INFORMATION AND COMPUTATION **87**, 241–263 (1990)

An Algebra and a Logic for NC^1

KEVIN J. COMPTON[*]

EECS Department, University of Michigan,
Ann Arbor, Michigan 48109

AND

CLAUDE LAFLAMME[†]

Department of Mathematics, University of Toronto,
Toronto, Ontario, Canada M5S 1A1

Presented here are an algebra and a logic characterizing the complexity class NC^1, which consists of functions computed by uniform families of polynomial size, log depth circuits. In both characterizations, NC^1 functions are regarded as functions from one class of finite relational structures to another. In the algebraic characterization a recursion scheme called *upward tree recursion* is applied to a class of simple functions. In the logical characterization, first-order logic is augmented by an operator for defining relations by primitive recursion where it is assumed that every structure has an underlying relation *BIT* giving the binary representations of integers. © 1990 Academic Press, Inc.

1. INTRODUCTION

Several recent papers have provided a new insights into the structure of the complexity class NC^1 (see Barrington (1986) and Barrington and Thérien (1987)), the class computed by a family of polynomial size, log depth circuits. We examine the structure of this class from a logical point of view by presenting an algebra and a logic characterizing it. This paper appeared in an earlier version as Compton and Laflamme (1988). At almost the same time, results similar to those in Section 3 of this paper appeared in Barrington, Immerman, and Straubing (1988). We will make precise the differences between their results and ours presently.

The original definition of NC^1 imposed a log space uniformity condition on the circuit family; that is, there is an algorithm generating the circuit for inputs of length n in space $O(\log n)$. However, later papers showed that

[*] Research partially supported by NSF Grant DCR 86-05358.

[†] Research partially supported by NSERC of Canada and FCAR of the Providence of Quebec.

0890-5401/90 $3.00

other uniformity conditions are more natural. Following Cook (1985), we adopt a uniformity condition known as *E^*-uniformity*, which is explained in Section 2.

Our algebra is in the spirit of the algebras Gurevich (1983) used to characterize *LOGSPACE* and *PTIME* sets. Gurevich characterized *LOGSPACE* by an algebra of functions on finite structures generated in the same way the primitive recursive functions are generated on the natural numbers. Similarly, we characterize NC^1 by an algebra $\mathscr{A}$ generated by applying a recursion scheme called *upward tree recursion* to a class of simple functions.

Our logic is in the spirit of the logics Immerman (1986) and Vardi (1982) used to characterize *PTIME*, and the logics Immerman (1987b) used to characterize other complexity classes. We show that first-order logic augmented by an operator for defining relations by *relational primitive recursion* (i.e., the scheme of primitive recursion restricted to relations) characterizes NC^1 provided that all structures have a relation *BIT* giving the binary representations of elements.

Algebras characterizing complexity classes date back to the beginnings of the complexity theory. Ritchie (1963) gave an algebra characterizing deterministic linear space and Cobham (1965) gave an algebra characterizing *PTIME*. These were algebras of functions over the natural numbers, unlike the algebra presented here. The algebra of primitive recursive functions is another example of a number theoretic algebra. Lind (1974) gave a number theoretic algebra characterizing *LOGSPACE*. Gurevich (1983) formulated a framework for characterizing complexity classes with algebras on finite structures. Earlier work of Fagin (1974), Immerman (1986, 1987b), Vardi (1982), and Lynch (1982) characterized various classes using logics on finite structures.

Other logical methods have been used to characterize complexity classes. Buss (1986) uses proof theory to characterize several classes. Clote (1989) uses Buss's framework to characterize *NC* (defined in the same way as NC^1 except that circuits may have depth $(\log n)^k$); this characterization draws on some ideas in an earlier unpublished number theoretic algebra for NC^1 (with *PTIME* and *LOGSPACE* uniformity conditions) due to Clote and Cook. Allen (1989) gives another algebra for *NC*.

Immerman (1987a) presents a logic similar in some respects to ours; it augments first-order logic with an operator for computing the transitive closures of width 5 graphs (where structures have the underlying relation *BIT*). The proof that this logic characterizes NC^1 can be found in Barrington, Immerman, and Straubing (1988). These authors remark that augmenting first-order logic by quantifiers for multiplying elements of some nonsolvable group such as S_5, rather than by an operator for computing transitive closures of width 5 graphs, obtains the same result.

Our logic instead incorporates the more natural operation of defining relations by primitive recursion. The relationship to logics capturing other complexity classes is clearer in this context. Least fixed-point logic, the logic Immerman (1986) and Vardi (1982) showed captures *PTIME* (assuming that structures have an underlying linear order), can be viewed as first-order logic together with an operator for defining relations inductively. From the results in Gurevich (1983), one can show that *LOGSPACE* is captured by first-order logic together with an operator for defining functions by primitive recursion. (An underlying linear order is not needed here—it can be defined using primitive recursion.) But primitive recursion is just a particular kind of inductive definition. We capture NC^1 by first-order logic together with an operator for defining *relations* by primitive recursion, but now it appears necessary to have *BIT* as an underlying relation.

Rather than considering the classes of structures satisfying fixed sentences in a logic (the usual method for capturing a complexity class) we interpret structures by means of a fixed sequence of formulas. This change in approach allows us to characterize complexity classes of functions rather than sets.

Why study algebras and logics for complexity classes?

One reason is that they give a better understanding of structure. For example, Barrington (1986) showed that the word problem for any finite nonsolvable group is complete for NC^1 via constant depth reductions. Evaluating a word over a finite group by multiplying elements in order is a paradigm of primitive recursion. Regarding Barrington's result, we may wonder just how much of the computational power of the constant depth reductions is needed. Our logic provides the answer: just enough to compute binary representations of integers and compute first-order connectives and quantifiers.

Another reason is that they provide an easy means to verify that problems are in a complexity class. This is perhaps more important for NC^1 because there is a messy uniformity condition to be verified. (For circuits of depth $O(\log^2 n)$ and higher, a simpler uniformity condition may be used; see Ruzzo (1981).) Very often it is ignored or glossed over. With our algebra or logic it is immediate.

We might hope that algebras and logics would provide the means to establish the complexities of problems not easily analyzed, although as yet there are no examples of this. Perhaps with NC^1 there will be. After all, constructing circuits or alternating Turing machines is like programming in a low level language, while algebras and logics are like high level languages.

This brings us to our last reason. Algebras and logics may provide the basis for programming language design. Our logic can be regarded as a purely sequential characterization of NC^1 since primitive recursion is usually evaluated sequentially. This means that it is easier to think about. Adding a richer selection of predefined functions to the logic may produce a language in which programs are easily understood, but also easily compiled into circuits or some other parallel architecture.

We thank the referee for helpful comments; in particular, the observation that downward tree recursion is a consequence of upward tree recursion (Lemma 3.1). The proof given there is a modified version of the proof supplied by the referee.

2. Background

In this section we review basic definitions and results. The following definition is one of several equivalent definitions found in the literature. It is the most convenient one for our purposes.

Definition. A log time bounded random access alternating Turing machines is given by a tuple

$$\langle Q_{\wedge}, Q_{\vee}, \sum, q_0, q_c, k, m, \delta_0, \delta_1\rangle$$

where $Q_{\wedge}$ and $Q_{\vee}$ are the universal and existential state sets, $\sum$ is the tape alphabet, $q_0 \in Q_{\wedge} \cup Q_{\vee}$ is the initial state, $q_c \notin Q_{\wedge} \cup Q_{\vee}$ is the check state, $m \log n$ is the time bound on inputs of length n, k is the number of work tapes, and δ_0 and δ_1 are the transition functions. M has $k+3$ tapes: an input tape with no head, and an index tape, a timer tape, and k work tapes, each with a read/write head. Each configuration of M has two possible successors given in the usual way by δ_0 and δ_1 (unless the state of the configuration is q_c, in which case there are no successors). A configuration whose state is q_c is *accepting* if and only if the content of the index tape is a, i, where a is a tape symbol, i is the binary representation of an integer, and the symbol at position i on the input tape is a. A configuration whose state is in $Q_{\wedge}$ is accepting if both of its successor configurations is accepting. A configuration whose state is in $Q_{\vee}$ is accepting if at least one of its successor configurations is accepting. At the beginning of the computation the head on the timer tape is at position $\lceil m \log n \rceil$. At each step of the computation the timer head moves one cell to the left. If the timer runs out, M enters q_c. We also assume that when M enters q_c, all tape heads are reset to the leftmost cell of each tape. A string is accepted if the initial configuration for that string is accepting.

We can define various complexity classes by means of random access alternating Turing machines. An alternate method is by circuits.

DEFINITION. A *circuit family* is a sequence $\alpha = \langle \alpha_1, \alpha_2, ... \rangle$ of fan-in 2 combinational circuits. An NC^1 function is a sequence $F = \langle F_1, F_2, ... \rangle$ of functions computed by a polynomial size, log depth circuit family α. Following Cook (1985), we also stipulate that α must be E^*-uniform, as defined below. (E^*-uniformity is a notion due to Ruzzo (1981) who called it U_{E^*}-uniformity.)

Fix a circuit family α and suppose that the gates in each circuit α_n are labeled with distinct integers. Further suppose that the two inputs of each AND-gate and OR-gate are labeled L and R. The *extended connection language* for α is the set of tuples (n, r, s, t, w), where r and s are bit sequences labeling gates in α_n, there is a direct chain of gates from gate r to gate s, the type (AND, OR, NOT) of gate r is t, and w is a string over the alphabet $\{L, R\}$ describing the input sequence along the path from r to s. Then a log depth circuit family α is E^*-uniform if membership of a tuple (n, r, s, t, w) in the extended connection language can be determined by a random access alternating Turing machine in time $O(\log n)$.

For the moment, think of NC^1 functions as mappings on $\{0, 1\}^*$. We usually distinguish between NC^1 functions, for which the associated circuits may have multiple outputs, and NC^1 sets, for which each circuit has a single output indicating acceptance or rejection. The two notions are easily related by defining for each NC^1 function F a set A_F consisting of all pairs of bit sequences (u, v) satisfying the following: $|v| = |F(u)|$; v has precisely one occurrence of 1, say in the ith position; and the ith bit of $F(u)$ is 1. Then it is easy to verify that F is an NC^1 function if and only if A_F is an NC^1 set.

The proof that our algebra $\mathscr{A}$ characterizes NC^1 depends on a theorem of Ruzzo showing that NC^1 sets are precisely those accepted by random access alternating Turing machines in time $O(\log n)$. In view of the discussion above we can reformulate Ruzzo's result as follows.

THEOREM 2.1. *A function F is in NC^1 if and only if A_F is recognized by a log time bounded alternating random access Turing machine.*

Now we review the framework for viewing functions as mappings between relational structures rather than strings.

DEFINITION. A *relational similarity type* σ is a set of relation symbols with assigned arities. (We assume throughout that σ is finite.) A σ-structure consists of a set (called the *domain*) together with relations of the appropriate arities corresponding to each symbol of σ.

Let $S(\sigma, n)$ denote the set of σ-structures with domain $n = \{0, 1, ..., n-1\}$. A relation of arity k on n is represented in the usual way by a bit sequence of length n^k. We identify each σ-structure in $S(\sigma, n)$ with the concatenation of bit sequences representing its relations.

Fix relational similarity types σ and τ. A *structural function* from σ-structures to τ-structures (written $F\colon \sigma \to \tau$) is a sequence $F = \langle F_1, F_2, ...\rangle$, where $F_n\colon S(\sigma, n) \to S(\tau, n)$.

Structural functions occur often in model theory, for example when one class of structures is interpreted in another.

Henceforth we will view NC^1 functions as structural functions computed by E^*-uniform circuit families. This point of view is not really a restriction since input and output strings can always be represented as structures, but it also seems to correspond more closely to the way problems are formulated for circuit complexity classes. For example, we would naturally represent n-bit integer addition as a structural function $F\colon \sigma \to \tau$ where the similarity type σ contains two unary relation symbols and τ contains a single unary relation symbol. To represent the highest carry bit in the sum, add a 0-ary relation symbol to τ. For Boolean matrix multiplication, σ would contain two binary relation symbols and τ would contain a single binary relation symbol.

DEFINITION. A *global relation* of arity k for σ is a function which, applied to a structure of $S(\sigma, n)$, yields a relation of arity k on n. A *global function* of arity k and coarity l is a function which, applied to a structure of $S(\sigma, n)$, yields a function from n^k to n^l. To avoid confusion, global functions will always be denoted by lower case letters and structural functions will be denoted by upper case letters. Assuming $n \geqslant 2$ we can associate with each global relation of arity k for σ its *characteristic global function* of arity k and coarity 1 in the obvious manner.

This is the same approach taken in the standard Tarski semantics for first-order logic: a formula defines a global relation, which is to say, it yields a relation on each structure.

DEFINITION. For a structural function $F\colon \sigma \to \tau$, we define χ_F, the *characteristic global function* of F, as follows. If F applied to a σ-structure $\langle n, S_1, ..., S_k\rangle$ yields a τ-structure $\langle n, T_1, ..., T_l\rangle$, then χ_F applied to $\langle n, S_1, ..., S_k\rangle$ yields a function $\chi_1 \times \cdots \times \chi_l$, where χ_i is the characteristic function of relation T_i. (When it is convenient we denote a relation symbol and the relation it interprets in a particular structure by the same symbol, for example symbols S_i and T_i above.)

3. AN ALGEBRA FOR NC^1

In this section we present an algebra $\mathscr{A}$ of global functions on σ-structures.

Definition. The class of *simple functions* is the smallest class of global functions containing the functions listed below (for each $k > 0$) and closed under composition. In each case $\mathbf{x}$ denotes a sequence $x_1, ..., x_k$ of elements from n. Wherever $\mathbf{y}$ and $\mathbf{z}$ appear, they also denote length k sequences of elements from n. We regard $\mathbf{x}$ as an n-ary representation of an integer less than n^k. The constant $\mathbf{0}$ is the length k sequence representing 0. K is the largest power of 2 not exceeding n^k.

(S1) $\text{Zero}_k(\mathbf{x}) = \mathbf{0}$, identically.

(S2) $\text{Ident}_k(\mathbf{x}) = \mathbf{x}$.

(S3) $$\text{Cond}_k(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \begin{cases} \mathbf{y}, & \text{if } \mathbf{x} \text{ is odd;} \\ \mathbf{z}, & \text{if } \mathbf{x} \text{ is even.} \end{cases}$$

(S4) $\text{Half}_k(\mathbf{x}) = \lfloor \mathbf{x}/2 \rfloor$.

(S5) $$\text{Double}_k(\mathbf{x}) = \begin{cases} 2\mathbf{x}, & \text{if } \mathbf{x} < K/2; \\ \mathbf{x}, & \text{otherwise} \end{cases}.$$

(S6) $$\text{Toggle}_k(\mathbf{x}) = \begin{cases} \mathbf{x}+1, & \text{if } \mathbf{x} \text{ is even and } \mathbf{x} < K; \\ \mathbf{x}-1, & \text{if } \mathbf{x} \text{ is odd and } \mathbf{x} < K; \\ \mathbf{x}, & \text{otherwise.} \end{cases}$$

Composition is defined in the usual way: if $h_1, ..., h_m$ are simple functions with coarities summing to the arity of a simple function g, then

$$f(\mathbf{x}) = g(h_1(\mathbf{x}), ..., h_m(\mathbf{x}))$$

is a simple function.

Definition. $\mathscr{A}$ is the smallest class containing the simple functions (for all integers k), the characteristic global functions of each global relation denoted by a symbol in σ, the functions

$$\text{Equal}_k(\mathbf{x}, \mathbf{y}) = \begin{cases} \mathbf{0}, & \text{if } \mathbf{x} \neq \mathbf{y}; \\ \mathbf{1}, & \text{if } \mathbf{x} = \mathbf{y}, \end{cases}$$

for each $k > 0$, and closed under composition, projection, and the scheme of upward tree recursion, given below.

We first describe a restricted form upward tree recursion, then a restricted version of the related scheme of downward tree recursion. We then describe how to generalize these schemes. In these recursion schemes the length of $\mathbf{y}$ is k, K is the largest power of 2 not exceeding n^k, and f, h, h_0, and h_1 have the same the coarity.

(T1) *Upward Tree Recursion* (UTR). If $g(\mathbf{x}, \mathbf{y})$ is in $\mathscr{A}$ and h is a simple function, then $\mathscr{A}$ also contains the function $f(\mathbf{x}, \mathbf{y})$ defined as follows:

$$f(\mathbf{x}, \mathbf{y}) = \begin{cases} g(\mathbf{x}, \mathbf{y}), & \text{if } \mathbf{y} \geqslant K/2 \text{ or } \mathbf{y} = \mathbf{0}; \\ h(\mathbf{x}, \mathbf{y}, f(\mathbf{x}, 2\mathbf{y}), f(\mathbf{x}, 2\mathbf{y}+1)), & \text{otherwise.} \end{cases}$$

(T2) *Downward Tree Recursion* (DTR). If $g(\mathbf{x})$ is in $\mathscr{A}$ and h_0, h_1 are simple functions, then $\mathscr{A}$ also contains the function $f(\mathbf{x}, \mathbf{y})$ defined as follows:

$$f(\mathbf{x}, \mathbf{y}) = \begin{cases} g(\mathbf{x}), & \text{if } \mathbf{y} = \mathbf{0} \text{ or } \mathbf{y} \geqslant K; \\ h_0(\mathbf{x}, \mathbf{y}, f(\mathbf{x}, \lfloor \mathbf{y}/2 \rfloor)), & \text{if } \mathbf{y} \text{ is even and } \mathbf{0} < \mathbf{y} < K; \\ h_1(\mathbf{x}, \mathbf{y}, f(\mathbf{x}, \lfloor \mathbf{y}/2 \rfloor)), & \text{if } \mathbf{y} \text{ is odd and } \mathbf{0} < \mathbf{y} < K. \end{cases}$$

The tree for these recursion schemes is pictured in Fig. 1. Each node **a** in the tree has left child 2**a** and right child 2**a** + 1. Thus, the node **0** has only a right child so for UTR $\mathbf{y} = \mathbf{0}$ must be treated as a special case. Also, to ensure that we have a full binary tree, we do not really use the nodes numbered K or higher. This simplifies our treatment, but is probably not necessary.

In the general forms of the schemes simultaneous recursions are allowed.

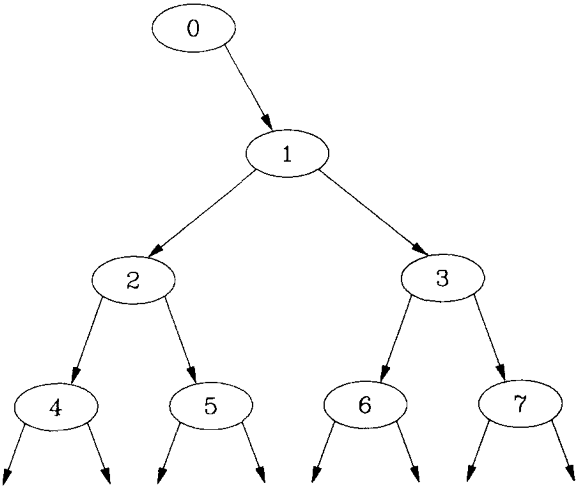

FIG. 1. Tree used in Recursion Schemes.

Thus, for UTR, functions $f_1, \dots, f_m$ can be defined simultaneously from $g_1, \dots, g_m$, $h_1, \dots, h_m$, and $h_{1,1}, \dots, h_{1,m}$; set $f_i(\mathbf{x}, \mathbf{y})$ equal to

$$\begin{array}{ll} g_i(\mathbf{x}, \mathbf{y}), & \text{if} \quad \mathbf{y} \geqslant K/2 \text{ or } \mathbf{y} = \mathbf{0}; \\ h(\mathbf{x}, \mathbf{y}, f_1(\mathbf{x}, 2\mathbf{y}), \dots, f_m(\mathbf{x}, 2\mathbf{y}), f_1(\mathbf{x}, 2\mathbf{y}+1), \dots, f_m(\mathbf{x}, 2\mathbf{y}+1)), & \text{otherwise.} \end{array}$$

The general form of DTR is defined similarly.

A form of downward tree recursion was used by Bennett (1962) in his investigations of the spectrum problem; he called his scheme *recursion on notation.*

The reason that downward tree recursion was not included in our definition of $\mathscr{A}$ is that it can be proved from upward tree recursion in our system. We thank the referee for pointing this out to us and supplying a proof, which appears in modified form below.

LEMMA 3.1. *$\mathscr{A}$ is closed under DTR.*

Proof. First note that the usual Boolean operations on $\{\mathbf{0}, \mathbf{1}\}$ can be defined as simple functions. $\text{Toggle}_k(\mathbf{x})$ is negation and $\text{Cond}_k(\mathbf{x}, \mathbf{y}, \mathbf{0})$ is conjunction. (We can assume that we have constants $\mathbf{0}$ and $\mathbf{1}$ at our disposal, using the functions $\text{Zero}_k(\mathbf{x})$ and $\text{Toggle}_k(\mathbf{x})$.) The function $\text{Cond}_k(\mathbf{x}, \mathbf{0}, \mathbf{0})$ is $\mathbf{1}$ if $\mathbf{x}$ is even, and $\mathbf{0}$ otherwise. We can also make use of Cond_k to define functions be cases below.

Suppose that $f(\mathbf{x}, \mathbf{y})$ is defined from g, h_0, and h_1 as in the definition of DTR. Use simultaneous UTR to define functions Reverse_k, Branch_k, and $\hat{f}$:

$$\text{Branch}_k(\mathbf{z}, \mathbf{y}) = \begin{cases} \text{Equal}_k(\mathbf{z}, \mathbf{y}), & \text{if} \quad \mathbf{y} \geqslant K/2 \text{ or } \mathbf{y} = \mathbf{0}; \\ \mathbf{1}, & \text{if} \quad \text{either } \text{Branch}_k(\mathbf{z}, 2\mathbf{y}) \text{ or} \\ & \text{Branch}_k(\mathbf{z}, 2\mathbf{y}+1) \text{ is odd;} \\ \mathbf{0}, & \text{otherwise;} \end{cases}$$

$$\text{Reverse}_k(\mathbf{z}, \mathbf{y}) = \begin{cases} \text{Equal}_k(\mathbf{z}, \mathbf{y}), & \text{if} \quad \mathbf{y} \geqslant K/2 \text{ or } \mathbf{y} = \mathbf{0}; \\ 2 \cdot \text{Reverse}_k(\mathbf{z}, 2\mathbf{y}), & \text{if} \quad \text{Branch}_k(\mathbf{z}, 2\mathbf{y}+1) \text{ is} \\ & \qquad \text{even and } \mathbf{0} < \mathbf{y} < K/2; \\ 2 \cdot \text{Reverse}_k(\mathbf{z}, 2\mathbf{y}+1)+1, & \text{if} \quad \text{Branch}_k(\mathbf{z}, 2\mathbf{y}+1) \\ & \qquad \text{is odd and } \mathbf{0} < \mathbf{y} < K/2; \end{cases}$$

$$\hat{f}(\mathbf{z}, \mathbf{x}, \mathbf{y}) = \begin{cases} h_1(\mathbf{x}, \mathbf{y}, g(\mathbf{x})), & \text{if} \quad \mathbf{y} \geqslant K/2 \text{ or } \mathbf{y} = \mathbf{0}; \\ h_0(\mathbf{x}, \mathbf{y}, \hat{f}(\mathbf{z}, \mathbf{x}, 2\mathbf{y})), & \text{if} \quad \text{Branch}_k(\mathbf{z}, 2\mathbf{y}+1) \text{ is} \\ & \qquad \text{even and } \mathbf{0} < \mathbf{y} < K/2; \\ h_1(\mathbf{x}, \mathbf{y}, \hat{f}(\mathbf{z}, \mathbf{x}, 2\mathbf{y}+1)), & \text{if} \quad \text{Branch}_k(\mathbf{z}, 2\mathbf{y}+1) \\ & \qquad \text{is odd and } \mathbf{0} < \mathbf{y} < K/2. \end{cases}$$

Suppose $K/2 \leqslant \mathbf{z} < K$; in other words, $\mathbf{z}$ is a leaf in the recursion tree. Suppose also that $\mathbf{z}$ is $1a_j a_{j-1} \cdots a_0$ in binary notation and $\mathbf{y}$ is a prefix of $\mathbf{z}$; i.e., $\mathbf{y}$ is $1a_j a_{j-1} \cdots a_i$ in binary notation. Then $\text{Branch}_k(\mathbf{z}, \mathbf{y}) = 1$ and $\text{Reverse}_k(\mathbf{z}, \mathbf{y})$ is equal to $1a_0 a_1 \cdots a_{i-1}$. In particular, $\text{Reverse}_k(\mathbf{z}, \mathbf{1})$ is equal to $1a_0 a_1 \cdots a_j$. Consider the tree in Fig. 1. The binary representation of a number describes a path to that number from the root: branch left for 0 and right for 1. Reverse_k allows us to reverse the binary representation (after the initial 1).

Thus, a recursion from $\mathbf{1}$ down the tree to $\mathbf{y}$ can be simulated by a recursion from $\text{Reverse}_k(\mathbf{z}, \mathbf{1})$ up the tree to $\text{Reverse}(\mathbf{z}, \mathbf{y})$. This is precisely what happens in the definition of $\hat{f}$. By induction on the depth of $\mathbf{y}$,

$$f(\mathbf{x}, \mathbf{y}) = \hat{f}(\text{Reverse}_k(\mathbf{z}, 1), \mathbf{x}, \text{Reverse}_k(z, \mathbf{y}))$$

whenever $K/2 \leqslant \mathbf{z} < K$ and $\mathbf{y} > \mathbf{0}$ is a prefix of $\mathbf{z}$.

Define

$$\text{Leaf}_k(\mathbf{y}, \mathbf{z}) = \begin{cases} \mathbf{y}, & \text{if } \ \mathbf{z} \geqslant K/2 \text{ or } \mathbf{z} = \mathbf{0}; \\ 2 \cdot \text{Leaf}_k(2\mathbf{z}) & \text{otherwise} \end{cases}$$

so that the function

$$\hat{f}(\text{Reverse}_k(\text{Leaf}_k(\mathbf{y}, \mathbf{1}), \mathbf{1}), \mathbf{x}, \text{Reverse}_k(\text{Leaf}_k(\mathbf{y}, \mathbf{1}), \mathbf{1}))$$

is equal to $f(\mathbf{x}, \mathbf{y})$ when $\mathbf{0} < \mathbf{y} < K$. If we can assign the correct value $f(\mathbf{x}, \mathbf{y}) = g(\mathbf{x})$ to the function when $\mathbf{y} = \mathbf{0}$ and $\mathbf{y} \geqslant K$ we are done. But this is easy to do since we have Boolean functions and $\mathbf{y} \geqslant K$ precisely when $\text{Leaf}_k(\text{Half}_k(\mathbf{y})) = \text{Half}_k(\mathbf{y})$. ▌

The main result of this section, Theorem 3.3 below, states that a necessary and sufficient condition for a structural function to be in NC^1 is that its characteristic global function be in $\mathscr{A}$. Sufficiency is a consequence of Lemma 3.2. The circuits constructed in the proof of this lemma must be regular enough that E^*-uniformity can be easily verified. The following definition is used in the proof.

DEFINITION. Let $\sigma = \{R_1, ..., R_j\}$ be a relational similarity type and f be a global function on σ-structures of arity k and coarity l. We will say that a circuit family $\langle \alpha_1, \alpha_2, ... \rangle$ *implements* f if the following conditions hold. Each α_n has inputs for σ-structures $\langle n, R_1, ..., R_j \rangle$; also, there are $\lceil k \log n \rceil$ inputs for the binary representation of an argument $\mathbf{x}$ for f and $\lceil l \log n \rceil$ outputs for the binary representation of $f(\mathbf{x})$ (evaluated on $\langle n, R_1, ..., R_j \rangle$).

LEMMA 3.2. *Every global function in* $\mathscr{A}$ *is implemented by an* E^**-uniform, polynomial size, log depth circuit family.*

Proof. Observe first that each simple function is implemented by an E^*-uniform, polynomial size, constant depth circuit family.

An E^*-uniform, polynomial size, log depth circuit family can be implemented for the composition of functions if each of the functions composing it is implemented by such a family. Also, projection (or more strictly, the collection of projection functions) and the global characteristic functions of global relations denoted by symbols in σ can be implemented by E^*-uniform, polynomial size, log depth circuit families. Notice that these assertions are not completely trivial since the value of a sequence of variables $\mathbf{x}$ is assigned by regarding it as an n-ary representation of an integer, whereas its circuit implementation uses its binary representation.

We now have only to show that if the functions used in an instance UTR are implemented by E^*-uniform, polynomial size, log depth circuit families, then so is the function resulting from the recursion. For simplicity consider the restricted (nonsimultaneous) form of the UTR scheme. The argument we give works just as well for the simultaneous form. Suppose in UTR scheme that g is implemented by an E^*-uniform, polynomial size, log depth circuit family and h is implemented by an E^*-uniform, polynomial size, constant depth circuit family. Construction of a circuit for $f(\mathbf{x}, \mathbf{y})$ is straightforward. In the recursion tree (Fig. 1) replace every node $\mathbf{a}$ that is a leaf node or the $\mathbf{0}$ node with a circuit to compute $g(\mathbf{x}, \mathbf{a})$. Replace every other node $\mathbf{a}$ with a circuit to compute $h(\mathbf{x}, \mathrm{y}, *, *)$, where the last two inputs become the outputs of the children of $\mathbf{a}$. Thus, the output of the circuit at the node $\mathbf{a}$ is $f(\mathbf{x}, \mathbf{a})$. It is a simple matter to add to the circuit a selector which for a given input $\mathbf{y}$ outputs $f(\mathbf{x}, \mathbf{y})$.

E^*-uniformity is not difficult to check. In fact, it is easy to see that membership of a tuple (n, r, s, t, w) in the extended connection language for the circuit can be determined by a random access *deterministic* Turing machine in time $O(\log n)$. This observation was made by Ruzzo (1981) in passing and later stated explicitly by Barrington, Immerman, and Straubing (1988). ∎

We now state the main result of this section.

THEOREM 3.3. *A structural function F is in NC^1 if and only if χ_F is in $\mathscr{A}$.*

Proof. Suppose $F\colon \sigma \to \tau$ is in NC^1. We show that χ_F is in our algebra $\mathscr{A}$. Suppose that τ consists of l relation symbols. Thus, when F is applied to a σ-structure it yields a τ-structure $\langle n, T_1, ..., T_l\rangle$. When χ_F is applied to the same structure it yields a function $\chi_1 \times \cdots \times \chi_l$, where χ_i is the characteristic function of T_i. We will show for each i that the global function which yields χ_i on σ-structures is in $\mathscr{A}$. The product $\chi_1 \times \cdots \times \chi_l$ is obtained by a slightly more elaborate version of the same argument. Therefore, we will assume that τ contains just one relation symbol T.

By Theorem 2.1, A_F is accepted by an alternating random access Turing machine M in time $m \log n$ for some constant m. We view $(u, v) \in A_F$ as a structure in $S(\sigma \cup \{T'\}, n)$ where T' is a relation symbol of the same arity as T, and $T'(\mathbf{x})$ holds if and only if $\mathbf{x}$ represents the position of the 1 bit in v. We modify M so that rather than writing a symbol and input tape position on the index tape, it writes a 0 or 1, a k-ary relation symbol from $\sigma \cup \{T'\}$, and a k-tuple of integers. Rather than checking that a symbol is at a given position on the input tape, M checks that the relation evaluated at the given tuple has the truth value specified. This change in the index tape format does not change the running time of M in any significant way.

We code each configuration of M as follows. Suppose the work tape and index tapes are numbered $1, ..., k$. The contents of the ith tape are encoded by strings $\mathbf{x}_i$ and $\mathbf{y}_i$, where $\mathbf{x}_i$ is a bit sequence consisting of a 1 followed by an encoding of the contents to the left of the head, rightmost tape cells corresponding to least significant bits of $\mathbf{x}_i$, and $\mathbf{y}_i$ encodes the contents to the right and under the head, leftmost tape cells corresponding to least significant bits of $\mathbf{y}_i$. (The leading 1 in $\mathbf{x}_i$ is needed to keep track of the number of cells to the left of the head.) For the contents of the timer tape we need only represent the contents to the left of the head. Code states in any reasonable way. We need to show that changes of configurations effected by the transition functions δ_0 and δ_1 can be represented by simultaneous simple functions.

Changing a configuration according to δ_0 or δ_1 entails determining the state and the symbols scanned on each tape by using Cond_k and Half_k to obtain the bits in their representation; applying appropriate Boolean functions to obtain bits of the new state, the symbols written and the head directions; and using Half_k, Double_k, and Toggle_k to modify the configuration accordingly. Hence, there are simple functions $h_{0,i}$ and $h_{1,i}$ simulating δ_0 and δ_1 on the strings that represent configurations. The entire computation tree of M can then be obtained using DTR on a tree of depth $m \log n$. Also computed, besides configurations, are integers coding the sequences of state types (i.e., whether a state is in $Q_\wedge$ or $Q_\vee$) along branches of the computation tree from the root to a given node. This can easily be made part of the DTR.

At the leaves, M must check that a particular relation S, evaluated at a given tuple of elements, has a specified truth value. We use the characteristic function χ_S and projection to do this.

Use UTR then to determine whether M accepts by passing acceptance or rejection bits up the tree. At each node the conjunction or disjunction of the bits passed to it will be taken according to whether the state for the corresponding configuration is in $Q_\wedge$ or $Q_\vee$. To know which to take, the sequences of state types collected down the branches during the DTR are passed back up the tree.

This gives a global function f which, applied to a structure in $S(\sigma \cup \{T'\}, n)$ representing an input (u, v), yields a 0-ary characteristic function; this function takes the value $\mathbf{1}$ if and only if $F(u)$ has a 1 bit at the position where the 1 occurs in v. If we replace each occurrence of $\chi_{T'}(\mathbf{y})$ in the description of f by $\text{Equal}_k(\mathbf{x}, \mathbf{y})$, we obtain the global function χ_F. Since the $\chi_{T'}(\mathbf{y})$ is evaluated only at the leaves of the Turing machine computation tree, $\text{Equal}_k(\mathbf{x}, \mathbf{y})$ will never be used in a recursion. ▌

4. A Logic for NC^1

In this section we present a logic for NC^1. The boundary between logics and algebras (such as the one in the previous section) is vague. We could present our logic as an algebra in the same way first-order logic is presented as a relational algebra in database theory (see Codd (1972)). However, it seems more natural to use a logical form here.

Formulas of a logic naturally give rise to structural functions. If σ and τ are relational similarity types with τ consisting of symbols $T_1, ..., T_l$, then formulas $\varphi_1(\mathbf{x}_1), ..., \varphi_l(\mathbf{x}_l)$ over σ, where the length of $\mathbf{x}_i$ is the arity of T_i, define a structural function F: $\sigma \rightarrow \tau$. Form the image of a σ-structure $\mathcal{M}$ under F by taking $\{\mathbf{a} \mid \mathcal{M} \models \varphi_i(\mathbf{a})\}$ as the interpretation of each T_i.

To motivate the choices made in the design of our logic, let us consider the characterization of the *LOGSPACE* functions in Gurevich (1983). He showed that the algebra of global functions generated by the basic functions and operations that define the primitive recursive functions on the natural numbers—viz., the zero, projection, and successor functions with the operations of composition and primitive recursion—characterizes *LOGSPACE* in the same way our algebra $\mathcal{A}$ characterizes NC^1. We suspect that *LOGSPACE* properly contains NC^1 so we ask which of these functions and operations would appear to take us out of NC^1. The only possibility is primitive recursion. How much of primitive recursion, then, can be retained if we wish to stay inside NC^1? It is not difficult to see that if we restrict the primitive recursion to functions of bounded range, we stay within NC^1. Rather than making awkward stipulations, about ranges of functions, we instead restrict ourselves to relations, which are, after all, functions with ranges contained in $\{0, 1\}$.

We define an extension of first-order logic (with equality) in which we may define new relations by *relational primitive recursion*, abbreviated RPR. We assume that one of the relation symbols in the vocabularly of the logic denotes a successor relation on each structure.

Formulas in this logic may contain any of the symbols occurring in first-order formulas and possibly some *relation variables*. We define formulas φ of this logic inductively, and at the same time define free(φ), the set of free variables in φ. An *atomic formula* φ is either a first-order atomic formula, or a formula $P(x_1, ..., x_j)$, where P is a relation variable of arity j; in the former free(φ) is the same as in first-order logic; in the latter, free(φ) = $\{P, x_1, ..., x_j\}$. More complex formulas φ may be constructed using the standard logical connectives and first-order quantifiers. In these cases free(φ) is defined just as in first-order logic.

The only other way to construct more complex formulas is by RPR. We first describe the simplest form of RPR, and then the general form.

Suppose P is a relation variable of arity j, $\mathbf{x}$ is a sequence of $j-1$ distinct variables, and y is a variable not in $\mathbf{x}$. Let ψ and $\theta(\mathbf{x}, y, P(\mathbf{x}, y))$ be formulas. In writing θ this way we mean that every subformula of θ containing P is of the form $P(\mathbf{x}, y)$ and that the variables $\mathbf{x}$, y occurring in these subformulas are free; $\mathbf{x}$ and y in the first two argument places of $\theta(\mathbf{x}, y, P(\mathbf{x}, y))$ stand for all free occurrences of $\mathbf{x}$ and y not in subformulas of the form $P(\mathbf{x}, y)$. Then φ given by

$$[P(\mathbf{x}, y) \equiv \theta(\mathbf{x}, y, P(\mathbf{x}, y-1))]\psi$$

is also a formula with

$$\text{free}(\varphi) = ((\text{free}(\theta) - \{\mathbf{x}, y\}) \cup \text{free}(\psi)) - \{P\}.$$

The part of φ within brackets defines the interpretation of P in ψ. That is, all free occurrences of P in ψ are interpreted by a relation defined as follows. $P(\mathbf{x}, 0)$ has the same truth value as $\theta(x, 0, \rho)$, where ρ is an invalid sentence; $P(\mathbf{x}, y)$ has the same truth value as $\theta(\mathbf{x}, y, P(\mathbf{x}, y-1))$ when $y > 0$. Thus, θ codes the relations used in both the basis and recursion parts of the usual primitive recursion scheme. It is useful to think of $P(\mathbf{x}, 0)$ as being obtained by taking $y = 0$ in the definition

$$[P(\mathbf{x}, y) \equiv \theta(\mathbf{x}, y, P(\mathbf{x}, y-1))]$$

and to observe the convention that $P(\mathbf{x}, -1)$ is always false. A *sentence* is a formula φ with free(φ) = $\varnothing$.

In the general form of RPR simultaneous definitions of relations are allowed. That is, we may have formulas of the form

$$\begin{bmatrix} P_1(\mathbf{x}_1, y) \equiv \theta_1 \\ P_2(\mathbf{x}_2, y) \equiv \theta_2 \\ \vdots \qquad \vdots \\ P_k(\mathbf{x}_k, y) \equiv \theta_k \end{bmatrix} \psi,$$

where θ_i is of the form

$$\theta_i(\mathbf{x}_i, y, P_1(\mathbf{x}_1, y-1), ..., P_k(\mathbf{x}_k, y-1)).$$

The intended meaning should be apparent: the values of $P_1(\mathbf{x}_1, y), ..., P_k(\mathbf{x}_k, y)$ are simultaneously determined by the values of $P_1(\mathbf{x}_1, y-1), ..., P_k(\mathbf{x}_k, y-1)$.

Denote by $FO+RPR$ the set of structural functions given by formulas of this logic. (We will refer to the logic itself as $FO+RPR$, also.) Let us first prove some simple facts about $FO+RPR$.

PROPOSITION 4.1. *The global relation $\leqslant$ ordering the elements of each structure is definable in $FO+RPR$.*

Proof. Write $P(x, y)$ for $x \leqslant y$. Define $P(x, y)$ by RPR as follows:

$$[P(x, y) \equiv x = y \vee P(x, y-1)]. \quad \blacksquare$$

Hence, the successor and predecessor relations can also be defined. Also, 0 and $n-1$, the first and last elements, can be defined.

The RPR recursion scheme allows only recursions of length n. We would like to extend to recursions of length n^k for fixed k. Replace the variable y in the RPR description with a sequence $\mathbf{y}$ of k variables. Order the possible assignments to $\mathbf{y}$ lexicographically and let $\mathbf{y}-1$ denote the predecessor to $\mathbf{y}$ in this order.

PROPOSITION 4.2. *Relational primitive recursions of length n^k, as described above, can be expressed in $FO+RPR$.*

Proof. Let P be defined by using extended RPR. That is,

$$[P(\mathbf{x}, \mathbf{y}) \equiv \theta(\mathbf{x}, \mathbf{y}, P(\mathbf{x}, \mathbf{y}-1))],$$

where $\mathbf{y} = y_1, ..., y_k$. We show by induction on k that P can be define in $FO+RPR$. Let $\mathbf{y}' = y_1, ..., y_{k-1}$ and define two relations P_t and P_f as follows:

$$[P_t(\mathbf{x}, \mathbf{y}', y_k) \equiv y_k = 0 \vee \theta(\mathbf{x}, \mathbf{y}', y_k, P_t(\mathbf{x}, \mathbf{y}', y_k - 1))]$$

$$[P_f(\mathbf{x}, \mathbf{y}', y_k) \equiv y_k \neq 0 \wedge \theta(\mathbf{x}, \mathbf{y}', y_k, P_f(\mathbf{x}, \mathbf{y}', y_k - 1))].$$

The idea behind these definitions is that $P(\mathbf{x}, \mathbf{y})$ can be computed by primitive recursion of length at most n on y_k, provided that the value of $P(\mathbf{x}, \mathbf{y}', 0)$ is known. $P_t(\mathbf{x}, \mathbf{y})$ is $P(\mathbf{x}, \mathbf{y})$ in the case where $P(\mathbf{x}, \mathbf{y}', 0)$ is true; $P_f(\mathbf{x}, \mathbf{y})$ is $P(\mathbf{x}, \mathbf{y})$ in the case where $P(\mathbf{x}, \mathbf{y}', 0)$ is false.

Define a relation $Q(\mathbf{x}, \mathbf{y}')$ by an extended RPR of length n^{k-1}:

$$[Q(\mathbf{x}, \mathbf{y}') \equiv (Q(\mathbf{x}, \mathbf{y}'-1) \wedge \theta(\mathbf{x}, \mathbf{y}', 0, P_t(\mathbf{x}, \mathbf{y}'-1, n-1))$$
$$\vee (\neg\, Q(\mathbf{x}, \mathbf{y}'-1) \wedge \theta(\mathbf{x}, \mathbf{y}', 0, P_f(\mathbf{x}, \mathbf{y}'-1, n-1))].$$

Here the idea is that $Q(\mathbf{x}, \mathbf{y}')$ has the same truth value as $P(\mathbf{x}, \mathbf{y}', 0)$. Then $P(\mathbf{x}, \mathbf{y})$ is equivalent to

$$(Q(\mathbf{x}, \mathbf{y}') \wedge P_t(\mathbf{x}, \mathbf{y})) \vee (\neg Q(\mathbf{x}, \mathbf{y}') \wedge P_f(\mathbf{x}, \mathbf{y})). \quad \blacksquare$$

Unfortunately, $FO + RPR$ does not capture NC^1.

PROPOSITION 4.3. *$FO + RPR$ is properly contained in NC^1.*

Proof. Proof of containment is part of the proof of Theorem 4.7 below. We show that $FO + RPR \neq NC^1$.

Suppose that σ contains just a unary relation symbol, so that σ-structures may be identified with strings in $\{0, 1\}^*$. We will determine which sets of σ-structures satisfy sentences in $FO + RPR$ when structures are assumed to have an underlying linear order.

It is not difficult to see that RPR can be defined in monadic second-order logic assuming that structures have an underlying linear order. Consider the $FO + RPR$ formula φ given by $[P(\mathbf{x}, y) \equiv \theta(\mathbf{x}, y, P(\mathbf{x}, y-1))]$. Let ρ be an invalid sentence and $\alpha(\mathbf{x}, Y)$ be a conjunction of the formulas

$$0 \in Y \leftrightarrow \theta(\mathbf{x}, 0, \rho)$$

and

$$(\forall y > 0)(y \in Y \leftrightarrow ((y-1 \in Y \wedge \theta(\mathbf{x}, y, \neg \rho) \vee (y-1 \notin Y \wedge \theta(\mathbf{x}, y, \rho)).$$

Then the relation $P(\mathbf{x}, y)$ defined by primitive recursion in φ is equivalent to

$$\exists Y(\alpha(\mathbf{x}, Y) \wedge y \in Y).$$

Substituting this monadic second-order formula for free occurrences of P in ψ gives a monadic second-order formula equivalent to φ.

A classical theorem of Büchi (1960) and Elgot (1961) states that the sets of σ-structures satisfying monadic second-order sentences are precisely the regular sets contained in $\{0, 1\}^*$. (In fact, any finite automaton can be simulated using RPR so the sets of σ-structures satisfying sentences in $FO + RPR$ are precisely the regular sets.) Hence, we do not get all of NC^1. $\blacksquare$

To see how to extend $FO + RPR$ to get NC^1 recall that Immerman (1986) and Vardi (1982) needed a linear order (or a successor relation) on each structure in order to obtain a logic characterizing *PTIME*. Immerman (1987b) also needed a linear order for logics characterizing other complexity classes. The reason is that all these classes are defined by restricting time or space resources on ordinary Turing machines. The input tape heads for these machines move sequentially. But we saw in Theorem 2.1 that the natural model of computation for NC^1 is a random access alternating Turing machine. The basic idea of a random access input is that the machine can access the nth cell of the input by writing the binary representation of n on the index tape. We might hope that by adding a relation giving the connection between an integer and its binary representation we get NC^1. This is precisely what happens.

Define a binary relation BIT on each structure in $S(\sigma, n)$ by stipulating that $BIT(x, y)$ holds precisely when the yth bit of the binary representation of x is 1. Fagin (1974) used this relation in his proof showing existential second-order logic captures NP. Denote by $FO+RPR+BIT$ the set of structural functions given by the logic $FO+RPR$ on structures having the global relation BIT. (As before, $FO+RPR+BIT$ also denotes the logic.) We must first prove some basic facts about the logic $FO+RPR+BIT$.

PROPOSITION 4.4. *The following global relations are definable in* $FO+RPR+BIT$.

(i) $\text{Plus}(x, y, z) = \{(x, y, z) \mid x+y=z\}$.

(ii) $\text{Power}(x, y) = \{(x, y) \mid 2^x = y\}$.

(iii) $\text{Ancestor}(x, y) = \{(x, y) \mid x = \lfloor y/2^i \rfloor$ *for some* $i\}$. *This is the ancestor relation for the tree in Fig.* 1.

Proof. (i) Plus is first-order definable from $\leqslant$ and BIT. Define a relation Carry(x, y, i) that holds precisely when there is a carry to the ith position when the binary representations of x and y are added. Thus, Carry(x, y, i) is defined by

$$(\exists j<i)(BIT(x,j) \wedge BIT(y,j)$$
$$\wedge\ (\forall k)(j<k<i \rightarrow (BIT(x,k) \leftrightarrow \neg\ BIT(y,k)))).$$

Then Plus(x, y, z) is a conjunction of the formulas

$$(BIT(z,0) \leftrightarrow (BIT(x,0) \leftrightarrow \neg\ BIT(y,0)))$$

and

$$(\forall i>0)(BIT(z,i) \leftrightarrow (Carry(x,y,i) \leftrightarrow (BIT(x,i) \leftrightarrow BIT(y,i)))).$$

(ii) Power is first-order definable from BIT. Power(x, y) is defined by

$$\forall i (BIT(y,i) \leftrightarrow i=x).$$

(iii) Ancestor is first-order definable from $\leqslant$, *BIT*, and Plus. Ancestor(x, y) is defined by

$$\exists i\, \forall j(BIT(x, j) \leftrightarrow BIT(y, i+j)). \quad \blacksquare$$

Next we introduce relational versions of the recursion schemes used by the algebra $\mathscr{A}$. For simplicity we first state restricted versions of the schemes. Let ρ denote an invalid sentence.

(R1) *Relational Downward Tree Recursion* (RDTR). Let $\theta(\mathbf{x}, y, P(\mathbf{x}, y))$ be a formula as in the RPR scheme. We define a relation P as follows:

$$P(\mathbf{x}, y) \equiv \begin{cases} \theta(\mathbf{x}, 0, \rho) & \text{if } y=0 \\ \theta(\mathbf{x}, yu, P(\mathbf{x}, \lfloor y/2 \rfloor)) & \text{if } y>0. \end{cases}$$

(R2) *Relational Upward Tree Recursion* (RUTR). Let $\theta(\mathbf{x}, y, P(\mathbf{x}, y), P'(\mathbf{x}, y))$ be a formula analogous to the one in the RPR scheme. We define a relation P as follows:

$$P(\mathbf{x}, y) \equiv \begin{cases} \theta(\mathbf{x}, y, \rho, \rho), & \text{if } 2y+1 \geqslant n \text{ or } y=0 \\ \theta(\mathbf{x}, y, P(\mathbf{x}, 2y), P(\mathbf{x}, 2y+1)), & \text{otherwise.} \end{cases}$$

Remark. More generally, we allow the schemes *RDTR* and *RUTR* to apply to trees of height $m \log n$ (for fixed m). Nodes in these trees correspond to sequences of variables $\mathbf{y} = y_1 \cdots y_k$, rather than single variables y. The variables $y_1, \ldots, y_{k-1}$ range over the values $0, 1, \ldots, 2^{\lfloor \log n \rfloor} - 1$ rather than $0, 1, \ldots, n-1$ so that the parents and children of nodes can be easily found. Also, we extend these schemes to simultaneous recursions.

LEMMA 4.5. *Definition by RDTR and RUTR can be expressed in FO + RPR + BIT.*

Proof. Let θ define a relation P as in (R1). Let $\theta'(\mathbf{x}, y, z, P'(\mathbf{x}, y, z-1))$ be the formula

$$\begin{aligned} &(\text{Ancestor}(z, y) \wedge \theta(\mathbf{x}, y, P'(\mathbf{x}, y, z-1))) \\ &\quad \vee (\neg\, \text{Ancestor}(z, y) \wedge P'(\mathbf{x}, y, z-1)). \end{aligned}$$

Use RPR to define a relation P' by

$$[P'(\mathbf{x}, y, z) \equiv \theta'(\mathbf{x}, y, z, P'(\mathbf{x}, y, z-1))].$$

Then $P(\mathbf{x}, y)$ is equivalent to $P'(\mathbf{x}, y, y)$. This shows that RDTR is expressible. Only minor modifications are needed to show that RDTR is expressible for simultaneous recursions on trees of height $m \log n$.

RUTR is more difficult. Let θ define a relation P as in (R2). That is, for fixed $\mathbf{x}$, $P(\mathbf{x}, y)$ is defined at every node y of a binary tree; θ may be regarded as giving the value of $P(\mathbf{x}, y)$ as a Boolean function α of $P(\mathbf{x}, 2y)$ and $P(\mathbf{x}, 2y+1)$. We may suppose that $\alpha(X, Y)$ is either a constant function, X, Y, $\neg X$, $\neg Y$, or $X \wedge Y$, since all other Boolean functions can be expressed as compositions of these functions. (To handle compositions we may need to lengthen the recursion tree and modify the functions at the leaves accordingly.)

We reduce the evaluation of this recursion to the word problem for the symmetric group S_5. This reduction is similar to the reduction of bounded width, polynomial size branching programs to the word problem for S_5 (see Barrington (1986)).

Let e be the identity of S_5 and a be any 3-cycle in S_5. We can define binary Boolean functions in S_5, where e is identified with 0 and a is identified with 1. To be specific, for each Boolean function $\alpha(X, Y)$ listed above there are elements k_1, k_2, k_3, k_4, $k_5 \in S_5$ such that $\alpha(X, Y) = k_1 X k_2 Y k_3 X k_4 Y k_5$.

Suppose, for example, that $\alpha(X, Y) = X \wedge Y$. There are 3-cycles b, c such that $a = bcb^{-1}c^{-1}$. Since all 3-cycles are conjugate, there are elements d_1, d_2, d_3, $d_4 \in S_5$ such that $b = d_1 a d_1^{-1}$, $c = d_2 a d_2^{-1}$, $b^{-1} = d_3 a d_3^{-1}$, $c^{-1} = d_4 a d_4^{-1}$. Let $k_1 = d_1$, $k_2 = d_1^{-1} d_2$, $k_3 = d_2^{-1} d_3$, $k_4 = d_3^{-1} d_4$, and $k_5 = d_4^{-1}$. Then for $X, Y \in \{e, a\}$, $\alpha(X, Y) = k_1 X k_2 Y k_3 X k_4 Y k_5$ is a only if $X = a$ and $Y = a$; in all other cases it is e.

The function $\alpha(X, Y) = X$ is similar. Since a and a^{-1} are conjugate, $a^{-1} = dad^{-1}$ for some element d. Then for $X, Y \in \{e, a\}$, $X = X^{-1}Y^{-1}X^{-1}Y$, so replacing X^{-1} with dXd^{-1} and Y^{-1} with dYd^{-1} everywhere we have $k_1 = d$, $k_2 = k_3 = k_5 = e$, and $k_4 = d^{-1}$. The function $\alpha(X, Y) = Y$ is similar. Constant functions are also similar: use $XYX^{-1}Y^{-1}$ in the manner described above to obtain the function identically equal to 0, for example.

We can negate X by replacing it with $dXd^{-1}a$, and similarly for Y, where d is as in the previous paragraph.

Now consider the recursion tree T for $P(\mathbf{x}, y)$. Transform this tree into a 4-ary tree T' applying the following procedure, beginning at the leaves and working upward to the root. For a node with children p and q, repeat the subtrees lying below p and q so that there are now four children labeled p, q, p, q (in that order). If T has n leaves, T' will have n^2 leaves. T' is easily defined in $FO + RPR + BIT$. Its nodes will be elements whose binary representations have an odd number of bits. If nodes of T have representations requiring l bits, nodes of T' have representations requiring $2l - 1$ bits.

(To be precise we should say that nodes of T' are represented by pairs of elements. Use Power to determine m, the largest power not exceeding n. By using pairs of elements less than m to represent integers, doubling, halving, and all other arithmetical operations needed can be easily defined.) The parent of a node in T' is found by deleting its two low bits. The label of a node in T' is found by deleting every other bit beginning with the second bit.

We describe how to define $P(\mathbf{x}, 0)$ in $FO+RPR+BIT$. We can define $P(\mathbf{x}, y)$ in the same way by considering just the subtree below y in T.

Rather than recursing upward in T, recurse downward in T'. (The proof of RDTR works as well for 4-ary trees as for binary trees.) Each node in T' receives a pair (b, c) from its parent, where $b, c \in S_5$. Which values will be passed on to its children? Suppose a node receives (b, c) from its parent. Find the label y of the node by deleting every other bit. Suppose the Boolean function given by θ at y can be represented as $\alpha(X, Y) = k_1 X k_2 Y k_3 X k_4 Y k_5$. The first child receives (bk_1, e), the second receives (k_2, e), the third receives (k_3, e), and the fourth receives $(k_4, k_5 c)$.

Each leaf of T' will have received a pair (b, c). Consider such a leaf and suppose it has label y. Evaluate $\theta(\mathbf{x}, y, \rho, \rho)$. If it is true associate the group element bac; if it is false associate the group element bc. This association can be done by means of a simple first-order formula. Observe that $P(\mathbf{x}, 0)$ is true if and only if the product of the group elements associated with leaves of T' (in order) is a. This product is easily determined using RPR.

Extending this proof to simultaneous recursions is cumbersome, but not difficult. We increase the branching in the recursion tree because the value of $P_i(\mathbf{x}, y)$ is determined by the values of $P_j(\mathbf{x}, 2y)$ and $P_j(\mathbf{x}, 2y+1)$, where $j = 0, 1, ..., l-1$. We may assume that l is a power of 2. The value of $P_i(\mathbf{x}, y)$ does not correspond to the yth node of the tree, but rather to the $yl+i$th node, but this is easily handled by taking l to be a power of 2.

Extending this proof to trees of height $m \log n$ is straightforward. ∎

LEMMA 4.6. *The relation* Ones(x, y) *holding precisely when the binary representation of y has exactly x 1's can be defined in $FO+RPR+BIT$.*

Proof. In Compton and Laflamme (1988) we sketched a complicated proof of this lemma showing that a bitonic sorting network can be defined in $FO+RPR+BIT$ to sort the bits in the binary representation of y. Barrington, Immerman, and Straubing (1988) gave a much simpler proof showing that Ones (which they call BSUM) is first-order definable from $\leqslant$ and BIT. Their idea is simple but clever. We sketch it here for the sake of completeness.

Let $k = \lceil \log(n+1) \rceil$ so that the bits in the binary representation of y are given by $BIT(y, 0), ..., BIT(y, k-1)$. Suppose that r of these bits are 1.

Consider the first-order formula Odd(y, j) asserting that there is a z such that $BIT(z, j)$ and

$$(BIT(z, 0) \leftrightarrow BIT(y, 0)) \wedge (\forall i > 0)$$
$$\times ((BIT(z, i) \leftrightarrow BIT(z, i-1)) \leftrightarrow (\neg\ BIT(y, i))).$$

This says there are an odd number of 1 bits in the binary representation of y among those in positions $0, ..., j$.

Let $k_1 = \lfloor k/2 \rfloor$. Define an integer y_1 such that among $BIT(y_1, 0), ..., BIT(y_1, k_1 - 1)$, $\lfloor r/2 \rfloor$ bits are 1. Assert that $BIT(y_1, i)$ holds if and only if

$$(\text{Odd}(y, 2i) \wedge \neg\ \text{Odd}(y, 2i+1))$$
$$\vee\ (\text{Odd}(y, 2i+1) \wedge \neg\ \text{Odd}(y, 2i+2) \wedge 2i+2 < k).$$

For each $i < \lceil \log(k+1) \rceil$, let $k_i = \lfloor k/2^i \rfloor$ and define an integer y_i such that among $BIT(y_i, 0), ..., BIT(y_i, k_i - 1)$, $\lfloor r/2^i \rfloor$ bits are 1. Note that the sum of the integers k_i is at most k so the pertinent bits of the integers y_i can be coded into a single variable y'. Set $y_0 = y$ and $k_0 = k$. It is easy to find a first-order formula asserting that y' exists and giving Odd$(y_i, k_i - 1)$. This is the ith bit in the binary representation of x, where x is the number of 1's in the binary representation of y. ▌

Remark. The main result of the section follows. Taking into consideration the proof of Proposition 4.3 we ask if $FO + RPR + BIT$ can be replaced with monadic second-order logic plus BIT in the statement of the theorem. A result of Lynch (1982) shows that this is unlikely. When BIT is present, RPR is strictly weaker than monadic second-order quantification. In Lemma 4.4 above we saw that addition on structures can be defined in $FO + RPR + BIT$. Lynch showed that even with just existential monadic second-order prefixes we get NP. Full monadic second-order quantification gives the entire $PTIME$ hierarchy.

THEOREM 4.7. $NC^1 = FO + RPR + BIT$.

Proof. The proof is along the same lines as the proof of Theorem 3.3. We generate the computation tree for a given random access alternating Turing machine and then evaluate the tree. As before, we have pairs $(x_0, y_0), ..., (x_k, y_k)$ representing the index tape and k work tapes at each node of the tree and z representing the state. Think of these as variables which have new values assigned to them as we follow a branch down the tree. The problem is that in the proof of Theorem 3.3 we needed full DTR to generate these values, whereas now we have only RDTR.

The solution is to generate values in a bitwise fashion. Consider, for example, what information is needed to determine the jth bit of x_i at a given node in the computation tree. This bit is determined from the following information in the parent node: the state, the first bits of $y_0, ..., y_k$ (these are the cells being scanned by the tape heads), and the $(j-1)$st, jth, and $(j+1)$st bits of x_i. Thus, there are just a constant number of bits, say l bits, that are needed. Moreover, the positions of these bits do not depend on tape contents. Now each one of these bits is determined in turn by l values in the grandparent node. Continuing, we obtain an l-branching tree. It is not difficult to define an equivalent binary tree (of greater height) in $FO + RPR + BIT$ by dividing each l-branching into $\log l$ levels of 2-branchings.

Now the leaves of this tree correspond to bits in the initial configuration. If there is a formula of $FO + RPR + BIT$ giving these initial bits, we can use RUTR to compute the value at the root. Such a formula can be specified using RPR and the relation Ones. The reason is that as we move from a node in the tree to one of its children we either change from the position we are looking at to the initial position of one of the y_i's, or change the position in the current string by -1, 0, or 1. Using RPR we can determine, for each branch, the lowermost node where we switch to the initial position of some y_i. Then use Ones to count the number of subsequent times the position is changed by 1 and the number of times it is changed by -1. In this way we can compute the position at the leaf.

Now let us return to evaluation of the original computation tree (not the trees of bit histories discussed in the previous paragraph). Here RUTR suffices since we only pass Boolean variables up the tree.

The other direction of the theorem is easy. Global relations constructed using the connectives and quantifiers of first-order logic and the relation BIT can clearly be evaluated by uniform polynomial size, log depth circuits. To evaluate global relations constructed by RPR use divide and conquer. Evaluate $P(\mathbf{x}, y)$ by dividing the interval $[0, y]$ in two. To do the evaluation on the second half we need to know the value of $P(\mathbf{x}, \lfloor y/2 \rfloor)$; but as there are just two possible values we do both computations and choose the correct one when the value of $P(\mathbf{x}, \lfloor y/2 \rfloor)$ becomes known. ▌

RECEIVED February 22, 1989; FINAL MANUSCRIPT RECEIVED December 5, 1989

REFERENCES

ALLEN, B. (1989), "Arithmetizing Uniform *NC*," Ph.D. thesis, University of Hawaii, Honolulu.

BARRINGTON, D. A. (1986), Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1, *in* "Proc. 18th ACM Symp. on Theory of Computing," pp. 1–5, Association for Computing Machinery, New York.

BARRINGTON, D. A., AND THÉRIEN, D. (1987), Finite monoids and the fine structure of NC^1, *in* "Proc. 19th ACM Symp. on Theory of Computing," pp. 101–109, Association for Computing Machinery, New York.

BARRINGTON, D. A. M., IMMERMAN, N., AND STRAUBING, H. (1988), On uniformity within NC^1, *in* "Proc. 3rd IEEE Conf. on Structure in Complexity Theory," pp. 47–59, IEEE Computer Society Press, Washington, DC.

BENNETT, J. H. (1962), "On Spectra," Ph.D. thesis, Princeton University.

BÜCHI, J. R. (1960), Weak second-order arithmetic and finite automata, *Z. Math. Logik Grundlag. Math.* **6**, 66–92.

BUSS, S. R. (1986), "Bounded Arithmetic," Studies in Proof Theory, Bibliopolis, Naples.

CLOTE, P. (1989), A first order theory for the parallel complexity class *NC*, Boston College Computer Science Department Technical Report BCCS-89-01, Chestnut Hill, MA.

COBHAM, A. (1965), The intrinsic computational difficulty of functions, *in* "Proc. 1964 Intl. Conf. Logic, Methodology and Philosophy of Science" (Y. Bar-Hillel, Ed.), pp. 24–30, North-Holland, Amsterdam.

CODD, E. F. (1972), Relational completeness of data base sublanguages, *in* "Data Base Systems" (R. Rustin, Ed.), pp. 65–98, Prentice-Hall, Englewood Cliffs, NJ.

COMPTON, K., AND LAFLAMME, C. (1988), A logic and an algebra for NC^1, *in* "Proc. 3rd IEEE Conf. on Logic in Computer Science," IEEE Computer Society Press, Washington.

COOK, S. A. (1985), A taxonomy of problems with fast parallel algorithms, *Inform. and Control* **64**, 2–22.

ELGOT, C. C. (1961), Decision problems of finite-automata design and related arithmetics, *Trans. Amer. Math. Soc.* **98**, 21–51.

FAGIN, R. (1974), Generalized first-order spectra and polynomial time recognizable sets, *in* "Complexity of Computations" (R. Karp, Ed.), pp. 43–73, SIAM–AMS Proceedings, Vol. 7, American Mathematical Society, New York.

GUREVICH, Y. (1983), Algebras of feasible functions, *in* "Proc. 24th IEEE Symp. on Foundations of Computer Science," pp. 210–214, IEEE Computer Society Press, Washington, DC.

IMMERMAN, N. (1986), Relational queries computable in polynomial time, *Inform. and Control* **68**, 86–104.

IMMERMAN, N. (1987a), Expressibility as a complexity measure: Results and directions, *in* "Proc. 2nd IEEE Conf. on Structure in Complexity Theory," pp. 194–202, IEEE Computer Society Press, Washington.

IMMERMAN, N. (1987b), Languages that capture complexity classes, *SIAM J. Comput.* **16**, 347–354.

LIND, J. (1974), "Computing in logarithmic space," Project MAC Technical Memo 52, Massachusetts Institute of Technology, Cambridge, MA.

LYNCH, J. (1989), Complexity classes and theories of finite models, *Math. Systems Theory* **15**, 127–144.

RITCHIE, R. (1963), Classes of predictably computable functions, *Trans. Amer. Math. Soc.* **106**, 139–173.

RUZZO, W. (1981), On uniform circuit complexity, *J. Comput. System Sci.* **22**, 365–383.

VARDI, M. (1982), Complexity of relational query languages, "Proc. 14th ACM Symp. on Theory of Computing," pp. 137–146, Association for Computing Machinery, New York.