disjoint infinite sets: *nodes* $\mathcal{N}$ (denoted n, m), *tags* Σ (denoted $a, b, \ldots$), *data values* $\mathcal{D}$ (denoted $\alpha, \beta, \ldots$), possibly with subscripts. A *tree* is a finite binary (parent) relation over $\mathcal{N}$ where all nodes have a single parent except for one (the root). A tree also has a labeling function assigning a tag or data value to every node, with data values only assigned to leaves. We also assume that the trees are *reduced*, i.e., a node cannot have two sibling subtrees that are isomorphic by a mapping preserving tags *and* data values. This is analogous to the set (rather than bag or list) semantics for relational databases. The set of data values occurring in a tree I is denoted $dom(I)$.

Tree queries Let Σ be a finite set of tags. We define the semantic notion of *computable query for trees* over Σ, by extending the classical notion of computable query for relational databases. The input trees may be constrained by a DTD Δ.

We use the following notions:

C-genericity: We extend the notion of *C*-genericity for some finite set C of data values. A relation $\mathcal{R}$ on trees with tags in Σ is *C*-generic if it is closed under all isomorphisms that preserve Σ and C (but may rename all other data values). More precisely, $\mathcal{R}$ is *C*-generic if for each one-to-one mapping ρ over $\mathcal{N} \cup \mathcal{D} \cup \Sigma$ such that $\rho(\mathcal{N}) \subseteq \mathcal{N}$, $\rho(\mathcal{D}) \subseteq \mathcal{D}$, and ρ is the identity on $\Sigma \cup C$, $(I, J) \in \mathcal{R}$ iff $(\rho(I), \rho(J)) \in \mathcal{R}$.

Computability: The notion of computable is standard: A relation $\mathcal{R}$ is computable if there exists a nondeterministic Turing machine $M_{\mathcal{R}}$ that, given any order $\leq$ on data values and a standard encoding $enc_{\leq}(I)$ of an input tree I on its tape, has a terminating computation on input $enc_{\leq}(I)$ with output $enc_{\leq}(J)$ iff $\langle I, J \rangle \in \mathcal{R}$.

Definition 2.1 *A tree query is a computable, C-generic relation $\mathcal{R}$ from trees over Σ satisfying Δ to trees over Σ, such that, for every $\langle I, J \rangle \in \mathcal{R}$: (i) $dom(J) \subseteq dom(I) \cup C$, and (ii) I and J have disjoint sets of nodes.*

Condition (ii) in the definition is motivated by the fact that we do not view the specific node ids as semantically significant. We say that a tree query language is *query complete* if it expresses exactly the set of all tree queries.

The definition of deterministic query is somewhat subtle. Since tree queries produce as outputs trees with new nodes, genericity precludes uniqueness of the result (intuitively, all choices of new nodes must be allowed). To overcome this problem we define a query $\mathcal{R}$ to be deterministic if it provides a unique answer for each input up to renaming of the nodes (labels remain unchanged). A tree query language is *deterministic query complete* if it expresses all deterministic tree queries.
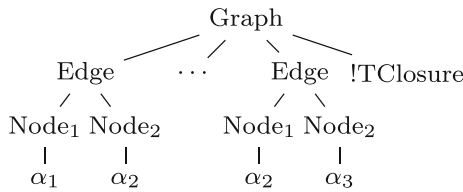
$$\begin{array}{c}
\text{Graph} \\
\diagup \quad \diagup \quad \diagdown \quad \diagdown \\
\text{Edge} \quad \cdots \quad \text{Edge} \quad \text{!TClosure} \\
\diagup \diagdown \qquad \diagup \diagdown \\
\text{Node}_1 \ \text{Node}_2 \quad \text{Node}_1 \ \text{Node}_2 \\
| \qquad | \qquad\quad | \qquad | \\
\alpha_1 \qquad \alpha_2 \qquad \alpha_2 \qquad \alpha_3
\end{array}$$

Fig. 1 AXML tree

3 AXML Query Languages

We introduce in this section several query languages based on an abstraction of AXML.

We assume given an infinite set $\mathcal{F}$ of *function names*. For each function name f, we also use the symbols $!f$ and $?f$, called *function symbols*, and denote by $\mathcal{F}^!$ the set $\{!f \mid f \in \mathcal{F}\}$ and by $\mathcal{F}^?$ the set $\{?f \mid f \in \mathcal{F}\}$. Intuitively, $!f$ labels a node where a call to function f can be made (possible call), and $?f$ labels a node where a call to f has been made and some result is expected (running call). After the answer of a call at node n is returned, the node n is deleted.

An AXML tree is a tree whose internal nodes are labeled with tags in Σ and whose leaves are labeled by either tags, function symbols, or data values. An AXML forest is a set of AXML trees. An example of AXML tree is given in Fig. 1.

To avoid repetitions of isomorphic sibling subtrees, we define the notion of reduced tree. A tree is *reduced* if it contains no distinct isomorphic sibling subtrees without running calls $?f$. We henceforth assume that all trees considered are reduced, unless stated otherwise. However, the forest of an instance may generally contain multiple isomorphic trees.

DTD Trees may be constrained using DTDs. Because our trees are unordered, we use a variant of DTDs that restricts, for each tag $a \in \Sigma$, the labels of children that a-nodes may have.[1] As our trees are unordered, we use Boolean combinations of statements of the form $|b| \geq k$ for $b \in \Sigma \cup \mathcal{F}^! \cup \mathcal{F}^? \cup \{dom\}$ and k a non-negative integer. Validity of trees and of forests relative to a DTD is defined in the standard way. For simplicity we assume that all DTDs specify trees with the same root labeled r. We call a DTD *static* if it does not allow function symbols, and *active* otherwise.

Patterns We use patterns as basic building blocks for our query languages. A *pattern P* is a *tree-pattern* together with a *condition*, defined next. We use two sorts of variables: *structural* variables $V, W, \ldots$ that bind to nodes labeled by tags and function symbols, and *data* variables $X, Y, \ldots$ binding to nodes labeled by data values. A *tree-pattern* is a tree whose nodes are labeled by distinct variables, and whose edges are labeled by / (child) or // (descendant), where descendant is reflexive. Additionally, each node has associated with it a *sign*: positive or negative. The default sign is positive, and we indicate nodes of negative sign by a label ¬. The root of each tree pattern must be positive. We call a node in the tree pattern T a *boundary node* if it

---

[1]Alternatively, we could use automata on unordered trees.

is the root or a node labeled $\neg$. For each subtree S of T rooted at a positive node, we denote by S^+ the tree obtained by removing all its subtrees rooted at negative nodes (including their roots). We associate to each boundary node b of T a set of variables $var(b)$ defined recursively as follows. For the root r, $var(r)$ is the set of variables in T^+. For an arbitrary boundary node b, $var(b)$ is the union of the variables in $var(b')$ for the boundary nodes b' that are ancestors of b, together with the variables in S_b^+, which is the subtree of T rooted at b where the sign of b is made positive. The *condition* of T is a mapping *cond* associating to each boundary node b a Boolean combination of equalities over $var(b)$ of the form:

- $V = t$, where V is a structural variable and t is a tag or function symbol; and
- $X = Y$, where X is a data variable and Y is a data variable or a data value.

A pattern P is a pair $(T, cond)$, where T is a tree pattern and *cond* a condition for T. By slight abuse, we sometimes refer to nodes of P, meaning nodes in its tree pattern T.
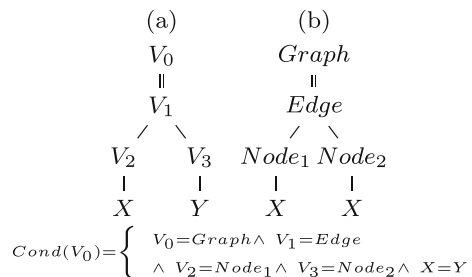
Let $P = (T, cond)$ be a pattern. The set of bindings of P into an AXML forest I is defined by structural recursion on P as follows. A binding of P into I is a mapping ν from $var(T^+)$ to the nodes of I such that:

- The child and descendant relations are preserved.
- For each data variable X, $\nu(X)$ is a node labeled by a data value.
- $cond(r)$ is satisfied. More precisely, an equality $V = t$ is satisfied for a structural variable V if the label of $\nu(V)$ equals t, and $X = Y$ is satisfied for data variables X, Y if the data values labeling $\nu(X)$ and $\nu(Y)$ are equal (and similarly when Y is a data value).
- For each maximal subtree N of T rooted at a negative node b, there is no extension of ν to a binding of $T \oplus N$ where $T \oplus N$ is obtained from T by removing the label $\neg$ from the root of N, such that ν satisfies $cond(b)$.

Given an AXML forest I and a pattern P, we denote by $Bind(P, I)$ the set of bindings of P into I. We say that I satisfies P, denoted $I \models P$, if $Bind(P, I) \neq \emptyset$.

Example 3.1 Figure 2 shows a very simple pattern. When conditions uniquely specify labels of nodes, we use an intuitive representation, as the right pattern in Fig. 2. This cannot always be done. For example, if for the same tree pattern the condition is $V0 = Graph \;\wedge\; V2 = Node1 \;\wedge\; V3 = Node2 \;\wedge\; (V1 = Self - Loop \rightarrow$

Fig. 2 A simple pattern: full specification (a) and concise version (b)

$X = Y) \wedge (V1 = Edge \rightarrow X \neq Y)$ then there is no fixed assignment of labels to nodes. Finally, a more complex pattern with negation, and its concise representation, are shown in Fig. 3.

We sometimes use patterns that are evaluated relative to a specified node in the tree. More precisely, a *relative* pattern is one whose conditions may use equalities of the form $V = self$ where *self* is a new symbol. A relative pattern is evaluated on a pair (I, n) where I is a forest and n is a node of I. An equality $V = self$ is satisfied by a binding ν if $\nu(V) = n$.

Pattern Queries As previously mentioned, patterns are the building blocks for our basic queries, as shown next. A *pattern query* is a finite set of rules of the form $Body \rightarrow Head$, where *Body* is a pattern and *Head* is a tree whose internal nodes are labeled by tags, and leaves are labeled by tags, function symbols in $\mathcal{F}^!$, or data variables in $Body^+$. In addition, all variables in *Head* occur under a designated *constructor node* (marked by set brackets), specifying a form of nesting. When evaluated on a forest I, the answer is obtained using the bindings of $Body^+$ into I. The answer for the rule is obtained by replacing in *Head* the subtree T rooted at the constructor node with a forest containing, for each $\nu \in Bind(Body, I)$ a new copy of T in which each label X is changed to the data value labeling $\nu(X)$. The answer to the pattern query is the union of the answers for each rule (so a *set* of trees). A simple example of a pattern query is shown in Fig. 3. Its body is the pattern in Fig. 3.

Note that, according to this definition, variables in heads of queries extract data values from the input. We will consider in Section 6 an extension allowing variables in heads to extract entire subtrees from the input.

$$Boundary\ nodes = \{V_0, V_2, V_4\}$$
$$Var(V_0) = \{V_0, V_1, X_1\}$$
$$Cond(V_0) = V_0 = r \wedge V_1 = P$$
$$Var(V_2) = Var(V_0) \cup \{V_2, V_3, X_2, X_3\}$$
$$Cond(V_2) = Cond(V_0) \wedge V_2 = R \wedge V_3 = tup \wedge X_1 = X_2$$
$$Var(V_4) = Var(V_2) \cup \{V_4, X_4\}$$
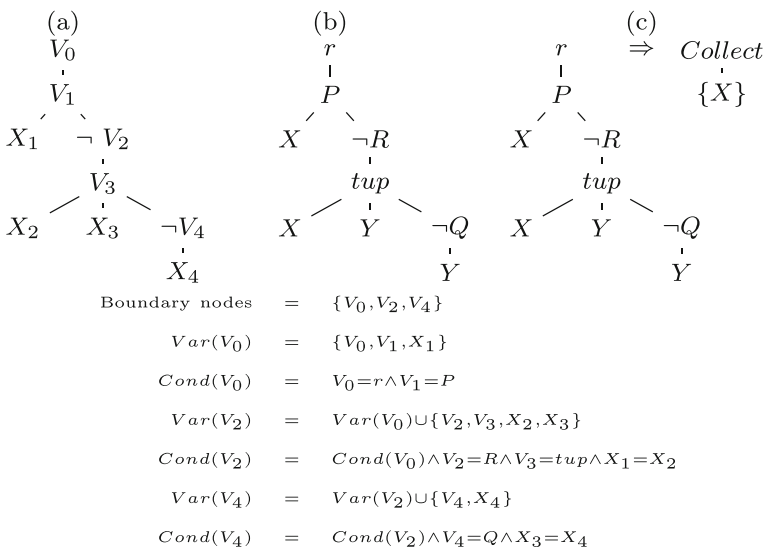$$Cond(V_4) = Cond(V_2) \wedge V_4 = Q \wedge X_3 = X_4$$

Fig. 3 A complex pattern: (a) full specification (b) concise version (c) a query using the pattern

As for patterns, we may consider queries evaluated relative to a specified node in the input tree. A *relative pattern query* is defined like a pattern query, except that the bodies of its rules are relative patterns.

Programs and instances A QAXML *program* $\mathcal{Q}$ is a pair (Φ, Δ) where Φ is a set of function definitions, and Δ is a DTD constraining the initial instance.

We next provide more details, starting with Φ. For each $f \in \mathcal{F}$, let a_f be a new distinct label in Σ. Intuitively, a_f will be the root of a subtree where a call to f is being evaluated (this may be seen as a workspace for the evaluation of the call). The specification of a function f of Φ provides a *call guard* (Boolean combination of patterns), its *input query* (a relative query), *return guard* (Boolean combination of patterns with roots labeled a_f), and *return query* (a pattern query with rules whose bodies have roots labeled a_f). When the input query is evaluated, *self* binds to the node at which the call $!f$ is made. The role of the input query is to define the initial state of the workspace of the call to f.

An AXML *instance* I is a pair $(\mathcal{T}, eval)$, where $\mathcal{T}$ is an AXML forest and *eval* an injective function over the set of nodes in $\mathcal{T}$ labeled with $?f$ for some $f \in \Phi$ such that: (i) for each n with label $?f$, $eval(n)$ is a tree in $\mathcal{T}$ with root label a_f (its workspace), and (ii) every tree in $\mathcal{T}$ with root label a_f is $eval(n)$ for some n labeled $?f$. Figure 4 shows an AXML instance.

The standard semantics of AXML is nondeterministic. At each step of a computation, one function is called or one function call returns its answer. Alternatively, one can provide a deterministic semantics, in which all calls and returns whose guards are true take place simultaneously. This distinction is in the spirit of a distinction that has been considered for Datalog programs, for which rules may be fired simultaneously or one at a time. We denote the nondeterministic variant by NQAXML and the deterministic one by DQAXML.

Nondeterministic semantics We first define the standard nondeterministic semantics, yielding the language NQAXML. Let $I = (\mathcal{T}, eval)$ and $I' = (\mathcal{T}', eval')$ be instances. The instance I' is a *possible next instance of* I iff I' is obtained from I by making a call to some function whose call guard is true, or by returning the answer to an existing call whose return guard is true. We denote by $I \vdash I'$ the fact that I' is a possible next instance of I.

We now provide more details. When a call to $!f$ is made at node n, the label of n is changed to $?f$ and we add to the graph of *eval* the pair (n, T') where T' is a tree consisting of a root a_f connected to the forest that is the result of evaluating the input query of f on input $(\mathcal{T}, n)$. When an answer to call $?f$ at node n is received, the trees in the answer are added as siblings of n, and n is deleted. The answer can be returned

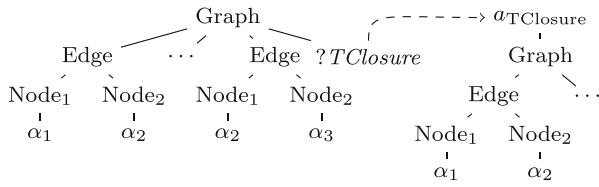**Fig. 4** An AXML instance with an **eval** link

only if $eval(n)$ contains no running calls $?g$, in which case the answer consists of the result of evaluating the return query of f on $eval(n)$, after which $(n, eval(n))$ is removed from the graph of $eval$.

Figure 4 shows a possible next instance for the instance of Fig. 1 after a call has been made to !TClosure.

We are interested in *computations* of NQAXML programs. An *initial* instance of program $Q = (\Phi, \Delta)$ is an instance consisting of a single tree satisfying Δ. A *computation* of Q is a maximal sequence $\{(I_i)\}_{0 \le i < n}$, such that $n \in \mathbb{N} \cup \{\omega\}$, I_0 satisfies Δ, and for each i, $0 < i < n$, $I_{i-1} \vdash I_i$. A computation is *terminating* if it is finite.

Deterministic semantics QAXML programs can be given deterministic semantics by firing at each transition all function calls and function returns whose guards hold in the current instance. The notion of computation is defined analogously to the nondeterministic case. Of course, a QAXML program with deterministic semantics has only one computation on each given input. QAXML programs with deterministic semantics are denoted by DQAXML. We continue to refer to QAXML to denote programs with either deterministic or nondeterministic semantics. $eval(n)$

QAXML as a query language Consider a tree query $\mathcal{R}$ with input DTD Δ. Recall that inputs and outputs of tree queries have no function symbols. In order to compute $\mathcal{R}$ using QAXML, we add function calls to the input under certain nodes. An answer is the tree found under a designated new tag *Out* whenever the program terminates. An example of a QAXML program computing the transitive closure of a graph is provided next.

Example 3.2 We exhibit a QAXML program with data variables, computing the transitive closure of a graph. A directed graph is represented as in Fig. 5. The QAXML program uses two functions: *TClosure* to initialize the output and *Iterate* to perform each iteration in the computation of the transitive closure.

The tree in Fig. 6 represents the initial instance of the QAXML program. It is obtained from the input tree in Fig. 5 by adding a function call !*TClosure* under the root. A call to this function returns a copy of the input graph and adds a function call !*Iterate* (Fig. 7). Each call to *Iterate* performs one iteration in the computation of transitive closure. It returns the edges obtained in the current iteration and, if the last iteration has not yet been reached, a new call !*Iterate*. In more detail, a call to *Iterate* first creates a workspace containing the edges of the current iteration (new and old) under tag *NewEdges*, and separately a copy of the old edges under tag *OldEdges*. The input query of *Iterate* is shown in Fig. 8. An instance obtained by the activation of *Iterate* is depicted in Fig. 9.
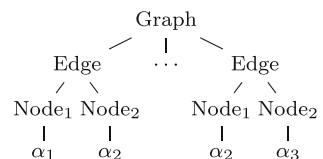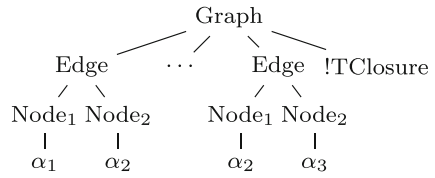
Fig. 5 Input graph

Fig. 6 QAXML initial instance

The function *Iterate* returns the set of edges under *NewEdges* that are not also under *OldEdges*. If this set is not empty (so the last iteration has not been reached), it also returns a new call to *Iterate*. The return query of *Iterate* is shown in Fig. 10.

The computation terminates when no new edges are added.

We will study two extreme function embedding policies: (i) the only function call allowed in the input is under the root, and (ii) function calls are placed under every node in the input (except those labeled by data values). Intermediate restrictions on embeddings can be defined by various means that we leave open, for instance by specifying the parents of function calls using their tags, by an MSO formula, etc. As we shall see, the allowed embedding of function calls into the input has drastic impact on expressiveness. We consider (i) in Section 4, then (ii) in Section 5.

4 QAXML with Isolated Functions

A main objective of the current study is to understand the impact on expressiveness of function calls embedded in the data. We first consider the case when there is no non-trivial embedding in the input, coupled with a restriction on how new functions can be introduced. Without loss of generality, we can assume that the initial instance contains only one function symbol $!f$ (other functions can be added if desired by a call to that function).

Definition 4.1 *A QAXML program with isolated functions is a pair* $\mathcal{Q} = (\Phi, \Delta)$ *where* Δ *is a static DTD and for every query rule* Body$\rightarrow$ Head *used in* Φ, *no function symbol occurs under the constructor node in Head. For an instance* I *satisfying* Δ, *we denote by* $I^!$ *the instance obtained by adding a call* $!f$ *under the root of* I. *The program* $\mathcal{Q}$ *expresses a tree query* $\mathcal{R}$ *with input DTD* Δ *if for every* I *satisfying*
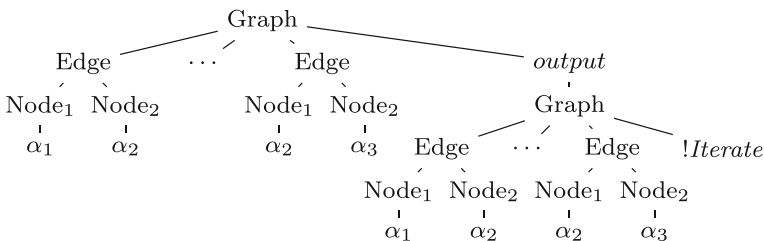

Fig. 7 QAXML instance after return of TClosure

Fig. 8 Input query of *Iterate*

Δ, $(I, O) \in \mathcal{R}$ *iff there exists a computation of* $\mathcal{Q}$ *on* $I^!$ *terminating with* O *as the unique subtree of a unique node labeled* Out *(where* Out *is a new tag).*

For instance, the QAXML program in Example 3.2 is a program with isolated functions. The isolation restriction places drastic limitations on the expressive power of QAXML programs. Rather surprisingly, it turns out that this is closely related to definability by FO with a bounded number of variables, a restriction well explored in the theory of relational query languages [16]. We first elaborate on this connection, which provides a key technical tool. We then use it to establish equivalencies to languages in the *while* family, extended to data trees, as well as to present a powerful normal form.

4.1 Isolated Functions and FO^k Definability

We begin with an informal description of the connection between QAXML with isolated functions and FO^k definability. Let $\mathcal{Q}$ be a QAXML program with isolated functions, with deterministic or nondeterministic semantics. Suppose $\mathcal{Q}$ uses a finite set C of data value constants in its patterns. Consider a computation of $\mathcal{Q}$ on input I. In the course of the computation, I remains unchanged and function calls generate another subtree under the root r, as well as a forest of workspaces siblings to r. When a tree pattern query is evaluated, a portion is bound to I and the rest to trees outside I. The bindings to I can be pre-computed for all relevant subpatterns and stored in a relational structure $\sigma(I)$. Now consider the trees built in the course of the computation. Recall that data values are introduced in such trees using pattern queries, by instantiating subtrees in the head rooted at constructor nodes with bindings of the data variables. Let us call nodes obtained by such instantiations *expanded* nodes. Let
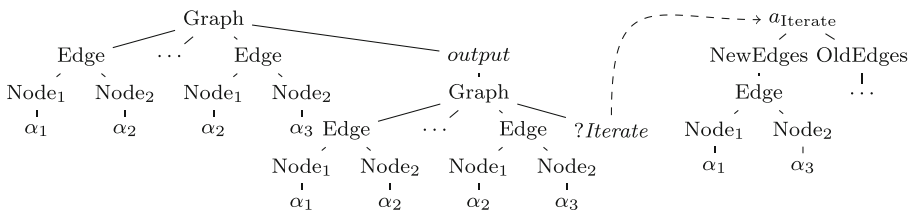

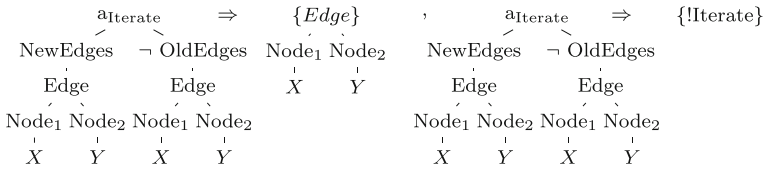**Fig. 9** QAXML program after activation of *Iterate*

$$\begin{array}{cccc} a_{\text{Iterate}} & \Rightarrow & \{Edge\} & , & a_{\text{Iterate}} & \Rightarrow & \{!\text{Iterate}\} \end{array}$$

NewEdges ¬ OldEdges Node₁ Node₂   NewEdges ¬ OldEdges

Edge   Edge X Y Edge   Edge

Node₁ Node₂ Node₁ Node₂   Node₁ Node₂ Node₁ Node₂

X Y X Y X Y X Y

Fig. 10 Return query of *Iterate*

R_ε be the relation consisting of all bindings used in a given step of the computation to produce expanded nodes by applying a particular query rule. We will show the following key fact:

There exists $k > 0$ depending only on $\mathcal{Q}$ such that each R_ε is definable from $\sigma(I)$ by an FO^k formula (using only constants in C).

Recall that every relation definable in FO^k from $\sigma(I)$ is a union of classes of the equivalence relation $\equiv_{\sigma(I),k,C}$. Intuitively, this captures the distinguishing power of $\mathcal{Q}$ with regard to data values. Computation on the actual data can in fact be replaced with computation on the equivalence classes of $\equiv_{\sigma(I),k,C}$, augmented with a total order on the classes and the structure $Action(I, k, C)$ summarizing the action of FO^k queries on the equivalence classes (see the preliminaries). These can be computed by a *fixpoint* query, so also by a QAXML program with isolated functions (in PTIME). Because of the total order, the classes of $\equiv_{\sigma(I),k,C}$ can henceforth be abstracted as integers. As we will see, this provides a powerful technical tool.

We now provide more details. Let $\mathcal{Q}$ be a QAXML program with isolated functions. In the course of the computation of $\mathcal{Q}$ on input I, a tree is generated next to I under r, together with a forest of workspaces sibling to r. As discussed earlier, when a tree pattern is evaluated, a portion is bound to I and the rest to trees outside I, which may be siblings of I under r, or workspaces rooted at a_g for some function g. We show how to pre-compute the relational structure $\sigma(I)$ holding the result of evaluating on I a set of subpatterns depending only on $\mathcal{Q}$. Consider a pattern $P = (T, cond_P)$ of $\mathcal{Q}$ where T has root r. Every child subtree S of r in T can generally extract some bindings from I. Recall that S can only extract data bindings using the data variables in S^+. However, the conditions attached to S use (i) structural variables in $var(T^+)$ and (ii) data variables in $var(T^+)$ which may include variables not in S. To evaluate each S independently, we do the following. To account for (i), we consider different instantiations of S for each assignment of tags, function symbols, or *self* to the structural variables in T^+. To account for (ii), we augment S with a subtree extracting all assignments of data values to the data variables in T^+ that are not in S^+. Now the bindings extracted by the different S can be combined by joining them. The relational structure $\sigma(I)$ contains the sets of bindings extracted by each such S, for all patterns P rooted at r.

In more detail, let $P = (T, cond_P)$ be a pattern of $\mathcal{Q}$ as above, where T has root r. Let $svar(T^+)$ be the set of structural variables of T^+, and $dvar(T^+)$ the set of data variables of T^+. Let Γ be the set of assignments of tags, function symbols of $\mathcal{Q}$, or *self* to $svar(T^+)$, and for each $\gamma \in \Gamma$, let $cond_\gamma$ be the condition $\wedge\{V = \gamma(V) \mid V \in svar(T^+)\}$. Let $\mathcal{S}$ be the set of subtrees S of T whose roots are children

of r. For each $S \in \mathcal{S}$ and each $\gamma \in \Gamma$, we define a pattern S_γ rooted at r with subtrees $/S$ and $\{ //X \mid X \in dvar(T^+) - dvar(S^+) \}$ and condition defined by $cond(r) = cond_P(r) \wedge cond_\gamma$ and $cond(b) = cond_P(b)$ for all boundary nodes of S.

Note that for each pattern P, the set of bindings of $dvar(T^+)$ on a given instance can be computed by applying independently the patterns extracted from T as above, and then combining the results. More precisely, the set of bindings is obtained by the following "formula" :

$$(\dagger) \quad \bigvee_{\gamma \in \Gamma} (\wedge_{S \in \mathcal{S}} S_\gamma)$$

To each pattern P, $\gamma \in \Gamma$ and S_γ as above we associate a relation $R_{S,\gamma}$ of arity $|dvar(T^+)|$. Let σ be the schema consisting of all such relations. For an input I, let $\sigma(I)$ be the relational structure obtained by evaluating each S_γ on I.

Now consider again the evaluation of a pattern P rooted at r in the course of the computation of $\mathcal{Q}$ on I. In view of $(\dagger)$, it follows that the set of bindings of $dvar(T^+)$ on the current instance can be obtained using only $\sigma(I)$ and evaluating the patterns of T on the tree from which I has been removed. We make this more precise. Let Pos by the set of patterns S_γ constructed from P as above where the root of S is positive, and Neg the set of S_γ with negative root. The set of bindings is:

$$(\ddagger) \quad \bigvee_{\gamma \in \Gamma} \left(\wedge_{S \in Pos} \left(R_{S,\gamma} \left(\bar{X} \right) \vee S_\gamma \left(\bar{X} \right) \right) \wedge_{S \in Neg} \left(R_{S,\gamma} \left(\bar{X} \right) \wedge S_\gamma \left(\bar{X} \right) \right) \right)$$

where $\bar{X} = dvar(T^+)$ and $S_\gamma(\bar{X})$ is evaluated on the current instance from which I has been removed. This assumes that the remaining instance contains all data values in I, which can be easily ensured.

Now consider a computation of $\mathcal{Q}$ on input I. Recall the definition of expanded nodes generated in the course of the computation. Consider the expanded trees obtained as the answer to a a a rule $Body \rightarrow Head$ of a pattern query, with set $\mathcal{B}$ of bindings for the m variables in the head of the rule. To each such set ε of trees we associate a relation R_ϵ of arity m containing the bindings in $\mathcal{B}$.

The following key fact can be shown.

Lemma 4.2 *Each relation R_ε generated in the course of the computation of $\mathcal{Q}$ on I as above is definable by an FO^k query from $\sigma(I)$, for some k depending only on $\mathcal{Q}$.*

Proof We provide the proof for NQAXML. The deterministic case is similar.

Recall the language $\text{while}_{\mathbb{N}}^*$ defined in preliminaries. We consider a nondeterministic variant N-$\text{while}_{\mathbb{N}}^*$ obtained by allowing a choice operator, $program_1 \mid program_2$. We will show that every relation R_ε is definable from $\sigma(I)$ by a program in N-$\text{while}_{\mathbb{N}}^*$. By results in [8] (where $\text{while}_{\mathbb{N}}^*$ is denoted while^{++}), each relation not containing integers and definable in $\text{while}_{\mathbb{N}}^*$ is also definable in $\text{while}_{\mathbb{N}}$ so in FO^k for some fixed k depending on the program. This can be easily extended to the nondeterministic variants of the languages.

In order to prove the lemma, we need to represent AXML instances generated in the computation of an NQAXML program $\mathcal{Q}$ as relational structures, constructed from $\sigma(I)$ by the N-$\text{while}_{\mathbb{N}}^*$ program. The basic approach is to represent trees as

binary relations on the nodes. As we will see, the nodes themselves can be represented as tuples of bounded width containing labels and integers. This is needed because the domain is restricted to the initial data values in $\sigma(I)$, but we must also represent the new nodes created throughout the computation of $\mathcal{Q}$. In addition, we must represent the *eval* links between function calls and workspaces. To deal with this, we use the ability of N-while$_{\mathbb{N}}^*$ to insert integers in relations in the course of the computation. Their main use is as *timestamps* indicating the step in the computation when a node was created. New nodes are represented as follows. Recall that these are created by function calls or returns.

Function calls Consider a function call to $!g$ made at time t and the workspace constructed by its argument query. First note that the data variable bindings for each pattern are easily defined by an FO query using the descendant relation, definable in while$_{\mathbb{N}}^*$. The formula uses $\sigma(I)$ as well as the new portion of the instance, mimicking (‡).

Assume that the query has a single rule $Body \rightarrow Head$ (the multiple rule case is similar). We denote by $Head_0$ the tree obtained from $Head$ by removing the subtree rooted at the constructor node $\{a\}$. The nodes of the workspace constructed at time t using the bindings are represented as tuples with the following components

- each node n labeled b that does not lie under the constructor node $\{a\}$ is represented as (h, t) where h is a string representation of $Head_0$ where each symbol is one coordinate of the tuple and the position of n in the tree is marked. The inclusion of the entire tree h is useful for technical reasons, as it enables ordering sibling nodes with the same label.
- each expanded node n corresponding to binding β is represented as (h, t, β) where h is the string representation of the constructor tree, with the position of n in the tree marked.

The link from the call $?g$ at node n to its workspace is represented by updating the timestamp of node n to t, the same as that of the workspace.

Function returns Consider the simulation of a function return. The answer to the call is computed similarly to the above, with nodes labeled by the time of the return (details omitted). A subtlety is that the program must also enforce the reduction of sibling isomorphic trees that may have resulted from the return. This can be done by recursively defining bottom-up pairs of roots of isomorphic subtrees. Suppose that two isomorphic sibling subtrees rooted at n and m are detected. Note that one of these subtrees, say $tree(n)$, must contain the answer to the call, so it contains the current maximum timestamp. This allows distinguishing n and m and deleting one of them, say $tree(m)$.

Choice of function call or return The function to be called or returned is nondeterministically chosen in the program $\mathcal{Q}$. This can be simulated by a nondeterministic computation of the N-while$_{\mathbb{N}}^*$ program as follows. Recall that new function calls are never introduced in an expanded subtree. This means that all function calls can be reached by a depth-first traversal of the trees that ignores the expanded subtrees. Since the roots of all maximal trees have distinct timestamps, the tree to explore is chosen nondeterministically by the timestamp of the root. Once the tree is chosen, its non-expanded portion must be explored in a depth-first manner. To do this, note that

in the non-expanded portion of each tree, siblings have one of the two following properties:

(i) they have different timestamp, or
(ii) they have the same timestamp, but are in distinct positions in h.

This allows the depth-first traversal of the tree and the nondeterministic choice of a function call. The choice of a function return is done similarly.

Now consider the relation R_ε generated at time t_0. This is defined by an N-while$_\mathbb{N}^*$ program $w(t_0)$ using a loop that carries out the first t_0 transitions, then selects the subinstance with timestamp t_0 and extracts R_ε. Thus each R_ε is defined by $w(t)$ so is definable in FO^k for some k depending on $w(t_0)$. It is easily seen that k does not depend on the particular timestamp t_0 parameterizing the program, so all R_ε are definable in FO^k. $\square$

Remark 4.3 Observe that the structure $\sigma(I)$ is built using the patterns of $\mathcal{Q}$. The construction can be made less dependent on the specific $\mathcal{Q}$ by using a more general syntactic criterion such as the maximum number of nodes k and the set C of constants used in patterns of $\mathcal{Q}$. The structure $\sigma(I)$ can then be replaced with a structure $\sigma_{k,C}(I)$ depending only on k and C, consisting of one relation for each pattern of size up to k using constants in C. Of course, the number of relations in $\sigma_{k,C}(I)$ may be exponential in the number of relations in $\sigma(I)$.

As we will see in Theorem 4.8, Lemma 4.2 can be used to show a powerful normal form for QAXML programs. Informally, a program in the normal form first produces $\sigma(I), \equiv_{I,k,C}$ with a total order, and $Action(I, k, C)$, and then carries out the rest of the computation on the quotient structure of the above instance with respect to $\equiv_{I,k,C}$, in which the ordered equivalence classes of $\equiv_{I,k,C}$ are replaced by corresponding integers (represented as paths).

Using the above development, we show next that QAXML with isolated functions is equivalent to natural analogs of $while_\mathbb{N}$ to trees. We consider first NQAXML, then DQAXML.

4.2 While$_\mathbb{N}$ Languages for Trees

We define an analog of the language $while_\mathbb{N}$ for trees. We first define a nondeterministic variant, denoted N-while$_\mathbb{N}^{tree}$, then a deterministic one denoted while$_\mathbb{N}^{tree}$. The language N-while$_\mathbb{N}^{tree}$ uses integer variables $i, j, \ldots$ (initialized to zero) and forest variables $X, Y \ldots$ including two distinguished variables *In* and *Out*, for *input* and *output* respectively. In addition, it is equipped with one stack on which the content of forest variables can be pushed and popped. This stack is used primarily to build the result. The basic instructions are:

- increment/decrement i
- $X := \{T\}$, where X is a forest variable and T is a constant AXML tree with no functions
- $X := Q(Y)$, where X and Y are forest variables and Q a tree pattern query applied to Y

- $X := Y \cup Z$ where X, Y, Z are forest variables distinct from *In*
- $X := a[Y]$, where X, Y are forest variables distinct from *In*, $a \in \Sigma$ (this assigns to X the tree with root labeled a and all trees in Y as its children)
- *push*(X) (push the contents of forest variable $X \neq In$ on the stack)
- $X := top$ (assign to X the top of the stack and pop it).

A program may consist of a single instruction. More complex programs may be obtained using the following constructs:

- **while** $i > 0$ do *program*
- **while** $X \neq \emptyset$ do *program*
- *program*1 ; *program*2 (composition)
- *program*1 | *program*2 (nondeterministic choice)

A program also comes equipped with a DTD Δ constraining its input, provided in the initial instance by variable *In*. An output is the content of variable *Out* in a final instance (whenever the computation terminates). A program W computes a tree query $\mathcal{R}$ if for each input tree I satisfying Δ, the set of possible outputs of W is $\{J \mid \langle I, J \rangle \in \mathcal{R}\}$.

The deterministic variant of N-while$_{\mathbb{N}}^{tree}$, denoted while$_{\mathbb{N}}^{tree}$, is obtained by disallowing nondeterministic choice. We next provide an example of a while$_{\mathbb{N}}^{tree}$ program.

Example 4.4 A while$_{\mathbb{N}}^{tree}$ program computing the transitive closure of the graph is sketched in Fig. 11. We explain the notation. Besides *Input* and *Output*, the program uses variables *Old* (containing a tree rooted at O), *New* (containing a tree rooted at N and sometimes also a tree rooted at O), and *Difference* (containing a tree roted at N). Query QN initializes variable *New* to *Input* in which the root label *Graph* is changed to N. The query QOld copies the contents of *New*, relabeling the root to O. The query QNewEdges computes the *new* edges of the next iteration (those not present in the tree of *New* rooted at O), similarly to the query in Fig. 10. The new edges are placed in a tree rooted at N. Note that $New \cup Difference$ is a forest containing two trees rooted at N. The query QMergeN merges the two trees into a single tree rooted at N (by taking the union of the subtrees under the two roots). Finally the query QAnswer copies *New* while changing the label N back to *Graph* for the final answer.

Data: A tree representing a graph stored in *Input*
Result: A tree representing the transitive closure of the graph stored in *Output*
begin
    $New := QN(Input)$
    $Difference := New$
    while Difference $!= \emptyset$ **do**
        $Old := QOld(New)$
        $New := New \cup Old$
        $Difference := QNewEdges(New)$
        $New := QMergeN(New \cup Difference)$
    $Output := QAnswer(New)$
end

Fig. 11 while$_{\mathbb{N}}^{tree}$ program for transitive closure

4.3 NQAXML with Isolated Functions

We now return to NQAXML and show the following main result.

Theorem 4.5 *NQAXML programs with isolated functions express the same set of tree queries as N-while$_{\mathbb{N}}^{tree}$.*

Proof We begin with the simulation of N-while$_{\mathbb{N}}^{tree}$ by NQAXML with isolated functions. □

Lemma 4.6 *Every tree query expressible by an N-while$_{\mathbb{N}}^{tree}$ program can also be expressed by an NQAXML program with isolated functions.*

Proof Let W be an N-while$_{\mathbb{N}}^{tree}$ program with input DTD Δ with root label r, defining a tree query $\mathcal{R}$. We outline its simulation by an NQAXML program P with isolated functions.

The program W uses forest variables $X, Y, \ldots$ and integer variables $i, j, \ldots$. Recall that the input to the program is the initial value of the variable In, which is a tree satisfying Δ. The DTD Δ_r for the initial instance of the NQAXML program P is Δ modified to require a child labeled $!f$ under the root (labeled r).

The initial call to $!f$ returns several function calls that will be needed in the simulation and that we describe as we go along.

There are several main components of the simulation:

- representing the contents of variables (forests and integers)
- simulating individual instructions modifying the variables
- simulating the control

To generate and represent the content of variable X in NQAXML, we use a tree with a special root r_X containing initially a function call $!X$. The content of X is represented by a forest of subtrees under r_X. To distinguish the current contents from previous ones, we mark the current subtrees by a function $!current$ (that vanishes when a new assignment is simulated). Some control (to be explained further) determines which instruction is simulated. We consider next the instructions in turn.

An assignment $X := Q(Y)$ is simulated by activating $!X$. The control determines that $!X$ is replaced by $!X_Q$ a function that simulates this particular instruction. The function $!X_Q$ applies Q to the current forest of r_Y. This generates a workspace with root a_X containing $Q(Y)$. The return query of $!X_Q$ simply copies the content of the workspace. This is easly done by turning the heads of rules of Q into bodies of rules for the return query.

Now consider the assignments $X := Y \cup Z$ and $X := a[Y]$. These are trickier because of the difficulty of "copying" a tree from X to Y in NQAXML (recall that pattern queries extract data values but cannot copy subtrees). Note also that the program W could, for instance, construct trees of unbounded depth bottom-up using these operators (and the stack), whereas P must construct trees top-down by repeated function calls. To deal with this, we will need to adorn all trees produced in the

simulation with additional functions used to facilitate copying in a manner similar to pebbles placed on nodes.

Consider a computation of a NQAXML program. Recall that data values are introduced in trees using pattern queries, by instantiating subtrees in the head rooted at constructor nodes with bindings for the data variables. As before, let us call nodes obtained by such instantiations *expanded* nodes, and the others *unexpanded*. One can easily modify the NQAXML program so that each unexpanded node has as child a node labeled by a particular function !*mark* that will be useful for copying trees in the simulation. Expanded nodes cannot be marked because functions cannot occur under constructor nodes.

Consider a tree T marked as above, whose root is not an expanded node. Suppose we wish to create a copy of a tree T starting with a call to a function !*copy*. We begin by marking the root r of T with a call to its child !*mark* and making a call to !*copy* that returns a tree with a root having the same label as r and as a child another call !*copy*. Next, subtrees of T are recursively copied by calls to !*copy*, following a nondeterministic depth-first traversal visiting the non-expanded nodes of T, making use of the marker provided by calls to !*mark*. Expanded subtrees occurring as children of non-expanded nodes are copied by pattern queries obtained by turning all heads of the pattern queries used in the original program into bodies that are copied in the head. Of course, the bodies have to be augmented with patterns limiting application of the query to subtrees of the current node in the depth-first traversal of T.

The instructions $X := Y \cup Z$ and $X := a[Y]$ can now be easily simulated using the above technique. For $X := Y \cup Z$, the trees in Y and Z are copied into X in arbitrary order. For $X := a[Y]$, a first call generates root a with child !*copy*, after which all trees of Y are copied under a.

Now consider the stack s of forests. We associate to s a function !s. The contents of s is represented by a chain of workspaces generated by calls to !s, each holding the corresponding forest. To simulate $push(X)$, a new call to !s is made, which generates the workspace corresponding to the new top of the stack. This contains another call to !s, as well as the contents of X, copied to the workspace by the same technique as above. To simulate $X := top$, the forest in the last workspace of the chain is copied in X and the call ?s in the previous workspace returns its answer !s (this pops the stack). Some straightforward bookkeeping is needed to prevent undesired multiple pops and pushes (details omitted).

Consider integer variables. To each integer variable i we associate a function !i. Increments and decrements are implemented similarly to the stack, by a chain of calls to !i. Thus, a call to !i produces a workspace containing another call !i. The content of i is represented by the number of active calls ?i in the instance. Increment is implemented by a call to !i in the last workspace, and decrement by returning an answer to the last call ?i. Again, some bookkeeping is needed to prevent undesired multiple increments and decrements.

We finally discuss the control. This is simulated using a function associated to each instruction of the program. The current instruction is identified by the presence of an active call to the corresponding function. At each point, at most one such active call is present, and guards are used for sequencing and to verify the loop conditions.

In addition, some bookkeeping is needed to signal the beginning and completion of the simulation of each instruction as described above.   □

The converse simulation is much more intricate and makes critical use of Lemma 4.2.

Lemma 4.7 *Each tree query expressible by a NQAXML program with isolated functions can also be expressed by an N-while$_{\mathbb{N}}^{tree}$ program.*

Proof The simulation of the NQAXML program $\mathcal{Q}$ consists of several stages:

  (i)   Compute from I a representation of the relational structure $\sigma(I)$;
  (ii) For the k provided by Lemma 4.2, compute from $\sigma(I)$ the ordered set of equivalence classes $\equiv_{I,k,C}$, and the instance $Action(I, k, C)$ defined in Section 2, where C is the set of data values mentioned in $\mathcal{Q}$;
  (iii) Compute a Turing Machine tape representation of $\sigma(I)$ and $Action(I, k, C)$, in which each class of $\equiv_{I,k,C}$ is represented by the corresponding integer;
  (iv) Simulate the Turing machine computing the answers to $\mathcal{Q}$ given as input the above tape; and
  (v)   For each terminating computation, produce in variable *Out* the output tree encoded on the tape.

We briefly outline each of the above stages. In the simulation, relational structures are represented in a standard way by bounded trees whose leaves are data values. The computation of $\sigma(I)$ is done by applying to I the pattern defining each relation in σ and assigning the result to a corresponding forest variable. For (ii), recall that $\equiv_{I,k,C}$ and $Action(I, k, C)$ can be computed from $\sigma(I)$ by a *fixpoint* query, so by *while* and N-while$_{\mathbb{N}}^{tree}$. Consider (iii). Recall that the equivalence classes $\equiv_{I,k,C}$ are totally ordered, so each class can be identified with an integer. Moreover, each relation in $\sigma(I)$ contains a set of such equivalence classes, and $Action(I, k, C)$ consists of relations on the equivalence classes. Thus, these can be represented on the tape using the integers corresponding to the classes. The sequence of symbols on the resulting tape encoding can be read as an integer (whose base is the number of tape alphabet symbols), which is easy to generate using an N-while$_{\mathbb{N}}^{tree}$ program (in fact even by a *while*$_{\mathbb{N}}$ program). Each terminating computation of the nondeterministic Turing machine simulating $\mathcal{Q}$ produces a final tape encoding an output tree, whose corresponding integer can be obtained again by an N-while$_{\mathbb{N}}^{tree}$ program, because N-while$_{\mathbb{N}}^{tree}$ is computationally complete on integers. The output tree can be represented on the final tape in a standard way as a string, except for nodes of expanded trees, because individual data values are not available to the Turing machine. Instead, consider an expanded subtree R_ε obtained by a query rule $Body \rightarrow Head$ with constructor subtree T using a set of bindings $\mathcal{B}$ for the variables in T. By Lemma 4.2, R_ε consists of a set $\{i_1, \ldots, i_m\}$ of equivalence classes of $\equiv_{I,k,C}$ (the i_j are the corresponding integers) and can be represented by $string(T)\{i_1, \ldots, i_m\}$. Finally, consider (v). The output tree is produced from the tape representation using the instructions $X := Y \cup Z$ and $X := a[Y]$, together with the stack. More precisely, subtrees of the output are generated in the order of a depth-first traversal of the tree

on the tape. Whenever a subtree is generated it is pushed on the stack. When all sibling subtrees sitting under a node labeled a are generated, they are popped from the stack one-by-one and accumulated in a forest variable X, and the subtree rooted at a is obtained by an instruction $X := a[X]$. Non-expanded leaves are easily generated using some constant trees, one for each tag. Expanded subtrees represented as $string(T)\{i_1, \ldots, i_m\}$ are generated as follows. The equivalence classes corresponding to $\{i_1, \ldots, i_m\}$ are first collected in a relation B using the order on $\equiv_{I,k,C}$. The expanded subtrees are obtained by applying to B a query $Body \rightarrow Head$ whose data variables in $Body$ bind to all tuples in B and whose head is T (with the root as constructor node). $\qquad\square$

This completes the proof of Theorem 4.5.

The two-way simulations above yield a powerful normal form. We use the notation of Section 4.1.

Theorem 4.8 *For each NQAXML program $\mathcal{Q}$ with isolated functions there is an equivalent program $\mathcal{Q}_{nf}$ effectively obtained from $\mathcal{Q}$, whose computation on input I consists of the following three phases:*

1. *a* PTIME *computation producing a standard tree representation of the relational structure $\sigma(I)$, $\equiv_{I,k,C}$ with a total order, and $Action(I, k, C)$;*
2. *an arbitrary computation on a representation of the quotient structure of the above instance with respect to $\equiv_{I,k,C}$, in which the ordered equivalence classes of $\equiv_{I,k,C}$ are replaced by their ranks;*
3. *a* PTIME *computation (in the size of the output) producing the result.*

In particular, note that (1) reduces in PTIME the computation to one without data values, (2) is a computation with no data values, and (3) produces in PTIME the final result with its data values. The ranks of equivalence classes in the quotient structure are represented by chains of function calls.

Remark 4.9 Observe that the index of $\equiv_{I,k,C}$, so the size of the input to phase (2), may be arbitrarily smaller than the input I. In fact, as shown in [3], for inputs that are standard tree representations of relations, there is a constant $M > 0$ so that the expected index of $\equiv_{I,k,C}$ (under uniform distribution) is asymptotically bounded by M. This suggests a potential opportunity for optimization, using the compressed representation provided by the quotient structure. The analysis is harder if the input is not a representation of a relation. Note also that $\sigma(I)$ may be arbitrarily smaller than I (for example, I may consist of a very deep tree with a single data value, and $\sigma(I)$ may use only that data value). Thus, in the best case, a double compression takes place: first from I to $\sigma(I)$, and then from $\sigma(I)$ to the quotient structure.

The following is now immediate.

Corollary 4.10 *The normal form of Theorem 4.8 also applies to N-while$_{\mathbb{N}}^{tree}$ programs. Additionally, phases (1) and (3) can be expressed by while$_{\mathbb{N}}^{tree}$ programs (i.e. without nondeterministic instruction choice).*

Remark 4.11 One might wonder if it is possible to relax the definition of QAXML with isolated functions while preserving Lemma 4.2 and Theorems 4.5 and 4.8. This can be done to a limited extent. For example one can show that the results continue to hold if we allow functions to be placed under tags that may occur only *once* in every valid input. Indeed, these can be simulated by NQAXML programs with isolated functions. Going further is non-trivial. To illustrate this, we note that one cannot even allow functions under tags that may appear *twice* in valid trees without losing the above results. Indeed, consider the DTD Δ

$$r \;\rightarrow\; a\,a, \qquad a \;\rightarrow\; |\mathbf{dom}| \geq 0$$

Suppose functions are allowed under a. One can write a NQAXML program which, on a given input, outputs nondeterministically one of the sets of data values under the a's. It is easy to see, by genericity, that there is no N-while$_{\mathbb{N}}^{tree}$ program computing this query. The problem can be circumvented in various ways, for instance by bounding the number of data values allowed under a. In fact, it remains open to characterize where functions can be placed so that Lemma 4.2 and equivalence to N-while$_{\mathbb{N}}^{tree}$ still hold.

4.4 DQAXML with Isolated Functions

We now consider deterministic QAXML with isolated functions. As we will see, much of the previous development transfers to this case.

Recall that while$_{\mathbb{N}}^{tree}$ denotes the language N-while$_{\mathbb{N}}^{tree}$ without the nondeterministic instruction choice construct. Thus, while$_{\mathbb{N}}^{tree}$ expresses a subset of the queries defined by N-while$_{\mathbb{N}}^{tree}$. For a language expressing both deterministic and nondeterministic queries, let us call the set of deterministic queries it expresses its *deterministic fragment*. It will be useful to note the following.

Theorem 4.12 *The language* while$_{\mathbb{N}}^{tree}$ *expresses precisely the deterministic fragment of* N-while$_{\mathbb{N}}^{tree}$.

Proof By definition, while$_{\mathbb{N}}^{tree}$ is included in the deterministic fragment of N-while$_{\mathbb{N}}^{tree}$. Conversely, consider an N-while$_{\mathbb{N}}^{tree}$ program W defining a deterministic query. We use the normal form provided by Corollary 4.10. The only nondeterministic portion of the normal form is Phase (2), consisting of the simulation of a nondeterministic Turing machine producing an encoding of the output. However, since the output is the same for all computations, the Turing machine can be determinized. Since while$_{\mathbb{N}}^{tree}$ is computationally complete on integers, Phase (2) can be computed by a while$_{\mathbb{N}}^{tree}$ program. Thus, W can be expressed by a while$_{\mathbb{N}}^{tree}$ program. $\square$

We now show the analog of Theorem 4.5.

Theorem 4.13 *DQAXML programs with isolated functions express the same set of tree queries as* while$_{\mathbb{N}}^{tree}$.

Proof The proof of Theorem 4.5 largely transfers to the deterministic case. Consider first a while$_{\mathbb{N}}^{tree}$ program. Theorem 4.12 in conjunction with Corollary 4.10 show that the normal form applies to while$_{\mathbb{N}}^{tree}$. Phase (1) can be easily expressed by a DQAXML program with isolated functions. Recall that Phase (2) consists of simulating a deterministic Turing machine computation producing an encoding of the output tree. This can also be carried out by a DQAXML program with isolated functions, by simulating integers similarly to the proof of Lemma 4.6. Finally, Phase (3) consists of building the output tree from its tape representation. In while$_{\mathbb{N}}^{tree}$, this is done using the stack and the instructions $X := Y \cup Z$ and $X := a[Y]$. However, QAXML can circumvent this and directly construct the output tree from the tape representation. This is done top-down, by mimicking a depth-first traversal of the tree on the tape and generating the corresponding nodes. The expanded subtrees in the output are generated as in the proof of Lemma 4.7. More precisely, consider an expanded forest represented on the tape as $string(T)\{i_1, \ldots, i_m\}$. To generate the forest, the equivalence classes corresponding to $\{i_1, \ldots, i_m\}$ are first collected in a relation B using the order on $\equiv_{I,k,C}$ produced in Phase (1). The expanded forest is obtained by applying to B a query $Body \rightarrow Head$ whose data variables in $Body$ bind to all tuples in B and whose head is T (with the root as constructor node).

Conversely, consider a DQAXML program with isolated functions. The simulation by while$_{\mathbb{N}}^{tree}$ is similar to that in the proof of Lemma 4.7, making once again crucial use of Lemma 4.2. □

As a consequence of Theorems 4.5, 4.12 and 4.13, we have the following nontrivial result.

Theorem 4.14 *DQAXML with isolated functions expresses precisely the deterministic fragment of NQAXML with isolated functions.*

Finally, the same normal forms hold for DQAXML with isolated functions and for while$_{\mathbb{N}}^{tree}$ as for their nondeterministic counterparts.

4.5 Boolean Queries

We consider here Boolean queries, for which some of the earlier results can be strengthened. In particular, constructing the answer is trivial for such queries. As we will see, this renders redundant some instructions and the stack in the *while* languages.

Consider an NQAXML program $\mathcal{Q}$. We say that $\mathcal{Q}$ is *Boolean* if whenever it terminates, it produces as output a tree consisting of a single node labeled *accept* or *reject*. A computation is accepting if it terminates with output *accept*. An input I is accepted by $\mathcal{Q}$ if $\mathcal{Q}$ has at least one accepting computation on I. Boolean N-while$_{\mathbb{N}}^{tree}$ programs are defined analogously. The definitions for Boolean *deterministic* QAXML and while$_{\mathbb{N}}^{tree}$ programs are similar. We say that two Boolean programs are equivalent (or define the same property) if they have the same input DTD and accept the same set of instances.

For Boolean queries, we are able to obtain a stronger version of Theorem 4.5.

Theorem 4.15 *The following languages express the same Boolean tree queries:*

(i)   NQAXML and DQAXML with isolated functions;
(ii) N-while$_\mathbb{N}^{tree}$ and while$_\mathbb{N}^{tree}$ with or without the stack and instructions of the form $X := Y \cup Z$, $X := a[Y]$;

Proof The equivalence of Boolean NQAXML and DQAXML follows from Theorem 4.14 and the following fact:

(†)   The set of Boolean queries expressed by NQAXML programs with isolated functions equals the set of Boolean queries in its deterministic fragment.

The proof of (†) relies once again on the normal form for NQAXML with isolated functions. In the normal form, the nondeterministic portion of the computation is simulated by a Turing machine, so acceptance can be determinized.

The equivalence of N-while$_\mathbb{N}^{tree}$ and while$_\mathbb{N}^{tree}$ is similar. Note from the proof of the normal form that the stack and instructions $X := Y \cup Z$, $X := a[Y]$ are only used to construct the output tree from the tape. For Boolean queries, the output can be constructed without these instructions or the stack, so these are redundant. Finally, the equivalence of DQAXML and while$_\mathbb{N}^{tree}$ follows from Theorem 4.13.   □

We additionally obtain the following stronger normal form for Boolean programs.

Corollary 4.16 *For each Boolean (non)deterministic QAXML program $\mathcal{Q}$ with isolated functions there is a (non)de-terministic Boolean QAXML program $\mathcal{Q}_{nf}$ with isolated functions, effectively computable from $\mathcal{Q}$, that defines the same property, whose computation consists of the following phases:*

1. *a* PTIME *computation (in the size of the input);*
2. *an arbitrary computation on an instance with no data values.*

The normal form shows that data values can be eliminated by a pre-processing phase in PTIME, regardless of the overall complexity of the property. The same normal form holds for Boolean (N)-while$_\mathbb{N}^{tree}$ programs, with the addition that no stack or instructions $X := Y \cup Z$, $X := a[Y]$ are used in the normal form.

Expressiveness of QAXML with isolated functions The above development points to limitations in the expressive power of QAXML with isolated functions that are reminiscent of limitations of $while_\mathbb{N}$ in the relational context. In particular, the 0/1 law for properties definable by $while_\mathbb{N}$ is inherited from the relational context, for inputs consisting of trees encoding relations. More precisely, consider an m-ary relation R and its standard tree representation described by the following DTD Δ_R:

$$R \quad \rightarrow \quad |tup| \geq 0$$
$$tup \rightarrow \quad |A_1| = 1 \,\wedge\ldots\wedge\, |A_m| = 1$$
$$A_i \quad \rightarrow \quad |dom| = 1, \quad 1 \leq i \leq m$$

It is easily seen that (non)deterministic QAXML with isolated functions, input DTD Δ_R, and no constant data values, has a 0-1 law. It would be interesting to characterize the class of input DTDs for which the 0-1 law continues to hold.

The 0-1 law for relational inputs shows that there are simple properties that cannot be expressed in QAXML with isolated functions, e.g., evenness of the number of data values in inputs over Δ_R. This is despite the fact that QAXML with isolated functions is *computationally* complete, since it can simulate arbitrary computations on integers. A reason for this limitation is the strict separation between data and computation, imposed by the isolation condition. We next show that this can be largely overcome by closer integration of the two, provided by embedded functions.

5 QAXML with Dense Functions

We now consider QAXML that can have embedded functions throughout the input. Intuitively, we would expect this to lead to completeness, alleviating the limitations of isolated functions. This turns out to be true for nondeterministic semantics, but false in the deterministic case. This is due to a variant of the "copy elimination problem".

Definition 5.1 *A QAXML program with dense functions is a pair $\mathcal{Q} = (\Phi, \Delta)$ where Φ is a set of function definitions and Δ a static DTD. For an instance I satisfying Δ, we denote by $I^{!*}$ the instance obtained by adding a call $!f$ under every node of I whose label is a tag. The program $\mathcal{Q}$ expresses a tree query $\mathcal{R}$ with input DTD Δ if for every I satisfying Δ, $(I, O) \in \mathcal{R}$ iff there exists a computation of $\mathcal{Q}$ on $I^{!*}$ terminating with O as the unique subtree of a unique node labeled* Out.*

In other words, a QAXML program with dense functions is one that has in the initial instance a function call $!f$ as a child of each tag.

Nondeterministic semantics The main result on NQAXML with dense functions is the following.

Theorem 5.2 *NQAXML with dense functions is query complete.*

Proof Let $\mathcal{R}$ be a tree query with input DTD Δ. Let I be an input to $\mathcal{R}$. We begin with a pre-processing stage. Recall that programs with dense functions adorn their inputs with a function call under each tag. However, individual data values are not marked with function calls in the same manner. As a preliminary step, we create a tree I^* that is identical to I, except that each data value d is replaced by a subtree $e[d \, !f]$ where e is a new tag. Such a tree can be generated by mimicking nondeterministically a depth-first traversal of I. Whenever a tag is encountered, a node labeled with the same tag is generated in I^* (with its accompanying function call). Data values are treated differently: all sibling data values are collected using a function call, which returns the desired set of trees $e[d! \, f]$ for each d. Thus, each occurrence of a data value now has associated to it a unique function call.

With I^* constructed, the computation of $\mathcal{R}$ has several phases:

  (i)   construct an ordering $\leq$ of the data values in I;
 (ii)   compute an encoding $enc_{\leq}(I)$ of I on a Turing machine input tape;
(iii)   simulate the Turing machine computing $\mathcal{R}$;

(iv)   if the Turing machine terminates, construct the tree J whose encoding $enc_{\leq}(J)$ is on the final tape of the machine.

We briefly outline steps (i)-(iv) above. An ordering $\leq$ can be constructed nondeterministically by marking in arbitrary order the data values using their associated function calls, and adding each newly detected value to the order. For (ii), we consider an encoding $enc_{\leq}(I^*)$ representing I^* in a standard string notation, where each data value is replaced by the integer corresponding to its rank w.r.t. $\leq$ (for simplicity integers are represented in unary). The encoding is produced, once again, by a nondeterministic depth-first traversal of I^* (consulting $\leq$ whenever a data value is encountered). The Turing machine tape is represented by a path where each node represents a tape cell and has associated to it its content. In addition, each such node is adorned by function calls used as markers, which allows simulating the Turing machine computation. The nondeterministic control is easily simulated by a nondeterministic choice of function calls whose guards test the conditions associated to each move. The output of the Turing machine, if it exists, consists of $enc_{\leq}(J)$ where J is the output of $\mathcal{R}$ on I. The tree J can then be generated from the tape in some depth-first order, replacing each integer by its corresponding data value given by $\leq$.   □

Deterministic semantics We now consider DQAXML with dense functions. Recall that in the case of isolated functions, DQAXML was as expressive as the deterministic fragment of NQAXML. Interestingly, this turns out not to be the case with dense functions, as shown next.

Theorem 5.3 *DQAXML with dense functions is not complete.*

Proof Recall the standard DTD Δ_R associated with a relation schema R. Let R be a unary relation schema and consider the query $\mathcal{R}$ whose input DTD is Δ_R. Thus, its input is essentially a set of n data values. The output consists of a tree rooted at r, with $n!$ subtrees, each representing a successor relation among the n data values. We claim that there is no DQAXML program with dense functions that computes $\mathcal{R}$. The proof relies on a structural property involving the automorphisms of instances produced in the computation of any DQAXML program on input I. The property shows that any program computing $\mathcal{R}$ must produce more than one copy of the answer.

Let $\mathcal{Q}$ be a DQAXML program with dense functions, running on the inputs of $\mathcal{R}$. Let I be an input with a set D of n data values (where n can be taken to be as large as needed). Consider an instance E obtained in the course of the computation of $\mathcal{Q}$ on I. Recall that E consists of a forest of which one tree is rooted at R and the others are workspaces of active function calls. Let $tree(E)$ be the tree obtained from E by adding all edges connecting nodes labeled by active function calls to the roots of their corresponding workspaces. For each node u, we denote by $tree(u)$ the subtree of $tree(E)$ rooted at u. For nodes u, v, where one is an ancestor of the other in $tree(E)$, we denote by $\delta(u, v)$ the distance between u and v. We show the following:

(†)   There exists a mapping σ from the nodes of $tree(E)$ to subsets of D such that:

1. $\sigma(root(tree(E))) = \emptyset$;
2. if u is an ancestor of v then $\sigma(u) \subseteq \sigma(v)$;
3. for each node u and permutation π of D fixing $\sigma(u)$, there is an extension $\bar{\pi}$ of π to an automorphism of $tree(u)$ that commutes with σ (so $\sigma(\bar{\pi}(v)) = \bar{\pi}(\sigma(v))$ for every v);
4. for all nodes u, v such that u is the parent of v in $tree(E)$, $|\sigma(v) - \sigma(u)| \leq k$, where k is the maximum number of variables in a query or pattern of $\mathcal{Q}$.

The proof of (†) is by induction on the number of steps leading to E. The intuition is the following. First, all permutations of D are automorphisms of the input, so by genericity every instance obtained throughout the computation has the same property. However, subtrees generated by function calls and returns may be dependent on a fixed set of data values, because heads of queries may attach new function calls to each binding of its variables. The mapping σ associates to each node u the set of data values on which $tree(u)$ depends.

We outline the induction. The basis is obvious. Suppose (†) holds in a reachable instance E and E' is obtained from E in one step. In the transition, all functions whose call guards are true are activated, and those whose return guards hold return their answer. Consider a function call $!h$ at a node u. We extend σ to the workspace created by the call as follows: for nodes v that are not expanded nodes, $\sigma(v) = \sigma(u)$. For an expanded node v obtained with binding β, $\sigma(v)$ is $\sigma(u)$ together with the data values in β (at most k). It is clear that (2)–(3) hold for the nodes in the workspace (and all workspaces generated in parallel by function calls), and (4) holds for the entire E'. Now consider call returns. Consider the answer returned by a call at node u. Similarly to the above, σ is extended to the nodes in the answer, by augmenting $\sigma(u)$ for expanded nodes with the data values in the corresponding bindings. Again, (2)–(3) hold for the new nodes, and (4) continues to hold for the entire E'. Finally, it can be easily shown that (3) continues to hold for nodes w of E' that already exists in E, because of the induction hypothesis and the fact that bindings of patterns applied to E compose with automorphisms of E. This establishes (†).

Now suppose that $\mathcal{Q}$ produces at some point in its computation an instance E containing a subtree rooted at node u labeled Out, having the output of $\mathcal{R}$ as unique subtree. Consider $\sigma(u)$. Suppose first that $\sigma(u) \neq \emptyset$. By (1,2,4), and assuming that $n > k$, there exists an ancestor w of u such that $\emptyset \neq \sigma(w) \neq D$. Consider a permutation π of D such that $\pi(\sigma(w)) \neq \sigma(w)$. Since $\sigma(root(tree(E))) = \emptyset$, and by (3), π can be extended to an automorphism $\bar{\pi}$ of $tree(E)$ that commutes with σ. It follows that $\bar{\pi}(w) \neq w$, so $\bar{\pi}(u) \neq u$. Thus, E contains at least two distinct subtrees rooted at Out. Now suppose that $\sigma(u) = \emptyset$. Recall that u has a child labeled r, under which sit the $n!$ trees representing the successor relations corresponding to the permutations of D. Consider the root w of such a tree. Note that the distance between u and w is 2. Thus, by (4), $|\sigma(w)| \leq 2 \cdot k$. Pick $a \in D - \sigma(w)$ (assuming without loss of generality that D is large enough) and consider a permutation π of D that fixes $\sigma(w)$ for which $\pi(a) \neq a$. By (3), π can be extended to an automorphism $\bar{\pi}$ of $tree(w)$. This is a contradiction, since the successor relation represented by $tree(w)$ is rigid (its only automorphism is the identity).

From the above it follows that no instance E computed by $\mathcal{Q}$ from I can contain a single output tree of the desired form. Thus, $\mathcal{R}$ cannot be computed by any DQAXML program with dense functions.   □

Note that the counterexample in the proof of Theorem 5.3 uses bounded-depth inputs and outputs. Thus, DQAXML with dense functions is not complete even in this case. However, it is complete for inputs and outputs encoding relations. Recall that Δ_R denotes the standard DTD corresponding to a relation schema R.

Theorem 5.4 *Let R and S be relation schemas and $\mathcal{R}$ be a deterministic tree query with input DTD Δ_R, such that every output satisfies Δ_S. Then there exists a DQAXML program with dense functions that expresses $\mathcal{R}$.*

Proof The computation of $\mathcal{R}$ on input I over Δ_R proceeds in stages similar to those in the proof of Theorem 5.2, but rendered more subtle by the lack of nondeterminism. In particular, *all* orderings of the data values in I must be constructed, and the Turing machine is simulated in parallel for all orderings.

In more detail, the computation of $\mathcal{R}$ on input I over Δ_R proceeds as follows:

(i)   construct *all* orderings of the set of data values in I;
(ii) for each ordering $\leq$ compute an encoding $enc_\leq(I)$ of I as a Turing machine input tape;
(iii) simulate the Turing machine computing $\mathcal{R}$, in parallel on all encodings;
(iv) if the Turing machine terminates, construct for each $\leq$ a subtree containing an isomorphic copy of the tree J whose encoding $enc_\leq(J)$ is on the final tape of the Turing machine corresponding to $\leq$;
(v)   Construct a single tree rooted at *Out* that contains J as unique subtree.

The construction of the orderings in (i) proceeds as follows. Suppose I has arity k and contains n data values. The construction is initiated by calling a function $!f$ whose body collects all data values using variable X and whose head is $\{a\}[X, !f]$. The function returns the same forest, producing n subtrees, each rooted at a and holding one data value (together with $!f$). After j iterations of parallel calls to all $!f$, we have constructed the prefixes of length j of all orderings of the n data values. In the next step, a call to $!f$ collects all data values X not yet in the prefix held by the path to $!f$, and returns as before a subtree $a[X, !f]$ for each such X. After n steps, all orderings of n data values have been constructed. Note that each path from root to leaf represents one ordering.

The computation of the encoding $enc_\leq(I)$ proceeds as follows. First, a copy of I is created by a function call attached to $\leq$. The encoding $enc_\leq(I)$ consists of the k-tuples of integers obtained from the tuples of I, by replacing each data value with its rank with respect to $\leq$. This is represented by a path where each node represents a tape cell and has associated to it its content (for simplicity integers are represented in unary). In addition, each such node is adorned by function calls used as markers of the cells, which allows simulating the Turing machine computation on this input. For each ordering $\leq$, the output of the Turing machine, if it exists, consists of $enc_\leq(J)$. A copy of J can then be generated by decoding it from the tape, which involves

replacing each integer by its corresponding data value given by $\leq$. Finally, in step (v), a new, unique copy of J under root *Out* is obtained using a query that collects the tuples from all copies of J.   □

The above proof relies crucially on the fact that the input and output of $\mathcal{R}$ are trees representing relations and thus have highly regular structure. In particular, constructing a single copy of the output is easily done in this case, but is impossible for arbitrary outputs. This follows from the proof of Theorem 5.3, since the query $\mathcal{R}$ shown not to be expressible has relational input but nonrelational (although bounded-depth) output. In this case, one can compute multiple copies of the answer, but a single final copy cannot be obtained. This is a technical problem similar to the well-known copy elimination problem arising in some relational and object-oriented query languages [5]. One can show the following.

Corollary 5.5 *Let R be a relation schema. For each deterministic tree query $\mathcal{R}$ with input DTD Δ_R, there exists a DQAXML program $\mathcal{Q}$ with dense functions and input DTD Δ_R which, for every input I of $\mathcal{R}$, produces an instance containing a set of subtrees with root Out, each containing a unique subtree isomorphic to the output of $\mathcal{R}$ on I.*

Thus, for relational input, DQAXML with dense functions is complete up to copy elimination.

Since for Boolean queries the output is relational, we have the following.

Corollary 5.6 *Let R be a relation schema. Every Boolean tree query with input DTD Δ_R is expressed by some DQAXML program with dense functions.*
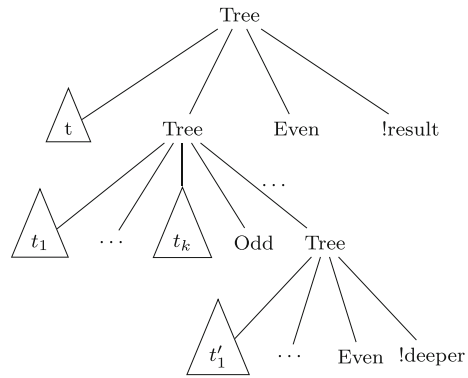
It remains open to give a precise characterization of the input and output DTDs for which DQAXML is complete as in Theorem 5.4 and Corollary 5.6.

Remark 5.7 Recall that the QAXML languages with isolated functions have natural counterparts in the *while* family of languages. As we will see in the next section, this also holds for QAXML with tree variables. We know of no *while* counterpart for the QAXML languages with dense functions and no tree variables.

6 QAXML with Tree Variables

In the previous sections, we considered the impact of embedding functions into data, where the queries used by functions extract bindings of data values. In particular, we showed that there are drastic differences in expressiveness between the isolated and dense cases. We now consider QAXML with more powerful queries equipped with *tree variables*, that can extract and compare entire subtrees from the input. We show how the picture changes in this case due to the increased power of the basic queries. First, programs with isolated functions are much more powerful. Indeed,

Fig. 12 Intermediate instance
in the computation of the
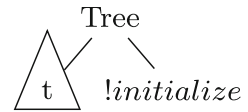QAXML$^\tau$ program *Parity*

in the deterministic case they become complete. In the nondeterministic case, the language is *not* complete, but remains so for a restricted kind of nondeterminism, occurring at the *control* level but not at the *data* level. With dense functions, this restriction can be lifted. In fact, only an intermediate form of density is needed for nondeterministic completeness, allowing functions to occur under constructor nodes of queries, but not embedded in the input.

We begin by defining QAXML with tree variables. We outline the differences with the model described in Section 3. We no longer distinguish in patterns between structural and data variables. Instead, each variable may bind to any node in the input tree. However, we introduce two types of equality: *shallow* equality $X = Y$ where X is a variable and Y is a variable, tag, function symbol, or data value, and *deep* equality $X =_d Y$, where X and Y are variables. The semantics is standard. Variables in heads of queries return an isomorphic copy of the entire *subtree* rooted at the node to which they bind. Relative patterns and queries are defined as before, by allowing equalities of the form $X = self$. We denote QAXML with tree variables by QAXML$^\tau$. The notion of isolated and dense program remains unchanged. We next provide an example of a QAXML$^\tau$ program.

Example 4.4 We exhibit a QAXML$^\tau$ program with isolated functions and tree variables computing the parity of the depth of the input tree (the depth is the maximum number of edges in a path from root to leaf). The root of the input tree is labeled *Tree*. The programs return a node with label *Even* if the depth of the input is even and *Odd* otherwise. The QAXML$^\tau$ program has isolated functions and computes the desired query with either deterministic or nondeterministic semantics. The main component of the QAXML$^\tau$ program is a function *deeper* that extracts, at each invocation, all subtrees whose roots are at a given depth in the input tree (the depth increases by one at each iteration). A parity flag is flipped at each invocation, and the function is called until no more subtrees are obtained. Figure 12 depicts an intermediate instance in the computation of the program.

In more detail, the initial instance is of the form shown in Fig. 13, with a function *!initialize* under the root. The computation starts with a call to *!initialize* that returns

Fig. 13 Initial instance for the QAXML$^\tau$ program *Parity*

a node labeled *Even* and two calls !*deeper* and !*result*. We call a subtree *proper* if its root is not labeled by a function symbol or a parity flag *Even* or *Odd*. The call guard of *deeper* ensures that the function is only called if the calling node has at least one proper sibling subtree. The input query of *deeper* is shown in Fig. 14. It copies the sibling parity flag *Even* or *Odd* and the proper siblings subtrees of the function call. The return query, shown in Fig. 15, returns under a root *Tree* all subtrees whose roots are at depth one in the copied subtrees, and flips the parity flag *Even* to *Odd* or conversely. The function *result* is called when *deeper* can no longer be activated, i.e. when the current call to *deeper* with no proper sibling subtree. The call to *result* returns a tree rooted at *Output* with one child labeled by the parity flag sibling to !*deeper*.   □

We now establish the expressive power of QAXML$^\tau$ with isolated functions and deterministic semantics, denoted DQAXML$^\tau$.

Theorem 6.2 *DQAXML$^\tau$ with isolated functions is query complete.*

Proof The high-level structure of the proof is similar to that of Theorem 5.2, but the lack of dense functions complicates the construction.

Let $\mathcal{R}$ be a deterministic tree query with input DTD Δ, computable by some Turing machine M. The simulation of $\mathcal{R}$ by a QAXML$^\tau$ program with isolated functions on input I consists of several phases:

$$Cond(V_0) = \begin{cases} & V_0 = \text{Tree} \\ \wedge & V_1 = \text{Tree} \\ \wedge & V_2 \neq \text{Even} \wedge V_2 \neq \text{Odd} \wedge V_2 \neq \text{!deeper} \wedge V_2 \neq \text{!result} \\ \wedge & V_3 = self \\ \wedge & V_4 = \text{Even} \end{cases}$$

$$Cond(V_0') = \begin{cases} & V_0' = \text{Tree} \\ \wedge & V_1' = \text{Tree} \\ \wedge & V_2' \neq \text{Even} \wedge V_2' \neq \text{Odd} \wedge V_2' \neq \text{!deeper} \wedge V_2' \neq \text{!result} \\ \wedge & V_3' = self \\ \wedge & V_4' = \text{Odd} \end{cases}$$

Fig. 14 The input query of *deeper*

$$
\begin{array}{ccc}
V_0 & \Rightarrow & \text{Tree} \\
| & & \diagup \; | \; \diagdown \\
V_1 & & \{V_3\} \quad Odd \quad \text{!deeper} \\
\diagup \; \diagdown \\
V_2 \quad V_4 \\
| \\
V_3
\end{array}
\qquad , \qquad
\begin{array}{ccc}
V_0' & \Rightarrow & \text{Tree} \\
| & & \diagup \; | \; \diagdown \\
V_1' & & \{V_3'\} \quad Even \quad \text{!deeper} \\
\diagup \; \diagdown \\
V_2' \quad V_4' \\
| \\
V_3'
\end{array}
$$

$$
Cond(V_0) = \left(V_4 = \text{Even} \right)
$$

$$
Cond(V_0') = \left(V_4' = \text{Odd} \right)
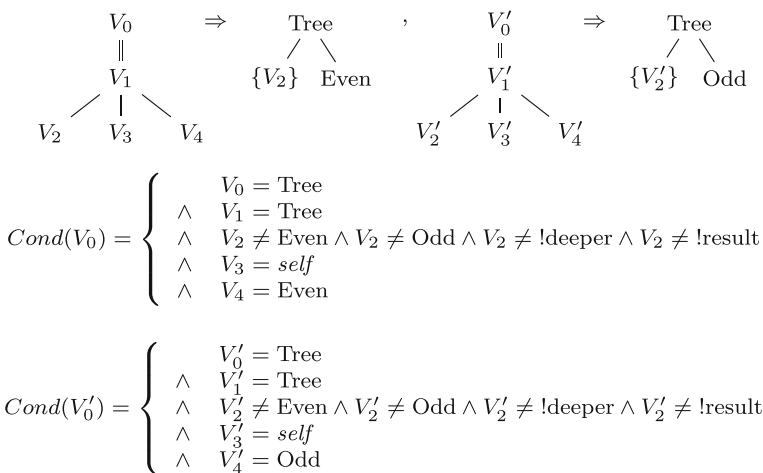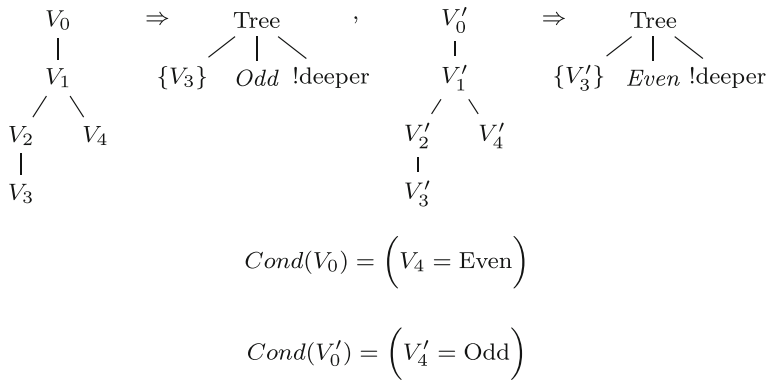$$

Fig. 15 The output query of *deeper*

(i)   construct simultaneously all orderings of the data values in I;

(ii) for each ordering $\leq$, construct a standard tape encoding $enc_{\leq}(I)$;

(iii) for each $\leq$, simulate the computation of M on $enc_{\leq}(I)$;

(iv) if M terminates, output J such that the final tape of M contains $enc_{\leq}(J)$; since $\mathcal{R}$ is deterministic, J independent of $\leq$. Produce one copy of J as the result.

We next elaborate on phases $(i) - -(iv)$. Consider (i). The simultaneous construction of all successor relations on data values is similar to (i) in the proof of Theorem 5.4, where dense functions were used. This is now feasible with isolated functions because entire subtrees defining partial successor relations can be copied simultaneously by a single query using tree variables. Suffixes of the successors are constructed recursively bottom-up. First, the set of data values is collected and each is placed under a node labeled a, representing all suffixes of length one. Suppose a forest of all suffixes of length n has been constructed, consisting of a path of nodes labeled a, each with a child holding a distinct data value. The suffixes are extended by one using a query whose body collects the bindings of two variables: S binding to a current suffix and X binding to a data value not occurring in S. The head of the query is $\{a\}[X, S]$ where a is a fixed tag. This extends each S with all choices of data values not yet included in S.

Consider (ii). For each ordering $\leq$ constructed above, we use the same string encoding $enc_{\leq}(I)$ as in Theorem 5.2. The construction can now be done simultaneously for all ordering by carrying the order as a parameter in a tree variable. The construction of the encoding has two phases. First, we inductively compute a total order $\prec$ on the set of subtrees of I (up to isomorphism) induced in some standard way by $\leq$ and a fixed order on tags. We then generate for each $\leq$ a path whose nodes are marked with the symbols of $enc_{\leq}(I)$. This is done by carrying out a depth-first traversal of I using $\prec$, and updating the encoding with an opening or closing tag every time a node is visited. In order to be able to backtrack, the ancestors of the current node in the traversal are marked until all their subtrees have been visited. Each modification of the marking involves reconstructing the tree bottom-up, which can be done due to the tree variables.

With $enc_{\leq}(I)$ constructed, the computation of M is easily simulated. Each move involves reconstructing the tape bottom-up and modifying markers corresponding to the state and position of the head. Extending the tape poses no problem. If M terminates, its final configuration is by definition $enc_{\leq}(J)$, where J is the unique answer of $\mathcal{R}$ on I (since $\mathcal{R}$ is deterministic). The tree J is constructed for each $\leq$ in a depth-first manner, using again markings to distinguish uncompleted trees. Each time a node is added, the entire tree is generated again bottom-up. It is straightforward to replace unary integers with their corresponding data values wrt $\leq$. Finally, the isomorphic versions of J obtained for each $\leq$ are copied under a new root labeled *Out* using a tree variable. Because isomorphic sibling trees are automatically reduced in our semantics, this results in a single copy of J, completing (iv). $\square$

We now consider QAXML^{τ} with nondeterministic semantics, denoted NQAXML^{τ}. It turns out that NQAXML^{τ} is *not* query complete. For example, it cannot express the query *Choice* that outputs one arbitrary data value from the input. Intuitively, this is because NQAXML^{τ} with isolated functions provides nondeterminism in the control, but not in choice of data. This can capture a limited form of nondeterminism that we call *weak nondeterminism*. For a tree T and automorphism π of T, we denote by π_d the restriction of π to the set of data values in T.

Definition 6.3 *A tree query $\mathcal{R}$ is weakly nondeterministic if for every input-output pair $\langle I, J \rangle$ of $\mathcal{R}$ and automorphism π of I, π_d can be extended to an automorphism of J.*

For example, the query *Choice* is *not* weakly nondeterministic. The query $Q_{a \vee b}$ that outputs either the set of data values under some tag a or the set of data values under tag b is weakly nondeterministic. Note that the input DTD is important: the same program may define a query that is weakly nondeterministic with respect to some input DTD, but not so with respect to another. The query $Q_{a \vee b}$ happens to be weakly nondeterministic for all input DTDs.

Theorem 6.4 *NQAXML^{τ} with isolated functions expresses precisely the weakly nondeterministic tree queries.*

Proof Let E be an instance obtained from input I in the course of the computation of a QAXML^{τ} program with isolated functions. An easy induction on the number of steps in the computation shows that for every automorphism π of I, π_d can be extended to an automorphism of E. Since the final instance contains a single tree rooted at *Out*, the output J also satisfies the property. Thus, every query expressible by QAXML^{τ} with isolated functions is weakly nondeterministic.

Conversely, let $\mathcal{R}$ be a weakly nondeterministic tree query computed by a Turing machine M. A NQAXML^{τ} program computing $\mathcal{R}$ is obtained similarly to (i)–(iv) in the proof of Theorem 6.2, with some modifications. The simulation now involves the following steps:

 (i)   construct simultaneously all orderings of the data values in I;

(ii)   for each ordering $\leq$, construct a standard tape encoding $enc_\leq(I)$;

(c)   select the set of $\mathcal{M}$ of orderings $\leq$ for which $enc_\leq(I)$ is minimum (as a string) among the encodings for all orderings and denote the minimum encoding by $enc_\mathcal{M}(I)$;

(d)   for all $\leq\ \in \mathcal{M}$, simulate the same computation of M on $enc_\mathcal{M}(I)$;

(e)   if M terminates, output, for each $\leq\ \in \mathcal{M}$, the tree J such that the final tape of M contains $enc_\leq(J)$; because $\mathcal{R}$ is weakly nondeterministic, all J's obtained for the $\leq\ \in \mathcal{M}$ are isomorphic. Produce one copy of J as the result.

Stages (i) and (ii) are carried out as before. Step (c) is straightforward, and (d) is done as before except that nondeterministic moves are simulated by having a different function for each move, of which one is activated nondeterministically at each transition. We justify the claim made in (e). Let $\leq_i\ \in \mathcal{M}$ and J_i be such that the final tape of M is $enc_{\leq_i}(J_i)$, $i = 1, 2$. We show that J_1 and J_2 are isomorphic. Let π_i be the isomorphism that replaces in I every data value by the integer representing its rank w.r.t. $\leq_i$. Since $enc_{\leq_1}(I) = enc_{\leq_2}(I)$, $\pi_1(I) = \pi_2(I)$ and $\pi_1 \circ \pi_2^{-1}$ is an isomorphism from J_1 to J_2. Thus, the outputs produced for all $\leq\ \in \mathcal{M}$ are isomorphic. To output a single copy of the output J, a final query collects all copies of J under a new root Out, using a tree variable. $\square$

To summarize the results in this section so far, QAXML$^\tau$ with isolated functions is complete for deterministic queries, but falls short for nondeterministic queries. It is clear that allowing dense functions leads to a complete language, as for QAXML. However, full density is not required. We say that a QAXML$^\tau$ program is *query-dense* if function calls can only occur under the root in the initial instance, but are allowed under constructor nodes in heads of queries. Thus, programs with query-dense functions are a hybrid allowing only isolated functions in the input but dense functions in queries. We have the following.

Theorem 6.5 *NQAXML$^\tau$ with query-dense functions is query complete.*

Finally, we note that Theorems 6.2 and 6.5 yield some strong normal forms for QAXML$^\tau$ programs.

Theorem 6.6 (*i*) *For every QAXML$^\tau$ program one can effectively construct an equivalent QAXML$^\tau$ program with isolated functions.* (*ii*) *For every NQAXML$^\tau$ program one can effectively construct an equivalent NQAXML$^\tau$ program with query-dense functions.*

While **with tree variables**

We next define simple variants of the *while* language that are equivalent to the (non)deterministic QAXML$^\tau$ languages. The deterministic language, denoted $while^\tau$, has forest variables $X, Y, Z, \ldots$, assignments $X := \varphi(Y)$ (where X a variable, Y is a variables or a constant tree, and φ is a tree pattern query with tree variables), and an iterator $while X \neq \emptyset\, do$. The nondeterministic version of the language, denoted $N - while^\tau$, is obtained by introducing control choice

*program*1|*program*2. As before, there are two distinguished variables, *In* and *Out* holding the input and output to the query.

Note that, unlike (N)-*while*$_{\mathbb{N}}^{tree}$, these languages have no integer variables, no stack, and no tree constructors, because all can be simulated using tree variables. We give an example of a *while*$^\tau$ program.

Example 6.7 A while$^\tau$ program computing the parity of the depth of the input tree (see Example 6.1) is sketched below.

> **Data**: A tree stored in *Input*
> **Result**: A node labeled by *Even* or *Odd*, stored in *Output*
> **begin**
>   *Parity* := *Even*
>   *Tree* := *Children*(*Input*)
>   **while** Tree! = ∅ **do**
>   *Parity* := *Flip*(*Parity*)
>   *Tree* := *Children*(*Tree*)
>
>   *Output* := *Parity*
> **end**

The query *Flip* changes the label *Even* to *Odd* and *Odd* to *Even*. The query *Children* returns all subtrees whose roots are at depth one in the forest to which it is applied.   □

The following establishes the connection between the $(N)-while^\tau$ and QAXML$^\tau$ languages. The proofs are similar to Theorems 6.2 and 6.4 and are omitted.

Theorem 6.8 (*i*) *while*$^\tau$ *is equivalent to DQAXML*$^\tau$ *with isolated functions and is query complete;* (*ii*) *N-while*$^\tau$ *is equivalent to NDQAXML*$^\tau$ *with isolated functions and expresses exactly the weakly nondeterministic tree queries.*

In order to obtain a complete nondeterministic language, $N - while^\tau$ has to be extended with a tree choice construct. To this end, we add an assignment $X := choose(Y)$, where X and Y are forest variables. This assigns to X one tree nondeterministically chosen from the forest in Y. We denote the language extended with this form of data nondeterminism by N^d-*while*$^\tau$. The following is immediate.

Theorem 6.9 N^d-*while*$^\tau$ *is query complete and therefore equivalent to NQAXML*$^\tau$ *with query-dense functions.*

It turns out that a single use of data nondeterminism at the end of the computation is sufficient to achieve completeness. This yields a normal form for $N^d - while^\tau$ programs that pushes all nondeterminism into the last step.

Corollary 6.10 *Every* N^d-*while*$^\tau$ *program P can be written as Q;*{Out := choose(Y)} *where Q is a deterministic* while$^\tau$ *program.*

Proof It is clear that every $N^d - while^\tau$ program P can be written as Q; {$Out :=$ $choose(Y)$} where Q is a $N - while^\tau$ program. Recall that the instruction nondeterminism in Q is needed for the simulation of a nondeterministic Turing machine on encodings of the input. However, the nondeterminism can be absorbed into the last step by deterministically generating all choices and keeping them until the end of the computation. This makes crucial use of tree variables and renders Q deterministic. □

Naturally, the determinization in the normal form comes at the cost of an exponential blowup in the size of intermediate instances generated in the computation.

7 Conclusion

We investigated highly expressive query languages on unordered data trees. We focused largely on QAXML, because this language turned out to be a very appropriate vehicle for understanding the impact and interplay of various language features on expressiveness: (i) the integration of data and computation, (ii) the use of tree versus data variables and (iii) the use of deterministic vs. nondeterministic control.

When patterns and queries do not have tree variables, QAXML with isolated functions has expressiveness limitations reminiscent of relational *while* languages. It also has similarly powerful normal forms, shown by adapting techniques related to FO^k definability. We see the presentation of these normal forms as a major contribution of the paper. We show in particular that NQAXML is equivalent to the much simpler N-while$_\mathbb{N}^{tree}$ and DQAXML to while$_\mathbb{N}^{tree}$. With dense functions, NQAXML becomes complete, while DQAXML falls short even for relational input, due to the copy elimination problem. Interestingly, the deterministic fragment of NQAXML is strictly more expressive than DQAXML (so nondeterminism increases the ability to express *deterministic* queries). We do not know of a natural deterministic complete language without deep equality and tree copying.

Tree variables in patterns and queries partly alleviate the limitations of isolated functions: DQAXML with isolated functions becomes complete with tree variables, but NQAXML falls short of capturing full nondeterminism. To obtain nondeterministic completeness for NQAXML, isolation must be relaxed. The results suggest that dense functions and tree variables are alternatives for achieving query completeness, modulo the subtle limitations mentioned above.

A number of interesting issues were raised by the present work. We mention a few:

- characterize relaxations of the isolation condition for which the results on isolated QAXML programs continue to hold.
- characterize the input and output DTDs for which DQAXML with dense functions is query-complete, or query-complete up to copy elimination.
- characterize the input DTDs for which properties defined by QAXML programs with isolated functions also follow 0-1 laws.

- find natural, deterministic, query-complete languages without deep equality or tree copying.

Many classical models of computation on trees are based on automata and transducers. We plan to consider in future work various forms of transducers for unordered data trees, and their connection to query languages. While a nondeterministic, query-complete transducer is easy to design, this appears to be more challenging for the deterministic case.

References

1. Abiteboul, S., Benjelloun, O., Milo, T.: The active XML project: an overview. VLDB J. **17**(5) (2008)
2. Abiteboul, S., Bourhis, P., Vianu, V.: Comparing workflow specification languages: a matter of views. ACM Trans. Database Syst. **37**(2) (2012). Also ICDT 2011
3. Abiteboul, S., Compton, K.J., Vianu, V.: Queries are easier than you thought (probably). In PODS (1992)
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Reading, MA (1995)
5. Abiteboul, S., Kanellakis, P.: Object identity as a query language primitive. J. Assoc. Comput. Mach. (JACM) **45**(5) (1998)
6. Abiteboul, S., Segoufin, L., Vianu, V.: Static analysis of active XML systems. ACM Trans. Database Syst. **34**(4) (2009). Also PODS 2008
7. Abiteboul, S., Vianu, V.: Generic computation and its complexity. In *STOC*, pp. 209–219 (1991)
8. Abiteboul, S., Vianu, V.: Computing with first-order logic. J. Comput. Syst. Sci. **50**(2) (1995)
9. Benedikt, M., Koch, C.: From XQuery to relational logics. ACM Trans. Database Syst. **34**(4) (2009)
10. Bojanczyk, M.: Automata for data words and data trees. In *RTA*, pp. 1–4 (2010)
11. Calvanese, D., Giacomo, G.D., Hull, R., Su, J.: Artifact-centric workflow dominance. In *ICSOC/ServiceWave* (2009)
12. Hidders, J., Marrara, S., Paredaens, J., Vercammen, R.: On the expressive power of XQuery fragments. In *DBPL* (2005)
13. Hidders, J., Paredaens, J., Vercammen, R., Demeyer, S.: A light but formal introduction to XQuery. In *XSym* (2004)
14. Janssen, W., Korlyukov, A., den Bussche, J.V.: On the tree-transformation power of XSLT. Acta Inf. **43**(6) (2007)
15. Koch, C.: On the complexity of nonrecursive XQuery and functional query languages on complex values. ACM Trans. Database Syst. **31**(4) (2006)
16. Libkin, L.: Elements of Finite Model Theory. Springer, Berlin (2004)
17. Neven, F.: Automata, logic, and XML. In *Computer Science Logic* (2002)
18. Schwentick, T.: Automata for XML - a survey. J. Comput. Syst. Sci. **73**(3) (2007)
19. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In *Computer Science Logic*, pp. 41–57 (2006)
20. Segoufin, L.: Static analysis of XML processing with data values. SIGMOD Record **36**(1), 31–38 (2007)