A Query Language for NC

Dan Suciu
University of Pennsylvania
suciu@saul.cis.upenn.edu

Val Breazu-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 polylogarithmic time with polynomially many processors on a *parallel random access machine* (PRAM). The query language for NC discussed here is centered around a *divide and conquer recursion* (dcr) on sets which has obvious potential for parallel evaluation and can easily express, for example, transitive closure and parity. dcr with parameters e, f, u defines the unique function $\varphi = dcr(e, f, u)$ 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 $e \stackrel{\text{def}}{=}$ *false*, $f(y) \stackrel{\text{def}}{=}$ *true* and $u(v_1, v_2) \stackrel{\text{def}}{=} v_1 \ xor \ v_2$. 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 subclass AC^k of NC where $k \geq 1$ (definitions 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 class of queries expressed by 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 phrasing above. In fact, we will use the family of query languages introduced in [BBW92] because it is semantically related to dcr (section 3).

- 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 2). This is similar to the bounded fixpoints studied in [Suc94], and, as with fixpoints, over flat relations dcr can always be expressed through bounded dcr (section 2).

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

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

the name *hom*, it 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] (see section 2). Theoretical studies of expressiveness, such as [Won93, BTBN91, Suc94] 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 sublanguage for which we are currently seeking efficient implementation techniques for a variety of parallel architectures.

$\angle$ **Computational complexity.** Following Vardi and Immerman's influential result [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 $t(n)$. For complex object databases, Grumbach and Vianu [GV91b, GV91a] 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 2), 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 6.6. 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 finite and ordered 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 kinds of recursions 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. While *dcr* is a form of recursion on finite sets, these recursions are on notations for elements of (linearly ordered) finite sets. Of course, we do not attempt to characterize *DLOGSPACE* or NC^1 or, for that matter, any class below AC^1, but see Immerman's characterizations of such classes in terms of languages more in the spirit of ours than of those of Gurevich, Compton, and Laflamme [Imm87b, Imm87a].

We should also mention here the work of Clote [Clo90] for related characterizations of most parallel complexity classes of *arithmetical* functions. Also of related interest, but in a different direction, is the work of Denninghoff and Vianu [DV91] who characterize *NC* in terms of a resource-restricted message-passing model with parallel semantics which computes object-oriented queries.

We should point out, however, one sense in which our language is not as neat as, say, first-order logic with least fixpoint, which captures *PTIME* in the presence of order, or first-order logic with transitive closure, which captures *NLOGSPACE* in the presence of order. 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 2). Of course, only a certain family of instances of *dcr* is actually needed in the simulations, and for these, the algebraic conditions

[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.

always hold (proposition 7.3). Hence, it is of theoretical interest that there is a decidable sublanguage 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 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. Section 3 gives precise definitions of the languages we show equivalent to NC, over complex objects and flat relations respectively. We briefly review the parallel complexity classes AC^k and NC in section 4, and extend them to classes of queries in section 5. The main results are stated and discussed in section 6. Finally, we sketch the proofs of the main results in section 7.

2 Recursion on Sets

A fruitful approach to the design of query languages for complex objects is to consider tuples and sets as orthogonal [BTBN91, BBW92]. Hence, there will be primitives that work on tuples, primitives that work on sets, and general primitives for combining other primitives. In this section we discuss *four* forms of recursion on sets (two of which are *structural* recursions) and the relationship between them. One of these forms of recursion—*dcr*—will be at the core of our "query languages for NC". A precise definition of the languages will be given in section 3. However, we already need to refer in this section to some bits and pieces of the language for complex objects, notably to its type system, defined as follows.

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

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

where $\mathbb{D}$ is some base type, $\mathbb{B}$ is the type of booleans [3] and *unit* is the type containing only the empty tuple ($\mathit{unit} = \{()\}$). The values of type $s \times t$ are pairs (x, y) with $x \in s, y \in t$, and the values of type $\{t\}$ are finite sets of elements from t. Products of types of the form $\{s\}$, with s a product of base types ($\mathbb{D}, \mathbb{B}, \mathit{unit}$), are called **flat types**. We will also need to consider **function types** having the form $s \to t$, where s and t are complex object types [4].

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 (i-commutativity and i-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*:

$$\frac{e : t \quad f : s \to t \quad u : t \times t \to t}{\mathit{sru}(e, f, u) : \{s\} \to t}$$

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

$\mathit{sru}(e, f, u)$ is well-defined when there is some subset of t 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* ($x \upharpoonleft s$ is the element insertion operation, $\{x\} \cup s$):

$$\frac{e : t \quad i : s \times t \to t}{\mathit{sri}(e, i) : \{s\} \to t}$$

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

$\mathit{sri}(e, i)$ is well-defined when there is some subset of t containing e on which i is *i-commutative*, $i(x, i(y, s)) = i(y, i(x, s))$, and *i-idempotent* $i(x, i(x, s)) = i(x, s)$.

In addition to *sru* and *sri* we consider two other forms of recursion on sets. One of them is *dcr* whose

[3] Not really necessary, could have been encoded as $\{\mathit{unit}\}$ [BBW92].

[4] Note that we never make use of functions that take functions as arguments. Thus, as a *functional* language, our language is first-order. Of course, as a set-theoretic language, the language is higher-order since it manipulates complex objects.

definition was already given in section 1. dcr is superficially related to sru, for example $sru(e,f,u)$ 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, we do not know if sru can express parity or transitive closure. An interesting remark is that over ordered databases, sru together with transitive closure expresses dcr.

One can also define a fourth form of recursion on sets which is related to sri similarly to the way dcr is related to sru, let's call it *element-step recursion*, esr. This is like sri, with the second clause modified as:

$$esr(e,i)(y \uparrow s) \stackrel{\text{def}}{=} i(y, esr(e,i)(s)) \text{ when } y \notin s$$

where i is required to be i-commutative (but not necessarily i-idempotent). Obviously, esr can express sri [5]. The non-immediate relationships between the four forms of recursion on sets are contained in:

Proposition 2.1 *sri can express sru [BTS91]. Similarly esr can express dcr. Moreover, sri can express esr [BT92]. All these are done with at most polynomial overhead.*

Proof.

$$\begin{aligned} dcr(e,f,u) &\stackrel{\text{def}}{=} esr(e, \lambda(x,y).u(f(x),y)) \\ esr(e,i) &\stackrel{\text{def}}{=} \pi_2(sri((\emptyset,e), \lambda(x,(s,y)).\ \textit{if } x \in s \\ &\qquad \textit{then } (s,y) \textit{ else } (x \uparrow s, i(x,y)))) \end{aligned}$$

□

One can easily see that over complex objects dcr (and even sru) can express *powerset* hence we need some restriction if we are to stay within NC. An analog to Peter Buneman's idea of *bounded* fixpoints [Suc94] does the job. We define a **PS-type** (product of sets type) to be either a set type, or a product of PS-types. Then, the **bounded** version of dcr is defined by:

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

where t is a PS-type, with the semantics:

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

(here $(u \cap b)(y,y') \stackrel{\text{def}}{=} u(y,y') \cap b$, etc). As for dcr, we define the **bounded** sri, $bsri(e,i,b)$, for some PS-type t, to be $sri(e \cap b, i \cap b)$. Proposition 2.1 easily extends to the bounded versions of recursion. Over flat relations the explicit bounding is unnecessary:

Proposition 2.2 *bdcr together with the relational algebra can express dcr when its arguments are flat relations and its values are of flat PS-type. Similarly for bsri and sri.*

What is the role played by the algebraic conditions such as associativity etc., that are required for the constructs presented above that perform recursion on sets? Immerman, Patnaik, and Stemple [IPS91] consider under the name *set-reduce* a form of recursion on sets which resembles somewhat sri, but whose definition relies on the existence of a linear ordering. Essentially, a function f may be defined by: $f(\{x_1,\ldots,x_n\}) \stackrel{\text{def}}{=} i(x_1, f(\{x_2,\ldots,x_n\}))$, provided that $x_1 < x_2 < \ldots < x_n$ (no conditions are imposed on i): we can prove that, in the presence of order, this form of recursion has the same expressive power as sri. Similarly, one can conceive a form of divide and conquer recursion that relies on the ordering, which allows us to define some function by $f(\{x_1,\ldots,x_n\}) \stackrel{\text{def}}{=} u(f(\{x_1,\ldots,x_{\frac{n}{2}}\}), f(\{x_{\frac{n}{2}+1},\ldots,x_n\}))$ (no conditions are imposed on u): again, we can prove that this form of recursion has the same expressive power as dcr. In contrasts, 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 most of these identities is as hard as testing the validity of a first-order formula in all finite models, hence it is a Π^0_1-complete question. For example, consider $u(x,y) \stackrel{\text{def}}{=}$ *if* p *then* $u_1(x,y)$ *else* $u_2(x,y)$, where u_1 is some associative, commutative operation (e.g. $u_1(x,y) = x \cup y$), while u_2 is not (e.g. $u_2(x,y) = x - y$), and p is some arbitrary predicate (independent on x,y). Then u is associative, commutative iff p is true. (See also [Sar92] for forms of recursion on sets that are at least as powerful as Datalog and [BTS91] for structural recursion on lists and bags.)

3 Query Languages

In this section we define the languages that offer characterizations of NC over complex objects and over flat relations. For complex objects, the language is obtained by adding $bdcr$ and an order predicate to

[5] 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.

the *nested relational algebra* $\mathcal{NRA}$. This name can be used to describe a language that has the same expressive power as Schek and Scholl's NF^2 relational algebra [SS86], or Thomas and Fischer's algebra [TF86], or Paredaens and Van Gucht's *nested algebra* [PVG88, PVG92], or Abiteboul and Beeri's *algebra without powerset* [AB88]. The particular formalization we use here was essentially introduced in [BBW92] and is based on what is called there the **monad calculus**.

We assume an infinite set of variables to be given, each having a complex object type associated with it. We write x^s for a variable of type s. 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 following expressions, presented as part of their typing rules:

$$\frac{}{x^t : t} \qquad \frac{e_1 : t_1 \quad e_2 : t_2}{(e_1, e_2) : (t_1, t_2)}$$

$$\frac{e : t_1 \times t_2}{\pi_1(e) : t_1} \qquad \frac{e : t_1 \times t_2}{\pi_2(e) : t_2}$$

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

$$\frac{e_1 : \mathbb{D} \quad e_2 : \mathbb{D}}{e_1 = e_2 : \mathbb{B}} \qquad \frac{}{() : unit}$$

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

$$\frac{e : t}{\lambda x^s.e : s \to t} \qquad \frac{f : s \to t \quad e : s}{f(e) : t}$$

$$\frac{f : s \to \{t\}}{ext(f) : \{s\} \to \{t\}}$$

We briefly describe the semantics of the expressions: (e_1, e_2) constructs a tuple, π_1, π_2 are the projections, $\{e\}$ is the singleton set, $empty(e)$ returns true iff $e = \emptyset$, $if\ e\ then\ e_1\ else\ e_2$ equals e_1 iff $e = true$ and e_2 otherwise, $\lambda x^s.e$ denotes a function whose input is the variable x^s, $f(e)$ is function application, and $ext(f)(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} f(x_1) \cup \ldots \cup f(x_n)$. A possible set Σ of **external functions** $p : dom(p) \to codom(p)$ could be added to the language; in this case, we denote the language by $\mathcal{NRA}(\Sigma)$. We abbreviate with $\lambda(x, y).e$ the expression $\lambda z.e[\pi_1(z)/x, \pi_2(z)/y]$.

$\mathcal{NRA}$ is powerful enough to express the following functions: set difference, set intersection, cartesian product, database projections, equalities at all types, selections over predicates definable in the language, nest and unnest [BBW92]. E.g., equality at type t, $=_t : t \times t \to \mathbb{B}$, is defined by induction on the type t. When $t = \{t'\}$, we start by defining $member? : t' \times \{t'\} \to \{unit\}$, with the semantics $member?(x, s) = \{()\}$ when $x \in s$, and $member?(x, s) = \emptyset$ otherwise. Namely:

$$member?(x, s) \stackrel{\text{def}}{=} ext(\lambda y.if\ x = y\ then\ \{()\}\ else\ \emptyset)(s)$$

Using *member?* we can define set inclusion, and, finally, we can define the equality at type $\{t'\}$.

We will make two extensions to $\mathcal{NRA}$ in order to obtain a language that characterizes NC. One extension is the addition of divide-and-conquer recursion on sets. *dcr* is too powerful on complex objects since it can express *powerset* (actually one can show that $\mathcal{NRA}(dcr)$ has the same expressive power as Abiteboul and Beeri's *algebra* [AB88]). So instead, we will add the bounded version of this kind of recursion and consider the language $\mathcal{NRA}(bdcr)$. We shall have occasion to contrast this languages with $\mathcal{NRA}(bsri)$. Note that proposition 2.1 states that $\mathcal{NRA}(bdcr) \subseteq \mathcal{NRA}(bsri)$, and this holds even in the presence of external functions.

The second extension concerns *ordering*. All known characterizations of complexity classes below $PTIME$ are about ordered databases, and ours is no exception. We isolate the use of ordering into 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 $\mathbb{D}$. The order relation can be lifted to all types (e.g. see [LW94]).

We have thus reached the language $\mathcal{NRA}(bdcr, \leq)$ which is the subject of one of our main theorems, characterizing the NC-computable queries on ordered databases of complex objects (theorem 6.1).

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. For example, for *bdcr*: $depth(bdcr(e, f, u)) \stackrel{\text{def}}{=} \max(depth(e), depth(f), 1 + depth(u))$ (only u is actually iterated). We denote $\mathcal{NRA}(bdcr^{(k)}, \Sigma)$ the restrictions of the language $\mathcal{NRA}(bdcr, \Sigma)$ to iteration depth $\leq k$. We have thus obtained the hierarchy of languages $\mathcal{NRA}(bdcr^{(k)}, \leq)$ which is the subject of the finer characterization given in theorem 6.1.

To characterize the NC-computable queries over ordered flat relations (theorem 6.2), we denote by $\mathcal{NRA}^1$ 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. Its expressive power is essentially the same as that of the relational algebra [BTBN91], hence first-order logic. We make the same two extensions as for $\mathcal{NRA}$, except that here it is safe to use *dcr*. For good measure, proposition 2.2 states that it doesn't matter: $\mathcal{NRA}^1(bdcr) = \mathcal{NRA}^1(dcr)$. Note however that this fails in the presence of certain external functions. Again we will have occasion to contrast $\mathcal{NRA}^1(dcr)$ with $\mathcal{NRA}^1(sri)$ so note that proposition 2.1 states that $\mathcal{NRA}^1(dcr) \subseteq \mathcal{NRA}^1(sri)$, and this holds even in the presence of external functions. Therefore, for flat relations, the languages of interest are $\mathcal{NRA}^1(dcr, \leq)$ and the hierarchy $\mathcal{NRA}^1(dcr^{(k)}, \leq)$.

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 while 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 $ext(f)(\{x_1, \ldots, x_n\})$.

4 Complexity Classes

Let $F : \{0,1\}^* \to \{0,1\}^*$; 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. $\forall w \in \{0,1\}^*$, $\mid f(w) \mid = Q(\mid w \mid)$. Thus, F is the union of its restrictions to inputs of length n, $F_n : \{0,1\}^n \to \{0,1\}^{Q(n)}$. (2) There is a family of circuits α_n made up of input gates, NOT gates, unbounded fan-in AND and OR gates, s.t. α_n has n inputs, $Q(n)$ outputs, and computes F_n, for all $n \geq 0$. (3) $size(\alpha_n) \leq P(n)$ 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. 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 6.4 deals with the class AC^0 and it remains true for the more restrictive *FO-DCL*-uniformity condition in [BIS90].

The **direct connection language** *DCL* for a family α_n of circuits, is the set of quadruples (n, g, g', t), where g, g' are gate numbers in α_n, such that g is a child of g', and t is the type of g', $t \in \{NOT, AND, OR, y_1, \ldots, y_{Q(n)}\}$. 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.

Now *NC* is defined as $\bigcup_{k \geq 0} AC^k$. The results in Stockmeyer and Vishkin [SV84] imply that this class coincides with the class of functions computable by a *CRCW PRAM* (Concurrent Read Concurrent Write Parallel Random Access Machine) in polylogarithmic time using polynomially many processors.

5 Encodings of Complex Objects

Our encodings 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 which preserves the order relation $\leq$. Next, we encode complex objects using the eight symbols from the alphabet $A = \{0, 1, \{, \}, (,), comma, blank\}$, as follows: elements from $\mathbb{D}$ are encoded in binary, *true* and *false* are encoded by 1 and 0 respectively, () is encoded by (), a pair is encoded by (X_1, X_2), 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 representing each of the eight symbols in A with three bits.

Removing duplicates 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 it would have sufficed to encode with possible duplicates and no blanks, while for AC^k, $k \geq 1$, we could ask both for blanks and duplicate elimination. But our choice is uniform for all AC^k, $k \geq 0$.

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 encoded by a string of bits of length n^k. For flat relations, we can translate between the two encodings in AC^0.

Consider $\mathbb{D}$ and $\mathbb{D}'$ two different interpretations of the base type. A **morphism** is some function $\varphi : \mathbb{D} \to \mathbb{D}'$ with the property $x \leq y \iff \varphi(x) \leq \varphi(y)$ (so φ is injective); for any type t, φ extends to a function $\varphi_t : t \to t'$, where t' is obtained from t by substituting

$\mathbb{D}$ with $\mathbb{D}'$. Adapting the definition in [CH80], we define a **database query** of type $s \to t$ to be some functions $f_{\mathbb{D}} : s \to t$ (one for each interpretation of the base type $\mathbb{D}$) such that, for any morphism φ, $\varphi_t \circ f_{\mathbb{D}} = f_{\mathbb{D}'} \circ \varphi_s$. We say that some query f is in AC^k, or in NC, iff there is some function $F : \{0,1\}^* \to \{0,1\}^*$ such that, $\forall x \in s, \forall X \in \{0,1\}^*, x \sim X \Rightarrow f(x) \sim F(X)$. In order to compute f on an input $x \in s$, it suffices to choose some **minimal encoding** X of x, namely without blanks and in which the atomic values of x are encoded by $0, 1, \ldots, m-1$, next to compute $Y = F(X)$, and finally to decode Y.

We define *FLAT-NC* and *CMPX-OBJ-NC* to be the class of queries over base types and flat relations, and complex objects respectively, which are in NC. Similarly, we define the subclasses *FLAT-*AC^k and *CMPX-OBJ-*AC^k.

6 Main Results

We only state the results here and give the proofs in section 7.

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

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

These languages are purely for complex objects, respectively relations. 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.*

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 results to complex objects.

Proposition 6.4 *Under the encoding of complex objects described in subsection 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 extended abstract.

Conservative extension. One may wonder in what sense theorem 6.2 is a "particular case" of theorem 6.1. Actually, even though the proof of theorem 6.2 is quite similar to that of theorem 6.1, and we do present them "together" in section 7, theorem 6.2 in fact follows from theorem 6.1, proposition 2.2 and the conservative extension result presented below.

Paredaens and Van Gucht in [PVG92], 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 6.5 *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 6.5 follows directly from the main theorems. For the case when $\Sigma \neq 0$, this proposition requires a separate proof, and we are only able to do it 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 2). Extending their result also to complex objects we have:

Proposition 6.6 $\mathcal{NRA}^1(sri^{(1)}, \leq) =$ *PTIME [IPS91] and* $\mathcal{NRA}(bsri^{(1)}, \leq) =$ *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*.

7 Proofs

7.1 Iteration over sets

The central technical tool in proving our main result, is to convert the two forms of recursion over sets, into more simple *loops*. The **logarithmic** and the **bounded logarithmic iterator** are defined by:

$$\frac{f : t \to t}{log\_loop(f) : \{s\} \times t \to t}$$
$$\frac{f : t \to t \quad b : t}{blog\_loop(f, b) : \{s\} \times t \to t}$$

with the semantics:

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

where $|\ x\ |$ is the cardinal of x. The bounded logarithmic iterator is define by $blog\_loop(f, b)(x, y) \stackrel{\text{def}}{=} log\_loop(f \cap b)(x, y \cap b)$. Thus, *log_loop* iterates some function f a number of times equal to the number of bits necessary to represent the cardinality of a set x.

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

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 7.1 *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* $n \stackrel{\text{def}}{=} |\ v\ |$*, and* $\circ$ *is relation composition.*

Example 7.2 *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. To iterate* $\log^2 n$ *times, we use a depth two of iteration nesting.*

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. $\mathcal{NRA}^1(log\_loop, \leq, BIT)$ 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. Similar, we can argue that $FO(\log^{O(1)} n)$ 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:

Proposition 7.3 *Let* $f : s \to t$*, with* t *some PS-type. Then* $f \in \mathcal{NRA}^1(\Sigma, log\_loop^{(k)}, \leq) \iff f \in \mathcal{NRA}^1(\Sigma, dcr^{(k)}, \leq)$*, and* $f \in \mathcal{NRA}(\Sigma, blog\_loop^{(k)}, \leq) \iff f \in \mathcal{NRA}(\Sigma, bdcr^{(k)}, \leq)$*,* $\forall k \geq 0$*. A similar relationship holds between loop and sri.*

Proof. Consider some function $h = dcr(e, f, u)$, $h : \{s\} \to t$, and $x = \{a_1, \ldots, a_n\} \in \{s\}$ some input to it. The idea of simulating h with *log_loop* is to first apply f to each element in x, obtaining $y = \{f(a_1), \ldots, f(a_n)\} \in \{t\}$, and then to iterate $\log n$ times some function g on y, which, for some set set $y = \{b_1, \ldots, b_m\}$, is defined by:

$$g(y) \stackrel{\text{def}}{=} \{u(b_1, b_2), u(b_3, b_4), \ldots, u(b_{n-2}, b_{m-1}), u(b_m, e)\}$$

(assuming m is odd): the order relation on y is used in the definition of g, and some transitive closure is computed to identify the odd and even positions. Thus, the number m of elements in y is initially n, and is halved at each step. Eventually, the set y will contain only one element, which one can prove to be $h(x)$ (associativity and commutativity of u is used here). Since t is a PS-type, one can extract the unique element out of a singleton set. To compute *bdcr*, one proceeds similarly, but use *blog_loop* instead of *log_loop*. Only one problem remains: the type of y is $\{t\}$, which has a set height one larger than t. To circumvent that, we use the fact that t is a PS-type and "flatten" y; to distinguish elements belonging to different subsets, we tag them with elements from x.

Conversely, consider some $log\_loop(f)(x, y)$; we can express it by divide and conquer recursion on the set x, by noting that:

$$\begin{aligned} log\_loop(f)(\emptyset, y) &= y \\ log\_loop(f)(\{a\}, y) &= f(y) \end{aligned}$$

For $log\_loop(f)(x_1 \cup x_2, y)$, assume $|\ x_1\ | \leq |\ x_2\ |$ (the case $|\ x_1\ | > |\ x_2\ |$ is similar). Then

$$log\_loop(f)(x_1 \cup x_2, y) = \begin{cases} f(log\_loop(x_1, y)) \text{ or} \\ log\_loop(x_1, y) \end{cases}$$

Namely $log\_loop(f)(x_1 \cup x_2, y) = f(log\_loop(x_1, y))$ if $|\ x_1 \cup x_2\ |$ has one more bit than $|\ x_1\ |$, and $log\_loop(f)(x_1 \cup x_2, y) = log\_loop(x_1, y)$ otherwise. So we only have to argue that we can answer the question about the number of bits. The idea is to use the set x as a set of numbers $0, 1, \ldots, n-1$, and to compute the function $\varphi(x') = (|\ x'\ |, (2^{\lceil \log(|x'|+1) \rceil}, f^{(\lceil \log(|x'|+1) \rceil)}(y)))$, for all $x' \subseteq x$. I.e., we return the cardinality of x', the next power of 2, and $log\_loop(f)(x', y)$. Addition and comparison on the "numbers" in x (which can be done with transitive closure) suffices to compute h using dcr. To avoid increasing the iteration depth we precompute addition and comparison, as subsets of $x \times x \times x$ and $x \times x$ respectively, using a expressions of iteration depth 1; thus, an expression in $\mathcal{NRA}(\Sigma, \mathit{blog\_loop}^{(k)})$ will be translated into some $\mathcal{NRA}(\Sigma, dcr)$ expression with iteration depth $\max(k, 1) = k$. The "u" used in this dcr will be associative and commutative on some set of the form $\{(i, c_i) \mid i \leq \mid x' \mid\}$, because u is defined as $u((i, c_i), (j, c_j)) = (i + j, c_{i+j})$. □

The annoying condition for t to be a PS-type is due to the fact that the function $get : \{\mathbb{D}\} \times \mathbb{D} \to \mathbb{D}$ defined by $get(x, y) \stackrel{\text{def}}{=} \mathit{if}\ x = \{z\}\ \mathit{then}\ z\ \mathit{else}\ y$ is definable with dcr, but not with $\mathit{log\_loop}$. But $\mathit{log\_loop}$ together with get can indeed express dcr at all types t.

The proof of Proposition 7.3 has an important consequence. Recall that the conditions for well-definedness of dcr are Π^0_1-complete 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 $\mathit{log\_loop}$ we obtain an r.e., in fact decidable, sublanguage $\mathcal{L}$ which has the same expressive power as the whole $\mathcal{NRA}^1(dcr, \leq)$.

7.2 Circuits

In order to prove that $\mathcal{NRA}(\mathit{blog\_loop}) \subseteq AC$, we first establish some technical lemmas.

Lemma 7.4 *For each type t, there is some function $F = \bigcup F_n$ in AC^0, $F_n : \{0,1\}^n \to \{0,1\}^{n^2}$ which identifies the pairs of parenthesis for any encoding of type t.*

Proof. The nesting depth of parenthesis for some type t is bounded by some d_t, so identifying the pairs of parenthesis can be done by some circuit of depth $O(d_t)$. □

Lemma 7.5 *For any set type $\{t\}$, there is some function $F = \bigcup F_n$ in AC^0, $F_n : \{0,1\}^n \to \{0,1\}^n$, 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 pair types (s, t).*

Proof. The circuit computing F_n identifies the outermost commas (i.e. those not included in any pair of parenthesis, except the outermost { }), and returns a 1 on each first nonblank position following such a comma, or following the leading left brace. □

As a consequence, we have:

Lemma 7.6 *For all types t, equality of objects of type t is computable in AC^0.*

Proposition 7.7 *$\mathcal{NRA}(\mathit{blog\_loop}^{(k)}) \subseteq AC^k$ for all $k \geq 0$. Hence, $\mathcal{NRA}^1(\mathit{log\_loop}^{(k)}) \subseteq AC^k$.*

Proof. We prove by induction on some complex object expression $e \in \mathcal{NRA}(\mathit{blog\_loop}^{(k)})$ of type t with free variables $x_1^{s_1}, \ldots, x_l^{s_l}$, that the function $\lambda(x_1^{s_1}, \ldots, x_l^{s_l}).e : s_1 \times \ldots s_l \to t$ is in AC^k. Simultaneously, we prove also by induction that, for any function f of type $s \to t$ with free variables $x_1^{s_1}, \ldots, x_l^{s_l}$, $\lambda(x^s, x_1^{s_1}, \ldots, x_l^{s_l}).f(x^s) : s \times s_1 \times \ldots \times s_l \to t$ is in AC^k. We only illustrate some of the cases.

- $e \cup e'$ **(union)** Let α_n be the circuit for e and α'_n the circuit for e'. Concatenate their result, eliminate the braces } { replacing them with blanks and conditionally placing a comma (the comma is placed only when both e and e' yield a nonempty set). Finally, eliminate the duplicates in the resulting set, using lemmas 7.5 and 7.6, by replacing them with blanks.
- $\mathit{ext}(f)$ For simplicity suppose $f : s \to \{t\}$ doesn't have free variables, and let α_n be a circuit for computing f. The circuit for $\mathit{ext}(f)$ will consists of $\frac{\frac{n}{3}(\frac{n}{3}+1)}{2}$ copies of α_n (recall that three bits are used to encode one character), identified by pairs (i, j), $1 \leq i \leq j \leq \frac{n}{3}$, and whose outputs are concatenated. Circuit (i, j) will have as inputs the symbols from position $3i - 2$ to $3j$, and its output will be overridden (i.e. replaced by blanks) unless in the input $\{X_1, \ldots, X_m\}$ there is some X_l starting at position $3i - 2$ and ending on position $3j$ (which can be determined using lemma 7.5). Finally we concatenate their results and eliminate the duplicates.
- $f(e)$ Construct the circuit for e and direct its outputs into the circuit for f.
- $\mathit{blog\_loop}(f, b)$ For clarity, assume the type of f is $\{t\} \to \{t\}$. First we construct the circuit α_n for computing b: let its output have size $Q(n)$, where Q is some polynomial. Let $\{X_1, \ldots, X_m\}$ be the output of α_n. Clearly, $m \leq P(n)$, so it suffices to make $\log Q(n)$ copies of the circuit $\beta_{Q(n)}$ for f. Each such copy receives some input Y of size $Q(n)$, computes $f(Y)$, and intersects it with $\{X_1, \ldots, X_m\}$ (the output of b) such that the result Y' has the same size $Q(n)$, so it can be fed into the next level. Of course, we have to bypass all levels above level $\log m$. For this we compute m by counting the number of 1's produced by lemma 7.5 on $\{X_1, \ldots, X_m\}$ (this can be done in AC^1), compute $\log m$ (again in

AC^1) and bypass all circuits for f whose level is larger than $\log m$. Finally, observe that, if the circuit $\beta_{Q(n)}$ for computing f had depth $O(\log^k n)$, then the resulting circuit has depth $O(\log^{k+1} n)$.

We skip the proof of the uniformity, which is tedious but straightforward. □

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 6.4. But we chose to construct the circuit for computing f in order to suggest that how f may compiled on a CRCW PRAM.

Proposition 7.8 *Let $f : s \to t$ be s.t. t is a PS-type. Then $f \in$ FLAT-$AC^k \Rightarrow f \in \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$ and $f \in$ CMPX-OBJ-$AC^k \Rightarrow f \in \mathcal{NRA}(blog\_loop^{(k)}, \leq$), for $k \geq 1$.*

Proof. Let $F \in$ *CMPX-OBJ-AC^k*, of type $s \to t$. F is given by: (1) A *DLOGSPACE* Turing Machine T accepting the *DCL* of a family of circuits, (2) Polynomials $P(n)$ and $Q(n)$ (see section 4). For some input $x \in s$, 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. Construct from the input x some set z having a cardinality $\geq n$. The type of z will have a set height which is at most equal to the set height of s; thus z is in $\mathcal{NRA}^1$ when F is in *FLAT-AC^k*. We omit the technical details for computing z: see [GV91a, Suc94].

2. Some power 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 $0, \ldots, p-1$. E.g. to compute addition, we use transitive closure, a technique found in [Imm87b]. Everything in this step is in $\mathcal{NRA}(blog\_loop^{(1)}, \leq)$.

3. Compute the minimal encoding X of x, of length n, without blanks: $X \in \{0,1\}^*$ is represented as a set of "numbers". The computation is done in $\mathcal{NRA}^1(blog\_loop, \leq)$, the *blog_loop* being needed to compute the sum of a set of numbers.

4. Simulate F on X, as described below, to get $Y = F(X)$. Then "decode" Y, to get $y \in t$. Decoding is done in $\mathcal{NRA}$.

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. 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 to the computation of transitive closure of the successor relation on the set of configurations. Second, 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 $\mathcal{NRA}^1(log\_loop^{(k)}, \leq)$. Note that *ext* is used in an essential way at each iteration stept, accounting for the parallelism in the evaluation of α_n.

This proved

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

If s 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 $\mathcal{NRA}^1(log\_loop^{(k)}, \leq)$. Hence, *FLAT-$AC^k$* $\subseteq \mathcal{NRA}^1(log\_loop^{(k)}, \leq)$. □

Now we can prove our main results. Theorems 6.1 and 6.2 follow from propositions 7.7 and 7.8, and the cases when the target type t of $f : s \to t$ is not a PS-type are handled separatedly, essentially by observing that *dcr* can express *get* (see subsection 7.1). Proposition 6.4 follows from proposition 7.7, and proposition 6.3 is proven by an straightforward extension of the proof of proposition 7.7.

8 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.

Acknowledgements. 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.

[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.

[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.

[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.

[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.

[LW94] 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.

[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.

[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.

[PVG88] 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.

[PVG92] 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.

[Sar92] Yatin Saraiya. Fixpoints and optimizations in a language based on structural recursion on sets, December 1992. Manuscript available from `Yatin@Bellcore.Com`.

[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. 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*, pages 263–281. Springer-Verlag, January 1994.

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

[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.