Structural Recursion on Ordered Trees and List-Based Complex Objects

ROBERTSON, Edward L.; LAWRENCE, Saxton V.; VAN GUCHT, Dirk & VANSUMMEREN, Stijn (2007) Structural Recursion on Ordered Trees and List-Based Complex Objects. In: Database Theory - ICDT 2007. p. 344-358.

DOI: 10.1007/11965893_24
Handle: http://hdl.handle.net/1942/7740

Structural Recursion on Ordered Trees and List-based Complex Objects

Expressiveness and PTIME Restrictions

Edward L. Robertson[1], Lawrence V. Saxton[2], Dirk Van Gucht[1], and Stijn Vansummeren[3]*

[1] Indiana University, USA
[2] University of Regina, Canada
[3] Hasselt University and Transnational University of Limburg, Belgium

Abstract. XML query languages need to provide some mechanism to inspect and manipulate nodes at all levels of an input tree. In this paper we investigate the expressive power provided in this regard by structural recursion. We show that the combination of vertical recursion down a tree combined with horizontal recursion across a list of trees gives rise to a robust class of transformations: it captures the class of all primitive recursive queries. Since queries are expected to be computable in at most polynomial time for all practical purposes, we next identify a restriction of structural recursion that captures the polynomial time queries. Although this restriction is semantical in nature, and therefore undecidable, we provide an effective syntax. We also give corresponding results for list-based complex objects.

1 Introduction

Over the past few years, the ordered, node-labeled tree data model of XML has emerged as the standard format for representing and exchanging data on the web. Often, there is no a priori bound on the width and depth of such trees. As such, an XML query language needs to provide some mechanism to inspect and manipulate nodes at all levels. XQuery, the standard XML query language currently under development by the World Wide Web Consortium [4, 14], uses *recursion* for this purpose. For example, to compute the table of contents of books in which sections can be arbitrarily nested, one would write:

```
function toc(t) {
  for s in t/section return <section>{ s/title, toc(s) }</section>
};

<toc> toc(book)</toc>
```

* Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

Here, *toc* is a recursive function returning for each `section` child s of its input tree t a new `section` node containing the title and table of contents of s.

XQuery allows arbitrary recursive function definitions, resulting in a Turing complete language. Turing completeness is an undesirable property for a query language however, as it makes optimization difficult and allows non-terminating queries. Therefore, it is desirable to look for suitable restrictions of arbitrary recursion in XQuery. Non-termination can be prevented by closely tying recursion to the structure of the data being operated upon, i.e., by restricting to *structural recursion*. For example, a structural recursive function computing on a tree t can only recursively call itself on the children of t. The function *toc* defined above is an example of such a structural recursion. Similarly, a structural recursive function computing on a list l can only recursively call itself on the tail of l. A typical example of such a structural recursion is the list reversal function *rev*:

$$\mathsf{function}\ rev(l)\ \{\ \mathsf{if}\ empty(l)\ \mathsf{then}\ l\ \mathsf{else}\ rev(tl(l)),\ hd(l)\ \};$$

Here *hd* returns the head of a nonempty list, *tl* returns the tail of a nonempty list, and the comma operator is concatenation of lists.

In this paper, we study the properties of structural recursion as a candidate replacement of arbitrary recursion in XQuery. In particular, we study the combination of vertical structural recursion down trees and horizontal structural recursion across lists of trees (as trees and lists of trees both naturally occur in the XQuery data model [4, 14].

Structural recursion is an important primitive in database theory. It has been used to query (nested) collections based on sets, or-sets, pomsets, bags, and lists [8, 17, 21, 24]; unordered trees and graphs [7]; and sequences and text documents [5]. Unrestricted structural recursion leads to highly expressive query languages. For example, Buneman et al. have shown [8] that structural recursion on nested relations is equivalent to the powerset algebra of Abiteboul and Beeri [1], which by a result of Hull and Su, captures exactly the class of elementary nested relational queries [20] (i.e., the queries with hyper-exponential time data complexity). Furthermore, Immerman et al. [21] and Suciu and Wong [27] have shown that, in the presence of object invention, the class of functions $f\colon \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ representable with structural recursion on sets coincides with the class of primitive recursive functions [6]. The resulting language is hence strictly more powerful than the elementary queries. This result was later extended to structural recursion on (nested) bags by Libkin and Wong [24].

Since tree construction is a form of object invention, it should come as no surprise that a similar result also hold for structural recursion in XQuery. We actually obtain a slightly stronger result than that of Immerman et al.: not only does the class of functions $f\colon \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ representable in our language coincide with the class of primitive recursive functions, but the class of expressible *queries* coincides with the class of queries that have primitive recursive time data complexity.

From a complexity point of view, structural recursion is hence too powerful a primitive, as queries are expected to be computable in at most polynomial time

for all practical purposes. A restriction of structural recursion to polynomial time is therefore desirable. Nevertheless, this restriction should still enable all polynomial time queries.

The first such restriction was given by Immerman et al. for structural recursion on sets, by disallowing all forms of nesting [21]. The resulting language captures exactly the polynomial time flat relational queries. Their restriction does not transfer to nested data models or data models with duplicates such as bags or lists, however. As such, it is not directly applicable to structural recursion in XQuery. A different restriction technique, known as *bounded recursion* dates back to Cobham [11], and was applied to structural recursion on flat lists by Grumbach and Milo [17]. Bounded recursion is best explained by means of an example. Consider the unbounded function that computes a list of size exponential in the size of l:

$$\mathsf{function}\ \mathit{explist}(l)\ \{\mathsf{if}\ \mathit{empty}(l)\ \mathsf{then}\ l\ \mathsf{else}\ \mathit{explist}(\mathit{tl}(l)),\mathit{explist}(\mathit{tl}(l))\};$$

Since *explist* generates exponential output, it certainly cannot be evaluated in polynomial time. Bounded recursion prevents the expression of *explist* by requiring each recursive function definition to halt computation whenever the result becomes larger than some explicitly given size bound b. That is, with bounded recursion, *explist* is required to have the following form:

$$\begin{array}{l}\mathsf{function}\ \mathit{explist}'(l,b)\ \{\\ \quad \mathsf{if}\ \mathit{empty}(l)\ \mathsf{then}\ l\ \mathsf{else}\\ \quad\quad \mathsf{let}\ r = \mathit{explist}'(\mathit{tl}(l),b),\mathit{explist}'(\mathit{tl}(l),b)\ \mathsf{in}\\ \quad\quad \mathsf{if}\ \mathit{sizeof}(r) \leq \mathit{sizeof}(b)\ \mathsf{then}\ r\ \mathsf{else}\ \mathit{explist}'(\mathit{tl}(l),b)\\ \};\end{array}$$

In particular, the size of $\mathit{explist}'(l,b)$ is always bounded by the size of b. Since the value for b will ultimately be computed by an expression that does not involve recursion, the size of recursively computed outputs is always polynomial, and this guarantees that all expressible queries can be evaluated in polynomial time (see [11, 17] for details). This way of bounding recursion has also been applied to query languages over nested relations and bags based on inflationary fixpoint operators [12, 25, 26].

Although bounded recursion is useful for capturing polynomial time, it is unsatisfactory from a practical point of view, as the programmer is required to give explicit complexity bounds upon each recursive function invocation. More intrinsic restrictions of structural recursion on the bitstrings by means of *predicative recursion* were proposed by Bellantoni and Cook [3] and Leivant [23]. Their restrictions were later generalized to arbitrary recursive functions operating on ranked trees generated by a free term algebra by Caseiro [9]. Her techniques were later explained by means of a type system based on linear and modal logic in the context of a higher-order functional programming language by Hofmann [18, 19].

In this paper, we apply Caseiro's observations and ideas to structural recursion operating on lists and *unranked* trees to obtain an intrinsic restriction

that captures exactly the class of polynomial time queries. In particular, we prevent the definition of *explist* above by disallowing all forms of *doubling* like $\mathit{explist}(\mathit{tl}(l)),\mathit{explist}(\mathit{tl}(l))$. Although this restriction is semantical in nature, and therefore undecidable, we provide an effective syntax for it.

For the formal development of our results, we find it convenient to not study structural recursion directly in XQuery itself, but in the *Nested Tree Calculus* $\mathcal{NTC}$. The $\mathcal{NTC}$ can be viewed as the combination of non-recursive for-let-where-return XQuery $\mathcal{XQ}$ and a complex object calculus for nested lists $\mathcal{COC}$. These languages blend naturally together, as it has repeatedly been observed in the literature that there is a close correspondence between $\mathcal{XQ}$ and calculi for complex objects [15, 22, 29]. In fact, we show $\mathcal{NTC}$ to be a *conservative* extension of both $\mathcal{XQ}$ and $\mathcal{COC}$, even in the presence of (restricted) structural recursion. As a consequence, results about (restricted) structural recursion in $\mathcal{NTC}$ transfer immediately to the respective sublanguages. As an important corollary we obtain that our polynomial time restriction of structural recursion also allows to capture the polynomial time queries on nested lists. Hence, suitably restricted structural recursion provides an elegant alternative to the rather awkward *list-trav* iteration construct of Colby et al. [13], which also captures polynomial time on nested lists.

Organization. This paper is further organized as follows. We start by introducing our data model and the notion of a query in Section 2. Next, we define the Nested Tree Calculus $\mathcal{NTC}$ and structural recursion in Section 3. The expressive power of structural recursion in $\mathcal{NTC}$ is in studied in Section 4, where we also show how to restrict it to polynomial time. Finally, we show $\mathcal{NTC}$ to be a conservative extension of both $\mathcal{XQ}$ and $\mathcal{COC}$ in Section 5.

2 Preliminaries

Our data model is a combination of the tree-based data model of XQuery and the list-based complex object data model [8]. That is, we consider the types given by the following grammar:

$$\sigma, \tau ::= \mathsf{atom} \mid \mathsf{tree} \mid \sigma \times \tau \mid [\tau].$$

Semantically, a type denotes a set of *values*. The values of the base type atom are *atoms* like the integers, the strings, and so on. The elements of the base type tree are finite trees. Here, a tree is a pair $\langle a \rangle\, v$ with a an atom and v a finite list of trees. Values of the product type $\sigma \times \tau$ are pairs (v, w) with v and w values of type σ and τ, respectively. Finally, values of the list type $[\tau]$ are finite lists of values of type τ. Note that our types are not meant to describe the structure of trees (as e.g., XML Schema types [28] do). They are used solely to define our data model and to structure $\mathcal{NTC}$ expressions.

According to the XQuery data model, an XQuery value is either an atomic data value; an ordered tree; a list of atoms; or a list of trees [4, 14]. Arbitrary nested combinations of atoms, pairs, lists, and trees are not allowed. Conversely, the list-based complex object data model typically does not include trees. We

therefore formally define an *XQuery type* (xq-type for short) to be either atom; tree; [atom]; or [tree], while a *complex object type* (co-type for short) is a type in which tree does not occur.

Notational convention. In what follows, we will range over atoms by letters from the beginning of the alphabet. Also, we will denote the empty list by $[\,]$, non-empty lists by for example $[a, b, c]$, and the concatenation of two lists l_1 and l_2 by $l_1 +\!\!+ l_2$. Following [8], we write $v \uparrow l$ for $[v] +\!\!+ l$. We feel free to omit parentheses in types and write $\tau_1 \times \cdots \times \tau_n$ for $(\ldots((\tau_1 \times \tau_2) \times \tau_3) \cdots \times \tau_n)$. Finally, we write $v : \tau$ to indicate that v is a value of type τ.

Queries. Queries on values are defined by extending the classical definition of Chandra and Harel [10] for relations. That is, a *query* is a function $q\colon \sigma \to \tau$ that maps values in σ to values in τ, for some types σ and τ. If $\sigma = \sigma' \times \cdots \times \sigma''$ where $\sigma', \ldots, \sigma'', \tau$ are xq-types, then q is an *xquery*. Similarly, if σ and τ are co-types, then q is a *complex object query*. Queries must be computable and generic (i.e., they must treat all but a finite set of atoms in uninterpreted way [2]). We will use the *domain Turing machine* of Hull and Su [20] as our model of computation. Domain Turing machines (DTMs for short) are augmented Turing machines that are specifically designed to express generic computations; in particular, they can work directly with an *infinite* alphabet on their tape. In contrast to normal Turing machines, there is hence no need to (rather clumsily) encode atoms as strings over finite alphabets. Nevertheless, DTMs can be simulated by ordinary Turing machines while respecting the complexity classes considered in this paper [20].

A query $q\colon \sigma \to \tau$ is said to *run in polynomial (resp. primitive recursive [6]) time* if there exists a DTM M that, starting from the standard encoding of a value $v\colon \sigma$, produces the standard encoding of $q(v)$ in at most polynomially many (resp. primitive recursive many) steps in terms of the size of the input. Note that the query itself is fixed (i.e., we consider data complexity, not combined complexity). Here, the standard encoding $str(v)$ of a value v on a DTM tape is as follows. The type constructor symbols $(,), [,$ and $]$ are part of the tape alphabet, as are all of the atoms. Then a is encoded by itself; $\langle a \rangle\, v$ is encoded by the string $(str(a)\, str(v))$; (v, w) is encoded by $(str(v) \ldots str(w))$; and $[v, \ldots, w]$ is encoded by $[str(v) \ldots str(w)]$. We write $\mathbf{s}\,(v)$ for the length of $str(v)$.

3 Query languages

In this section, we define the *Nested Tree Calculus* $\mathcal{NTC}$, a first-order calculus that extends both non-recursive for-let-where-return XQuery and a complex object calculus for nested lists. We will show in Section 5 that $\mathcal{NTC}$ is a *conservative* extension of these languages, even in the presence of structural recursion. As a consequence, in order to prove our expressiveness results claimed in the Introduction, it suffices to prove them for $\mathcal{NTC}$; they immediately transfer to the respective sublanguages.

$$\frac{}{x^\tau : \tau} \qquad \frac{}{a : \mathsf{atom}} \qquad \frac{e_1 : \sigma \quad e_2 : \sigma \quad e_3 : \tau \quad e_4 : \tau}{\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4 : \tau} \qquad \frac{e : \sigma \quad e' : \tau}{(e, e') : \sigma \times \tau}$$

$$\frac{e : \sigma \times \tau \quad e' : \tau'}{\mathsf{case}\ e\ \mathsf{of}\ \{(x^\sigma, y^\tau) \to e'\} : \tau'} \qquad \frac{}{[\,] : [\tau]} \qquad \frac{e : \tau}{[e] : [\tau]} \qquad \frac{e_1 : [\tau] \quad e_2 : [\tau]}{e_1 +\!\!+ e_2 : [\tau]}$$

$$\frac{e : [\sigma] \quad e_1' : \tau \quad e_2' : \tau}{\mathsf{case}\ e\ \mathsf{of}\ \{[\,] \to e_1';\ x^\sigma \uparrow y^{[\sigma]} \to e_2'\} : \tau} \qquad \frac{e : [\sigma] \quad e' : [\tau]}{\mathsf{for}\ x^\sigma\ \mathsf{in}\ e\ \mathsf{return}\ e' : [\tau]}$$

$$\frac{e_1 : \mathsf{atom} \quad e_2 : [\mathsf{tree}]}{\langle e_1 \rangle\, e_2 : \mathsf{tree}} \qquad \frac{e : \mathsf{tree} \quad e' : \sigma}{\mathsf{case}\ e\ \mathsf{of}\ \{\langle x^{\mathsf{atom}} \rangle\, y^{[\mathsf{tree}]} \to e'\} : \sigma}$$

$$\frac{e : \tau}{\lambda x^\sigma . e : \sigma \to \tau} \qquad \frac{f : \sigma \to \tau \quad e : \sigma}{fe : \tau}$$

Fig. 1. Expressions of $\mathcal{NTC}$.

3.1 Nested Tree Calculus

To avoid confusion, we note that $\mathcal{NTC}$ is a first-order language; structural recursion operators will be added in Section 3.2. The expressions of $\mathcal{NTC}$ are explicitly typed, and are formed according to the typing rules of Fig. 1. There are two sorts of expressions: *value expressions* and *function expressions*. Value expressions like $a : \mathsf{atom}$ intuitively evaluate to values and are typed by a normal type, while function expressions like $\lambda x^\sigma . e$ intuitively evaluate to queries and are typed by a function type $\sigma \to \tau$. Note that variables are also explicitly typed; we write x^τ to denote that x is a variable of type τ. To ease notation, we will often omit the explicit type annotations in superscript when they are clear from the context. We use an ML-like notation for value inspection. For example, $\mathsf{case}\ e\ \mathsf{of}\ \{(x, y) \to e'\}$ should be understood to be the expression that first evaluates e to a pair (v, w) and then evaluates e' with x bound to v and y bound to w. This non-standard syntax for inspection of pairs, lists, and trees will allow us to easily define our polynomial time restriction in Section 4. The set $FV(e)$ of *free variables* of an expression e is defined as usual, with lambda abstraction and the case expressions acting as binders. For example, $FV(\lambda x.e) = FV(e) - \{x\}$ and $FV(\mathsf{case}\ e\ \mathsf{of}\ \{[\,] \to e_1;\ x \uparrow y \to e_2\}) = FV(e) \cup FV(e_1) \cup (FV(e_2) - \{x, y\})$. We will refer to expressions without free variables such as $\lambda x.(x, x)$ as *closed expressions*.

Semantics. Intuitively, a value expression $e : \tau$ evaluates to a value in τ when given values for its free variables, while a function expression $f : \sigma \to \tau$ evaluates to a query mapping values in σ to values in τ. Formally, a value expression $e : \tau$ denotes a value $[\![e]\!]_\kappa$ under context κ, while a function expression $f : \sigma \to \tau$ denotes a query $[\![f]\!]_\kappa$ under context κ. Here, a *context* κ is a function from variables to values respecting types (i.e., $\kappa(x) : \tau$ for all x^τ). We denote by $x : v, \kappa$ the context that equals κ on all variables except x, which it maps to v. The denotation of all expressions is inductively defined in Table 1. It is easy to see that the denotation of an expression depends only on its free variables: if κ and κ' agree on $FV(e)$ then $[\![e]\!]_\kappa = [\![e]\!]_{\kappa'}$. As such, the input context to an expression

$$
\begin{array}{ll}
[\![x]\!]_\kappa & = \kappa(x) \\
[\![a]\!]_\kappa & = a \\
[\![\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4]\!]_\kappa & = \begin{cases} [\![e_3]\!]_\kappa & \text{if } [\![e_1]\!]_\kappa = [\![e_2]\!]_\kappa \\ [\![e_4]\!]_\kappa & \text{otherwise} \end{cases} \\
[\![(e, e')]\!]_\kappa & = \big([\![e]\!]_\kappa, [\![e']\!]_\kappa\big) \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \{(x, y) \to e'\}]\!]_\kappa & = [\![e']\!]_{x\colon v, y\colon w, \kappa} \quad \text{where } [\![e]\!]_\kappa = (v, w) \\
[\![[\,]]\!]_\kappa & = [\,] \\
[\![[e]]\!]_\kappa & = [[\![e]\!]_\kappa] \\
[\![e +\!\!+ e']\!]_\kappa & = [\![e]\!]_\kappa +\!\!+ [\![e']\!]_\kappa \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \{[\,] \to e'_1;\ x \uparrow y \to e'_2\}]\!]_\kappa & = \begin{cases} [\![e'_1]\!]_\kappa & \text{when } [\![e]\!]_\kappa = [\,] \\ [\![e'_2]\!]_{x\colon v, y\colon w, \kappa} & \text{when } [\![e]\!]_\kappa = v \uparrow w \end{cases} \\
[\![\mathsf{for}\ x\ \mathsf{in}\ e\ \mathsf{return}\ e']\!]_\kappa & = [\![e']\!]_{x\colon v, \kappa} +\!\!+ \cdots +\!\!+ [\![e']\!]_{x\colon w, \kappa} \\
 & \quad \text{where } [\![e]\!]_\kappa = [v, \ldots, w] \\
[\![\langle e\rangle\, e']\!]_\kappa & = \langle [\![e]\!]_\kappa \rangle\, [\![e']\!]_\kappa \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \{\langle x\rangle\, y \to e'\}]\!]_\kappa & = [\![e']\!]_{x\colon a, y\colon v, \kappa} \quad \text{where } [\![e]\!]_\kappa = \langle a\rangle\, v \\
[\![\lambda x^{\sigma}.e]\!]_\kappa & = f \quad \text{where } f\colon [\![\sigma]\!] \to [\![\tau]\!]\colon v \mapsto [\![e]\!]_{x\colon v, \kappa} \\
[\![f\ e]\!]_\kappa & = [\![f]\!]_\kappa\big([\![e]\!]_\kappa\big)
\end{array}
$$

Table 1. Semantics of $\mathcal{NTC}$.

can always be finitely represented. Moreover, the denotation of closed expressions e without free variables is independent of the context. We simply write $[\![e]\!]$ in that case.

Syntactic sugar. We will abbreviate $\lambda x.\,\mathsf{case}\ x\ \mathsf{of}\ \{(x_1, x_2) \to e\}$ by $\lambda(x_1, x_2).\,e$. Furthermore, we abbreviate $\mathsf{case}\ e\ \mathsf{of}\ \{(x, y) \to x\}$ and $\mathsf{case}\ e\ \mathsf{of}\ \{(x, y) \to y\}$ by $\pi_1(e)$ and $\pi_2(e)$, respectively. Similarly, we abbreviate $\mathsf{case}\ e\ \mathsf{of}\ \{\langle x\rangle\, y \to x\}$ by $\mathit{name}(e)$ and $\mathsf{case}\ e\ \mathsf{of}\ \{\langle x\rangle\, y \to y\}$ by $\mathit{children}(e)$. Also, we abbreviate the iteration $\mathsf{for}\ x\ \mathsf{in}\ \mathit{children}(e)\ \mathsf{return}\ \big(\mathsf{if}\ \mathit{name}(x) = a\ \mathsf{then}\ [\,x\,]\ \mathsf{else}\ [\,]\big)$ by e/a. Finally, we simulate general tuple construction by nested pairs. For example, we write (e_1, e_2, e_3) for $((e_1, e_2), e_3)$. General tuple inspection is defined similarly. For example, the expression $\mathsf{case}\ x\ \mathsf{of}\ \{(x_1, x_2, x_3) \to e\}$ is a shorthand for $\mathsf{case}\ x\ \mathsf{of}\ \{(y, x_3) \to \mathsf{case}\ y\ \mathsf{of}\ \{(x_1, x_2) \to e\}\}$.

Example 1. Let *friends* be a variable of type $[\mathsf{atom} \times \mathsf{atom}]$ whose value is a set of friends, as a list of pairs of atoms. The following closed function expression generates a list of trees, each tree grouping the friends of a single person.

$$
\begin{array}{l}
\lambda \mathit{friends}.\ \mathsf{for}\ x\ \mathsf{in}\ \mathit{friends}\ \mathsf{return}\ [\\
\quad \langle \pi_1(x) \rangle\ \ \mathsf{for}\ y\ \mathsf{in}\ \mathit{friends}\ \mathsf{return}\ \mathsf{if}\ \pi_1(x) = \pi_2(y)\ \mathsf{then}\ [\,\langle \pi_1(y) \rangle\,[\,]\,]\ \mathsf{else}\ [\,] \\
]
\end{array}
$$

□

3.2 Structural recursion operators

To $\mathcal{NTC}$ we add *structural recursion on lists* (*srl*) and *structural recursion on trees* (*srt*):

$$\frac{e\colon \tau \quad f\colon \sigma \times \tau \to \tau}{srl(e,f)\colon [\sigma] \to \tau} \qquad \frac{f\colon \mathsf{atom} \times [\tau] \to \tau}{srt(f)\colon \mathsf{tree} \to \tau}$$

Here, $[\![srl(e,f)]\!]_\kappa$ is the unique function that maps the empty list to $[\![e]\!]_\kappa$ and non-empty lists $u \uparrow v$ to $[\![f]\!]_\kappa(u, [\![srl(e,f)]\!]_\kappa(v))$. Similarly, $[\![srt(f)]\!]_\kappa$ is the unique function h defined by $h(\langle a\rangle\,[t_1,\ldots,t_n]) = [\![f]\!]_\kappa\big(a, [h(t_1),\ldots,h(t_n)]\big)$. We denote by $\mathcal{NTC}(V)$ the language obtained by adding operators in $V \subseteq \{srl, srt\}$ to $\mathcal{NTC}$.

Definition 1. *Let $V \subseteq \{srl, srt\}$ and let σ, τ be types. A query $q\colon \sigma \to \tau$ is* expressible *in $\mathcal{NTC}(V)$ if there exists a closed function expression $f\colon \sigma \to \tau$ in $\mathcal{NTC}(V)$ such that $q = [\![f]\!]$.*

Example 2. We can compute the transitive closure of a graph in $\mathcal{NTC}(srl)$ as follows. Let a directed graph G be represented by a pair (V, E) with V a list containing the nodes in G (represented by atoms) and E a list containing the edges in G (represented by pairs of atoms). Transitive closure is then expressed in $\mathcal{NTC}(srl)$ by $\lambda(V,E).\,\big(V, srl(E,f)(V)\big)$ with f the function expression

$$\begin{array}{l}\lambda(y, closure).\ closure +\!\!+\\ \quad \mathsf{for}\ x\ \mathsf{in}\ V\ \mathsf{return}\\ \quad\quad \mathsf{for}\ z\ \mathsf{in}\ V\ \mathsf{return}\\ \quad\quad\quad \mathsf{if}\ (x,y) \in closure\ \mathsf{and}\ (y,z) \in closure\ \mathsf{then}\ [(x,z)]\ \mathsf{else}\ [\,]\end{array}$$

Here, $(x,y) \in closure$ checks whether the edge (x,y) occurs in *closure*. It is an abbreviation of $srl(false, g)(closure) = true$ with g the expression:

$$\lambda(edge, res).\,\mathsf{if}\ edge = (x,y)\ \mathsf{then}\ \ true\ \mathsf{else}\ res.$$

Example 3. To express *toc* from the Introduction by means of *srt* we face a problem: in a computation of $srt(f)$ on a tree t the function expression f must compute the output based solely on the label of t and the recursive result on the children of t. To express *toc*, it is clear that f also needs to inspect the children of t themselves. This problem is solved by letting f return a pair of trees where the first component contains the actual table of contents (a list of trees) and the second component is t itself. Then *toc* is expressed in $\mathcal{NTC}(srt)$ by $\lambda t.\,\pi_1(srt(f)(t))$ where $f\colon \mathsf{atom} \times [[\mathsf{tree}] \times \mathsf{tree}] \to ([\mathsf{tree}] \times \mathsf{tree})$ is $\lambda(lab, res).\,(e_1, e_2)$. Here, e_1 is:

$$\begin{array}{l}\mathsf{for}\ x\ \mathsf{in}\ res\ \mathsf{return}\\ \quad \mathsf{case}\ x\ \mathsf{of}\ \{(stoc, s) \to\\ \quad\quad \mathsf{if}\ name(s) = \texttt{section}\ \mathsf{then}\ [\,\langle\texttt{section}\rangle\,(s/\texttt{title} +\!\!+ stoc)\,]\ \mathsf{else}\ [\,]\,\}\end{array}$$

and e_2 is $\langle lab\rangle\,\mathsf{for}\ x\ \mathsf{in}\ res\ \mathsf{return}\ [\pi_2(x)]$. □

4 Expressive power

4.1 Primitive recursion

In this section we investigate the class of queries expressible in $\mathcal{NTC}(srl, srt)$. As a first result we have:

Theorem 1. *A query is expressible in $\mathcal{NTC}(srl, srt)$ if, and only if, it is computable in primitive recursive time.*

This result is slightly stronger than that of Immerman et al. [21]; Suciu and Wong [27]; and Libkin and Wong [24], who have shown that, in the presence of object invention, the class of functions $f\colon \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ representable with structural recursion on (nested) sets and bags, coincides with the class of primitive recursive functions on natural numbers [6]. Indeed, if we fix a representation of natural numbers as values, then the theorem above implies that the class of functions $f\colon \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ representable in $\mathcal{NTC}(srl, srt)$ coincides with the class of primitive recursive functions, as it is known that the primitive recursive functions are exactly those functions on the natural numbers that can be computed in primitive recursive time. Note, however, that their results do not necessarily imply that the class of expressible *queries* coincides with the class of primitive recursive time queries.

Theorem 1 shows that the combination of structural recursion on lists and trees taken together gives rise to a robust class of queries. Unfortunately, the expressiveness drops dramatically when we consider structural recursion on lists or trees separately. Indeed, let $\mathit{lastlab}\colon \mathsf{tree} \to \mathsf{atom}$ be the query that maps its input tree t to the label of the last node visited when traversing t in pre-order. This query is clearly computable in linear time. Nevertheless:

Theorem 2. *The query lastlab is inexpressible in both $\mathcal{NTC}(srl)$ and $\mathcal{NTC}(srt)$. Hence, structural recursion on lists or trees alone is not strong enough to express all linear time queries.*

Intuitively, this is because *srl* only provides "horizontal" recursion along lists, while *srt* only provides "vertical" recursion down trees. As such, $\mathcal{NTC}(srl)$ can only manipulate inputs up to bounded depth, while $\mathcal{NTC}(srt)$ can only manipulate inputs up to bounded width.

4.2 Taming Structural Recursion

From a complexity point of view, it follows from Theorem 1 that $\mathcal{NTC}(srl, srt)$ is too powerful a query language. In this section we investigate intrinsic restrictions on structural recursion that capture exactly the polynomial time queries. We start with a semantical restriction, from which we next derive a suitable syntactical restriction.

Let us refer to the function expressions g in $srl(e, g)$ or $srt(g)$ as *step expressions*. It is clear that, in order for the function expressed by a function

expression f to be computable in polynomial time, f should never create intermediate results of more than polynomial size. This condition is trivially satisfied if f does not use structural recursion. To see how structural recursion can create results of exponential size or more, consider the function expression $\mathit{explist} := \mathit{srl}([a], \lambda(x, y).\, y +\!\!+ y)$. It is clear that, if v is a list of length k, then $[\![\mathit{explist}]\!](v)$ returns a list of length 2^k. As Caseiro [9] was the first to note, the problem here is that the step expression $\lambda(x, y).\, y +\!\!+ y$ *doubles* the size of the result at each recursive invocation. A similar problem arises with structural tree recursion. Indeed, consider $\mathit{exptree} := \mathit{srt}(\lambda(x, y).\, \langle x \rangle\, y +\!\!+ y)$. It is clear that, if v is a linear tree (i.e., a tree in which each node has at most one child) of depth k, then $[\![\mathit{exptree}]\!](v)$ returns a tree of size 2^k. Again, the problem is that the step expression $\lambda(x, y).\, \langle x \rangle\, y +\!\!+ y$ of *exptree* doubles its result at each recursive invocation. This leads us to the following definitions.

Definition 2 (Tamed expressions). *A function expression $f\colon \sigma \times \sigma' \to \tau$ is* non-multiplying *(in its second argument) if there exists a polynomial P such that for all contexts κ; all $v\colon \sigma$; and all $w\colon \sigma'$, the size of $[\![f]\!]_\kappa(v, w)$ is bounded by*

$$P\left(\mathbf{s}\,(v) + \sum_{x \in FV(f)} \mathbf{s}\,(\kappa(x))\right) + \mathbf{s}\,(w)\,.$$

An expression $e \in \mathcal{NTC}(\mathit{srl}, \mathit{srt})$ is tamed *if every step expression occurring in it is non-multiplying.*

Clearly, *explist* and *exptree* are not tamed. The following proposition shows that being tamed is a strong enough restriction to ensure polynomial time computability.

Proposition 1. *Every tamed function expression in $\mathcal{NTC}(\mathit{srl}, \mathit{srt})$ expresses a polynomial time query.*

Proof (Crux). The proof proceeds by induction on tamed expressions. We only illustrate the reasoning involved in showing that tamed *srl* and *srt* expressions can be computed in polynomial time, as these are the hard cases.

First, consider a closed function expression f of the form $\mathit{srl}(e, f')$ with e and f' also closed. Assume by induction that $[\![f']\!]$ is computable in polynomial time T'. Since $[\![f']\!]$ is non-multiplying, there exists a polynomial P such that $\mathbf{s}\,([\![f']\!](v, w)) \leq P(\mathbf{s}\,(v)) + \mathbf{s}\,(w)$ for all v and w of the correct type. We assume without loss of generality that T' and P are monotone increasing. To compute $[\![f]\!](v)$ for a given list $v = [w_1, \ldots, w_m]$ of size n we first compute $w = [\![e]\!]$. Since e is closed, this can be done in constant time. Next, we compute $[\![f']\!](w_1, [\![f']\!](w_2, \ldots [\![f']\!](w_m, w) \ldots))$. In order to do so, we need to evaluate $[\![f']\!]$ at most $m \leq n$ times. Every w_i has size at most n and the size of w is some constant c. Because $[\![f']\!]$ is non-multiplying, $[\![f']\!](w_m, w)$ then has size at most $P(n) + c$; $[\![f']\!](w_{m-1}, [\![f']\!](w_m, w))$ has size at most $P(n) + P(n) + c$; and so on. The maximum size of an input to f' is hence bounded by $n \times P(n) + c$. The total

time needed to compute $[\![f]\!](v)$ is then bounded by $O(n \times T'(n \times P(n) + c))$, which is clearly a polynomial in n.

Next, consider a closed function expression f of the form $srt(f')$ with f' also closed. Assume that $[\![f']\!](v, w)$ can be computed in polynomial time T'. Since $[\![f']\!]$ is non-multiplying, there exists a polynomial P such that $\mathbf{s}([\![f']\!](v, w)) \leq P(\mathbf{s}(v)) + \mathbf{s}(w)$. Again, we assume without loss of generality that T' and P are monotone increasing. Using the fact that $[\![f']\!]$ is non-multiplying, it is straightforward to prove by induction on a tree t that $\mathbf{s}([\![f]\!](t)) \leq \mathbf{s}(t) \times (P(1) + 2)$. To compute $[\![f]\!](t)$ for a given input tree $t = \langle a \rangle [t_1, \ldots, t_m]$ of size n we must compute $[\![f']\!](a, [[\![f]\!](t_1), \ldots, [\![f]\!](t_m)])$. Hence, we first need to compute $[\![f]\!](t_i)$ for every i. This involves calling $[\![f']\!]$ again multiple times. Note, however, that the total number of times that $[\![f']\!]$ gets called is bounded by n. Furthermore, at each such call, the size of the input to $[\![f']\!]$ is bounded by $n \times (P(1)+2)$. The total time needed to compute $[\![f]\!](t)$ is hence bounded by $O(n \times T'(n \times (P(1) + 2)))$, which is clearly a polynomial in n. □

The converse is also true: every polynomial time query can be expressed by a tamed function expression, as we will show below. Note that "non-multiplying" and "tamed" are *semantical* notions. Using a standard reduction from the satisfiability problem of the relational algebra, it is straightforward to show that checking whether an expression satisfies one of these semantical properties is undecidable. We can, however, restrict the syntax of expressions in $\mathcal{NTC}(srl, srt)$ in such a way that all expressions are tamed, as we shown next.

To motivate our syntactical restriction, consider again the problematic step expression $\lambda(x, y).\, y +\!\!+ y$ from *explist*. Since this step expression is multiplying (and *explist* is hence not tamed), we want our syntactical restriction to exclude it. The first solution that comes to mind is to require that y occurs at most once in the body e of a step expression $\lambda(x, y).e$. This solution is defective in multiple ways. On the one hand it is too restrictive. Indeed, harmless, non-multiplying step expressions like $\lambda(x, y).\, \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ x \uparrow y\ \mathsf{else}\ y$ with y occurring in e_1 or e_2 are excluded. Clearly, there is a difference between testing a variable and actually using it to construct the output. On the other hand, the solution is not restrictive enough. Indeed, the step expression, $\lambda(x, y).\, \mathsf{for}\ x\ \mathsf{in}\ [a, b]\ \mathsf{return}\ y$ would be accepted, although it is equivalent to the problematic $\lambda(x, y).\, y +\!\!+ y$ above. For these reasons, a more fine-grained restriction is in order.

Definition 3 (Testing and outputting). *An expression e* tests *a variable x if every free occurrence of x as a subexpression in e is in e_1 or e_2 of a conditional test $\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4$. An expression e* outputs *x if x is free in e and e does not test x.*

Example 4. The expression $\mathsf{if}\ x = y\ \mathsf{then}\ (y, z)\ \mathsf{else}\ (z, z)$ tests x and outputs y and z. □

Next, we define linearity. Here, linearity should be understood in the sense of Caseiro [9] and Hofmann [18]: if e is linear in a variable x, then e uses x to compute its output at most once.

Definition 4 (Linearity). *A value expression in $\mathcal{NTC}(srl, srt)$ is* linear *in a variable x if either*

- *it is an expression of the form y, a, or $[\,]$;*
- *it is a conditional test if $e_1 = e_2$ then e_3 else e_4 with e_3 and e_4 linear in x;*
- *it is $[e]$ or $\langle e' \rangle\, e$ with e linear in x;*
- *it is (e, e') or $e +\!\!+ e'$ with e and e' linear in x and at most one of e and e' outputting x;*
- *it is a case expression of the form case e' of $\{(y, z) \to e'_2\}$, case e' of $\{[\,] \to e'_1;\ y \uparrow z \to e'_2\}$, or case e' of $\{\langle y \rangle\, z \to e'_2\}$ with (1) e', e'_1, and e'_2 linear in x; and (2) if e' outputs x, then e'_2 tests x and e'_2 is linear in y and z;*
- *it is for y in e_1 return e_2 with e_1 and e_2 testing x;*
- *it is for y in e_1 return y with e_1 linear in x; or*
- *it is for y in e_1 return $children(y)$ with e_1 linear in x.*

We clarify this definition with some examples.

Example 5. The expression $y +\!\!+ y$ is not linear in y. The expression from Example 4 is linear in x and y, but not in z. The expression for y in x return (if $y = z$ then $[y]$ else $[\,]$) is linear in z, but not in x. The expression case x of $\{(y, z) \to (z, y)\}$ is linear in x. The expression e_1 from Example 3 is not linear in the variable *res* because the for-loop does not have the required form. Finally, the expression e from Example 2 is linear in x. □

Definition 5 (Safety). *An expression in $\mathcal{NTC}(srl, srt)$ is* safe *if every step expression occurring in it is of the form $\lambda(x, y).\, e$ with e linear in y.*

From Example 5 above, it follows that the function expression computing the transitive closure of a graph given in Example 2 is safe, whereas the function expression computing the table of contents of a book given in Example 3 is not.

Lemma 1. *If $e \in \mathcal{NTC}(srl, srt)$ is a value expression linear in x then there exists a polynomial $P\colon \mathbb{N} \to \mathbb{N}$ such that for all environments κ:*

$$\mathbf{s}\left([\![e]\!]_\kappa\right) \leq P\left(\sum_{y \in FV(e) - \{x\}} \mathbf{s}\left(\kappa(y)\right)\right) + \mathbf{s}\left(\kappa(x)\right).$$

It immediately follows that safe expressions are tamed; they are hence computable in polynomial time by Proposition 1. Note, however, that some function expressions, like the one expressing *toc* from the Introduction in Example 3 denote polynomial time queries, but are not safe. This hence raises the question how powerful safe expressions are. Fortunately,

Proposition 2. *Every polynomial time query is expressible by a safe, closed function expression in $\mathcal{NTC}(srl, srt)$.*

In particular, *toc* from Example 3 can hence be expressed in a safe way. From Lemma 1 and Propositions 1 and 2 it immediately follows that safe expressions provide an effective syntax for the polynomial time queries.

Theorem 3. *A query is expressible in safe $\mathcal{NTC}(srl, srt)$ if, and only if, it is computable in polynomial time.*

5 Natural Sublanguages

Note that the results of Section 4 do not necessarily imply anything about the expressiveness of structural recursion in XQuery or about the expressiveness of structural recursion on list-based complex objects. Indeed, the expressions of $\mathcal{NTC}(srl, srt)$ can create and manipulate arbitrary values (including e.g., lists of lists and list of pairs) during their computation, while XQuery only manipulate XQuery values (i.e., values in some xq-type). Conversely, the expressions of $\mathcal{NTC}(srl, srt)$ can create and manipulate trees, while trees are not present in complex object data models. Nevertheless, the results for $\mathcal{NTC}(srl, srt)$ transfer cleanly to both structural recursion in XQuery and structural recursion on list-based complex objects, as we show in this section.

Let us define *structural recursive XQuery* to be the natural sublanguage of $\mathcal{NTC}(srl, srt)$ in which we restrict expressions to only manipulate XQuery values. Since we still want to be able to define and call multiple-argument functions however, we do allow to create and manipulate tuples of XQuery values, but only in function abstraction and application.

Definition 6 (Structural recursive XQuery). *If $V \subseteq \{srl, srt\}$ then $\mathcal{XQ}(V)$ is the set of expressions $e \in \mathcal{NTC}(V)$ in which every subexpression e' of e has type either $e' : \tau$ or $e' : \sigma \times \cdots \times \sigma' \to \tau$ with $\sigma, \ldots, \sigma', \tau$ xq-types, except when e' is a variable $x^{\sigma \times \cdots \times \sigma'}$ in a function abstraction $\lambda x.\, \mathsf{case}\ x\ \mathsf{of}\ \{(y, \ldots, y') \to e''\}$ or e' is a product $(e_1, \ldots, e_n)$ in a function application $f\, e'$.*

The function expression from Example 1 is not in $\mathcal{XQ}(srl, srt)$ as the subexpression *friends* has type $[\mathsf{atom} \times \mathsf{atom}]$, which is not an xq-type. The expression $\lambda x^{\mathsf{tree} \times \mathsf{tree}}.\, \mathsf{case}\ x\ \mathsf{of}\ \{(y, z) \to \langle name\, y \rangle\ children\, z\}$ which we would normally abbreviate by $\lambda(y^{\mathsf{tree}}, z^{\mathsf{tree}}).\, \langle name(y) \rangle\ children(z)$ does belong to $\mathcal{XQ}(srl, srt)$, however. Unfortunately, the function expression $\lambda(lab, res).\, (e_1, e_2)$ from Example 3 that is used to simulate *toc* from the Introduction is not in $\mathcal{XQ}(srl, srt)$. Indeed, the subexpression (e_1, e_2) creates a pair without directly giving it as argument to a function. Nevertheless, *toc* is expressible in $\mathcal{XQ}(srt)$, as Proposition 3 below shows.

The structural recursive complex object calculus is the natural sublanguage of $\mathcal{NTC}(srl, srt)$ in which we restrict expressions to only manipulate complex objects. Such expressions hence cannot create or manipulate trees. In particular, they cannot recur on trees.

Definition 7 (Complex object calculus). *If $V \subseteq \{srl\}$, then $\mathcal{COC}(V)$ is the subset of expressions e in $\mathcal{NTC}(V)$ in which every subexpression e' of e has type either $e' : \tau$ or $e' : \sigma \to \tau$ with σ and τ complex object types.*

The Nested Tree Calculus is a conservative extension of both XQuery and the complex object calculus, as the following proposition shows.

Proposition 3. *Let $V \subseteq \{srl, srt\}$.*

1. *An xquery is expressible in $\mathcal{NTC}(V)$ if, and only if, it is expressible in $\mathcal{XQ}(V)$.*

2. *An xquery is expressible by a safe expression in* $\mathcal{NTC}(V)$ *if, and only if, it is expressible by a safe expression in* $\mathcal{XQ}(V)$.
3. *A complex object query is expressible in* $\mathcal{NTC}(srl, srt)$ *if, and only if, it is expressible in* $\mathcal{COC}(srl)$.
4. *A complex object query is expressible by a safe expression in* $\mathcal{NTC}(srl, srt)$ *if, and only if, it is expressible by a safe expression in* $\mathcal{COC}(srl)$.

It follows that our results about the expressiveness of (safe) structural recursion in $\mathcal{NTC}$ as stated in Theorems 1,2, and 3 transfer to $\mathcal{XQ}$ and $\mathcal{COC}$.

In particular, a complex object query is hence expressible in $\mathcal{COC}(srl)$ if, and only if, it is primitive recursive. We note that this result may seem in contrast to that of Grumbach and Milo [17], who consider a language that includes structural recursion on pomsets (a datatype that generalizes sets, bags, and lists), which is claimed to capture the elementary queries on pomsets. It seems counter-intuitive that a language that generalizes $\mathcal{COC}(srl)$ has lower complexity. There is an error in their upper-bound proof, however; also non-elementary queries can be expressed [16].

We also note that the polynomial time queries on list-based complex objects have already been captured by means of the *list-trav* iteration construct of Colby et al. [13]. This iteration construct is rather awkward, however, and we think that safe structural recursion provides an elegant alternative.

Another such alternative in the restricted case of list of atomic values was given by Bonner and Mecca, in their work on *Sequence Datalog* [5]. Sequence Datalog is a query language that extends Datalog with functions on lists of atomic values. Using suitable syntactic restrictions, they give a query language sound and complete for the *flat* relational queries. In these relations, tuple components may either contain atomic values or lists of atomic values.

References

1. S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, 1995.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations Of Databases*. Addison-Wesley, 1995.
3. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *STOC 1992*, pages 283–293. ACM Press, 1992.
4. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C Candidate Recommendation, November 2005.
5. A. J. Bonner and G. Mecca. Sequences, datalog, and transducers. *J. Comput. Syst. Sci.*, 57(3):234–259, 1998.
6. G. Boolos and R. Jeffrey. *Computability and Logic*. Cambridge University Press, third edition, 1989.
7. P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

8. P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Comput. Sci.*, 149(1):3–48, 1995.
9. V. Caseiro. *Equations for Defining Poly-time Functions*. PhD thesis, University of Oslo, 1997.
10. A. K. Chandra and D. Harel. Computable queries for relational data bases. *J. Comput. Syst. Sci.*, 21(2):156–178, 1980.
11. A. Cobham. The intrinsic computational difficulty of functions. In *Logic, Methodology, and Philosophy of Science II*, pages 24–30. Springer Verlag, 1965.
12. L. S. Colby and L. Libkin. Tractable iteration mechanisms for bag languages. In *ICDT 1997*, volume 1186 of *LNCS*, pages 461–475. Springer, 1997.
13. L. S. Colby, E. L. Robertson, L. V. Saxton, and D. V. Gucht. A query language for list-based complex objects. In *PODS 1994*, pages 179–189. ACM Press, 1994.
14. D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Candidate Recommendation, June 2006.
15. M. F. Fernández, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In *ICDT 2001*, volume 1973 of *LNCS*, pages 263–300. Springer, 2001.
16. S. Grumbach and T. Milo. Personal communication.
17. S. Grumbach and T. Milo. An algebra for pomsets. *Inf. Comput.*, 150(2):268–306, 1999.
18. M. Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *CSL 1997*, volume 1414 of *LNCS*, pages 275–294. Springer, 1997.
19. M. Hofmann. Semantics of linear/modal lambda calculus. *Journal of Functional Programming*, 9(3):247–277, 1999.
20. R. Hull and J. Su. Algebraic and calculus query languages for recursively typed complex objects. *J. Comput. Syst. Sci.*, 47(1):121–156, 1993.
21. N. Immerman, S. Patnaik, and D. W. Stemple. The expressiveness of a family of finite set languages. *Theor. Comput. Sci.*, 155(1):111–140, 1996.
22. C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. In *PODS 2005*, pages 84–97. ACM, 2005.
23. D. Leivant. Stratified functional programs and computational complexity. In *POPL 1993*, pages 325–333. ACM Press, 1993.
24. L. Libkin and L. Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.
25. V. Y. Sazonov. Hereditarily-finite sets, data bases and polynomial-time computability. *Theor. Comput. Sci.*, 119(1):187–214, 1993.
26. D. Suciu. Bounded fixpoints for complex objects. *Theor. Comput. Sci.*, 176(1-2):283–328, 1997.
27. D. Suciu and L. Wong. On two forms of structural recursion. In *ICDT 1995*, volume 893 of *LNCS*, pages 111–124. Springer, 1995.
28. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*. W3C Recommendation, May 2001.
29. J. Van den Bussche, D. Van Gucht, and S. Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus and xquery. In *ICDT 2005*, volume 3363 of *LNCS*, pages 99–113. Springer, 2005.