Theory Comput Syst (2009) 44: 590–619
DOI 10.1007/s00224-008-9110-5

Structural Recursion as a Query Language on Lists and Ordered Trees

Edward L. Robertson · Lawrence V. Saxton · Dirk Van Gucht · Stijn Vansummeren

Published online: 15 April 2008

Abstract XML query languages need to provide some mechanism to inspect and manipulate nodes at all levels of an input tree. We investigate the expressive power provided in this regard by structural recursion. In particular, 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. We also give corresponding results for list-based complex objects.

Keywords Structural recursion · Primitive recursion · XML · Complex objects · Database

S. Vansummeren is a postdoctoral Fellow of the Research Foundation—Flanders (FWO).

E.L. Robertson · D. Van Gucht
Indiana University, Bloomington, IN, USA

E.L. Robertson
e-mail: edrbtsn@cs.indiana.edu

D. Van Gucht
e-mail: vgucht@cs.indiana.edu

L.V. Saxton
University of Regina, Regina, SK, Canada
e-mail: saxton@cs.uregina.ca

S. Vansummeren (✉)
Hasselt University and Translational University of Limburg, 3560 Diepenbeek, Belgium
e-mail: stijn.vansummeren@uhasselt.be

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 developed 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>
```

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 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*:

```
function rev(l) { if empty(l) then l 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. As in XQuery, we assume in this example that the single item returned by $hd(l)$ is the same as the singleton list containing that item.

In this article, 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, 19, 25, 27]; 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 [24] (i.e., the queries with hyper-exponential time data complexity). Furthermore, Immerman et al. [25] and Suciu and Wong [30] have shown that, in the presence of object invention, the class of functions $F: \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ 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 [27].

Since tree construction is a form of object invention, it should come as no surprise that a similar result also holds 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 $F: \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ 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 [25]. 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 [19]. 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:

```
function $explist(l)$ {if $empty(l)$ then $[1]$ else $explist(tl(l)),explist(tl(l))$};
```

Here, [1] constructs the singleton list containing the integer 1. 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:

```
function $explist'(l, b)$ {
    if $empty(l)$ then $l$ else
        let $r = explist'(tl(l), b),explist'(tl(l), b)$ in
        if $sizeof(r) \leq sizeof(b)$ then $r$ else $explist'(tl(l), b)$
};
```

In particular, the size of $explist'(l, b)$ 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, 19] 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, 28, 29].

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 [26]. 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 [20, 22].

In this article, 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 use it to derive an effective syntax.

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 Language* $\mathcal{NTL}$. This language can be viewed both as (1) the natural extension of non-recursive for-let-where-return XQuery with complex objects (i.e., tuples and nested lists) and (2) a list-based interpretation of the nested relational language of Buneman et al [8] extended with ordered trees. Apart from being theoretically interesting in its own right, we show $\mathcal{NTL}$ to be a *conservative* extension of both original languages, even in the presence of (restricted) structural recursion. As a consequence, results about (restricted) structural recursion in $\mathcal{NTL}$ 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 article is further organized as follows. We start with the definition of $\mathcal{NTL}$ in Sect. 2 and study the expressive power of structural recursion in $\mathcal{NTL}$ in Sect. 3. In Sect. 4 we give intrinsic restrictions of structural recursion that capture polynomial time. Finally, we show how these results transfer to structural recursion in XQuery and languages for nested lists in Sect. 5, and conclude in Sect. 6.

2 The Nested Tree Language

Let us first introduce the *nested tree language* $\mathcal{NTL}$, our ambient query language throughout the article. One can view $\mathcal{NTL}$ both as (1) the extension of non-recursive for-let-where-return XQuery with tuples and arbitrary nesting of lists; and (2) a list-based interpretation of the nested relational language of Buneman et al. [8] extended with ordered trees. In fact, we will show in Section 5 that $\mathcal{NTL}$ is a *conservative* extension of these languages, even in the presence of structural recursion. As a consequence, our results on and restrictions of structural recursion in $\mathcal{NTL}$ immediately transfer to the respective sublanguages.

General Operators

$$Ka\colon unit \to atom \qquad id^s\colon s \to s \qquad \frac{h\colon r \to s \quad g\colon s \to t}{g \circ h\colon r \to t}$$

$$!^s\colon s \to unit \qquad \pi_1^{s,t}\colon s \times t \to s \qquad \pi_2^{s,t}\colon s \times t \to t \qquad \frac{h\colon r \to s \quad g\colon r \to t}{\langle g, h\rangle\colon r \to s \times t}$$

List Operators

$$sng^s\colon s \to [s] \qquad flatten^s\colon [[s]] \to [s] \qquad \texttt{[]}^s\colon unit \to [s]$$

$$+\!\!+^s\colon [s] \times [s] \to [s] \qquad \rho_1^{s,t}\colon [s] \times t \to [s \times t] \qquad \rho_2^{s,t}\colon s \times [t] \to [s \times t]$$

$$\frac{f\colon s \to t}{map(f)\colon [s] \to [t]} \qquad head^s\colon [s] \to s \qquad tail^s\colon [s] \to [s]$$

Tree Operators

$$tree\colon atom \times [tree] \to tree \qquad lab\colon tree \to atom \qquad children\colon tree \to [tree]$$

Conditional

$$cond^{s,t}\colon s \times s \times t \times t \to t$$

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

$\mathcal{NTL}$ operates on nested combinations of pairs, trees, and lists, whose types are given by the following grammar:

$$s, t ::= unit \mid atom \mid tree \mid s \times t \mid [s].$$

Types denote sets of *values*. The type *unit* consists only of the empty tuple (); values of type *atom* are atomic data values like the integers, the strings, and so on; values of type *tree* are finite trees $tree(a, v)$ with a the tree's label and v the finite list of its child trees; values of $s \times t$ are pairs (v, w) with v and w of type s and t, respectively; and values of $[s]$ are finite lists $[v, \ldots, w]$ of values, each of type s. We write $v\colon s$ to indicate that v is a value of type s. Furthermore, we feel free to omit parentheses and write $s_1 \times \cdots \times s_n$ for $s_1 \times (s_2 \times (\cdots \times s_n)\ldots)$ and similarly $(v_1, \ldots, v_n)$ for $(v_1, (v_2, (\ldots, (v_{n-1}, v_n)\ldots)))$.

The nested tree language $\mathcal{NTL}$ consists of two equally expressive components: the *nested tree algebra* $\mathcal{NTA}$ and the *nested tree calculus* $\mathcal{NTC}$. We will see that the $\mathcal{NTA}$ component is convenient in proving upper bounds of the language, while the $\mathcal{NTC}$ component is more convenient to program in. The expressions of both components are explicitly typed and are formed according to the typing rules of Figs. 1 and 2, respectively. There, we use g and h to range over $\mathcal{NTA}$ expressions, and e to range over $\mathcal{NTC}$ expressions. We will often omit the explicit type annotation in superscript when they are clear from the context.

Lambda Calculus, Products, and Conditional

$$a\colon \mathit{atom} \qquad x^s\colon s \qquad \frac{e\colon t}{\lambda x^s.\, e\colon s \to t} \qquad \frac{e_1\colon s \to t \quad e_2\colon s}{e_1\; e_2\colon t} \qquad ()\colon \mathit{unit}$$

$$\frac{e\colon s \times t}{\pi_1\, e\colon s \quad \pi_2\, e\colon t} \qquad \frac{e_1\colon s \quad e_2\colon t}{(e_1, e_2)\colon s \times t} \qquad \frac{e_1\colon s \quad e_2\colon s \quad e_t\colon t \quad e_f\colon t}{\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f\colon t}$$

List Expressions

$$\texttt{[]}\colon [s] \qquad \frac{e\colon s}{\texttt{[}e\texttt{]}\colon [s]} \qquad \frac{e_1\colon [s] \quad e_2\colon [s]}{e_1 \mathbin{+\!\!+} e_2\colon [s]}$$

$$\frac{e\colon [s]}{\mathit{head}(e)\colon s \quad \mathit{tail}(e)\colon [s]} \qquad \frac{e_1\colon [s] \quad e_2\colon [t]}{\mathsf{for}\ x^s\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\colon [t]}$$

Tree Expressions

$$\frac{e_1\colon \mathit{atom} \quad e_2\colon [\mathit{tree}]}{\mathit{tree}(e_1, e_2)\colon \mathit{tree}} \qquad \frac{e\colon \mathit{tree}}{\mathit{lab}(e)\colon \mathit{atom} \quad \mathit{children}(e)\colon [\mathit{tree}]}$$

Fig. 2 Expressions of $\mathcal{NTC}$

Semantics of $\mathcal{NTA}$ Every expression $f\colon s \to t$ of $\mathcal{NTA}$ defines a function from s to t. The expression Ka is the constant function that always produces the atom a; *id* is the identity function; and $g \circ h$ is function composition, i.e., $(g \circ h)(v) = g(h(v))$. Then follow the pair operators: ! produces () on all inputs; π_1 and π_2 are respectively the left and right projections; and $\langle g, h\rangle$ is pair formation: $\langle g, h\rangle(v) = (gv, hv)$. Next come the list operators: *sng* forms singletons: $\mathit{sng}(v) = [v]$; *flatten* flattens lists of lists: $\mathit{flatten}([v, \dots, w]) = v \mathbin{+\!\!+} \dots \mathbin{+\!\!+} w$; `[ ]` is the constant function that always produces the empty list; $\mathbin{+\!\!+}$ is list concatenation; ρ_1 and ρ_2 are respectively the left and right tensor product: $\rho_1([u, \dots, v], w) = [(u, w), \dots, (v, w)]$ and $\rho_2(u, [v, \dots, w]) = [(u, v), \dots, (u, w)]$; and $\mathit{map}(f)$ applies f to every object in its input list: $\mathit{map}(f)([v, \dots, w]) = [f(v), \dots, f(w)]$. The functions *head* and *tail* retrieve the head and tail of a list, respectively: $\mathit{head}([u, v, \dots, w]) = u$ and $\mathit{tail}([u, v, \dots, w]) = [v, \dots, w]$. For simplicity, we take the head of the empty list to be an arbitrarily fixed value, and the tail of the empty list to be the empty list itself. Then come the tree operators: *tree* constructs a new tree given its label and list of child trees; *lab* retrieves the label of a tree: $\mathit{lab}(\mathit{tree}(a, v)) = a$; and *children* retrieves the children of a tree: $\mathit{children}(\mathit{tree}(a, v)) = v$. Finally, *cond* is the conditional that, when applied to a tuple (v, v', w, w') returns w if $v = v'$, and returns w' otherwise.

Example 1 Here are some simple examples of the functions definable in $\mathcal{NTA}$. The projections $\Pi_1\colon [s \times t] \to [s]$ and $\Pi_2\colon [s \times t] \to [t]$ on list of pairs are given by $\Pi_1 := \mathit{map}(\pi_1)$ and $\Pi_2 := \mathit{map}(\pi_2)$, respectively. The function $\mathit{swap}\colon s \times t \to t \times s$ that swaps the components of a pair is given by $\mathit{swap} := \langle \pi_2, \pi_1\rangle$. The Cartesian product $\mathit{prod}\colon [s] \times [t] \to [s \times t]$ of two lists is then given by $\mathit{prod} := \mathit{flatten} \circ \mathit{map}(\rho_1) \circ \rho_2$.

We should note that in principle only one of ρ_1 or ρ_2 is necessary, as they are interdefinable. For instance $\rho_2 = \mathit{map}(\mathit{swap}) \circ \rho_1 \circ \mathit{swap}$. We have chosen to include them both here for reasons of symmetry.

Semantics of $\mathcal{NTC}$ $\mathcal{NTC}$ has two sorts of expressions: *value expressions* and *function expressions*. Value expressions $e\colon t$ denote values of type t, while function expressions $e\colon s \to t$ denote first-order functions from s to t. In particular, the semantics of $\mathcal{NTC}$ is that of the first-order, simply typed lambda calculus with products, lists, and trees. As such, expression a denotes the constant atom a; x^s is the explicitly typed variable that can be bound to values of type s; $\lambda x.e$ is standard lambda abstraction; and $e_1 e_2$ is function application. Furthermore, expression () denotes the empty tuple; $\pi_1 e$ and $\pi_2 e$ are respectively the left and right projection on pairs; and (e_1, e_2) is pair construction. The conditional expression $\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f$ returns e_t if the denotations of e_1 and e_2 are equal and returns e_f otherwise. Expression $[\ ]$ denotes the empty list; $[e]$ is singleton list construction; $e_1 +\!\!+ e_2$ is list concatenation; $\mathit{head}(e)$ and $\mathit{tail}(e)$ return the head and tail of a list, respectively; and $\mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2$ is list comprehension. That is, $\mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 = f(v) +\!\!+ \cdots +\!\!+ f(v')$ where $f = \lambda x.e_2$ and e_1 denotes $[v, \ldots, v']$. Finally, the expression $\mathit{tree}(e_1, e_2)$ is tree construction and $\mathit{lab}(e)$ and $\mathit{children}(e)$ are label and children extraction, respectively.

Example 2 Here are some simple examples of the functions definable in $\mathcal{NTC}$. The projection $\Pi_1\colon [s \times t] \to [t]$ is given by $\lambda x.\mathsf{for}\ y\ \mathsf{in}\ x\ \mathsf{return}\ [\pi_1 y]$. The projection $\Pi_2\colon [s \times t] \to [t]$ is defined similarly. The Cartesian product $\mathit{prod}\colon [s] \times [t] \to [s \times t]$ of two lists is given by

$$\lambda x.\mathsf{for}\ y\ \mathsf{in}\ \pi_1(x)\ \mathsf{return}\ \mathsf{for}\ z\ \mathsf{in}\ \pi_2(x)\ \mathsf{return}\ [(y, z)].$$

The flattening function $\mathit{flatten}\colon [[s]] \to [s]$ is given by $\lambda x.\mathsf{for}\ y\ \mathsf{in}\ x\ \mathsf{return}\ y$. Finally, the function $\mathit{fltr}\colon [\mathit{tree}] \times \mathit{atom} \to [\mathit{tree}]$ such that $\mathit{fltr}(v, a)$ filters from its input list v all trees whose root node is labeled a is given by

$$\lambda x.\mathsf{for}\ y\ \mathsf{in}\ \pi_1(x)\ \mathsf{return}\ \mathsf{if}\ \mathit{lab}(y) = \pi_2(x)\ \mathsf{then}\ [y]\ \mathsf{else}\ [\ ].$$

Lambda abstraction and the for loop act as binders in $\mathcal{NTC}$ expressions. The free variables $FV(e)$ of an $\mathcal{NTC}$ expression e are hence inductively defined as follows: $FV(x) = \{x\}$; $FV(\lambda x.e) = FV(e) - \{x\}$; $FV(\mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2) = FV(e_1) \cup (FV(e_2) - \{x\})$; and $FV(e)$ is the union of the free variables of e's immediate subexpressions, otherwise. As usual, an expression without free variables is called *closed*.

Definition 3 A *query* of type $s \to t$ is nothing more than a function mapping values of type s to values of type t. We say that a query is *expressible* in $\mathcal{NTA}$ if there exists an expression $f\colon s \to t$ that defines it. Similarly, we say that a query is *expressible* in $\mathcal{NTC}$ if there exists a closed expression $e\colon s \to t$ that defines it.

The following proposition shows that $\mathcal{NTA}$ and $\mathcal{NTC}$ are equally expressive. Its proof is an easy extension of the proof that the nested relational algebra and the nested relational calculus of Buneman et al. are equivalent [8, theorem 3.1] and [29, proposition 2.5].

Proposition 4 *$\mathcal{NTA} \equiv \mathcal{NTC}$ in the sense that every query definable by an expression $f: s \to t$ in $\mathcal{NTA}$ is definable by a closed expression $e: s \to t$ in $\mathcal{NTC}$, and vice versa.*

Since both languages are equally expressive, we can conceptually view them as a *single* language: the nested tree language $\mathcal{NTL} = \mathcal{NTC} + \mathcal{NTA}$. This also implies that we can view expressions in both languages as "syntactic sugar" for expressions in the other language whenever convenient. For instance, we can simply write $e_2 \circ e_1$ for the composition of the closed $\mathcal{NTC}$ expressions $e_1: r \to s$ and $e_2: s \to t$. Similarly, we can write $f\ e$ for the application of $\mathcal{NTA}$ expression $f: s \to t$ to the value denoted by $\mathcal{NTC}$ expression $e: s$.

Notational Convention To aid readability we will use a more liberal notation in lambda abstractions. For instance, we write $swap: s \times t \to t \times s$ as $\lambda(x^s, y^t).(y^t, x^s)$ instead of the more verbose $\lambda z^{s\times t}.(\pi_2(z^{s\times t}), \pi_1(z^{s\times t}))$. Also, we write π_i^n for the i-th projection of an n-tuple, $\pi_i^n: s_1 \times \cdots \times s_n \to s_i$, which is clearly definable in $\mathcal{NTL}$. For instance, $\pi_2^3 = \pi_1 \circ \pi_2$.

3 Structural Recursion and Its Expressiveness

3.1 Structural Recursion Operators

In this section, we begin our study of structural recursion on lists and trees. To this end, we add the operators $\mathsf{srl}(f)$ and $\mathsf{srt}(f)$ to $\mathcal{NTL}$:

$$\frac{f: s \times t \to t}{\mathsf{srl}(f): [s] \times t \to t}, \qquad \frac{f: atom \times [t] \times s \to t}{\mathsf{srt}(f): tree \times s \to t}.$$

(Here, f is required to be closed if it is a calculus expression.) The expression $\mathsf{srl}(f)$ stands for *structural recursion on lists* and defines the function $g: [s] \times t \to t$ such that

$$g([\,], w) = w \quad \text{and} \quad g([u] +\!\!+ v, w) = f(u, g(v, w)).$$

In other words, $g([v_1, \dots, v_n], w) = f(v_1, f(v_2, \dots f(v_n, w) \dots))$.

Example 5 For instance, if $f = \lambda(x, y).y +\!\!+ [x]$ then $\mathsf{srl}(f)(l, [\,])$ reverses the list l. Also, if we represent a directed graph G as a pair (V, E) with $V: [s]$ the nodes in G and $E: [s \times s]$ the edges in G, then $\mathsf{srl}(f)(V, E)$ computes the transitive closure of G when f is

```
$\lambda(x, closure).closure +\!\!+$
  for $y$ in $closure$ return
    for $z$ in $closure$ return
      if $x = \pi_2(y)$ and $x = \pi_1(z)$ then $[(\pi_1(y), \pi_2(z))]$ else [ ]
```

The expression $\mathsf{srt}(f)$ stands for *structural recursion on trees*. It defines the function $g' \colon \mathit{tree} \times s \to t$ such that

$$g'(\mathit{tree}(a, v), w) = f(a, \mathit{map}(g') \circ \rho_1(v, w), w).$$

Observe in particular that $g'(\mathit{tree}(a, \texttt{[ ]}), w) = f(a, \texttt{[ ]}, w)$. The extra parameter w in g' can be used, for instance, to search for a particular pattern in the input tree, as the following example shows.

Example 6 With $f \colon \mathit{atom} \times [[\mathit{unit}]] \times \mathit{atom} \to [\mathit{unit}]$ defined as follows, the expression $\mathsf{srt}(f) \colon \mathit{tree} \times \mathit{atom} \to [\mathit{unit}]$ returns an empty list on input (t, a) if and only if a does not occur in t as a label.

$$f := \lambda(\mathit{lbl}, \mathit{res}, \mathit{param}).\ \text{if } \mathit{lbl} = \mathit{param} \text{ then } [()] \text{ else } \mathit{flatten}(\mathit{res}).$$

Example 7 To express *toc* from the Introduction by means of srt we face a problem: in a computation of $\mathsf{srt}(f)$ on a tree t the 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 v themselves. This problem is solved by letting f return a pair 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 $\mathcal{NTC}(\mathsf{srt})$ by $\lambda t.\pi_1(\mathsf{srt}(f)(t, ()))$ where $f \colon \mathit{atom} \times [[\mathit{tree}] \times \mathit{tree} \times \mathit{unit}] \to ([\mathit{tree}] \times \mathit{tree})$ is $\lambda(\mathit{lbl}, \mathit{res}, \mathit{param}).(e_1, e_2)$. Here, e_1 is:

```
for $x$ in $res$ return
  if $name(\pi_2(x))$ = section then
    [$tree$(section, $fltr(children(\pi_2(x))$, title)$+\!\!+\pi_1(x)$)]
  else [ ]
```

with $\mathit{fltr} \colon [\mathit{tree}] \times \mathit{atom} \to [\mathit{tree}]$ as in Example 2, and

$$e_2 := \mathit{tree}(\mathit{lbl}, \text{for } x \text{ in } \mathit{res} \text{ return } [\pi_2(x)]).$$

Let $\mathcal{NTA}(V)$ and $\mathcal{NTC}(V)$ be the languages we obtain by adding operators in $V \subseteq \{\mathsf{srl}, \mathsf{srt}\}$ to $\mathcal{NTA}$ and $\mathcal{NTC}$, respectively. From Proposition 4 it readily follows:

Proposition 8 $\mathcal{NTA}(V) \equiv \mathcal{NTC}(V)$ *in the sense that every function definable by an expression* $f \colon s \to t$ *in* $\mathcal{NTA}(V)$ *is definable by a closed expression* $e \colon s \to t$ *in* $\mathcal{NTC}(V)$, *and vice versa.*

Hence, we can continue to conceptually view $\mathcal{NTA}(V)$ and $\mathcal{NTC}(V)$ as two components of a single language, which we denote by $\mathcal{NTL}(V)$.

Remark 9 We should note that in the presence of srl and srt, some of the basic operations of $\mathcal{NTL}$ become definable by other basic operations. For instance, $map(f) = \lambda x.\mathsf{srl}(\lambda(y,z).[f(y)] +\!\!+ z)(x, [\ ])$. We will nevertheless continue to use all of $\mathcal{NTL}$ in what follows because this situation changes when we come to restrict the allowed forms of recursion in Sect. 4 in order to tame expressiveness.

3.2 Expressiveness

As in the classical setting of Chandra and Harel [10], it is clear that all queries in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ are *generic* in the sense that they interpret only the atomic data values that appear as constants in the query [2]. It is also clear that all queries in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ are computable. Less clear, however, is their computational complexity, which forms the subject of this section.

As our model of computation, we will use the *domain Turing machine* of Hull and Su [24]. 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 article [24].

Intuitively, a DTM M is a Turing machine with a two-way infinite tape and a *register*, which can be used to store a single atomic data value. Also, a finite set A of constant atomic data values is explicitly specified in M — these correspond to the constants used in a query, and the computation of M will be generic. The transition function of M explicitly uses the members of A to refer to tape symbols, and it also uses the distinguished variables μ and ν, to refer to atoms not in A. The formal definition of a domain Turing machine is as follows. For simplicity, we first fix the alphabet Σ of *work symbols* to consist of the parentheses '**(**' and '**)**' and the brackets '**[**' and '**]**' in addition to the standard bit symbols '**0**' and '**1**' and the blank symbol '$\perp$'. We assume that the work symbols are disjoint with the atoms.

Definition 10 A *(deterministic) domain Turing machine* (DTM) is a tuple $M = (Q, A, \delta, q_s, q_h)$ consisting of:

- a finite set of *states* Q;
- a finite set of atoms A;
- a distinguished *start state* $q_s \in Q$;
- a distinguished *halting state* $q_h \in Q$;
- a *transition function* δ from $(Q \times (\Sigma \cup A \cup \{\mu\}) \times (\Sigma \cup A \cup \{\mu, \nu\})$ to $Q \times (\Sigma \cup A \cup \{\mu, \nu\})^2 \times \{\leftarrow, \rightarrow\}$. In a transition definition $\delta(state, reg, tape) = (state', reg', tape', dir)$, we require that $tape = \nu$ only if $reg = \mu$; $\mu \in \{reg', tape'\}$ only if $\mu \in \{reg, tape\}$; and $\nu \in \{reg', tape'\}$ only if $\nu \in \{reg, tape\}$.

The DTM M is viewed as having a register in addition to the normal two-way infinite tape. Let the set Δ of *tape symbols* consist of the work symbols in Σ plus all atoms. A *configuration* of M is then a five-tuple (q, a, l, b, r) where q is the current state; $a \in \Delta$ is the register contents; $l, r \in \Delta^*$ are the tape contents to the left and right of the head respectively; and $b \in \Delta$ is the tape symbol under the head. A transition value $\delta(\mathit{state}, \mathit{reg}, \mathit{tape}) = (\mathit{state}', \mathit{reg}', \mathit{tape}', \mathit{dir})$ is *generic* if $\mu \in \{\mathit{reg}, \mathit{tape}\}$. Intuitively, a generic transition value is used as a template for an infinite set of transition values which are formed by letting μ (and ν if it occurs) range over (distinct) atoms not in A. To illustrate, the transition value $\delta(q, a, \mu) = (q', \mu, \mathbf{1}, \rightarrow)$ is applicable to all configurations (q, a, l, b, r) in which the register contents is $a \in \Sigma \cup A$ and the atom under the head is an arbitrary $b \notin A$. It specifies that M should move to state q', with new register contents the atom not in A under the head, writing a $\mathbf{1}$ on the current tape cell, and moving to the right. In other words, from configuration (q, a, l, b, cr), M moves to the new configuration $(q', b, l\mathbf{1}, c, r)$. In a similar manner, the transition value $\delta(q, \mu, \mu) = (q', a, \mathbf{1}, \rightarrow)$ is applicable to all configurations (q, b, l, b, r) in which the register contents is the same as the atom $b \notin A$ under the head. In this case, M should move to state q', with new register contents the constant atom $a \in A$, writing a $\mathbf{1}$ on the current tape cell, and moving to the right. Hence, from configuration (q, b, l, b, cr), M moves to the new configuration $(q', a, l\mathbf{1}, c, r)$. Finally, the transition value $\delta(q, \mu, \nu) = (q', \mu, \mathbf{1}, \rightarrow)$ is applicable to all configurations (q, a, l, b, r) with distinct register and tape head contents, both not in A. Under these provisions, the *move* relation on configurations is defined in the usual fashion. The restriction that $\mathit{tape} = \nu$ only if $\mathit{reg} = \mu$ ensures that M is deterministic; without this restriction, the possibly different transition values $\delta(q, a, \mu)$ and $\delta(q, a, \nu)$ with $a \in A$ would be both applicable to the configuration (q, a, l, b, r) with b an atom not in A. At the beginning of the computation the register holds the blank symbol $\perp$. A computation of M is then defined as a (possibly infinite) sequence of configurations, where each pair of subsequent configurations respects the transition function.

Note that every value v is naturally represented as a string $\mathit{str}(v)$ over Δ: the atom a is represented by itself; a tree $\mathit{tree}(a, v)$ is represented by the string $\mathbf{(}\mathit{str}(a)\mathit{str}(v)\mathbf{)}$; a pair (v, w) is represented by the string $\mathbf{(}\mathit{str}(v)\mathit{str}(w)\mathbf{)}$; and a list $[v, \dots, w]$ is represented by the string $\mathbf{[}\mathit{str}(v)\dots\mathit{str}(w)\mathbf{]}$. This leads us to the following definition of a computable query.

Definition 11 (Computable Query) A query $q : s \to t$ is *computable* if there exists a DTM M that, starting with tape contents $\mathit{str}(v)$, halts with tape contents $\mathit{str}(q(v))$, for every value $v : s$.

We are particularly interested in characterizing the computational complexity of the queries expressible in $\mathcal{NTC}(\mathsf{srl}, \mathsf{srt})$. If q is computable by a DTM that halts after at most $T(n)$ steps on every input v with n the size of v and $T : \mathbb{N} \to \mathbb{N}$ a function in some class of functions $\mathcal{C}$, then we say that q is a '$\mathcal{C}$ query' or is 'computable in $\mathcal{C}$ time'. Here, the *size* $\mathrm{size}(v)$ of a value is the length of $\mathit{str}(v)$.

Theorem 12 *The class of queries expressible in* $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *coincides with the class of queries that are computable in primitive recursive time.*

Recall that the primitive recursive functions are those functions $F : \mathbb{N} \times \dots \times \mathbb{N} \to \mathbb{N}$ built by composition and primitive recursion from the constant function zero, the successor function, and the projection functions [6]. Here, F is built by primitive recursion from G and H if

$$F(0, m, \dots, m') = G(m, \dots, m') \quad \text{and}$$
$$F(n+1, m, \dots, m) = H(n, m, \dots, m', F(n, m, \dots, m')).$$

Before proving Theorem 12, let us argue that it is not unexpected. Indeed, Immerman et al. [25]; Suciu and Wong [30]; and Libkin and Wong [27] have already 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]. Combined with the fact that the primitive recursive functions are exactly those functions on the natural numbers that can be computed in primitive recursive time, this actually shows that the class of functions $F\colon \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ representable by structural recursion on (nested) sets and bags coincides with the class of functions that are computable in primitive recursive time. In this light, Theorem 12 is not surprising as $\mathcal{NTC}(\mathsf{srl}, \mathsf{srt})$ has an innate ability for object creation in the form of tree and list construction. Note that Theorem 12 is slightly more general, however, as it captures the class of all $\mathcal{NTC}(\mathsf{srt}, \mathsf{srl})$ expressible *queries*, not just those queries that represent functions on the natural numbers under some fixed encoding of the natural numbers as values.

To prove Theorem 12, we first prove the following upper bound.

Proposition 13 *Every* $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *expressible query is computable in primitive recursive time.*

Proof The proof proceeds by a straightforward induction on $\mathcal{NTA}(\mathsf{srl}, \mathsf{srt})$ expressions. We only illustrate the following interesting cases.

- Case $g \circ h$. Immediately follows from the induction hypothesis since the class of primitive recursive functions is closed under composition.
- Case $\mathsf{srl}(f)$. By the induction hypothesis, f is computable in primitive recursive time $T\colon \mathbb{N} \to \mathbb{N}$, where we may assume w.l.o.g. that T is monotone increasing. Computing $\mathsf{srl}(f)(u)$ for a given value $u = ([v_1, \dots, v_m], w)$ of size n is equivalent to computing $f(v_1, f(v_2, \dots f(v_m, w)\dots))$. Now observe that $f(v_m, w)$ has size at most $T(n)$ since the size of (v_m, w) is at most n. Similarly, $f(w_{m-1}; f(w_m, w))$ then has size at most $T(n + T(n))$. Continuing this reasoning, we see that the maximum size of an input to f is bounded by $S(m, n)$ with $S\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by primitive recursion:

$$S(0, n) = 0,$$
$$S(m+1, n) = T(n + S(m, n)).$$

  Since we need to evaluate f at most $m \leq n$ times, the total time needed to compute $\mathsf{srl}(f)(u)$ is hence bounded by $O(n \times T(S(n, n)))$, which is primitive recursive.
- Case $\mathsf{srt}(f)$. By the induction hypothesis, f is computable in primitive recursive time $T\colon \mathbb{N} \to \mathbb{N}$, where we may assume without loss of generality that T is strict monotone increasing. It is straightforward to prove by induction on a tree v that

size($\mathsf{srt}(f)(v,w)) \leq S(\text{size}(v), \text{size}(w))$ with $S\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ defined by primitive recursion:

$$S(0,l)=l,$$
$$S(k,l)=T(k+k\times S(k,l)).$$

To compute $\mathsf{srt}(f)(u)$ for a given value $u=(\mathit{tree}(a,[v_1,\dots,v_m]),w)$ of size n we must compute $f(a,[\mathsf{srt}(f)(v_1,w),\dots,\mathsf{srt}(f)(v_m,w)])$. Hence, we first need to compute $\mathsf{srt}(f)(v_i,w)$ 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 $S(n,n)$. The total time needed to compute $\mathsf{srt}(f)(u)$ is hence bounded by $O(n\times T(S(n,n)))$, which is primitive recursive. □

To conclude the proof of Theorem 12, we require the following technical lemma. It is reminiscent of the results of Immerman et al. [25]; Suciu and Wong [30]; and Libkin and Wong [27].

Lemma 14 *For every primitive recursive function $T\colon \mathbb{N}\times\cdots\times\mathbb{N}\to\mathbb{N}$ and every type s there exists $\phi_T\colon [s]\times\cdots\times[s]\to[s]$ in $\mathcal{NTL}(\mathsf{srl})$ such that $\phi_T(v,\dots,w)$ is a list of length $T(n,\dots,m)$ when $v,\dots,w$ are lists of length $n,\dots,m$ respectively.*

Proof Although the primitive recursive functions are traditionally viewed as the class of functions built by composition and primitive recursion from the constant function zero, the successor function, and the projection functions, they are equally obtained from these initial functions by composition and *pure iteration with one parameter* [17]. Here, $T(n,m)\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ is built from $S\colon\mathbb{N}\to\mathbb{N}$ by pure iteration with one parameter if $T(n,m)=S^{(n)}(m)$, i.e., $T(0,m)=m$ and $T(n+1,m)=S(T(n,m))$.

Using this view of the primitive recursive functions, we construct ϕ_T by induction on T. If T is the constant function zero, i.e., $T(n)=0$, then $\phi_T := \lambda x.[\,]$. If T is the successor function, i.e., $T(n)=n+1$, then $\phi_T := \lambda x.[e]+\!\!+x$ where $e\colon s$ is an arbitrary but fixed closed expression. If T is a projection function, i.e., $T(n_1,\dots,n_k)=n_i$, then $\phi_T := \lambda(x_1,\dots,x_k).x_i$. If T is defined by composition from the primitive recursive functions $U, S_1,\dots,S_l$, i.e.,

$$T(n_1,\dots,n_k)=U(S_1(n_1,\dots,n_k),\dots,S_l(n_1,\dots,n_k)),$$

then $\phi_T := \lambda(x_1,\dots,x_k).\phi_U(\phi_{S_1}(x_1,\dots,x_k),\dots,\phi_{S_l}(x_1,\dots,x_k))$. Finally, if $T(n,m)$ is defined by iteration from $S(m)$, i.e, $T(n,m)=S^{(n)}(m)$, then $\phi_T := \lambda(x,y).\mathsf{srl}(\phi_S\circ\pi_2)\ (x,y)$. □

We conclude the proof of Theorem 12 by providing the following lower bound:

Proposition 15 *Every primitive recursive query is expressible in $\mathcal{NTL}(\mathsf{srl},\mathsf{srt})$.*

Proof Let $q: s \to t$ be a query computable by a DTM M in primitive recursive time $T: \mathbb{N} \to \mathbb{N}$. Intuitively, we express $q: s \to t$ by simulating M: first, we encode the input $v: s$ to q as a DTM tape using an expression $encod^s$; next, we simulate $T(\text{size}(v))$ steps of M on this tape; and finally, we decode the resulting DTM tape $str(q(v))$ into $q(v)$ using an expression $decod^t$. The detailed simulation is as follows.

The Encoding Function Recall that the tape symbols Δ consist of the work symbols in Σ plus the atoms. We represent $\sigma \in \Delta$ by a pair $rep(\sigma)$ of type $[unit] \times atom$: $rep(\sigma) := ([\,], \sigma)$ if σ is an atom, and $rep(\sigma) := ([()], a_\sigma)$ if σ is a work symbol in Σ. Here, a_σ is an arbitrary but fixed atom such that $a_\sigma \neq a_{\sigma'}$ for distinct work symbols σ and σ'. Note in particular that $rep(\sigma) \neq rep(\sigma')$ when σ and σ' are distinct tape symbols. We can then represent the tape contents $\sigma \ldots \sigma'$ of M at any given time by the value $[rep(\sigma), \ldots, rep(\sigma')]$. Let $symb$ abbreviate the type $[unit] \times atom$. The encoding function $encod^s: s \to [symb]$ such that $encod^s(v)$ is the list representation of $str(v)$ is then readily expressed by induction on s:

- $encod^{atom} := \lambda x.([\,], x)$;
- $encod^{s \times s'} := \lambda(x, y).[rep(()] + \!\!+ encod^s(x) + \!\!+ encod^{s'}(y) + \!\!+ [rep())]$;
- $encod^{[s]} := \lambda x.[rep([)] + \!\!+ (\text{for } y \text{ in } x \text{ return } encod^s(y)) + \!\!+ [rep(])]$; and
- $encod^{tree} := \lambda x.\text{srt}(f)(x, [\,])$ with

$$f := \lambda(lbl, rec, param).[rep(()] + \!\!+ [([\,], lbl)] + \!\!+ [rep([)]$$
$$+ \!\!+ \big(\text{for } x \text{ in } rec \text{ return } x\big) + \!\!+ [rep(])] + \!\!+ [rep())].$$

The Step Function Next, we represent a configuration (q, a, l, b, r) of M as the value $(q, rep(a), l', rep(b), r')$, where we assume without loss of generality that the states of M are atoms; r' is the list representation of r; and l' is the *reverse* of the list representation of l. The function *step* such that $step(c)$ represents the configuration after executing one step of M on c is then expressed as follows. Let $\{q_1, \ldots, q_m\}$ be the set of M's states, let A be the atoms interpreted by M, let δ be M's transition function, and let $\{\sigma_1, \ldots, \sigma_k\} = \Sigma \cup A$. Then *step* is the nested conditional

```
$\lambda(state, reg, left, hd, right).$
        if $state = q_1$ then $e_{q_1}$ else ... else if $state = q_{m-1}$ then $e_{q_{m-1}}$ else $e_{q_m}$,
```

with e_{q_i} the expression

```
if $hd = rep(\sigma_1)$ then
  if $reg = rep(\sigma_1)$ then $e_{q_i,\sigma_1,\sigma_1}$ else ... else if $reg = rep(\sigma_k)$ then $e_{q_i,\sigma_k,\sigma_1}$
  else $e_{q_i,\mu,\sigma_1}$
else if $hd = rep(\sigma_2)$ then
  if $reg = rep(\sigma_1)$ then $e_{q_i,\sigma_1,\sigma_2}$ else ... else if $reg = rep(\sigma_k)$ then $e_{q_i,\sigma_k,\sigma_2}$
  else $e_{q_i,\mu,\sigma_2}$
else ...
else if $hd = rep(\sigma_k)$ then
  if $reg = rep(\sigma_1)$ then $e_{q_i,\sigma_1,\sigma_k}$ else ... else if $reg = rep(\sigma_k)$ then $e_{q_i,\sigma_k,\sigma_k}$
  else $e_{q_i,\mu,\sigma_k}$
else if $hd = reg$ then $e_{q_i,\mu,\mu}$ else $e_{q_i,\mu,\nu}$.
```

Here, the expression $e_{q_i,o,p}$ with $o, p \in \Sigma \cup A \cup \{\mu, \nu\}$ is responsible for simulating the actual step of M taken when the transition $\delta(q_i, o, p)$ applies. It is defined as follows. Suppose that $\delta(q_i, o, p) = (q_j, o', p', dir)$. Let the expression $e_{o'}$ be $rep(o')$ if $o' \in \Sigma \cup A$, and let it be the variable *reg* otherwise. Let $e_{p'}$ be $rep(p')$ if $p' \in \Sigma \cup A$, and let it be the variable *hd* otherwise. If $dir = \rightarrow$ then $e_{q_i,o,p}$ is

```
if $right = [\,]$ then $(q_j, e_{o'}, [e_{p'}] +\!\!+ left, rep(\bot), [\,])$
                                        else $(q_j, e_{o'}, [e_{p'}] +\!\!+ left, head(r), tail(r))$,
```

otherwise it is

```
if $left = [\,]$ then $(q_j, e_{o'}, [\,], rep(\bot), [e_{p'}] +\!\!+ right)$
                                   else $(q_j, e_{o'}, tail(left), head(left), [e_{p'}] +\!\!+ right)$.
```

The Decoding Function First define the function $collect$: $[symb] \to [[symb]]$ that, on the representation of $str((v, w))$ or $str([v, \dots, w])$ returns the list containing the list representation of $str(v), \dots, str(w)$ (in order). It suffices to define $collect := \lambda rep.(\pi_3^3 \circ \mathsf{srl}(f))\ (rep, ([\,], [\,], [\,]))$, with $f : symb \times ([unit] \times [symb] \times [[symb]]) \to ([unit] \times [symb] \times [[symb]])$ the expression

```
$\lambda(symb, (depth, current, accu))$.
    if $symb = rep())$ or $symb = rep(])$ then
      if $depth = [\,]$ then $([()], [\,], accu)$
      else $([()] +\!\!+ depth, [symb] +\!\!+ current, accu)$
    else if $symb = rep(()$ or $symb = rep([)$ then
      if $depth = [()]$ then
        $([\,], [\,], [current] +\!\!+ accu)$
      else if $depth = [(), ()]$ then
        $([()], [\,], [[symb] +\!\!+ current] +\!\!+ accu)$
      else
        $(tail(depth), [symb] +\!\!+ current, accu)$
    else
      $(depth, [symb] +\!\!+ current, accu)$
```

Now define the decoding function $decod^t : [atom \times atom] \to t$ that decodes the list representation of $str(w)$ back into w for any $w : t$ by induction on t:

- $decod^{atom} := \lambda x.head(x)$;
- $decod^{t \times t'} := \langle decod^t \circ head, decod^{t'} \circ head \circ tail \rangle \circ collect$;
- $decod^{[t]} := \lambda x.$ for y in $collect(x)$ return $[decod^t(y)]$
- $decod_{tree} := head \circ head \circ \mathsf{srl}(f) \circ \langle id, [\,] \rangle$, where $f : symb \times [[tree]] \to [[tree]]$ is the function expression which reconstructs the represented tree by maintaining a stack (i.e., a list of list of trees), defined as follows:

```
  $\lambda(symb, rec)$.
    if $symb = rep(])$ then $[[\,]] +\!\!+ rec$
```

```
else if $symb = rep([) \text{ or } symb = rep(() \text{ or } symb = rep())$ then $rec$
else if $rec = [\,]$ then $[[tree(\pi_2\, symb, [\,])]]$
else if $tail(rec) = [\,]$ then $[[tree(\pi_2\, symb, head(rec))]]$
else $[tree(\pi_2\, symb, head(rec) +\!\!+ head(tail(rec)))] +\!\!+ tail(tail(rec))$
```

The Entire Simulation It remains to express $q\colon s \to t$ by simulating M. Let q_s be M's start state. We assume without loss of generality that M always halts with its head one tape cell to the left of the output. Let ϕ_T be the expression simulating the primitive recursive function $T\colon \mathbb{N} \to \mathbb{N}$, as given by Lemma 14. Observe in particular that $(\phi_T \circ encod^s)(v)$ returns a list of length $T(\text{size}(v))$. Let *clean* be the function which removes all blank symbols from a tape representation:

$$clean := \lambda x.\ \text{for } y \text{ in } x \text{ return } (\text{if } y = rep(\bot) \text{ then } [\,] \text{ else } [y]).$$

Let *initconf* be the function such that $initconf(v)$ is the initial configuration of M on v:

$$initconf := \lambda x.(q_s, rep(\bot), [\,], head(encod^s(x)), tail(encod^s(x))).$$

We then express $q\colon s \to t$ by simulating $T(n)$ steps of M on the initial configuration, cleaning the resulting tape and subsequently decoding the output:

$$decod^t \circ clean \circ \pi_4^4 \circ \mathsf{srl}(step \circ \pi_2) \circ \langle \phi_T \circ encod^s, initiconf\rangle. \qquad \square$$

Theorem 12 states that structural recursion on lists and trees taken together gives rise to a robust and very expressive class of queries. It is interesting to note that this expressiveness drops dramatically when we consider structural recursion on lists or trees separately. Intuitively, this is because srl only provides "horizontal" recursion across lists, while srt only provides "vertical" recursion down trees. As such, $\mathcal{NTL}(\mathsf{srl})$ can only manipulate inputs up to bounded depth, while $\mathcal{NTL}(\mathsf{srt})$ can only manipulate inputs up to bounded width. For instance, let $lastlab\colon tree \to atom$ be the query that maps its input tree v to the label of the last node visited when traversing v in pre-order. This query is clearly computable in linear time. Nevertheless:

Theorem 16 *The query lastlab*: $tree \to atom$ *is inexpressible in both* $\mathcal{NTL}(\mathsf{srl})$ *and* $\mathcal{NTL}(\mathsf{srt})$. *Hence, structural recursion on lists or trees alone is not strong enough to express all linear time queries.*

Proof Suppose, for the purpose of contradiction, that *lastlab* is expressible in $\mathcal{NTA}(\mathsf{srl})$. Let $\text{atoms}^k(v)$ be the set of atoms occurring in value v "up to tree-depth k":

$$\begin{aligned}
\text{atoms}^k(a) &:= \{a\},\\
\text{atoms}^k(v, w) &:= \text{atoms}^k(v) \cup \text{atoms}^k(w),\\
\text{atoms}^k([v, \dots, w]) &:= \text{atoms}^k(v) \cup \dots \cup \text{atoms}^k(w),\\
\text{atoms}^0(tree(a, v)) &:= \emptyset,\\
\text{atoms}^{k+1}(tree(a, v)) &:= \{a\} \cup \text{atoms}^k(v).
\end{aligned}$$

Let $\mathrm{depth}(f)$ be the number of occurrences of the functions *lab* and *children* in an $\mathcal{NTA}(\mathsf{srl})$ expression $f: s \to t$, and let $\mathrm{atoms}(f)$ be the set of atoms for which a subexpression Ka occurs in f. It is readily verified by induction on $f: s \to t$ that $\mathrm{atoms}^k(f(v)) \subseteq \mathrm{atoms}^{k+\mathrm{depth}(f)}(v) \cup \mathrm{atoms}(f)$, for any $v: s$. Then let v: *tree* be a tree for which the node last visited in a pre-order traversal lies at $\mathrm{depth}(\mathit{lastlab}) + 1$ and carries a unique label a not occurring anywhere else in v, nor in $\mathrm{atoms}(\mathit{lastlab})$. Then $a \notin \mathrm{atoms}^0(\mathit{lastlab}(v))$ since

$$\mathrm{atoms}^0(\mathit{lastlab}(v)) \subseteq \mathrm{atoms}^{\mathrm{depth}(\mathit{lastlab})} \cup \mathrm{atoms}(\mathit{lastlab}),$$

and $a \notin \mathrm{atoms}^{\mathrm{depth}(\mathit{lastlab})}(v) \cup \mathrm{atoms}(\mathit{lastlab})$. Since $\mathrm{atoms}^0(a) = \{a\}$, this implies that $\mathit{lastlab}(v) \neq a$ although a is the label of the last node visited when traversing v in pre-order, which gives the desired contradiction.

A similar reasoning shows that *lastlab* is inexpressible in $\mathcal{NTA}(\mathsf{srt})$. □

4 Taming Structural Recursion

Since queries are expected to be computable in polynomial time for all practical purposes, it follows from Theorem 12 that $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ is too powerful a query language. In this section we therefore investigate intrinsic restrictions of structural recursion that capture exactly the polynomial time queries. We start with a semantic restriction, from which we next derive a suitable syntactic restriction.

4.1 A Semantic Restriction

Let us refer to the expressions g in $\mathsf{srl}(g)$ or $\mathsf{srt}(g)$ as *step expressions*. It is clear that, in order for a query $f: s \to t$ to be computable in polynomial time, f should never create intermediate results of more than polynomial size. This condition is trivially satisfied if f is an expression that does not use structural recursion. To see how structural recursion can create results of exponential size or more, consider the query *explist*: $[s] \to [s]$ defined by $\mathit{explist} := \lambda z.\ \mathsf{srl}(\lambda(x, y).y{+\!\!+}y)\ (z, [a])$. Clearly, if v is a list of length k, then $\mathit{explist}(v)$ is 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 *exptree*: *tree* $\to$ *tree* defined by $\mathit{exptree} := \lambda u.\ \mathsf{srt}(\lambda(x, y, z).\mathit{tree}(x, y{+\!\!+}y))\ (u,\ [\ ])$. 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 $\lambda(x, y, z).\mathit{tree}(x, y{+\!\!+}y)$ of *exptree* doubles its result at each recursive invocation. This leads us to the following definition.

Definition 17 (Tamed expressions) An $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ expression is *tamed* if

1. for every subexpression $\mathsf{srl}(f)$ with $f: s \times t \to t$ there exists a function $F: \mathbb{N} \to \mathbb{N}$ such that $\mathrm{size}(f(v, w)) \leq F(\mathrm{size}(v)) + \mathrm{size}(w)$; and
2. for every subexpression $\mathsf{srt}(f)$ with f: *atom* $\times [t] \times s \to t$ there exists a function $F: \mathbb{N} \to \mathbb{N}$ such that $\mathrm{size}(f(u, v, w)) \leq F(\mathrm{size}(u) + \mathrm{size}(w)) + \mathrm{size}(v)$.

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 18 *Every query definable by a tamed expression in* $\mathcal{NTL}(\mathsf{srl},\mathsf{srt})$ *is computable in polynomial time.*

Proof The proof proceeds by induction on tamed $\mathcal{NTA}(\mathsf{srl},\mathsf{srt})$ expressions. We only illustrate the following interesting cases.

- Case $\mathsf{srl}(f)$. By the induction hypothesis, f is computable in polynomial time $T\colon \mathbb{N}\to\mathbb{N}$, where we may assume w.l.o.g. that T is monotone increasing. Since $\mathsf{srl}(f)$ is tamed, there exists a function $F\colon \mathbb{N}\to\mathbb{N}$ such that $\mathrm{size}(f(v,w)) \leq F(\mathrm{size}(v)) + \mathrm{size}(w)$ for all v and w. Since in polynomial time one can construct at most a polynomial output, it follows that F can be taken a polynomial. Computing $\mathsf{srl}(f)(u)$ for a given value $u = ([v_1,\dots,v_m], w)$ of size n is equivalent to computing $f(v_1, f(v_2,\dots f(v_m,w)\dots))$. Now observe that $f(v_m,w)$ has size at most $F(n)+n$ since the size of v_m and w is at most n. Similarly, $f(w_{m-1}; f(w_m,w))$ then has size at most $F(n)+F(n)+n$. Continuing this reasoning, we see that the maximum size of an input to f is bounded by $m\times F(n)+n$. Since we need to evaluate f at most $m\leq n$ times, the total time needed to compute $\mathsf{srl}(f)(u)$ is hence bounded by $O(n\times T(n\times F(n)+n))$, which is a polynomial in n.
- Case $\mathsf{srt}(f)$. By the induction hypothesis, f is computable in polynomial time $T\colon \mathbb{N}\to\mathbb{N}$, where we may assume without loss of generality that T is monotone increasing. Since $\mathsf{srt}(f)$ is tamed, there exists a function $F\colon \mathbb{N}\to\mathbb{N}$ such that $\mathrm{size}(f(u,v,w)) \leq F(\mathrm{size}(u)+\mathrm{size}(w)) + \mathrm{size}(v)$. Since in polynomial time one can construct at most a polynomial output, it follows that F can be taken a polynomial. It is then straightforward to prove by induction on a tree v that $\mathrm{size}(\mathsf{srt}(f)(v,w)) \leq \mathrm{size}(v)\times(F(1+\mathrm{size}(w))+2)$, for any w. Indeed, if $v = \mathit{tree}(a, [\,])$, then

$$\begin{aligned}\mathrm{size}(\mathsf{srt}(f)\ (v,w)) &= \mathrm{size}(f(a,\ [\,],w))\\ &\leq F(\mathrm{size}(a)+\mathrm{size}(w)) + \mathrm{size}([\,])\\ &\leq \mathrm{size}(\mathit{tree}(a,\ [\,]))\times(F(1+\mathrm{size}(w))+2).\end{aligned}$$

  If $v = \mathit{tree}(a,[v_1,\dots,v_k])$, then

$$\begin{aligned}\mathrm{size}(\mathsf{srt}(f)\ (v,w)) &= \mathrm{size}(f(a,[\mathsf{srt}(f)(v_1,w),\dots,\mathsf{srt}(f)(v_k,w)],w))\\ &\leq F(\mathrm{size}(a)+\mathrm{size}(w)) + \mathrm{size}([\mathsf{srt}(f)(v_1,w),\dots,\mathsf{srt}(f)(v_k,w)])\\ &= F(1+\mathrm{size}(w)) + \mathrm{size}(\mathsf{srt}(f)(v_1,w)) + \cdots + \mathrm{size}(\mathsf{srt}(f)(v_k,w)) + 2\\ &\leq (1+\mathrm{size}(v_1)+\cdots+\mathrm{size}(v_k))\times(F(1+\mathrm{size}(w))+2)\\ &\leq \mathrm{size}(v)\times(F(1+\mathrm{size}(w))+2).\end{aligned}$$

  To compute $\mathsf{srt}(f)(u)$ for a given value $u = (\mathit{tree}(a,[v_1,\dots,v_m]),w)$ of size n we must compute $f(a,[\mathsf{srt}(f)(v_1,w),\dots,\mathsf{srt}(f)(v_m,w)])$. Hence, we first need to compute $\mathsf{srt}(f)(v_i,w)$ 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 $n \times (F(1 + n) + 2)$ by our observation above. The total time needed to compute $\mathsf{srt}(f)(u)$ is hence bounded by $O(n \times T(n \times (F(1 + n) + 2)))$, which is a polynomial in n. $\square$

The converse to Proposition 18 is also true: every polynomial time query can be defined by a tamed $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ expression. In fact, in the following subsection we will show an even stronger statement. Say that $f : s \to t$ is *linearly bounded* if there exists a constant c such that $\text{size}(f(v)) \le \text{size}(v) + c$ for every $v : s$.

Proposition 19 *Every polynomial time query can be defined by an expression in* $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *in which* f *is linearly bounded for every subexpression* $\mathsf{srl}(f)$ *or* $\mathsf{srt}(f)$.

It readily follows that every polynomial time query can be defined by means of a tamed $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ expression.

4.2 A Syntactic Restriction

We prove Proposition 19 by providing an effective syntax for the polynomial time queries in which all step expressions are linearly bounded. To motivate this syntax, consider again the problematic step expression $\lambda(x, y).y +\!\!+ y$ from *explist* as defined in Sect. 4.1. Since this step expression doubles the recursive argument y, it is not linearly bounded, and our syntax should therefore disallow it. The first solution that comes to mind is to require that x and y occur 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, linearly bounded step expressions like $\lambda(x, y).\ \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ x +\!\!+ y\ \mathsf{else}\ y$ with x and 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, $\lambda(x, y).\ \mathsf{for}\ z\ \mathsf{in}\ [a, b]\ \mathsf{return}\ y$ would be accepted, although it is equivalent to the problematic $\lambda(x, y).y +\!\!+ y$ above. For these reasons, a more fine-grained restriction of step expressions along the lines of Caseiro [9] is in order.

Let $e[x/e_1, y/e_2]$ stand for the expression we obtain by simultaneously replacing every free occurrence of x and y in e by e_1 and e_2, respectively.

Definition 20 (Linear $\mathcal{NTC}$) An $\mathcal{NTC}$ expression is *linear* if

- it is a variable x, an atom a, the empty tuple () or the empty list `[ ]`;
- it is $[e]$, (e, e'), $e +\!\!+ e'$, or $\mathit{tree}(e, e')$ with e and e' linear and $FV(e) \cap FV(e') = \emptyset$;
- it is $\mathsf{for}\ y\ \mathsf{in}\ e\ \mathsf{return}\ y$ or $\mathsf{for}\ y\ \mathsf{in}\ e\ \mathsf{return}\ \mathit{children}(y)$ with e linear;
- it is $\mathsf{if}\ e = e'\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f$ with e and e' arbitrary and e_t and e_f linear;
- it is $e[x/g(e'), y/h(e')]$ with e and e' linear, $FV(e) \cap FV(e') = \emptyset$, and g and h distinct elements of $\{\pi_1, \pi_2, \mathit{head}, \mathit{tail}, \mathit{lab}, \mathit{children}\}$; or
- it is $\lambda(x, \dots, y).e$ with e linear.

For instance, $y \mathbin{+\!\!+} y$ is not linear as y occurs free in both arguments of $\mathbin{+\!\!+}$. The step expression f from Example 5 used to compute the transitive closure of a graph is also not linear, as the for loops used do not have the required form. In contrast, if $e_1 = e_2$ then $x \mathbin{+\!\!+} y$ else y is linear since test expressions can be arbitrary and since the branches $x \mathbin{+\!\!+} y$ and y are both linear. Also, $\mathit{tail}(x)$ and $(\mathit{head}(\pi_1 x), \mathit{tail}(\pi_1 x))$ are linear:

$$\mathit{tail}(x) = y[y/\mathit{tail}(x), z/\mathit{head}(x)] \quad \text{and}$$

$$(\mathit{head}(\pi_1 x), \mathit{tail}(\pi_1 x)) = (z_1, z_2)[z_1/\mathit{head}(\pi_1 x), z_2/\mathit{tail}(\pi_1 x)].$$

Observe in particular that in case of $e[x/g(e'), y/h(e')]$ with e and e' linear, $FV(e) \cap FV(e') = \emptyset$, and $g \neq h$ elements of $\{\pi_1, \pi_2, \mathit{head}, \mathit{tail}, \mathit{lab}, \mathit{children}\}$, x and y need not occur in e.

We next introduce the corresponding restriction on $\mathcal{NTA}$ expressions. For two expressions $g\colon s \to s'$ and $h\colon t \to t'$, let $g \times h\colon s \times t \to s' \times t'$ abbreviate $\langle g \circ \pi_1, h \circ \pi_2 \rangle$ and let $\alpha\colon r \times (s \times t) \to (r \times s) \times t$ abbreviate $\langle \langle \pi_1, \pi_1 \circ \pi_2 \rangle, \pi_2 \circ \pi_2 \rangle$.

Definition 21 (Linear $\mathcal{NTA}$) An $\mathcal{NTA}$ expression is *linear* if

- it is Ka, id, $!$, π_1, π_2, sng, $[\ ]$, $\mathbin{+\!\!+}$, $\mathit{flatten}$, $\mathit{flatten} \circ \mathit{map}(\mathit{children})$, head, tail, tree, lab, $\mathit{children}$, α, or $\langle !, \mathit{id} \rangle$;
- it is $g \circ h$ or $g \times h$ with g and h linear;
- it is $\langle g, h \rangle$ with g and h distinct elements of $\{\pi_1, \pi_2, \mathit{head}, \mathit{tail}, \mathit{lab}, \mathit{children}\}$; or
- it is $\mathit{cond} \circ \langle g_1, g_2, h_1, h_2 \rangle$ with g_1, g_2 arbitrary and h_1, h_2 linear.

For instance, $\langle \mathit{id}, \mathit{id} \rangle$ is not linear, whereas $\mathit{swap}\colon s \times t \to t \times s$ given by $\mathit{swap} := \langle \pi_2, \pi_1 \rangle$ is linear. The linear $\mathcal{NTC}$ expression $\lambda x.(\mathit{head}(\pi_1 x), \mathit{tail}(\pi_1 x))$ can also be linearly expressed in $\mathcal{NTA}$ as $\langle \mathit{head}, \mathit{tail} \rangle \circ \pi_1$. This is no coincidence, as the following proposition shows.

Proposition 22 *Linear $\mathcal{NTA} \equiv$ linear $\mathcal{NTC}$ in the sense that every query definable by a linear $\mathcal{NTA}$ expression $f\colon s \to t$ is definable by a closed linear $\mathcal{NTC}$ expression $e\colon s \to t$, and vice versa.*

Proof First observe that the $\mathcal{NTC}$ expressions

$$\pi_1(x) \mathbin{+\!\!+} \pi_2(x) \quad \text{and} \quad \big((\pi_1(x), \pi_1(\pi_2(x))), \pi_2(\pi_2(x))\big)$$

are linear since

$$\pi_1(x) \mathbin{+\!\!+} \pi_2(x) = (y \mathbin{+\!\!+} z)[y/\pi_1(x), z/\pi_2(x)] \quad \text{and}$$

$$\begin{aligned}&\big((\pi_1(x), \pi_1\pi_2(x)), \pi_2\pi_2(x)\big)\\ &\quad = ((y_1, y_2), y_3)[y_2/\pi_1(z), y_3/\pi_2(z)][y_1/\pi_1(x), z/\pi_2(x)].\end{aligned}$$

Next, observe that $\pi_1(x)$, $\pi_2(x)$, $\mathit{head}(x)$, $\mathit{tail}(x)$, $\mathit{lab}(x)$, and $\mathit{children}(x)$ all linear since for instance $\pi_1(x) = y[y/\pi_1(x), z/\pi_2(x)]$. Finally, verify the following claim by induction on an $\mathcal{NTC}$ expression e: if $FV(e) \cap FV(e') = \emptyset$ and e and e' are linear, then $e[x/e']$ is also linear.

Then every linear $\mathcal{NTA}$ expression $f: s \to t$ can be translated into the linear $\mathcal{NTC}$ expression $f_x^*: t$ such that $FV(f_x^*) \subseteq \{x\}$ and $f \equiv \lambda x^s. f_x^*$:

$$\begin{aligned}
(Ka)_x^* &= a & id_x^* &= x \\
!_x^* &= () & (\pi_1)_x^* &= \pi_1(x) \\
(\pi_2)_x^* &= \pi_2(x) & sng_x^* &= [x] \\
(\texttt{[ ]})_x^* &= \texttt{[ ]} & +\!\!+_x^* &= \pi_1(x) +\!\!+ \pi_2(x) \\
head_x^* &= head(x) & tail_x^* &= tail(x) \\
tree_x^* &= tree(x) & lab_x^* &= lab(x) \\
children_x^* &= children(x) & \alpha_x^* &= ((\pi_1(x), \pi_1\pi_2(x)), \pi_2\pi_2(x)) \\
\langle !, id\rangle_x^* &= ((), x) & (g \circ h)_x^* &= g_y^*[y/h_x^*] \\
(g \times h)_x^* &= (g_y^*, h_z^*)[y/\pi_1(x), z/\pi_2(x)] & \langle g, h\rangle_x^* &= (y, z)[y/g(x), z/h(x)]
\end{aligned}$$

Furthermore,

$$\begin{aligned}
flatten_x^* &= \textsf{for } y \textsf{ in } x \textsf{ return } y, \\
flatten \circ map(children)_x^* &= \textsf{for } y \textsf{ in } x \textsf{ return } children(y), \text{ and} \\
cond \circ \langle f_1, f_2, g, h\rangle_x^* &= \textsf{if } f_1'\, x = f_2'\, x \textsf{ then } g_x^* \textsf{ else } h_x^*,
\end{aligned}$$

where f_1' and f_2' are the $\mathcal{NTC}$ expressions equivalent to f_1 and f_2 respectively, which exist by Proposition 4.

For the converse, recall our convention that $(v_1, \ldots, v_n)$ is an abbreviation of $(v_1, (v_2, (\ldots, (v_{n-1}, v_n)\ldots)))$ and observe that the function

$$split^n_{i_1,\ldots,i_k}(v_1, \ldots, v_n) = ((v_{i_1}, \ldots, v_{i_k}), (v_{j_1}, \ldots, v_{j_l}))$$

with $0 \leq k \leq n$; $1 \leq i_1 < \cdots < i_k \leq n$ a (possibly empty) subset of $\{1, \ldots, n\}$; and $1 \leq j_1 < \cdots < j_l \leq n = \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$, is linearly expressible in $\mathcal{NTA}$. Indeed,

$$split^n_{i_1,\ldots,i_k} = \begin{cases} \langle !, id\rangle & \text{when } 0 = k \leq n \\ front^n_{i_1} & \text{when } 1 = k \leq n \\ \alpha \circ (id \times split^{n-1}_{i_2-1,\ldots,i_k-1}) \circ front^n_{i_1} & \text{when } 2 \leq k < n \\ \langle \pi_2, \pi_1\rangle \circ \langle !, id\rangle & \text{when } 2 \leq k = n \end{cases}$$

where $front^n_i$ with $1 \leq i \leq n$ is the linear $\mathcal{NTA}$ expression such that

$$front^n_i(v_1, \ldots, v_n) = (v_i, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n),$$

expressed as follows:

$$front^n_i = \begin{cases} id & \text{when } 1 = i \leq n \\ \langle \pi_2, \pi_1\rangle & \text{when } 2 = i = n \\ \langle \pi_2, \pi_1\rangle \circ \alpha \circ (id \times \langle \pi_2, \pi_1\rangle \circ front^{n-1}_{i-1}) & \text{when } 2 \leq i, 3 \leq n \end{cases}$$

Using $split^n_{i_1,\dots,i_k}$ we can translate every linear and closed $\mathcal{NTC}$ expression $\lambda(x_1^{s_1},\dots,x_n^{s_n}).e$ with $e\colon t$ and $FV(e)\subseteq\{x_1,\dots,x_n\}$, into an equivalent linear $\mathcal{NTA}$ expression $f_e^{x_1,\dots,x_n}\colon s_1\times\cdots\times s_n\to t$. The translation proceeds by induction on the linearity of e.

- When e is one of x, a, $()$, or $[\,]$, we take

$$f_{x_i}^{x_1,\dots,x_n}:=\pi_i^n,\qquad f_a^{x_1,\dots,x_n}:=Ka,\qquad f_{()}^{x_1,\dots,x_n}:=!,\qquad f_{[\,]}^{x_1,\dots,x_n}:=[\,].$$

- When e is (e_1,e_2) with $FV(e_1)\cap FV(e_2)=\emptyset$, then let $x_{i_1},\dots,x_{i_k}$ with $1\le i_1<\cdots<i_k\le n$ be the free variables of e_1 and let $x_{j_1},\dots,x_{j_k}$ with $1<j_1<\cdots<j_l\le n$ be $\{x_1,\dots,x_n\}-\{x_{i_1},\dots,x_{i_k}\}$. Intuitively, to simulate (e_1,e_2) we first split the free variables e into a pair: the first component consisting of the free variables of e_1, the second of the free variables of e_2, and then call e_1 and e_2 on these respective components. That is, we take

$$f_{(e_1,e_2)}^{x_1,\dots,x_n}:=(f_{e_1}^{x_{i_1},\dots,x_{i_k}}\times f_{e_2}^{x_{j_1},\dots,x_{j_l}})\circ split^n_{i_1,\dots,i_k}.$$

- The case for $e=e_1{+\!\!+}e_2$ and $e=tree(e_1,e_2)$ is similar.
- When $e=\textsf{for}\ y\ \textsf{in}\ e'\ \textsf{return}\ y$ or $e=\textsf{for}\ y\ \textsf{in}\ e'\ \textsf{return}\ children(y)$, we take

$$f_{\textsf{for}\ y\ \textsf{in}\ e'\ \textsf{return}\ y}^{x_1,\dots,x_n}=flatten\circ f_{e'}^{x_1,\dots,x_n},$$

$$f_{\textsf{for}\ y\ \textsf{in}\ e'\ \textsf{return}\ children(y)}^{x_1,\dots,x_n}=(flatten\circ map(children))\circ f_{e'}^{x_1,\dots,x_n}.$$

- When $e=\textsf{if}\ e_1=e_2\ \textsf{then}\ e_3\ \textsf{else}\ e_4$ then let g_1 and g_2 be the $\mathcal{NTA}$ expressions equivalent to $\lambda(x_1,\dots,x_n).e_1$ and $\lambda(x_1,\dots,x_n).e_2$, respectively. (These exist by Proposition 4). Then we take

$$f_{\textsf{if}\ e_1=e_2\ \textsf{then}\ e_3\ \textsf{else}\ e_4}^{x_1,\dots,x_n}=cond\circ\langle g,h,f_{e_3}^{x_1,\dots,x_n},f_{e_4}^{x_1,\dots,x_n}\rangle.$$

- When $e=e_1[y/g(e_2),z/h(e_2)]$ with $FV(e_1)\cap FV(e_2)=\emptyset$ and g and h distinct elements of $\{\pi_1,\pi_2,head,tail,lab,children\}$, then let $x_{j_1},\dots,x_{j_l}$ with $1\le j_1<\cdots<j_l\le n$ be the free variables of e_2 and let $x_{i_1},\dots,x_{i_k}$ with $1\le i_1<\cdots<i_l\le n$ be $\{x_1,\dots,x_n\}-\{x_{j_1},\dots,x_{j_l}\}$. Then we take

$$f_e^{x_1,\dots,x_n}:=f_{e_1}^{y,z,x_{i_1},\dots,x_{i_k}}\circ\alpha^{-1}\circ((\langle g,h\rangle\circ f_{e_2}^{x_{j_1},\dots,x_{j_l}})\times id)\circ split^n_{j_1,\dots,j_l}$$

where α^{-1} such that $\alpha^{-1}((y,z),(x_{i_1},\dots,x_{i_k}))=(y,z,x_{i_1},\dots,x_{i_k})$ is

$$\alpha^{-1}:=\langle\pi_2,\pi_1\rangle\circ(\langle\pi_2,\pi_1\rangle\times id)\circ\alpha\circ\langle\pi_2,\pi_1\rangle\circ(\langle\pi_2,\pi_1\rangle\times id).\qquad\square$$

An easy induction on linear $\mathcal{NTA}$ expressions then shows that linearity implies linear boundedness:

Proposition 23 *Every query definable by a linear $\mathcal{NTL}$ expression is linearly bounded.*

Definition 24 (Safety) An expression in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ is *safe* if every step expression occurring in it is linear.

It immediately follows from Proposition 23 that safe expressions are tamed; they are hence computable in polynomial time by Proposition 18. Note, however, that some expressions, like the one computing the transitive closure of a graph in Example 5 or the one expressing *toc* from the Introduction in Example 7 denote polynomial time queries, but are not safe. This hence raises the question how powerful safe expressions are. Fortunately,

Proposition 25 *Every polynomial time query is expressible by a safe, closed function expression in* $\mathcal{NTC}(\mathsf{srl}, \mathsf{srt})$.

Proof Let $q : s \to t$ be a query computable by a DTM M in polynomial time $T : \mathbb{N} \to \mathbb{N}$. Since T serves only as an upper bound on the running time of M, we may assume without loss of generality that $T(n) = c_k n^k + \cdots + c_2 n^2 + c_1 n + c_0$ with $c_k, \ldots, c_0$ natural number coefficients. It is readily verified that for every type s there exists $\phi_T : [s] \to [s]$ in $\mathcal{NTL}$ such that $\phi_T(v)$ is a list of length $T(n)$ when v is a list of length n. Indeed, it suffices to note that we can simulate natural number addition by concatenation and multiplication by the for loop: if v is a list of length m and w is a list of length n, then $v{+\!\!+}w$ is a list of length $m+n$ and $\mathsf{for}\ x\ \mathsf{in}\ v\ \mathsf{return}\ w$ is a list of length $m \times n$. It is then straightforward to obtain ϕ_T by suitable combination of constant lists (to represent the coefficients), concatenation, and the for loop.

As in the proof of Proposition 15, we can now express $q : s \to t$ by simulating M: first, we encode the input $v : s$ to q as a DTM tape using an expression $encod^s$; next, we simulate $T(\text{size}(v))$ steps of M on this tape; and finally, we decode the resulting DTM tape $str(q(v))$ into $q(v)$ using an expression $decod^t$. The detailed simulation is as in the proof of Proposition 15 except that in order to simulate M the required $T(\text{size}(v))$ number of steps, we use the function ϕ_T defined above. The proposition then follows, as it is readily verified that every step expression used in the simulation is linear. □

In particular, Examples 5 and 7 can hence be expressed in a safe way. Proposition 19 immediately follows from Propositions 23 and 25. Moreover, from Propositions 23, 18, and 25 it immediately follows that safe expressions provide an effective syntax for the polynomial time queries.

Theorem 26 *The class of queries expressible by safe expressions in* $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *coincides with the class of queries that are computable in polynomial time.*

5 Natural Sublanguages

Note that the results of Sects. 3 and 4 do not necessarily imply anything about the expressiveness of structural recursion in XQuery, our main motivation for considering structural recursion in the first place. Indeed, the expressions of $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ can create and manipulate arbitrary values (including e.g., lists of lists and list of pairs) during their computation, whereas XQuery expressions can only manipulate atoms, trees, lists of atoms, and lists of trees according to the XQuery data model [15][1]. We will nevertheless show in this section that $\mathcal{NTL}$ is a conservative extension of the non-recursive for-let-where-return fragment $\mathcal{XQ}$ of XQuery: a query from XQuery values to XQuery values is expressible in $\mathcal{NTL}$ if, and only if, it is already expressible in $\mathcal{XQ}$. As this conservativity continues to hold in the presence of (safe) structural recursion, our results of Section 3 and 4 immediately transfer to (safe) structural recursion in XQuery.

We also show that $\mathcal{NTL}$ is a conservative extension of $\mathcal{NLL}$ — a list-based interpretation of the nested relational language of Buneman et al. [8]. Whereas $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ expressions can create and manipulate arbitrary values (including trees), $\mathcal{NLL}$ can only manipulate *list-based complex objects* — values without trees. As this conservativity continues to hold in the presence of (safe) structural recursion, our results of Section 3 and 4 hence also transfer to (safe) structural recursion on list-based complex objects.

5.1 Structural Recursion in XQuery

An *XQuery type* is a type in which lists can only hold atoms or trees, as given by the following grammar:

$$s, t ::= atom \mid tree \mid s \times t \mid [atom] \mid [tree].$$

Let an *XQuery value* be a value in some XQuery type and let an *xquery* be a query from an XQuery type s to an XQuery type t.

For $V \subseteq \{\mathsf{srl}, \mathsf{srt}\}$ we define $\mathcal{XQ}(V)$ to be the natural sublanguage of $\mathcal{NTC}(V)$ in which we restrict expressions to only manipulate XQuery values. Formally, $\mathcal{XQ}(V)$ is the subset of $\mathcal{NTC}(V)$ expressions in which every subexpression e has type $e\colon t$ or $e\colon s \to t$ with s and t XQuery types. For instance,

$$\lambda x^{tree \times tree}.tree(lab(\pi_1 x), children(\pi_2 x))$$

is an expression in $\mathcal{XQ}$. The expression $\mathsf{srt}(f)$ from Example 7 that simulates *toc* from the Introduction is not in $\mathcal{XQ}(\mathsf{srl}, \mathsf{srt})$ however. Indeed, f has type $f\colon atom \times [[tree] \times tree] \times unit \to ([tree] \times tree)$, where $[[tree] \times tree]$ is not an XQuery type. Nevertheless, *toc* is expressible in $\mathcal{XQ}(\mathsf{srt})$, as the following proposition shows.

Proposition 27 *Let $V \subseteq \{\mathsf{srl}, \mathsf{srt}\}$. $\mathcal{NTL}(V)$ is a conservative extension of $\mathcal{XQ}(V)$ in the sense that every xquery definable by an expression in $\mathcal{NTL}(V)$ is definable by an expression in $\mathcal{XQ}(V)$. Similarly, safe $\mathcal{NTL}(V)$ is a conservative extension of safe $\mathcal{XQ}(V)$.*

[1]To be completely precise, XQuery expressions can only manipulate list of items, where every item is either an atom or a tree node. A single item is identified with the singleton list containing that item. To enable the mixture of atoms and trees in lists, the types in XQuery are based on so-called *regular expression types*. To keep our presentation simple, we have chosen to consider instead a standard type system and disallow mixture of atoms and trees.

Proof Observe that every value can be represented as a single tree. For example, $v = [(a, \mathit{tree}(b, [\,]))]$ of type $[\mathit{atom} \times \mathit{tree}]$ can be represented as $\mathit{to}(v) := \mathit{tree}(\mathsf{list}, [\mathit{tree}(\mathsf{pair}, [\mathit{tree}(a, [\,]), \mathit{tree}(b, [\,])])])$ where list and pair are arbitrarily fixed distinct atoms. We can then simulate any xquery $f\colon s \to t$ in $\mathcal{NTC}(V)$ by consistently replacing subexpressions that create non-XQuery values with subexpressions that create the tree representation of these values and by replacing subexpressions that operate on non-XQuery values with subexpressions that perform these operations on the corresponding tree representation. The detailed simulation is as follows.

For a general type s, let $\mathit{to}^s\colon s \to \mathit{tree}$ be the $\mathcal{NTC}$ expression that transforms values into their tree representations, defined by induction on s as follows:

- $\mathit{to}^{\mathit{unit}} = \lambda x.\mathit{tree}(\mathsf{unit}, [\,])$;
- $\mathit{to}^{\mathit{atom}} = \lambda x.\mathit{tree}(x, [\,])$;
- $\mathit{to}^{\mathit{tree}} = \lambda x.x$;
- $\mathit{to}^{s\times t} = \lambda(x, y).\mathit{tree}(\mathsf{pair}, [\mathit{to}^s(x)]{+\!\!+}[\mathit{to}^t(y)])$;
- $\mathit{to}^{[s]} = \lambda x.\mathit{tree}(\mathsf{list}, \mathsf{for}\ y\ \mathsf{in}\ x\ \mathsf{return}\ \mathit{to}^s(x))$.

Next, let $\mathit{from}^s\colon \mathit{tree} \to s$ be the $\mathcal{NTC}$ expression that transforms tree representations back into their original values, defined by induction on s as follows.

- $\mathit{from}^{\mathit{unit}} = \lambda x.()$;
- $\mathit{from}^{\mathit{atom}} = \lambda x.\mathit{lab}(x)$;
- $\mathit{from}^{\mathit{tree}} = \lambda x.x$;
- $\mathit{from}^{s\times t} = \lambda x.(\mathit{from}^s(\mathit{head}(\mathit{children}(x))), \mathit{from}^t(\mathit{head}(\mathit{tail}(\mathit{children}(x)))))$;
- $\mathit{from}^{[s]} = \lambda x.\mathsf{for}\ y\ \mathsf{in}\ \mathit{children}(x)\ \mathsf{return}\ \mathit{from}^s(y)$.

Clearly, *to* and *from* form a lossless encode-decode pair in the sense that $\mathit{from}^s \circ \mathit{to}^s = \mathit{id}^s$, for any type s.

Now define, for every expression e in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ the expression $\mathcal{X}[e]$ such that $\mathcal{X}[e]\colon \mathit{tree}$ if $e\colon t$ and $\mathcal{X}[e]\colon \mathit{tree} \to \mathit{tree}$ if $e\colon s \to t$ as follows.

- $\mathcal{X}[x^s] := x^{\mathit{tree}}$;
- $\mathcal{X}[a] := \mathit{tree}(a, [\,])$;
- $\mathcal{X}[\lambda x^s.e] := \lambda x^{\mathit{tree}}.\mathcal{X}[e]$;
- $\mathcal{X}[e_1 e_2] := \mathcal{X}[e_1]\mathcal{X}[e_2]$;
- $\mathcal{X}[()] := \mathit{tree}(\mathsf{unit}, [\,])$;
- $\mathcal{X}[\pi_1 e] := \mathit{head}(\mathit{children}(\mathcal{X}[e]))$;
- $\mathcal{X}[\pi_2 e] := \mathit{head}(\mathit{tail}(\mathit{children}(\mathcal{X}[e])))$;
- $\mathcal{X}[(e_1, e_2)] := \mathit{tree}(\mathsf{pair}, [\mathcal{X}[e_1]]{+\!\!+}[\mathcal{X}[e_2]])$;
- $\mathcal{X}[\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f] := \mathsf{if}\ \mathcal{X}[e_1] = \mathcal{X}[e_2]\ \mathsf{then}\ \mathcal{X}[e_t]\ \mathsf{else}\ \mathcal{X}[e_f]$;
- $\mathcal{X}[[\,]] := \mathit{tree}(\mathsf{list}, [\,])$;
- $\mathcal{X}[[e]] := \mathit{tree}(\mathsf{list}, \mathcal{X}[e])$;
- $\mathcal{X}[e_1{+\!\!+}e_2] := \mathit{children}(\mathcal{X}[e_1]){+\!\!+}\mathit{children}(\mathcal{X}[e_2])$;
- $\mathcal{X}[\mathit{head}(e)] := \mathit{head}(\mathit{children}(\mathcal{X}[e]))$;
- $\mathcal{X}[\mathit{tail}(e)] := \mathit{tail}(\mathit{children}(\mathcal{X}[e]))$;
- $\mathcal{X}[\mathsf{for}\ x^s\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2] := \mathit{tree}(\mathsf{list}, \mathsf{for}\ x^{\mathit{tree}}\ \mathsf{in}\ \mathit{children}(\mathcal{X}[e_1])\ \mathsf{return}$ $\mathit{children}(\mathcal{X}[e_2]))$;
- $\mathcal{X}[\mathit{tree}(e_1, e_2)] := \mathit{tree}(\mathit{lab}(\mathcal{X}[e_1]), \mathit{children}(\mathcal{X}[e_2]))$;

- $\mathcal{X}[lab(e)] := tree(lab(\mathcal{X}[e]), \texttt{[ ]})$;
- $\mathcal{X}[children(e)] := tree(\mathsf{list}, children(\mathcal{X}[e]))$;
- $\mathcal{X}[\mathsf{srl}(f)] \ := \ \lambda z^{tree}.\mathsf{srl}(\lambda(x^{tree}, y^{tree}).\mathcal{X}[f(x^s, y^t)])(\mathcal{X}[\pi_1 z^{[s]\times t}], \mathcal{X}[\pi_2 z^{[s]\times t}])$ when f has type $f\colon s \times t \to t$; and
- finally,

$$\mathcal{X}[\mathsf{srt}(f)] := \lambda u^{tree}.\mathsf{srt}\big(\lambda(x^{atom}, y^{[tree]}, z^{tree}).\mathcal{X}[f(x^{atom}, y^{[t]}, z^s)]\big)$$
$$(\mathcal{X}[\pi_1 z^{tree\times s}], \mathcal{X}[\pi_2 z^{tree\times s}])$$

  when f has type $f\colon atom \times [t] \times s \to t$.

It is straightforward to show by induction on e that for every superset $x, \dots, y$ of the free variables of e,

1. if $e\colon t$, then $to \circ (\lambda(x, \dots, y).e) \equiv \mathcal{X}[\lambda(x, \dots, y).e] \circ to$; and
2. if $e\colon s \to t$ then $to \circ (\lambda(x, \dots, y, z).ez) \equiv \mathcal{X}[\lambda(x, \dots, y, z).(ez)] \circ to$.

Hence, if $f\colon s \to t$ is a closed expression, then

$$f \equiv id^t \circ (\lambda z.fz) \equiv from^t \circ to^t \circ (\lambda z.fz) \equiv from^t \circ \mathcal{X}[\lambda z.fz] \circ to^s.$$

Observe that if f is an expression in $\mathcal{NTL}(V)$, then $\mathcal{X}[f]$ is an expression in $\mathcal{XQ}(V)$. Moreover, if s and t are XQuery types, then to^s and $from^t$ are $\mathcal{XQ}$ expressions. In other words, $from^t \circ \mathcal{X}[\lambda z.(fz)] \circ to^s$ is an expression in $\mathcal{XQ}(V)$ equivalent to f, which shows that every xquery definable by a closed $\mathcal{NTC}(V)$ expression is definable by a $\mathcal{XQ}(V)$ expression. Hence, $\mathcal{NTL}(V)$ is a conservative extension of $\mathcal{XQ}(V)$.

It remains to show that safe $\mathcal{NTL}(V)$ is a conservative extension of safe $\mathcal{XQ}(V)$. Hereto, first modify the definition of $\mathcal{X}[\cdot]$ as follows:

- $\mathcal{X}[\mathsf{for}\ y\ \mathsf{in}\ e\ \mathsf{return}\ children(y)] := tree(\mathsf{list}, \mathsf{for}\ y\ \mathsf{in}\ children(\mathcal{X}[e])\ \mathsf{return}\ children(y))$;
- $\mathcal{X}[\mathsf{srl}(\lambda(x^s, y^t).e)] \ := \ \lambda z^{tree}.\mathsf{srl}(\lambda(x^{tree}, y^{tree}).\mathcal{X}[e])(\mathcal{X}[\pi_1 z^{[s]\times t}], \mathcal{X}[\pi_2 z^{[s]\times t}])$; and
- $\mathcal{X}[\mathsf{srt}(\lambda(x^{atom}, y^{[tree]}, z^s))] := \lambda u^{tree}.\mathsf{srt}(\lambda(x^{atom}, y^{[tree]}, z^{tree}).\mathcal{X}[e])(\mathcal{X}[\pi_1 u^{tree\times s}], \mathcal{X}[\pi_2 u^{tree\times s}])$.

It is readily verified that equations (1) and (2) above continue to hold. Therefore, if $f\colon s \to t$ is a safe closed expression in $\mathcal{NTC}(V)$ with s and t XQuery types, then $from^t \circ \mathcal{X}[\lambda z.fz] \circ to^s$ is an equivalent expression in $\mathcal{XQ}(V)$. It remains to show that this expression is also safe. Since *to* and *from* do not use structural recursion, it suffices to show that every step expression occurring in $\mathcal{X}[\lambda z.f, z]$ is linear. Since all step expressions in f are linear and hence of the form $\lambda(x, y).e$ or $\lambda(x, y, z).e$ and since these step expressions are translated into step expressions of the form $\lambda(x, y).\mathcal{X}[e]$ and $\lambda(x, y, z).\mathcal{X}[e]$ respectively, it suffices to show that $\mathcal{X}[e]$ is linear if e is. We do so by induction on the linearity of e.

- $\mathcal{X}[e]$ is clearly linear if e is a variable x, atom a, the empty tuple $()$, or the empty list `[ ]`.

- If $e = [e']$, then $\mathcal{X}[e] = \mathit{tree}(\mathsf{list}, \mathcal{X}[e'])$. Since $\mathcal{X}[e']$ is linear, so is $\mathcal{X}[e]$.
- If $e = (e_1, e_2)$ with $FV(e_1) \cap FV(e_2) = \emptyset$ then

$$\mathcal{X}[e] = \mathit{tree}(\mathsf{pair}, [\mathcal{X}[e_1]] +\!\!+ [\mathcal{X}[e_2]]).$$

  This expression is linear since $\mathcal{X}[\cdot]$ preserves the set of free variables of an expression, and since $\mathcal{X}[e_1]$ and $\mathcal{X}[e_2]$ are linear by the induction hypothesis.
- If $e = e_1 +\!\!+ e_2$ or $e = \mathit{tree}(e_1, e_2)$ the reasoning is similar.
- If $e = \mathsf{for}\ x^s\ \mathsf{in}\ e_1\ \mathsf{return}\ x$ then

$$\mathcal{X}[e] = \mathit{tree}(\mathsf{list}, \mathsf{for}\ x\ \mathsf{in}\ \mathit{children}(\mathcal{X}[e_1])\ \mathsf{return}\ \mathit{children}(x)).$$

  This expression is easily seen linear since $\mathcal{X}[e_1]$ is linear by the induction hypothesis and since $\mathit{children}(\mathcal{X}[e_1]) = y[y/\mathit{children}(\mathcal{X}[e_1]), z/\mathit{lab}(\mathcal{X}[e_1])]$.
- If $e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ \mathit{children}(x)$ then

$$\mathcal{X}[e] = \mathit{tree}(\mathsf{list}, \mathsf{for}\ x\ \mathsf{in}\ \mathit{children}(\mathcal{X}[e_1])\ \mathsf{return}\ \mathit{children}(x)).$$

  This expression is easily seen linear since $\mathcal{X}[e_1]$ is linear by the induction hypothesis and since $\mathit{children}(\mathcal{X}[e_1]) = y[y/\mathit{children}(\mathcal{X}[e_1]), z/\mathit{lab}(\mathcal{X}[e_1])]$.
- If $e = \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f$, then

$$\mathcal{X}[e] = \mathsf{if}\ \mathcal{X}[e_1] = \mathcal{X}[e_2]\ \mathsf{then}\ \mathcal{X}[e_t]\ \mathsf{else}\ \mathcal{X}[e_f],$$

  which is linear by a straightforward application of the induction hypothesis.
- If $e = e_1[x/g(e_2), y/h(e_2)]$ with g and h distinct elements of $\{\pi_1, \pi_2, \mathit{head}, \mathit{tail}, \mathit{lab}, \mathit{children}\}$, then it is easily seen that

$$\mathcal{X}[e] = \mathcal{X}[e_1][x/\mathcal{X}[g(e_2)], y/\mathcal{X}[h(e_2)]].$$

  We show that this expression is linear by a case analysis on g and h. First observe, however, that since g and h must both be applicable to the same type t with $e_2 : t$, it is impossible for g to be π_1 and h to be *head* or for g to be *children* and h to be π_2, and so on. This leaves us with the following cases.

  1. Case $f = \pi_1$ and $g = \pi_2$. Then $\mathcal{X}[\pi_1(e_2)] = \mathit{head}(\mathit{children}(\mathcal{X}[e_2]))$ and $\mathcal{X}[\pi_2(e_2)] = \mathit{head}(\mathit{tail}(\mathit{children}(\mathcal{X}[e_2])))$. Then let z be a variable not occurring in e_1 or e_2. It is readily verified by induction on $\mathcal{X}[e_1]$ that $\mathcal{X}[e_1][y/\mathit{head}(z)]$ is also linear. Hence, so is

$$\mathcal{X}[e_1][y/\mathit{head}(z)][x/\mathit{head}(\mathit{children}(\mathcal{X}[e_2])), z/\mathit{tail}(\mathit{children}(\mathcal{X}[e_2]))],$$

     which equals $\mathcal{X}[e]$.
  2. The reasoning for $f = \mathit{head}$ and $g = \mathit{tail}$; $f = \mathit{lab}$ and $g = \mathit{children}$; and their symmetrical versions are similar. □

Combined with Theorems 12 and 26 and the fact that (safe) $\mathcal{XQ}(V)$ is a syntactical subset of (safe) $\mathcal{NTL}(V)$, we hence immediately obtain:

Corollary 28

1. *The class of queries expressible in* $\mathcal{XQ}(\mathsf{srl}, \mathsf{srt})$ *coincides with the class of xqueries that are computable in primitive recursive time.*
2. *The class of queries expressible in safe* $\mathcal{XQ}(\mathsf{srl}, \mathsf{srt})$ *coincides with the class of xqueries that are computable in polynomial time.*

5.2 Structural Recursion on List-Based Complex Objects

A *complex object type* is a type in which *tree* does not occur, as generated by the following grammar.

$$s, t ::= \mathit{unit} \mid \mathit{atom} \mid s \times t \mid [s].$$

Let a *complex object* be a value in some complex object type, and let a *complex object query* be a query from a complex object type s to a complex object type t.

For $V \subseteq \{\mathsf{srl}\}$ we define the *nested list language* $\mathcal{NLL}(V)$ to be the natural sublanguage of $\mathcal{NTL}(V)$ in which we restrict expressions to only create and manipulate complex objects. Formally, $\mathcal{NLL}$ is the subset of $\mathcal{NTL}$ expressions in which every subexpression e has type $e\colon s$ or $e\colon s \to t$ with s and t complex object types. For instance, $\langle Ka, [\ ] \circ ! \rangle$ is an $\mathcal{NLL}$ expression, but $\mathit{lab} \circ \mathit{tree}(a, [\ ])$ is not as $\mathit{lab}\colon \mathit{tree} \to \mathit{atom}$ and $\mathit{tree}(a, [\ ])\colon \mathit{tree}$ do not have complex object types.

Proposition 29 $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *is a conservative extension of* $\mathcal{NLL}(\mathsf{srl})$ *in the sense that every complex object query definable by an expression in* $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *is definable by an expression in* $\mathcal{NLL}(\mathsf{srl})$. *Similarly, safe* $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ *is a conservative extension of safe* $\mathcal{NLL}(\mathsf{srl})$.

Proof If $f\colon s \to t$ is a complex object query definable by an expression in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$, then f is primitive recursive by Proposition 13. Hence, by Proposition 15 there exists an expression $g\colon s \to t$ in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ that defines f by simulating a DTM M for f. Checking the proof of this proposition, we see that since s and t do not contain *tree* the expression g simulating M never constructs or inspects trees, nor does it use srt. In other words, g is an expression in $\mathcal{NLL}(\mathsf{srl})$ that defines f.

Similarly, if $f\colon s \to t$ is a complex object query definable by a *safe* expression in $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$, then f is computable in polynomial time by Theorem 26. In particular, there exists a safe expression $g\colon s \to t$ in $\mathcal{NTC}(\mathsf{srl}, \mathsf{srt})$ that defines f by simulating a DTM M for f as shown in the proof of Proposition 25. Checking this proof, we see that since s and t do not contain *tree* the expression g never constructs or inspects trees, nor does it use srt. In other words, g is a safe expression in $\mathcal{NLL}(\mathsf{srl})$ equivalent to f, from which the proposition follows. □

Combined with Theorems 12 and 26 and the fact that (safe) $\mathcal{NLL}(\mathsf{srl})$ is a syntactical fragment of (safe) $\mathcal{NTL}(\mathsf{srl}, \mathsf{srt})$ we hence immediately obtain:

Corollary 30

1. *The class of queries expressible in* $\mathcal{NLL}(\mathsf{srl})$ *coincides with the class of complex object queries that are computable in primitive recursive time.*

2. *The class of queries expressible in safe* $\mathcal{NLL}(\mathsf{srl})$ *coincides with the class of complex object queries that are computable in polynomial time.*

The fact that $\mathcal{NLL}(\mathsf{srl})$ captures the class of primitive recursive complex object queries may seem in contrast to the result of Grumbach and Milo [19] 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 essentially generalizes $\mathcal{NLL}(\mathsf{srl})$ has lower complexity. There is an error in their upper-bound proof, however; also non-elementary queries can be expressed [18].

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

We also note that in the restricted case of queries mappings tuples of lists of atomic data values to lists of atomic data values, the polynomial time queries were already captured by Bonner and Mecca in their work on Sequence Datalog [5].

6 Conclusions and Future Work

In conclusion, structural recursion is a powerful query primitive on lists and trees. Unrestricted, it captures the class of all primitive recursive queries, while by disallowing doubling of the recursive arguments we obtain the class of all polynomial time queries.

We should stress, however, that although our effective syntax for the polynomial time queries in terms of the safe expressions is satisfying from a theoretical point of view, it is unsatisfying from a programmer's point of view as it for instance prohibits the natural formulation of the transitive closure and table-of-contents queries as given in Examples 5 and 7. Indeed, the programmer must resort to Turing machine simulation to express these queries, which is clumsy at best. In the theory of programming languages there has been considerable research on type systems that ensure linear boundedness while still maintaining programmability [16, 20, 21, 23]. We therefore feel that a further investigation of the properties of these systems as they apply to database query languages in general and query languages on lists and trees in particular is warranted.

References

1. Abiteboul, S., Beeri, C.: The power of languages for the manipulation of complex values. VLDB J. **4**(4), 727–794 (1995)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
3. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions (extended abstract). In: STOC 1992, pp. 283–293. ACM Press, New York (1992)
4. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. W3C Recommendation, January 2007

5. Bonner, A.J., Mecca, G.: Sequences, datalog, and transducers. J. Comput. Syst. Sci. **57**(3), 234–259 (1998)
6. Boolos, G.S., Jeffrey, R.C.: Computability and Logic, 3rd edn. Cambridge University Press, Cambridge (1989)
7. Buneman, P., Fernandez, M.F., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. VLDB J. **9**(1), 76–110 (2000)
8. Buneman, P., Naqvi, S.A., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theor. Comput. Sci. **149**(1), 3–48 (1995)
9. Caseiro, V.-H.: Equations for defining poly-time functions. PhD thesis, University of Oslo (1997)
10. Chandra, A.K., Harel, D.: Computable queries for relational data bases. J. Comput. Syst. Sci. **21**(2), 156–178 (1980)
11. Cobham, A.: The intrinsic computational difficulty of functions. In: Logic, Methodology, and Philosophy of Science II, pp. 24–30. Springer, Berlin (1965)
12. Colby, L.S., Libkin, L.: Tractable iteration mechanisms for bag languages. In: ICDT 1997. Lecture Notes in Computer Science, vol. 1186, pp. 461–475. Springer, Berlin (1997)
13. Colby, L.S., Robertson, E.L., Saxton, L.V., Van Gucht, D.: A query language for list-based complex objects. In: PODS 1994, pp. 179–189. ACM Press, New York (1994)
14. Draper, D., Fankhauser, P., Fernández, M.F., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation, January 2007
15. Fernández, M.F., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 Data Model. W3C Recommendation, January 2007
16. Girard, J.-Y.: Light linear logic. Inf. Comput. **143**(2), 175–204 (1998)
17. Gladstone, M.D.: Simplifications of the recursion scheme. J. Symb. Log. **36**(4), 653–665 (1971)
18. Grumbach, S., Milo, T.: Personal communication
19. Grumbach, S., Milo, T.: An algebra for pomsets. Inf. Comput. **150**(2), 268–306 (1999)
20. Hofmann, M.: A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In: CSL 1997. Lecture Notes in Computer Science, vol. 1414, pp. 275–294. Springer, Berlin (1998)
21. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. In: LICS, pp. 464–473 (1999)
22. Hofmann, M.: Semantics of linear/modal lambda calculus. J. Funct. Program. **9**(3), 247–277 (1999)
23. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL, pp. 185–197 (2003)
24. Hull, R., Su, J.: Algebraic and calculus query languages for recursively typed complex objects. J. Comput. Syst. Sci. **47**(1), 121–156 (1993)
25. Immerman, N., Patnaik, S., Stemple, D.W.: The expressiveness of a family of finite set languages. Theor. Comput. Sci. **155**(1), 111–140 (1996)
26. Leivant, D.: Stratified functional programs and computational complexity. In: POPL 1993, pp. 325–333. ACM Press, New York (1993)
27. Libkin, L., Wong, L.: Query languages for bags and aggregate functions. J. Comput. Syst. Sci. **55**(2), 241–272 (1997)
28. Sazonov, V.Yu.: Hereditarily-finite sets, data bases and polynomial-time computability. Theor. Comput. Sci. **119**(1), 187–214 (1993)
29. Suciu, D.: Bounded fixpoints for complex objects. Theor. Comput. Sci. **176**(1–2), 283–328 (1997)
30. Suciu, D., Wong, L.: On two forms of structural recursion. In: ICDT 1995. Lecture Notes in Computer Science, vol. 893, pp. 111–124. Springer, Berlin (1995)