Highly expressive query languages for unordered data trees

Serge Abiteboul, Pierre Bourhis, Victor Vianu

Highly Expressive Query Languages for Unordered Data Trees*

Serge Abiteboul
INRIA & ENS Cachan
Serge.Abiteboul@inria.fr

Pierre Bourhis
University of Paris-Sud & Oxford University
bourhis@comlab.ox.ac.uk

Victor Vianu†
U.C. San Diego & INRIA
vianu@cs.ucsd.edu

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. Other languages are "almost" complete, but fall short because of subtle limitations reminiscent of the copy elimination problem in object databases.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Query languages

Keywords

Expressiveness, XML, data trees

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 preserving 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. We highlight two important factors affecting expressiveness: the power of the basic tree pattern queries, and the ability to embed code into trees. In order to understand the latter, we use as a vehicle for our study the language Active XML (AXML) that provides a clean, flexible model of XML with embedded programs. 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, arising in expressive relational and object-oriented languages.

The main vehicle for our study, AXML, provides an extension of XML with embedded service calls. This 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 [5, 1], 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* [10], 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 [5]. An instance consists of a forest of unordered, unranked trees whose internal nodes are labeled by tags from a finite alphabet, and whose leaves

*This work has been partially funded by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant Webdam, agreement 226513. http://webdam.inria.fr

†This author was supported in part by the NSF under award IIS-0916515. Work done in part while visiting INRIA and ENS-Cachan.

ICDT 2012, March 26–30, 2012, Berlin, Germany.

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 patterns. 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 fundamental aspects of querying trees and the expressibility of query languages for trees. It can be viewed as a continuation of works on relational languages (see, e.g., [3]) and object-oriented languages [4]. We study in particular the impact on expressiveness of the *embedding* of functions into data, which is a main distinguishing characteristic of AXML. We consider this 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), 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 [6]. 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 preprocessing 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 and also highlights potential 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 [3, 15]. 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. In this case, QAXML with non-deterministic semantics allows expressing any arbitraty computable queries 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 nondeterministically computing 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.

Interestingly, we also 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* [4], 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 yields powerful subtree manipulations by the queries of QAXML functions. 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 sublass 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 earlier, we can show 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 forms 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 [16, 17] and [13]). Many of these studies have focused on trees without data. More recently, trees with data (or over infinite alphabets) 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 [9, 18, 19]. 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) [8, 11, 12, 14]. 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 immetarial. 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. Additional examples are provided in an appendix.

2. PRELIMINARIES

We informally recall some background on relational query languages. See [3, 15] 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) relational 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 relational variables is increasing. The loop stops when two consecutive iterations produce no change to the contents of the relational 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 of a formula in this language to be true for the 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. 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. It is not hard to see 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 $while_{\mathbb{N}}$ [7].

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* [4].

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 $a, b, c, \ldots$), *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 (data values can only be assigned to leaves). We also assume that the trees are *reduced*, i.e., there are no siblings subtrees that are isomorphic by a mapping preserving the tags *and* data values. This is analogous to the set (rather than bag) 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 $\langle I, J \rangle \in \mathcal{R}$.

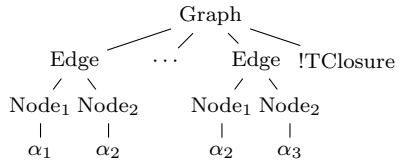


Figure 1: AXML tree

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) 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.

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 Figure 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

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

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 $\neg$. The root of each tree pattern must be positive. We call a node in the tree pattern T a *boundary node* if it 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 of 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 Figure 2. This cannot always be done. For example, if for the same tree pattern the condition is $V0 = Graph \wedge V2 = Node1 \wedge V3 = Node2 \wedge (V1 = Self\text{-}Loop \rightarrow X = Y) \wedge (V1 = Edge \rightarrow X \neq Y)$ then there no fixed assignment of labels to nodes. Finally, a more complex pattern and its concise representation are shown in Figure 3.

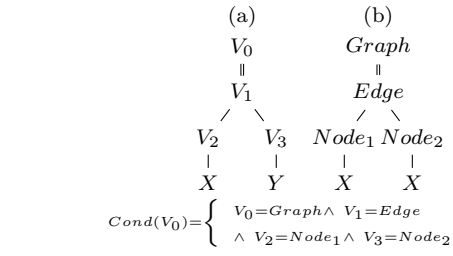


Figure 2: A simple pattern: full specification (a) and concise version (b)

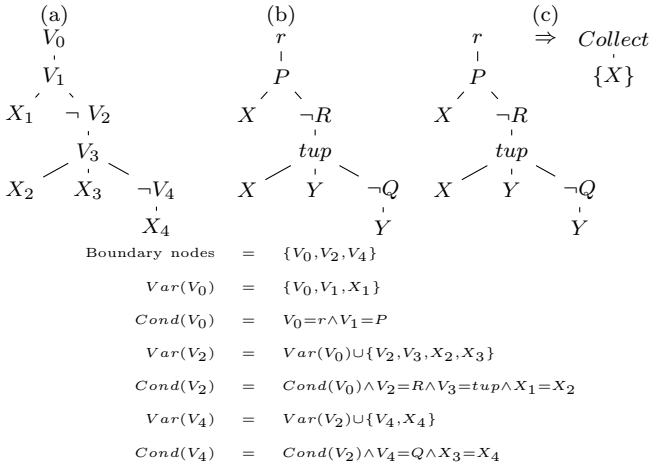


Figure 3: A complex pattern: (a) full specification (b) concise version (c) a query using the pattern

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 Figure 3. Its body is the pattern in Figure 3.

Note that, in the above, variables in heads of queries extract data values from the input. We will consider in Section

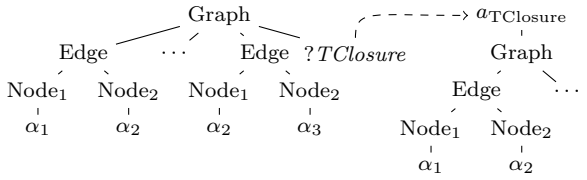


Figure 4: An AXML instance with an eval link

6 an extension allowing variables in heads to extract entire subtrees from the input.

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 tuple (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 the initial state of the workspace of the call to f.

An AXML *instance* I is a pair $(\mathcal{T}, \mathit{eval})$, 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 $?f$, $\mathit{eval}(n)$ 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 $\mathit{eval}(n)$ 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 $I = (\mathcal{T}, \mathit{eval})$ and $I' = (\mathcal{T}', \mathit{eval}')$ 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 only if $\mathit{eval}(n)$ contains no running calls $?g$, in which case the answer consists of the result of evaluating the return query of f on $\mathit{eval}(n)$, after which $(n, \mathit{eval}(n))$ is removed from the graph of *eval*.

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

We are interested in *computations* of NQAXML programs. An *initial* instance of program $\mathcal{Q} = (\Phi, \Delta)$ is an instance consisting of a single tree satisfying Δ. A *computation* of $\mathcal{Q}$ 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.

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 in the appendix.

We consider two extremes: (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 next 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 nontrivial 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 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).*

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, well explored in the theory of relational query languages [15]. 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 $R_{\mathcal{E}}$ 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 $R_{\mathcal{E}}$ is definable from $\sigma(I)$ by an FO^k formula (using 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 (†), 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:

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

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 rule $Body \rightarrow Head$ of a pattern query, with set of bindings $\mathcal{B}$ for the m variables in the head of the rule. To each such set $\mathcal{E}$ of trees we associate a relation $R_{\mathcal{E}}$ of arity m containing the bindings in $\mathcal{B}$.

The following key fact can be shown.

LEMMA 4.2. *Each relation $R_{\mathcal{E}}$ 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 $\mathcal{Q}$.*

The proof uses the language while$^*_{\mathbb{N}}$ defined in preliminaries. We consider a nondeterministic variant N-while$^*_{\mathbb{N}}$ obtained by allowing a choice operator, $program_1 \mid program_2$. By results in [7], (where while$^*_{\mathbb{N}}$ is denoted while^{++}) every relation not containing integers and definable in while$^*_{\mathbb{N}}$ is also definable in while$_{\mathbb{N}}$, so also in FO^k for some fixed k depending on the program. This easily extends to the nondeterministic variants. We then show that every relation $R_{\mathcal{E}}$ is definable from $\sigma(I)$ by a program in N-while$^*_{\mathbb{N}}$. The key idea is to represent AXML instances generated in the computation of a QAXML program $\mathcal{Q}$ as relational structures, constructed from $\sigma(I)$ by the N-while$^*_{\mathbb{N}}$ program. In particular, new tree nodes are represented by tuples containing both data values from $\sigma(I)$ and integers. Details are omitted.

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 4.5, 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$^{tree}_{\mathbb{N}}$, then a deterministic one denoted while$^{tree}_{\mathbb{N}}$. The language N-while$^{tree}_{\mathbb{N}}$ 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$^{tree}_{\mathbb{N}}$, denoted while$^{tree}_{\mathbb{N}}$, is obtained by disallowing nondeterministic choice. An example of a while$^{tree}_{\mathbb{N}}$ program is provided in the appendix.

4.3 NQAXML with Isolated Functions

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

THEOREM 4.4. *NQAXML programs with isolated functions express the same set of tree queries as N-while$^{\text{tree}}_{\mathbb{N}}$.*

The simulation of N-while$^{tree}_{\mathbb{N}}$ by NQAXML with isolated functions is rather straightforward. The converse simulation is much more intricate and makes crucial use of Lemma 4.2. The simulation of a NQAXML program $\mathcal{Q}$ consists of several stages:

(i) Compute from input 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;

(v) For each terminating computation, produce in variable *Out* the output tree encoded on the tape.

Note that the stack and instructions of the form $X := Y \cup Z$, $X := a[Y]$ are only needed in step (v) of the simulation.

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

THEOREM 4.5. *For each NQAXML program $\mathcal{Q}$ with isolated functions there is an equivalent program $\mathcal{Q}_{\text{nf}}$ 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) eliminates in PTIME all data values from the input tree, (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 4.6. *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 [2], 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. 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 4.7. *The normal form of Theorem 4.5 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 4.8. *One might wonder if it is possible to relax the definition of QAXML with isolated functions while preserving Lemma 4.2 and Theorems 4.4 and 4.5. 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*

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

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$_{\mathbb{N}}^{\text{tree}}$ 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$_{\mathbb{N}}^{\text{tree}}$ 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. The proof relies on the normal form provided by Corollary 4.7.

THEOREM 4.9. *The language while$_{\mathbb{N}}^{\text{tree}}$ expresses precisely the deterministic fragment of N-while$_{\mathbb{N}}^{\text{tree}}$.*

We now state the analog of Theorem 4.4.

THEOREM 4.10. *DQAXML programs with isolated functions express the same set of tree queries as while$_{\mathbb{N}}^{\text{tree}}$.*

As a consequence of Theorems 4.4, 4.9 and 4.10, we have the following nontrivial result.

THEOREM 4.11. *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 a 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 4.4.

THEOREM 4.12. *The following languages express the same Boolean tree queries:*

(i) NQAXML and DQAXML with isolated functions;

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

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

COROLLARY 4.13. *For each Boolean (non)deterministic QAXML program $\mathcal{Q}$ with isolated functions there is a (non)deterministic Boolean QAXML program $\mathcal{Q}_{\text{nf}}$ 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:

$$\begin{array}{lcl} R & \rightarrow & |tup| \geq 0 \\ tup & \rightarrow & |A_1| = 1 \ \wedge \ldots \wedge \ |A_m| = 1 \\ A_i & \rightarrow & |dom| = 1, \quad 1 \leq i \leq m \end{array}$$

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 embeddeed 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. The computation of $\mathcal{R}$ by an NQAXML program is done in several phases:

(i) on input I, nondeterministically 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.

□

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. Consider the query $\mathcal{R}$ whose input I is a set of n data values, and whose 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. □

Note that the counterexample in the proof of Theorem 5.3 uses bounded 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 $\mathcal{R}$.*

The 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 [4]. We 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 ouput 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 Theorem 5.4 and Corollary 5.6 hold.

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, 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 (other than under the root).

We begin by informally 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$^{\mathcal{T}}$. The notion of isolated and dense program remains unchanged. An example of a QAXML$^{\mathcal{T}}$ program is provided in the appendix.

We first consider QAXML$^{\mathcal{T}}$ with isolated functions and deterministic semantics, denoted DQAXML$^{\mathcal{T}}$.

THEOREM 6.1. *DQAXML$^{\mathcal{T}}$ with isolated functions is query complete.*

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

We now consider QAXML$^{\mathcal{T}}$ with nondeterministic semantics, denoted NQAXML$^{\mathcal{T}}$. It turns out that NQAXML$^{\mathcal{T}}$ is *not* query complete. For example, it cannot express the query that outputs one arbitrary data value from the input. Intuitively, this is because NQAXML$^{\mathcal{T}}$ 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 6.2. *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 above-mentioned query that outputs one arbitrary data value from a set of input values is *not* weakly nondeterministic. The query 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 second query happens to be weakly nondeterministic for all input DTDs.

THEOREM 6.3. *NQAXML$^{\mathcal{T}}$ with isolated functions expresses precisely the weakly nondeterministic tree queries.*

To summarize the results in this section so far, QAXML$^{\mathcal{T}}$ 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$^{\mathcal{T}}$
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 6.4. *NQAXML$^{\mathcal{T}}$ with query-dense functions is query complete.*

Finally, we note that Theorems 6.1 and 6.4 yield some strong normal forms for QAXML$^{\mathcal{T}}$ programs.

THEOREM 6.5. (*i*) *For every DQAXML$^{\mathcal{T}}$ program one can effectively construct an equivalent DQAXML$^{\mathcal{T}}$ program with isolated functions.* (*ii*) *For every NQAXML$^{\mathcal{T}}$ program one can effectively construct an equivalent NQAXML$^{\mathcal{T}}$ 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$^{\mathcal{T}}$ languages. The deterministic language, denoted *while*$^{\mathcal{T}}$ 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*$^{\mathcal{T}}$, 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)-$while^{tree}_{\mathbb{N}}$, these languages have no integer variables, no stack, and no tree constructors, because all can be simulated using tree variables. A simple example of a $while^{\mathcal{T}}$ program is given in the appendix.

The following establishes the connection between the (N)-$while^{\mathcal{T}}$ and QAXML$^{\mathcal{T}}$languages. The proofs are similar to Theorems 6.1 and 6.3.

THEOREM 6.6. *(i) $while^{\mathcal{T}}$ is equivalent to DQAXML$^{\mathcal{T}}$ with isolated functions and is query complete; (ii) N-$while^{\mathcal{T}}$ is equivalent to NQAXML$^{\mathcal{T}}$ with isolated functions and expresses exactly the weakly nondeterministic tree queries.*

In order to obtain a complete nondeterministic language, N-$while^{\mathcal{T}}$ 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-$while^{\mathcal{T}}$. The following is immediate.

THEOREM 6.7. *N^d-while$^{\mathcal{T}}$ is query complete and therefore equivalent to NQAXML$^{\mathcal{T}}$ 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-$while^{\mathcal{T}}$ programs that pushes all nondeterminism into the last step.

COROLLARY 6.8. *Every* N^d-while$^{\mathcal{T}}$ *program P can be written as Q;*{Out := choose(Y)} *where Q is a deterministic* while$^{\mathcal{T}}$ *program.*

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$^{tree}_{\mathbb{N}}$ and DQAXML to while$^{tree}_{\mathbb{N}}$. 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.

8. REFERENCES

[1] S. Abiteboul, P. Bourhis, and V. Vianu. Comparing workflow specification languages: a matter of views. In *ICDT*, 2011.

[2] S. Abiteboul, K. J. Compton, and V. Vianu. Queries are easier than you thought (probably). In *PODS*, 1992.

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

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. *Journal of the Association for Computing Machinery (JACM)*, 45(5), 1998.

[5] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34(4), 2009. Also PODS 2008.

[6] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *STOC*, pages 209–219, 1991.

[7] S. Abiteboul and V. Vianu. Computing with first-order logic. *J. Comput. Syst. Sci.*, 50(2), 1995.

[8] M. Benedikt and C. Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34(4), 2009.

[9] M. Bojanczyk. Automata for data words and data trees. In *RTA*, pages 1–4, 2010.

[10] D. Calvanese, G. D. Giacomo, R. Hull, and J. Su. Artifact-centric workflow dominance. In *ICSOC/ServiceWave*, 2009.

[11] J. Hidders, S. Marrara, J. Paredaens, and R. Vercammen. On the expressive power of XQuery fragments. In *DBPL*, 2005.

[12] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *XSym*, 2004.

[13] W. Janssen, A. Korlyukov, and J. V. den Bussche. On the tree-transformation power of xslt. *Acta Inf.*, 43(6), 2007.

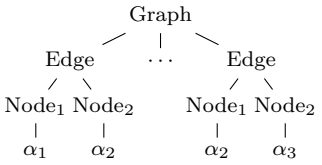


Figure 5: Input graph

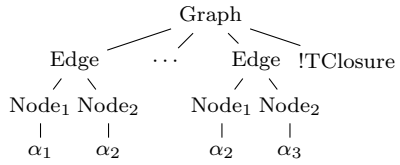


Figure 6: QAXML initial instance

[14] C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4), 2006.

[15] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[16] F. Neven. Automata, logic, and XML. In *CSL*, 2002.

[17] T. Schwentick. Automata for XML - a survey. *J. Comput. Syst. Sci.*, 73(3), 2007.

[18] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57, 2006.

[19] L. Segoufin. Static analysis of xml processing with data values. *SIGMOD Record*, 36(1):31–38, 2007.

APPENDIX

A. ADDITIONAL EXAMPLES

We illustrate the QAXML and *while* languages with two examples. The first shows a QAXML and a while$_{\mathbb{N}}^{tree}$ program with data variables computing the transitive closure of a graph. The second exhibits QAXML$^{\mathcal{T}}$ and *while*$^{\mathcal{T}}$ programs with tree variables computing the parity of the depth of a tree.

A.1 Transitive Closure

A directed graph is represented as in Figure 5. We exhibit a QAXML program with isolated functions and a while$_{\mathbb{N}}^{tree}$ program computing a representation of the transitive closure of the graph.

A.1.1 QAXML Program

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 Figure 6 represents the initial instance of the QAXML program. It is obtained from the input tree in Figure 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* (Figure 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 Figure 8. An instance obtained by the activation of *Iterate* is depicted in Figure 9.

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 Figure 10. The computation terminates when no new edges are added.

A.1.2 While$_{\mathbb{N}}^{tree}$ Program

A while$_{\mathbb{N}}^{tree}$ program computing the transitive closure of the graph is sketched below.

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

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 Figure 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.

A.2 Depth Parity

We exhibit a QAXML$^{\mathcal{T}}$ program with isolated functions and tree variables and a While$^{\mathcal{T}}$ program 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.

A.2.1 QAXML$^{\mathcal{T}}$ Program

The QAXML$^{\mathcal{T}}$ program we exhibit has isolated functions and computes the desired query with either deterministic or nondeterministic semantics. The main component of the QAXML$^{\mathcal{T}}$ 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. Fig-

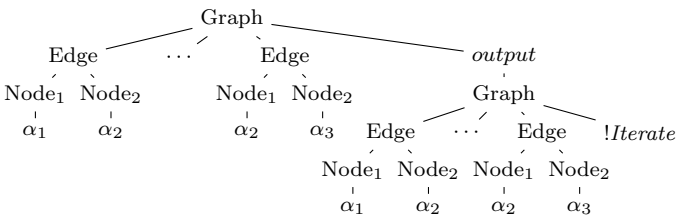


Figure 7: QAXML instance after return of TClosure

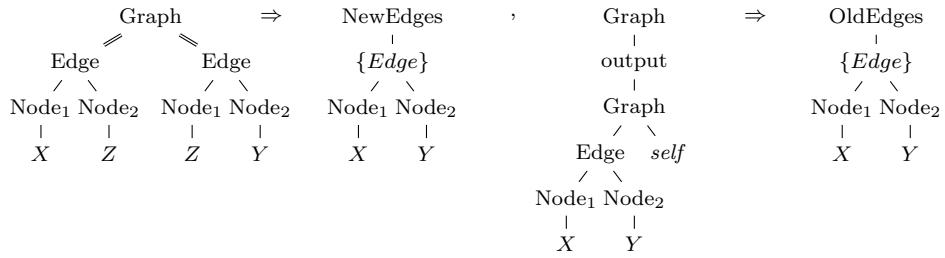


Figure 8: Input query of *Iterate*

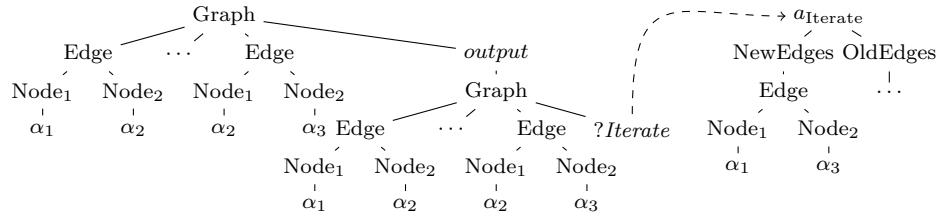


Figure 9: QAXML program after activation of *Iterate*

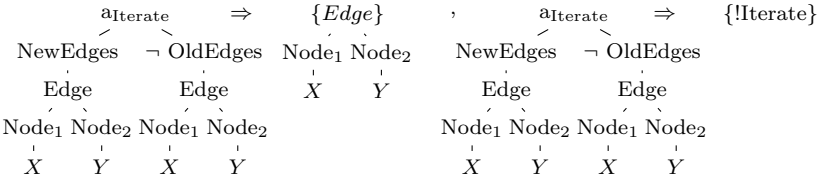


Figure 10: Return query of *Iterate*

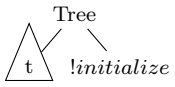


Figure 11: Initial instance for the QAXML$^{\mathcal{T}}$ program *Parity*

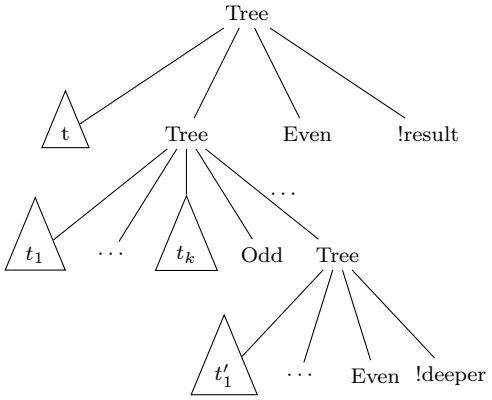


Figure 12: Intermediate instance in the computation of the QAXML$^{\mathcal{T}}$ program *Parity*

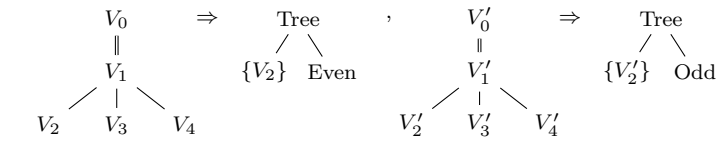
ure 12 depicts an intermediate instance in the computation of the program.

In more detail, the initial instance is of the form shown in Figure 11, with a function !*initialize* under the root. The computation starts with a call to !*initialize* that returns 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 Figure 13. It copies the sibling parity flag *Even* or *Odd* and the proper siblings subtrees of the function call. The return query, shown in Figure 14, 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*.

A.2.2 While$^{\mathcal{T}}$ Program

A *While*$^{\mathcal{T}}$ program computing the parity of the depth of the input tree is sketched below.

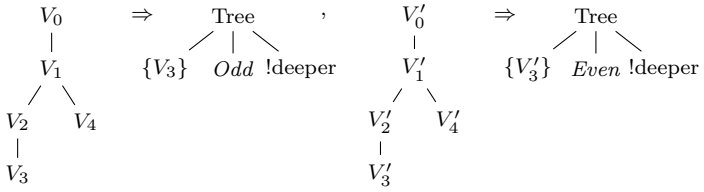
Data: A tree stored in *Input*
Result: A node labeled by *Even* or *Odd*, stored in *Output*
begin
  Parity := *Even*
  Tree := *Children*(*Input*)
  while Tree! = ∅ **do**
    Parity := *Flip*(*Parity*)
    Tree := *Children*(*Tree*)
  Output := *Parity*
end



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

$$Cond(V'_0) = \begin{cases} & V'_0 = \text{Tree} \\ \wedge & V'_1 = \text{Tree} \\ \wedge & V'_2 \neq \text{Even} \wedge V'_2 \neq \text{Odd} \wedge V'_2 \neq \text{!deeper} \wedge V'_2 \neq \text{!result} \\ \wedge & V'_3 = \textit{self} \\ \wedge & V'_4 = \text{Odd} \end{cases}$$

Figure 13: The input query of *deeper*



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

$$Cond(V'_0) = \Big(V'_4 = \text{Odd}\Big)$$

Figure 14: The output query of *deeper*

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.