Extensions to Barrington's M-program model

François Bédard,* François Lemieux** and Pierre McKenzie***
Département d'informatique et recherche opérationnelle, Université de Montréal, C.P. 6128, succursale A. Montréal (Québec), Canada H3C 3J7

Abstract

Bédard, F., F. Lemieux and P. McKenzie, Extensions to Barrington's M-program model, Theoretical Computer Science 107 (1993) 31–61.

Barrington's "polynomial-length program over a monoid" is a model of computation which has been studied intensively in connection with the structure of the complexity class NC^1 [Barrington (1986), Barrington and Thérien (1987, 1988), McKenzie and Thérien (1989), Péladeau (1989)]. Here two extensions of the model are considered. First, with the use of nonassociative structures (hence, groupoids) instead of (associative) monoids, polynomial-length program characterizations of complexity classes TC^0, NL, and LOGCFL, as well as new characterizations of NC^1, are given. New "word problems" complete for LOGCFL, for NL and for NC^1 under DLOGTIME-reductions are obtained as corollaries. Second, using monoids but permitting the use of a different monoid to handle each input length, new complexity classes are defined. Combinatorial arguments are then developed to resolve the relationships between various such classes defined in terms of polynomial-length programs over growing abelian monoid sequences. Then the orders of growing abelian group and monoid sequences required to accept specific languages defined in terms of the presence of a given substring are investigated. Finally, the two extensions are combined to obtain characterizations of L and NL in terms of polynomial-length programs defined over polynomially growing groupoid sequences. It is further argued that such programs are generally no more powerful than LOGCFL.

1. Introduction

An important open problem in complexity theory concerns the relationship between logarithmic space (class L) and polynomial time (class P): Is L indeed a *proper* subset of P? A well-known strategy [11] which would answer this question affirmatively requires proving that a given language $Y \in P$ cannot be accepted by a

* Supported by an NSERC of Canada graduate scholarship.
** Supported in part by an NSERC of Canada graduate scholarship and in part by Québec FCAR Grant 89-EQ-2933.
*** Supported by Canada NSERC Grant A9979 and by Québec FCAR Grant 89-EQ-2933.

polynomial-size branching program. Although unsuccessful to this date because of the notorious difficulty of obtaining significant branching program size lower bounds, this strategy has fostered the investigation of a wide variety of branching program restrictions (see [36] for a partial account). One such restriction of particular interest to us is the polynomial-length, bounded-width branching program [6, 9].

Whereas polynomial-length, bounded-width branching programs were thought at first to be quite weak [6], Barrington [2] proved to the contrary that such programs precisely characterize the complexity class NC^1 [27, 12] of languages accepted by logarithmic depth boolean circuits. Aside from securing the importance of the polynomial-length, bounded-width branching program model, Barrington's approach relied on a consequence of the nonsolvability of the symmetric group of degree 5. This approach suggested a wealth of further branching program restrictions based on the theory of groups and the more general theory of monoids (a *monoid* is a set equipped with an associative binary operation and an identity element). Barrington's "permutation branching programs" thus evolved into the concept of "nonuniform automata" [2, 4, 5] and these later became known more descriptively as "programs over a monoid M" or "M-programs" [25, 23]: loosely speaking, an M-program consists of sequences of simple instructions, one sequence per input length, which "translate" an input string into a word over a monoid M whose value when multiplied out in M determines acceptance or rejection of the input.

Exploring the new concept of an M-program, Barrington and Thérien exhibited a striking relationship between natural subclasses of NC^1 and classical algebraic properties of monoids M over which polynomial-length programs are defined [5, 4]. For example, unbounded fan-in, polynomial-size, bounded-depth circuits with gates chosen from $\{\wedge, \vee, \neg\}$, from $\{MOD_q\}$ and from $\{\wedge, \vee, \neg, MOD_q\}$ have the power of polynomial-length programs defined over "group-free" monoids, solvable groups and solvable monoids, respectively (where a MOD_q gate outputs 0 iff the number of its binary inputs set to 1 is a multiple of q). Drawing from the well-established algebraic theory of finite automata, subsequent work on the power of polynomial-length M-programs has both refined the stratification of NC^1 into meaningful subclasses [23] and reformulated the usual conjectures about the structure of NC^1 in algebraic and in logical terms [3, 23, 25]. Most subclasses of NC^1 considered in the recent literature have now been characterized in terms of polynomial-length M-programs by restricting the monoid M to belong to appropriate natural subclasses of all monoids (see [22]). A notable exception so far has been the class TC^0 defined in terms of bounded depth circuits of unbounded fan-in gates from $\{\wedge, \vee, MAJORITY\}$.

Given the deep connection between the various polynomial-length M-program *restrictions* and the structure of NC^1, in this paper we consider *extensions* of the model. Our goals are, on the one hand, to capture more complexity classes, and on the other, to offer the possibility of parametrizing the presumed "gap" between NC^1 and L using algebraic criteria, in the hope of facilitating the development of lower-bound techniques applicable to separate NC^1 or its subclasses from P or even L.

While maintaining the polynomial-length requirement, we define two "orthogonal" extensions. The first extension consists of allowing a nonassociative structure G (called a *groupoid*) instead of just a monoid M. Since "multiplying out a sequence of groupoid elements" is no longer uniquely defined, we must make precise how evaluation of a word over a groupoid is to be carried out for the purpose of determining acceptance or rejection of an input. A successful definition consists of allowing all possible bracketings and accepting the input iff one of the bracketings evaluates to an accepting groupoid element. (See Section 2 for precise definitions of this model and of acceptance.) The second extension retains associativity but permits the use of a different monoid M for each input length. We note as in [2] a consequence of [13] that in a certain sense polynomial-length computation over groups of polynomial degree (hence, of possibly exponential order) captures L. Hence, this second extension in effect allows an algebraically controlled parametrization of the *width* parameter in a polynomial-length branching program. Interesting parameters pertaining to our second extension are then the algebraic properties of the monoids used and the growth rate of their orders as a function of input length.

Our first extension, from monoid M to groupoid G, allows capturing complexity classes which seemed unattainable in Barrington's model. Indeed we first show that a language is accepted by a polynomial-length G-program iff it belongs to the class LOGCFL of languages reducible in logarithmic space to a context-free language [32, 12, 35]. This provides yet another characterization of the class LOGCFL and shows that Barrington's M-program model extends meaningfully to describe classes beyond NC^1. (The class LOGCFL is believed to properly contain nondeterministic logarithmic space, denoted NL, and is contained in the class NC^2 defined in terms of polynomial-size $(\log n)^2$-depth boolean circuits.)

Then we prove the existence of a groupoid G such that a language is accepted by a polynomial-length G-program iff the language belongs to NL, itself presumed to be larger than NC^1. We can also obtain TC^0, a subclass of NC^1 left out of the M-program framework: we prove that there is a family $\mathscr{F}$ of groupoids such that a language is accepted by a polynomial-length G-program with G drawn from $\mathscr{F}$ iff the language belongs to TC^0. Finally, we exhibit a groupoid of order 10 over which polynomial-length programs characterize NC^1. (This in contrast with the conjecture [5] that polynomial-length M-programs require a monoid M of order 60 in order to capture NC^1.)

The above results remain valid when the very strict DLOGTIME uniformity criterion [7, 3] is imposed on the relevant G-programs. As corollaries, we exhibit specific groupoids, whose "word problems" (loosely defined as languages of strings of groupoid elements which can be bracketed in such a way as to evaluate to a prescribed element) are complete for LOGCFL, for NL and for NC^1, respectively, under DLOGTIME-reductions. Our groupoid in the case of NC^1 has order 10, again in contrast with Barrington's NC^1-complete word problem over the smallest nonsolvable group [2]: this traces the somewhat obscure difference between the NC^1-

complete formula value [7, 8] and width-5 graph accessibility [2, 3] problems to the structural difference between groupoids and monoids.

Our second extension, from monoid M to monoid family $\{M_n\}$, yields much more unusual complexity classes. In the present paper we restrict our attention to the case of nonuniform $\{M_n\}$-programs in which $\{M_n\}$ is a sequence of abelian monoids. (Inputs of length n are handled by the nth monoid in the sequence, see Section 2 for precise definitions.) Simple observations are that such programs can be taken to have linear length and that a sequence of cyclic groups growing exponentially (linearly) in order can serve to recognize any language (any symmetric language).

But intriguing combinatorial questions arise when we consider even simple languages like that over alphabet Σ prescribed by the regular expression $\Sigma^*w\Sigma^*$ for $w\in\Sigma^*$. Anderson and Barrington [1] observed (against all odds) that $\{0, 1\}^*01\{0, 1\}^*$ can be accepted using a cyclic group sequence of order $O(n^3)$. We explain this here by giving necessary and sufficient conditions under which $\Sigma^*w\Sigma^*$ can be accepted by a polynomial-length program using a polynomial-order group sequence: in essence Anderson and Barrington hit upon the only significant exception to the rule that $\Sigma^*w\Sigma^*$ can be accepted using a subexponential order abelian monoid sequence iff the length of w is less than 2 (Theorem 5.11). To further illustrate the somewhat subtle behavior of programs over growing abelian monoid families, we also investigate the languages $\Sigma^*w_1\Sigma^*w_2\cdots w_l\Sigma^*$ with $w_i\in\Sigma$; see Theorem 5.12 for the precise results.

More generally, we compare the classes of languages accepted (nonuniformly) by polynomial-length programs over polynomial-order and subexponential-order cyclic group, cyclic monoid, abelian group and abelian monoid sequences. We develop combinatorial arguments which determine, with four exceptions involving cyclic groups and monoids, the exact relationship between any pair of the eight classes which arise. The precise results may be found as Theorem 5.10.

What happens if we combine our two extensions and consider polynomial-length $\{G_n\}$-programs with G_n a groupoid dedicated to inputs of length n? We show that such programs with polynomially growing groupoid sequences $\{G_n\}$ are no more powerful than LOGCFL. We further show that when the program also specifies a "left-to-right" order of evaluation of sequences of groupoid elements (see Section 2 for precise definitions), polynomial-length, polynomial-order programs characterize the class L. Finally, when the order of evaluation is only partially specified (in a suitably restricted way, see Section 2), polynomial-length, polynomial-order programs yield another characterization of the class NL.

The organization of this paper is as follows. Section 2 provides background and defines our extended model. Section 3 deals with the case of programs over fixed groupoids and with the characterizations of LOGCFL, NL, NC^1 and TC^0. Section 4 treats the cases of programs over polynomially growing groupoid sequences describing classes L and NL. Section 5 discusses the unusual complexity classes arising from programs over growing abelian monoid families. Finally, Section 6 lists some open questions and concludes.

2. Background and definitions

We encounter the following complexity classes: $AC^0 \subset TC^0 \subseteq NC^1 \subseteq L \subseteq NL \subseteq$ LOGCFL. The classes AC^0 [14] and TC^0 [24, 3] are defined in terms of polynomial-size, bounded-depth circuits of unbounded fan-in, the former with gates from $\{\vee, \wedge, \neg\}$ and the latter with gates from $\{\vee, \wedge, \text{MAJORITY}\}$. (A MAJORITY-gate outputs 1 iff at least half of its inputs are 1.) The class NC^1 is the set of languages accepted by logarithmic depth boolean circuits with fan-in two gates from $\{\wedge, \vee, \neg\}$. The circuits have access to both the inputs and their negations. The classes L and NL are deterministic and nondeterministic logarithmic space, respectively (see [18]). Finally, LOGCFL is the set of languages reducible in log space to a context-free language [32]. Alternatively, LOGCFL equals NC^1CFL [12], the set of languages reducible via an NC^1-computable function to a context-free language.

Following [3], we take the circuit-based complexity classes to be DLOGTIME-uniform, i.e., the "direct connection languages" [30] of circuit families have to be recognizable by random access logarithmic time deterministic Turing machines. For definiteness, we choose as direct connection language of a circuit family [3] the set of tuples $\langle t, a, b, y\rangle$, specifying that gate numbered a, of type t, is input to gate numbered b in the family member handling inputs of length equal to the length of y. In the special case in which there are explicit functions $d(n)$ and $f(n)$ such that, for each n, the nth circuit in the family is a tree of depth at most $d(n)$ and fan-in at most $f(n)$, we further insist that each circuit node be numbered as a $d(n)$-tuple of $\lceil \lg(f(n))\rceil$-bit "bytes", encoding the path from the root to this node (with the number of a shallow node including "empty trailing bytes"). This special case occurs in our definition of TC_k^0 below.

A DLOGTIME-reduction [7] from language A to language B is a function many-one reducing A to B such that f increases the length of strings only polynomially and the predicate $A_f(c, i, z)$, specifying that the ith symbol of $f(z)$ is c, is recognized in DLOGTIME. The non-uniform versions of the classes L and NL are denoted by L/poly and NL/poly, respectively [20], and we denote by nonuniform LOGCFL the set of languages reducible via a nonuniform NC^1-computable function to a context-free language. It can be shown that an equivalent natural definition of nonuniform LOGCFL can be given in terms of nonuniform semi-unbounded fan-in circuits of log depth, as in [35].

Fix $k \geqslant 0$. We now define class TC_k^0, i.e., depth-k TC^0.

Definition 2.1. A language is in TC_k^0 iff it is accepted by a DLOGTIME-uniform circuit family having access to boolean constants, to both the inputs and their negations, and satisfying three properties. First, the gates allowed are AND, OR, and MAJORITY. Second, each circuit is a tree of depth at most k. Third, for each n, the fan-in of each nonleaf node in the circuit handling inputs of size n is precisely n.

The extended first-order characterization of TC^0 [3, Theorem 9.1] implies that $\mathrm{TC}^0 = \bigcup_{k\geq 0}\mathrm{TC}^0_k$. (Note that the tree depth does not exactly match the formula quantifier depth because of the need to evaluate the constant portion of the formula.) Furthermore, a TC^0_k circuit family can be simulated by a DLOGTIME-uniform family in which the nth circuit is a full, depth-k, $2n$-ary tree of MAJORITY-gates. Indeed, consider the nth circuit C_n in a family satisfying Definition 2.1. Fan-in n ANDs and ORs can be replaced with fan-in $2n$ MAJORITYs with n constant inputs. Then more constants can be introduced to double the fan-in of the original fan-in n MAJORITYs. Finally, each short path from root to leaf can be extended to depth k by the attachment of a full $2n$-ary subtree of MAJORITYs with replicated leaves. Now to preserve DLOGTIME-uniformity, the n-ary tree C_n is first embedded in a $2n$-ary tree T_{2n} in a natural way. Then the technique used in proving [3, Theorem 9.1 ($1\Rightarrow 2$)] applies, noting that the input or constant at a given leaf v of T_{2n} is largely determined by the gate type of the C_n leaf embedded along the path from v to the root of T_{2n}.

Throughout this paper a *grammar* refers to a context-free grammar $D=(V, T, P, S)$ (see for instance [18, 16]). Grammar D is in *Chomsky normal form* if its rules are of the form $A\rightarrow BC$ or of the form $A\rightarrow a$ for nonterminals $A, B, C\in V$ and terminal $a\in T$ (with the possible exception of rule $S\rightarrow\varepsilon$ for ε the empty string). Grammar D is *linear* if the right-hand side of each rule in P contains at most one nonterminal, and it is *invertible* if no two distinct rules have identical right-hand sides [16]. A language is said to be linear if it is generated by a linear grammar (see [16]).

Fact 2.2 (Greibach [15]). *There is a complete language for the class of context-free languages under many-one* NC^1*-computable reducibility.*

Fact 2.3 (Sudburough [31]). *There is a complete language for the class of linear context-free languages under many-one* NC^1*-computable reducibility.*

Fact 2.4 (see Harrison [16]). *There exists an algorithm which transforms a grammar D into an equivalent invertible grammar D' in such a way that if D is in Chomsky normal form (D is linear) then D' is in Chomsky normal form (D' is linear), with the proviso that in D' rules of the form $S\rightarrow A$ for S the starting symbol and A a nonterminal are permitted and are the only rules (except possibly $S\rightarrow\varepsilon$) involving S.*

For our purposes a *groupoid* is a set equipped with a binary operation and an identity element. Our interest is in finite groupoids only. A *monoid* is an associative groupoid. A monoid M is *abelian* if $g\cdot h=h\cdot g$ for any $g, h\in M$. The symmetric group of degree 5 is denoted S_5. The *order* of a groupoid by G, denoted by $|G|$, is the number of elements in G.

Let Σ be a finite alphabet.The length of $w\in\Sigma^*$ is denoted by $|w|$ and w_i refers to the ith symbol in w. Even when G is a finite groupoid, we denote by G^* the monoid of all finite sequences of elements of G under concatenation. Now let G be a finite groupoid and let $w\in G^*$. We denote by $G(w)$ the set of all groupoid elements which can be obtained by parenthesizing w in a legitimate way and evaluating the resulting expression by "multiplying out" in G. When w is the empty string ε we define $G(w)$ to be the singleton containing the identity of G. If $F \subseteq G$ the F *word problem over* G is the language $\{w \in G^*\colon G(w) \cap F \neq \emptyset\}$. For $H \subseteq G$ we further define $W(H, F, G)$ as the F word problem over G restricted to H^*.

We now describe our extended M-program model. In its full generality the model is conveniently obtained from Barrington's M-programs by replacing *sequences* of instructions by *ordered trees* with one instruction per leaf. The purpose of these trees is to specify a (partial or complete) bracketing of the program instructions. Informally speaking, the "program" first assigns to each leaf of the tree a groupoid element obtained by "translating" the appropriate input symbol. The program then evaluates each node of the tree in a bottom-up fashion: a node is assigned the *set* made up of *all* groupoid elements which can be obtained by multiplying the groupoid elements associated with its children (the order of the children cannot be changed, but each ordered sequence which can be obtained by drawing a single groupoid element from each child can be bracketed in any legitimate way). This, in fact, exactly corresponds to the evaluation of a partially bracketed sequence of instructions. To evaluate such a sequence, we have to find all complete bracketings of the sequence that are consistent with the partial bracketing already given and evaluate each one of them: the set of all elements that can be obtained is the value of the partially bracketed sequence. We now make this formal. Let $\mathscr{V}$ be a family of groupoids.

Definition 2.5. A *structured* $\mathscr{V}$*-program* is an infinite sequence $\Pi = \Pi_0, \Pi_1, \Pi_2, \ldots$ where Π_n is a triple $\langle G_n, T_n, F_n\rangle$ dealing with inputs of length n. Here G_n is an element of $\mathscr{V}$ and $F_n \subseteq G_n$ is a set of accepting elements. Finally, T_n is a rooted ordered tree having at least 2 descendants per internal node and having at each leaf an *instruction* (i, f) for $i \in \{1, 2, \ldots, n\}$ an input position and $f\colon \Sigma \to G_n$ a total function. The program *length* is a function mapping n to the number of leaves in T_n for each n and the program *order* is a function mapping n to $|G_n|$ for each n.

Now pick $w \in \Sigma^*$ and say $|w| = n$. We describe how a structured program Π operates on input w by recursively defining a function *eval* which assigns to each node of T_n a subset of G_n. If u is a leaf of T_n with instruction (i, f) then *eval*(u) is the singleton $\{f(w_i)\}$. Otherwise, if $u_1, \ldots, u_k$ are the children of u, then *eval*(u) is defined as

$$\bigcup_{x_1 \in eval(u_1), \ldots, x_k \in eval(u_k)} G_n(x_1 x_2 \ldots x_k).$$

We let $\Pi(w)$ denote the subset of G_n assigned by *eval* to the root of T_n.

In the context of a program over a groupoid G, let $c_x\colon \Sigma \to G$ be a "constant function" which assigns to each $a \in \Sigma$ the same $x \in G$. Then we will denote the "constant program instruction" $(1, c_x)$ simply by x.

Definition 2.6. The language accepted by a $\mathscr{V}$-program Π is $L(\Pi) = \{w \in \Sigma^*\colon \Pi(w) \cap F_{|w|} \neq \emptyset\}$.

Definition 2.7. A flat $\mathscr{V}$-program is a structured $\mathscr{V}$-program in which each tree is of height 1.

Important remark. Observe that in a *flat* $\mathscr{V}$-program all ordered trees are, in fact, sequences, in direct analogy with the case of Barrington's M-programs. Observe further that when $\mathscr{V}$ contains monoids alone, the tree structures have no effect on the evaluation of a $\mathscr{V}$-program on any given input so that $\mathscr{V}$-programs can be assumed to be flat with no loss of generality. Finally, although this may not be true in general, in all the cases considered in this paper it has been possible to transform structured programs into equivalent flat programs over sometimes slightly more complicated groupoids. *Unless the word "structured" explicitly appears, any mention of a $\mathscr{V}$-program in this paper, therefore, refers to a flat $\mathscr{V}$-program.*

Families $\mathscr{V}$ which we will consider are $\mathscr{G}d$ (groupoids), $\mathscr{M}$ (monoids), $\mathscr{A}\mathscr{M}$ (abelian monoids), $\mathscr{C}\mathscr{M}$ (cyclic monoids), $\mathscr{A}\mathscr{G}$ (abelian groups), and $\mathscr{C}\mathscr{G}$ (cyclic groups). All $\mathscr{V}$-programs considered in this paper have polynomial length. If G is a particular groupoid we define a G-program to be a $\{G\}$-program with the additional restriction that it must use the same set of accepting elements F_n for each input of length n. Note then that the $\mathscr{V}$-program notation is consistent with the M-program notation since acceptance by structured G-programs reduces to Barrington's notion when G is a monoid.

It will be helpful to further classify programs as follows. A P-$\mathscr{V}$-program (SE-$\mathscr{V}$-program) refers to a polynomial order (subexponential order) $\mathscr{V}$-program. We will say that a structured $\mathscr{V}$-program is *deterministic* if all its trees are binary, i.e., if it corresponds to a *fully bracketed* sequence of instructions; it is nondeterministic otherwise. If a deterministic $\mathscr{V}$-program Π is such that for some constant c all the (binary) trees of Π have the property that the right subtree of any node has size at most c, then Π is deemed "*left-to-right*" (abbreviated LTR-$\mathscr{V}$-program). A structured $\mathscr{V}$-program obtained from a deterministic LTR-$\mathscr{V}$-program Π by possibly replacing each constant size right binary subtree of Π by a constant-size arbitrary subtree is called a *non-deterministic* LTR-$\mathscr{V}$-program. These trees correspond to fully or partially bracketed sequences of the form $(\cdots((B_1 \cdot B_2) \cdot B_3) \cdots B_m)$, where B_i is a fully or partially bracketed sequence of instructions of length at most c; such sequences can be evaluated (nondeterministically) from left-to-right without ever having to keep track of more than c instructions at a time.

We use $\mathscr{L}(\ )$ to mean "the set of languages accepted by polynomial-length programs of a certain type", to wit, for example, $\mathscr{L}(\text{SE-}\mathscr{A}\mathscr{M})$ and $\mathscr{L}(\text{LTR-P-}\mathscr{G}d)$.

In the spirit of [2, 3], we define our uniformity notion in the context of $\mathscr{V}$-programs as follows. We say that a G-program Π is DLOGTIME-uniform if each of the following three languages is accepted in DLOGTIME:

- $\{\langle w, a, b, c\rangle\colon a \cdot b = c \text{ in } G_{|w|}\}$,
- $\{\langle w, k, S\rangle\colon$ the kth "symbol" of the bracketed expression of instructions representing $T_{|w|}$ is S (a "symbol" being a bracket or an instruction)$\}$,
- $\{\langle w, a\rangle\colon a \in F_{|w|}\}$.

3. Programs over fixed groupoids

We say that a language $Y \subseteq A^*$ is *recognized* by a groupoid G if there exists a subset $F \subseteq G$ and a "translation function" $\theta: A \to G$, extending to a monoid morphism $\theta: A^* \to G^*$, such that $Y = \{x \in A^* \mid G(\theta(x)) \cap F \neq \emptyset\}$. The cornerstone of the algebraic theory of finite automata is the well-known fact (see for instance [26]) that a language is regular iff it is recognized in this sense by a finite monoid. Hence, the following observation, to some extent already implicit in Valiant's work [34], is of independent interest.

Lemma 3.1. *A language is context-free iff it is recognized by a finite groupoid.*

Proof. ($\Leftarrow$): Let G be a groupoid with set of elements $[k] = \{1, 2, \ldots, k\}$ and 1 the identity of G. Let F be a subset of G, A a finite set, $Y \subseteq A^*$ a language and $\theta: A^* \to G^*$ a monoid morphism such that $Y = \{x \in A^* \mid G(x)) \cap F \neq \emptyset\}$. We construct a grammar $D = \langle V, T, P, S \rangle$ for Y as follows:

$$V = \{q_i\colon 0 \leqslant i \leqslant k\},$$

$$T = A = \{a_1, \ldots, a_m\},$$

$$P = \{q_i \to a\colon a \in A,\ \theta(a) = i\}$$

$$\cup \{q_0 \to q_i\colon i \in F\} \cup \{q_1 \to \varepsilon\}$$

$$\cup \{q_i \to q_j q_l\colon i, j, l \in [k] \text{ and } j \cdot l = i\},$$

$$S = q_0.$$

An induction on the length of x proves

$$(\forall x \in A^*)(\forall y \in G)\ [(y \in G(\theta(x))) \text{ iff } (q_y \overset{*}{\Rightarrow} x)].$$

($\Rightarrow$): Let $Y \subseteq A^*$ be a context-free language produced by a grammar $D = \langle V, A, P, q_0 \rangle$, where $A = \{a_1, \ldots, a_m\}$ and $V = \{q_0, \ldots, q_k\}$. By Fact 2.4, we can assume that D is invertible and in Chomsky normal form with the only rules involving q_0 of the form $q_0 \to \varepsilon$ or of the form $q_0 \to q$ for $q \in V$. We define the groupoid $G = (V \setminus \{q_0\}) \cup \{e, \$\}$ such that e is the identity, $\$ \cdot a = a \cdot \$ = \$$ for every $a \in G$ and $a \cdot b = c$ iff $c \to ab$ is in P, for every $a, b, c \in V$.

Now define $X = \{q \in V \setminus \{q_0\} \mid (q_0 \to q) \in P\}$ and $F = X \cup \{e\}$ if $\varepsilon \in Y$ and $F = X$ otherwise. Define also the monoid morphism $\theta: A^* \to G^*$ induced by $\theta(a) = q$ iff $q \to a$ is in P, for each $a \in A$. As above we can show that

$$(\forall x \in A^*)(\forall q \in V \setminus \{q_0\})\ [(q \overset{*}{\Rightarrow} x) \text{ iff } (q \in G(\theta(x)))].$$

This concludes the proof. $\square$

From now on we only consider languages over $\{0, 1\}$ but this is easily generalized.

Proposition 3.2. *Nonuniform* LOGCFL *is characterized by nonuniform, polynomial-length programs over fixed groupoids.*

Proof. ($\supseteq$): A nonuniform, polynomial-length G-program accepting a language Y provides a nonuniform NC^1-reduction from Y to a word problem over G. Such a word problem is trivially recognized by G and is, thus, context-free by Lemma 3.1. Hence, $Y \in NC^1 CFL = LOGCFL$ in the nonuniform setting.

($\subseteq$): Let $Y \in NC^1 CFL$. The output of the NC^1-computable function f reducing Y to a context-free language $W \subseteq \{0, 1\}^*$ may be viewed as the output of a sequence of S_5-programs [2] each of which would multiply out individually to an element of S_5, say a or e depending on whether the corresponding NC^1-subcircuit output is 1 or 0, where e is the identity in S_5. By Lemma 3.1, there exists a groupoid G_0, a translation function $\theta: \{0, 1\} \to G_0$, and $F \subseteq G_0$ such that $w \in Y$ iff $f(w) \in W$ iff $G_0(\theta(f(w)))$ contains an element of F. We define a G-program for G a groupoid whose set of elements is the disjoint union of the monoid S_5, the groupoid G_0 and the set $\{e_G, \#\}$. Products within the subgroupoid S_5 and within the subgroupoid G_0 are defined in the obvious way, e_G is the identity of G, and we further define $a \cdot \# = \theta(1)$, $e \cdot \# = \theta(0)$, $\# \cdot \# = \#$ and in all other cases $x \cdot y = \$$, where $\$$ is the absorbing element of G_0. Then we insert at the left of each "sequence of S_5 instructions" producing an output bit of the NC^1-reduction the constant program instruction $\#$: this yields the instruction sequence $T_{|w|}$ in our (flat) G-program. Finally, we define each accepting subset in the program to be F. □

Theorem 3.3. LOGCFL *is characterized by* DLOGTIME-*uniform, polynomial-length programs over fixed groupoids.*

Proof. ($\supseteq$): Same as in the corresponding part of the proof of Proposition 3.2, noting here that a DLOGTIME-uniform program provides a (uniform) NC^1-reduction.

($\subseteq$): Let $Y \in NC^1 CFL$. Then Y NC^1-reduces to a context-free language W, but also to the context-free language 0^*1W, where the latter reduction allows us to assume that the length of the output of the NC^1-reduction on inputs of length n is of the form $m(n) = 2^{k(n)}$. For a fixed n write $k = k(n)$ and $m = m(n)$. We require that the NC^1-subcircuits computing the reduction to a context-free language be of the same depth and appropriately balanced. We define B_i to be the DLOGTIME-uniform S_5-program associated with the ith output of the reduction (see Proposition 3.2). We view B_i here as a full-fledged program (Definition 2.5) in which for each n such that $m(n) < i$ the nth component of B_i is irrelevant. Now write $B_{i,n}$ for the sequence of instructions prescribed by the nth component of B_i.

Let G be the groupoid constructed in the proof of Proposition 3.2 and let Z_n be a full binary tree of depth k (only used as a tool to construct our program inductively). For each node N of Z_n we recursively construct a sequence of G-instructions σ_N as follows. If N is the ith leaf then $\sigma_N = B_{i,n}$. If N is an internal node then N_1 and N_2 are its sons for which we have constructed the sequences σ_{N_1} and σ_{N_2}. Let t be the length of σ_{N_1} and σ_{N_2} and let the sequence $\#^t$ be the constant instruction $\#$ repeated t times. Then the sequence associated with N is $\sigma_N = \sigma_{N_1} \#^t \sigma_{N_2} \#^t$. Now for each input of length n we define the (height one) tree T_n as the instruction sequence σ_{N_n}, where N_n is the root of Z_n. We claim that $\Pi_n = \langle G, T_n, F \rangle$ is the nth component of a DLOGTIME-uniform G-program accepting Y, where F is as in the proof of Proposition 3.2.

Indeed it is easy to verify that Y is accepted because the program constructed is the same as that in Proposition 3.2, except that here we inserted more than one constant instruction between each S_5-program (the program behavior is unchanged because we have constructed G such that $\# \cdot \# = \#$). To see the uniformity, observe that the k first blocks of two bits in the (binary) index of any instruction tell us whether this instruction is a $\#$. When the instruction is not a $\#$, the first bits of these blocks give us the index of the S_5-subprogram in which the instruction lies, and we can then appeal to the known DLOGTIME-uniformity [3] of each S_5-subprogram used in the construction to locate the specified instruction in DLOGTIME. This proves our claim and concludes the proof. □

Corollary 3.4. *There is a fixed groupoid G and a fixed $F \subset G$ such that the F word problem over G is* LOGCFL-*complete under* DLOGTIME-*reductions.*

Proof. By Fact 2.2, there is a context-free language complete for $\mathrm{NC}^1\mathrm{CFL} = \mathrm{LOGCFL}$ via an NC^1-computable function, so that the construction of Theorem 3.3 then yields the desired groupoid and word problem. This word problem is context-free; hence, is in LOGCFL, by Lemma 3.1. □

We note parenthetically that, cast into the first-order expressibility logical framework used in [3], Theorem 3.3 extends the characterization of languages expressible in "FO with monoidal quantifiers" to the case of their obvious generalization as "*groupoidal* quantifiers": There is a groupoidal quantifier Q_f such that $\mathrm{LOGCFL} = (FO + Q_f)$.

The rest of this section is devoted to characterizations of subclasses of LOGCFL in terms of DLOGTIME-uniform, polynomial-length programs. In view of Theorem 3.3, it is not surprising that there should exist groupoids complicated enough to simulate NL, NC^1, or TC^0 computations with polynomial-length programs. However, a cumbersome technical difficulty resides in the verification that the groupoids used are simple enough for the target complexity classes to include their word problems. Often this amounts to a somewhat detailed case-by-case analysis which explains the lengthy arguments sometimes required.

We note the following consequence of the facts that the boolean formula value problem is NC^1-complete and that parenthesis languages belong to NC^1 [7].

Theorem 3.5. *Deterministic structured programs of polynomial-length over fixed groupoids characterize* (*nonuniform*) NC^1.

Proof. First, we show that any NC^1-circuit can be simulated by a structured program over groupoid $G=\{e,0,1\}$, where $\forall x\in G$: $e\cdot x=x=x\cdot e$ and $\forall x,y\in\{0,1\}$: $x\cdot y=\neg(x\wedge y)$. We first convert the circuit to a boolean formula of depth $O(\log n)$ containing only NOTs, ANDs, ORs and inputs. With groupoid G, we can compute any such formula F using instruction sequence $Ins(F)$, where $Ins(\neg F_1)=(Ins(F_1)\cdot Ins(F_1))$, $Ins(F_1\wedge F_2)=((Ins(F_1)\cdot Ins(F_2))\cdot Ins(F_1)\cdot Ins(F_2)))$, $Ins(F_1\vee F_2)=((Ins(F_1)\cdot Ins(F_1))\cdot(Ins(F_2)\cdot Ins(F_2)))$, and $Ins(x_i)=(i,f)$, where $f(0)=0$ and $f(1)=1$. It is quite obvious that $Ins(F)$ is a fully parenthesized sequence of instructions computing F and that its length will be at most $4^{O(\log n)}$, which is polynomial. The set of accepting elements F_n will be $\{e,1\}$ if ε is in the language to be accepted and $\{1\}$ if not.

One can also simulate a deterministic G-program by an NC^1-circuit. First note that if we construct from the groupoid G a context-free grammar with alphabet $G\cup\{(,)\}$, productions $A\to(BC)$ iff $B\cdot C=A$ and start symbol $S\in G$, then the result is a parenthesis language generating all fully parenthesized products yielding S. Since all parenthesis languages are in NC^1 [7], one can test via an NC^1-circuit if the product of a given fully parenthesized sequence of instructions is S for any particular $S\in G$. So, to simulate a G-program by an NC^1-circuit, we first compute the value of each instruction (which can be done by a simple projection), insert parentheses at the right places, verify for each $S\in F_n$ if the product is S and finally output the OR of those tests. □

By considering parentheses as elements of the groupoid we can construct a groupoid G such that the class of languages recognized by flat programs over G also corresponds to NC^1.

Theorem 3.6. *There is a groupoid G of order* 10 *such that* DLOGTIME-*uniform polynomial-length programs over G characterize* NC^1.

Proof. Recall the binary boolean function NAND defined as the negation of the AND function. The language of boolean formulae which evaluate to 1 when NAND is the sole (implicit) boolean operator is generated from symbol I by the grammar rules

$$O\to(II)\,|\,(O)\,|\,0,$$

$$I\to(OO)\,|\,(OI)\,|\,(IO)\,|\,(I)\,|\,1.$$

A groupoid G will be constructed using this grammar modified so that it is in Chomsky normal form and invertible. Define $M=(V,T,P,I)$, where $V=\{O,I,A,B,C,D,X,Y\}$, $T=\{0,1,(,)\}$ and the rules of P are:

$$O\to AB\,|\,XD\,|\,CY\,|\,0,$$

$$I\to CD\,|\,CB\,|\,AD\,|\,AY\,|\,XB\,|\,1,$$

$$A\to XI,\quad B\to IY,$$

$$C\to XO,\quad D\to OY,$$

$$X\to(,\quad Y\to).$$

Note that the rules $O \to CY$ and $I \to XB$ can be removed without changing the language generated by M but are included in order to facilitate the proof that $\mathscr{L}(G) \subseteq \mathrm{NC}^1$. Let $G = V \cup \{\$, e\}$ with the following operation

$$a \cdot b = \begin{cases} a & \text{if } b = e, \\ b & \text{if } a = e, \\ c & \text{if } c \to ab \in P, \\ \$ & \text{otherwise.} \end{cases}$$

It is easy to see that the language generated by M is recognized by G using the monoid morphism $\phi : T^* \to G^*$ induced by $\phi(1) = I$, $\phi(0) = O$, $\phi(() = X$, $\phi()) = Y$ and using $F = \{I\}$.

To prove that $\mathrm{NC}^1 \subseteq$ DLOGTIME-uniform $\mathscr{L}(G)$ we show that for every language $U \subseteq \{0, 1\}^*$ recognized by an NC^1-circuit of depth d there is a G-program B of length 16^d recognizing U. Without loss of generality we can assume that the circuit is a full binary tree. The proof is by induction on d. If $d = 0$ the circuit is a constant, a variable or its negation and in this case B is defined as a single instruction. Suppose now $d > 0$ and the statement is true for every circuit of depth less than d. Let C be the output gate of the circuit. Both inputs to C (say C_1 and C_2) are outputs of depth $d - 1$ circuits and by the induction hypothesis the languages of these circuits are recognized by G-programs (say B_1 and B_2) of length 16^{d-1}. Let $B = [[[[[B_1 B_2][B_1 B_2]]]]]$ if C is an AND-gate and $B = [[[[[B_1 B_1][B_2 B_2]]]]]$ if C is an OR-gate, where [(resp.]) is the constant instruction X (resp. Y) repeated 16^{d-1} times. We see that the length of B is 16^d and it is a simple exercise to show that $x \in U$ iff $I \in B(x)$. This concludes the induction. Now, because we have ensured that our G-programs grow in a controlled way as a function of circuit depth, the same argument as in [3] applies to prove that the G-program constructed is DLOGTIME-uniform.

To prove that DLOGTIME-uniform $\mathscr{L}(G) \subseteq \mathrm{NC}^1$ we need to show that $W(G, F, G) \in \mathrm{NC}^1$ for every $F \subseteq G$. It suffices to show that $W(G, \{Z\}, G) \in \mathrm{NC}^1$ for each $Z \in G$. The case $Z = e$ is clear, and the case $Z = \$$ follows from the observation that any sequence of 4 elements from $G \setminus \{e\}$ can be bracketed in such a way as to multiply out to the absorbing element \$. Cases $Z = X$ and $Z = Y$ are easy because neither X nor Y appear within the multiplication table of G. Now let Σ denote the set $\{O, I, X, Y\}$. We already know that $W(\Sigma, \{I\}, G) \in \mathrm{NC}^1$ because this is a parenthesis language [7]. Hence, the case $Z = I$ is handled by the following claim: $w \in W(G \setminus \{e, \$\}, \{I\}, G)$ iff the string obtained from x by replacing every occurrence of A (resp. B, C, D) by XI (resp. IY, XO, OY) belongs to $W(\Sigma, \{I\}, G)$.

To see this claim, consider for example $uIYv$ for $u, v \in G^*$. Clearly $G(uBv) \subseteq G(uIYv)$. Now suppose that a bracketing of $uIYv$ yields I when multiplied out in G. Since multiplication on the left by Y yields the absorbing element \$, Y is necessarily "consumed from the left". Now except for the product $X \cdot I$, multiplication on the right by I also yields \$. Hence, either $I \cdot Y = B$ is used to consume I, in which case clearly $I \in G(uBv)$ and we are done, or $X \cdot I$ is eventually used, resulting in an intermediate expression which we can take to be $u'AYv$ for some $u' \in G^*$. Since multiplication on the right by A yields \$, Y must then be consumed using $A \cdot Y = I$, resulting in $u'Iv$. But then $I \in G(u'Iv) \subseteq G(u'XBv) \subseteq G(uBv)$. (Note the necessity of the redundant rule $I \rightarrow XB$.) Similar arguments apply to the replacements of A, C and D, proving our claim.

Now let $w \in G^*$ and suppose we want to know if $w \in W(G \setminus \{e, \$\}, \{Z\}, G)$ for $Z \in V \setminus \{I, X, Y\}$. For $|w| = 1$ the answer is immediate and if $|w| > 1$ we can prove the following:

(1) $w \in W(G, \{O\}, G)$ iff $XwB \in W(G, \{I\}, G)$.
(2) $w \in W(G, \{A\}, G)$ iff $wB \in W(G, \{O\}, G)$.
(3) $w \in W(G, \{B\}, G)$ iff $Aw \in W(G, \{O\}, G)$.
(4) $w \in W(G, \{C\}, G)$ iff $wB \in W(G, \{I\}, G)$.
(5) $w \in W(G, \{D\}, G)$ iff $Aw \in W(G, \{I\}, G)$.

Let us prove the first fact. The "only if" is trivial so consider the "if". In any evaluation of XwB, when we consume the rightmost B the result must be I or O (refer to grammar M). But there is no way to evaluate a sequence to I if the last symbol is O and we can evaluate a string vI to I only if v is the empty string. Hence, Xw must evaluate to X or to C (the only two symbols giving I when multiplied with B). The first case is impossible since w is not the empty string; so, Xw must evaluate to C. Then the only possibility is that w be evaluated to O. Similar arguments are used to prove the other facts.

This takes care of all the cases, concluding the proof of Theorem 3.6. □

Corollary 3.7. *There is a fixed groupoid G of order* 10 *and a fixed $F \subset G$ such that the F word problem over G* is NC^1-*complete under* DLOGTIME-*reductions.*

It is interesting to note that while we do not know the order of a minimal groupoid accepting exactly NC^1 with flat programs, we can show that 3 elements are necessary (and sufficient) to accept exactly NC^1 with deterministic programs. This is because the groupoids of order less than 3 are abelian monoids and they cannot accept $\{0,1\}^*11\{0,1\}^* \in \mathrm{NC}^1$ (see Section 5).

Theorem 3.8. *For each $k \geqslant 1$ there is a fixed groupoid G_k such that* $\mathrm{TC}_k^0 \subseteq \mathscr{L}(G_k) \subseteq \mathrm{TC}^0$.

Proof. Let C be a depth-k circuit of MAJORITY-gates. For each gate g of C we denote by $g(x)$ the value output by g when x is input to C. If g is on level i of the circuit we recursively construct a well-parenthesized expression $f_g(x) \in \{0, 1, \langle, \rangle\}^*$ of nesting depth $i-1$, as follows. In the case where $i = 1$, $f_g(x)$ is simply the sequence of bits used as input to g. If $i > 1$ we define $f_g(x) = \langle f_{g_1}(x) \rangle \cdots \langle f_{g_m}(x) \rangle$, where $g_1, \ldots, g_m$ are the input gates to g. It is clear that $g(x) = 1$ iff $f_g(x)$ evaluates to 1 when we recursively apply the MAJORITY function to the list of operands at a given level. For each k we will construct groupoid G_k from a grammar D_k generating any such depth-$k-1$ expression that evaluates to 1.

For $k \geqslant 1$ then define the grammar $D_k = (V_k, T, \pi_k, S)$, where $T = \{0, 1, \langle, \rangle\}$, $V_1 = \{S\} \cup \{M_1, P_1, Q_1, A_1, B_1\}$ and $V_k = V_{k-1} \cup \{M_k, P_k, Q_k, A_k, B_k, E_k, F_k, L, R\}$

for $k > 1$. In order to describe π_k we first define for all $i \geqslant 1$ and all $j > 1$ the following set of rules:

$$
\begin{aligned}
U_i = \{ & M_i \to A_i B_i \mid B_i A_i,\\
& P_i \to A_i P_i \mid A_i A_i,\\
& Q_i \to B_i Q_i \mid B_i B_i \},\\
W_1 = \{ & A_1 \to M_1 A_1 \mid A_1 M_1 \mid 1,\\
& B_1 \to M_1 B_1 \mid B_1 M_1 \mid 0 \},\\
W_j = \{ & A_j \to M_j A_j \mid A_j M_j \mid L E_j,\\
& B_j \to M_j B_j \mid B_j M_j \mid L F_j,\\
& E_j \to M_{j-1} R \mid P_{j-1} R \mid A_{j-1} R,\\
& F_j \to Q_{j-1} R \mid B_{j-1} R,\\
& L \to \langle,\\
& R \to \rangle \}.
\end{aligned}
$$

Then we define $\pi_k = \{S \to M_k \mid P_k \mid A_k\} \cup \bigcup_{1 \leqslant i \leqslant k} (U_i \cup W_i)$. To see what language is generated by D_k note that starting from M_k we can produce any string $w \in \{A_k, B_k\}^+$ such that $\sharp_{A_k}(w) = \sharp_{B_k}(w)$ ($\sharp_c(w)$ denotes the number of occurrences of the symbol c in w). Starting from A_k (B_k) we produce exactly all $w \in \{A_k, B_k\}^*$ such that $\sharp_{A_k}(w) = \sharp_{B_k}(w) + 1$ ($\sharp_{B_k}(w) = \sharp_{A_k}(w) + 1$) and starting from P_k (Q_k) we produce exactly all $w \in \{A_k, B_k\}^*$ such that $\sharp_{A_k}(w) \geqslant \sharp_{B_k}(w) + 2$ ($\sharp_{B_k}(w) \geqslant \sharp_{A_k}(w) + 2$).

Hence, grammar D_k generates exactly the set of well-parenthesized expressions of nesting depth $k-1$, which evaluate to 1 when MAJORITY is taken recursively at each level. Now construct a groupoid G_k from grammar D_k as groupoid G was constructed from grammar M in the proof of Theorem 3.6, but excluding from G_k the element labelled S.

To prove $\mathrm{TC}_k^0 \subseteq$ DLOGTIME-uniform $\mathscr{L}(G_k)$, let $Y \in \mathrm{TC}_k^0$, let $x \in \{0, 1\}^n$ and let g denote the output gate of a TC_k^0-circuit C_n determining whether $x \in Y$. Recall from Section 2 that we can take C_n to be a full depth-k, $2n$-ary tree of MAJORITY-gates. Then $x \in Y$ iff $f_g(x)$ is generated by grammar D_k, i.e., iff $G_k(f_g(x)) \cap \{M_k, A_k, P_k\} \neq \emptyset$. Hence, a program Π of length $|f_g(x)|$ over G_k accepts all strings of length $|x|$ in Y. This program can be made DLOGTIME-uniform exactly as the "generalized expressions" obtained from an FO formula are made DLOGTIME-uniform [3, Proof of Theorem 9.1 ("1⇒4")], with the role of the "space character" played here by the constant instruction e, for e the groupoid identity.

We now turn to the proof that DLOGTIME-uniform $\mathscr{L}(G_k) \subseteq \mathrm{TC}^0$, i.e., $G_k(w)$ can be computed in TC^0 for any word $w \in G_k^*$. First note that given a word $w \in G_k^*$ we have that for all $i \geqslant 0$ if there is a symbol E_i or F_i that is not immediately preceeded by an L then the only possible evaluation for w is \$ (see grammar D_k). Otherwise we just have to replace each occurrence of LE_i by A_i and each LF_i by B_i and this does not change $G_k(w)$. So in the following we will not consider symbols E_i and F_i. We will proceed by proving the two following claims:

Claim I. Every evaluation of a word over G_1 can be done in TC^0.

Claim II. Every word w can be transformed in TC^0 into a word v such that $X_{l+1} \in G_{l+1}(w)$ iff $X_l \in G_l(v)$ (where X is a place holder for any nonterminal).

Let $u, v \in \{A_1, B_1\}^*$, $E = \{M_1, P_1, Q_1, A_1, B_1\}$ and $X \in E$. It is a simple exercise to show the following facts about the grammar N_1 (where we have suppressed the subscripts for clarity):

(1) For $|uv| > 1$, $X \overset{*}{\Rightarrow} uMv$ iff $X \overset{*}{\Rightarrow} uv$.

(2) For $u \neq \varepsilon$, $P \overset{*}{\Rightarrow} uP$ iff $P \overset{*}{\Rightarrow} uA$.

(3) For $u \neq \varepsilon$, $Q \overset{*}{\Rightarrow} uQ$ iff $Q \overset{*}{\Rightarrow} uB$.

(4) If $X \overset{*}{\Rightarrow} uPv$ then $X = P$ and $v = \varepsilon$.

(5) If $X \overset{*}{\Rightarrow} uQv$ then $X = Q$ and $v = \varepsilon$.

To evaluate a word w over G_1 we do the following: (i) Verify that there is at most one P_1 (Q_1) in w: using (4) and (5) the P_1 (Q_1) must be at the end in which case the only possible evaluation different from \$ is P_1 (Q_1). Then replace P_1 (Q_1) by A_1 (B_1) using (2) ((3)). (ii) Replace each M_1 by the identity e of G_1 (fact(1)). (iii) We now have a word $v \in \{A_1, B_1, e\}^*$ that can be easily evaluated in TC^0. To see this let v' be obtained from v by interchanging the A_1's and the B_1's and then observe the following.

- $M_1 \in G(v)$ iff $EQUAL(v) \equiv MAJ(v) \wedge MAJ(v')$
- $A_1 \in G(v)$ iff $EQUAL(vB_1)$
- $B_1 \in G(v)$ iff $EQUAL(vA_1)$
- $P_1 \in G(v)$ iff $MAJ(v) \wedge \neg EQUAL(vB_1) \wedge \neg EQUAL(v)$
- $Q_1 \in G(v)$ iff $MAJ(v') \wedge \neg EQUAL(vA_1) \wedge \neg EQUAL(v)$

where $MAJ(v)$ is true iff v has at least as many A_1 than B_1. This proves our Claim I.

We now describe the TC^0-transformation from $w \in G_{l+1}^*$ to $v \in G_l^*$ which will prove Claim II. Recall that there is no E_i or F_i symbol in w.

(1) Check whether there is a symbol $x \in G_1$ that is not inside a substring of the form $\langle v \rangle$ for $v \in (G_1)^*$. In such a case the only possible result is \$ and we can determine this in AC^0.

(2) Look for any substring of the form $\langle v \rangle$ for $v \in (G_1)^*$ and replace it (including the brackets) with

- A_2 if $G_1(v) \cap \{M_1, P_1, A_1\} \neq \emptyset$,
- B_2 if $G_1(v) \cap \{Q_1, B_1\} \neq \emptyset$,
- \$ otherwise.

This step is feasible in TC^0 by Claim I.

(3) Replace every symbol X_i by X_{i-1} for each $2 \leqslant i \leqslant l+1$ and for each "nonterminal" X (this does not affect parentheses). We are left with a new word v, which evaluates to some $X_l \in G_l$ iff w evaluates to the corresponding $X_{l+1} \in G_{l+1}$. This step is easily performed in AC^0.

This concludes the proof. □

Theorem 3.9. *There is a groupoid G such that* DLOGTIME-*uniform polynomial-length programs over G characterize* NL.

Proof. Say a grammar $G=\langle V, T, P, S\rangle$ is in *Chomsky-linear form* if there exists a partition $V_1 \cup V_2$ of V, such that:

(1) Each rule using S is of the form $S\rightarrow\varepsilon$ or $S\rightarrow A$ for $A\in V$.

(2) Each rule not using S has the form $A\rightarrow BC$ or $A\rightarrow a$, where $a\in T$, B, $C\in V$ and at most one of B or C is in V_2.

(3) If $q\rightarrow\alpha$ is a rule in P and $q\in V_1$ then $\alpha\in T$.

It is not hard to see that a language is linear iff it is generated by some Chomsky-linear grammar. Furthermore, using Fact 2.4 we can take this grammar to be invertible.

Let D_0 be a Chomsky-linear invertible grammar that generates the NL-complete linear context-free language L_0 (Fact 2.3) and let G_0 be the groupoid constructed from D_0 as in Lemma 3.1. Let $G=G_0\cup S_5\cup\{\#, e_G\}$ (where $\#$ is a new element) be a groupoid defined as in Proposition 3.2.

Since the proof that NL $\subseteq$ DLOGTIME-uniform $\mathscr{L}(G)$ is identical to that of Theorem 3.3, we only prove the reverse inclusion, i.e., $W(G, \{Z\}, G)\in$NL for every $Z\in G$. We claim the following.

Claim I. $W(G_0, \{Z\}, G_0)$ is a linear language for every $Z\in G_0$.

Claim II. $W(S_5, F, S_5)\in$NL for any $F\subseteq S_5$.

Claim I follows from two observations. First, if a grammar is linear then the same grammar using a different start symbol still generates a linear language. Second, the language of sentential forms of any linear grammar is linear. This shows that $W(G_0, \{Z\}, G_0)$ is linear for any $Z\neq\$$ (where $\$$ is the absorbing element of G_0). We note that any string of elements of G_0 that is composed of at least 4 elements different from the identity can be evaluated to $\$$. Hence, $W(G_0, \{\$\}, G_0)$ is also a linear language. This proves Claim I, and Claim II is clear because $W(S_5, F, S_5)\in\text{NC}^1$ for any $F\subseteq S_5$ [2].

Now suppose we want to determine if $w\in W(G, \{Z\}, G)$ for $w\in G^*$ and $Z\in G$. Clearly $w\in W(G, \{\#\}, G)$ iff $w\in\{\#\}^+$; so, we only have to consider cases $Z\neq\#$. If we restrict w to be in S_5^* or in G_0^* then by the above claims the problem is in NL. If $w=v_0u_1\#v_1\ldots u_k\#v_k$, where $u_i\in S_5^+$ and $v_i\in G_0^*$ then we can check whether w evaluates to Z with a nondeterministic logspace Turing machine as follows. This machine uses two pointers, one pointing at the beginning of the input (moving towards the right) and the other at the end (moving towards the left). It simulates the generation of string $w\in W(G_0, \{Z\}, G_0)$ by a linear grammar by guessing which rule is to be used and moving the two pointers in accordance with this rule. To do this it has to replace each substring $u_i\#$ by the corresponding element of G_0. So, if the left pointer scans an element of S_5 (or the right pointer scans $\#$) then the machine memorizes the current nonterminal of the grammar and multiplies out the string of S_5 elements delimited by the next occurrence of $\#$ (the next occurrence of $\#$ or of an element in G_0 for the right pointer). It then computes the product of the result with # and continues. The entire construction is straightforward and the details are omitted. □

4. Programs over growing groupoids

In this section we consider programs over polynomially growing groupoid sequences, and we defer the subcase of (abelian) monoid sequences until the next section.

Theorem 4.1. LOGCFL *is characterized by* DLOGTIME-*uniform polynomial-length programs over polynomial-order groupoids.*

Proof. ($\subseteq$): This inclusion follows immediately from Theorem 3.3.

($\supseteq$): Let Y be a language recognized by a DLOGTIME-uniform polynomial-length program over polynomial-order groupoids where the nth component is $B_n = \langle G_n, T_n, F_n \rangle$. Clearly there is a logspace reduction from Y to $X = \{(G_n, F_n, x) \mid G_n(x) \cap F_n \neq \emptyset,\ n = |x|\}$. It suffices to argue that $X \in$ LOGCFL. Ruzzo [29] presents an algorithm to recognize any language in LOGCFL on an alternating Turing machine using O(log n) space and O(log n) alternations (where n is the size of the input). The alternating Turing machine used by Ruzzo possesses the semi-unboundedness property described by Venkateswaran [35], who proves that, in fact, such machines operating in O(log n) space and O(log n) alternations characterize LOGCFL. We observe that Ruzzo's algorithm extends in a straightforward manner to show that the word problem remains in this class even if the groupoid is part of the input. Hence, $X \in$ LOGCFL. □

This result shows that, in the case of flat programs over groupoids, letting the groupoid grow does not change the power at all (provided that the order is bounded by a polynomial). But we also saw that if we use a special fixed groupoid, we can characterize NL. There is another way to restrict the power of programs over groupoids to characterize NL (and also L) without restricting the groupoids used and by letting them have polynomial order. The restriction will be based on the structure of the tree of instructions. To prove these results, we will need some preliminary lemmas.

Say a Turing machine is *oblivious* (see [28]) if the motion of its input head (and advice head if the machine is nonuniform) depends only on $|w|$.

Lemma 4.2. *Let M be a Turing machine using only* O(log n) *space. Then M can be simulated by an oblivious logspace bounded Turing machine that always halts, for which the motions of the input head and of the advice head are identical, and for which the position of the heads at time t is computable in* DLOGTIME. *This theorem holds for all 4 combinations of uniform/non-uniform deterministic/nondeterministic Turing machines.*

Proof. We prove the nonuniform nondeterministic case, the deterministic case and the uniform case being special cases (since the simulation does not use nondeterminism and does not modify the advice). Let M be a nonuniform Turing machine $\mathrm{O}(\log n)$ space. Let $I(n)$ be the contents of the advice tape for input length n.

First, construct M_2 from M so that it always halts on any input. This is a well-known transformation that uses a counter to keep track of the number of steps of M that have been simulated. Define $L(n)=$ the least power of 2 greater than $\max(n, |I(n)|)$ and construct M_3 from M_2 so that both input heads (input and advice tapes) occupy only positions 0 to $L(n)$. This can easily be done using a counter for each of the two input tapes to keep track of where the heads are supposed to be, without moving them left of 0 or right of $L(n)$. Construct M_4 from M_3 so that the advice head and the input head are always at the same position. This can also be done using counters to keep track of the head positions in M_3, but always moving the heads together (the heads are not forced to move at each step).

The last transformation will give an oblivious machine M_5 constructed from M_4: in M_5, the input heads will always move from left to right until they reach $L(n)$, then from right to left until they reach 0, and so on. For this, we will use a counter for the position of the input heads and another counter for the position where they are supposed to be in M_4. The heads are also permitted not to move. In order for the machine to be oblivious, we will force the heads to move exactly once every $b(n)$ steps, where $b(n)$ is a bound on the number of steps required by M_5 to update its counters ($b(n)=$ the least power of 2 greater than or equal to $c_1|n|+c_2$). This bound will be computed at the beginning of the execution and a tape will be initialized to $10^{b(n)-1}1$ (this computation will certainly be oblivious since it will depend only on n).

Then M_5 is an oblivious machine that simulates M, and has all the desired properties. □

Theorem 4.3. L/poly *is characterized by deterministic LTR-P-$\mathscr{G}$d-programs.*

Proof. It is not hard to see that any deterministic LTR-P-$\mathscr{G}d$-program can be simulated by a nonuniform Turing machine using only $\mathrm{O}(\log n)$ space. The advice tape is used to store the groupoid multiplication table and the parenthesized sequence of instructions (representing the binary tree). Since the size of any right subtree is bounded by a constant, such a subtree can be evaluated in $\mathrm{O}(\log n)$ space. Since the sequence is parenthesized from left to right, only the result of at most 2 subtrees need to be stored at any time, so the entire algorithm needs only $\mathrm{O}(\log n)$ space.

Now let M be a nonuniform Turing machine using $\mathrm{O}(\log n)$ space. Using Lemma 4.2, we can choose M to be oblivious and so that it always halts. Let $I(n)$ be the contents of the advice tape for input length n. Let $q(n)$ be a polynomial that bounds the execution time of M and let $s(n)\in\mathrm{O}(\log n)$ be a function that bounds the space used by M.

Let B be the blank symbol of M and Σ the input alphabet of M. Let $G_n=\{e, Y, N\} \cup (\Sigma\cup\{B\})^2 \cup \{c|c$ is a configuration of M that gives the first $s(n)$ symbols of each work tape, along with head positions and state}. The identity is e. We define $c\cdot(\sigma_I, \sigma_A)$ = "the next configuration of M, starting from c with σ_I under the input head and σ_A under the advice head". If M halts in this situation, then $c\cdot(\sigma_I, \sigma_A)=Y$ if M has accepted and N if not. We also define $Y\cdot(\sigma_I, \sigma_A)=Y$ and all other products to be N.

Fix $n \geqslant 1$. Let $H(t)$ be the position of the input heads of M at time t, for inputs of length n. Then the following simple instruction sequence (parenthesized from left to right) simulates M: $[\cdots[[(1, f)\cdot(p_{H(1)}, g_{H(1)})]\cdot(p_{H(2)}, g_{H(2)})]\cdots(p_{H(q(n))}, g_{H(q(n))})]$, where

(1) $f(\sigma)$ = initial configuration of M or Y if the initial state is accepting,

(2) $p_i = i$ if $1 \leqslant i \leqslant n$ and 1 otherwise,

(3) $g_i(\sigma) = (\mathit{input}(i, \sigma), \mathit{advice}(i))$ (where $\mathit{input}(i, \sigma)=\sigma$ if $1 \leqslant i \leqslant n$ and B if not; $\mathit{advice}(i) = (I(n))_i$ if $1 \leqslant i \leqslant |I(n)|$ and B otherwise).

This instruction sequence yields Y if M accepts its input and N otherwise; so, F_n will be defined as $\{Y\}$ if $\varepsilon \notin L(M')$ and $\{e, Y\}$ otherwise. □

Corollary 4.4. L *is characterised by* DLOGTIME-*uniform deterministic* LTR-P-$\mathcal{G}d$-*programs.*

Proof. Since DLOGTIME $\subseteq$ L, we can simulate any DLOGTIME-uniform LTR-P-$\mathcal{G}d$-program in L. The simulation proceeds as in Theorem 4.3, but instead of using an advice tape, the machine will simulate the DLOGTIME-machine computing the description of the LTR-P-$\mathcal{G}d$-program each time it needs a bit in this description.

For the other direction, we proceed as in Theorem 4.3. We have to prove that each bit of the description of the resulting LTR-P-$\mathcal{G}d$-program can be computed in DLOGTIME. The machine has to be able to compute any bit in the multiplication table of $G_{|w|}$, but the only nontrivial case is $c\cdot(\sigma_I, \sigma_A)$ (note that σ_A is always the blank symbol). This requires only a simulation of one step of a L-machine, which can be done in DLOGTIME. Computing the set of final elements (which is always $\{Y\}$ or always $\{e, Y\}$) can trivially be done in DLOGTIME. Computing bits in the sequence of instructions can also be done in DLOGTIME, since the function $H(t)$ giving the head position at time t is computable in DLOGTIME. □

Theorem 4.5. NL/poly *is characterized by nondeterministic LTR-P-$\mathcal{G}d$-programs.*

Proof. The proof is similar to that of Theorem 4.3. To simulate a nondeterministic LTR-P-$\mathcal{G}d$-program by a nondeterministic nonuniform Turing machine, we use the nondeterminism of the Turing machine to evaluate the constant-size arbitrary right subtrees and proceed as in the deterministic case.

For the other direction, we take a Turing machine that works in $O(\log n)$ space, force it to have only 2 nondeterministic choices in each configuration, then use Lemma 4.2 to transform it into an oblivious machine M that always halts. It can be verified that this transformation will not add more nondeterministic choices. We then use a groupoid very similar to the one used in Theorem 4.3. We add new elements

$C1$, $C2$ and the elements of $(\Sigma \cup \{B\})^2 \times \{1, 2\}$; define $c \cdot (\sigma_I, \sigma_A, j)$ = "the next configuration of M, starting from c with σ_I under the input head and σ_A under the advice head, when it makes the nondeterministic choice j", $C1 \cdot (\sigma_I, \sigma_A) = (\sigma_I, \sigma_A, 1)$, $C1 \cdot (\sigma_I, \sigma_A, j) = (\sigma_I, \sigma_A, j)$, $(\sigma_I, \sigma_A) \cdot C2 = (\sigma_I, \sigma_A, 2)$, $(\sigma_I, \sigma_A, j) \cdot C2 = (\sigma_I, \sigma_A, j)$, all other products involving those new elements $= N$; and redefine $c \cdot (\sigma_I, \sigma_A) = N$.

The program used is much the same except that each instruction (but the first) is replaced by $[C1 \cdot (p_{T(t)}, g_{T(t)}) \cdot C2]$, where $C1$ and $C2$ are constant instructions. □

Corollary 4.6. DLOGTIME-*uniform nondeterministic LTR-P-$\mathcal{G}$d-programs characterize* NL.

Proof. Similar to that of Corollary 4.4. □

Theorem 4.7. *There is a family $\mathcal{G}$ of groupoids such that* DLOGTIME-*uniform programs over $\mathcal{G}$ characterize* L.

Proof. First note that if the structured programs in the proof of Theorem 4.3 are replaced by flat programs over the same groupoid families, then by construction of these groupoids the only way not to obtain groupoid element N when "multiplying out in the groupoids" is essentially by forming the products as if the parentheses were still present. Now there exists for L a complete language under quantifier-free projections [19] and it suffices to use the groupoid families constructed (as in Theorem 4.3) from the fixed Turing machines accepting these languages. □

5. Growing abelian monoids

In this section we investigate the power of nonuniform programs over various families of abelian monoids. Simple facts about such programs are that they can be taken to have linear-length and that nonuniform programs over cyclic groups of exponential order recognize any language whatsoever. Observe further that programs over linear-order cyclic groups capture all symmetric languages.

Let us first look at the power of $\mathcal{CM}$-programs compared to $\mathcal{CG}$-programs. Lemma 5.1 and Corollary 5.2 show that they are very similar.

Lemma 5.1. *Any $\mathcal{CM}$-program of order $f(n)$ can be simulated by a $\mathcal{CG}$-program of order $nf(n) - n + 1$.*

Proof. To simulate a $\mathcal{CM}$-program by a $\mathcal{CG}$-program, we first note that since any cyclic monoid G_n has a generator, any computation in G_n can be carried out in $\mathbf{N}$ (the natural numbers) using only the exponents, and then converted back into an element of G_n simply by a subtraction and a modulo operation. Since it is always possible to force the number of instructions to be n, the maximal number used in that integer computation will be at most $n(f(n)-1)$. Therefore, it is sufficient to use a cyclic group of order $n(f(n)-1)+1=nf(n)-n+1$ to simulate the $\mathcal{CM}$-program. $\square$

Corollary 5.2. *The class of languages accepted by P-$\mathcal{CM}$-programs is equal to the class of languages accepted by P-$\mathcal{CG}$-programs. The same is true with SE-$\mathcal{CM}$-programs and SE-$\mathcal{CG}$-programs.*

Programs over abelian groups can also be simulated by $\mathcal{CM}$-programs, but since their structure is more complex than cyclic monoids, the upper bound we obtain is not as good as the one from Lemma 5.1.

Lemma 5.3. *An order $f(n)$ $\mathcal{AG}$-program can be simulated by a $\mathcal{CG}$-program of order $(f(n))^{\lg(n)+1}$.*

Proof. To simulate an $\mathcal{AG}$-program by a $\mathcal{CG}$-program, we use the fact that any finite abelian group is isomorphic to a finite product of finite cyclic groups (see [17]). Suppose $G=Z_{a_1}\times Z_{a_2}\times\cdots\times Z_{a_k}$, where $a_i\geqslant 2$ (so $k\leqslant\lg(f(n))$). Using a method similar to the one used in the preceding simulation, we can simulate any computation in G by a computation in Z_c, where $c=\prod_{i=1}^{k}(n(a_i-1)+1)$. For $n\geqslant 1$, we have $c\leqslant\prod_{i=1}^{k}na_i=n^kf(n)\leqslant n^{\lg(f(n))}f(n)=(f(n))^{\lg(n)+1}$. $\square$

Corollary 5.4. *The class of languages accepted by P-$\mathcal{AG}$-programs is included in the class of languages accepted by SE-$\mathcal{CG}$-programs.*

Thus, $\mathcal{CM}$-programs can be simulated by $\mathcal{CG}$-programs by roughly multiplying their order by n, and $\mathcal{AG}$-programs can be simulated by $\mathcal{CG}$-programs by raising their order to the $(\lg(n)+1)$th power. One can expect having difficulties simulating $\mathcal{AM}$-programs with $\mathcal{CG}$-programs. In fact, we can show that an exponential blow-up in the order of the program is sometimes *required* to simulate $\mathcal{AM}$-programs even by $\mathcal{AG}$-programs. To prove that, we need techniques to find lower bounds on the order of $\mathcal{AG}$-programs that accept a language L. The following theorems are preliminary steps towards the goal of resolving the relationship between the classes of languages defined by programs over abelian monoids.

Theorem 5.5. *Let L be a language over an alphabet Σ. Fix $n\geqslant 0$. Suppose there is a set $S\subseteq\{1,2,\ldots,n\}$ of input positions and two symbols a and b in Σ such that for each pair of distinct subsets T' and T'' of S, there exists a word $X\in\Sigma^n$ such that exactly one of X' and X'' is in L (X' is defined from X by replacing each symbol at a position in T' by symbol a and each symbol at a position in $S\backslash T'$ by symbol b; X'' is defined analogously from X using T''). Then any $\mathcal{AM}$-program recognizing L requires $|M_n|\geqslant 2^{|S|}$.*

Proof. Consider two distinct subsets T' and T'' of S. By hypothesis, there is a word X of length n such that exactly one of X' and X'' is in L. But X' and X'' are identical everywhere except (at least once) in the positions of S. So, if we consider only the instructions that depend on input positions in S, their product (on inputs X' and X'') must differ because every other instruction depends on positions that contain the same symbol in X' and X'' and the product over all input positions must differ in order to distinguish between the two words (this part requires that M_n be associative and commutative, so that the product can be done in any way). So we can associate a different element of M_n with each subset of S. Since there are $2^{|S|}$ subsets of S, there are at least that many elements in M_n. □

Theorem 5.6. *Let L be a language over an alphabet Σ. Let $n \geqslant 0$. Suppose there is a set $S \subseteq \{1, 2, \ldots, n\}$ of input positions and two symbols a and b in Σ such that for each pair of distinct subsets T' and T'' of S with empty intersection, there exists a word Y of length n such that exactly one of Y' and Y'' is in L (Y' is the word obtained from Y by replacing each symbol at a position in T' by symbol a and each symbol at a position in T'' by symbol b; and vice-versa for Y''). Then any $\mathscr{AG}$-program recognizing L requires $|G_n| \geqslant 2^{|S|}$.*

Proof. The proof is similar to that of Theorem 5.5, except that the existence of inverses for every element allows us to ignore positions on which both "substitutions" act the same. Hence, we get away with a weaker hypothesis concerned only with the case in which $T' \cap T'' = \emptyset$ and substitute only the positions in $T' \cup T''$. □

Corollary 5.7. *The classes of languages defined with abelian monoids (i.e. SE-$\mathscr{AM}$-programs and P-$\mathscr{AM}$-programs) are not included in any of the classes defined with abelian groups (i.e. SE-$\mathscr{AG}$-programs and P-$\mathscr{AG}$-programs).*

Proof. It suffices to show the existence of a language L that is accepted by a P-$\mathscr{AM}$-program without being accepted by any SE-$\mathscr{AG}$-program. Let $L = (00)^*1\Sigma^*$, where $\Sigma = \{0, 1\}$ (L is the set of binary strings that have their first 1 in an odd position). L is easily recognized by a P-$\mathscr{AM}$-program using the monoids MIN_{n+1} (MIN_{n+1} is the monoid with elements $\{0, 1, \ldots, n\}$ and operation $a \cdot b = \min(a, b)$).

To prove that L cannot be recognized by any SE-$\mathscr{AG}$-program, we use Theorem 5.6 with $S = \{2, 4, \ldots, 2\lfloor \frac{n}{2} \rfloor\}$, $a = 0$ and $b = 1$. Let T' and T'' be distinct subsets of S that do not intersect. Let c be the least element in $T' \cup T''$ and suppose, without loss of generality, that it is in T'. If we take $Y = 0^c 1 0^{n-c-1}$, then the first 1 of $Y' = Y_{T' \leftarrow 0,\, T'' \leftarrow 1}$ is in position $c+1$ while the first 1 of $Y'' = Y_{T' \leftarrow 1,\, T'' \leftarrow 0}$ is in position c, i.e. $Y' \in L$ and $Y'' \notin L$. We can then conclude that any $\mathscr{AG}$-program accepting L must have order greater than or equal to $2^{|S|} = 2^{\lfloor n/2 \rfloor}$, which is not subexponential. □

This last corollary shows that families of growing abelian monoids are much more powerful than families of growing abelian groups, even if the latter are permitted to grow much faster than the former. We can show, though, that the program order has a very important influence on the languages that can be accepted, and that abelian monoids are not always more powerful than abelian groups.

Theorem 5.8. *Let $f: N \to N^*$ be a function such that $f(n) \leqslant 2^{n/2}$ when $n \geqslant n_0$. Then there exists a language L that can be accepted by an $\mathscr{A}\mathscr{G}$-program of order $f(n)$, but that cannot be accepted by an $\mathscr{A}\mathscr{M}$-program of order $g(n)$, for any function $g: N \to N^*$ such that $g(n) \leqslant \frac{1}{2} f(n)$ for sufficiently large n.*

Proof. Let $a(n) = \lfloor \lg(f(n)) \rfloor$. When $n \geqslant n_0$, we have $f(n) \leqslant 2^{n/2}$; so, $a(n) = \lfloor \lg(f(n)) \rfloor \leqslant \lg(f(n)) \leqslant n/2$.

Let $L = \bigcup_{n \geqslant n_0} (\text{Parity}_2)^{a(n)} \Sigma^{n-2a(n)}$, where $\Sigma = \{0, 1\}$ and Parity_2 is the language $\{00, 11\}$.

L can easily be accepted by an $\mathscr{A}\mathscr{G}$-program using the groups $(Z_2)^{a(n)}$. The order of the program is then $2^{a(n)} = 2^{\lfloor \lg(f(n)) \rfloor} \leqslant 2^{\lg(f(n))} = f(n)$.

Now, let g be a function as described above. We then have, for sufficiently large n, $g(n) \leqslant \frac{1}{2} f(n) = 2^{\lg(f(n))}/2 = 2^{\lg(f(n))-1} < 2^{\lfloor \lg(f(n)) \rfloor} = 2^{a(n)}$. But it can be shown, using Theorem 5.5 with $S = \{1, 3, 5, \ldots, 2a(n) - 1\}$, $a = 0$ and $b = 1$, that any $\mathscr{A}\mathscr{M}$-program that accepts L must have order $|M_n| \geqslant 2^{a(n)}$.

Let n be a "sufficiently large integer" and let $T_1, T_2 \subseteq S$ such that $T_1 \neq T_2$. Without loss of generality, let $J \in T_1$ but $J \notin T_2$. Let $X \in \Sigma^n$ such that

$$X_i = \begin{cases} 1 & \text{if } i-1 \in T_1, \\ 0 & \text{if not.} \end{cases}$$

One can easily see that $X' = X_{T_1 \leftarrow 1, S \setminus T_1 \leftarrow 0}$ has, in each "Parity_2 component", two 1's or no 1's; so, $X' \in L$.

Let $X'' = X_{T_2 \leftarrow 1, S_n \setminus T_2 \leftarrow 0}$. In the "$\text{Parity}_2$ component" that includes position J, X'' has only one 1; so, $X'' \notin L$.

We can then conclude that any $\mathscr{A}\mathscr{M}$-program accepting L requires $|M_n| \geqslant 2^{|S|} = 2^{a(n)}$. Since $g(n) < 2^{a(n)}$, an $\mathscr{A}\mathscr{M}$-program of order $g(n)$ cannot accept L. □

Corollary 5.9. *The classes of languages defined with subexponential order Abelian monoids (i.e. SE-$\mathscr{A}\mathscr{M}$-programs, SE-$\mathscr{A}\mathscr{G}$-programs, SE-$\mathscr{C}\mathscr{M}$-programs and SE-$\mathscr{C}\mathscr{G}$-programs) are not included in any of the classes defined with polynomial-order abelian monoids (i.e. P-$\mathscr{A}\mathscr{M}$-programs, P-$\mathscr{A}\mathscr{G}$-programs, P-$\mathscr{C}\mathscr{M}$-programs and P-$\mathscr{C}\mathscr{G}$-programs).*

Proof. It suffices to show there is a language accepted by a SE-$\mathscr{C}\mathscr{G}$-program that is not accepted by any P-$\mathscr{A}\mathscr{M}$-program. Take $f(n) = \lfloor 2^{n/[\lg^2(n) + \lg(n)]} \rfloor$ (for $n \geqslant 2$). It is easy to see that $f(n) \leqslant 2^{n/2}$ when n is large. According to Theorem 5.8, there exists a language L that can be accepted by an $\mathscr{A}\mathscr{G}$-program of order $f(n)$ such that no $\mathscr{A}\mathscr{M}$-program of order $g(n)$ can accept it if $g(n) \leqslant \frac{1}{2} f(n)$ for large n. Of course, all P-$\mathscr{A}\mathscr{M}$-programs have that last property since $\frac{1}{2} f(n)$ is super-polynomial; so, they cannot accept L.

We still have to show that L can be accepted by a SE-$\mathcal{CG}$-program. We know it can be accepted by an $\mathcal{AG}$-program of order $\lfloor 2^{n/\lceil \lg^2(n)+\lg(n)\rceil} \rfloor$; so, we use Lemma 5.3 to accept it with a $\mathcal{CG}$-program of order $\lfloor 2^{n/\lceil \lg^2(n)+\lg(n)\rceil} \rfloor^{\lg(n)+1} \leqslant 2^{n/\lg(n)}$, which is subexponential. □

Combining Corollaries 5.2, 5.4, 5.9 and 5.7 along with trivial inclusions, we have proved Theorem 5.10.

Theorem 5.10. *The relationships between the classes of languages defined by $\mathcal{V}$-programs over polynomial-order and subexponential-order cyclic group, cyclic monoid, abelian group and abelian monoid families are given by Table* 1.

In this table, there are still some relations that are not strict: it is not known if SE-$\mathcal{AG}$-programs are more powerful than SE-$\mathcal{CG}$-programs (nor if P-$\mathcal{AG}$-programs are more powerful than P-$\mathcal{CG}$-programs). We conjecture, though, that both these inclusions are strict.

The complexity classes that arise from P-$\mathcal{V}$-programs and SE-$\mathcal{V}$-programs over abelian monoids are quite unusual. As we have seen, P-$\mathcal{AG}$-programs can accept any symmetric language but cannot accept a simple language like $(00)^*1\{0,1\}^*$. What can $\mathcal{AM}$-programs do? Consider the language C_{01} of all binary strings having 01 as substring. Since the instructions of a program over an abelian monoid commute, we expect to have difficulty recognizing that a 0 precedes a 1. Yet Klapper and Longpré [21] proposed an order-n^2 abelian monoid family recognizing C_{01} and more surprisingly Anderson and Barrington [1] obtained an order-n^3 cyclic group family recognizing C_{01}. We are able to completely characterize the polynomial-order abelian monoid languages defined by the presence or absence of a substring:

Theorem 5.11. *For any one of the* 8 *families of abelian monoids mentioned in Theorem* 5.10, *the language $C_w = \Sigma^* w \Sigma^*$ over finite alphabet Σ can be accepted by a program over this family iff w and Σ respect the conditions summarized in Table* 2.

Table 1
Inclusion relations between classes ($A \perp B$ means $A \nsubseteq B$ and $B \nsubseteq A$)

		SE- $\mathcal{AM}$	SE- $\mathcal{AG}$	SE- $\mathcal{CM}$	SE- $\mathcal{CG}$	P- $\mathcal{AM}$	P- $\mathcal{AG}$	P- $\mathcal{CM}$	P- $\mathcal{CG}$
SE-	$\mathcal{AM}$	$=$							
	$\mathcal{AG}$	$\subset$	$=$						
	$\mathcal{CM}$	$\subset$	$\subseteq$	$=$					
	$\mathcal{CG}$	$\subset$	$\subseteq$	$=$	$=$				
P-	$\mathcal{AM}$	$\subset$	$\perp$	$\perp$	$\perp$	$=$			
	$\mathcal{AG}$	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$	$=$		
	$\mathcal{CM}$	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$	$\subseteq$	$=$	
	$\mathcal{CG}$	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$	$\subseteq$	$=$	$=$

Table 2
Acceptance of C_w by $\mathcal{AM}$-programs ($a, b\in\Sigma$ are distinct)

	$w\in\{\varepsilon\}\cup\Sigma$	$w\in\{ab, ba\}$	other w's
$\|\Sigma\|=1$	Y	–	Y
$\|\Sigma\|=2$	Y	Y	N
$\|\Sigma\|\geqslant 3$	Y	N	N

Proof. (Upper bounds): The only nontrivial upper bound arises when $\Sigma=\{0,1\}$ and $w=01$ (the case $w=10$ is dual). One can take [1] $\Pi_n=(Z_{n+1}\times Z_{n(n+1)/2+1}, T_n, F_n)$ with (i, f_i) the instruction at the ith child of the root of T_n, where

$$f_i(x)=\begin{cases}(0,0) & \text{if } x=0,\\ (1,i) & \text{if } x=1,\end{cases}$$

for $1\leqslant i\leqslant n$. Then for any input $v\in\Sigma^n$, $\Pi(v)$ describes the number of 1's and the sum of the positions of these 1's within v. This information suffices to determine whether $v\in C_w$ (and, hence, to define F_n appropriately). Techniques similar to those of Lemma 5.3 yield an $O(n^3)$-order family of cyclic groups accepting C_w.

(Lower bounds): Now pick any Σ and $w\in\Sigma^l$ for which Theorem 5.11 claims that accepting C_w requires exponential order. Observe that there must exist an $x\in\Sigma$ and distinct positions c and d within w such that $w_c\neq x\neq w_d$. Rename the elements of Σ such that $x=0$ and $w_c=1$. We will use Theorem 5.5 with $S=\{i\,|\,2l-1\leqslant i\leqslant n-2l+2$ and $i\equiv 0 \pmod{2l-1}\}$, $a=1$ and $b=0$. Let T' and T'' be distinct subsets of S. Without loss of generality, let $I\in S$ such that $I\in T'$ and $I\notin T''$. Consider the word $X=0^{I-c}w0^{n-I+c-l}$. Then $X'=X_{T'\leftarrow 1,\, S\setminus T'\leftarrow 0}$ also contains w because the "w part" of X is not affected by this substitution (X_I becomes a 1, but $X_I=w_c=1$). On the other hand, $X''=X_{T''\leftarrow 1,\, S\setminus T''\leftarrow 0}$ does not contain w because in X'', there are never enough non-0 symbols in a single contiguous block of length l to make w (which contains at least two non-0 symbols). Therefore, $X'\in C_w$ and $X''\notin C_w$, and we can conclude that the order of the program is at least $2^{|S|}\in\Omega(2^{n/(2l-1)})$ which is exponential. □

Hence, Anderson and Barrington hit upon an "anomaly" due to the binary alphabet. What about languages defined by the absence or presence of a substring that can appear nonconsecutively? (Note that the language C_{01} above can be so defined.) The next theorem shows that $\mathcal{AM}$-programs are a little better at this than they are with consecutive substrings, but still the binary alphabet seems to be easier to deal with:

Theorem 5.12. *The ability for a polynomial-order $\mathcal{V}$-program over abelian groups or over abelian monoids to accept the language $L_w=\Sigma^*w_1\Sigma^*w_2\ldots\Sigma^*w_l\Sigma^*$ with each $w_i\in\Sigma$ depends on $w=w_1w_2\ldots w_l$ and $|\Sigma|$; relevent results are summarized in Tables 3 and 4.*

Table 3
Acceptance of L_w by a P-$\mathscr{AG}$-program ($a \in \Sigma$)

	$w \in a^*$	other w's
$\lvert\Sigma\rvert = 1$	Y	–
$\lvert\Sigma\rvert = 2$	Y	?
$\lvert\Sigma\rvert \geqslant 3$	Y	N

Table 4
Acceptance of L_w by a P-$\mathscr{AM}$-program ($a, b, c \in \Sigma$ are all distinct)

	$w \in a^*$	$w \in a^+b^+$	$w \in a^+b^+c^+$	other w's
$\lvert\Sigma\rvert = 1$	Y	–	–	–
$\lvert\Sigma\rvert = 2$	Y	Y	–	?
$\lvert\Sigma\rvert = 3$	Y	Y	?	N
$\lvert\Sigma\rvert \geqslant 4$	Y	Y	N	N

Proof. (Upper bounds): The nontrivial upper bound occurs when $w \in a^+b^+$. Let $w = a^ib^j$. To recognize L_w in this case, we use monoids $MIN_{n,i} \times MAX_{n,j}$, where $MIN_{n,i} = \{S \subseteq \{1,2,\ldots,n\} \mid |S| \leqslant i\}$ with operation $S \cdot T =$ the set containing the i least elements of $S \cup T$ (or $S \cup T$ if there are less than i elements in it). $MAX_{n,j}$ is defined analogously. We can use these monoids to memorize the positions of the first i a's and of the last j b's. Comparing the position of the last of the first a's with the first of the last b's is enough to determine if the input contains a^ib^j nonconsecutively.

(Lower bounds): Let $w \in \Sigma^l$ be not of the form a^*, with $|\Sigma| \geqslant 3$. Let c be the first position in w such that $w_c \neq w_1$. Rename the elements of Σ such that $w_1 = 0$, $w_c = 1$ and $2 \in \Sigma$. To prove that L_w cannot be accepted by a P-$\mathscr{AG}$-program, we use Theorem 5.6 with $S = \{i \mid l \leqslant i \leqslant n - l + 1 \text{ and } i \equiv 0 \pmod{l}\}$, $a = 1$ and $b = 2$. Let T' and T'' be distinct subsets of S with empty intersection. Let m be the greatest element in $T' \cup T''$ and suppose without loss of generality that $m \in T'$. Consider $Y = 2^{m-c}w2^{n-m-l+c}$. Then $Y' = Y_{T' \leftarrow 1,\, T'' \leftarrow 2}$ contains $w_1, w_2, \ldots, w_l$ because this substitution does not affect the "w part" in Y (Y_m becomes a 1 but $Y_m = w_c = 1$). But $Y'' = Y_{T' \leftarrow 2,\, T'' \leftarrow 1}$ does not contain $w_1, w_2, \ldots, w_l$ because in Y'', the first $w_1 (= 0)$ is in position $m - c + 1$ and there are not enough w_c's $(= 1)$ after that to find $w_2, \ldots, w_l$. Therefore, $Y' \in L$ and $Y'' \notin L$, and we can conclude that $|G_n| \geqslant 2^{|S|}$, which is not polynomial.

Now pick any Σ and $w \in \Sigma^l$ corresponding to a N entry in Table 4. Then there exist three positions $i < j < k$ in w such that $w_i \neq w_j$, $w_j \neq w_k$ and $\{w_i, w_j, w_k\} \neq \Sigma$. To choose i, j, k as above, first choose any j that works, then choose the minimal i that works for j, and finally the maximal k that works for i and j. Rename the elements of Σ such that $w_i = 0$, $w_j = 1$ and $2 \in \Sigma \setminus \{w_i, w_j, w_k\}$. We use Theorem 5.5 with $S = \{j' \mid j' \equiv j \pmod{l}\}$, $a = 1$ and $b = 2$. Let T' and T'' be distinct subsets of S. Without loss of generality, let

$J \in T'$ and $J \notin T''$. Consider $X = 2^{J-j} w 2^{n-J-l+j}$. Then $X' = X_{T' \leftarrow 1, S \setminus T' \leftarrow 2}$ contains $w_1, w_2, \ldots, w_l$ because this substitution does not affect the "w part" of X (X_J becomes a 1 but $X_J = w_j = 1$). But in $X'' = X_{T'' \leftarrow 1, S \setminus T'' \leftarrow 2}$, the distance between the first w_i ($=0$) and the last w_k ($\neq 1$, $\neq 2$) is exactly $k-i$ while one of the symbols in between ($X_J = 1$) has been changed (to $X''_J = 2$). Because of the minimality of i and the maximality of k, X'' cannot contain $w_1, w_2, \ldots, w_l$. Therefore, $X' \in L_w$ and $X'' \notin L_w$, and we can conclude that $|G_n| \geqslant 2^{|S|}$, which is not polynomial. $\square$

It is quite possible that all L_w languages can be accepted when $|\Sigma| = 2$ with P-$\mathscr{A}\mathscr{G}$-programs, but the only cases for which we can prove this are $w \in a^*$ and $w = ab$. With $\mathscr{A}\mathscr{M}$-programs, though, we are able to accept L_w when $w \in a^*b^*a^*$ or $w \in a^*bab^*$. Theorems 5.5 and 5.6 (in their more general form not stated in this article) cannot help us with those languages since they can only give polynomial lower bounds on the order of the programs that accept them.

6. Conclusion

A question often heard about Barrington's appealing M-program model was whether it could find uses beyond the complexity level of the important but restricted class NC^1. We showed here that the model extends to capture LOGCFL (Theorem 3.3), NL (Theorem 3.9) and TC^0 (Theorem 3.8) as polynomial-length computations over fixed structures. This holds under the same tight uniformity notion as that used to capture NC^1 and its subclasses, offering a new perspective on the relationships between all these subclasses of LOGCFL. The existing structure theory of finite monoids gives insight into NC^1, and any similar structure theory of finite groupoids would give similar insight into LOGCFL. No elegant theory of finite groupoids seems at hand, however. Since groupoids are so closely related to context-free grammars, perhaps elements of such a theory are already part of the theory of context-free grammars.

For their part, programs over monoids growing with the input size should be viewed as an intermediate step between bounded-width and arbitrary width polynomial-order branching programs (describing NC^1 and L, respectively). Since the combinatorics involved in the relationship between NC^1 and L are poorly understood, it may prove useful to parametrize the (presumed) "gap" between these classes using well-studied criteria like, here, algebraic properties of the structure over which computation is defined. The abelian requirement is, to be sure, too restrictive. The combinatorics underlying the behavior of $\mathscr{V}$-programs for $\mathscr{V}$ an abelian monoid sequence nonetheless offer a reasonable challenge as we have seen. Indeed even in this seemingly simple context open questions remain with regard to the exact relationship between the computational power of cyclic groups and that of abelian groups, and with regard to the ability of abelian groups and monoids to recognize languages of the form $\Sigma^* w_1 \Sigma^* w_2 \ldots \Sigma^* w_l \Sigma^*$.

However, the most natural open question concerning polynomial-length programs over growing monoids is whether these programs over polynomially growing group (or monoid) sequences characterize the class L (L clearly contains all languages accepted by such programs which are DLOGTIME-uniform). Problems complete for L include the word problem over the symmetric group S_n and several permutation group problems [13, 19]. Perhaps one could combine a "constant-order nonsolvable part" (handling an NC^1-reduction, as was done in Theorem 3.3) and a "polynomial-order part" (representing the computation of an L-complete problem) to obtain a polynomial-order group sequence handling all logspace computations. Then the situation already encountered at the NC^1-level, where the full complexity seems to appear only in the presence of nonsolvable groups [2], may repeat itself at the level of the class L. Could we then hope to tie the relationship between NC^1 and its subclasses to the relationship between L and NC^1? The experience gained here in the restricted setting of abelian monoids should in any case be helpful in tackling the case of nilpotent group families (which are reasonably well understood in the "old" M-program model setting [4, 33]) and then the case of solvable monoids (which play a crucial role in M-program characterizations of NC^1-subclasses [5, 25, 23]).

Several other questions arise. Can we find a fixed groupoid G for which polynomial-length (flat or arbitrary) G-programs characterize L? Such a groupoid could perhaps be constructed from a context-free language complete for the class L. Can we find a fixed groupoid G for which polynomial-length (flat or arbitrary) G-programs characterize the class LOGDCFL of languages reducible in log space to a deterministic context-free language [32]? Can we always transform a structured program into a flat one in a uniform way? Can we state a general theorem which would relate the complexity of a context-free language to the complexities of the word problems over some "canonical" groupoid recognizing this language? Can we hope that the characterizations discussed here might help understanding the real separations between subclasses of LOGCFL? Can we capture other natural complexity classes by considering non-polynomial-length or nonpolynomial-order $\mathscr{V}$-programs?

Acknowledgment

We thank Dave Barrington, Paul Beame, Sheila Greibach, Luc Longpré, Yves Marcoux, and Denis Thérien for helpful discussions. We are also grateful to Larry Ruzzo, who pointed out Fact 2.3; this led to Theorem 3.9. Finally we thank the anonymous referees for a very careful reading of our manuscript.

References

[1] R. Anderson and D.A.M. Barrington, private communication, 1989.

[2] D.A. Barrington, Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1, in: *Proc. 18th ACM Symp. on the Theory of Computing* (1986) 1–5; *J. Comput. System Sci.* **38** (1) (1989) 150–164.

[3] D. Mix Barrington, N. Immerman and H. Straubing, On uniformity within NC^1, in: *Proc. 3rd Ann. Conf. on the Structure in Complexity Theory* (IEEE Computer Society Press, 1988) 47–59; *J. Comput. System Sci.* **41** (3) (1990) 274–306.

[4] D.A. Barrington and D. Thérien, Non-uniform automata over groups, in: *Proc. 14th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 267 (Springer, Berlin, 1987) 163–173; D.A. Barrington, H. Straubing and D. Thérien, Non-uniform automata over groups, *Inform. and Comput.*, in press.

[5] D. Barrington and D. Thérien, Finite monoids and the fine structure of NC^1, *J. Assoc. Comput. Mach.* **35** (4) (1988) 941–952.

[6] A. Borodin, D. Dolev, F. Fich and W. Paul, Bounds for width-two branching programs, in: *Proc. 15th ACM Symp. on the Theory of Computing* (1983) 87–93.

[7] S.R. Buss, The boolean formula value problem is in ALOGTIME, in: *Proc. 19th ACM Symp. on the Theory of Computing* (1987) 123–131.

[8] S.R. Buss, S. Cook, A. Gupta and V. Ramachandran, An optimal parallel algorithm for formula evaluation, preprint, 1989; *SIAM J. Comput.* to appear.

[9] A. Chandra, M. Furst and R. Lipton, Multi-party protocols, in: *Proc. 15th ACM Symp. on the Theory of Computing* (1983) 94–99.

[10] A. Chandra, L. Stockmeyer and U. Vishkin, Constant depth reducibility, *SIAM J. Comput.* **13** (2) (1984) 423–439.

[11] A. Cobham, The recognition problem for the set of perfect squares, Research Paper RC-1704, IBM Watson Research Center, Yorktown Heights, NY, 1966.

[12] S.A. Cook, A taxonomy of problems with fast parallel algorithms, *Inform. and Comput.* **64** (1985) 2–22.

[13] S.A. Cook and P. McKenzie, Problems complete for deterministic logarithmic space, *J. Algorithms* **8** (1987) 385–394.

[14] M.L. Furst, J.B. Saxe and M. Sipser, Parity, circuits, and the polynomial-time hierarchy, in: *Proc. 22nd IEEE Symp. on the Foundations of Computer Science* (1981) 260–270; *Math. Systems Theory* **17** (1984) 13–27.

[15] S. Greibach, The hardest context-free language, *SIAM J. Comput.* **2** (4) (1973) 304–310.

[16] M.A. Harrison, *Introduction to Formal Language Theory* (Addison-Wesley, Reading, MA, 1978).

[17] I. Herstein, *Topics in Algebra* (Wiley, New York, 2nd ed., 1975).

[18] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).

[19] N. Immerman and S. Landau, The complexity of iterated multiplication, in: *Proc. 4th Ann. Conf. on the Structure in Complexity Theory* (IEEE Computer Society Press, 1989) 104–111.

[20] R. Karp and R. Lipton, Turing machines that take advice, *Enseign. Math.* **28** (1982) 191–209.

[21] A. Klapper and L. Longpré, private communication, 1988.

[22] P. McKenzie, P. Péladeau and Thérien, NC^1: the automata-theoretic viewpoint, *Computational Complexity* **1** (1991) 330–359.

[23] P. McKenzie and D. Thérien, Automata theory meets circuit complexity, in: *Proc. 16th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 372 (Springer, Berlin, 1989) 589–602.

[24] I. Parberry and G. Schnitger, Parallel computation with threshold functions, in: *Proc. 1st Structure in Complexity Theory Conf.* Lecture Notes in Computer Science, Vol. 223 (Springer, Berlin, 1986) 272–290; *J. Comput. System Sci.* **36** (3) (1988) 278–302.

[25] P. Péladeau, Classes of boolean circuits and varieties of finite monoids, LITP Tech. Report 89-25, Université Paris 7, 1989.

[26] J.-E. Pin, *Variétés de languages formels* (Masson, Paris, 1984); *Varieties of Formal Languages* (Plenum, New York, 1986).

[27] N. Pippenger, On simultaneous resource bounds, in: *Proc. 20th IEEE Symp. on the Foundations of Computer Science* (1979) 307–311.

[28] N. Pippenger and M.J Fischer, Relations among complexity measures, *J. Assoc. Comput. Mach.* **26** (2) (1979) 361–381.

[29] W.L. Ruzzo, Tree-size bounded alternation, *J. Comput. System Sci.* **21** (1980) 218–235.

[30] W.L. Ruzzo, On uniform circuit complexity, *J. Comput. System Sci.* **22** (3) (1981) 365–383.

[31] I.H. Sudburough, A note on tape-bounded complexity classes and linear context-free languages, *J. ACM* **22** (4) (1975) 499–500.

[32] I. Sudburough, On the tape complexity of deterministic context-free languages, *J. ACM* **25** (3) (1978) 405–414.

[33] D. Thérien and P. Péladeau, Sur les Langages Reconnus par des Groupes Nilpotents, Comptesrendus de l'Académie des Sciences de Paris, t. 306, Série I (1988) 93–95.

[34] L.G. Valiant, General context-free recognition in less than cubic time, *J. Comput. System Sci.* **10** (2) (1975) 308–315.

[35] H. Venkateswaran, Properties that characterize LOGCFL, in: *Proc. 19th ACM Symp. on the Theory of Computing* (1987) 141–150.

[36] I. Wegener, *The Complexity of Boolean Functions* (Wiley, New York, 1987).