A Query Language for NC

Dan Suciu
AT&T Labs
suciu@research.att.com

Val Tannen*
University of Pennsylvania
val@cis.upenn.edu

Abstract

We show that a form of divide and conquer recursion on sets together with the relational algebra expresses exactly the queries over ordered relational databases which are NC-computable. At a finer level, we relate k nested uses of recursion exactly to AC^k, $k \geq 1$. We also give corresponding results for complex objects.

1 Introduction

NC is the complexity class of functions that are computable in poly-logarithmic time with polynomially many processors on a *parallel random access machine* (PRAM). The query language for NC discussed here is centered around a form of *divide and conquer recursion* (dcr) on finite sets which has obvious potential for parallel evaluation and can easily express, for example, transitive closure and parity. Divide and conquer with parameters e, f, u defines the unique function φ, notation $dcr(e, f, u)$, taking finite sets as arguments, such that:

$$\begin{aligned} \varphi(\emptyset) &\stackrel{\text{def}}{=} e \\ \varphi(\{y\}) &\stackrel{\text{def}}{=} f(y) \\ \varphi(s_1 \cup s_2) &\stackrel{\text{def}}{=} u(\varphi(s_1), \varphi(s_2)) \text{ when } s_1 \cap s_2 = \emptyset \end{aligned}$$

For parity, we take $e \stackrel{\text{def}}{=} \mathit{false}$, $f(y) \stackrel{\text{def}}{=} \mathit{true}$ and $u(v_1, v_2) \stackrel{\text{def}}{=} v_1 \ \mathit{xor}\ v_2$. To compute the transitive closure of some binary relation r, take $e \stackrel{\text{def}}{=} \emptyset$, $f(y) \stackrel{\text{def}}{=} r$ and $u(r_1, r_2) \stackrel{\text{def}}{=} r_1 \cup r_2 \cup r_1 \circ r_2$. Then, the transitive closure of r is obtained by applying φ to the set of nodes of the relation r, namely $tc(r) = \varphi(\Pi_1(r) \cup \Pi_2(r))$, where Π_1, Π_2 are the relational projections. In general, $dcr(e, f, u)$ is well-defined when there is some set containing e and the range of f, on which u is associative, commutative and has the identity e. For parity, this is the set $\mathbb{B}$ of booleans, while for transitive closure, it is the set $\{r \cup r^2 \cup \ldots \cup r^n \mid n \geq 0\}$.

We show that dcr together with the relational algebra expresses exactly the queries over *ordered* databases of flat relations that are NC-computable. We also show that a bounded version of dcr together with the nested relational algebra expresses exactly the queries over *ordered* databases of complex objects that are NC-computable. In fact, we prove the more refined versions that relate k nested uses of (bounded) dcr exactly to

*The authors were partially supported by NSF Grant CCR-90-57570 and ONR Contract NOOO14-93-11284

the subclass AC^k of NC where $k \geq 1$ (the definitions of these complexity classes are reviewed in Section 4). Some explanations are in order:

- Computable queries are in the sense of Chandra and Harel [CH80], with a natural extension to complex objects (Section 5).

- Any language that can express the same class of queries as first-order logic would do just as well as the relational algebra. Similarly for complex objects, where a corresponding class of tractable queries has emerged from several equivalent formalisms. Some of these formalisms are syntactically restricted higher-order logics, others are algebraic languages, often called nested relational algebras, hence our statement above. In fact, we will use the family of query languages introduced in [BBW92, BNTW95] because it is semantically related to *dcr* (Section 2).

- *dcr* and (nested) relational algebra have meaning over any (nested) relational database. But, as with all known characterizations of query complexity classes below *NP*, we know how to capture the entire *NC* only over ordered databases. Formally, we do this by extending the language with an order predicate.

- A bounded version of *dcr* is necessary over complex objects, otherwise queries of high complexity such as *powerset* will be expressible. The bounded version is obtained by intersecting the result with a bounding set at each recursion step (Section 3). This is similar to the bounded fixpoints studied in [Suc94], an idea due to Peter Buneman, and, as with fixpoints, over flat relations *dcr* can always be expressed through bounded *dcr* (Section 3).

We believe that these results are of interest from two angles.

∠ **Query language design.** *dcr* is a well-known construct. It appears under the name *pump*, in a language specifically designed for a parallel database machine, FAD [BBKV87]. Following FAD, but under the name *hom*, this construct was included in Machiavelli [OBB89] where it fit nicely into the language's type system. Called (a form of) *transducer*, it is part of SVP [PSV92], precisely in order to support divide and conquer parallelism. Some limitations of its theoretical expressive power were examined (under the name *hom*) by Immerman, Patnaik, and Stemple ([IPS91] Theorem 7.8). They also note that *dcr* is in *NC*.

As part of a larger group of researchers, we became interested in *dcr* because it fits into a natural hierarchy of query languages that share a common semantic basis, built around forms of structural recursion on collection types [BTS91, BTBN91, BBW92, BNTW95] (see Section 3). Theoretical studies of expressiveness, such as [Won93, LW94a, Suc94, LW94b, SW95] and the present paper help us with the choice and mix of primitives, as well as implementation strategies. In particular, *dcr* is at the core of a sub-language for which we are currently seeking efficient implementation techniques for a variety of parallel architectures.

∠ **Computational complexity.** Following Immerman and Vardi's influential result [Imm82, Var82, Imm86] that first-order logic with least fixed point captures exactly the *PTIME*-computable queries on flat relations over ordered databases, several characterizations of low complexity classes in terms of logics or algebras used in databases have been discovered with the hope that logical methods may give insights into the difficult problem of complexity class separation. We mention first a few of these characterizations which have had a direct influence on the work here.

For parallel complexity classes, Immerman [Imm89] shows that the class of finite and ordered relational structures recognizable in parallel time $t(n)$ (n is the size of the structure) on a certain CRCW (concurrent read - concurrent write) PRAM coincides with the class of structures definable by a first-order induction [Mos74] of depth up to $t(n)$. Denninghoff and Vianu [DV91] characterize *NC* in terms of a resource-restricted messagepassing model with parallel semantics which computes object-oriented queries. For complex object databases, Grumbach and Vianu [GV91b, GV91a, GV95] give a syntactic restriction of the ramified higher-order logic CALC which, together with inflationary fixpoints and in the presence of order, captures exactly the *PTIME*-computable complex-object queries. Suciu [Suc94] shows that, in the presence of order, the same class of queries is captured by the nested relational algebra augmented with an inflationary bounded fixpoint operator.

To the best of our knowledge, no characterization of parallel complexity classes of queries over complex objects has been given before. What is more likely to set our results apart, however, is the *intrinsic* nature of the language we are proposing: the semantics of *dcr* puts it naturally in *NC*; there is no need to impose logarithmic bounds on the number of iterations or recursion depth. Moreover, it can be shown that a different kind of recursion on sets, namely structural recursion on the insert presentation of sets ([BTS91]; notation *sri*; definitions reviewed in Section 3), together with the relational algebra expresses exactly the *PTIME*-computable queries on ordered databases[1]. This follows from results in [IPS91]; we state the corresponding result for complex objects in Proposition 6.7. Hence, at least over ordered databases, the difference between *NC* and *PTIME* boils down to two different ways of recurring on sets, divide and conquer vs. element by element.

Gurevich [Gur83] and Compton and Laflamme [CL90] characterize the *DLOGSPACE*- and respectively the NC^1-computable global functions on ordered finite relational structures as algebras with certain primitive recursion schemas. Compton and Laflamme capture NC^1 also with first-order logic augmented with BIT[2], and with an operator for *defining relations by primitive recursion*. The recursion forms used in these two papers are very different from *dcr* because they depend on some linear ordering of the underlying structures for their actual definition, and while *dcr* is a form of recursion on finite sets, the recursion forms in [Gur83, CL90] are on *notations* for elements of (linearly ordered) finite sets. In Section 3 we consider a related form of recursion on sets, *set-divide*, whose definition relies on an order relation of the base type. This is a recursion *on sets*, hence it is different from the recursion considered in [CL90], which is a recursion *on the elements* of the domain; *set-divide* is related in spirit to *set-reduce* [IPS91]. Clote [Clo90] gives related characterizations of most parallel complexity classes, of *arithmetical* functions however.

Since *dcr* can be defined for all structures, not just ordered ones, our characterization of *NC* is instead closer in style to the abovementioned fixpoint characterization of *PTIME* by Immerman and Vardi, or to Immerman's characterizations of *DLOGSPACE* and *NLOGSPACE* by first-order logic extended with deterministic and nondeterministic transitive closure [Imm87b, Imm87a]. We must warn the reader however about one sense in which our language is not as neat as these extensions of first-order logic. For *dcr* to be well-defined, the operations involved in it must satisfy certain algebraic identities (associativity, commutativity, identity) and this turns out to be an undecidable condition (in fact Π^0_1 complete; see Section 3). Of course, only a certain family of instances of *dcr* is actually needed in the simulations, and for these, the algebraic conditions always hold (Proposition 8.3). Hence, it is of theoretical interest that there is a decidable sub-language of *dcr* plus relational algebra which captures exactly *NC* in the presence of order. In practice, we have found it useful to provide special syntax for some instances of *dcr* in which the algebraic conditions are automatically satisfied, but we found it counterproductive to limit *dcr* to these instances, as other uses kept appearing.

This paper is organized as follows. Section 2 reviews some basic query language constructs. Section 3 introduces four different forms of recursions over sets (one of which is *dcr*) and establishes the relationships between them, then defines bounded versions of recursion on sets, which are necessary for controlling complexity when we work with nested sets. It also briefly discusses the undecidability of the well-definedness of the *dcr* construct. We briefly review the parallel complexity classes AC^k and *NC* in Section 4, and extend

[1] Of course, so does least fixpoint recursion, for example, but it is not a recursion *on* sets.

[2] A relation giving the binary representation of integers.

them to classes of queries in Section 5. The main results are stated and discussed in Section 6. We prove our main results by translating *dcr* into a series of equivalent constructs, which are of interest of their own: into order dependent forms of recursion (Section 7), and into iterations (Section 8). Sections 9 and 10 complete the proofs.

2 Query Languages: the Background

Our language for the *NC*-computable queries over flat relations consists of a background language, equivalent to the relational algebra, and a form of recursion on sets, *dcr*. For complex objects, we also have a background language, this one equivalent to the nested relational algebra, and a form of recursion on sets called *bounded dcr*. The reader interested only in flat relations may skip the complex-object part; some of the proofs in this paper however are easier to understand in the more general context of complex objects than in that of flat relations.

We could have simply chosen the relational algebra or, equivalently, first order logic (*FO*) as our background language for flat relations. But we do not do so, because there is a mismatch between the syntax of, say, *FO*, and the best notation for *dcr*. To illustrate the point, note that $dcr(e, f, u)$ is a *higher-order* construct, expecting *functions* f and u as arguments. Such functions are cumbersome to denote in *FO*. Indeed *FO* is well equipped to express *queries* which are functions taking relations as inputs and returning a relation as output, but the input relation names are fixed by the input schema. In order to extend *FO* with higher order constructs like *dcr*, one needs a notation for defining arbitrary *functions*. In fact, a similar situation arises for the extension of *FO* with fixpoints, another higher-order construct [AHV95]. The usual notation for a fixpoint is of the form $\mathit{fix}_T(\varphi(T))$ where $\varphi(T)$ is a formula with m free variables, in which T is a new m-ary relation name. The argument of the fixpoint is the query $\varphi(T)$, and the symbol fix_T is needed to single out the name, T, of the input relation to that query. This works fine for fixpoints, because they take only a single query as argument, and, moreover, this query takes a single relation as input. For our *dcr* however, f is not a traditional query, because its input is an atomic value, or a tuple of atomic values, rather than a relation, and moreover u is a query expecting two relations as inputs. Extending the syntax of *FO* in the same way as for fixpoints would lead us to the following notation for $dcr(e, f, u)$: $dcr_{x_1,\ldots,x_k,T_1,T_2}(\gamma, \psi(x_1, \ldots, x_k), \varphi(T_1, T_2))$ where γ is a formula with m free variables (denoting e), $\psi(x_1, \ldots, x_k)$ is a formula with $m + k$ free variables, denoting the function f in which $x_1, \ldots, x_k$ are the inputs, and $\varphi(T_1, T_2)$ is again a formula with m free variables and two additional m-ary relation names T_1 and T_2, denoting the query u. We found this notation cumbersome. Instead we chose to follow [BTBN91, BBW92, BNTW95] in presenting a language equivalent to the relational algebra, with a concise notation for functions. This will make the addition of a syntax for *dcr* a simple task. Moreover, the presentation in [BNTW95] is such that the transition to complex objects is natural and straightforward.

Flat relations, and their extensions to complex objects are built essentially from tuples and finite sets. To describe them, we define the **complex object types** by the grammar:

$$t \stackrel{\text{def}}{=} \mathbb{D} \mid \mathbb{B} \mid t \times \ldots \times t \mid \{t\}$$

where $\mathbb{D}$ is some base type, which we assume to be infinite and denumerable, and $\mathbb{B}$ is the type of booleans [3]. The values of type $t_1 \times \ldots \times t_k$ are k-tuples $\langle x_1, \ldots, x_k \rangle$ with x_i a value of type t_i, for $i = 1, k$. The values of type $\{t\}$ are finite sets $\{x_1, \ldots, x_n\}$ of values of type t. We denote with *unit* the empty product, obtained by taking $k = 0$ in $t_1 \times \ldots \times t_k$: it has a single value, the empty tuple denoted $\langle\rangle$. E.g. the type of finite

[3] Not really necessary, could have been encoded as $\{\mathit{unit}\}$ [BNTW95].

binary relations is $\{\mathbb{D} \times \mathbb{D}\}$, and is populated with relations like $\{\langle a, c\rangle, \langle b, a\rangle, \langle a, d\rangle\}$ with $a, b, c, d \in \mathbb{D}$. The type $\{\mathbb{D} \times \{\mathbb{D}\}\}$ contains nested relations, like $\{\langle a, \{b, c\}\rangle, \langle b, \{\}\rangle\}$.

A **scalar type** is either one of the types $\mathbb{D}, \mathbb{B}$, or a product of two scalar types. A **flat relation type**, or a flat type in short, is either a type of the form $\{t\}$, with t a scalar type, or a product of two flat types. E.g. $\mathbb{D} \times \mathbb{B} \times \mathbb{D}$ is a scalar type, while $\{\mathbb{D} \times \mathbb{D}\}$ and $\{\mathit{unit}\} \times \{\mathbb{D} \times \mathbb{B}\}$ are flat types. Relational algebra and first order logic deal only with values of flat types.

Preparing the way for our higher order constructs, like *dcr* and its relatives, we consider also **function types**. These have the form $t_1 \to t_2$, where t_1 and t_2 are complex object types [4]. The syntactic construct for functions of type $t_1 \to t_2$ is $\lambda x^{t_1}.e$, where x^{t_1} is a variable of type t_1 and e is an expression of type t_2. E.g. $\lambda x^{\{\mathbb{N}\}}.x^{\{\mathbb{N}\}} \cup \{5\}$ denotes the function which inserts 5 into a set of natural numbers. This will make it easy to introduce *dcr* later on as a construct $dcr(e, f, u)$ where $e : t_2$ is a complex object expression, and $f : t_1 \to t_2, u : t_2 \times t_2 \to t_2$ are function expressions.

We present now the formalization of the *nested relational algebra*, $\mathcal{NRA}$, that was essentially introduced in [BNTW95] and is based on what is called there the *monad calculus*. This language has the same expressive power (see [Won94]) as Thomas and Fischer's algebra [TF86], hence also as Schek and Scholl's NF^2 relational algebra [SS86], or Paredaens and Van Gucht's *nested algebra* [PG88, PG92], or Abiteboul and Beeri's *algebra without powerset* [AB88]. When restricted to flat relations, it is equivalent to relational algebra.

The basic idea, which proved quite fruitful, is to design query languages for complex objects by considering tuples and sets as orthogonal. Hence, there will be primitives that work on tuples, primitives that work on sets, and general primitives for combining other primitives.

We assume an infinite set of variables to be given, each having a complex object type associated with it. We write x^t for a variable of type t. As usual, we distinguish between free and bound variables, and we identify those expressions that differ only in the name of some bound variables. $\mathcal{NRA}$ is defined by the expressions in Figure 2, presented as part of their typing rules.

We briefly describe the semantics of the expressions: $\langle e_1, \ldots, e_k\rangle$ constructs a k-tuple, $\pi_i^{t_1,\ldots,t_k}$ are the projections, $\{e\}$ is the singleton set, $\mathit{empty?}(e)$ returns true iff $e = \emptyset$, $f(e)$ is function application, and $\mathit{ext}(f)(\{x_1, \ldots, x_n\}) \stackrel{\mathrm{def}}{=} f(x_1) \cup \ldots \cup f(x_n)$. Finally, $\mathit{get}\langle s, y\rangle$ gives x if s is the singleton $\{x\}$ and y otherwise; this is a rather unusual operation in the context of a database query language; we introduce it for technical reasons, and we will comment on it shortly. We drop type superscripts, writing π_i and x instead of $\pi_i^{t_1,\ldots,t_k}$ and x^t, when no confusion arises. We denote $e[e'/x]$ the result of substituting e' for x in e.

A possible set Σ of **external functions** $p : \mathit{dom}(p) \to \mathit{codom}(p)$ could be added to the language; in this case, we denote the language by $\mathcal{NRA}(\Sigma)$.

To illustrate with some simple examples, consider the following[5]:

$$\begin{aligned} \mathit{pairwith}\langle x, s\rangle &\stackrel{\mathrm{def}}{=} \mathit{ext}(\lambda y.\{\langle x, y\rangle\})(s) \\ \mathit{product}\langle s, s'\rangle &\stackrel{\mathrm{def}}{=} \mathit{ext}(\lambda x.\mathit{pairwith}\langle x, s'\rangle)(s) \end{aligned}$$

[4] Note that we never allow functions to take other functions as arguments. Thus, in the parlance of the theory of functional programming languages, our language is first-order; in that of logic however, it is higher-order since it manipulates complex objects.

[5] Here *not* e is an abbreviation for **if** e **then** *false* **else** *true*; *true* can be taken to be $\mathit{empty?}(\emptyset)$, while $\mathit{false} \stackrel{\mathrm{def}}{=} \mathit{empty?}(\{\langle\rangle\})$. Also, we prefer to define functions using the notation $\mathit{pairwith}\langle x, s\rangle \stackrel{\mathrm{def}}{=} \mathit{ext}(\lambda y.\{\langle x, y\rangle\})(s)$ instead of the official $\mathit{pairwith} \stackrel{\mathrm{def}}{=} \lambda z.\mathit{ext}(\lambda y.\{\langle \pi_1(z), y\rangle\})(\pi_2(z))$.

$$\frac{}{x^t : t} \qquad \frac{e_1 : t_1 \quad \ldots \quad e_k : t_k}{\langle e_1, \ldots, e_k \rangle : t_1 \times \ldots \times t_k}$$

$$\frac{e : t_1 \times \ldots \times t_k}{\pi_i^{t_1,\ldots,t_k}(e) : t_i}(i = 1, k)$$

$$\frac{}{\emptyset : \{t\}} \qquad \frac{e : t}{\{e\} : \{t\}} \qquad \frac{e_1 : \{t\} \quad e_2 : \{t\}}{e_1 \cup e_2 : \{t\}}$$

$$\frac{e_1 : \mathbb{D} \quad e_2 : \mathbb{D}}{e_1 = e_2 : \mathbb{B}} \qquad \frac{}{\langle\rangle : \mathit{unit}}$$

$$\frac{e : \{t\}}{\mathit{empty?}(e) : \mathbb{B}} \qquad \frac{e : \mathbb{B} \quad e_1 : t \quad e_2 : t}{\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 : t}$$

$$\frac{e : t_2}{\lambda x^{t_1}.e : t_1 \to t_2} \qquad \frac{f : t_1 \to t_2 \quad e : t_1}{f(e) : t_2} \qquad \frac{f : t_1 \to \{t_2\}}{\mathit{ext}(f) : \{t_1\} \to \{t_2\}}$$

$$\frac{}{\mathit{get} : \{\mathbb{D}\} \times \mathbb{D} \to \mathbb{D}}$$

Figure 1: The definition of the Nested Relational Algebra $\mathcal{NRA}$

$$\begin{aligned} \mathit{member}\langle x, s\rangle &\stackrel{\text{def}}{=} \mathit{not}(\mathit{empty?}(\mathit{ext}(\lambda y.\textbf{if } x = y \textbf{ then } \{\langle\rangle\} \textbf{ else } \emptyset)(s))) \\ \mathit{difference}\langle s, s'\rangle &\stackrel{\text{def}}{=} \mathit{ext}(\lambda x.\textbf{if } \mathit{member}\langle x, s'\rangle \textbf{ then } \{x\} \textbf{ else } \emptyset)(s) \end{aligned}$$

Here $\mathit{pairwith}\langle x, \{y_1, \ldots, y_n\}\rangle = \{\langle x, y_1\rangle, \ldots, \langle x, y_n\rangle\}$, $\mathit{product}\langle s, s'\rangle = s \times s'$ computes the cartesian product of s and s'.

In fact, it is easy to see that $\mathcal{NRA}$ can express all operations in the relational algebra. But, besides being able to deal with complex objects, $\mathcal{NRA}$ adds to the relational algebra the ability of dealing with scalar values. The ability to treat uniformly relations and scalar values is of importance to us, because, as we saw, the functions f and u in $\mathit{dcr}(e, f, u)$ take scalar values, and flat relations respectively, as inputs. Therefore in defining queries we again depart slightly from the traditional approach, where a query f is a function taking relations as inputs and returning a relation as an output, i.e. has type $f : \{t_1\} \times \ldots \times \{t_k\} \to \{t\}$. Instead we define a **query** in $\mathcal{NRA}$ to be any closed[6] function expression $f : t_1 \to t_2$. This is only a slight generalization, in that it allows queries to also take scalar values as inputs, and return both relations and/or scalar values as outputs.

All queries in $\mathcal{NRA}$ are *generic*: see appendix A for a review of the definition.

As we said earlier, the reader interested only in flat relations and not in complex objects may skip the part dealing with complex objects. Formally, we consider a certain fragment of $\mathcal{NRA}$ which has the same expressive power as relational algebra. First we define the **set height** of a type t as follows:

$$\begin{aligned} \mathit{height}(\mathbb{D}) &\stackrel{\text{def}}{=} \mathit{height}(\mathbb{B}) \stackrel{\text{def}}{=} \mathit{height}(\mathit{unit}) \stackrel{\text{def}}{=} 0 \\ \mathit{height}(t_1 \times \ldots \times t_k) &\stackrel{\text{def}}{=} \max(\mathit{height}(t_1), \ldots, \mathit{height}(t_k)) \\ \mathit{height}(\{t\}) &\stackrel{\text{def}}{=} 1 + \mathit{height}(t) \end{aligned}$$

[6] I.e. with no free variables.

Thus, $height(t) = 0$ iff t is a scalar type, and $height(t) \leq 1$ iff t is a product of scalar types and flat types. The fragment that interests us is $\mathcal{NRA}^1$, defined as the restriction of $\mathcal{NRA}$ to types of set height ≤ 1, i.e. the only types allowed in $\mathcal{NRA}^1$ as inputs, outputs and intermediate types are products of base and flat types. Indeed:

Theorem 2.1 ([BTBN91]) *The set of $\mathcal{NRA}^1$ queries $f : t_1 \to t_2$ with t_1, t_2 flat types coincides with the set of queries expressible in the Relational Algebra and, hence, in First Order Logic.*

For all practical purposes $\mathcal{NRA}^1$ is equivalent to relational algebra. In the rest of the paper we will consider in parallel both $\mathcal{NRA}^1$ (i.e. flat relations) and $\mathcal{NRA}$ (i.e. complex objects).

Finally we comment on the operation *get* which we included in $\mathcal{NRA}$. It is obviously a generic function, in the sense of Definition A.1, and it is computable in NC, in the sense of Definition 5.2. Hence it must be expressible in our language for NC. In fact *get* turns out to be expressible with *dcr* (Example 3.3). But this is unsatisfactory for us, because it ruins our correspondence between the classes AC^k and the nesting depth of *dcr*. Worse, when we go to complex objects and replace *dcr* with a *bounded dcr*, *get* is no longer expressible (Proposition C.1). To simplify the exposition, we chose to include *get* in all our languages.

3 Query Languages: Recursion on Sets

In this section we discuss the properties of the divide and conquer construct, *dcr*, which is at the core of our query languages for NC, in the context of related operations.

We also discuss three additional forms of recursion on sets (two of which are *structural* recursions) and the relationship between them. While *dcr* corresponds to NC, another form, *element-step-recursion*, *esr*, corresponds to $PTIME$. When dealing with complex objects instead of flat relations we need to add an additional twist to these structural recursions to make them still capture NC and $PTIME$ respectively: we call the resulting forms of recursion *bounded* recursions.

We start by formally defining these forms of recursion, establish the relationships between them, and between the bounded and unbounded versions, and finally discuss the complexity of checking the "side-conditions" associated to recursions on sets.

3.1 Forms of recursion on sets

There seem to be two basic ways of describing the structure of finite sets. In one way, they are generated by finitely many (maybe zero!) binary unions of singleton sets. We call this the *union presentation*. In another way, they are generated by finitely many insertions of one element, starting with the empty set. We call this the *insert presentation*. Recognizing the relevant algebraic identities satisfied by union (associativity, commutativity, idempotence, has $\emptyset$ as an identity) and by element insertion (left-commutativity and left-idempotence) gives us two different algebraic structures on finite sets. Both these algebras are characterized by universality properties, which amount to definitions of functions by *structural recursion* [BTS91, BTBN91]. We have a structural recursion construct on the union presentation, *sru*:

$$\frac{e : t_2 \quad f : t_1 \to t_2 \quad u : t_2 \times t_2 \to t_2}{sru(e, f, u) : \{t_1\} \to t_2}$$

$$\begin{aligned}
sru(e, f, u)(\emptyset) &\stackrel{\text{def}}{=} e \\
sru(e, f, u)(\{y\}) &\stackrel{\text{def}}{=} f(y) \\
sru(e, f, u)(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle sru(e, f, u)(s_1), sru(e, f, u)(s_2)\rangle
\end{aligned}$$

$sru(e, f, u)$ is well-defined when there is some subset of t_2 containing e and the range of f, on which u is associative, commutative, idempotent, and has the identity e. We also have a structural recursion construct on the insert presentation, sri (we denote by $x \nearrow s$ the element insertion operation, $\{x\} \cup s$):

$$\frac{e : t_2 \quad i : t_1 \times t_2 \to t_2}{sri(e, i) : \{t_1\} \to t_2}$$

$$\begin{aligned}
sri(e, i)(\emptyset) &\stackrel{\text{def}}{=} e \\
sri(e, i)(y \nearrow s) &\stackrel{\text{def}}{=} i\langle y, sri(e, i)(s)\rangle
\end{aligned}$$

$sri(e, i)$ is well-defined when there is some subset of t_2 containing e on which i is *left-commutative*,

$$i\langle x, i\langle y, s\rangle\rangle = i\langle y, i\langle x, s\rangle\rangle$$

and *left-idempotent*

$$i\langle x, i\langle x, s\rangle\rangle = i\langle x, s\rangle$$

We can now present our central operation, divide and conquer recursion, as a "no duplicates" variation of sru. As was already sketched in Section 1:

$$\frac{e : t_2 \quad f : t_1 \to t_2 \quad u : t_2 \times t_2 \to t_2}{dcr(e, f, u) : \{t_1\} \to t_2}$$

$$\begin{aligned}
dcr(e, f, u)(\emptyset) &\stackrel{\text{def}}{=} e \\
dcr(e, f, u)(\{y\}) &\stackrel{\text{def}}{=} f(y) \\
dcr(e, f, u)(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle dcr(e, f, u)(s_1), dcr(e, f, u)(s_2)\rangle \\
&\quad \text{when } s_1 \cap s_2 = \emptyset
\end{aligned}$$

$dcr(e, f, u)$ is defined only when u is associative, commutative and has e as identity, on some subset of t_2 containing $Im(f)$.

If $sru(e, f, u)$ is well-defined then so is $dcr(e, f, u)$ and they are equal. But dcr is potentially more expressive, since u need not be idempotent. In fact, it is open whether $\mathcal{NRA}^1(sru)$, or even $\mathcal{NRA}^1(sru, \leq)$ can express parity or transitive closure. However, over ordered databases sru together with transitive closure has the same expressive power as dcr, Proposition 7.4.

One can also define a no duplicates variant of *sri*, let's call it *element-step recursion*, *esr*. This is like *sri*, with the second clause modified as:

$$esr(e,i)(y \nearrow s) \quad \stackrel{\text{def}}{=} \quad i\langle y, esr(e,i)(s)\rangle \text{ when } y \notin s$$

where i is required to be left-commutative (but not necessarily left-idempotent). Obviously, *esr* can express *sri* [7]. The non-immediate relationships between the four forms of recursion on sets are contained in:

Proposition 3.1 ([BTS91, BT92]) *In the presence of the operations of $\mathcal{NRA}$: sri can express sru and similarly esr can express dcr; moreover, sri can express esr. All this can be done without increase in set height and with at most polynomial overhead.*

Schematically:

$$sru \quad \subseteq \quad dcr \quad \subseteq \quad esr \quad = \quad sri.$$

For the case of flat relations, the statement becomes:

Corollary 3.2 $\mathcal{NRA}^1(sru) \subseteq \mathcal{NRA}^1(dcr) \subseteq \mathcal{NRA}^1(esr) = \mathcal{NRA}^1(sri)$.

Example 3.3 *We show here that get can be defined with sru. Indeed, get* $\stackrel{\text{def}}{=} \lambda\langle s,y\rangle.\pi_2(\varphi(s))$ *where , for each* $y \in \mathbb{D}$*, we define define* $\varphi : \{\mathbb{D}\} \to \{\mathbb{D}\} \times \mathbb{D}$ *to be* $\varphi \stackrel{\text{def}}{=} sru(e,f,u)$*, with:*

$$\begin{aligned}
e &\stackrel{\text{def}}{=} \langle \emptyset, y\rangle \\
f(x) &\stackrel{\text{def}}{=} \langle \{x\}, x\rangle \\
u\langle\langle s_1,x_1\rangle,\langle s_2,x_2\rangle\rangle &\stackrel{\text{def}}{=} \textbf{if } empty?(s_1) \textbf{ then } \langle s_2,x_2\rangle \\
&\quad\;\; \textbf{else if } empty?(s_2) \textbf{ then } \langle s_1,x_1\rangle \\
&\quad\;\; \textbf{else if } s_1 = s_2 \wedge card(s_1) = 1 \textbf{ then } \langle s_1,x_1\rangle \\
&\quad\;\; \textbf{else } \langle s_1 \cup s_2, y\rangle
\end{aligned}$$

Note that the test $card(s_1) = 1$ *is expressible in* $\mathcal{NRA}^1$*. Then u is associative, commutative and idempotent on the subset* $\{\langle\{x\},x\rangle \mid x \in \mathbb{D}\} \cup \{\langle s,y\rangle \mid card(s) \neq 1\}$ *of* $\{\mathbb{D}\} \times \mathbb{D}$*, and this set also contains e and Im(f). Hence* φ *is well defined.*

As a consequence get can be expressed with any of the four forms of recursion in the presence of the other $\mathcal{NRA}$ *operations.*

We have thus reached the language $\mathcal{NRA}^1(dcr)$, $k \geq 0$. Adding a linear order predicate to this language gives us the subject of one of our main theorems, the characterization of the *NC*-computable queries on ordered flat databases (Theorem 6.1).

[7] *sru* and *sri* are easier to reason about than *dcr* or *esr* because they define functions that preserve the algebraic structure, i.e. *homomorphisms*, hence the "structural" in their names. A good way to think about $dcr(e,f,u)$ is as the composition of the canonical coercion from sets to bags followed by the structural recursion on the sum presentation of bags [BTS91], with parameters e, f, u. Similarly, *esr* can be expressed via structural recursion on the increment presentation of bags.

In the same theorem, we obtain a finer characterization, involving the AC-hierarchy, for which we need the notion of **the depth of recursion nesting** $depth(e)$, of some expression e. We define this to be to be the maximum depth of recursions occurring in e. More precisely, $depth(dcr(e, f, u)) \stackrel{\text{def}}{=} \max(depth(e), depth(f), 1 + depth(u))$ (only u is actually iterated), $depth(\langle e_1, \ldots, e_k\rangle) \stackrel{\text{def}}{=} \max(depth(e_1), \ldots, depth(e_k))$, $depth(f(e)) \stackrel{\text{def}}{=} \max(depth(f), depth(e))$, etc. We denote $\mathcal{NRA}^1(dcr^{(k)})$ the restrictions of the language $\mathcal{NRA}^1(dcr)$ to recursion depth $\leq k$. We have thus obtained the hierarchy of languages $\mathcal{NRA}^1(dcr^{(k)})$. Adding a linear order predicate to these languages gives us the subject of the finer characterization given in Theorem 6.1.

The reader may have noticed that some redundancy in expressibility will appear when we add dcr. Indeed, it turns out that $ext(f)$ can be expressed with sru (and hence with dcr) as $sru(\emptyset, \lambda x.\{x\}, \cup)$. It is important however to keep $ext(f)$ as a separate construct in the language because the expression derived through dcr would be computed in $\log n$ parallel steps, when in fact a direct one-step parallel computation is possible: obtain in parallel and independently $f(x_1), \ldots, f(x_n)$, and then take their union to compute $ext(f)(\{x_1, \ldots, x_n\})$.

3.2 Bounded recursion

The four forms of recursion discussed so far do their job on flat relations. The reader interested in complex objects however, may have noticed that each of them can express *powerset* on complex objects, and hence are too powerful for languages intended to capture complexity classes like NC or $PTIME$. All four forms of recursion can simulate each other, using **powerset** (at the cost of a high complexity), hence $\mathcal{NRA}(sru) = \mathcal{NRA}(dcr) = \mathcal{NRA}(sri) = \mathcal{NRA}(esr)$ and they have essentially the same expressive power as Abiteboul and Beeri's *algebra* [BNTW95]. To keep our languages for complex objects tractable, we will define *bounded* versions of these recursions, an analog to Peter Buneman's idea of *bounded fixpoints* [Suc94], and also related to Sazonov's boundedness for a language without constants [Saz93].

A **PS-type** (product of sets type) is either a set type, or a product of PS-types. E.g. $\{\mathbb{D} \times \{\mathbb{D}\}\} \times \{unit\}$ and $\{\mathbb{D}\} \times (\{\mathbb{B} \times \mathbb{B}\} \times \{\mathbb{D}\})$ are PS-types, while $\mathbb{D} \times \{\mathbb{D}\}$ is not. Set-theoretic operations like $\cup, \cap, \subseteq$ extend to PS-types component-wise.

The **bounded** version of dcr is defined by:

$$\frac{e : t_2 \quad f : t_1 \to t_2 \quad u : t_2 \times t_2 \to t_2 \quad b : t_2}{bdcr(e, f, u, b) : \{t_1\} \to t_2}$$

with the restriction that t_2 is a PS-type, and with the semantics:

$$bdcr(e, f, u, b) \stackrel{\text{def}}{=} dcr(e \cap b, f \cap b, u \cap b)$$

Here $(u \cap b)\langle s_1, s_2\rangle \stackrel{\text{def}}{=} u\langle s_1, s_2\rangle \cap b$. As for dcr, we define bounded versions for the other forms of recursions on sets, $bsru, bsri, besr$. Proposition 3.1 easily extends to the bounded versions of recursion:

Corollary 3.4 $\mathcal{NRA}(bsru) \subseteq \mathcal{NRA}(bdcr) \subseteq \mathcal{NRA}(bsri) = \mathcal{NRA}(besr)$.

Over flat relations the explicit bounding is unnecessary: we will prove in Appendix B that $\mathcal{NRA}^1(bdcr)$ and $\mathcal{NRA}^1(dcr)$ have the same expressive power:

Proposition 3.5 *$\mathcal{NRA}^1(bdcr) = \mathcal{NRA}^1(dcr)$. Moreover, the equivalence preserves the nesting depth of iterations, i.e. $\forall k \geq 0$, $\mathcal{NRA}^1(bdcr^{(k)}) = \mathcal{NRA}^1(dcr^{(k)})$. Similarly for sri.*

Note that, if we drop *get* from $\mathcal{NRA}$, then this proposition fails, since $\mathcal{NRA}^1(dcr)$ can express *get* (Example 3.3), while $\mathcal{NRA}^1(bdcr)$ cannot (Proposition C.1). However, they still express the same set of queries $f : t \to t'$, where t' is a PS-type. Also, Proposition 3.5 fails in the presence of certain external functions, see Proposition 6.3.

3.3 The cost of order-independence

The algebraic conditions, besides the fact that they arise from principled mathematical characterizations of finite sets, provide us with an elegant alternative for ensuring the well-definedness of various forms of recursion on sets. Unfortunately, for a language at least as expressive as first-order logic, verifying in general most of these identities is as hard as testing the validity of a first-order formula in all finite models. We prove:

Theorem 3.6 *Deciding whether some expression $dcr(e, f, u)$ in $\mathcal{NRA}^1(dcr)$ is well defined is Π^0_1 complete. Similar results hold for sri, as well as for their bounded versions.*

As a consequence it is undecidable whether an arbitrary expression is a correct $\mathcal{NRA}^1(dcr)$ expression. To prove this result we use Trakhtenbrot's theorem (see, for example [Fag93]):

Theorem 3.7 (Trakhtenbrot) *Assume that a first order language $\mathcal{L}$ contains some relation symbol that is not unary. Then the set of first-order sentences over $\mathcal{L}$ valid in all finite structures is Π^0_1-complete.*

Proof. (of Theorem 3.6) First we show that checking well-definedness is co-r.e. For some values of the free variables in e, f, u, let A denote the set obtained by closing $\{e\} \cup Im(f)$ under applications of u. The expression $dcr(e, f, u)$ is well defined iff, for any values of the free variables in e, f, u, the function u is associative, commutative, and has identity e, on the set A. Since A is enumerable, testing whether $dcr(e, f, u)$ is not well defined is obviously r.e. Now we show that well-definedness is co-r.e. complete (Π^0_1-complete), by reducing the validity problem over finite models to the decision whether some expression $dcr(e, f, u)$ is well defined. Let $\mathcal{L}$ be a first order language having at least some non unary relation symbol. To keep our argument simple, suppose $\mathcal{L}$ has exactly one binary symbol R. Since first order logic is equivalent to the relational algebra, and the latter is essentially equivalent to $\mathcal{NRA}^1$, we conclude that for any sentence φ in $\mathcal{L}$ there exists a query $g : \{\mathbb{D}\} \times \{\mathbb{D} \times \mathbb{D}\} \to \mathbb{B}$ in $\mathcal{NRA}^1$ such that φ is true in the finite model (D, R), $R \subseteq D \times D$, iff $g(D, R) = true$. Then consider the following expression $dcr(e, f, u) : \{\mathbb{D}\} \to \{\mathbb{D}\}$ with free variables D, R: $e \stackrel{\text{def}}{=} \emptyset$, $f \stackrel{\text{def}}{=} \lambda x^{\mathbb{D}}.\{x^{\mathbb{D}}\}$, $u(s_1, s_2) \stackrel{\text{def}}{=}$ **if** $g(D \cup \Pi_1(R) \cup \Pi_2(R), R)$ **then** $s_1 \cup s_2$ **else** $s_1 - s_2$. When φ is valid in all finite models, then $g(D \cup \Pi_1(R) \cup \Pi_2(R))$ is true for all values of the free variables D and R, hence the function u coincides with union. In this case $dcr(e, f, u)$ is well defined, and is actually the identity function. On the other hand, if φ is not true in some finite model (D, R), then for that particular value of the free variables D and R the function u coincides with set difference, and hence $dcr(e, f, u)$ is not well defined. $\square$

4 Complexity Classes

We review here the definitions of the complexity classes AC^k and NC.

For some $W \in \{0,1\}^*$ we denote *length*(W) the length of the string W. Let $F : \{0,1\}^* \to \{0,1\}^*$.

Definition 4.1 ([RB90], pp. 766) *We say that F is in AC^k, for $k \geq 0$ iff the following conditions are met:*

1. *There is some polynomial $Q(n)$ s.t. $\forall W \in \{0,1\}^*$, $length(f(W)) = Q(length(W))$. Thus, F is the union of its restrictions to inputs of length n, $F = \bigcup_{n\geq 0} F_n$, where $F_n : \{0,1\}^n \to \{0,1\}^{Q(n)}$.*
2. *There is a family of circuits $(\alpha_n)_{n\geq 0}$, where α_n is a circuit computing F_n, with n input gates, $Q(n)$ outputs, and consisting of NOT gates, unbounded fan-in AND and OR gates.*
3. *For every $n \geq 0$, $size(\alpha_n) \leq P(n)$ for some polynomial P (the size is the number of gates), and $depth(\alpha_n) = O(\log^k n)$.*
4. *The family α_n is "uniform", as described below.*

Following Cook (see [Coo85], Proposition 4.7), we impose as uniformity condition the *DLOGSPACE-DCL* uniformity, reviewed below. Barrington, Immerman and Straubing in [BIS90] give a weaker uniformity condition called *FO-DCL*-uniformity which is equivalent to the *DLOGSPACE-DCL* uniformity for the classes AC^k, $k \geq 1$, and which provide a more satisfactory characterization for AC^0. In this paper, only Proposition 6.5 deals with the class AC^0 and it remains true for the more restrictive *FO-DCL*-uniformity condition in [BIS90].

For completeness, we briefly review the definition of *DLOGSPACE-DCL* uniformity [Coo85]. Let $\alpha_n, n \geq 0$ be a family of circuits. The **direct connection language** *DCL* for this family is the set of quadruples (n, g, g', t), where g, g' are gate numbers in α_n (i.e. $1 \leq g, g' \leq P(n)$), such that the output of g is one of the inputs of g' in α_n, and t indicates what kind of gate g' is in α_n, i.e. $t \in \{NOT, AND, OR, y_1, \ldots, y_{Q(n)}\}$. When $t = y_i$, then g' is the output bit i. We use the convention that the input gates corresponding to $x_1, \ldots, x_n$ are identified by assigning them the numbers $1, \ldots, n$. We say that the family of circuits α_n is *DLOGSPACE-DCL* **uniform**, iff the *DCL* can be accepted by some $O(\log n)$ space deterministic Turing Machine T.

Definition 4.2 $NC \stackrel{\text{def}}{=} \bigcup_{k\geq 0} AC^k$.

The results in Stockmeyer and Vishkin [SV84] imply that *NC* coincides with the class of functions computable by a *CRCW PRAM* (Concurrent Read Concurrent Write Parallel Random Access Machine) in poly-logarithmic time using polynomially many processors.

5 Computing Complex Objects Queries

In order to give precise definitions for query complexity classes, we must specify an encoding of complex objects into strings that can be given as input to a computational model such as PRAM's or families of circuits.

Our encoding of complex objects with strings over some fixed alphabet is related to that in [GV91b]. We start with an encoding of the base type $\mathbb{D}$ into natural numbers, i.e. we assume some bijection $\Phi : \mathbb{D} \to \mathbb{N}$ to be given. When dealing with ordered databases, we require this encoding to preserve the order relation $\leq$ on $\mathbb{D}$. Next, we encode complex objects using the eight symbols from the alphabet $A = \{0, 1, \{, \}, \langle, \rangle, \mathit{comma}, \mathit{blank}\}$, as follows: elements from $\mathbb{D}$ are encoded in binary, *true* and *false* are encoded by 1 and 0 respectively, a k-tuple is encoded by $\langle X_1, \ldots, X_k \rangle$, and a set by $\{X_1, \ldots, X_n\}$. No duplicates are allowed in the encoding of a set. However, blanks may be scattered arbitrarily inside some encoding, but not inside the binary numbers. Since the encoding of some complex object x is not unique, we define an encoding relation $x \sim X$ to denote the fact that X is a valid encoding of x. We view encodings as strings in $\{0, 1\}^*$, by further encoding each of the eight symbols in A by a string of length 3 in $\{0, 1\}^*$.

Example 5.1 *Let* $x = \{\{a, c\}, \{c\}, \{d\}, \{\}\}$. *Assuming that* $a, c, d \in \mathbb{D}$ *are encoded (by* Φ*) as* $0, 2$, *and 3 respectively, then* x *can be encoded by the string*

$$X = \text{“}\{\{0 \textit{ comma } 10\} \textit{ comma } \{10\} \textit{ comma } \{11\} \textit{ comma } \{\}\}\text{''}$$

but also by, say:

$$X' = \text{“}\textit{blank}\{\textit{blank}\{0 \textit{ comma blank}10\} \textit{ comma } \{10\} \textit{ comma } \{11\} \textit{ comma blank}\{\}\}\text{''}$$

When viewed as strings in $\{0, 1\}^*$, X *has length* $3 \times 21 = 63$, *while* X' *has length* $3 \times 25 = 75$.

Removing duplicates in sets is essential in the presence of recursors or iterators; else the size of some representation could grow beyond any polynomial. Duplicates can be removed in AC^0, by replacing them with blanks, and blanks can be removed (more precisely: moved at the end) in AC^1. So, within AC^0 we have the alternative choice of encoding with possible duplicates and no blanks, because there are no iterations allowed within AC^0. Within AC^k, $k \geq 1$, we could ask both for blanks and duplicate elimination. Our choice of encoding without duplicates, but to allow blanks, works across all AC^k, for $k \geq 0$.

Note that this encoding is different from that considered by Immerman in [Imm89], who only deals with flat relations. Under that encoding, a relation of type $\{\mathbb{D}^k\}$ is represented by a string of bits of length n^k, where n is the size of the *active domain*, see Appendix D. This encoding does not extend to complex objects, because it would require exponential space to encode objects of higher types, even when their size is only polynomial in the size of the active domain. For flat relations, we show in Appendix D that we can translate between the two encodings in AC^0, see Proposition D.1.

Adapting the definition in [CH80], we define a **database query** of type $t_1 \to t_2$ to be a function $f : t_1 \to t_2$ which is *generic* (see Definition A.1 for a review). We say that a query $f : t_1 \to t_2$ is **computed** by the function $F : \{0, 1\}^* \to \{0, 1\}^*$ iff $\forall x \in t_1$, $\forall X \in \{0, 1\}^*$, $x \sim X \Longrightarrow f(x) \sim F(X)$.

The fact that we only deal with *generic* queries gives us more liberty in the encoding of complex objects, without using the bijection Φ. Suppose the query f is computed by F. To compute $f(x)$ for some input $x \in t_1$, we may proceed as follows. Let $d_0, d_1, \ldots, d_{m-1}$ be the **active domain** of x, i.e. all values of type $\mathbb{D}$ mentioned in x. In the case of ordered databases, assume $d_0 < d_1 < \ldots < d_{m-1}$. Obviously $\Phi(d_0), \ldots, \Phi(d_{m-1})$ can be any numbers, but we will choose to encode $d_0, \ldots, d_{m-1}$ with the numbers $0, 1, \ldots, m-1$; we call this **the minimal encoding**, to distinguish it from the **standard encoding** using Φ. Let X be the resulting minimal encoding of x. Apply F on X to get $Y = F(X)$, and finally decode Y under the minimal encoding, to get y. Now we prove that y is the correct result, i.e. $y = f(x)$. Indeed, let $\mathbb{D}' = \{d'_0, d'_1, \ldots, d'_{m-1}\}$, where $d'_0, \ldots, d'_{m-1}$ are such that $\Phi(d'_i) = i$, for $i = 0, m-1$. Then the function $\varphi : \mathbb{D}' \to \mathbb{D}$, $\varphi(d'_i) \stackrel{\text{def}}{=} d_i$ for

$i = 0, m-1$ is injective and, in the case of ordered databases, also order-preserving. Let $\varphi_{t_1} : t'_1 \to t_1$ be its extension to t_1, where t'_1 is obtained from t_1 by replacing every occurrence of $\mathbb{D}$ with $\mathbb{D}'$ (see Appendix A). Let $x' = \varphi_{t_1}^{-1}(x)$. Obviously X is the standard encoding of x' and, since F computes f, we have that $Y = F(X)$ is the standard encoding of $y' = f(x')$. A moment of thought will convince the reader that decoding Y under the minimal encoding yields $\varphi_{t_2}(y')$, i.e. $y = \varphi_{t_2}(y')$. Now we use the fact that f is generic to argue $\varphi_{t_2}(y') = \varphi_{t_2}(f(x')) = f(\varphi_{t_1}(x')) = f(x)$. Hence, when we decode Y according to the minimal encoding, we obtain $f(x)$.

Definition 5.2 *We say that a query f is in NC iff there is some function $F : \{0,1\}^* \to \{0,1\}^*$ computing f which is in NC. We denote by CQ-NC the class of queries which are in NC, and by Q-NC the class of queries over types of set height ≤ 1 which are in NC. Similarly, for some $k \geq 0$, we define queries in AC^k, and the classes CQ-AC^k and Q-AC^k.*

In short, *Q-NC* is the class of *NC* queries over flat relations, and *CQ-NC* is that of *NC* queries over complex objects.

6 Main Results

Our languages capture the classes *Q-NC* and *CQ-NC* only on ordered databases. The presence of order means for us the addition of an external function $\leq : \mathbb{D} \times \mathbb{D} \to \mathbb{B}$, which is understood always to denote a linear order on $\mathbb{D}$. This immediately gives us an order relation at all types, that is, for every type t there is an expression in $\mathcal{NRA}(\leq)$, $\leq_t : t \times t \to \mathbb{B}$, whose meaning is a total order on t (e.g. see [LW94a]).

Theorem 6.1 $\mathcal{NRA}^1(dcr, \leq) =$ *Q-NC. More precisely* $\mathcal{NRA}^1(dcr^{(k)}, \leq) =$ *Q-AC^k for every $k \geq 1$.*

Theorem 6.2 $\mathcal{NRA}(bdcr, \leq) =$ *CQ-NC. More precisely* $\mathcal{NRA}(bdcr^{(k)}, \leq) =$ *CQ-AC^k for every $k \geq 1$.*

These languages are purely for flat relations, respectively complex objects. But many external functions of practical interest such as the usual arithmetical operations $(+, *, -, /, etc)$, and the usual aggregate functions (cardinality, sum, average, etc.) are also in *NC*. Can they be added in? The answer is yes for *bdcr* but no for *dcr*:

Proposition 6.3 *Let Σ be an extension consisting of possible additional base types and a set of functions computable in NC. Then $\mathcal{NRA}(\Sigma, bdcr) \subseteq NC$. However, $\mathcal{NRA}^1(\mathbb{N}, +, dcr)$ can express exponential space queries.*

Of course, in general $\mathcal{NRA}(\Sigma, bdcr) \not\subseteq$ *CQ-NC*, because the external functions in Σ may not necessarily be generic. We prove the inclusion $\mathcal{NRA}(\Sigma, bdcr) \subseteq NC$ in Section 9. The following example shows that $\mathcal{NRA}^1(\mathbb{N}, +, dcr)$ can express exponential space queries.

Example 6.4 *Consider the subset of $\{\mathbb{N}\}$ containing all sets of the form $\{0, 1, 2, \ldots, k\}$; we view such a set as the encoding of the integer k. Then we can define the functions $plus, times, exp : \{\mathbb{N}\} \times \{\mathbb{N}\} \to \{\mathbb{N}\}$,* *with the meaning:* $x = \{0, 1, 2, \ldots, m\}, y = \{0, 1, 2, \ldots, n\} \Rightarrow plus\langle x, y\rangle = \{0, 1, 2, \ldots, m+n\}, times\langle x, y\rangle = \{0, 1, 2, \ldots, m * n\}, exp\langle x, y\rangle = \{0, 1, 2, \ldots, n^m\}$, *in* $\mathcal{NRA}^1(\mathbb{N}, +, dcr)$. *Namely we define plus directly, while for times and exp we hint on their definition with dcr:*

$$
\begin{array}{rcll}
plus\langle x, y\rangle & \stackrel{\text{def}}{=} & ext(\lambda\langle u, v\rangle.\{u+v\})(x \times y) & \\
times\langle x \cup x', y\rangle & \stackrel{\text{def}}{=} & plus\langle times\langle x, y\rangle, times\langle x', y\rangle\rangle & when\ x \cap x' = \emptyset \\
exp\langle x \cup x', y\rangle & \stackrel{\text{def}}{=} & times\langle exp\langle x, y\rangle, exp\langle x', y\rangle\rangle & when\ x \cap x' = \emptyset
\end{array}
$$

In particular Example 6.4 shows that *dcr* is strictly more powerful than *bdcr* over flat types, in the presence of certain external functions, because when all external functions are in NC (and this is the case of + above), then with *bdcr* we can express only NC queries, while *exp* is obviously not in NC. This shows that, in general, Proposition 3.5 fails in the presence of external functions.

Immerman in [Imm89] and Barrington, Immerman and Straubing in [BIS90] prove that FO is included in *FO-DCL*-uniform AC^0, and that FO together with order and BIT relation has the same expressive power as AC^0. Here, we prove that $\mathcal{NRA}$ is included in AC^0, thus extending half of their result to complex objects.

Proposition 6.5 *Under the encoding of complex objects described in Section 5, all queries in* $\mathcal{NRA}(\leq)$, *are in FO-DCL-uniform* AC^0 *(see [BIS90])*

We state two more results which help us put the main theorems in perspective. Their proofs are omitted from this paper.

Conservative extension One may wonder in what sense Theorem 6.1 is a "particular case" of Theorem 6.2. Actually, even though the proof of Theorem 6.1 is quite similar to that of Theorem 6.2, and we do present them "together" in Sections 9 and 10, Theorem 6.1 in fact follows from Theorem 6.2, Proposition 3.5 and the conservative extension result presented below.

Paredaens and Van Gucht in [PG92], and Wong in [Won93] prove that $\mathcal{NRA}$ is a conservative extension of $\mathcal{NRA}^1$. Suciu in [Suc94] proves that $\mathcal{NRA}(bfix)$ is a conservative extension of $\mathcal{NRA}^1(fix)$, where fix is the usual inflationary fixpoint, and $bfix$ is a bounded version of fix. Using the techniques in [Suc94], we can prove the following:

Proposition 6.6 *Let* Σ *be a set of external functions which have set heights* ≤ 1. *Then,* $\mathcal{NRA}(\Sigma, bdcr, \leq)$ *is a conservative extension of* $\mathcal{NRA}^1(\Sigma, bdcr, \leq)$.

Note that for the case when $\Sigma = \emptyset$, we can turn the tables and Proposition 6.6 follows directly from the main theorems. For the case when $\Sigma \neq \emptyset$, this proposition requires a separate proof, and we are able to do it only in the presence of order. However, we conjecture that $\mathcal{NRA}(bdcr)$ is a conservative extension of $\mathcal{NRA}^1(dcr)$.

PTIME vs. NC Immerman, Patnaik and Stemple [IPS91] show that $PTIME$ is captured by a language built around *set-reduce* (see Section 3). Extending their result also to complex objects we have:

Proposition 6.7 *$\mathcal{NRA}^1(sri^{(1)}, \leq) = Q\text{-}PTIME$ [IPS91] and $\mathcal{NRA}(bsri^{(1)}, \leq) = CQ\text{-}PTIME$.*

Here *Q-PTIME* and *CQ-PTIME* are the set of queries over flat relations, and over complex objects respectively, computable by some function in *PTIME*.

Thus, by the main theorems and this proposition, the difference between *PTIME* and *NC* computable queries over ordered databases can be characterized by the difference between two kinds of recursion on sets. It is interesting to note that only one level of recursion nesting suffices for *sri* and *PTIME*, as opposed to *dcr* and *NC*. For the latter, the collapse of the *dcr* hierarchy is equivalent to the collapse of the AC^k, $k \geq 1$ hierarchy, and hence is still open.

The rest of the paper is devoted to the proof of the main result, Theorems 6.1 and 6.2. Basically, we do this in three steps:

1. We show that, over ordered databases, *dcr* is equivalent to another form of recursion, called *set-divide*, whose definition depends essentially on the order relation. Moreover, the equivalence preserves the nesting depth.
2. Next we show that *set-divide* is equivalent, on ordered databases, to a form of iteration, which we call *log_loop*. Again, the nesting depth is preserved under the equivalence.
3. Finally we show that $\mathcal{NRA}^1(log\_loop, \leq) = Q\text{-}NC$.

The new form of recursion *set-divide* is of interest in itself as an order dependent variant of *dcr*. The other forms of recursion, *bdcr*, *sri*, and *bsri* have order-dependent cousins themselves, which we discuss too. Next, the *log_loop* construct is also of interest, because its definition is order independent (unlike *set-divide*), it requires no side-conditions (as *dcr* does), and, hence, forms the core of a decidable query language which, over ordered databases expresses exactly *Q-NC*. Of course, there is a bounded version too, which does the job for complex objects.

7 Recursion and Order

One way of interpreting the roles of conditions like associativity, commutativity, etc., in the definition of *dcr* and *sri*, is as simple sufficient conditions for order independence. As we saw in Subsection 3.3, checking these conditions is undecidable in general. On *ordered* structures on the other hand, one can define forms of recursion on sets that do not require conditions. For instance, Immerman, Patnaik, and Stemple [IPS91] consider under the name *set-reduce* a form of recursion on sets which resembles somewhat *sri*. Set-reduce does not require conditions such as left-commutativity, etc. Instead, its definition relies on the existence of a linear ordering on the elements of the sets to which it is applied. We prove that, in the presence of order, this form of recursion has the same expressive power as *sri*. We also formulate a similar order-dependent form of recursion that corresponds to *dcr* in the presence of order. Finally, an interesting relationship holds between *sru* and *dcr* in the presence of order.

Set-reduce [IPS91] Let $<: t \times t \to \mathbb{B}$ be a linear ordering on t, and for $e : t, j : t \times t' \to t'$, let *set-reduce(e,j)* : $\{t\} \to t'$ be defined by:

$$set\text{-}reduce(e, j)(\emptyset) \stackrel{\text{def}}{=} e$$

$$set\text{-}reduce(e,j)(\{x_1,\ldots,x_n\}) \stackrel{\text{def}}{=} j\langle x_1, set\text{-}reduce(\{x_2,\ldots,x_n\})\rangle \text{ when } x_1 < x_2 < \ldots < x_n$$

No conditions are imposed on j.

Set-divide Similarly, one can conceive a form of divide and conquer recursion that relies on the ordering, which allows us to define some function by $\varphi(\{x_1,\ldots,x_n\}) \stackrel{\text{def}}{=} u(\varphi(\{x_1,\ldots,x_{\lfloor\frac{n}{2}\rfloor}\}), \varphi(\{x_{\lfloor\frac{n}{2}\rfloor+1},\ldots,x_n\}))$ (no conditions are imposed on u). That is, we make the arbitrary choice of dividing a set into two almost equal halves at each iteration step[8]. Again, we can prove that this form of recursion has the same expressive power as dcr.

Set-divide is defined formally as follows. Let $<: t \times t \to \mathbb{B}$ be a linear ordering on t, and for $e : t', f : t \to t', v : t' \times t' \to t'$ define $set\text{-}divide(e,f,v) : \{t\} \to t'$ by:

$$\begin{aligned}
set\text{-}divide(e,f,v)(\emptyset) &\stackrel{\text{def}}{=} e \\
set\text{-}divide(e,f,v)(\{a\}) &\stackrel{\text{def}}{=} f(a) \\
set\text{-}divide(e,f,v)(\{x_1,\ldots,x_n\}) &\stackrel{\text{def}}{=} v(set\text{-}divide(e,f,v)(\{x_1,\ldots,x_{\lfloor\frac{n}{2}\rfloor}\}), set\text{-}divide(e,f,v)(\{x_{\lfloor\frac{n}{2}\rfloor+1},\ldots,x_n\})) \\
&\quad\ \text{when } x_1 < x_2 < \ldots < x_n
\end{aligned}$$

No conditions are imposed on v. Then:

Proposition 7.1 *Over ordered flat relations, set-divide is equivalent to dcr and set-reduce is equivalent to sri*[9]*:*

$$\begin{aligned}
\mathcal{NRA}^1(set\text{-}divide, \leq) &= \mathcal{NRA}^1(dcr, \leq) \\
\mathcal{NRA}^1(set\text{-}reduce, \leq) &= \mathcal{NRA}^1(sri, \leq)
\end{aligned}$$

Similarly, over complex objects we have:

$$\begin{aligned}
\mathcal{NRA}(bset\text{-}divide, \leq) &= \mathcal{NRA}(bdcr, \leq) \\
\mathcal{NRA}(bset\text{-}reduce, \leq) &= \mathcal{NRA}(bsri, \leq)
\end{aligned}$$

Moreover, these equivalences preserve the iteration depth, e.g. $\mathcal{NRA}^1(set\text{-}divide^{(k)}, \leq) = \mathcal{NRA}^1(dcr^{(k)}, \leq)$, *for all* $k \geq 0$*, etc.*

Main technique Proving one inclusion is easy: $\mathcal{NRA}^1(dcr, \leq) \subseteq \mathcal{NRA}^1(set\text{-}divide, \leq)$, because whenever $dcr(e,f,u)$ is a correct expression, it is equivalent to $set\text{-}divide(e,f,u)$. The other direction is more involved: given a $set\text{-}divide(e,f,v)$ expression, in general v is not associative and does not have identity e, hence the expression $dcr(e,f,v)$ is not well formed: we need a more involved dcr expression. Let $\psi = set\text{-}divide(e,f,v)$, $\psi : \{t\} \to t'$. The idea is to compute $\psi(x)$, where $x = \{x_1,\ldots,x_n\}$, $x_1 < x_2 < \ldots < x_n$, by computing ψ iteratively on larger and larger subsets of x and "memorizing" all results. For that, we define, for every number k, $0 \leq k \leq n$, a complex object $\tilde{k}$ which "encodes' all applications of ψ to "intervals" of length $\leq k$ in x, i.e. to sets $s \subseteq \{x_1,\ldots,x_n\}$ of the form $s = \{x_p, x_{p+1},\ldots,x_q\}$, of cardinality $\leq k$ (i.e. $q-p+1 \leq k$). There is a

[8]Another ad-hoc way of defining recursion, related to *set-divide*, but on vectors instead of sets, can be found in the equational parallel language EL^* [RW93]

[9]The equivalence holds for complex objects too, but we are only interested in bounded iterations over complex objects here.

single interval of length 0, namely $\emptyset$, there are n intervals of length 1 ($\{x_1\}, \ldots, \{x_n\}$), $n-1$ intervals of length 2 ($\{x_1, x_2\}, \{x_2, x_3\}, \ldots, \{x_{n-1}, x_n\}$), etc. Thus the complex object, say, $\tilde{2}$, will contain enough information to allow us to extract from it all values $\psi(\emptyset), \psi(\{x_1\}), \ldots, \psi(\{x_n\}), \psi(\{x_1, x_2\}), \ldots, \psi(\{x_{n-1}, x_n\})$. In general $\tilde{k}$ encodes the result of applying ψ on $1 + n + (n-1) + \ldots + (n-k+1) = 1 + \frac{k(n-k+1)}{2}$ intervals. So, in order to compute $\psi(x)$, we compute $\tilde{k}$ for larger and larger k's, until we get $\tilde{n}$, from which we extract $\psi(x)$. We need to be a bit careful however, because to show, say $\mathcal{NRA}^1(\textit{set-divide}, \leq) \subseteq \mathcal{NRA}^1(dcr, \leq)$, we don't want $\tilde{k}$ to have a larger set height than ψ. Typically t' is a PS-type and, for the sake of clarity, we will assume that t' is a product of two set types, that is $t' = \{t'_1\} \times \{t'_2\}$. Then for every number k, $0 \leq k \leq n$, we define the *encoding* of k w.r.t. ψ to be $\tilde{k} \in \{t \times t\} \times \{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\}$:

$$\begin{aligned}\tilde{k} \stackrel{\text{def}}{=} \langle \quad & \{\langle x_p, x_q\rangle \mid 1 \leq p, q \leq n, 0 \leq q-p+1 \leq k\}, \\ & \{\langle\langle x_p, x_q\rangle, a_1\rangle \mid 1 \leq p, q \leq n, 0 \leq q-p+1 \leq k, a_1 \in \pi_1(\psi(\{x_p, x_{p+1}, \ldots, x_q\}))\}, \\ & \{\langle\langle x_p, x_q\rangle, a_2\rangle \mid 1 \leq p, q \leq n, 0 \leq q-p+1 \leq k, a_2 \in \pi_2(\psi(\{x_p, x_{p+1}, \ldots, x_q\}))\}\rangle\end{aligned}$$

We adopt the convention $\tilde{k} \stackrel{\text{def}}{=} \tilde{n}$, whenever $k > n$.

Thus $\tilde{k}$ contains the following informations: (1) the set of all pairs $\langle x_p, x_q\rangle$ for which $\{x_p, x_{p+1}, \ldots, x_q\}$ has between 0 and k elements, and (2) for each such pair $\langle x_p, x_q\rangle$, all values of $\psi(\{x_p, x_{p+1}, \ldots, x_q\})$, tagged with the pair $\langle x_p, x_q\rangle$. Note that the set height of $\tilde{k}$ is no larger than the maximum of the set heights of $\{t\}$ or t'. In particular, when ψ takes as input a flat relation and returns two flat relations, then $\tilde{k}$ consists of three flat relations.

Given an encoding $\tilde{k}$ and $x_p, x_q \in x$, we can extract $\psi(\{x_p, \ldots, x_q\})$. Namely we define:

$$\textit{extract}\langle\tilde{k}, \langle x_p, x_q\rangle\rangle \stackrel{\text{def}}{=} \langle\{a_1 \mid \langle\langle x_p, x_q\rangle, a_1\rangle \in \pi_2(\tilde{k})\}, \{a_2 \mid \langle\langle x_p, x_q\rangle, a_2\rangle \in \pi_3(\tilde{k})\}\rangle$$

Obviously *extract* can be expressed in $\mathcal{NRA}$ (or in $\mathcal{NRA}^1$, when the type t' has set height ≤ 1), and $\textit{extract}\langle\langle x_p, x_q\rangle, \tilde{k}\rangle = \psi(\{x_p, x_{p+1}, \ldots, x_q\})$, for $0 \leq q-p+1 \leq k$.

We will use this encoding to compute ψ with *dcr*, or with *esr* (and, hence, with *sri*). To compute ψ with *dcr*, assume that the following are expressible in $\mathcal{NRA}$: $\tilde{0}$, $\tilde{1}$, and the function $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k+k'}$. Then we can express the function $\gamma : \{t\} \to \{t \times t\} \times \{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\}$ defined by

$$\gamma(s) = \begin{cases} \tilde{k} & \text{when } \textit{card}(s) = k \leq n \\ \tilde{n} & \text{when } \textit{card}(s) > n \end{cases}$$

as $\gamma = dcr(\tilde{0}, \lambda y.\tilde{1}, \lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k+k'})$. Obviously the function $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k+k'}$ is associative, commutative, and has identity $\tilde{0}$, and the *dcr*-expression is well defined. Finally, to compute $\psi(x)$, we take

$$\psi(x) = \textit{extract}\langle\psi(x), \langle x_1, x_n\rangle\rangle$$

Proof. (Of Proposition 7.1) As argued earlier, the inclusions $\mathcal{NRA}^1(dcr, \leq) \subseteq \mathcal{NRA}^1(\textit{set-divide}, \leq)$, $\mathcal{NRA}^1(sri, \leq) \subseteq \mathcal{NRA}^1(\textit{set-reduce}, \leq)$, $\mathcal{NRA}(bdcr, \leq) \subseteq \mathcal{NRA}(\textit{bset-divide}, \leq)$, and $\mathcal{NRA}(bsri, \leq) \subseteq \mathcal{NRA}(\textit{bset-reduce}, \leq)$ follow immediately, so we show only the other four inclusions. We start by showing $\mathcal{NRA}(\textit{bset-reduce}, \leq) \subseteq \mathcal{NRA}(bsri, \leq)$, let $\psi = \textit{bset-reduce}(e, j, b) : \{t\} \to t'$. In order to express

$\psi(x)$ with *sri*, where $x = \{x_1, \ldots, x_n\}$, with $x_1 < x_2 < \ldots < x_n$, we will use the encoding described earlier. Since t' is a PS-type, we will assume, for sake of clarity, that it is the product of two set types, $t' = \{t'_1\} \times \{t'_2\}$. Then we associate to each number k, its *encoding* $\tilde{k} \in \{t \times t\} \times \{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\}$, as before. It is easy to check that, given the fact that $\psi = \mathit{bset\text{-}reduce}(e, j, b)$, then $\tilde{0}$, and the function $\lambda\tilde{k}.\widetilde{k+1}$ are expressible in $\mathcal{NRA}(\mathit{bsri}, \leq)$. For the latter, suppose we are given some $\tilde{k}$. To compute $\widetilde{k+1}$, consider all pairs $\langle x_p, x_q\rangle \in \pi_1(\tilde{k})$ with $p > 1$. Compute x_{p-1} (using the order relation on $x = \{x_1, \ldots, x_n\}$), and compute $\psi(\{x_p, \ldots, x_q\}) = \mathit{extract}(\tilde{k}, \langle x_p, x_q\rangle)$. To compute $\psi(\{x_{p-1}, x_p, \ldots, x_q\})$, observe that $\psi(\{x_{p-1}, x_p, \ldots, x_q\}) = j(x_{p-1}, \psi(\{x_p, \ldots, x_q\}))$. Finally this allows us to assemble together the value $\widetilde{k+1}$.

Then we define in $\mathcal{NRA}(\mathit{bsri}, \leq)$ the function $\gamma : \{t\} \to \{t \times t\} \times \{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\}$, $\gamma = \mathit{esr}(\tilde{0}, \lambda\langle y, \tilde{k}\rangle.\widetilde{k+1})$, whose meaning is:

$$\gamma(s) \stackrel{\text{def}}{=} \begin{cases} \tilde{k} & \text{when } \mathit{card}(s) = k \leq n \\ \tilde{n} & \text{when } \mathit{card}(s) > n \end{cases}$$

Next we convert the definition of γ from *esr* into *besr*, by noting that $\forall k,\ 0 \leq k \leq n,\ \tilde{k} \subseteq b'$, where $b' = (x \times x) \times ((x \times x) \times \Pi_1(b)) \times ((x \times x) \times \Pi_2(b))$. Hence $\gamma = \mathit{besr}(\tilde{0}, \lambda\langle y, \tilde{k}\rangle.\widetilde{k+1}, b')$. Finally we convert it into a definition with *bsri*, using Proposition 3.1, and observe that $\psi(x) = \mathit{extract}\langle\langle x_1, x_n\rangle, \gamma(x)\rangle$.

To check $\mathcal{NRA}^1(\mathit{set\text{-}reduce}, \leq) \subseteq \mathcal{NRA}^1(\mathit{sri}, \leq)$, we argue as follows. First, by extending Proposition 3.5 to *set-reduce*, we obtain $\mathcal{NRA}^1(\mathit{set\text{-}reduce}, \leq) = \mathcal{NRA}^1(\mathit{bset\text{-}reduce}, \leq)$. Next, we observe that in the translation given above of $\mathit{bset\text{-}reduce} : \{t\} \to t'$ into *bsri*, the resulting expression uses only types of set height $\leq \max(\mathit{height}(\{t\}), \mathit{height}(t'), 1))$, which proves $\mathcal{NRA}^1(\mathit{bset\text{-}reduce}, \leq) \subseteq \mathcal{NRA}^1(\mathit{bsri}, \leq)$. Finally we use Proposition 3.5 to argue that $\mathcal{NRA}^1(\mathit{bsri}, \leq) = \mathcal{NRA}^1(\mathit{sri}, \leq)$.

We prove that $\mathcal{NRA}(\mathit{bset\text{-}divide}, \leq) \subseteq \mathcal{NRA}(\mathit{bdcr}, \leq)$ using the same idea, with minor additional complications. Here, given that $\psi = \mathit{bset\text{-}divide}(e, f, v, b)$, the following are expressible in $\mathcal{NRA}(\mathit{bdcr}, \leq)$: $\tilde{0}$, $\tilde{1}$, and, the function $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k+k'}$. We will explain how to express the latter. Recall that $\tilde{k}$ encodes all values of $\psi(s)$, for s an interval of $x_1 < x_2 < \ldots < x_n$, of cardinality $\leq k$. To compute $\widetilde{k+k'}$, we need to apply ψ to all intervals intervals $s'' \subseteq \{x_1, \ldots, x_n\}$ of cardinality $k'' \leq k + k'$ (there are $n - (k + k') + 1$ such intervals). To do this, split $s'' = \{x_p, x_{p+1}, \ldots, x_q\}$ into two halves of cardinalities $\lfloor \frac{k''}{2} \rfloor$ and $\lceil \frac{k''}{2} \rceil$ respectively, $s'' = s_1 \cup s_2$, with $s_1 = \{x_p, x_{p+1}, \ldots, x_r\}$, $s_2 = \{x_{r+1}, x_{r+2}, \ldots, x_q\}$. This can be done using Lemma 7.2 below, given that *bdcr* can express transitive closure. We will not compute transitive closure for every pair $\langle x_p, x_q\rangle$ however, because this would increase the nesting depth of the *bdcr* expression (it would be 1 higher than the original *bset-divide* expression): instead, we precompute a relation containing all triples $\langle x_p, x_q, x_r\rangle$, $0 \leq q - p + 1, r = \lfloor \frac{p+q}{2} \rfloor$, using a single level of *bdcr*, then look up this table every time we need to split some set s''. Now we argue that both s_1 and s_2 have cardinalities which are less than or equal to the largest of k, k': if not, then $\mathit{card}(s_1) \geq k$ (because $\mathit{card}(s_1) \geq \mathit{card}(s_2) - 1$), $\mathit{card}(s_2) > k'$, hence $\mathit{card}(s'') = \mathit{card}(s_1) + \mathit{card}(s_2) > k + k'$, contradiction. Therefore, assuming $k \geq k'$, it suffices to "look up" the values $\psi(s_1)$ and $\psi(s_2)$ in $\tilde{k}$, and to compute $\psi(s'') = v(\psi(s_1), \psi(s_2))$.

Next, as in the proof of Proposition 7.1, we will consider the function $\gamma \stackrel{\text{def}}{=} \mathit{dcr}(\tilde{0}, \lambda y.\tilde{1}, \lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k+k'})$, with the meaning:

$$\gamma(s) = \begin{cases} \tilde{k} & \text{when } \mathit{card}(s) = k \leq n \\ \tilde{n} & \text{when } \mathit{card}(s) > n \end{cases}$$

The definition of γ is obviously correct, since $\lambda\langle\tilde{k},\tilde{k}'\rangle.\widetilde{k+k'})$ is associative, commutative, and has identity $\tilde{0}$. As in the proof of Proposition 7.1, we observe that $\gamma(x) = \mathit{bdcr}(\tilde{0},\tilde{1},\lambda\langle\tilde{k},\tilde{k}'\rangle.\widetilde{k+k'},b')(x)$, where $b' = (x\times x)\times(((x\times x)\times\Pi_1(b))\times((x\times x)\times\Pi_2(b)))$. Finally $\psi(x) = \mathit{extract}\langle\psi(x),\langle x_1,x_n\rangle\rangle$.

To prove $\mathcal{NRA}^1(\mathit{set\text{-}divide},\leq)\subseteq\mathcal{NRA}^1(\mathit{dcr},\leq)$, we proceed as in the proof of Proposition 7.1. Namely we first extend Proposition 3.5 to $\mathcal{NRA}^1(\mathit{set\text{-}divide},\leq) = \mathcal{NRA}^1(\mathit{bset\text{-}divide},\leq)$, then observe that the above construction also proves that $\mathcal{NRA}^1(\mathit{bset\text{-}divide},\leq)\subseteq\mathcal{NRA}^1(\mathit{bdcr},\leq)$, since the auxiliary types never exceed in set height the heights of the input and output types. Finally we use Proposition 3.5 to argue that $\mathcal{NRA}^1(\mathit{bdcr},\leq) = \mathcal{NRA}^1(\mathit{dcr},\leq)$.

$\square$

In the proof above we needed the following Lemma:

Lemma 7.2 ([Imm87b]) *Let $tc_t : \{t\times t\}\to\{t\times t\}$ be the function computing the transitive closure of binary relations of elements of type t, and let $\mathcal{NRA}(tc)$ stand for $\mathcal{NRA}$ extended with tc_t for every type t. Then, the function eq-cardinality $: \{t\}\times\{t\}\to\mathbb{B}$, defined by eq-cardinality$\langle x,y\rangle$ = true iff $card(x) = card(y)$, is expressible in $\mathcal{NRA}(tc,\leq)$.*

Proof. Let $x = \{x_1,\ldots,x_m\}$, $y = \{y_1,\ldots,y_m\}$, and assume $x_1 < x_2 < \ldots < x_m$, $y_1 < y_2 < \ldots < y_n$. Let $r\in\{(t\times t)\times(t\times t)\}$ be $r = \{\langle\langle x_i,y_j\rangle,\langle x_{i+1},y_{j+1}\rangle\rangle \mid i = 1,m-1; j = 1,n-1\}$. Obviously r can be computed in $\mathcal{NRA}(\leq)$ from x and y. Compute the transitive closure of r, $q \stackrel{\text{def}}{=} tc_{t\times t}(r)$. Then $m = n$ iff $\langle\langle x_1,y_1\rangle,\langle x_m,y_n\rangle\rangle\in q$. $\square$

sru vs. dcr We end this section with a comment on the expressive power of *sru*. Although superficially related to *dcr*, *sru* seems to have less expressive power. In fact it remains open whether transitive closure can be expressed in $\mathcal{NRA}^1(\mathit{sru})$, or in $\mathcal{NRA}(\mathit{bsru})$, or even in $\mathcal{NRA}(\mathit{bsru},\leq)$. We do not venture a conjecture about the last language, but there is some reason to think that the following is true:

Conjecture 7.3 *Transitive closure cannot be expressed in $\mathcal{NRA}^1(\mathit{sru})$, nor in $\mathcal{NRA}(\mathit{bsru})$.*

Interestingly, with order and *tc*, *sru* becomes as powerful as *dcr*:

Proposition 7.4 $\mathcal{NRA}^1(\mathit{sru},\leq,tc) = \mathcal{NRA}^1(\mathit{dcr},\leq)$ *and* $\mathcal{NRA}(\mathit{bsru},\leq,tc) = \mathcal{NRA}(\mathit{bdcr},\leq)$

Proof. To show $\mathcal{NRA}(\mathit{bdcr},\leq)\subseteq\mathcal{NRA}(\mathit{bsru},\leq,tc)$ consider some expression $\psi = \mathit{bdcr}(e,f,u,b)$ in $\mathcal{NRA}(\mathit{bdcr},\leq)$. Without loss of generality we may assume that e,f,u,b are in $\mathcal{NRA}(\mathit{bsru},\leq)$.

To compute $\psi(\{x_1,\ldots,x_n\})$, $x_1 < x_2 < \ldots < x_n$, using *bsru*, we use the encodings of numbers $\tilde{k}$, $0\leq k\leq n$ mentioned earlier. Namely we define the function φ to be $\varphi(s) \stackrel{\text{def}}{=} \langle\tilde{k},s\rangle$, where $k = card(s)$. φ can be defined using *bsru*. Obviously $\varphi(\emptyset) \stackrel{\text{def}}{=} \langle\tilde{0},s\rangle$ and $\varphi(\{y\}) \stackrel{\text{def}}{=} \langle\tilde{1},\{y\}\rangle$. For $\varphi(s\cup s')$, let $k = card(s), k' = card(s')$ and $k'' = card(s\cup s')$. We have $\varphi(s\cup s') = \langle\widetilde{k''},s\cup s'\rangle$, so we have to argue that $\widetilde{k''}$ can be computed, given $\tilde{k}$, $\tilde{k}'$, and s,s'. First note that $\tilde{k}''\subseteq\widetilde{k+k'}$. So all we have to do is to select from $\widetilde{k+k'}$ all those values which are tagged with pairs $\langle x_p,x_q\rangle$ with $q-p+1\leq k''$. ($\lambda\langle\tilde{k},\tilde{k}'\rangle.\widetilde{k+k'}$ can be computed with *bsru* in the same way as with *bset-divide* in the proof of Proposition 7.1; in fact it is even simpler, since we don't need to test for equal cardinality.) The latter is equivalent to $card(\{x_p, \ldots, x_q\}) \leq card(s \cup s')$, which can be tested in $\mathcal{NRA}(\leq, tc)$, using Lemma 7.2. Finally we have $\psi(\{x_1, \ldots, x_n\}) = extract(\pi_1(\varphi(\{x_1, \ldots, x_n\})), \langle x_1, x_n\rangle)$. $\Box$

8 Iteration over Sets

Next we show that the order dependent forms of recursion of the previous section are equivalent, over ordered databases, with more simple *loops*. The **logarithmic** and the **bounded logarithmic iterator** are defined by:

$$\frac{f : t \to t}{log\_loop(f) : \{t'\} \times t \to t} \quad \frac{f : t \to t \quad b : t}{blog\_loop(f, b) : \{t'\} \times t \to t}$$

with the semantics:

$$log\_loop(f)\langle x, y\rangle \stackrel{\text{def}}{=} f^{(\lceil \log(card(x)+1)\rceil)}(y)$$

where $card(x)$ is the number of elements of x. Thus, *log_loop* iterates some function f a number of times equal to the number of bits necessary to represent the number $card(x)$. The bounded logarithmic iterator is define by:

$$blog\_loop(f, b)\langle x, y\rangle \stackrel{\text{def}}{=} log\_loop(f \cap b)\langle x, y \cap b\rangle$$

Similarly, we define the **iterator** and the **bounded iterator** *loop* and *bloop*, which iterates some function $card(x)$ times, instead of $\lceil \log(card(x) + 1)\rceil$ times:

$$\frac{f : t \to t}{loop(f) : \{t'\} \times t \to t} \quad \frac{f : t \to t \quad b : t}{bloop(f, b) : \{t'\} \times t \to t}$$

with the semantics:

$$\begin{aligned} loop(f)\langle x, y\rangle &\stackrel{\text{def}}{=} f^{card(x)}(y) \\ bloop(f, b)\langle x, y\rangle &\stackrel{\text{def}}{=} loop(f \cap b)\langle x, y \cap b\rangle \end{aligned}$$

We extend the definition of depth of recursion nesting to *depth of iteration nesting* for these construct, by defining $depth(log\_loop(f)(e)) \stackrel{\text{def}}{=} \max(1 + depth(f), depth(e))$, etc.

Both *log_loop* and *loop* are powerful enough to express *powerset*. Hence, we will only consider the unbounded versions in conjunction with flat relations, and use their bounded versions for complex objects.

Example 8.1 *log_loop can express transitive closure, $tc : \{t \times t\} \to \{t \times t\}$. Indeed, let $r \in \{t \times t\}$ be some relation. First compute $v = \Pi_1(r) \cup \Pi_2(r)$ (the set of all elements mentioned in r), then, repeat $\lceil \log(n + 1)\rceil$ times $r \leftarrow r \cup r \circ r$, where $n \stackrel{\text{def}}{=} card(v)$, and $\circ$ is relation composition. That is $tc(r) = log\_loop(\lambda r.r \cup r \circ r)\langle v, r\rangle$.*

Example 8.2 *Let* $n = card(x)$. *Then* $loop(f)$ *and* $log\_loop(f)$ *allow us to iterate* n *and* $\log n$ *times respectively. To iterate* n^2 *times, it suffices to loop over* $x \times x$*, which has* n^2 *elements:* $f^{(n^2)}(y) = loop(f)\langle x \times x, y\rangle$. *But in order to iterate* $\log^2 n$ *times, we use two levels of iterations:*

$$f^{(\lceil \log(n+1) \rceil^2)}(y) = log\_loop(\lambda z.log\_loop(f)\langle x, z\rangle)\langle x, y\rangle$$

Immerman defines $FO(t(n))$ in [Imm89] to be first order logic, with order and with a binary relation BIT, extended with those inductive definitions which close after $t(n)$ steps. The languages $\mathcal{NRA}^1(log\_loop, \leq, BIT)$ and $\mathcal{NRA}^1(loop, \leq, BIT)$ have essentially the same expressive power as $FO[\log^{O(1)} n]$ and $FO[n^{O(1)}]$ respectively. However, without order, these two are no longer equivalent: *loop* can express parity, while $FO[n^{O(1)}]$ (without order and BIT) is included in $FO + LFP$, and hence it cannot express parity. Similarly, we can argue that $FO[\log^{O(1)} n]$ without order is less powerful than $\mathcal{NRA}^1(log\_loop)$.

The key technical lemma in proving the main results states that *dcr* and *log_loop* have the same expressive power over ordered databases. Expressions with *dcr* can be translated into expressions with *log_loop* and conversely. Moreover, this translation preserves the depth of iteration nesting.

Proposition 8.3 *Over ordered flat relations, set-divide is equivalent to log_loop, and set-reduce is equivalent to loop:*

$$\begin{aligned} \mathcal{NRA}^1(log\_loop, \leq) &= \mathcal{NRA}^1(set\text{-}divide, \leq) \\ \mathcal{NRA}^1(loop, \leq) &= \mathcal{NRA}^1(set\text{-}reduce, \leq) \end{aligned}$$

Similarly, over complex objects we have:

$$\begin{aligned} \mathcal{NRA}(blog\_loop, \leq) &= \mathcal{NRA}(bset\text{-}divide, \leq) \\ \mathcal{NRA}(bloop, \leq) &= \mathcal{NRA}(bset\text{-}reduce, \leq) \end{aligned}$$

Moreover, these equivalences preserve the iteration depth, e.g $\mathcal{NRA}^1(log\_loop^{(k)}, \leq) = \mathcal{NRA}^1(set\text{-}divide^{(k)}, \leq)$ *for all* $k \geq 0$*, etc.*

Proof. We saw in Proposition 3.5 that $\mathcal{NRA}^1(bdcr) = \mathcal{NRA}^1(dcr)$. Similarly, one can show that $\mathcal{NRA}^1(blog\_loop) = \mathcal{NRA}^1(log\_loop)$ and $\mathcal{NRA}^1(bset\text{-}divide) = \mathcal{NRA}^1(set\text{-}divide)$. Hence, once we prove $\mathcal{NRA}(blog\_loop, \leq) = \mathcal{NRA}(bset\text{-}divide)$, the equivalence $\mathcal{NRA}^1(log\_loop, \leq) = \mathcal{NRA}^1(set\text{-}divide, \leq)$ follows. So we prove the first equivalence, and start by showing $\mathcal{NRA}(bset\text{-}divide, \leq) \subseteq \mathcal{NRA}(blog\_loop, \leq)$. Consider some function $\varphi = bset\text{-}divide(e, f, u, b)$, $\varphi : \{t\} \to t'$. Let $x = \{x_1, \ldots, x_n\} \in \{t\}$. We will use the encoding $\tilde{k}$ described in Section 7. Define g to be the function $\lambda \tilde{k}.\widetilde{k + k}$ (see the techniques used in the proof of Proposition 7.1). Then the sequence $\tilde{1}, g(\tilde{1}), g(g(\tilde{1})), g(g(g(\tilde{1}))), \ldots$ is $\tilde{1}, \tilde{2}, \tilde{4}, \tilde{8}, \ldots$, hence it suffices to iterate g $\lceil \log(n+1) \rceil$ times, and to apply it to $\tilde{1}$, to get: $log\_loop(g)\langle x, \tilde{1}\rangle = \tilde{n}$. Using the techniques from the proof of Proposition 7.1, given a bound b for $bset\text{-}divide(e, f, u, b)$, we can compute a bound b' for $\tilde{k}$, for all $k \leq n$, hence $log\_loop(g)\langle x, \tilde{1}\rangle$ is equivalent to $blog\_loop(g, b')\langle x, \tilde{1}\rangle$. Finally we obtain $\varphi(x) = extract\langle \tilde{n}, \langle x_1, x_n\rangle\rangle$.

The converse, $\mathcal{NRA}(blog\_loop, \leq) \subseteq \mathcal{NRA}(bset\text{-}divide, \leq)$, follows immediately from $blog\_loop(f, b)\langle x, y\rangle = bset\text{-}divide(y, \lambda z.f(y), \lambda\langle z, z'\rangle.f(z), b)(x)$.

Finally note that the translations preserve the nesting depth. The equivalence between *bloop* and *bset-reduce* is proved similarly. $\square$

It follows that the languages $\mathcal{NRA}^1(dcr)$ and $\mathcal{NRA}^1(log\_loop)$ coincide on ordered structures. However we do not know what their relationship is without order. Still Proposition 8.3 has an important consequence. Recall that the conditions for well-definedness of dcr are not r.e. hence the language $\mathcal{NRA}^1(dcr, \leq)$ is not r.e. But, by restricting it to the instances of dcr used in the simulation of *log_loop* we obtain an r.e., in fact decidable, sub-language $\mathcal{L}$ which has the same expressive power as the whole $\mathcal{NRA}^1(dcr, \leq)$.

9 Circuits

We will show here that $\mathcal{NRA}^1(log\_loop) \subseteq Q\text{-}NC$. First we establish some technical lemmas. For some string $W \in \{0,1\}^n$ of length n and numbers i, j, $0 \leq i \leq j \leq n-1$, we denote with $W[i:j]$ the substring of W containing all elements from positions i to j inclusive. Recall that we encode each symbol of the alphabet $A = \{0, 1, \{, \}, \langle, \rangle, comma, blank\}$ with a string of length 3. Thus if X is an encoding of the complex object x, then for every i for which $i \ mod \ 3 = 0$, $X[i:i+2]$ will be the binary encoding of some character in A.

Lemma 9.1 *For every $d \geq 0$ there exists a function $F^d = \bigcup_{n \geq 0} F_n^d$ in AC^0, $F_n^d : \{0,1\}^n \to \{0,1\}^{n^2}$, finding the "matching parentheses" of nesting depth $\leq d$.*

More precisely, for every string $X \in \{0,1\}^n$ and for every i, j, $0 \leq i, j \leq n-1$, $F_n^d(X)$ will have a 1 on position $i \cdot n + j$ iff $i \ mod \ 3 = j \ mod \ 3 = 0$, and there is a left parenthesis $\{$ on positions $i, i+1, i+2$, and a matching right parenthesis $\}$ on positions $j, j+1, j+2$ (or similarly, for a pair of matching angle parentheses $\langle$ and $\rangle$ on these positions), and the nesting depth of parentheses enclosed between the positions i and j is $< d$. E.g. consider the string $X =$ "$\}_0\{_1\{_2\}_3\{_4\{_5\}_6\{_7\}_8\}_9\}_{10}\{_{11}$" which, after encoding each parenthesis with three bits, will be translated into a string $X \in \{0,1\}^{36}$. Then $F_{36}^2(X)$ reports the matching parentheses $\{_2, \}_3$, $\{_5, \}_6$, $\{_7, \}_8$, and $\{_4, \}_9$ (i.e. it will output an 1 on the positions $36i + j$, for $i, j = \langle 6, 9\rangle, \langle 15, 18\rangle, \langle 21, 24\rangle$, and $\langle 12, 27\rangle$). It will not report the matching parentheses $\{_1, \}_{10}$, because their nesting level is 3.

Proof. (of Lemma 9.1) We prove the statement by induction on d. For $d = 0$ there is nothing to check: $F_n^0(X)$ just returns the string $000\ldots0$. For $d = 1$, $F_n^1(X)$ returns an 1 on those positions $i \cdot n + j$ for which: both i and j are divisible by 3, $i < j$, $X[i:i+2]$ and $X[j:j+2]$ contain $\{$ and $\}$ respectively (or $\langle$ and $\rangle$), and $\forall k$, $k \ mod \ 3 = 0$, $i < k < j$, there is no parenthesis in $X[k:k+2]$.

For $d > 1$ we proceed as follows. First we compute $Y = F_n^{d-1}(X)$. Next replace in X every character corresponding to a matched parenthesis in Y with *blank*, and call X' the result, i.e. $\forall i. 0 \leq i \leq n-1$, if $i \ mod \ 3 = 0$, then, if $\exists j$ s.t. $Y[i \cdot n + j] = 1$ or $Y[j \cdot n + i] = 1$, then $X'[i:i+2] \stackrel{\text{def}}{=} blank$, else $X'[i:i+2] \stackrel{\text{def}}{=} X[i:i+2]$. Compute $Y' = F_n^1(X')$, and finally output Z, where $Z[i] \stackrel{\text{def}}{=} AND(Y[i], Y'[i])$, for $0 \leq i \leq n^2 - 1$.

We invite the reader to check that, for $d \geq 1$, a circuit α_n^d can be built for F_n^d, for which $height(\alpha_n^d) \leq 8d - 4$, and $\mathsf{size}(\alpha_n^d) \leq d \cdot (2n^3 + 3n^2 + 6n) - 5n$. $\square$

Corollary 9.2 *For each type t, there is some function $F^t = \bigcup_{n \geq 0} F_n^t$ in AC^0, $F_n^t : \{0,1\}^n \to \{0,1\}^{n^2}$ which identifies the pairs of parenthesis for any encoding of type t. More precisely for any encoding X of an object x of type t, $F_n^t(X)$ will have an 1 on position $i \cdot n + j$ iff there is a left parenthesis $\{$ on positions $i, i+1, i+2$, and a matching right parenthesis $\}$ on positions $j, j+1, j+2$, or similarly for a pair of matching angle parentheses $\langle\rangle$.*

Proof. This follows directly from Lemma 9.1, since any encoding X of an object of type x will have at most d nested parentheses, where d depends only on the type t. □

Lemma 9.3 *For any set type $\{t\}$, there is some function $B^{\{t\}} = \bigcup_{n\geq 0} B_n^{\{t\}}$ in AC^0, $B_n^{\{t\}} : \{0,1\}^n \to \{0,1\}^n$, which, for some encoding $\{X_1, \ldots, X_m\}$ of type $\{t\}$, returns a string containing exactly m 1's, namely on those positions where some X_i begins. Similarly, for every product type $t_1 \times \ldots \times t_k$, there exists a function $B^{t_1 \times \ldots \times t_k}$ in AC^0 which, when given an encoding $\langle X_1, \ldots, X_k\rangle$ of length n of $\langle x_1, \ldots, x_k\rangle \in t_1 \times \ldots \times t_k$, returns a string of length n containing exactly k 1's, namely on the positions where $X_1, \ldots, X_k$ start.*

Proof. The circuit computing $B_n^{\{t\}}$ identifies the outermost commas (i.e. those not included in any pair of matching parenthesis, except the outermost { }), and returns a 1 on each first non-blank position following such a comma, or following the leading left brace. □

As a consequence, we have:

Lemma 9.4 *For all types t, equality of objects of type t is computable in AC^0.*

Proof. We construct a family of circuits $E_n^t : \{0,1\}^{2n} \to \{0,1\}$ which, when given two encodings X, Y of $x, y \in t$, both of length n, returns 1 iff $x = y$. We proceed by induction on the type t. For the base case $t = \mathbb{D}$, we use essentially the fact that blanks are not allowed inside the binary representation of numbers. Then $E_n^{\mathbb{D}}(XY)$ will return 1 iff $\exists i, i', j, j'$, $0 \leq i < i' \leq n-1, 0 \leq j < j' \leq n-1$, $X[i : i'] = Y[j : j']$ (in particular $i' - i = j' - j$), and in $X[0 : i-1], X[i'+1 : n-1], Y[0 : j-1], Y[j'+1 : n-1]$ there are only blanks. For the induction case $t = \{t'\}$, we have $X = \{X_1, \ldots, X_p\}$ and $Y = \{Y_1, \ldots, Y_q\}$, with possible blanks scattered. We essentially test the following two conditions:

$$\forall i \leq p.\exists j \leq q.X_i = Y_j \tag{1}$$
$$\forall j \leq q.\exists i \leq p.X_i = Y_j \tag{2}$$

To do that, we start by using Lemma 9.3 to compute the stings U, V of length n, which identify the p positions in X and the q positions in Y where some element starts. Then condition 1 becomes:

$$\begin{aligned}\forall i, i' \quad . \quad & (i < i' \wedge U[i] = U[i'] = 1 \wedge \forall i'' i < i'' < i', U[i''] = 0) \Rightarrow \\ & \exists j, j'.j < j' \wedge V[j] = V[j'] = 1 \wedge (\forall j''.j < j'' < j' \Rightarrow V[j''] = 0) \wedge \\ & E_n^{t'}(U[i : i'-1]V[j : j'-1]) = 1\end{aligned}$$

In fact the expression $E_n^{t'}(U[i : i'-1]V[j : j'-1])$ is not quite correct: we have to eliminate the trailing *comma* (i.e. replace it with *blank*) in $U[i : i'-1]$ and $V[j : j'-1]$, and add add $n - i' + i - 1$ trailing blanks to $U[i : i'-1]$ and $V[j : j'-1]$ (to make them of length n) before computing $E_n^{t'}(U[i : i'-1]V[j : j'-1])$. We invite the reader to fill in the details. Condition 2 is handled similarly. □

Lemma 9.5 *For every set type $\{t\}$ there exists a "duplicate elimination function" $D^t = \bigcup D_n^t$, where D_n^t : $\{0,1\}^n \to \{0,1\}^n$, with the following meaning. Given any encoding $X = \{X_1, \ldots, X_m\}$ of some object of type $\{t\}$, with possible duplicates, $D^t(X)$ returns a string of the same length, in which all duplicates are replaced with blanks.*

Proof. $D_n^t(X)$ will work as follows. First compute $Y = B_n^t(X)$, i.e. identify where each of the m substrings $X_1, \ldots, X_m$ begins in X. Now we have to replace each character $X[i : i+2]$ belonging to some duplicate X_p, with $0 \leq i \leq n-1$, $i \ mod \ 3 = 0$, with a *blank*. More precisely we replace $X[i : i+2]$ with *blank* iff the following holds:

$$\begin{aligned}
\exists j \leq i.\exists k > i \quad . \quad & Y[j] = Y[k] = 1 \wedge (\forall l.j < l < k \Rightarrow Y[l] = 0) \wedge \\
& \exists j', k'.k \leq j' < k' \wedge Y[i'] = Y[k] = 1 \wedge (\forall l.j' < l < k' \Rightarrow Y[l] = 0) \wedge \\
& E_n^t(X[j : k-1]X[j' : k'-1])
\end{aligned}$$

The condition says that $X[i : i+2]$ lies inside some string X_p stretching between positions j and k, and that X_p equals X_q, with $p < q$, where X_q stretches between the positions j' and k'. The expression $E_n^t(X[j : k-1]X[j' : k'-1])$ is not quite correct: the possible trailing *commas* have to be eliminated (replaced with *blank*) in both $X[j : k-1]$ and $X[j' : k'-1]$, and then they have to be padded with *blanks*, to make them strings of length n. We invite the reader to fill in the details. $\square$

Finally we show that all queries in $\mathcal{NRA}^1(log\_loop)$ are in NC. We want to prove this by "induction" on the structure of a query, but here we run into the following technical difficulty. In our presentation of $\mathcal{NRA}$, queries do not admit an inductive definition, because subexpressions of some query f may not be queries themselves (they may be functions $f : t_1 \to t_2$ with free variables, or they may be complex object expressions $e : t$). To circumvent that, we associate to each expression a query, as follows. To any complex object expression $e : t$, and any set of variables $x_1^{t_1}, \ldots, x_k^{t_k}$ including all free variables of e, we associate the query $\lambda\langle x_1^{t_1}, x_2^{t_2}, \ldots x_k^{t_k}\rangle.e : t_1 \times t_2 \times \ldots t_k \to t$. Similarly, to any function expression $f : t \to t'$ we associate the query $\lambda\langle x^t, x_1^{t_1}, x_2^{t_2}, \ldots x_k^{t_k}\rangle.f(x^t) : t \times t_1 \times t_2 \times \ldots t_k \to t'$. When no confusion arises, we shall omit mentioning explicitly the set of variables, and talk about "the query f associated to some expression e".

Proposition 9.6 *The following hold for all $k \geq 0$:*

$$\begin{aligned}
\mathcal{NRA}^1(log_loop^{(k)}) & \subseteq Q\text{-}AC^k \\
\mathcal{NRA}(blog_loop^{(k)}) & \subseteq CQ\text{-}AC^k
\end{aligned}$$

Proof. It suffices to prove only the second inclusion: the first one follows. For that we prove by induction on some complex object expression $e \in \mathcal{NRA}(blog\_loop^{(k)})$ of type t that for any set of variables $x_1^{t_1}, \ldots, x_l^{t_l}$ which includes all free variables in e, the associated query

$$\lambda\langle x_1^{t_1}, x_2^{t_2}, \ldots, x_l^{t_l}\rangle.e : t_1 \times t_2 \times \ldots t_l \to t$$

is in AC^k. Simultaneously, we prove that for any function expression $f \in \mathcal{NRA}(blog\_loop^{(k)})$, the correspondent query is in AC^k. We illustrate only some of the cases for e and f.

Union $e \cup e'$. Let f and f' be the queries associated to e and e' respectively. Then the query associated to $e \cup e'$ is $g = \lambda x.f(x) \cup f'(x)$. Let f, f' be computed by the functions F and F' respectively, and let α_n and α'_n be circuits associated to F_n and F'_n. To compute g concatenate the outputs of α_n and α'_n, eliminate the braces }{ by replacing them with blanks and conditionally placing a comma (the comma is placed only when both outputs encode a nonempty set). Finally, eliminate the duplicates in the resulting set using Lemma 9.5.

Function application $f(e)$, where $f : t \to t'$ and $e : t$. Let $g : t \times t_1 \times t_2 \times \ldots t_l \to t$ be the query corresponding to f, and $h : t_1 \times t_2 \times \ldots t_l \to t$ be the query corresponding to e. Then the query corresponding to $f(e)$ is $k = \lambda x.g\langle h(x), x\rangle$, and this leads us directly to a circuit. Namely let α_n be the circuit computing g and β_n be the circuit computing h; let $Q(n)$ be the size of the output of β_n. Then the circuit computing k will consists in a copy of β_n, and a copy of $\alpha_{n+Q(n)+9}$. The inputs of $\alpha_{n+Q(n)+9}$ will consists in the concatenation of the following strings:

$$\text{“}\langle\text{''} \text{ the output of } \beta_n \text{ “,” the input } X \text{“}\rangle\text{''}$$

Extension *ext*(f). For sake of clarity, suppose that $f : t \to \{t'\}$ does not have free variables, and that $l = 0$. Then the query associated to f is f itself, and let α_n be a circuit for computing f; let $Q(n)$ be the size of the output of α_n. The circuit β_n for *ext*(f) will receive an input $X = \{X_1, X_2, \ldots, X_m\}$. Essentially it has to "apply" α_n to each substring X_p. It starts by computing $Y = B_n^t(X)$. Since it cannot anticipate where each substring X_p lies, it will have a copy of the circuit α_{j-i} for every pair $\langle i, j\rangle$, $0 \le i < j \le n-1$ for which $i \ mod \ 3 = j \ mod \ 3 = 0$, which will receive as input $X[i : j-1]$. The output of the circuit corresponding to $\langle i, j\rangle$ will be invalidated however (overwritten with *blank*s), unless $Y[i] = Y[j] = 1$ and $\forall k, i < k < j, Y[k] = 0$, i.e. unless there exists indeed some substring X_p which stretches from position i to position $j - 1$. All $n(n+1)/2$ results are concatenated, all pairs the inner parentheses $\}\{$ replaced with a *comma*, and finally we feed the resulting string (of size $\le Q(n) \cdot n \cdot (n+1)/2$) into $E^t_{Q(n)\cdot n\cdot(n+1)/2}$ to eliminate the duplicates.

Iteration *blog_loop*(f, b). Since the output type is a PS-types, we will assume, for sake of clarity, it is a set type, i.e. $f : \{t\} \to \{t\}$. Let $g : \{t\} \times t_1 \times \ldots \to \{t\}$ be the query associated to f, and $h : t_1 \times \ldots \to \{t\}$ be the query associate to b. Let α_n and β_n be the circuits for computing g and h respectively, and let G_n, H_n be the functions computed by them. Assume their output sizes to be $Q_g(n)$ and $Q_h(n)$ respectively. The input set y in $f\langle x, y\rangle$ cannot have more than n elements, (since at most $n/3$ characters are used to encode $\langle x, y\rangle$), hence it would suffice to generate $\lceil \log n\rceil$ copies of α_n, and to feed the output of every circuit as input to the next. However this naive approach does not work. Indeed, the output of the first α_n has size $Q_g(n)$, then the output of the second $\alpha_{Q_g(n)}$ has size $Q_g(Q_g(n))$, the output of the third $\alpha_{Q_g(Q_g(n))}$ has size $Q_g(Q_g(Q_g(n)))$, etc. Hence the output size grows more than a polynomial. We need a more subtle technique which uses the bound b in an essential way.

Let $p : (\{t'\} \times \{t\}) \times (t_1 \times \ldots) \to \{t\}$ be the query associated to *blog_loop*(f, b). The circuit computing p will receive as input a string $\langle\langle X, Y\rangle, Z\rangle$ of length n. It will start by computing H_n on the string $\langle Y, Z\rangle$ of length n (obtained from the input by overriding some characters with blanks). Let $U = H_n\langle Y, Z\rangle$. U is the encoding of the bounding set, $u = \{u_1, \ldots, u_l\}$, and *length*$(U) = Q_h(n)$. Next, the circuit for p contains $\lceil \log(n+1)\rceil$ copies of the circuit $\alpha_{Q_h(n)}$. The idea is that all intermediate results are subsets of u, hence $Q_h(n)$ bits will suffice to represent them. Namely we feed the first circuit $\alpha_{Q_h(n)}$ with the intersection of Y and U; we obtain Y from the input $\langle\langle X, Y\rangle, Z\rangle$ by overriding the rest with blanks, and we compute the intersection by overriding in U all elements which are not in Y with blanks, so the result has the same size as U. Similarly, we feed the $i + 1$ copy of the circuit $\alpha_{Q_h(n)}$ with the output of the i's copy, intersected with U, which is again a string of size $Q_h(n)$. Of course, we have to bypass all circuits $\alpha_{Q_h(n)}$ beyond level $\lceil \log(m+1)\rceil$, where m is the cardinality of the set encoded by X. To do that, we start by computing $B_n^{t'}(X)$. Next we compute a string of length $\lceil \log(n+1)\rceil$ representing the binary encoding of the number of 1's in $B_n^{t'}(X)$, i.e. the binary encoding of m: this can be done in AC^1, since it is a particular case of the problem of adding n binary numbers. Now we establish a correspondence between the $\lceil \log(n+1)\rceil$ bits representing m and the $\lceil \log(n+1)\rceil$ copies of $\alpha_{Q_h(n)}$: the least significant bit will correspond to the first copy, etc. We will bypass copy $\alpha_{Q_h(n)}$ all bits $i, i+1, i+2, \ldots$ in m are 0.

Finally, observe that, if the circuit $\alpha_{Q(n)}$ for computing g had depth $O(\log^k n)$, then the circuit for computing p has depth $O(\log^{k+1} n)$.

The other cases are treated in a similar fashion. We skip the proof of the uniformity, which is tedious but straightforward. ☐

Since the inequality $\leq$ can be computed in AC^0, Proposition 9.6 immediately implies:

Corollary 9.7 *$\mathcal{NRA}(blog\_loop^{(k)}, \leq) \subseteq CQ\text{-}AC^k$ and $\mathcal{NRA}^1(log\_loop^{(k)}, \leq) \subseteq Q\text{-}AC^k$*

Instead of designing a circuit for computing f, we could have shown that f can be computed in $FO[\log^k n] + \leq +BIT$, and then using the results in [Imm89, BIS90] to conclude $f \in AC^k$: in fact, this is the way we prove Proposition 6.5. But we chose to construct the circuit for computing f in order to suggest how f may compiled on a CRCW PRAM.

Putting together Propositions 7.1, 8.3 and 9.6, we get one direction of the inclusions in Theorems 6.1 and 6.2:

Proposition 9.8 *For all $k \geq 0$, $\mathcal{NRA}^1(dcr^{(k)}) \subseteq Q\text{-}AC^k$ and $\mathcal{NRA}(bdcr^{(k)}) \subseteq CQ\text{-}AC^k$. Moreover, if every operator $p \in \Sigma$ is in NC, then $\mathcal{NRA}^1(dcr, \Sigma) \subseteq NC$.*

10 Simulating Circuits

It remains to prove the other inclusion: $Q\text{-}AC^k \subseteq \mathcal{NRA}^1(dcr^{(k)})$. For that we start with an NC query f which we want to simulate in $\mathcal{NRA}^1(log\_loop, \leq)$. We do this in three steps: (1) Given an input x to f, we compute the minimal encoding X for x, and represent it as a complex object. (2) Assuming $n = length(X)$, we simulate the Turing Machine from the DLOGSPACE-DCL uniformity, and construct the circuit α_n which computes f on inputs of size n; again, we encode α_n as a complex object. (3) We simulate the circuit α_n on the input X in $\mathcal{NRA}^1(log\_loop, \leq)$. Here we have to iterate $\log^k n$ times, where $\log^k n$ is the height of α_n: k nestings of *log_loop* suffice for that. (4) Finally we "decode" the string Y resulting from the simulation of α_n.

For step (2) above we adopt techniques for simulating *PTIME* Turing Machines with query languages over finite, ordered structures [Imm86, Var82], and over complex objects [GV91c]. For step (1) we take the (ordered) input x and construct some set z whose cardinality is at least as large as the size of the minimal encoding of x (see section 4). The type of z may depend on the value x: but there are a fixed number of choices, and we may test (using **if**'s) which one is the right choice. E.g., if $t = \{\mathbb{D}\}$, then $x = \{x_1, \ldots x_m\}$, where $m = card(x)$. Then x can be encoded with $n \leq 6 * m^2 + 3$ bits, since each element in x can be encoded with at most $\lceil \log(m+1) \rceil \leq m$ characters 0,1, each is followed by one additional comma (and the last element is followed by the right brace), and each character uses, in fact, 3 bits. Finally we need the left brace (hence the $\ldots + 3$). For this case, it suffices to take $z \stackrel{\text{def}}{=} x \times x \times x$, of type $\{\mathbb{D} \times \mathbb{D} \times \mathbb{D}\}$ when $m > 2$, because $card(z) = m^3 \geq 6 * m^2 + 3$ for $m \geq 3$. But for $m \leq 2$, we choose $z \stackrel{\text{def}}{=} \{true, false\}^5$ of type $\{\mathbb{B}^5\}$, which has $32 > 6 * m^2 + 3$ elements.

Lemma 10.1 *For every type t there exists n_t conditions $C_1^t, \ldots, C_{n_t}^t : t \to \mathbb{B}$, n_t types $s_1^t, \ldots, s_{n_t}^t$, and n_t functions $b_1^t : t \to \{s_1^t\}, \ldots, b_{n_t}^t : t \to \{s_{n_t}^t\}$ in $\mathcal{NRA}$, such that the following hold. Let $x \in t$, and let $X \in \{0,1\}^*$ be the minimal encoding of x. Then:*

- $\exists i$ *such that* $C_i^t(x) = true$.
- $\forall i$, $C_i^t(x) = true \Rightarrow length(X) \leq card(b_i^t(x))$.

Moreover, $height(\{s_i^t\}) \leq \max(1, height(t))$, *for* $i = 1, n_t$.

This says that, for every type t and any object x of type t, we may construct a set z whose cardinality is greater than or equal to the length of the minimal encoding for x: for different values x we may be forced to choose between n_t different types for the set z, but there are conditions $C_1^t, \ldots, C_{n_t}^t$ telling us, for every x, which particular type for z to choose.

Proof. We prove the lemma by induction on the type t. We will abbreviate the set $\{false, true\} \times \{false, true\}$ with 4, and the set $\{false, true\}$ with 2.

Case $t = \mathbb{D}$ Then we take $n_t = 1$, $C_1^t(x) \stackrel{\text{def}}{=} true$, $s_1^t \stackrel{\text{def}}{=} \mathbb{B} \times \mathbb{B}$, and $b_1^t(x) \stackrel{\text{def}}{=} 4$. Indeed, $\forall x \in \mathbb{D}$, the minimal encoding of x is 0, which has length 3 (recall that we encode each symbol of the alphabet A with three bits).

Case $t = \{t'\}$ Let $x \in t$, $x = \{x_1, \ldots, x_n\}$, and let $a \stackrel{\text{def}}{=} atoms(x)$, $a \in \{\mathbb{D}\}$. We first apply induction hypothesis to t', to get conditions $C_1^{t'}, \ldots, C_{n_{t'}}^{t'}$, and types $s_1^{t'}, \ldots, s_{n_{t'}}^{t'}$. Next observe that the minimal encoding of X is $\{X_1, \ldots, X_n\}$. Obviously $X_1, \ldots, X_n$ are *not* minimal encodings of $x_1, \ldots, x_n$ respectively (because x_j does not necessarily mention all atoms in a), but their length is at most a factor of $3 \cdot \lceil \log(card(a) + 1) \rceil$ larger than the length of their minimal encoding (this is a generous upper bound). Observe that $3 \cdot \lceil \log(card(a) + 1) \rceil \leq 4 \times card(a)$, when $a \neq \emptyset$. Hence, to obtain a set of cardinality $\geq length(X_j)$ when $C_i^{t'}(x_j)$ is true and $a \neq \emptyset$, it suffices to take the set $4 \times a \times b_i^{t'}(x_j)$. The latter set is included in $4 \times a \times ext(b_i^{t'})(x)$. Finally, if x_j has the longest encoding among $x_1, \ldots, x_n$, then the length of the minimal encoding of x is at most $2n$ times larger than the encoding of x_j: hence the cardinality of the set $2 \times x \times (4 \times a \times ext(b_i^{t'})(x))$ is larger than the length of X. However, when $a = \emptyset$, then the latter set has cardinality 0. Here we argue that size of X is bounded by a constant which depends only on the type t, and we take as bounding set $\{false, true\}^k$, with k sufficiently large. In conclusion we define $n_t \stackrel{\text{def}}{=} n_{t'} + 1$, and:

$$
\begin{aligned}
C_i^t(x) &\stackrel{\text{def}}{=} a \neq \emptyset \wedge \exists x_j \in x.(C_i^{t'}(x_j) \wedge \forall x_{j'} \in x(C_{i'}^{t'}(x_{j'}) \Longrightarrow card(b_i^{t'}(x_j)) \geq card(b_{i'}^{t'}(x_{j'}))))\\
s_i^t &\stackrel{\text{def}}{=} \mathbb{B} \times t' \times (\mathbb{B}^2 \times \mathbb{D} \times s_i^{t'})\\
b_i^t(x) &\stackrel{\text{def}}{=} 2 \times x \times (4 \times a \times ext(b_i^{t'})(x))
\end{aligned}
$$

for $i = 1, n_{t'}$, and

$$
C_{n_{t'}+1}^t(x) \stackrel{\text{def}}{=} a = \emptyset \quad s_{n_{t'}+1}^t \stackrel{\text{def}}{=} \mathbb{B}^k \quad b_{n_{t'}+1}^t(x) \stackrel{\text{def}}{=} \{false, true\}^k
$$

The other cases are handled using similar techniques. □

Proposition 10.2 *For all* $k \geq 1$ *CQ-AC*$^k \subseteq \mathcal{NRA}(blog\_loop^{(k)}, \leq)$ *and Q-AC*$^k \subseteq \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$.

Proof. Let f be in $CQ\text{-}AC^k$, $f : t \to t'$. We shall construct an expression f' in $\mathcal{NRA}(blog\_loop^{(k)}, \leq)$ equivalent to f. This proves $CQ\text{-}AC^k \subseteq \mathcal{NRA}(blog\_loop^{(k)}, \leq)$. For $Q\text{-}AC^k \subseteq \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$ it will suffice to observe that, when $height(t) \leq 1$ and $height(t') \leq 1$, then $f' \in \mathcal{NRA}^1(blog\_loop^{(k)}, \leq)$ and, hence, $f' \in \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$, by a simple extension of Proposition 3.5 to *blog_loop*.

Since $f \in CQ\text{-}AC^k$ it is computed by some function $F : \{0,1\}^* \to \{0,1\}^*$. F is given by: (1) A *DLOGSPACE* Turing Machine T accepting the *DCL* of a family of circuits, and (2) Polynomials $P(n)$ and $Q(n)$ (see Section 4). For some input $x \in t$, let n be the length of the *minimal encoding* X of x (see Section 5). The simulation of F in $\mathcal{NRA}^k(blog\_loop, \leq)$ is described below.

1. Apply Lemma 10.1 to construct some set z having a cardinality $\geq n$. The type of z may depend on the value x: but there are a fixed number of choices, and we may test (using **if**'s) which one is the right choice. Observe that when $height(t) \leq 1$ then the type of z will be flat and z can be computed in $\mathcal{NRA}^1$.
2. Some power z^l of z will have $p = n^l$ elements, enough to perform all the arithmetic needed in the sequel. Over this ordered set, we precompute the functions plus, minus, multiplication, and bit, on the numbers $0, \ldots, p-1$. See e.g. the proof of Proposition 8.3 for the computation of addition, and see Example 6.4 for a hint on how to compute multiplication using *bdcr* and, hence, *blog_loop*. As a result, we obtain four relations *plus*, *minus*, *mult*, and *bit* of type $\{t \times t \times t\}$, where $\{t\}$ is the type of z^l. We only need one nesting level of *blog_loop* to compute each of them.
3. Compute the minimal encoding X of x, of length n, without blanks: a string $X \in \{0,1\}^*$ of length n is represented as a set of "numbers", i.e. a subset of z^l. The computation is done in $\mathcal{NRA}^1(blog\_loop, \leq)$, using the previously computed relations *plus*, *minus*, *mult*, *bit* as parameters; the *blog_loop* is needed to compute the sum of a set of numbers. As a byproduct of this encoding, we also obtain a translation table from atomic values in x to numbers.
4. Simulate F on X, as described below, to get $Y = F(X)$. Again we use *plus*, *minus*, *mult*, *bit* as parameters.
5. Finally "decode" Y, to get $y \in t$. Decoding is done in $\mathcal{NRA}$, i.e. no loops are necessary, and uses the translation table constructed at point 3. The decoding consists essentially in parsing the string Y and constructing y. The parser depends on the type t. E.g., when $t = \{t'\}$, then we construct y as follows. We built the set of pairs of number (i, j), $i < j$ (this is just a subset of $z^l \times z^l$), and for each such pair, we use the parser for the type t' to test whether some element is encoded in Y between the positions i and j, and if so, to decode that element. Finally we construct the set of all such elements. So essentially we apply *ext* on the parser for the type t'.

There are two ways of simulating F on X. One is to use the result in [Imm89] which says that, since F is in AC^k, F is also in $FO[\log^k n]+ \leq +BIT$, and to observe that, for $k \geq 1$, $FO[\log^k n]+ \leq +BIT \subseteq \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$. The second way is to use the *DLOGSPACE-DCL*-uniformity definition of AC^k. We will follow the second path here. First, we simulate the $O(\log n)$ space Turing Machine computing the *DCL* of α_n: this can be done, since there are only polynomially many configurations for T, and deciding whether T accepts some input (n, g, g', t) reduces, using standard techniques, to the computation of the transitive closure of the successor relation on the set of configurations of T. Secondly, we simulate the circuit α_n itself, by computing step by step the outputs of the gates at each level: this only requires $\log^k n$ iterations, so it can be done in $\mathcal{NRA}^1(log\_loop^{(k)}, \leq)$ and, hence, in $\mathcal{NRA}(blog\_loop^{(k)}, \leq)$. We observe that *ext* is used in an essential way at each iteration step, accounting for the parallelism in the evaluation of α_n.

Finally we argue that the nesting depth of the resulting expression is no larger than k. We saw already that the circuit α_n can be simulated by some function, say $h(X)$, where the nesting depth of h is k, but where h uses the tables $plus, minus, \ldots$ as free variables. We have to show that these tables do not increase the nesting depth. Indeed, the full expression of the simulation of α_n will then be

$$e \stackrel{\text{def}}{=} (\lambda\langle plus, minus, \ldots\rangle.h(X))\langle f_{plus}(z^l), f_{minus}(z^l), \ldots\rangle$$

where $f_{plus}(z^l)$ computes the addition table *plus* on z^l, etc. By definition of the nesting depth (Subsection 3.1) we have $depth(e) \stackrel{\text{def}}{=} \max(depth(h(X)), depth(f_{plus}), depth(f_{minus}), \ldots) = \max(depth(h(x)), 1) = depth(h(X)) = k$ when $k \geq 1$: indeed, the iterations in f_{plus} and $h(X)$ are not *nested* but computed *sequentially.*

This proved

$$CQ\text{-}AC^k \subseteq \mathcal{NRA}(blog\_loop^{(k)}, \leq), \quad \forall k \geq 1$$

If t and t' are both flat types, then all the computations describe above are in $\mathcal{NRA}^1(blog\_loop^{(k)}, \leq)$, which is equal to $\mathcal{NRA}^1(log\_loop^{(k)}, \leq)$, by a straightforward extension of Proposition 3.5 to *log_loop*. Hence, $Q\text{-}AC^k \subseteq \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$. □

Now we can prove our main results. Theorems 6.2 and 6.1 follow from Corollary 9.7 and Proposition 10.2. Proposition 6.5 follows from Proposition 9.6, and Proposition 6.3 is proven by an straightforward extension of the proof of Proposition 9.6.

11 Conclusions

Ordering seems to play a crucial role in capturing complexity classes below *NP* and our characterization is no exception. Indeed, it follows from Theorem 7.8 in [IPS91] that in the absence of ordering $FO + dcr$ cannot express the lower bound in [CFI92] which is in AC^0 plus parity gates ([CFI92] remark 7.2). As with *PTIME*, *DLOGSPACE*, etc., it remains an important open question whether there exists an r.e. set of "programs" that express exactly the *NC*-computable queries over arbitrary relational databases.

On the other hand, studying the expressiveness of the various forms of recursion on sets in the absence of ordering is quite relevant to query language design. It may also be relevant to complexity theory, if an analog to the surprising result of Abiteboul and Vianu [AV91] holds. They have shown that $PTIME \neq PSPACE$ iff first-order least fixpoint queries $\neq$ first-order *while* queries. (Vardi had shown that in the presence of order the *FO+while* captures *PSPACE* [Var82].) Dawar, Lindell, and Weinstein [DLW93] give a machine-independent proof of the Abiteboul and Vianu result making use of properties of bounded variable logics. Abiteboul, Vardi and Vianu [AVV92] give evidence for the robustness of the idea with several such results for other pairs of complexity classes. In our case, the analog would be: $NC \neq PTIME$ iff $FO + dcr \neq FO + sri$ (in our formalism, $\mathcal{NRA}^1(dcr) \neq \mathcal{NRA}^1(sri)$). By setting aside the ordering, with its potential for tricky encodings, this would strengthen the observation (Section 6) that the difference between tractable sequential and tractable parallel computation can be characterized as the difference between two ways of recurring on sets.

Acknowledgments. We thank Scott Weinstein for many illuminating discussions, Neil Immerman for answering our sometimes naive queries, Peter Buneman and Leonid Libkin for suggestions from a careful reading of an earlier version of this paper, and Peter, Leonid, and Limsoon Wong for their constant help.

References

[AB88] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988. Also available as INRIA Technical Report 846.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[AV91] Serge Abiteboul and Victor Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, 1991.

[AVV92] S. Abiteboul, M. Vardi, and V. Vianu. Fixpoint logics, relational machines, and computational complexity. In *Structure and Complexity*, 1992.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proceedings of 13th International Conference on Very Large Data Bases*, pages 97–105, 1987.

[BBW92] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[BIS90] David Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306, 1990.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of proramming with collection types. *Theoretical Computer Science*, 1995. To appear.

[BT92] Val Breazu-Tannen. Generalized structural recursion and sets vs. bags, January 1992. Manuscript available from `val@saul.cis.upenn.edu`.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[CFI92] Jin-Yi Cai, Martin Furer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[CH80] Ashok Chandra and David Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[CL90] Kevin L. Compton and Claude Laflamme. An algebra and a logic for NC. *Information and Computation*, 87(1/2):240–262, 1990.

[Clo90] P. Clote. Sequential, machine-independent characterizations of the parallel complexity classes $AlogTime$, AC^k, NC^k, and NC. In Samuel R. Buss and Philip J. Scot, editors, *Feasible Mathematics*. Birkhäuser, Boston, 1990.

[Coo85] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[DLW93] Anuj Dawar, Steven Lindell, and Scott Weinstein. Infinitary logic and inductive definability over finite structures. *Information and Computation*, 1993. To appear. Available as UPenn Technical Report MS-CIS-91-97.

[DV91] K. Denninghof and V. Vianu. The power of methods with parallel semantics. In *Proceedings of 17th International Conference on Very Large Databases*, 1991.

[Fag93] R. Fagin. Finite model theory — a personal perspective. *Theoretical Computer Science*, 116(1):3–32, August 1993.

[Gur83] Y. Gurevich. Algebra of feasible functions. In *Proceedings of 24th IEEE Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

[GV91a] Stephane Grumbach and Victor Vianu. Expressiveness and complexity of restricted languages for complex objects. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 191–202. Morgan Kaufmann, August 1991.

[GV91b] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. Technical Report 1573, INRIA, Rocquencourt BP 105, 78153 Le Chesnay, France, December 1991. Extended abstract appeared in PODS 91.

[GV91c] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, 1991.

[GV95] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. *Journal of Computer and System Sciences*, 51(2):149–167, 1995.

[Imm82] Neil Immerman. Upper and lower bounds for first-order expressibility. *Journal of Computer and Systems Sciences*, 25:76–98, 1982.

[Imm86] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Imm87a] Neil Immerman. Expressibility as a complexity measure: Results and directions. In *Proceedings of 2nd Conference on Structure in Complexity Theory*, pages 194–202, 1987.

[Imm87b] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[Imm89] Neil Immerman. Expressibility and parallel complexity. *SIAM Journal of Computing*, 18:625–638, 1989.

[IPS91] Neil Immerman, Sushant Patnaik, and David Stemple. The expressiveness of a family of finite set languages. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, pages 37–52, 1991.

[LW94a] Leonid Libkin and Limsoon Wong. Aggregate functions, conservative extension, and linear orders. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 282–294. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-36.

[LW94b] Leonid Libkin and Limsoon Wong. New techniques for studying set languages, bag languages, and aggregate functions. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 155–166, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-93-95.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North Holland, 1974.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[PG88] Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proceedings of 7th ACM Symposium on Principles of Database Systems,Austin, Texas*, pages 29–38, 1988.

[PG92] Jan Paredaens and Dirk Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[PSV92] D. Stott Parker, Eric Simon, and Patrick Valduriez. SVP: A model capturing sets, streams, and parallelism. In Li-Yan Yuan, editor, *Proceedings of 18th International Conference on Very Large Databases, Vancouver, August 1992*, pages 115–126, San Mateo, California, August 1992. Morgan-Kaufmann.

[RB90] M. Sipser R. Boppana. The complexity of finite functions. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science. Vol A: Algorithms and Complexity.* MIT Press, 1990.

[RW93] Pushpa Rao and Clifford Walinsky. An equational language for data-parallelism. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 112–118. ACM Press, 1993.

[Saz93] Vladimir Yu. Sazonov. Hereditarily-finite sets, data bases and polynomial-time computability. *Theoretica Computer Science*, 119:187–214, 1993.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[Suc94] Dan Suciu. Fixpoints and bounded fixpoints for complex objects. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 263–281. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-32.

[SV84] Larry Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal of Computing*, 13:409–422, May 1984.

[SW95] Dan Suciu and Limsoon Wong. On two forms of structural recursion, January 1995.

[TF86] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis and F. P. Preparata, editors, *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of 14th ACM SIGACT Symposium on the Theory of Computing*, pages 137–146, San Francisco, California, 1982.

[Won93] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993. See also UPenn Technical Report MS-CIS-92-59.

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

APPENDICES

A Genericity

In this Appendix we review Chandra and Harel's definition of query genericity [CH80] and extend it to the case of queries over ordered databases. To define genericity, consider some subset $\mathbb{D}' \subseteq \mathbb{D}$. Then for every type t, let t' be the type obtained by replacing each occurrence of $\mathbb{D}$ with $\mathbb{D}'$. Obviously $t' \subseteq t$. Let $\varphi : \mathbb{D}' \to \mathbb{D}$ be an injective function. For every type t, we define the extension of φ to t to be $\varphi_t : t' \to t$ as follows:

$$
\begin{aligned}
\varphi_{\mathbb{D}}(x) &\stackrel{\text{def}}{=} \varphi(x) \\
\varphi_{\mathbb{B}}(x) &\stackrel{\text{def}}{=} x \\
\varphi_{t_1 \times \ldots \times t_k}(\langle x_1, \ldots, x_k \rangle) &\stackrel{\text{def}}{=} \langle \varphi_{t_1}(x_1), \ldots, \varphi_{t_k}(x_k) \rangle \\
\varphi_{\{t\}}(\{x_1, \ldots, x_n\}) &\stackrel{\text{def}}{=} \{\varphi_t(x_1), \ldots, \varphi_t(x_n)\}
\end{aligned}
$$

Definition A.1 *We call a function $f : t_1 \to t_2$* **generic** *if, for every injective function $\varphi : \mathbb{D}' \to \mathbb{D}$, the following holds: $\forall x \in t_1'.\varphi_{t_2}(f(x)) = f(\varphi_{t_1}(x))$.*

In the case of *ordered databases*, i.e. in which a total order relation is defined on the base type $(\mathbb{D}, \leq)$ we call f generic if $\varphi_{t_2} \circ f = f \circ \varphi_{t_1}$ for all *order-preserving* injective functions $\varphi : (\mathbb{D}', \leq) \to (\mathbb{D}, \leq)$.

This definition A.1 naturally extends that of Chandra and Harel [CH80] to ordered databases. In their case φ only ranges over bijections $\varphi : \mathbb{D} \to \mathbb{D}$. For ordered databases, this is not enough for our purposes. Indeed, consider the case in which $\mathbb{D}$ is the set of natural numbers. Then, there is only order-preserving bijection $\varphi : (\mathbb{D}, \leq) \to (\mathbb{D}, \leq)$, namely the identity, and in this case, under the definition in [CH80] all queries are generic. By slightly changing the definition to allow the domain of φ to be any subset $\mathbb{D}'$ of $\mathbb{D}$, we force f to commute with order-preserving renamings of elements of $\mathbb{D}$.

B Type Lifting

We prove here Proposition 3.5. For this we need the technique of *type lifting*. For any type t we define the **lifted type** $\bar{t}$ to be the type obtained by putting set parenthesis around all the base types it contains. Formally:

$$
\begin{aligned}
\overline{\mathbb{B}} &\stackrel{\text{def}}{=} \{\mathbb{B}\} \\
\overline{\mathbb{D}} &\stackrel{\text{def}}{=} \{\mathbb{D}\} \\
\overline{\{t\}} &\stackrel{\text{def}}{=} \{t\} \\
\overline{t_1 \times \ldots \times t_k} &\stackrel{\text{def}}{=} \bar{t}_1 \times \ldots \times \bar{t}_k
\end{aligned}
$$

Note that for every t, $\bar{t}$ is a PS-type. In addition we define $\mathit{lift}_t : t \to \bar{t}$, which puts set parenthesis around all the elements of base type that it contains, e.g. $\mathit{lift}_{\mathbb{D}} \stackrel{\text{def}}{=} \{x\}$, $\mathit{lift}_{t_1\times\ldots\times t_k}\langle x_1,\ldots,x_k\rangle \stackrel{\text{def}}{=} \langle \mathit{lift}_{t_1}(x_1),\ldots,\mathit{lift}_{t_k}(x_k)\rangle$, and $\mathit{lift}_{\{t\}}(x) \stackrel{\text{def}}{=} x$. Note that when t is a PS-type, then $\bar{t} = t$ and $\mathit{lift}_t(x) = x$.

For a given function $f : t \to t'$ where t' is a PS-types, we define $\bar{f} : \bar{t} \to t'$ as follows:

$$\bar{f}(x) \stackrel{\text{def}}{=} \bigcup\{f(z) \mid \mathit{lift}_t(z) \subseteq x\}$$

(Recall that we extend the union operation to PS-types.) In the particular case when $t = \mathbb{D}$ and t' is a set type, $t' = \{t''\}$, then we have $f : \mathbb{D} \to \{t''\}$, $\bar{f} : \{\mathbb{D}\} \to \{t''\}$, and $\bar{f} = \mathit{ext}(f)$. Similarly, when $f : \mathbb{D}\times\mathbb{D} \to \{t''\}$ then $\bar{f} : \{\mathbb{D}\} \times \{\mathbb{D}\} \to \{t''\}$ is $\bar{f}\langle x, y\rangle = \mathit{ext}(\lambda u.\mathit{ext}(\lambda v.f\langle u, v\rangle)(y))(x)$. In general:

Proposition B.1 *Whenever f is expressible in $\mathcal{NRA}^1$, $\mathcal{NRA}^1(dcr)$, or $\mathcal{NRA}^1(bdcr)$, then $\bar{f}$ is expressible in the same language.*

Example B.2 *Let $t = \mathbb{D} \times (\{\mathbb{D}\times\mathbb{D}\} \times \mathbb{B})$, $t' = \{\mathbb{D}\} \times \{\mathbb{D}\}$, and $f : t \to t'$. Then $\bar{f}$ is defined by:*

$$\bar{f}\langle x_1, \langle x_2, x_3\rangle\rangle \stackrel{\text{def}}{=} \bigcup\{f\langle z_1, \langle x_2, z_3\rangle\rangle \mid z_1 \in x_1, z_3 \in x_3\}$$

and can be defined as:

$$\bar{f}\langle x_1, \langle x_2, x_3\rangle\rangle = \mathit{ext}(\lambda z_1.\mathit{ext}(\lambda z_3.f\langle z_1, \langle x_2, z_3\rangle\rangle)(x_3))(x_1)$$

Fact B.3 *Let $f : t \to t'$, where t' is a PS-type. Then $\bar{f}(\mathit{lift}_t(x)) = f(x)$.*

Proof. (of Proposition 3.5) The inclusion $\mathcal{NRA}^1(bdcr) \subseteq \mathcal{NRA}^1(dcr)$ is obvious. For the inclusion $\mathcal{NRA}^1(dcr) \subseteq \mathcal{NRA}^1(bdcr)$, consider some expression $dcr(e, f, u) : \{t\} \to t'$ in $\mathcal{NRA}^1(dcr)$. First we lift t' to a PS-type $\bar{t}'$. Also, we define $e' : \bar{t}'$, $f' : t \to \bar{t}'$ and $u' : \bar{t}' \times \bar{t}' \to \bar{t}'$ as follows:

$$\begin{aligned}
e' &\stackrel{\text{def}}{=} \mathit{lift}_{t'}(e) \\
f' &\stackrel{\text{def}}{=} \mathit{lift}_{t'} \circ f \\
u' &\stackrel{\text{def}}{=} \overline{\mathit{lift}_{t'} \circ u}
\end{aligned}$$

If u is associative, commutative, and has identity e on some set $s \subseteq t'$, then u' will still be associative, commutative, and will have identity e', on the subset $\{\mathit{lift}(x) \mid x \in s\}$ of $\bar{t}'$. Moreover, we have $dcr(e', f', u') = \mathit{lift}_{t'} \circ dcr(e, f, u)$. To see that, consider for illustration a set with three elements, $s = \{x_1, x_2, x_3\}$. Then:

$$\begin{aligned}
dcr(e', f', u')(\{x_1, x_2, x_3\}) &= u'\langle f'(x_1), u'\langle f'(x_2), f'(x_3)\rangle\rangle \\
&= u'\langle f'(x_1), \overline{\mathit{lift}_{t'} \circ u}\langle \mathit{lift}_{t'}(f(x_2)), \mathit{lift}_{t'}(f(x_3))\rangle\rangle \\
&= u'\langle f'(x_1), \overline{\mathit{lift}_{t'} \circ u}(\mathit{lift}_{t'\times t'}\langle f(x_2), f(x_3)\rangle)\rangle \quad \text{by the definition of } \mathit{lift}_{t'\times t'} \\
&= u'\langle f'(x_1), \mathit{lift}_{t'}(u\langle f(x_2), f(x_3)\rangle)\rangle \quad \text{by Fact B.3} \\
&= \mathit{lift}_{t'}(u\langle f(x_1), u\langle f(x_2), f(x_3)\rangle\rangle) \quad \text{by repeating the steps above} \\
&= \mathit{lift}_{t'}(dcr(e, f, u))(\{x_1, x_2, x_3\})
\end{aligned}$$

Now assume without loss of generality that e, f, u have equivalent expressions in $\mathcal{NRA}^1(bdcr)$, and, hence, so do e', f', u'. Let $x_1, \ldots, x_k$ be all free variables in the expressions e, f, and u, and let s be the input set to $dcr(e', f', u')$. We compute a *bound* for $dcr(e', f', u')$ as follows. First we define $b_{\mathbb{D}} \in \{\mathbb{D}\}$ to be the set of all values of type $\mathbb{D}$ mentioned in $x_1, \ldots, x_k$, and s; obviously $b_{\mathbb{D}}$ can be computed in $\mathcal{NRA}^1$. Next, we define $b_{\mathbb{B}} \in \{\mathbb{B}\}$ to be $b_{\mathbb{B}} \stackrel{\text{def}}{=} \{false, true\}$. Finally we compute the bound b of type $\bar{t}'$ by pairing proper cartesian products of $b_{\mathbb{D}}$ and $b_{\mathbb{B}}$. E.g. when $\bar{t}' = \{\mathbb{D}\} \times (\{\mathbb{D} \times \mathbb{D}\} \times \{\mathbb{B}\})$, we take $b \stackrel{\text{def}}{=} \langle b_{\mathbb{D}}, \langle b_{\mathbb{D}} \times b_{\mathbb{D}}, b_{\mathbb{B}} \rangle\rangle$. Then it is easy to check that $\lambda s.bdcr(e', f', u', b)(s) = dcr(e', f', u')$ (note that s is a free variable occurring in the expression b).

So let $s \in \{t\}$ and $r = dcr(e, f, u)(s)$. We have shown how $lift_{t'}(r)$ can be computed in $\mathcal{NRA}^1(bdcr)$. Finally observe that we can recover r from $lift_{t'}(r)$ by using the function get. □

C The Role of *get*

We show in this Appendix that even at flat types dcr is slightly more expressive than $bdcr$, because it can extract an element of type $\mathbb{D}$ from a singleton set, as we have seen in Example 3.3.

Throughout this appendix we shall denote with $\mathcal{NRA}_-$ the language $\mathcal{NRA}$ without get.

Proposition C.1 *get cannot be expressed in $\mathcal{NRA}_-(bdcr)$.*

Proof. (Sketch) Here we prove by induction on the structure of an expression e in $\mathcal{NRA}_-(bdcr)$ that the following holds. Let $f : t \to t'$ be the associated query (i.e. the function $\lambda\langle x_1, \ldots, x_k\rangle.e$, where $x_1, \ldots, x_k$ are all free variables in e; see Section 9). Then we show that each "scalar component" of $f(x)$ is one of the "scalar components" of x. More precisely, for every type t define $scalar_t : t \to \{\mathbb{D}\}$ to be:

$$\begin{aligned} scalar_{\mathbb{B}}(x) &\stackrel{\text{def}}{=} \emptyset \\ scalar_{\mathbb{D}}(x) &\stackrel{\text{def}}{=} \{x\} \\ scalar_{\{t\}}(x) &\stackrel{\text{def}}{=} \emptyset \\ scalar_{t_1 \times \ldots \times t_k}\langle x_1, \ldots, x_k\rangle &\stackrel{\text{def}}{=} scalar_{t_1}(x_1) \cup \ldots \cup scalar_{t_k}(x_k) \end{aligned}$$

Then, for every query $f : t \to t'$ associated to some expression in $\mathcal{NRA}_-(bdcr)$ we prove that the following condition holds:

$$\forall x \in t, scalar_{t'}(f(x)) \subseteq scalar_t(x) \tag{3}$$

Certainly get does not satisfy condition 3, because $scalar_{\mathbb{D}}(get\langle\{d_1\}, d_2\rangle) = scalar_{\mathbb{D}}(d_1) = \{d_1\}$, while $scalar_{\{\mathbb{D}\}\times\mathbb{D}}(\langle\{d_1\}, d_2\rangle) = \{d_2\}$.

We prove condition 3 by induction on the structure of an expression e in $\mathcal{NRA}_-(bdcr)$. We consider some relevant cases for e:

Projection For illustration take $e = \pi_1(e')$, where $e' : t' \times t'', e : t'$. Then $f = \lambda x.\pi_1(f'(x))$, where f' is the query associated to e'. By induction hypothesis we have $scalar_{t' \times t''}(f'(x)) \subseteq scalar_t(x)$. It suffices to observe that: $scalar_{t'}(\pi_1(f'(x))) \subseteq scalar_{t' \times t''}(f'(x))$.

Union $e = e_1 \cup e_2$. Since the type t' is now a set type, $scalar_{t'}(f(x)) = \emptyset$, and there is nothing to prove.

Function application $e = g(e')$. For the sake of clarity suppose that $g : t'' \to t'$ does not have any free variables, hence its associated query is g itself. Let $f' : t \to t''$ be the query associated to e', so the query associated to e is $f = g \circ f'$. Then, using the induction hypotheses we have:

$$\begin{array}{rcl} scalar_{t'}(f(x)) & = & scalar_{t'}(g(f'(x))) \\ & \subseteq & scalar_{t''}(f'(x)) \\ & \subseteq & scalar_t(x) \end{array}$$

Bounded recursion As in the case of union, there is nothing to prove here, since the resulting type is a PS-type.

The remaining cases can be easily checked. $\square$

This result justifies the inclusion of *get* in the language $\mathcal{NRA}$. The next result shows that the additional expressive power brought by *get* is largely cosmetic. Indeed, we consider getting around the function *get* through type lifting, a technique for transforming a non PS-type into a PS-type defined in Subsection 3.2.

Proposition C.2 *Let $f : t \to t'$ be a query in $\mathcal{NRA}(bdcr)$ (or in $\mathcal{NRA}^1(bdcr)$). Then $lift \circ f : t \to \bar{t}'$ is expressible in $\mathcal{NRA}_-(bdcr)$ (or in $\mathcal{NRA}^1_-(bdcr)$ respectively), i.e. without get. Moreover, $\forall k \geq 0$, if $f \in \mathcal{NRA}(bdcr^{(k)})$ (or $f \in \mathcal{NRA}^1(bdcr^{(k)})$), then $lift \circ f \in \mathcal{NRA}_-(bdcr^{(k)})$ (or $lift \circ f \in \mathcal{NRA}^1_-(bdcr^{(k)})$ respectively).*

As a consequence, adding or dropping *get* from $\mathcal{NRA}$ does not affect in any way the queries whose result is a PS-type.

Proof. We proof by induction on the structure of an expression e in $\mathcal{NRA}^1(bdcr, get)$ that its associated query f (in the sense of Section 9) satisfies the above property. The case when e is the query $get : \{\mathbb{D}\} \times \mathbb{D} \to \mathbb{D}$ is trivial, because $lift_{\mathbb{D}} \circ get : \{\mathbb{D}\} \times \mathbb{D} \to \{\mathbb{D}\}$ is:

$$\lambda\langle x, y\rangle.\mathbf{if}\ card(x) = 1\ \mathbf{then}\ x\ \mathbf{else}\ \{y\}$$

which is expressible in $\mathcal{NRA}^1$. The only interesting case is when e is function application, i.e. $e = g(e')$. Let $f' : t \to t''$ be the query associated to e', and assume, for sake of clarity, that g has no free variables, i.e. the query associated to g is g itself. Then $f = g \circ f'$. By induction hypothesis we know that $lift_{t''} \circ f'$ and $lift_{t'} \circ g$ are expressible in $\mathcal{NRA}^1_-(bdcr)$, say by $h : t \to \bar{t}''$ and $h' : t'' \to \bar{t}'$. We prove that $lift \circ f = \bar{h}' \circ h$ and, since the latter is in $\mathcal{NRA}^1_-(bdcr)$, this concludes our proof. Indeed:

$$\begin{array}{rcll} \bar{h}'(h(x)) & = & \bar{h}'(lift_{t''}(f'(x))) & \\ & = & h'(f'(x)) & \text{by Fact B.3} \end{array}$$

$\square$

D Alternative Encoding of Flat Relations

Immerman [Imm89] proves a relationship between queries computable in extensions of FO and parallel complexity classes. His encoding of flat relations is different from ours; it is more elegant when dealing with flat relations, but does not extend to complex objects. We prove here that over flat relations our encoding is AC^0 equivalent to Immerman's, i.e. that the functions translating between the two encodings are in AC^0.

For sake of simplicity we will consider only the base type $\mathbb{D}$, and drop the types *unit* and $\mathbb{B}$. Then a *flat relation type* is a type of the form $\{\mathbb{D}^k\}$. For a flat relation $x \in \{\mathbb{D}^k\}$ we define the **bit-wise encoding** of r to be the following string $X \in \{0,1\}^*$. Let $d = \{d_0, d_1, \ldots, d_{n-1}\} \in \{\mathbb{D}\}$ be a set containing the *active domain* of x, i.e. the set of all values of type $\mathbb{D}$ mentioned in x, and assume $d_0 < d_1 < \ldots < d_{n-1}$. Then the encoding X associated to x and d will have length n^k and will be defined as follows. For every $i_1, i_2, \ldots, i_k$, $0 \leq i_1, i_2, \ldots i_k \leq n-1$, $X[i_1 n^{k-1} + i_2 n^{k-2} + \ldots + i_k] = 1$ iff $\langle d_{i_1}, \ldots, d_{i_k}\rangle \in x$.

Proposition D.1 *There exists a function $F : \{0,1\}^{n^k} \to \{0,1\}^{n^k(3kn+3k+6)+3}$ in AC^0 such that, for every string X representing the bit-wise encoding of some flat relation x, we have $x \sim F(X)$ and, moreover, $F(X)$ is a minimal encoding of x. Conversely, there exists a function $G : \{0,1\}^n \to \{0,1\}^{n^k}$ in AC^0 such that for every string X encoding some flat relation x, $G(X)$ is a bit-wise encoding of x.*

Proof. To compute $F(X)$ we proceed as follows. The output string will be formed of a leading left parentheses, $\{$, followed by n^k "cells" of $3kn+3k+6$ bits each. Each cell is split into a "body" built from the first $3kn+3k+3$ bits, and a "tail", containing the last three bits. We fill the cell corresponding to $i_1, i_2, \ldots, i_k$ either with blanks, when $X[i_1 n^{k-1} + i_2 n^{k-2} + \ldots + i_k] = 0$, or with $\langle i_1, \ldots, i_k\rangle$, otherwise. For the second case observe that each $i_1, \ldots, i_k$ are between 0 and $n-1$, hence each requires only $3\lceil\log(n+1)\rceil \leq 3n$ bits, for a total of $3kn$ bits, and we need an additional $3k+3$ bits to encode the brackets $\langle\rangle$ and the commas. In the first case we fill the "tail" with a blank, while in the second case we fill it with a comma, except for the last non-blank body, where we put a right parenthesis $\}$.

For $G(X)$ we observe first that the active domain of x has less than n elements, where $n = \mathit{length}(X)$. We start by computing $Y = B_n^{\{\mathbb{D}^k\}}(X)$, see Lemma 9.3. Then, for every $i_1, \ldots, i_k$, the bit number $i_1 n^{k-1} + i_2 n^{k-2} + \ldots + i_k$ of the output will be 1 iff $\exists j, j'\ 0 \leq j < j' \leq n-1$ s.t. $Y[j] = Y[j'] = 1$ and $\forall k. j < k < j', Y[k] = 0$, and $X[j : j'-1]$ equals $\langle i_1, \ldots, i_k\rangle$. The latter test can be made using a circuit of depth $O(k)$ (since we need k' levels of comparisons to identify the position of $y_{k'}$, for $1 \leq k' \leq k$). $\square$