National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Finite Groupoids and their Applications to Computational Complexity

François Lemieux

School of Computer Science
McGill University, Montréal
May 1996

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfilment of the requirements of the degree of Doctor of Philosophy

ISBN 0-612-12411-8

Abstract

In our Master thesis the notions of recognition by semigroups and by programs over semigroups were extended to groupoids. As a consequence of this transformation, we obtained context-free languages instead of regular with recognition by groupoids, and we obtained SAC^1 instead of NC^1 with recognition by programs over groupoids. In this thesis, we continue the investigation of the computational power of finite groupoids.

We consider different restrictions on the original model. We examine the effect of restricting the kind of groupoids used, the way parentheses are placed, and we distinguish between the case where parenthesis are explicitly given and the case where they are guessed nondeterministically.

We introduce the notions of linear recognition by groupoids and by programs over groupoids. This leads to new characterizations of linear context-free languages and NL. We also prove that the algebraic structure of finite groupoids induces a strict hierarchy on the classes of languages they linearly recognized.

We investigate the classes obtained when the groupoids are restricted to be quasigroups (i.e. the multiplication table forms a latin square). We prove that languages recognized by quasigroups are regular and that programs over quasigroups characterize NC^1.

We also consider the problem of evaluating a well-parenthesized expression over a finite loop (a quasigroup with an identity). This problem is in NC^1 for any finite loop, and we give algebraic conditions for its completeness. In particular, we prove that it is sufficient that the loop be nonsolvable, extending a well-known theorem of Barrington.

Finally, we consider programs where the groupoids are allowed to grow with the input length. We study the relationship between these programs and more classical models of computation like Turing machines, pushdown automata, and owner-read owner-write PRAM. As a consequence, we find a restriction on Boolean circuits that has some interesting properties. In particular, circuits that characterize NP and NL are shown to correspond, in presence of our restriction, to P and L respectively.

Résumé

Cette thèse est la continuation des travaux entrepris au cours de nos études de maîtrise. Les notions de reconnaissance par un semigroupe et par un programme sur un semigroupe avaient alors été généralisées aux groupoïdes. Ici nous poursuivons les recherches sur la puissance de calcul des groupoïdes finis.

Nous considérons différentes restrictions du modèle original. Nous examinons les conséquences de restreindre la classe de groupoïdes utilisés et la façon de disposer les parenthèses. Nous distinguons le cas où les parenthèses sont données explicitement avec le programme de celui où elles sont placées de façon non déterministe.

Nous introduisons les notions de reconnaissance linéaire par un groupoïde et par un programme sur un groupoïde. Nous montrons que cela permet de donner une nouvelle caractérisation des langages hors-contextes linéaires ainsi que de la classe de complexité NL. Nous prouvons aussi que la structure algébrique des groupoïdes finis induit une hiérarchie stricte parmis les langages linéaires.

Nous étudions les classes de langages obtenues lorsque les groupoïdes sont restreints à être des quasigroupes (c'est-à-dire que leur table de multiplication forme un carré latin). Nous prouvons que les langages reconnus par un quasigroupe sont réguliers et que les programmes sur quasigroupes reconnaissent précisément la classe NC^1.

Nous considérons aussi le problème d'évaluer une expression avec parenthèses sur une boucle finie (une boucle est un quasigroupe avec un élément neutre). Ce problème est dans NC^1 quelle que soit la boucle et nous donnons des conditions algébriques pour qu'il soit complet. En particulier, nous mon-

trons qu'il est suffisant que la boucle soit non résoluble, généralisant ainsi un théorème bien connu de Barrington

Finalement nous considérons les programmes où le groupoïde utilisé peut croître avec la longueur de l'entrée. Nous étudions les relations existant entre ces programmes et des modèles de calcul plus classiques comme les machines de Turing, les automates à pile et des modèles parallèles de type PRAM. Cela nous permet de définir une restriction sur les circuits booléens ayant d'intéressantes propriétés. En particulier, nous montrons que les circuits caractérisant les classes NP et NL reconnaissent respectivement P et L en présence de cette restriction.

Acknowledgments

I wish to express my deep gratitude to my supervisor Denis Thérien. I benefited a lot from his rich experience, ideas, and advice. I highly appreciated his generosity and human qualities. What I learned from him goes far beyond the scope of computer science.

I am also grateful to my colleague and friend Alexis Maciel. His comments and suggestions have been useful and greatly appreciated. The time I spent at McGill would have not been so pleasant without his presence.

Many thanks to Pierre McKenzie and Martin Beaudry for helpful discussions. A particular thanks to Martin Beaudry for inviting me while he was on leave at Universität Würzburg.

I am very grateful to Hervé Caussinus for many discussions we had. His help was invaluable. Some parts of Chapter 4 have been done in collaboration with him and the pictures have been made by him.

I would like to thank Anca Muscholl, Klaus-Jörn Lange, and particularly David Mix Barrington for helpful discussions and encouragement. I am grateful to Anca Muscholl for inviting me at Universität Stuttgart. My thanks to Eric Allender for his helpful comments on this thesis.

I have appreciated the pleasant atmosphere at the School of Computer Science of McGill University and I would like to thank the academic, technical and administrative staff. A particular thanks to Lorraine Harper who facilitated my life so much.

Many thanks to Denis Thérien, Pierre McKenzie, and Martin Beaudry for financial support. I also received financial assistance from FCAR and NSERC.

Je voudrait aussi remercier ma famille pour tout le support et l'encoura-

gement qu'elle m'a donnés. Merci aussi à Marie-Josée pour m'avoir aidé à supporter les difficiles derniers mois de la rédaction. Finalement, je voudrais exprimer toute ma gratitude à mes parents Georges et Pierrette. Par leur infinie générosité et leurs encouragements constants, il ont une grande part de responsabilité dans la réalisation de cette thèse.

À mes parents,

Georges et Pierrette

Contents

List of Symbols

	Section
AuxDPDA, AuxNPDA	1.2.3
DTIME, DSPACE, DTIME-SPACE	1.3
NTIME, NSPACE, NTIME-SPACE	1.3
P, L, NP, NL	1.3.1
NC^1, ACC^0, AC^0, MOD_q, MAJ	1.3.2
LOGCFL, LOGDCFL, SAC^1	1.3.3
$A^{(*)}$, $A^{(+)}$, G^0, G^1	2.1
$\mathcal{Z}$, $G_1 \prec G_2$, $\langle S \rangle$	2.1.1
$R(x)$, $L(x)$, $\mathcal{M}(G)$	2.1.2
$G(w)$, $W(G, A, F)$	2.2
$u \sim_L v$, CC^0	2.2.1
$\mathcal{P}(G)$	2.3
yield(T)	2.4
$t \cdot^a t'$	2.4.1
LTR, RD_k	2.5
A^{LIN}, G_{LIN}	3.1
$\tilde{w}$	3.3
$\mathcal{J}(L)$, $T(x)$, $R(x,y)$, $L(x,y)$	4.1.2
$[x,y]$, $[x,y,z]$, $S_{(N,L)}$, (N,L), $N(\mathcal{J})$	4.1.3
$a/b = c$, $a \circ b = c$, $a \not/ b = c$, $a \backslash\!\!\!/ b = c$	4.1.6
$T_1 \cdot T_2$, $v(T)$	4.2
semi-bounded ATM, skew ATM	5.3
OROW-PRAM, CROW-PRAM	5.6

Chapter 1

Introduction

The theory of complexity is concerned with the classification of problems in terms of their computational difficulty. In particular, we are interested in proving that specific problems cannot be solved by a computer using some bounded amount of resources. This aspect of complexity theory is well illustrated by the P vs NP question.

NP is the class of languages that can be recognized by a nondeterministic Turing machine in polynomial time. P is defined similarly by restricting the Turing machines to be deterministic.

The problem of determining if P and NP are equal appeared soon in the development of computer science (see [38]). However, it remains unsolved in spite of thirty years of intensive research. Since this question seems too difficult to be attacked directly, people have turned to simpler problems, following two opposite directions. The first one consists in separating P from a complexity class that would be larger than NP: for example PSPACE, the class of problems solvable by a Turing machine using polynomial space. In the other direction, people try to separate a subclass of P from NP: for example, the subclass NL of problems solvable on a nondeterministic Turing machine using logarithmic space (observe that none of these examples has been settled yet).

This research has resulted in the definition of a large number of complexity classes, and complexity theory has evolved into an investigation of the lattice structure of these classes.

Two complexity classes can be compared more easily if they are defined

with the same model of computation. However, there is no single type of machine that can be used to define all classes. For that reason, it is important to characterize each complexity class using various computational models.

Programs over semigroups, introduced in [5], are models of computation that rely on finite semigroups, i.e. finite sets with an associative binary operation. Using these programs, Barrington and Thérien characterized different complexity classes simply by varying the type of semigroups involved (see [8]). Their results not only give new characterizations of complexity classes, they establish a close relationship between the algebraic theory of semigroups and complexity theory.

At the heart of this research there was a well known result due to Kleene (e.g. see [56]) relating finite semigroups to regular languages. In [12] (see also [48]) we investigated the effect of replacing semigroups by their nonassociative analogues, called groupoids. We proved a generalization of Kleene's theorem giving a natural correspondence between finite groupoids and context-free languages. We also showed that using programs over groupoids yields a characterization of many complexity classes that could not be captured with semigroups.

These results suggest that the relationship between finite groupoids and complexity theory deserves to be investigated further. This is the subject of this thesis.

Essential definitions and background are given in the rest of this chapter whose last section expose the contributions made by this thesis. In Chapter 2, we define groupoids and discuss some of their algebraic properties. Then, we introduce recognition by groupoids and by programs over groupoids. We examine two important parameters: the algebraic structure of groupoids and the domain of parenthesization. In Chapter 3, we study a natural restriction of the domain of parenthesization and we show that this is equivalent, from a computational point of view, to considering only those groupoids satisfying certain algebraic conditions. We show how these restricted groupoids are related with rational transducers and we give a strict hierarchy of linear lan-

guages using a method that relies on finite groupoids. Chapter 4 is devoted to quasigroups which are those finite groupoids whose multiplication table forms a latin square. In Chapter 5, we generalize the above ideas by considering families of groupoids. Instead of using a fixed groupoid we allow groupoids to grow with the input length. We study how this growth influences the kind of languages that can be captured. Finally, we conclude in Chapter 6 with some questions raised by this thesis.

1.1 Languages and Reductions

With any language $L \subseteq A^*$ we associate a decision problem. This is the problem of determining, given a word $x \in A^*$, if x belongs to L or not. We say that L is *recognized* by an algorithm M whenever M correctly solves the above problem.

Let A and B be finite alphabets, and let $L_A \subseteq A^*$ and $L_B \subseteq B^*$ be two languages. L_A is said to be reducible to L_B, denoted $L_A \leq L_B$, whenever there exists a *reduction function* $f : A^* \to B^*$ such that for any $x \in A^*$, $x \in L_A$ if and only if $f(x) \in L_B$.

Reductions have a fundamental role in complexity theory (e.g. see [33]). For example, suppose that the above reduction function f is recursive. Then, given an algorithm M for L_B, one can construct an algorithm M' for L_A working as follows. First, on input $x \in A^*$, M' computes $y = f(x)$. Then, M' simulates algorithm M on input y and accepts if and only if M accepts. Suppose moreover that the complexity of computing f is negligible compared to the difficulty of recognizing L_B. Then, this implies that recognizing L_A is no more difficult than recognizing L_B (justifying the notation).

Observe that it is very important to put a restriction on the complexity of computing f. Otherwise, nothing could be said on the relative complexity of recognizing L_A and L_B. In this thesis we will use two kinds of reductions. When the reduction function f is computable by a deterministic Turing machine using logarithmic space, then f is said to be a *logspace reduction*.

Logspace reductions are too powerful for dealing with very small complexity classes. Thus, we will also use a weaker type of reduction called *dlogtime-uniform projection.*

Given any alphabet A, we denote the length of a word $w \in A^*$ with $|w|$. A function $f : A^* \to B^*$ is called a *projection* (see [79, 42]) if for any $x \in A^n$, f satisfies the following two conditions. First, the length of $f(x)$ depends only on the length of x. Moreover, for any $i \in \{1, \ldots, |f(x)|\}$ there exists $j \in \{1, \ldots, |x|\}$ such that the i^{th} symbol in $f(x)$ depends only of the j^{th} symbol in x. The first condition makes possible the definition of the *length of a projection* f as the function mapping the length of its input to the length of its image. In the following we will consider only polynomial length projections.

Projections are still too powerful and we must restrict them further. This is because one can see a projection f as a family of functions $f_n : A^n \to B^*$, one for each input length, and if no uniformity condition is imposed on the f_i's, then nothing guarantees that f will be computable at all.

First, we define a *direct-access* Turing machine as a Turing machine with a read-only input tape, a constant number of working tapes and a special tape called the address tape. The content of the address tape denotes a position in the input. At any step, the machine moves its input head to the position written on the address tape (we assume that the machine can determine if this position is larger than the length of the input). Since direct-access Turing machines can check any bit of the input in logarithmic time, then it makes sense to speak of languages recognized in logarithmic time by such machines (see [6] for more details).

A projection is *dlogtime-uniform* (see [6]) if there exists a direct-access deterministic Turing machine M such that for any $x \in A^*$, $i \geq 1$ and any $b \in B$, M determines in logarithmic time, on input (x, i, b), whether the i^{th} symbol of $f(x)$ is b.

1.2 Models of Computation

Models of computation can be divided in two categories. First, there are those models that, like Turing machines, can be described with a finite number of symbols. They are said to be intrinsically uniform. The other category consists of those models defined as an infinite family $\{M_1, M_2, \ldots\}$ of machines, the n^{th} machine M_n dealing with inputs of length n.

Nonuniform models are very powerful and can compute even nonrecursive function, which is sometime undesirable. For this reason we must impose some uniformity conditions on machines in a given family. For example, we can ask for the existence of a Turing machine that, on input n, constructs the n^{th} machine of the family.

In the following, we will introduce three models of computation that, like Turing machines, are standard in complexity theory. The first two models are nonuniform: the oldest one is the Boolean circuit introduced by Shannon in [67, 68], and the other model is the branching program of Lee [47]. Finally, we also examine the auxiliary pushdown automaton of Cook [23].

1.2.1 Boolean circuits

A *Boolean circuit* is a finite directed acyclic graph that contains three different types of vertices: the input gates are those vertices having indegree 0; the *output gates* are those vertices having outdegree 0; other vertices are called *inner gates*. Inner gates and output gates are labeled with the name of a function from $\{0,1\}^*$ to $\{0,1\}$.

Unless otherwise specified, we use AND and OR for labeling inner gates and output gates, and each input gate is labeled with the name of an input variable or its complement. We say that a gate g' is an input to a gate g if there is an edge from g' to g. We suppose that gates are ordered in some way such that it makes sense to talk, for example, of the first input gate, the fifth output gate or the second input of an inner gate.

A Boolean circuit C with n inputs and m outputs computes a function

$\phi : \{0,1\}^n \to \{0,1\}^m$ as follows. Given $w = a_1 \cdots a_n \in \{0,1\}^n$, we will recursively assign a value to the gates of C, and the value of the j^{th} bit of $\phi(w)$ will be the value of the j^{th} output gate. An input gate g labeled with the Boolean variable x_i has value a_i. If it is labeled with $\bar{x}_i$, then it has value $1 - a_i$. Suppose now that g is not an input gate, let the indegree of g be k and let $g_1, \ldots, g_k$ be the inputs of g. Then, the value of g is $f(v_1 \cdots v_k)$ where f is the label function of g and v_i is the value of g_i, $1 \leq i \leq k$.

In order to compute a function $f : \{0,1\}^* \to \{0,1\}$ we need a family of circuits, $C = \{C_0, C_1, C_2 \ldots\}$, where C_n is a Boolean circuit with n inputs and 1 output.

The *size* of a Boolean circuit is the number of gates in it, and the *depth* is the maximum length over the paths from an input gate to an output gate. Size and depth can be viewed as function of n, the input size. Thus, we can talk about families of polynomial-size circuits or logarithmic-depth circuits, for example. A circuit is said to have *bounded (unbounded) fan-in* whenever the indegree of the gates is bounded by a constant (unbounded). If only OR gates have unbounded indegree then circuits are said to have *semibounded*[1] fan-in.

Uniformity is not an issue in this thesis. However, we need to talk about it if we want to compare the power of the different models of computation. In particular, it is necessary to define uniformity in the case of Boolean circuits.

Let $C = (C_n)_{n \geq 0}$ be a family of Boolean circuits and consider any numbering of the gates in the circuits. The direct connection language of C (see [63, 6]) is the set of tuples $\langle t, a, b, y \rangle$, where a is the number of a gate in C_n, y is any string of length n, and t either indicates that a is an input gate, in which case b corresponds to the input index labeling a, or t corresponds to the function labeling a and b is the number of a child of a.

Let $\mathcal{D}$ be any complexity class. The family $C = (C_n)_{n \geq 0}$ is said to be $\mathcal{D}$-uniform if there exists a gate numbering for each circuit in C such that the direct connection language belongs to $\mathcal{D}$. We simply say that C is uniform

[1] Other authors have called this *semiunbounded*.

if its direct connection language can be recognized by a deterministic Turing machine in time $O(\log s(n))$, where $s(n)$ is the size of C_n.

1.2.2 Branching programs

Let us fix a finite alphabet A and a nonnegative integer n. A *branching program* B_n for inputs of length n is a finite acyclic directed graph with two distinguished vertices: one is called the *root* and the other is called the *sink*. All vertices are labeled with an index in $\{1, \ldots, n\}$. Each edge is labeled with an element of A. A branching program is called *deterministic* if for any gate g, no two edges going out of g are labeled with the same element of A.

A word $w \in A^n$ is accepted by B_n whenever starting from the root and iterating the following procedure one can reach the sink. The procedure is: let i be the label of the current vertex; follow an edge labeled with the i^{th} symbol of w.

A language $L \subseteq A^*$ is accepted by a family $M = \{B_0, B_1, \ldots\}$ of branching programs if for any $n \geq 0$ and any $w \in A^n$, w belongs to L if and only if w is accepted by B_n.

The size of a branching program B_n is a function mapping n to the number of vertices in B_n. In this work, we will only consider polynomial-size branching programs.

We define a *bounded-width* branching program (see [5]) as a branching program B that forms a rectangular array of nodes with k rows and l columns, for some positive k and l. Every edge in B is directed from left to right, i.e. the descendants of any vertex appears on its right. Hence, the root and the sink lie respectively in the first and the last column. The integer k is the *width* of B and l is its *length*.

1.2.3 Auxiliary pushdown automata

We call *auxiliary pushdown automaton* a Turing machine with a special tape that is used as a stack. We will suppose that such a machine accepts its input

whenever it reaches a final state, but other conditions can also be applied (such as emptying its stack) without loss of generality.

We define the space used by an auxiliary pushdown automaton as the number of cells used on the working tape only, not the input tape nor the special tape.

1.3 Complexity Classes

Various complexity classes can be defined using the different models of computations introduced in the previous section. In this section, we will examine some of those that are of particular interest for this work.

First, we observe that a complexity class can be a class of functions $\mathbb{N} \to \mathbb{N}$ or a class of languages (or Boolean functions). In this thesis we use no special notation to distinguish these two kinds of complexity classes: the context will exclude any ambiguity. For example, any complexity class defined from branching programs or pushdown automata is a class of languages. It is a class of functions if it is defined from multiple-output Boolean circuits.

We write DTIME($t(n)$), DSPACE($s(n)$), and DTIME-SPACE($t(n), s(n)$) to denote the classes of languages recognized by deterministic Turing machines using respectively time $O(t(n))$, space $O(s(n))$, and both time $O(t(n))$ and space $O(s(n))$. NTIME($t(n)$), NSPACE($s(n)$), and NTIME-SPACE($t(n), s(n)$) are defined similarly using nondeterministic Turing machines. We use the same notation to describe the classes of functions computed by these models.

1.3.1 Polynomial time and logarithmic space

Among the most important topics in complexity theory are the questions concerning the relationship between time and space, and between determinism and nondeterminism (see [33, 41]). Here, we define four basic complexity classes that are very good examples to illustrate these questions.

The class P is defined as the set of all languages that can be recognized by a deterministic Turing machine running in polynomial time. The class NP is

defined similarly except that Turing machines are allowed to be nondeterministic.

P has many equivalent definitions. In particular, it can be defined as the class of languages recognized by a uniform family of polynomial-size Boolean circuits [45, 14]. Moreover, it is also equal to the class of languages recognized by an auxiliary pushdown automaton using only logarithmic space [23]. Observe that in the last definition, time is unbounded and the model can be deterministic or not without changing the class of languages defined.

NP also has a characterization in terms of Boolean circuits (see [81]). It is the class of languages recognized by a uniform family of semibounded fan-in exponential-size logarithmic-depth Boolean circuits.

Turning our attention to space complexity, we define the class L as the set of languages recognized by a deterministic Turing machine using only logarithmic space. The class NL is defined similarly, using nondeterministic Turing machines.

L can be defined in terms of branching programs. Specifically, it is the class of languages recognized by a uniform family of polynomial-size deterministic branching programs. In Section 5.5, we will also give a definition of L in terms of Boolean circuits.

NL is equivalent to the class of languages recognized by a polynomial-size family of nondeterministic branching programs. It has also been proved that NL corresponds to those languages that are recognized by a family of polynomial-size *skew circuits*—a skew circuit is a Boolean circuits whose AND gates have at most one input that is not an input gate (see [81] for more details).

Clearly we have $P \subseteq NP$ and $L \subseteq NL$. Furthermore, it not difficult to see that the problem of determining if the sink is accessible in a nondeterministic branching program can be solved with a deterministic Turing machine in a polynomial number of steps (e.g. see [4]). This implies that $NL \subseteq P$. Even if all these inclusions are conjectured to be strict nothing has been proved yet.

1.3.2 Subclasses of L

Boolean circuits are sometime referred to as a parallel model of computation. This comes from results relating the depth in Boolean circuits to the time in models of computation such as alternating Turing machines [20, 63] and parallel random access machines [74]. The fact is that bounded-depth Boolean circuits appear to be very useful for characterizing subclasses of L.

We define NC^1 as the set of languages recognized by a family of bounded fan-in logarithmic-depth polynomial-size Boolean circuits. The class AC^0 is defined as the class of languages recognized by a family of unbounded fan-in constant-depth polynomial-size Boolean circuits.

NC^1 circuits can be evaluated via a depth first search using only logarithmic space. This implies that $NC^1 \subseteq L$. Also, since any AND/OR gate of unbounded fan-in can be expanded into a NC^1 circuit, we have $AC^0 \subseteq NC^1$.

We give for NC^1 two other characterizations. First, it corresponds to those languages recognized by a family of polynomial-size Boolean formulae — a formula is a circuit that is also a tree (see [70, 17]). Moreover, NC^1 has been proved to be equal to the class of languages recognized by a family of polynomial-size constant-width branching programs [5]. Observe that nondeterministic and deterministic constant-width branching programs have the same power.

We introduce two other classes between AC^0 and NC^1. Both of them are defined similarly to AC^0 except that we allow gates in the circuits to be labeled with other functions than AND and OR.

The class ACC^0 (see [8]) corresponds to those languages recognized by a family of polynomial-size constant-depth Boolean circuits where gates are labeled with AND, OR and MOD_q ($q > 1$) which output 1 if and only if the number of 1's in the input is congruent to 0 modulo q. The class TC^0 (see [39]) is defined similarly except that MOD_q gates are replaced by MAJ gates which output 1 whenever at least half of their input bits are 1.

The relations $AC^0 \subseteq ACC^0$ and $AC^0 \subseteq TC^0$ are immediate. Moreover, it

has been proved that the function MAJ can be computed with NC^1 circuits (e.g. see [64]), implying that $TC^0 \subseteq NC^1$. It is not difficult to show that MOD_q functions are in TC^0, and thus $ACC^0 \subseteq TC^0$ (see [31]).

In defining these small complexity classes, researchers were hoping to prove strict inclusions and get more intuition to deal with larger classes. Actually, there have been some remarkable results. In particular, it has been proved that MOD_2 and MAJORITY are not in AC^0(see [31]). As a consequence, AC^0 is strictly included in ACC^0 (see also [1]). It still remains to determine the nature of the relations between ACC^0, TC^0 and NC^1.

1.3.3 Subclasses of P

The last two classes that we want to introduce are subclasses of P. The class LOGCFL is the set of languages logspace reducible to a context-free language, and LOGDCFL is the class of languages logspace reducible to a deterministic context-free language.

LOGCFL (LOGDCFL) is also equal to the set of languages recognized by a nondeterministic (deterministic) auxiliary pushdown automaton running in polynomial time and using logarithmic space (see [75]). Recall that we have mentioned above that determinism has no influence on the class of languages recognized by an auxiliary pushdown machine when time is not limited. But this does not apply here, since the number of steps is bounded by a polynomial.

Another characterization of LOGCFL is given in [81] using Boolean circuits. That is, LOGCFL is equal to SAC^1, the class of languages recognized by a family of semibounded fan-in polynomial-size logarithmic-depth Boolean circuits. This result is particularly interesting in view of the characterization of NP in terms of semibounded fan-in exponential-size logarithmic-depth Boolean circuits.

By definition, we have LOGDCFL $\subseteq$ LOGCFL. The characterizations of LOGDCFL, LOGCFL and P in terms of logspace auxiliary pushdown automata give L $\subseteq$ LOGDCFL, NL $\subseteq$ LOGCFL and LOGCFL $\subseteq$ P.

We summarize in the following diagram the known relations between the

different complexity classes that we have introduced in this section.

$$AC^0 \subset ACC^0 \subseteq TC^0 \subseteq NC^1 \subseteq L \subseteq \begin{matrix} NL \\ LOGDCFL \end{matrix} \subseteq SAC^1 \subseteq P \subseteq NP$$

1.4 Our contributions

In our Master thesis [48], we introduced groupoids as language recognizers. Our objective was to generalize the notions of recognition by semigroups and by programs over semigroups and study the associated class of languages. This was done by replacing the semigroups by their nonassociative counterpart. Let us explain informally what is meant by recognition by a groupoid. A more formal definition is given in Chapter 2.

A language $L \subseteq A^*$ is recognized by programs over a groupoid G if there exist an accepting set $F \subseteq G$ and a projection[2] $\phi : A^* \to G^*$, such that a word $w \in A^*$ belongs to L if and only if there exist a way of evaluating $\phi(w)$ such that the result belongs to F. Whenever ϕ is an homomorphism, L is simply said to be recognized by G.

We showed that the class of languages recognized by finite groupoids corresponds precisely to the context-free languages. Moreover, when ϕ is restricted to have polynomial length (in function of the input length) and satisfies some uniformity conditions, the class of languages recognized by programs over a groupoid corresponds to SAC^1.

Here, we investigate the computational power of finite groupoids following many directions. We now detail the contributions made in this thesis.

We define three variations of the recognition by programs over groupoids. In the first one we allow the programs to use a different groupoid for each input length. These programs over growing groupoids, introduced in [12], are no more powerful than *standard* programs whenever the growth of the groupoids does not exceed some polynomial (as a function of the input size).

[2]By projection we mean a mapping, where for any $i \geq 1$ there exist $j \geq 1$ such that the i^{th} symbol in $\phi(w)$ is determined by the j^{th} symbol of w.

In the definition of recognition by programs we have to choose how we evaluate $\phi(w)$ in order to get an element in F, and this choice is taken among all well-formed parenthesizations. In the second variation that we introduce in Chapter 2, we restrict the parenthesization to be of some particular form. We then talk of *restricted* recognition by programs. For example, we may want that the depth of a parenthesization be logarithmic in terms of the length of the programs.

In the third variation (introduced in [12]), we explicitly give the parenthesization with the programs which are then called *parenthesized* programs.

These three variations can be combined in various ways. For example, we can talk of restricted recognition by parenthesized programs over growing groupoids.

An important question is how the algebraic structure of finite groupoids influence the kind of languages they can recognize (by programs or by homomorphism). In this thesis we will thus examine different kinds of groupoids. Some of them have already been studied. This is the case of loops, quasigroups (e.g. see [16]), and weakly linear groupoids (called *linear* in [53]). We also introduce other types of groupoids: weakly associative, one-sided, Lie groupoids, weakly cancellative, and linear groupoids. The computational power of weakly associative, one-sided, and Lie groupoids is examined in Chapter 2. Weakly linear and linear groupoids are studied in Chapter 3 while loops, quasigroups, and weakly-associative groupoids are treated in detail in Chapter 4.

In Chapter 2, we simplify the proof of [48] (see also [12]), showing that a language belongs to SAC^1 if and only if it is recognized by polynomial-length programs over a fixed groupoid. We also construct groupoids such that the class of languages recognized by programs over them corresponds respectively to TC^0 and NC^1. Indeed it was already known that NC^1 corresponds to the languages recognized by polynomial-length programs over the symmetric group S_5 (see [5]) but S_5 has order 60 while our groupoid contains only 9 elements.

We introduce in this thesis different forms of restricted recognition. For example, we discuss in Chapter 2 the left-to-right recognition by programs,

where the evaluation is restricted to be from left to right. We also define constant right-depth recognition that restrict the evaluation trees[3] to be such that any path from the root to a leaf contains a number of right edges that is bounded by some constant. We show that with these notions we can recognize only (and all) languages in NC^1. In Chapter 3, we discuss in detail linear recognition, where the evaluation trees are such that each node has at most one child that is not a leaf (they are called linear trees).

We also investigate parenthesized programs (called structured programs in [12]). In Chapter 2 we show that they recognize precisely the languages in NC^1. We prove that any parenthesized programs over any groupoid can be simulated by parenthesized programs over some commutative groupoid. Moreover, if G_1 and G_2 are two isotopic groupoids (i.e. one can be obtained by permuting the rows and the columns of the multiplication table of the other) and if G_1 possesses an identity, then parenthesized programs over G_1 can be simulated by parenthesized programs over G_2. We also show that programs over a monoid M can be simulated by parenthesized programs over any groupoid G, whenever the mappings $G \to G$ induced by the rows and the columns of G generate a monoid (called the multiplication monoid) that contains M.

Up to now, we used the word linear in three different ways. We used it twice to qualify some special kind of groupoids and also to specify a form of restricted recognition. A groupoid is called weakly linear if it possesses an absorbing element 0, and if it satisfies $(ab)(cd) = 0$ for any $a, b, c, d \in G$. A groupoid is linear if for any nonempty word $w \in G^*$, any element that results for the evaluation of w using any parenthesization can also be obtained using an evaluation tree that is linear. In Chapter 3, we show that linear recognition by a finite groupoid, recognition by a linear groupoid, and recognition by a weakly linear groupoid are all equivalent notions of recognition. We also prove that linear groupoids recognize precisely the linear context-free languages, and

[3]Given an alphabet A and a word over A, any parenthesization of w can be represented in the obvious way by some binary tree that we call an evaluation tree.

that programs over linear groupoids recognize all and only those languages in NL.

Linear context-free languages are well known to correspond to those languages that can be expressed as $L = \{uv \mid (\tilde{u}, v) \in R_L\}$, where $\tilde{u}$ represent the mirror image of u and R_L is a relation recognized by a rational transducer (see [13]). We study in Chapter 3, the relationship between the algebraic structure of the transducers that recognize R_L and the groupoids that recognize L. In particular we define the transformation monoid of a transducer (given in some normal form) recognizing R_L, and we show that it closely corresponds to the multiplication monoid of a groupoid linearly recognizing L.

We show that the multiplication monoid of a groupoid G is an important parameter for determining what languages can be linearly recognized by G. In particular we prove in Chapter 3 that the language $L = \{a^n b^n w \mid n \geq 0, w \in L_0\}$, where L_0 is a regular language satisfying some algebraic conditions, cannot be recognized by G if the syntactic monoid of L_0 is not contained in the multiplication monoid of G.

When the multiplication monoid of a groupoid is a group, the groupoid itself is a quasigroup (or a loop when it possesses an identity). We prove in Chapter 4 that quasigroups can recognize or linearly recognize only regular languages. More precisely, we show that any language recognized by a quasigroup is the finite union of languages of the form $L_1 L_2 \cdots L_m$, where each L_i is a language recognized by a finite group. We also show that any cofinite language is recognized by some finite quasigroup but that no finite language can be so recognized. As a consequence, we show that the class of languages recognized by a finite quasigroup is not closed under complementation.

As an important tool for obtaining the above results we define the weakly cancellative groupoids which are those groupoids G with an absorbing element 0 such that, for any $a, x, y \in G$, if $ax = ay \neq 0$ or $xa = ya \neq 0$ then $x = y$. We show that any language recognized by a weakly cancellative groupoid with 0 in the accepting set is also recognized by some finite quasigroup. This result is very useful since weakly cancellative groupoids are much easier to construct

than quasigroups.

We investigate parenthesized programs over quasigroups in Chapter 4. In particular, we study the restricted case of the well-parenthesized expressions with variables over loops. We prove a loop analogue of the Maurer-Rhodes Theorem [51] saying that any function $L^n \to L$ can be represented by a well-parenthesized expression over a simple nonabelian loop L. This leads to a generalization of the Barrington Theorem saying that any language in NC^1 is recognized by parenthesized programs over any nonsolvable[4] loop. Solvable loops are shown to be as powerful as those that are nonsolvable whenever the multiplication monoid is itself nonsolvable.

Programs over growing groupoids are investigated in Chapter 5. We begin by giving several simulations between machines and programs. Deterministic machines are related to parenthesized programs while nondeterministic machines correspond more closely to general programs. In particular, we characterize SAC^1 and NP as those languages recognized by programs over groupoids growing polynomially and exponentially, respectively (the last result was first proved in [53]). On the other hand, we characterize NL and NP as those languages linearly recognized by programs over groupoids growing polynomially and exponentially, respectively. The classes L and P are proved to correspond to the languages linearly recognized by parenthesized programs over groupoids growing polynomially and exponentially, respectively. All these results assume some proper uniformity conditions on the programs.

We define a normal form for semibounded circuits that we call tree-like. It is shown that any semibounded circuit can be transformed into a tree-like circuit without modifying too much its parameters (for example polynomial size is preserved). This is used to construct a family of groupoids $G_1 \subseteq G_2 \subseteq \cdots$ growing polynomially and having some nice properties. In particular they are very simple to define, and they can be used to capture complexity classes such as SAC^1, NL, L, and NC^1. These results can be used to restate some

[4]The notions of simple and solvable loops are natural generalizations of simple and solvable groups. Formal definitions are given in Chapter 4.

open questions in complexity theory. For example, we show that $NC^1 \neq SAC^1$ if and only if recognition by programs over $(G_n)_{n\geq 0}$ is strictly more powerful than left-to-right recognition by programs over G_6.

Tree-like circuits are built up from blocks of input gates and blocks of depth-two semi-bounded circuits connected together in the manner of a tree. A tree-like circuit having the property that on any input at most one gate in each block evaluates to 1 is called a clean circuit. We show in Chapter 5 that clean circuits are closely related to parenthesized programs over growing groupoids. This leads to a characterization of L and P in terms of clean circuits. More precisely, P corresponds to the languages recognized by a uniform family of clean circuits of exponential size and polynomial degree, where the degree corresponds roughly to the number of blocks in the circuits. Similarly, L is the class of languages recognized by polynomial size clean circuits that are skew, i.e. AND-gates have at most one child that is not an input gate. These results are interesting because they give a characterization of deterministic complexity classes using a restriction on the class of circuits that recognize their nondeterministic version. For example, NP corresponds to uniform families of tree-like circuits of exponential size and polynomial degree.

Finally, we show at the end of Chapter 5 that parenthesized programs over groupoids growing polynomially can be simulated by Owner-Read, Owner-Write PRAMs using a polynomial number of processors and running in time proportional to the depth of the parenthesization.

Chapter 2

Groupoids and Languages

2.1 Introduction to groupoids

A *groupoid* is a pair $(G, \cdot)$, where G is a set and $\cdot$ is binary operation called product and defined over G. Whenever there is no confusion on the operation, we simply denote $(G, \cdot)$ by G and the multiplication is denoted using concatenation. A groupoid G is *commutative* if $ab = ba$ for any $a, b \in G$, it is associative if $(ab)c = a(bc)$ for all $a, b, c \in G$. When the product is associative, G is called a *semigroup*.

An element 1 of a groupoid G is called an *identity* if for any $g \in G$, $1g = g1 = g$. An element 0 of G is said to be *absorbing* if for any $g \in G$, $0g = g0 = 0$. A semigroup with identity is called a *monoid*. We denote by G^1 the smallest groupoid that contains G and possesses an identity element. Moreover, we denote by G^0 the smallest groupoid that contains G and that possesses an absorbing element.

Example 2.1.1 *Let A be a finite set and define the set $A^{(+)}$ as follows: Any $a \in A$ is in $A^{(+)}$; if $u, v \in A^{(+)}$ then $(uv) \in A^{(+)}$; nothing else is in $A^{(+)}$. An element of $A^{(+)}$ can be viewed as a binary tree having its leaves labeled with elements in A.*

A product can be defined on this set such that for all $a, b \in A^{(+)}$ we have $a \cdot b = (ab)$ (observe that this product is not associative). Then, $A^{(+)}$ is a groupoid and $A^{(*)} = A^{(+)} \cup \{\epsilon\}$, where ϵ is the empty word, is called the free groupoid *over A.*

Example 2.1.2 *The hypercomplex numbers of rank n are the expressions of the form* $\alpha = a_0 + a_1 i_1 + \cdots + a_n i_n$, *where* a_i *is a real number and* i_j *an abstract symbol. The conjugate of* α *is the expression* $\overline{\alpha} = a_0 - a_1 i_1 - \cdots a_n i_n$. *Hypercomplex numbers of rank 0 correspond to real numbers. Furthermore, given a system U of rank n, we can define the doubling of U as the set* $\{a+be : a, b \in U\}$ *together with the addition* $(a+be)+(c+de) = (a+c)+(b+d)e$ *and the product* $(a+be)(c+de) = (ac-\overline{d}b)+(ad+b\overline{c})e$. *Hence, hypercomplex numbers of rank 1 correspond to complex numbers, those of rank 2 to quaternions, and those of rank 4 are called Cayley numbers. One can observe that the product on Cayley numbers is neither commutative nor associative.*

Example 2.1.3 *Let R be any associative ring. Then, by preserving the addition and by redefining the product of* x *and* y *as* $xy - yx$, *we get a Lie ring, whose product is in general nonassociative.*

All these groupoids contains an infinite number of elements. However, in this work, we will be mainly concerned with finite groupoids. The number of elements in a finite groupoid G is called the *order* of G.

2.1.1 Subgroupoids and homomorphisms

Given a groupoid G and a subset $S \subseteq G$, we denote by $\langle S \rangle$ the subgroupoid generated by S. We say that a groupoid H divides G (denoted $H \prec G$) whenever there exists a homomorphism from a subgroupoid of G onto H.

We now mention some remarkable subgroupoids contained in any groupoid G. The left, middle and right nucleus of a groupoid, respectively denoted N_λ, N_μ and N_ρ, are defined as followed.

$$N_\lambda = \{g \in G \mid \forall x, y \; g(xy) = (gx)y\}$$

$$N_\mu = \{g \in G \mid \forall x, y \; x(gy) = (xg)y\}$$

$$N_\rho = \{g \in G \mid \forall x, y \; x(yg) = (xy)g\}$$

The *nucleus* of G is defined as $N = N_\lambda \cap N_\mu \cap N_\rho$. It is easily seen that N_λ, N_μ, N_ρ and N are associative subgroupoids of G.

The *center* $\mathcal{Z}$ of G is the subset of elements n of N satisfying $nx = xn$, for all $x \in G$. It is a subgroupoid of G that is both associative and commutative.

A subgroupoid I of a finite groupoid G is called a *left* (resp. *right*) *ideal* of G if $IG \subseteq I$ (resp. $GI \subseteq I$). If I is both a left and a right ideal, then it is simply called an ideal. There is also a notion of ideal for transfinite groupoids (see [15] p.253). For a discussion of the Green's relations on groupoids see [44].

2.1.2 The multiplication semigroup and monoid

In this subsection and the next one, we define two concepts that play an important role in the theory of groupoids.

With any element $g \in G$ we associate two functions $R(g), L(g) : G \to G$ called respectively the *left* and *right multiplication functions* and defined as $aR(g) = ag$ and $aL(g) = ga$, for any $a \in G$. The *multiplication semigroup* of G is defined to be the semigroup $\mathcal{S}(G)$ generated by $\{R(a), L(a) \mid a \in G\}$, where the operation is composition. For $a \in G$ and $U \in \mathcal{S}(G)$, we denote by aU the element of G obtained by applying the function U to a. Moreover, if $V \in \mathcal{S}(G)$, then $a(UV) = (aU)V$, and we simply denote it by aUV. The *multiplication monoid* of G is the multiplication semigroup of G^1. It is denoted with $\mathcal{M}(G)$. One can check that $L(1) = R(1)$ is then the identity of $\mathcal{M}(G)$.

The following lemmas are straightforward generalization of similar results in loop theory (see [16]).

Lemma 2.1.1 *A groupoid G is associative if and only if for all $a, b \in G$, $R(a)R(b)=R(ab)$.*

Proof. If G is associative then $xR(a)R(b) = (xa)b = x(ab) = xR(ab)$ showing that $R(a)R(b) = R(ab)$. On the other hand, if $R(a)R(b) = R(ab)$ for all $a, b \in G$ then $(xa)b = xR(a)R(b) = xR(ab) = x(ab)$ and G is associative. □

Lemma 2.1.2 *A groupoid G is commutative and associative if and only if $\mathcal{M}(G)$ is commutative.*

Proof. If G is commutative and associative then for any $a, b, x \in G$ we have $xb = bx$ implying that $xR(b) = xL(b)$ and that $L(b) = R(b)$. Furthermore since $a(xb) = (ax)b$ then $xR(b)L(a) = xL(a)R(b)$ and $R(b)L(a) = L(a)R(b)$. Hence, we find that $R(a)R(b) = R(b)R(a)$ and, by symmetry, $L(a)L(b) = L(b)L(a)$, proving $\mathcal{M}(G)$ is commutative. Suppose now that $\mathcal{M}(G)$ is commutative. Then we have $a(xb) = xR(b)L(a) = xL(a)R(b) = (ax)b$ showing that G is associative. Furthermore since $ab = 1R(a)R(b) = 1R(b)R(a) = ba$ then G is also commutative. □

Lemma 2.1.3 *A groupoid is commutative and associative if and only if $\mathcal{S}(G)$ is isomorphic to G.*

Proof. Let G be commutative and associative. Then, $\mathcal{S}(G)$ is generated by $\{R(a) : a \in G\}$. We have $xR(a)R(b) = (xa)b = x(ab) = xR(ab)$ proving that $\mathcal{S}(G) = \{R(a) : a \in G\}$ and that $\mathcal{S}(G)$ is isomorphic to G. Now if G is isomorphic to $\mathcal{S}(G)$ then G is associative. It remains to show that G is also commutative. Since G is a semigroup then by Lemma 2.1.1 it is isomorphic to $\{R(a) : a \in G\}$. This means that for all $b \in G$ we have $L(b) \in \{R(a) : a \in G\}$. Hence there exists $a \in G$ such that $L(b) = R(a)$. Hence $bb = bL(b) = bR(a) = ba$ and therefore $a = b$, by the cancellation laws, and $L(a) = R(a)$ for all $a \in G$. This shows that G is commutative and concludes the proof. □

Lemma 2.1.4 *Let G be a groupoid. If S is a subgroupoid of G, then $\mathcal{M}(S) \prec \mathcal{M}(G)$.*

Proof. Let $N = \langle R(a), L(a) : a \in S \rangle$ and define the homomorphism $h : N \to \mathcal{M}(S)$ such that for all $X \in N$, $h(X)$ is the unique $U \in \mathcal{M}(S)$ such that $aX = aU$ for every $a \in S^1$. □

Lemma 2.1.5 *Let G and Q be groupoids. If $h : G \to Q$ is a homomorphism, then there exists a homomorphism $\phi : \mathcal{M}(G) \to \mathcal{M}(Q)$.*

Proof. We have that h induces a congruence $\equiv$ on G such that $G/\equiv$ is isomorphic to Q. Then, $\phi : \mathcal{M}(G) \to \mathcal{M}(G/\equiv)$, the morphism induced by $\phi(R(a)) = R(h(a))$ and $\phi(L(a)) = L(h(a))$, is the desired homomorphism. □

Proposition 2.1.6 *If $Q \prec G$ then $\mathcal{M}(Q) \prec \mathcal{M}(G)$.*

Proof. If $Q \prec G$, then there exist a subgroupoid $H \subseteq G$ and a homomorphism $h : H \to Q$ such that $\mathcal{M}(Q) \prec \mathcal{M}(H)$ (by Lemma 2.1.5) and $\mathcal{M}(H) \prec \mathcal{M}(G)$ (by Lemma 2.1.4). Hence, $\mathcal{M}(Q) \prec \mathcal{M}(G)$. □

Proposition 2.1.7 *Let G and H be two groupoids. Then $\mathcal{M}(G \times H) \subseteq \mathcal{M}(G) \times \mathcal{M}(H)$.*

Proof. Simply observe that $\mathcal{M}(G \times H)$ is isomorphic to the submonoid of $\mathcal{M}(G) \times \mathcal{M}(H)$ generated by the set $\{(R(g), R(h)), (L(g), L(h)) \mid g \in G^1 \text{ and } h \in H^1\}$. □

2.1.3 Isotopy

Two groupoids $(G, \cdot)$ and $(H, *)$ are said to be *isotopic* if there exist three bijections $\alpha, \beta, \gamma : G \to H$ such that $\alpha(x) * \beta(y) = \gamma(x \cdot y)$. Then, (α, β, γ) is called an *isotopy* of G onto H. Considering the Cayley table of G, one can construct the Cayley table of H by permuting the lines of G with α, permuting the columns of G with β and then renaming elements inside the table with γ. So, an isomorphism is just a particular isotopy where $\alpha = \beta = \gamma$.

If ι is the identity mapping, then (α, β, ι) is called a *principal isotopy* of G onto H. In general it is sufficient to consider only principal isotopies. This is justified by the following theorem.

Theorem 2.1.8 ([3]) *If G and H are isotopic groupoids then H is isomorphic to a principal isotope of G.*

Proof. Let (α, β, γ) be an isotopy of G onto H, let $\delta = \alpha\gamma^{-1}$, and let $\eta = \beta\gamma^{-1}$. We have $(\alpha, \beta, \gamma) = (\delta\gamma, \eta\gamma, \gamma)$. Hence, there exists a groupoid K such that (δ, η, ι) is a principal isotopy of G onto K and γ is an isomorphism of K onto H. □

For groupoids that possess an identity element, we can give a stronger result.

Theorem 2.1.9 ([3]) *Let $(G, \cdot)$ and $(G, *)$ be isotopic groupoids, and suppose that $(G, *)$ possesses an identity 1. Then, there exist $f, g \in G$ such that $(R(f), L(g), \iota)$ is a principal isotopy of $(G, \cdot)$ onto $(G, *)$.*

Proof. Assume that $x \cdot y = \delta(x) * \eta(y)$ for some permutations $\delta, \eta : G \to G$. Let $g = \delta^{-1}(1)$ and $f = \eta^{-1}(1)$. We have

$$y = 1 * y = \delta^{-1}(1) \cdot \eta^{-1}(y) = \eta^{-1}(y)L(g)$$

$$x = x * 1 = \delta^{-1}(x) \cdot \eta^{-1}(1) = \delta^{-1}(x)R(f)$$

Then $\eta = L(g)$, $\delta = R(f)$ and

$$x * y = xR^{-1}(f) \cdot yL^{-1}(g)$$

This shows that $(R(g), L(f), \iota)$ is a principal isotopy of $(G, \cdot)$ onto $(G, *)$. □

Theorem 2.1.10 (see [54]) *Let Q and G be two groupoids with identity. If Q is isotopic to G, then $\mathcal{M}(Q)$ is isomorphic to $\mathcal{M}(G)$.*

Proof. Without loss of generality we can suppose that $G = (G, \cdot)$ and $Q = (G, *)$. For any element x of the set G, denote by $R(x)$ and $L(x)$ the right and left multiplication functions of $(G, \cdot)$, and by $R_*(x)$ and $L_*(x)$ those of $(G, *)$.

Since $(G, \cdot)$ is isotopic to $(G, *)$ which contains an identity, there exist $R(g), L(f) \in \mathcal{M}(G)$ such that $x * y = xR^{-1}(f) \cdot yL^{-1}(g)$.

Hence, we can write

$$R_*(y) = R^{-1}(f)R(yL^{-1}(g))$$

$$L_*(x) = L^{-1}(g)L(xR^{-1}(f))$$

Observe that since $L(g)$ and $R(f)$ are permutations, there exists an integer k such that $L^k(g) = L^{-1}(g)$ and $R^k(f) = R^{-1}(f)$. Thus $L^{-1}(g)$ and $R^{-1}(f)$ belong to $\mathcal{M}(G)$.

This shows that $\mathcal{M}(Q) \subseteq \mathcal{M}(G)$. The other direction is proved similarly.

□

The proof of the next theorem can be found in [15] pp.250-253.

Theorem 2.1.11 *Let $(G,\cdot)$ and $(G,*)$ be two isotopic groupoids both with an identity. Then, $(G,\cdot)$ and $(G,*)$ have isomorphic left, middle and right nucleus, and they have isomorphic centers. Moreover, their ideals are isotopic in pairs.*

As a corollary we have

Corollary 2.1.12 *Let G be a groupoid with identity isotopic to a monoid M. Then, G and M are isomorphic.*

Observe that the above corollary is false if G does not possess an identity. A counterexample of order 2 can easily be constructed. Indeed, the semigroup AND $= \{0,1\}$, defined as $00 = 01 = 10 = 0$ and $11 = 1$ is isotopic to the groupoid NAND $= \{0,1\}$, defined as $00 = 01 = 10 = 1$ and $11 = 0$, but AND and NAND are not isomorphic. We conclude this subsection with a theorem stating some limits of what can be preserved by isotopy.

Theorem 2.1.13 (see [16] p.58) *There exist isotopic groupoids with identity that satisfy any of the following conditions.*

- *Only one of them is commutative.*
- *They have a different number of generators.*
- *Their nuclei are not isomorphic.*

2.2 Recognition by finite groupoids

Let A and B be two finite sets, and let $\varphi : A^* \to B^*$ be a monoid homomorphism. Then, φ is said to be *strictly alphabetical*[1] if $\varphi(A) \subseteq B$.

Given a groupoid G and a word $w \in G^*$, we denote by $G(w)$ the set of all elements in G that can be obtained by evaluating w using any parenthesization. A groupoid G is a semigroup if and only if for all $w \in G^*$, $G(w)$ is a singleton.

Definition 2.2.1 *A language $L \subseteq A^*$ is* recognized *by a groupoid G, if there exist a strictly alphabetical homomorphism $\varphi : A^* \to G^*$ and an* accepting *set $F \subseteq G$ such that*

$$L = \{x \in A^* \mid G(\varphi(w)) \cap F \neq \emptyset\}.$$

The following theorem is fundamental to this thesis.

Theorem 2.2.2 ([12]) *A language is context-free if and only if it is recognized by a finite groupoid.*

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

$$\begin{aligned} V &= \{q_i : 0 \leq i \leq k\} \\ T &= A = \{a_1, \ldots, a_m\} \\ P &= \{q_i \to a : a \in A\,,\ \theta(a) = i\} \cup \\ &\quad \{q_0 \to q_i : i \in F\} \cup \\ &\quad \{q_i \to q_j q_l : i, j, l \in [k] \text{ and } j \cdot l = i\} \\ S &= q_0 \end{aligned}$$

If $\epsilon \in Y$, then we add the rule $q_0 \to \epsilon$.

An induction on the length of x proves :

$$(\forall x \in A^*)[G(\theta(x)) \cap F \neq \emptyset) \text{ iff } (q_0 \stackrel{*}{\Rightarrow} x)].$$

[1] This terminology comes from [13]

($\Rightarrow$) 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 $V = \{q_0, \ldots, q_k\}$. We can assume that D is in Chomsky normal form with the only rules involving q_0 of the form $q_0 \to \epsilon$ or of the form $q_0 \to q$ for $q \in V$. Moreover, we can assume that D is invertible, i.e. if two productions have the form $q_i \to q_k q_l$ and $q_j \to q_k q_l$ then $q_i = q_j$ (see [40]).

We define the groupoid $G = (V \setminus \{q_0\}) \cup \{c, \$\}$ such that c 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 $a, b, c \in V$. In all other cases, $a \cdot b = \$$.

Now define $X = \{q \in V \setminus \{q_0\} \mid (q_0 \to q) \in P\}$ and $F = X \cup \{e\}$ if $\epsilon \in Y$ 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:

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

This concludes the proof. □

The fact that homomorphisms are restricted to be strictly alphabetical in the above definition, implies that the complexity of a context-free language $L \subseteq A^*$ depends only on the structure of those groupoids G that can recognize it. In particular, we will often assume that $A \subseteq G$ and that φ maps each $a \in A$ to itself. We denote by $W(G, A, F)$ the language consisting of all $w \in A^+$ that can be evaluated to some element in $F \subseteq G$. The problem of determining what words belongs to $W(G, A, F)$ is called a *word problem* over G.

We now give some examples of groupoids and the languages they recognize.

2.2.1 Finite semigroups

We observe that when G is a finite semigroup, Definition 2.2.1 corresponds precisely to the notion of recognition used in algebraic theory of automata (see [56]). Hence, Theorem 2.2.2 can be seen as a generalization of the following result.

Theorem 2.2.3 (see [56, 28]) *A language is regular if and only it is recognized by a finite semigroup.*

The very close relationship between subclasses of regular languages and subclasses of finite semigroups is best expressed using two concepts: syntactic semigroups and varieties.

Given a language $L \subseteq A^*$, the syntactic congruence $\sim_L$ is the equivalence on A^* that satisfies: $u \sim_L v$ iff

$$(\forall x, y \in A^*)[xuy \in L \Leftrightarrow xvy \in L]$$

The syntactic semigroup of L is the quotient semigroup $S_L = A^+/\sim_L$, and the syntactic monoid of L is the quotient monoid $S_L = A^*/\sim_L$.

The following two propositions indicate the importance of the syntactic semigroup. Observe that these results remain true if we replace the semigroups by monoids. The proof can be found in [56, 28].

Proposition 2.2.4 *Let L be a language and S_L its syntactic semigroup. Then, a semigroup S recognizes L if and only if $S_L \prec S$.*

In other words, S_L is the smallest semigroup recognizing L, i.e. any semigroup that recognizes L must 'contain' S_L. As a corollary of Theorem 2.2.3 we have the following result.

Proposition 2.2.5 *A language is regular if and only if its syntactic semigroup is finite.*

Thus, the syntactic congruence induces a mapping from regular languages to finite semigroups. However, this mapping is not bijective. In particular, not every semigroup is the syntactic semigroup of some language. This is where the notion of varieties is required.

A class of semigroups or monoids V forms a variety if it is closed under finite direct product and division[2]

[2]The standard definition of variety (see [28, 46]) allows infinite direct product and our definition corresponds to pseudo-variety. Since we are only interested in finite groupoids, we prefer to use the term variety even if only finite direct products are considered.

A class of languages $\mathcal{L}$ forms a $\star$-variety (resp. $+$-variety) if it is closed under the Boolean operations, inverse homomorphism (resp. inverse non-erasing homomorphism), and right and left division by a letter, where the right division of $L \subseteq A^*$ by $a \in A$ is the set $\{v \in A^* \mid va \in L\}$ (left division is defined similarly). Observe that this distinction between $\star$-variety and $+$-variety is necessary since $\star$-varieties correspond to varieties of monoids and $+$-varieties correspond to varieties of semigroups. Moreover, a $+$-variety is not necessarily a $\star$-variety. An example is given by the $+$-variety of all finite and cofinite languages.

The following proposition is known as the Eilenberg Theorem for pseudovarieties. We state it only for $+$-varieties and semigroups but it can also be formulated for $\star$-varieties and monoids.

Proposition 2.2.6 ([28]) *There is a bijection between $+$-varieties of languages and varieties of finite semigroups. More precisely, let $\mathbf{V}$ be a class of semigroups, let $A^+\mathbf{V}$ be the set of subsets of A^+ recognized by a semigroup in $\mathbf{V}$, let $V = \cup_A A^+\mathbf{V}$, and let $\mathbf{U}$ be the variety generated by the syntactic semigroups of the languages in V. Then, $\mathbf{V}$ is a variety only if V is a $+$-variety, and in this case we have $\mathbf{V} = \mathbf{U}$.*

Indeed the set of all semigroups forms a variety corresponding to the variety of all regular languages. We now give some other examples.

Example 2.2.1 The variety of nilpotent semigroups consists of those semigroups S that satisfies the identity $es = e = se$, for all $s, e \in S$ where $ee = e$.

A language is recognized by a nilpotent semigroup if and only if it is finite or co-finite.

Example 2.2.2 ([65]) The variety of aperiodic semigroups consists of those semigroups S for which there exist $k > 0$ such that $a^k = a^{k+1}$, for all $a \in S$.

A regular language $L \subseteq A^*$ is *star-free* if it is in the closure of $\{\{a\} \mid a \in A\}$ under Boolean operations and concatenation.

A language is recognized by an aperiodic semigroup iff it is star-free.

Example 2.2.3 ([66]) A semigroup M is J-trivial if for every $m \in M$, the set $M^1 m M^1$ is a singleton. The class of all J-trivial monoids forms a variety denoted **J**.

We say that $u \in A^*$ is a *subword* of $w \in A^*$ whenever $u = a_1 a_2 \cdots a_n$, $a_i \in A$, and $w = w_0 a_1 w_1 \cdots w_{n-1} a_n w_n$. Let $L_u \in A^*$ be the set of words that contain the subword u. A language $L \subseteq A^*$ is *piecewise-testable* if it belongs to the Boolean closure of languages of the form L_u.

Then, a language is piecewise-testable if and only if its syntactic monoid is J-trivial.

Example 2.2.4 ([77]) The variety of nilpotent groups can be defined recursively as follows. Any abelian group is nilpotent. Let G be a nontrivial group and let Z be the center of G. Then, G is nilpotent iff Z is nontrivial and G/Z is nilpotent.

Let $\binom{w}{u}$ denote the number of occurrences of the subword u in $w \in A^*$. For any $u \in A^*$, $t \geq 0$, $q \geq 1$ and $0 \leq k < q$, let $[u, t, q, k]$ be the set of all words $w \in A^*$ such that $\binom{w}{u} \geq t$ and $\binom{w}{u} \equiv k \pmod{q}$.

A language $L \subseteq A^*$ is in the Boolean closure of languages of the form $[u, 0, q, k]$ iff it is recognized by a nilpotent group.

Example 2.2.5 Commutative semigroups also form a variety. A language $L \in A^+$ is recognized by a commutative semigroup iff it is in the Boolean closure of languages of the form $[a, t, q, k]$, where $a \in A$.

Example 2.2.6 ([76]) The variety of *solvable* semigroups consists of those semigroups that contains no nonsolvable group[3].

For any $w \in A^*$, $a_1, \ldots, a_n \in A$, and $L_0, L_1, \ldots, L_n \subseteq A^*$, we denote by $|w|_{[L_0 a_1 L_1 \cdots a_n L_n]}$, the number of factorizations of w of the form $u_0 a_1 u_1 \cdots a_n u_n$, where $u_i \in L_i$. Define the language $[L_0 a_1 L_1 \cdots a_n L_n]_{t,q,k}$ as the set of words $w \in A^+$ such that $|w|_{[L_0 a_1 L_1 \cdots a_n L_n]} \geq t$ and $|w|_{[L_0 a_1 L_1 \cdots a_n L_n]} \equiv k \pmod{q}$. Finally, let S_0 be the Boolean closure of languages of the form A^+, $\emptyset$, and

[3]Solvability is defined in Section 1 of Chapter 4 in the more general context of loops.

let S_m be the Boolean closure of languages of the form $[L_0 a_1 L_1 \cdots a_n L_n]_{t,q,k}$, where $L_j \in S_{m-1}$. Then $\mathcal{S} = \cup_{m \geq 0} S_m$ forms a variety of languages.

A language is in $\mathcal{S}$ if and only if it is recognized by a finite solvable semigroup.

2.2.2 Weakly associative groupoids

A groupoid G is *weakly associative* if for all $a, b, c \in G^0$,

$$a(bc) \neq 0 \text{ and } (ab)c \neq 0 \Rightarrow a(bc) = (ab)c.$$

In particular, any semigroup is a weakly associative groupoid. The complexity of the languages recognized by weakly associative groupoids may depend on whether or not 0 belongs to the accepting set.

Lemma 2.2.7 *Let G be a weakly associative groupoid with an absorbing element 0. Let F be a subset of G that contains 0. Then, $L = W(G, G, F)$ is regular.*

Proof. For all $a \in G$, define $L_a \subseteq G^*$ as the set of words that left-to-right evaluate to a. Clearly L_a is a regular language , and $L = \bigcup_{a \in F} L_a \cup L'$, where L' is the set of words over G that can be evaluated to 0 in some way.

Let $w \in G^*$. If $G(w)$ does not contain 0 then there is a unique element $a \in G(w)$. In this case, w can be evaluated in any way and will always yield the same result. Thus, the problem reduces to determine if $0 \in G(w)$.

Let s be a shortest segment of w that evaluates to 0. Observe that for any proper decomposition $s = uv$, $G(u)$ and $G(v)$ are singletons. There exist $u, v \in G^+$ such that $s = uv$ and $ab = 0$ where $\{a\} = G(u)$ and $\{b\} = G(v)$. Hence, we can write

$$L' = \bigcup_{ab=0} G^* L_a L_b G^* \cup G^* 0 G^*,$$

proving that L is regular. □

It seems that when 0 does not belong to the accepting set, the word problem of G could be more difficult. Actually, we know little, even for *weakly com-*

mutative and associative groupoids, i.e. those weakly associative groupoids G with the property that for any $a, b \in G^0$,

$$ab \neq 0 \text{ and } ba \neq 0 \Rightarrow ab = ba.$$

At least, we can easily show that the word problem for weakly associative and even weakly commutative and associative groupoids is hard for NC^1 when 0 does not belong to the accepting set. To see this, we will define a special kind of weakly commutative and associative groupoids. An element a of a groupoid G is called *left-sided* (resp. *right-sided*) if for all $g \in G$, we have $ag = 0$ (resp. $ga = 0$). A groupoid G is called *one-sided* if it contains only left-sided and right-sided elements.

If G is a one-sided groupoid and $a, b, c \in G$, then it must satisfy $ab = 0$ or $ba = 0$, as well as $a(bc) = 0$ or $(ab)c = 0$. Hence, we have the following observation.

Lemma 2.2.8 *Any one-sided groupoid is weakly commutative and associative.*

□

It is not known if one-sided groupoids are strictly less powerful than other weakly associative groupoids, but it can be shown that they recognize fewer languages than general groupoids.

Lemma 2.2.9 *Any language recognized by a one-sided groupoid is deterministic context-free.*

Proof. Let G be a one-sided groupoid. We will show how to evaluate a word $w \in G^*$ to its unique possible solution different from 0 using a deterministic pushdown automaton. In the following algorithm, the impossibility of moving must be interpreted as if we were returning 0. Initially, the stack is empty.

```
b ← read the first symbol
While there is an unread symbol or the stack is not empty do
        If b is left-sided then
           pop the top of the stack a
           If a is right-sided then b ← ab
           Else return 0
        Else push b
             b ← read next symbol
Return b
```

□

An interesting question concerns the complexity of the word problem for one-sided groupoids. In view of the preceding lemma, it is natural to ask if this problem is complete for LOGDCFL. We will see in Section 2.3 that it is as hard as any problem in NC^1.

2.2.3 Lie groupoids

Let G be a finite group and let a and b be two elements of G. The *commutator* of a and b is denoted $[a, b]$ and defined as $[a, b] = a^{-1}b^{-1}ab$.

The *Lie groupoid* of G is the groupoid C defined over the same underlying set G with product $ab = [a, b]$. In general, C is not associative.

Observe how the identity in G becomes an absorbing element in C. In particular, if G is a commutative group, then C is a 0-simple semigroup, i.e. it contains an absorbing element and any product ab is equal to that absorbing element.

Proposition 2.2.10 *If G is nilpotent then C can recognize only finite and cofinite languages.*

Proof. Define $G_1 = \{[a, b] \mid a, b \in G\}$, and for $k > 1$ let $G_k = \{[a, b] \mid a \in$

$G_i,\ b \in G_j,\ i+j=k\}$. Since G is nilpotent, we must have $G_n = \{1\}$ for some $n \geq 1$ (see [36]).

Therefore, for all words $w \in C^*$ of length at least n, we have $C(w) = \{1\}$. This means that if L is recognized by C with accepting set $F \subseteq C$, then L contains all words of length at least n whenever $1 \in F$, and is finite otherwise. □

The commutators of a group G generate a subgroup called the *commutator subgroup* of G. We define a solvable group of depth 2 as a group whose commutator subgroup is commutative. An example of such a group is S_3, the group of permutations on 3 points. We will see in Chapter 3 that any language recognized by the Lie groupoid of a solvable group of depth 2 is in NL.

2.3 Recognition by programs

Mckenzie ([50]) suggested the idea of extending the notion of programs over semigroups (see [5, 8]) by using groupoids instead of semigroups. This leads to generalizing Definition 2.2.1 by allowing φ to be any projection from A^* to G^*.

Definition 2.3.1 *Let G be a groupoid and let $F \subseteq G$. We define a program P_n over G as a sequence of instructions $I_1 I_2 \cdots I_m$ of the form $I_j = \langle i_j, f_j \rangle$, where $1 \leq i_j \leq n$ and $f_j : A \to G$ is a function. Given an input $x \in A^n$, each instruction I_j outputs the element $h_j = f_j(x_{i_j}) \in G$. The program P_n* accepts *w if and only if $G(h_1 h_2 \cdots h_m) \cap F \neq \emptyset$.*

A language $L \subseteq A^*$ is said to be *recognized by programs over G* if there exists a family of programs $(P_n)_{n \geq 0}$ such that, for every $n \geq 0$ and every $w \in A^n$, program P_n accepts w if and only if w belongs to L. The *length* of $(P_n)_{n \geq 0}$ is a function mapping n to the number of instructions in P_n.

Hence, a projection from any language to a word problem over any groupoid G, forms a family of programs over G. Unless explicitely mentioned, we will always assume that programs have polynomial length. If G is a class of

groupoids, we denote by $\mathcal{P}(\mathrm{G})$ the class of languages recognized by polynomial-length programs over a groupoid in G. In particular, when G contains a single groupoid G, then we write $\mathcal{P}(G)$ instead of $\mathcal{P}(\mathrm{G})$.

Definition 2.3.2 *Given a complexity class C, a polynomial length family P of programs is said to be C-uniform if, given (w, k), the problem of computing the length of $P_{|w|}$ and its k^{th} instruction belongs to C. We simply say that P is uniform whenever $C = \mathrm{DTIME}(\log |w|)$.*

The power of programs over finite groupoid is given by the next theorem that will be proved in subsection 2.3.2.

Theorem 2.3.3 ([12]) *A language is in SAC^1 if and only if it is recognized by programs over a finite groupoid.*

2.3.1 Programs over semigroups

When G is a semigroup (monoid) in Definition 2.3.3, the model is called program over a semigroup (monoid). In [5], Barrington observed that programs over finite semigroups is a model of computation equivalent to bounded-width branching programs. Theorem 2.3.3 is thus the generalization of the following result.

Theorem 2.3.4 *A language is in NC^1 iff it is recognized by programs over a finite semigroup.*

In fact, Barrington's theorem is more precise. It says that if S is any nonsolvable semigroup (i.e. there is a nonsolvable group that divides S), then $\mathcal{P}(S) = \mathrm{NC}^1$.

Results analogous to Theorem 2.3.4 exist for subclasses of NC^1.

Theorem 2.3.5 ([8]) *If V is the variety of solvable semigroups, then $\mathcal{P}(\mathrm{V}) = \mathrm{ACC}^0$.*

Let CC^0 be the class of languages recognized by unbounded fan-in constant-depth Boolean circuits using only MOD_q gates, where $q > 1$. We thus have $CC^0 \subseteq ACC^0$.

Theorem 2.3.6 ([8]) *If* V *is the variety of solvable groups, then* $\mathcal{P}(V) = CC^0$.

Theorem 2.3.7 ([8]) *If* V *is the variety of aperiodic semigroups, then* $\mathcal{P}(V) = AC^0$.

We observe that a crucial point in determining the power of a semigroup S concerns the kind of groups it contains. If S contains no nontrivial groups then it can recognize only languages in AC^0. If it contains a nonsolvable group, then the word problem on S is complete for NC^1. Furthermore, if S contains nontrivial groups but these are all solvable, then S can recognize only languages in ACC^0. However, in this last case, S recognizes languages that are provably not in AC^0. This is because, by standard results in group theory (see [36]), there must exists a cyclic group of prime order that divides S, and it is proved in [69] that the word problem on this kind of groups is not in AC^0.

We conclude this subsection with a proof that any family of programs over a semigroup can be simulated by a family of programs over a one-sided groupoid having the same length.

Proposition 2.3.8 *For any semigroup S, there exists a one-sided groupoid G such that any language recognized by a family of programs over S is also recognized by a family of programs over G.*

Proof. Let $G = S \cup S' \cup \{0\}$, where S' is a copy of S and 0 is a new element. Let $a, b, c \in S$ and define the product on G by

- $a'b = c'$, where a' and c' are copies of a and c.
- All other products yield 0.

This groupoid is one-sided since all elements in S are left-sided, all elements in S' are right-sided, and 0 is both left-sided and right-sided.

Moreover, given any word $a'w \in S'S^*$, the only way to evaluate $a'w$ to an element different from 0 is to use a left-to-right parenthesization. In this case, $a'w$ yields an element $s' \in S'$ that is the copy of the element $s \in S$ resulting from the evaluation of $a'w$ in S. A program over S can thus be transformed into an equivalent program over G by only changing the first instruction. □

2.3.2 Groupoids and SAC^1

In this subsection we show that SAC^1 corresponds to the languages recognized by programs over a groupoid.

First, observe that programs over finite groupoids recognize only language in SAC^1. This is because programs over groupoids are projection-reducible to sets recognized by groupoids, and by Theorem 2.2.2 all such set are CFL's.

In order to show that any language in SAC^1 is recognized by uniform programs over a groupoid, we simply observe that Sudburough's logspace reduction from a language recognized by an AuxNPDA to a context-free language is a uniform projection. Recall the main steps of the proof.

Step 1 ([37]): We know that if $L \in$ LOGCFL, then it is recognized by an AuxNPDA M in space $c \log n$ and polynomial time. This machine can be simulated by a multiple-head PDA M_1 (i.e. constant working space): the working tape of M is divided into c blocks of size $\log n$; the content of each block is represented in M_1 by the position of a head. Some extra heads are used to manage the process.

Step 2 ([32]): If L is recognized by a NPDA M with k heads, then L is reducible to a language L' which is recognized by a NPDA M' with $\lceil k/2 \rceil$ heads. The reduction is the projection $\phi : w \mapsto (cwd)^{|cwd|}$. Two heads h_1, h_2 in M are simulated by one head h' in M': if h_1 is at position i and h_2 is at position j in w, then h' will be at position i of the j^{th} block cwd.

Step 3 ([75]): Any L recognized by a (2-way) NPDA M can be reduced

to a language L' recognized by a 1-way NPDA M'. The reduction is the uniform projection $\psi : w \mapsto (cwd)^{p(|w|)}$, where p is a polynomial bounding the time taken by M. The NPDA M' simulates M without moving its head left: whenever M moves left, M' simply moves to the next block cwd using its stack to find the proper position.

Step 4 ([75]): By applying Step 1, iterating Step 2, and finally applying Step 3, we can reduce any language $L \in \mathrm{SAC}^1$ to a context-free language L'. The reduction is a compositions of projections, and thus, it can be transformed into polynomial-length programs over any groupoid G that recognizes L'.

This shows that LOGCFL corresponds to the languages reducible to a context-free language via a polynomial-length projection. This projection can be made DLOGTIME-uniform by modifying the above four steps such that the length of each projection is a power of two. This is done by padding each word obtained at each step with the appropriate number of blank symbols using the technique of [6] (see also [12]). Since projections to context-free languages are also programs over groupoid, this yields Theorem 2.3.3.

However, we can give a stronger result. It is shown in [35] that there exists a context-free language L_0 such that any context-free language is reducible to L_0 via a nonerasing homomorphism. This immediately yields the following result.

Theorem 2.3.9 *[12] Let G_0 be any finite groupoid recognizing L_0. Then* $\mathrm{SAC}^1 = \mathcal{P}(G_0)$.

2.3.3 Groupoids and TC^0

The complexity class TC^0_k is the class of languages recognized by uniform unbounded circuits $(C_n)_{n\geq 0}$ of depth k constructed with MAJORITY gates. Without loss of generality, we can assume that all paths in C_n have the same length and that all negations are at the input gates. By definition, we have $\mathrm{TC}^0 = \bigcup_{k\geq 0} \mathrm{TC}^0_k$.

We will show that there exists a sequence of groupoids $(G_k)_{k\geq 1}$ such that

$\mathrm{TC}^0_k \subseteq \mathcal{P}(G_k) \subseteq \mathrm{TC}^0$. Observe that there exists no characterization of TC^0 in terms of finite semigroups (unless $\mathrm{TC}^0 = \mathrm{NC}^1$ or $\mathrm{TC}^0 = \mathrm{ACC}^0$).

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 $f_g(x) \in \{0,1,(,)\}^*$ 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 $f_g(x) = (f_{g_1}(x)) \cdots (f_{g_m}(x))$ 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 $k \geq 1$, define the grammar $D_k = (V_k, T, \pi_k, S)$ where $T = \{0,1,(,)\}$, $V_1 = \{S\} \cup \{M_1, F_1, Q_1, A_1, B_1\}$ 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 $i \geq 1$ and all $j > 1$ the following set of rules:

$$
\begin{aligned}
U_i &= \{M_i \to A_iB_i | B_iA_i,\\
&\quad\ P_i \to A_iP_i | A_iA_i,\\
&\quad\ Q_i \to B_iQ_i | B_iB_i\}\\
W_1 &= \{A_1 \to M_1A_1 | A_1M_1 | 1,\\
&\quad\ B_1 \to M_1B_1 | B_1M_1 | 0\}
\end{aligned}
\qquad
\begin{aligned}
W_j &= \{A_j \to M_jA_j | A_jM_j | LE_j,\\
&\quad\ B_j \to M_jB_j | B_jM_j | LF_j,\\
&\quad\ E_j \to M_{j-1}R | P_{j-1}R | A_{j-1}R,\\
&\quad\ F_j \to Q_{j-1}R | B_{j-1}R,\\
&\quad\ L \to (,\\
&\quad\ R \to)\}
\end{aligned}
$$

Then we define $\pi_k = \{S \to M_k | P_k | A_k\} \cup \bigcup_{1 \leq i \leq k}(U_i \cup W_i)$. 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 $|w|_{A_k} = |w|_{B_k}$ ($|w|_c$ denotes the number of occurrences of the symbol c in w). Starting from A_k (resp. B_k) we produce exactly all $w \in \{A_k, B_k\}^*$ such that $|w|_{A_k} = |w|_{B_k} + 1$ (resp. $|w|_{B_k} = |w|_{A_k} + 1$) and starting from P_k (resp. Q_k) we produce exactly all $w \in \{A_k, B_k\}^*$ such that $|w|_{A_k} \geq |w|_{B_k} + 2$ (resp. $|w|_{B_k} \geq |w|_{A_k} + 2$).

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 groupoid G_k from grammar D_k as

groupoid G was constructed from grammar M in the proof of Theorem 2.2.2.

To prove $\mathrm{TC}^0_k \subseteq \mathcal{P}(G_k)$, let $Y \in \mathrm{TC}^0_k$, let $x \in \{0,1\}^n$ and let g denote the output gate of a TC^0_k circuit C_n determining whether $x \in Y$. 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, that is, 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 uniform exactly as the "generalized expressions" obtained from an FO formula are made DLOGTIME-uniform [6, Proof of Theorem 9.1 ("1 $\Rightarrow$ 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 $\mathcal{P}(G_k) \subseteq \mathrm{TC}^0$, that is, $G_k(w)$ 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 $i \geq 0$, if there is a symbol E_i or F_i that is not immediately preceded by an L then the only possible evaluation for w is the zero of G_k \$ (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 non-terminal).

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 D_1 (where we have suppressed the subscripts for clarity):

1. For $|uv| > 1$, $X \stackrel{*}{\Rightarrow} uMv$ iff $X \stackrel{*}{\Rightarrow} uv$.
2. For $u \neq \epsilon$, $P \stackrel{*}{\Rightarrow} uP$ iff $P \stackrel{*}{\Rightarrow} uA$.
3. For $u \neq \epsilon$, $Q \stackrel{*}{\Rightarrow} uQ$ iff $Q \stackrel{*}{\Rightarrow} uB$.

4. If $X \stackrel{*}{\Rightarrow} uPv$ then $X = P$ and $v = \epsilon$.

5. If $X \stackrel{*}{\Rightarrow} uQv$ then $X = Q$ and $v = \epsilon$.

To evaluate a word w over G_1 we do the following:

(i) Verify that there is at most one P_1 (resp. Q_1) in w: using facts 4 and 5 the P_1 (resp. Q_1) must be at the end in which case the only possible evaluation different from \$ is P_1 (resp. Q_1). Then replace P_1 (resp. Q_1) by A_1 (resp. B_1) using fact 2 (resp. 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, let $MAJ(v)$ be true iff v has at least as many A_1's as B_1's, and define $EQUAL(v) = MAJ(v) \wedge MAJ(v')$.

We have the following observations.

- $M_1 \in G(v)$ iff $EQUAL(v)$
- $A_1 \in G(v)$ 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 $MAJ(v') \wedge \neg EQUAL(vA_1) \wedge \neg EQUAL(v)$

This proves our Claim I.

We now describe the TC^0 transformation from $w \in G_{i+1}^*$ to $v \in G_i^*$ 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 $2 \leq i \leq l+1$ 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.

2.3.4 Groupoids and NC^1

We have seen that if G is any nonsolvable semigroup then any language in NC^1 is recognized by programs over G. The smallest semigroup having this property is the alternating group A_5 and has order 60. In this section, we will see that there exists a groupoid of order 9 that recognizes, via uniform programs, any language in NC^1.

Let $a = (1,2)$ and $b = (1,2,3,4,5)$ be the cycle representation of two permutations on five points.

Lemma 2.3.10 *The symmetric group S_5 is generated by a and b.*

Proof. Define a transposition as a permutation of the form (i,j), where $1 \leq i \leq j \leq 5$. Let (i,j) be called an adjacent transposition whenever $j = i+1$. It is well known that any permutation is the product of transpositions. Moreover, any transposition (i,j) can be written as the product of adjacent transpositions since $(i,j) = (i,i+1)\cdots(j-1,j)(j-2,j-1)\cdots(i,i+1)$. So, it suffices to show that any adjacent transposition can be expressed with a and b. This is done with $(i,i+1) = b^{i-1}ab^{6-i}$. □

Consider now the groupoid $G = \{0,1,2,3,4,5,a,b,i\}$, where the product

is defined as follows.

	0	1	2	3	4	5	a	b	i
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	2	2	1
2	0	0	0	0	0	0	1	3	2
3	0	0	0	0	0	0	3	4	3
4	0	0	0	0	0	0	4	5	4
5	0	0	0	0	0	0	5	1	5
a	0	0	0	0	0	0	1	3	a
b	0	0	0	0	0	0	1	3	b
i	0	1	2	3	4	5	a	b	i

Let $w \in \{a,b\}^*$ and let p be the resulting permutation when w is evaluated in S_5. When w is evaluated in G, we obtain a nonzero element e only if the evaluation is done from left to right. Moreover, by construction, e is equal to the image of 1 under the permutation p.

Now, we know from Barrington's theorem that any language L in NC^1 is recognized by programs over S_5 such that the programs map any word in the language to the identity permutation, all other words being mapped on b.

Consider a program on S_5 accepting a language L. Since a and b are generators of S_5, we can suppose without loss of generality that the instructions of this program only yield a,b or the identity i. So, this program can be viewed as a program on G: A word is accepted whenever the program left-to-right evaluates to 1, it is rejected if it left-to-right evaluates to 2.

2.4 Tree languages

Let A be a finite alphabet and let $A^{(*)}$ be the free groupoid over A (see Example 2.1.1). A subset $T \subseteq A^{(*)}$ is called a (binary) *tree language* over A.[4]

The *yield* of a tree $t \in A^{(*)}$ is defined recursively as follows. The yield of $a \in A$ is a. If u and v are the respective yields of $x \in A^{(*)}$ and $y \in A^{(*)}$, then the yield of (xy) is uv. The yield of a tree language $T \subseteq A^{(*)}$ is the set $\text{yield}(T) = \{w \in A^* \mid w \text{ is the yield of some } t \in T\}$.

[4]In this work we will only consider this restricted kind of tree languages.

2.4.1 Regular tree languages

Let A be an alphabet and let T be a tree language over A. For any $a \in A$ and $t, t' \in T$, we denote by $t \cdot^a t'$ the tree in T obtained when every occurrence of a in t is replaced by t'. This operation can be generalized to tree languages T and T' by defining $T \cdot^a T' = \{t \cdot^a t' \mid t \in T, t' \in T'\}$. The iterative product ^{a} is defined by $T^a = \{a\} \cup T \cup T \cdot^a T \cup \ldots$.

A regular expression over A is a finite expression defined as follows.

- Finite subsets of $A^{(*)}$ are regular expressions.
- If $a \in A$ and s, t are regular expressions, then $s \cup t$, $A^{(*)} - s$, $s \cdot^a t$, and s^a are regular expressions.
- Nothing else is a regular expression over A.

Hence, regular expressions for tree languages are defined similarly to regular expressions for word languages. They differ only by the interpretation we give to the concatenation and the iterative product.

A *regular tree language* is a tree language defined, in the obvious way, from a regular expression. A regular expression is called star-free if it contains no iterative product.

The relationship between context-free languages and regular tree languages is given by the following proposition.

Proposition 2.4.1 ([34]) *A word language is context-free if and only if it is the yield of some regular tree language.*

2.4.2 Tree automata

There exist two ways of recognizing a tree language depending if we use a bottom-up or a top down method. Each case has its deterministic and nondeterministic versions.

A (nondeterministic) top-down tree automaton (NTDTA) is a 5-tuple $M = (Q, A, q_0, \delta, \alpha)$, where Q is a finite set of states, q_0 is the initial state, A is a

finite alphabet, $\delta : Q \to Q^2$ is the transition function, and $\alpha : A \to 2^Q$ is the final assignment. A NTDTA is called deterministic (and referred as DTDTA) if for all $a \in A$, $\alpha(a)$ is a singleton.

Given a NTDTA M and a tree $T \in A^{(*)}$, we mark each node of T with a state in Q as follows. We first, mark the root with the initial state q_0. Given any internal node marked with some state q, we mark the left child with q_1 and the right child with q_2, where $\delta(q) = (q_1, q_2)$.

A tree is said to be accepted by M if each leaf labeled with a letter $a \in A$ is marked with a state q such that $q \in \alpha(a)$. A tree language is recognized by a NTDTA M if and only if M accepts precisely those trees in T.

A (nondeterministic) bottom-up tree automaton (NBUTA) is a 5-tuple $M = (Q, A, \delta, \alpha, F)$, where Q is a finite set of states, A is a finite alphabet, $\delta : Q^2 \to 2^Q$ is the transition function, and $\alpha : a \to 2^Q$ is the initial assignment. When $\delta(p, q)$ and $\alpha(a)$ are singletons, for any $p, q \in Q$ and $a \in A$, M is said to be deterministic and is denoted DBUTA.

Given any tree $T \in A^{(*)}$, and any NBUTA M, we can assign to the root of T a subset $S \subseteq Q$ using functions α and δ. Then, T is said to be accepted by M if $S \cap F \neq \emptyset$. A tree language is recognized by a NBUTA M if and only if M accepts precisely those trees in T.

Proposition 2.4.2 *NTDTA, NBUTA, and DBUTA recognize the same class of tree languages which is the class of regular tree languages.*

Provably, DTDTA are weaker than the other tree automata. For example the tree language $\{(1,0),(0,1)\}$ cannot be recognized by any DTDTA (see [43]).

In our restricted context, recognition by DBUTA can be expressed in terms of finite groupoids.

Definition 2.4.3 *A tree language $T \in A^{(*)}$ is said to be recognized by a groupoid G if there exist a groupoid morphism $\varphi : A^{(*)} \to G$ and an accepting set $F \subseteq G$ such that $T = \{t \in A^{(*)} \mid \varphi(t) \in F\}$.*

Consequently, we have the following result.

Theorem 2.4.4 *A tree language is* regular *if and only if it is recognized by a finite groupoid.*

2.4.3 Syntactic groupoids

Let A be a finite alphabet and let x be a symbol not in A. A tree over $A \cup \{x\}$ is called a *special*[5] tree if it contains only one occurrence of x.

The *syntactic congruence* of a subset $L \subseteq A^{(*)}$ is denoted by $\sim_L$ and defined by $u \sim_L v$ if and only if for any special tree t over $A \cup \{x\}$ we have $t \cdot^x u \in L$ iff $t \cdot^x v$. Clearly $\sim_L$ is an equivalence relation on $A^{(*)}$. To see that it is a congruence, let $u_1 \sim_L v_1$ and $u_2 \sim_L v_2$. Then , $t \cdot^x (u_1 u_2) \in L$ iff $t \cdot^x (v_1 u_2) \in L$, and $t \cdot^x (v_1 u_2) \in L$ iff $t \cdot^x (v_1 v_2) \in L$. Hence $t \cdot^x (u_1 u_2) \in L$ iff $t \cdot^x (v_1 v_2) \in L$ and therefore $(u_1 u_2) \sim_L (v_1 v_2)$. Now for any tree language $L \subseteq A^{(*)}$ we define the *syntactic groupoid* of L as the quotient groupoid $A^{(*)}/\sim_L$. Many results in the algebraic study of recognizable languages in A^* (see [56]) can be directly translated for subsets of $A^{(*)}$. In particular we have:

Proposition 2.4.5 *Let T be a tree language and S_T its syntactic groupoid. Then, a groupoid G recognizes T if and only if $S_T \prec G$.*

Proof. Suppose that $T \subseteq G^{(*)}$ is recognized by G. Let t_1 and t_2 be two trees in $G^{(*)}$such that both of them evaluate to the same element in G. Then, for any special tree t, $t \cdot^x t_1 = t \cdot^x t_2$ in G. This means that $t_1 \sim_T t_2$ and that all trees evaluating to the same element in G belong to the same syntactic congruence class in $G^{(*)}$. This proves that the syntactic groupoid S_T of T divides G. □

As a corollary, we have the following result.

Theorem 2.4.6 *A tree language $T \subseteq A^{(*)}$ is regular if and only if its syntactic groupoid is finite.*

[5]This terminology comes from [78]

2.4.4 Varieties of tree languages

Let A be a finite alphabet and let x be a symbol not in A. For any special tree w over $A \cup \{x\}$ and for any tree language T, we define the *quotient* of T by w as the set $\{t \in A^{(*)} \mid w \cdot^x t \in T\}$.

A class $\mathcal{V}$ of tree languages forms a variety if it is closed under Boolean operations, inverse homomorphism, and quotient.

Proposition 2.4.7 ([71]) *The class of regular tree languages forms a variety.*

Varieties of groupoids are defined in the same way as varieties of semigroups: they are the classes of groupoids closed under finite direct product and division.

The theorem of varieties still hold in the tree language context.

Theorem 2.4.8 ([71]) *There is a bijection between varieties of regular tree languages and varieties of finite groupoids.*

In particular any variety of finite groupoids is generated by a set of syntactic groupoids. More information on syntactic algebras and varieties of general tree languages can be found in [71].

2.5 Restricted parenthesization

The relationship between word languages and tree languages discussed in Section 2.4 provides an interesting way of restricting the power of programs over groupoids. First, we generalize Definition 2.2.1 as follows.

Definition 2.5.1 *Given $T \subseteq A^{(*)}$, a language $L \subseteq A^*$ is said to be T-*recognized *by a groupoid G if there exist a morphism $\varphi : A^{(*)} \to G$ and an accepting set $F \subseteq G$, such that $L = \{\text{yield}(t) \mid (t \in T) \wedge (\varphi(t) \in F)\}$. T-recognition by programs over a groupoid is defined in the obvious way.*

Clearly, Definition 2.2.1 is equivalent to Definition 2.5.1 when $T = A^{(*)}$. This is however not the case for arbitrary T. Observe first that T may be regular or not.

Fact 2.5.2 *In Definition 2.5.1, if T is regular, then L is context-free.*

However, the converse is not true: L can be context-free while T is even not computable. For example, let G be a semigroup and let $T \subseteq A^{(*)}$ be any set of trees such that $A^* = \text{yield}(T)$. Then, any language $L \subseteq A^*$ that is $G^{(*)}$-recognized by G is also T-recognized by G, because G is associative.

When T is regular, the class of languages T-recognized by a finite groupoid G can form a strict subset of the context-free languages.

Example 2.5.1 Let G be a finite groupoid and let $\text{LTR} \subseteq G^{(*)}$ be the set of trees defined recursively as follows. Any $g \in G$ is also in LTR; if $t \in \text{LTR}$ and $g \in G$ then $(tg) \in \text{LTR}$; nothing else is in LTR. Thus, LTR corresponds exactly to the *left-to-right* parenthesizations of words over G.

It is a simple exercise to verify that a language is LTR-recognized by G if and only if it is regular.

Example 2.5.2 Let G be a finite groupoid and, for any $k \geq 1$, let $\text{RD}_k \subseteq G^{(*)}$ be defined as follows: $\text{RD}_1 = \text{LTR}$; $\text{RD}_{k+1} = \text{RD}_k \cup T_k$, where T_k consists of those trees that can be decomposed as $(\cdots((t_1t_2)t_3)\cdots t_n)$ where $t_1, \ldots, t_n \in \text{RD}_k$.

We say that a language L is recognized in *constant right-depth* by a groupoid G if there exists $k \geq 1$ such that L is RD_k-recognized by G.

Lemma 2.5.3 ***If a language L is recognized in constant right-depth by a finite groupoid G then it is regular.***

Proof. Observe that we can determine if a word $w \in G^+$ evaluates to some element in F with a nondeterministic pushdown automaton working as follows.

```
Read the first input; Set it as the current value;
While there is a nonread input or the stack is not empty do
      Choose nondeterministically between
      (1) Push the current value in the stack,
          read next input, and
          set it as the new current value;
      (2) Pop the top of the stack,
          multiply it with the current value, and
          set the result as the new current value;
If the current value is in F then accept
Else reject
```

We observe that the program's nondeterministic choices, on input w, induce a tree T whose yield is w and that evaluates to an element in F if and only if the program accepts w. By assumption, there exists a sequence of choices such that the right-depth of T is constant. Given that sequence of choices, the execution of the program on w consists essentially of a depth-first search evaluation of T by the left: an element is pushed in the stack precisely when a right edge is taken in direction of the leaves, and the same element is popped only when the same edge is used in the reverse direction. Thus, the stack never needs to be of height larger than the right-depth of T, which is constant. Hence L is regular, since a nondeterministic automaton with a stack of constant height can be simulated by a finite automaton. □

In the above examples we have used a regular set of trees. However, it is also interesting to remove this restriction and to chose a nonregular set T.

Example 2.5.3 Let G be a finite groupoid and let $T_c \subseteq G^{(\bullet)}$ be the set of trees of depth smaller than $c \log n$ where n is the number of leaves. Using the pigeon-hole Principle we easily see that T_c is not regular. However, any language L that is T_c-recognized by G is also recognized by a nondeterministic

pushdown automaton using a stack of height at most $c \log n$, on input of length n. Thus, L belongs to NL.

Next chapter is devoted to an important kind of regular restriction that we call linear.

2.6 Parenthesized programs

The previous discussion on tree languages motivates the investigation of programs over groupoids that are deterministic in the sense that parenthesis appear explicitly and so do not need to be guessed.

Definition 2.6.1 *Let A be a finite alphabet, let G be a groupoid and let F be a subset of G. For any integer n, let $\mathcal{I}_n$ be the set of all instructions of the form (i, f) where $1 \leq i \leq n$ and $f : A \to G$ is a function. We define a parenthesized program P_n over G as a tree T_n over $\mathcal{I}_n$. On input w of length n, the instructions of P_n yield elements of G that can be multiplied according to the structure of T_n. Then, P_n is said to accept w if the resulting element belongs to F.*

Another way of seeing parenthesized programs is to consider the existence of constant instructions that produce open or closed parenthesis. On a given input, the program yields a well parenthesized expression over elements of G.

A language $L \subseteq A^*$ is said to be recognized by a family of parenthesized programs $(P_n)_{n \geq 0}$ over a groupoid G if P_n accepts precisely those words in $L \cap A^n$.

Definition 2.6.2 *A polynomial length family of parenthesized programs is said to be uniform if, given (w, k), we can compute in time $O(\log |w|)$ the length of $P_{|w|}$ and its k^{th} instruction (which can be a parenthesis).*

Theorem 2.6.3 ([12]) *A language is in NC^1 if and only if it is recognized by a uniform parenthesized program over a finite groupoid.*

Proof. To prove that any language recognized by a uniform family of parenthesized programs is in NC^1, it suffices to show that any well-parenthesized expression over a finite groupoid can be evaluated in NC^1. This follows from Buss' result [17] that any parenthesis context-free language belongs to NC^1. Recall that a parenthesis context-free language is a language generated by a grammar whose productions have the form $A \rightarrow (\alpha)$, where A is a variable and α contains no parenthesis. Given a groupoid G, we can define a grammar D_G whose set of variables is G and which contains a production $a \rightarrow (bc)$ for any product $bc = a$ in G. Then, a well-parenthesized expression w evaluates to $a \in G$ if and only if w is generated by a in D_G.

It remains to show that any language in NC^1 is recognized by a uniform family of parenthesized programs over some groupoid.

Let NAND $= \{0,1\}$ be the groupoid with product $1 \cdot 1 = 0$ and $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 1$. It is not difficult to see that polynomial-length parenthesized programs over NAND recognized precisely those languages in NC^1 Observe first that the negation NOT of a bit x can be expressed over NAND by (xx), and the AND and the OR of two bits x and y can be expressed over NAND by $((xy)(xy))$ and $((xx)(yy))$, respectively.

Hence, any Boolean formula f can be expressed as a well-parenthesized expression w over NAND. Uniformity can be obtained by adding extra parenthesis to control the growth of w as a function of the depth of f (see [12]). □

An interesting observation concerning parenthesized programs is that, in this setting, any two isotopic groupoids have the same computational power.

Theorem 2.6.4 *Let $(G,\cdot)$ and $(G,*)$ be two isotopic groupoids such that $(G,*)$ possesses an identity 1. Then any programs over $(G,*)$ of length $l(n)$ can be simulated by a program over $(G,.)$ having length $O(l(n))$.*

Proof. By Theorem 2.1.9, there exist elements $f,g \in G$ such that for any $x,y \in G$, $x * y = xR^{-1}(f) \cdot yL^{-1}(g)$. Observe that both $R^{-1}(f)$ and

$L^{-1}(g)$ are permutations. Hence, there exists a positive integer k such that $L^{-1}(g) = L^k(g)$ and $R^{-1}(f) = R^k(f)$. We thus have $x * y = xR^k(f) \cdot yL^k(g)$, and so a well-parenthesized expression over $(G, *)$ can be expressed with an expression over $(G, \cdots)$ with the length increased by a factor of $k + 1$. □

In the above theorem, it is essential that $(G, \cdot)$ has an identity. For example, let $U_1 = \{0, 1\}$ be the semigroup with product $0 \cdot 0 = 0$ and $0 \cdot 1 = 1 \cdot 0 = 1 \cdot 1 = 1$. Then U_1 is isotopic to NAND since the product of xy in U_1 can be expressed by $(x \cdot 1)(y \cdot 1)$ in NAND. However, there are programs over NAND that cannot be simulated by programs over U_1.

For example, the expression $P_2 = (x(y1))((x1)y)$ evaluates to 0 (when the product is taken over NAND) only when $xy \in \{00, 11\}$.

Suppose that there exists a program P_2 over U_1 that recognizes $\{00, 11\}$. Since U_1 is commutative, we can assume without loss of generality that P_2 has only 2 instructions: the instruction I_1 looks for x and the instruction I_2 looks for y.

Let a_0, a_1 be the elements produced by I_1 when x has value 0 and 1, respectively. Similarly, let b_0, b_1 be the elements produced by I_2 when y is respectively 0 or 1.

On input 01, P_2 yields $a_0 b_1$, and on input 10, it yields $a_1 b_0$. If the accepting element of P_2 is 0, then, we must have $a_0 = a_1 = b_0 = b_1 = 0$, and P_2 accepts any input. Thus, the accepting element must be 1.

Since P_2 accepts 00, there must be one of a_0 or b_0 which is 1. Suppose that $a_0 = 1$ (the other case is treated similarly). Then, on input 01, P_2 produces $a_0 b_1 = 1 b_1 = 1$. This contradicts the assumption that 01 is not accepted by P_2, proving that no program over U_1 can recognize $\{00, 11\}$.

As a corollary of Theorem 2.6.4, we observe that restricting groupoids to be commutative does not remove the power of parenthesized programs.

Theorem 2.6.5 *Any language L recognized by a parenthesized program over a groupoid G is also recognized by a parenthesized program Q over a commutative groupoid H.*

Proof. Let k be the order of G^1 and let $G' = \{a' \mid a \in G^1\}$.
Define the commutative groupoid $H = G^1 \cup G'$ as follows.

- 1 is the identity of H
- $1'a = a1' = a'$, for all $a \in G^1$
- $ab' = b'a = 1'$, for all $a, b \in G - \{1\}$
- $ab = b'a' = c$, where $ab = c$ in G^1

Define the isotopy (α, ι, ι), where $\alpha(a) = a'$, $\alpha(a') = a$, and ι is the identity mapping. On can verify that using this isotopy, we obtain from H a groupoid H' that is commutative.

Since G is a subgroupoid of H, L is recognized by parenthesized programs over H. Moreover, by Theorem 2.6.4, L is also recognized by parenthesized programs over H'. □

Another important criterion determining the computational power of parenthesized programs over a groupoid concerns the multiplication monoid.

Theorem 2.6.6 *Let S be a finite semigroup and let a and b be two elements of S. Let P_n be a program over S such that P_n evaluates to a whenever it accepts its input, and P_n evaluates to b otherwise. Then P_n can be simulated by a parenthesized program over G, for any groupoid G with identity 1 such that S is isomorphic to a subsemigroup of $\mathcal{M}(G)$. Moreover, the length of P_n is increased only by a constant factor.*

Proof. Let M be a subsemigroup of $\mathcal{M}(G)$, and let $\phi : S \to M$ be an isomorphism. Let $U = \phi(a)$, $V = \phi(b)$ and $g \in G$ be such that $gU \neq gV$. Since $\mathcal{M}(G)$ is generated by the set $\{R(x), L(x) : x \in G\}$ and contains the identity $R(1) = L(1)$, we can transform this program in such a way that each instruction is of the form $\langle i, R(x), R(y)\rangle$ or $\langle i, L(x), L(y)\rangle$. The length of the resulting program will be increased only by a constant factor.

This new program over $\mathcal{M}(G)$ can be transformed into a parenthesized program over G. This can be done recursively as follows. A program consisting of a single instruction of the form $\langle i, R(x), R(y)\rangle$ or $\langle i, L(x), L(y)\rangle$ is transformed into $(\langle 1, g, g\rangle\langle i, a, b\rangle)$ or $(\langle i, a, b\rangle\langle 1, g, g\rangle)$, respectively. Let W be any sequence of instructions over $\mathcal{M}(G)$ and let w be its transformation over G. Then, the sequences $W\ \langle i, R(x), R(y)\rangle$ and $W\ \langle i, L(x), L(y)\rangle$ are transformed into $(w\ \langle i, a, b\rangle)$ and $(\langle i, a, b\rangle\ w)$ respectively. The resulting parenthesized program accepts if it evaluates to gU and rejects if it evaluates to gV.

□

Chapter 3

Linearity

3.1 Linear recognition

Let A be a finite alphabet. The *linear subset* of $A^{(*)}$ is the set A^{LIN} defined as follows: any $a \in A$ is in A^{LIN}; if $u \in A^{LIN}$ and $a \in A$ then (au) and (ua) are in A^{LIN}. A tree $T \in A^{\mathrm{LIN}}$ is called a *linear tree over* A, and a subset $L \subseteq A^{\mathrm{LIN}}$ is called a *linear tree language*. In other words a tree is linear if at least one child of every internal node is a leaf.

We denote by $G_{LIN}(w)$ the set of all evaluations of w following all possible linear parenthesizations. If $g \in G_{\mathrm{LIN}}(w)$, then we say that w linearly evaluates to g.

A context-free language is called *linear* if it is generated by a context-free grammar such that every production is of the form $A \rightarrow w$, where A is a variable and w contains at most 1 variable. In the following, we call a context-free language that is linear a linear language.

Linear languages can also be characterized using groupoids. In order to do this we need to restrict the notion of recognition by groupoid.

Definition 3.1.1 *We say that a language $L \subseteq A^*$ is linearly recognized by a groupoid G if there exist a subset $F \subseteq G$ and a morphism $\phi : A^* \rightarrow G^*$ such that $L = \{w \in A^* \mid G_{LIN}(\phi(w)) \cap F \neq \emptyset\}$.*

In other words, linear recognition corresponds precisely to A^{LIN}-recognition. Observe that when the groupoid G is a semigroup then linear and general

recognitions are equivalent due to the associativity of G.

Theorem 3.1.2 *A language is linear if and only if it is linearly recognized by a finite groupoid.*

Proof. Let D be an invertible linear context-free grammar in Chomsky normal form generating a language L. We saw in the proof of Theorem 2.2.2 how to construct a finite groupoid G from D such that L is recognized by G. One can check that any non-linear tree over G evaluates to the absorbing element which does not belong to the accepting set. Hence, L is linearly recognized by G.

Suppose now that A is some alphabet and $L \subseteq A^*$ is a language linearly recognized by a finite groupoid G. More specifically let $F \subseteq G$ and let $\phi : A^* \to G^*$ be an alphabetic morphism such that $L = \{w \in A^* \mid G_{LIN}(\phi(w)) \cap F \neq \emptyset\}$. Then, we can construct a linear grammar M for L as follows. Let $G \cup \{S\}$ be the set of variables where S is not in G. Let A be the set of terminals of M, and let S be the start variable. For each $X, Y, Z \in G$ such that $Z = XY$ we define the productions $Z \to Xb$, $Z \to aY$ and $Z \to ab$ for all $a \in \phi^{-1}(X)$ and $b \in \phi^{-1}(Y)$. Finally, we define the production $S \to Z$ for all $Z \in F$. Clearly, M is a linear grammar generating the language L. □

Definition 3.1.3 *We say that a language is linearly recognized by a family of programs P over a groupoid G if it is G^{LIN}-recognized by P.*

Proposition 3.1.4 ([Su75]) *There exist a linear language L_n such that L_n is complete for* NL. *Moreover, the reduction is a DTIME($\log n$)-uniform projection.*

Theorem 3.1.5 *A language is in* NL *if and only if it is linearly recognized by uniform polynomial-length programs over a finite groupoid.*

Proof. Observe that any linear language belongs to NL since we only need two pointers to parse a word. The other direction is given by Theorem 3.1.2

and Proposition 3.1.4. □

Another consequence of Theorem 3.1.2 and Proposition 3.1.4 is the following observation.

Theorem 3.1.6 *There exists a groupoid G_0 such that a language is in* NL *if and only if it is linearly recognized by a uniform family of programs over G_0.*

3.2 Linear and weakly linear groupoids

Definition 3.2.1 *A groupoid G is called* weakly linear *if it possesses an absorbing element 0 and $(ab)(cd) = 0$, for any $a, b, c, d \in G$.*

Theorem 3.2.2 ([53]) *A language is linear if and only if it is recognized by a weakly linear groupoid.*

Proof. Let L be a language linearly recognized by a finite groupoid G, and let G' be a copy of the set G. Let $a, b, c \in G$ be such that $ab = c$ in G, and let $a', b', c' \in G'$ be the respective copies of a, b and c. We define the following product on $H = G \cup G' \cup \{0\}$.

- $a \cdot b = c'$
- $a \cdot b' = a'b = c'$
- All other products yield 0

Groupoid H is weakly linear, and any word $w \in G^*$ of length two or more linearly evaluates to an element $a \in G$ if and only if w can be evaluated to a' in H. Thus, if the accepting set of G is $F \subseteq G$, then H recognized L with the accepting set $F \cup \{f' \in G' \mid f \in F\}$. □

The construction of groupoid H in the proof of the above theorem preserves some algebraic properties of G. For example, G is commutative if and only if H is commutative. The next theorem shows that the multiplication monoids of G and H have isomorphic subgroups.

Theorem 3.2.3 *Let G and H be as in Theorem 3.2.2. A group is isomorphic to a subgroup of $\mathcal{M}(G)$ if and only if it is isomorphic to a subgroup of $\mathcal{M}(H)$.*

Proof. For any $x \in G$, let $D_G(x) \in \{R_G(x), L_G(x)\} \subseteq \mathcal{M}(G)$ and let $D_H(x) \in \{R_H(x), L_H(x)\} \subseteq \mathcal{M}(H)$. Let U be an element of $\mathcal{M}(G)$. There exist $a_1, \ldots, a_k \in G$ such that $U = D_G(a_1) \cdots D_G(a_k)$. We observe that for any $a, b \in G$, $aU = b$ if and only if $aD_H(a_1) \cdots D_H(a_k) = a'D_H(a_1) \cdots D_H(a_k) = b'$. This shows that the homomorphism $\phi : \mathcal{M}(G) \rightarrow \mathcal{M}(H)$ defined by $\phi(R_G(x)) \rightarrow R_H(x)$ and $\phi(L_G(x)) \rightarrow L_H(x)$ is injective. Hence any group in $\mathcal{M}(G)$ is isomorphic to a group in $\mathcal{M}(H)$.

To prove the other direction, we just observe that no element $D_H(x)$, for $x \in H-G$, can be used to generate an element of a subgroup of $\mathcal{M}(H)$. Indeed, if $W \in \mathcal{M}(H)$ can be expressed as $W = XD_H(x)Y$, where $X, Y \in \mathcal{M}(H)^*$, then WW maps any element of H to 0. □

Corollary 3.2.4 *Let G and H be as in Theorem 3.2.2. Then, $\mathcal{M}(G)$ is aperiodic (resp. solvable) if and only if $\mathcal{M}(H)$ is aperiodic (resp. solvable).* □

Recall that a solvable group has depth k if its composition series has length k. In particular, a solvable group has depth 2 if and only if its commutator subgroup is commutative. Theorem 3.1.5 and Theorem 3.2.2 have the following interesting consequences.

Proposition 3.2.5 *Any language recognized by the Lie groupoid of a solvable group of depth 2 is in* NL.

Proof. Let C be the Lie groupoid of a solvable group G of depth 2. Let 0 be the identity of G. For any $a, b, c, d \in G$, we have $[[a, b][c, d]] = 0$. This is reflected in C by the fact that $(ab)(cd) = 0$, where 0 is the absorbing element of C.

Thus C is a weakly linear groupoid that can only recognize languages in NL. □

The definition of weakly linear groupoids is not satisfactory on one point.

That is, semigroups are not weakly linear in spite of the fact that linear and general recognition by semigroups are equivalent. For this reason, we define a stronger notion of linearity.

Definition 3.2.6 *A groupoid G is called* linear *if for any word $w \in G^*$, we have $G_{\text{LIN}}(w) = G(w)$.*

Theorem 3.2.7 *A language is recognized by a linear groupoid if and only if it is recognized by a weakly linear groupoid.*

Proof. If L is recognized by a linear groupoid B, then L is also linearly recognized by B. By Theorem 3.2.2, L is recognized by a weakly linear groupoid.

The other direction is not immediate because there is no guaranty that a word over a weakly linear groupoid can be linearly evaluated to the absorbing element.

Suppose that L is recognized by a weakly linear groupoid G with accepting set F. Define $H = \{\langle a, b\rangle \mid a, b \in G\} \cup \{\lfloor a, b\rfloor \mid a, b \in G\} \cup G \cup \{0\}$, where 0 is a new element. For any $a, b, c, d \in G$, we define a product on H as follows.

- $a \cdot b = \lfloor a, b\rfloor$
- $\lfloor a, b\rfloor \cdot c = \langle (ab)c, a(bc)\rangle$
- $c \cdot \lfloor a, b\rfloor = 0$
- $c \cdot \langle a, b\rangle = \langle ca, cb\rangle$
- $\langle a, b\rangle \cdot c = \langle ac, bc\rangle$
- All other products yield 0.

We observe that any word $w \in G^*$ of length at least three linearly evaluates, in H, to 0 and to some elements of the form $\langle x, y\rangle$. Any non-linear evaluation of w yields 0. Thus, L is recognized by H using the accepting set $F \cup \{\lfloor a, b\rfloor \mid ab \in F\} \cup \{\langle a, b\rangle \mid a \in F \text{ or } b \in F\}$. □

Corollary 3.2.8 *A language is linear if and only if it is recognized by a linear groupoid.* □

3.3 Transducers and Groupoids

Let M be a semigroup (possibly infinite). The family of rational subsets of M denoted RAT(M) is defined as

1. $\emptyset, \{m\} \in$ RAT(M) for all $m \in M$
2. If $X, Y \in$ RAT(M) then $X \cup Y, XY \in$ RAT(M)
3. If $X \in$ RAT(M) then $X^* \in$ RAT(M)
4. Nothing else is in RAT(M).

Given alphabets A and B, rational subsets of the semigroup $A^* \times B^*$ are called *rational relations* over X and Y.

A *rational transducer* is a 6-tuple $T = (X, Y, Q, q_0, F, \delta)$ where X and Y are finite alphabets, Q is a finite set of states, $q_0 \in Q$ is called the initial state, $F \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times (X^* \times Y^*) \times Q$ is a finite set of labeled edges. We represent T as a finite graph where the vertices are the elements of Q, and the edges are labeled with elements of $X^* \times Y^*$. An element $(u, v) \in X^* \times Y^*$ is said to be accepted by the above transducer if and only if there exist an element $q_F \in F$ and a sequence of edges $(q_0, q_{i_1})(q_{i_1}, q_{i_2}) \cdots (q_{i_n}, q_F)$ such that the product of the labels is (u, v). Otherwise (u, v) is rejected. The relation $R_T \subseteq X^* \times Y^*$ accepted by T is the set of pairs $(u, v) \in X^* \times Y^*$ accepted by T.

Let A be a finite alphabet. For any integer n and any word $w = x_1 \cdots x_n \in A^n$, we define the *mirror image* of w as the word $\tilde{w} = x_n \cdots x_1$. The proof of the following theorems can be found in [13].

Theorem 3.3.1 *A relation $R \subseteq X^* \times Y^*$ is rational if and only if it is accepted by some rational transducer.* □

Theorem 3.3.2 ([60]) *A language L is linear if and only if there is a rational relation R such that $L = \{uv \mid (\tilde{u}, v) \in R\}$.* □

In [13] it is shown that any rational relation R can be accepted by a transducer $T = (X, Y, Q, q_0, F, \delta)$ such that δ is restricted to $Q \times (X \cup \{\epsilon\}) \times (Y \cup \{\epsilon\}) \times Q$. It is is not difficult, given a transducer with the above property, to find an equivalent transducer T (i.e. a transducer accepting the same relation) such that δ is further restricted to $Q \times ((X \times \{\epsilon\}) \cup (\{\epsilon\} \times Y)) \times Q$. We then say that T is in *normal form.*

The interest of this normal form is that it allows us to see any normal transducer $T = (X, Y, Q, q_0, F, \delta)$ as a finite automaton accepting some language $L_T \subseteq \{(X \times \{\epsilon\}) \cup (\{\epsilon\} \times Y)\}^*$. We can thus see T as both a relation and a language recognizer. We define the *transformation monoid* of a normal transducer as the transformation monoid of the induced automaton (e.g. see [56] for a discussion of the transformation monoid of finite automata).

Let G be a finite groupoid, let $A \subseteq G$, and let $L \subseteq A^*$ be a language linearly recognized by G with accepting set $F \subseteq G$. Moreover, let $M_A = \{R(a), L(a) \mid a \in A\}$. We define two monoid morphisms $\alpha, \beta : M_A^* \to A^*$ as follows. For all $a \in A$, $\alpha(R(a)) = \epsilon$, $\alpha(L(a)) = a$, $\beta(R(a)) = a$, and $\beta(L(a)) = \epsilon$.

Definition 3.3.3 *The* derived *language of L according to G is defined as the set $D_L = \{W \in M_A^* \mid 1W \in F\}$.*

Definition 3.3.4 *The* derived *relation of L according to G is defined as the set $R_L = \{(\alpha(W), \beta(W)) \mid W \in D(L)\}$.*

We will often assume that a word $W \in D_L$ represents, in the natural way, a linear tree $T \in A^{\mathrm{LIN}}$ with the yield of T being $\alpha(\widetilde{W})\beta(W)$.

Proposition 3.3.5 *The language D_L is recognized by $\mathcal{M}(G)$.*

Proof. It suffices to use the accepting set $\{U \in \mathcal{M}(G) \mid 1U \in F\}$ □

Proposition 3.3.6 *R_L is a rational relation such that $L = \{uv \mid (\tilde{u}, v) \in R_L\}$.*

Proof. To prove that R_L is rational, define $K = A \times \{\epsilon\} \cup \{\epsilon\} \times A$ and consider the morphism $\psi : M_A^* \to K^*$ defined by $\psi(R(a)) = (\epsilon, a)$ and $\psi(L(a)) = (a, \epsilon)$. Since D_L is regular then there exists a regular expression for $\psi(D_L)$. Now this regular expression over K^* can be viewed as a rational relation over $A^* \times A^*$, that is easily seen to be R_L.

A simple induction shows that $L = \{uv \mid (\tilde{u}, v) \in R_L\}$. □

In the rest of this section, we will compare the groupoids that linearly recognize a language L with the transducers that recognize a relation R such that $L = \{uv \mid (\tilde{u}, v) \in R\}$. This comparison will be done via the multiplication monoid of the groupoids and the transformation monoid of the transducers.

Theorem 3.3.7 *Let $L \subseteq A^*$ be a language linearly recognized by a groupoid G. There exists a normal transducer T for R_L such that the transformation monoid of T divides $\mathcal{M}(G)$.*

Proof. Let $M = (G^1, M_A, \delta, 1, F)$ be a finite automaton, where $M_A = \{L(a), R(a) \mid a \in A\}$, $F \subseteq G$ is the accepting set, and δ is defined by $\delta(g, R(a)) = ga$ and $\delta(g, L(a)) = ag$, $g \in G^1$ and $a \in A$. Then, the transformation monoid of M is generated by M_A and thus divides $\mathcal{M}(G)$.

Define the rational transducer $T = (G^1, A \times \{\epsilon\}, \{\epsilon\} \times A, \rho, 1, F)$ from M by substituting the edge label (ϵ, a) for $R(a)$ and (a, ϵ) for $L(a)$, $a \in A$. Then, T is a transducer recognizing R_L and having a transformation monoid isomorphic to that of M. □

The converse of the above result needs more work to be proved. Before doing so, we need to discuss some technicalities. We saw that for any linear language L linearly recognized by a groupoid G, there exists a finite automaton recognizing the derived language of L according to G. However, a language D recognized by an arbitrary (normal) transducer is not necessarily the derived language D_L of some language L recognized by a groupoid G. More precisely,

necessary conditions for a language $D \subseteq ((A \times \{\epsilon\}) \cup (\{\epsilon\} \times A))$ recognized by a transducer T to be the derived language of some linear language are:

1. for any $a \in A$, both or none of (a, ϵ) and (ϵ, a) are in D.

2. for any $a, b \in A$ and any $m \in (A \times \{\epsilon\} \cup \{\epsilon\} \times A)^*$, all or none of $(a, \epsilon)(\epsilon, b)m$, $(\epsilon, a)(\epsilon, b)m$, $(b, \epsilon)(a, \epsilon)m$ and $(\epsilon, b)(a, \epsilon)m$ are in D.

The above conditions are justified by the fact that a word $W \in D_L^*$ belongs to D_L if and only if $\alpha(\widetilde{W})\beta(W)$ belongs to L.

A normal transducer recognizing a language that satisfies the above conditions will be called a *special* transducer.

The next proposition is given for groupoids with aperiodic and solvable multiplication monoids but also applies to any groupoid whose multiplication monoid belongs to a *group variety*.[1]

Proposition 3.3.8 *Let R be a relation recognized by a special transducer T with aperiodic (solvable) transformation monoid. Then the language $L = \{uv \mid (\widetilde{u}, v) \in R\}$ is linearly recognized by a finite groupoid with aperiodic (solvable) multiplication monoid.*

Proof. Let $B = (A \times \{\epsilon\} \cup \{\epsilon\} \times A)$ and let $K \subseteq B^*$ be the language recognized by T viewed as an automaton. Let $M = (S, B, i, \delta, F)$ be the minimal automaton recognizing K. It should be clear that M is also a transducer recognizing R. Also, it is well known from automata theory that the transformation monoid of M is still aperiodic (solvable).

Remark that because M is minimal and because both or none of $(\epsilon, a)V$ and $(a, \epsilon)V$ belong to K, for any V, then starting from the initial state of M, both (a, ϵ) and (ϵ, a) go to the same state.

We define a groupoid G on the set $A \cup S \cup \{0\}$, where 0 is a new element. The product in G is defined as follows. Let $s, t \in S$ and $a, b \in A$.

1. $ab = s$, where $s = \delta(\delta(i, (\epsilon, a)), (\epsilon, b))$

[1] Given any variety of groups $\mathbf{V}$, the class of all monoids containing only subgroups in $\mathbf{V}$ forms a variety of monoids called a group variety (see [56]).

2. $as = t$, where $t = \delta(s, (a, \epsilon))$

3. $sa = t$, where $t = \delta(s, (\epsilon, a))$

4. All other product yield 0.

To see that this groupoid linearly recognizes L, we just have to show that the multiplication monoid $\mathcal{M}(G)$ of G recognizes K with the accepting set $\{U \in \mathcal{M}(G) \mid 1U \in F\}$. This follows from the observation that the transformation monoid of M is isomorphic to the submonoid $D \subseteq \mathcal{M}(G)$ generated by $L(a)$ and $R(a)$ for all $a \in A$. Indeed, the action of $L(a)$ (resp. $R(a)$) and (a, ϵ) (resp. (ϵ, a)) are identical on S. Furthermore, by the above remark, we have that for any $b \in A$ there exists $s \in S$ such that $L(a)$ and $R(a)$ map b and s to the same element in S, for all $a \in A$. In other words, all transformations in D act identically on b and on s.

To complete the proof it suffices to show that all groups contained in $\mathcal{M}(G)$ are isomorphic to a subgroup of the transformation monoid of M. This follows from the fact that any element not in D evaluates to the absorbing element of $\mathcal{M}(G)$ when multiplied by itself. □

Lemma 3.3.9 *Let R be a relation recognized by a normal transducer T with aperiodic (solvable) transformation monoid. There exists a special transducer T' with aperiodic (solvable) transformation monoid that recognizes a relation R' such that $\{uv \mid (\tilde{u}, v) \in R\} = \{uv \mid (\tilde{u}, v) \in R'\}$*

Proof. Let $B = (A \times \{\epsilon\} \cup \{\epsilon\} \times A)$, and let $K \subseteq B^*$ be the language recognized by T viewed as an automaton. The language K is recognized by a finite monoid M with accepting set $F \subseteq M$. For any $(a, b), (c, d) \in B$, let

$$K(a, b, c, d) = \{w \in B^* \mid (a, b)(c, d)w \in K\}$$

It is well known (see [56]) that $K(a, b, c, d)$ is also recognized by M.

For any $a, b \in A$ define the sets

$$Q(a) = \{(a, \epsilon), (\epsilon, a)\}$$

$$L(a,b) = \{(a,\epsilon)(\epsilon,b),(\epsilon,a)(\epsilon,a),(b,\epsilon)(a,\epsilon),(\epsilon,b)(a,\epsilon)\}$$

Let $L_R = \{uv \mid (\bar{u},v) \in R\}$ and let $K' \subseteq B^*$ be defined as follows. $K' = P \cup L_1 \cup L_2 \cup L_3 \cup L_4$, where

$$P = \bigcup_{a \in L_R} Q(a)$$

$$L_1 = \bigcup_{a,b \in A} L(a,b)K(a,\epsilon,\epsilon,b)$$

$$L_2 = \bigcup_{a,b \in A} L(a,b)K(\epsilon,a,\epsilon,b)$$

$$L_3 = \bigcup_{a,b \in A} L(a,b)K(b,\epsilon,a,\epsilon)$$

$$L_4 = \bigcup_{a,b \in A} L(a,b)K(\epsilon,b,a,\epsilon)$$

Since $L(a,b)$ is finite for every $a,b \in A$, and since languages recognized by aperiodic (resp. solvable) monoids are closed under finite union and concatenation with finite sets, the language K' is recognized by some aperiodic (resp. solvable) monoid. This automaton can be seen as a special transducer recognizing a relation R'. One easily check that $\{uv \mid (\bar{u},v) \in R\} = \{uv \mid (\bar{u},v) \in R'\}$. □

Proposition 3.3.8 and Lemma 3.3.9 together yield the following theorem.

Theorem 3.3.10 *Let R be a relation recognized by a normal rational transducer with aperiodic (solvable) transformation monoid. Then $L = \{xy \mid (\bar{x},y) \in R\}$ is linearly recognized by a finite groupoid with aperiodic (solvable) multiplication monoid.* □

3.4 Hierarchy of linear languages

Let L be a regular language over the alphabet $\{0,1\}$, and let T be a linear tree language over $A = \{0,1,a,b\}$ such that the yield of T is $\{a^n b^n w : n \geq 0,\ w \in L\}$.

We will show that the multiplication monoid of any finite groupoid that recognizes T must be as 'complicated' as the syntactic monoid of L. As a

consequence, we get a hierarchy of linear languages and linear tree languages corresponding to the hierarchy of regular languages. For example, no groupoid with aperiodic multiplication monoid can linearly recognize the yield of T if L is the set of words that contain an even number of 1's.

Let T be recognized by a groupoid $G \supseteq \{a, b, 0, 1\}$ with accepting set F and suppose that the multiplication monoid of G has order k. Let v be a tree in T such that the yield of v is $a^n b^n w$, where $n > k$, $|w| > 0$, and $w \in L$.

Let $E = \{R(x), L(x) \mid x \in \{0, 1, a, b\}\}$ and define $\mathcal{T} = \{\alpha \in E \mid 1\alpha \in F\}$ to be the set of words over E representing a tree in T. Observe that $\mathcal{T}$ is recognized by $\mathcal{M}(G)$ with the accepting set $\{U \in \mathcal{M}(G) \mid 1U \in F\}$, where F is the accepting set of G.

Let v be represented by a word $V \in \mathcal{T}$, and for any $x \in A$, let $D(x)$ denote any element in $\{R(x), L(x)\}$. Define X as the longest suffix of V that contains no symbol of the form $D(x)$ for $x \in \{0, 1\}$ (i.e., no $D(w_i)$, where w_i is a symbol of w). Then, there exists $1 \leq e \leq n$ such that

$$V = XD(w_e)Y \tag{3.1}$$

Let $S = \{R(a), L(a), R(b), L(b)\}$ and $B = \{R(0), R(1)\}$.

Lemma 3.4.1 *$Y \in (B^* L(b))^i (B^* L(a))^j B^*$, where $i \leq k$ and $j \leq k$.*

Proof. Having an occurrence of $R(a)$ or $R(b)$ after an occurrence of $D(w_i)$ or having an occurrence of $L(b)$ after an occurrence of $L(a)$ would contradict the fact that the yield of v belongs to $a^* b^* \{0, 1\}^*$. This shows that Y belongs to $(B^* L(b))^i (B^* L(a))^j B^*$, for some integers i and j.

Let $Y = Y_1 Y_2 Y_3$, where $Y_1 \in (B^* L(b))^i$, $Y_2 \in (B^* L(a))^j$, and $Y_3 \in B^*$.

Suppose that $j > k$. Then, by the Pigeon-hole Principle, there must be a decomposition $Y_2 = Z_1 L(a) Z_2 L(a) Z_3$ such that $Z_1 L(a) = Z_1 L(a) Z_2 L(a)$. This means that $V = XD(w_e)Y_1 Y_2 Y_3 = XD(w_e)Y_1 Z_1 L(a) Z_3 Y_3$. Thus, there exists a tree in T whose yield is $a^n b^{n-c} w'$, where $c > 0$ and $w' \in \{0, 1\}^*$, contradicting the definition of T.

We prove that $j \leq k$ in the same way. □

Lemma 3.4.2 *In Equation 3.1, we have* $D(w_e) = R(w_1)$.

Proof. Since $n > k$ and since, by the above lemma, there are at most k occurences of $D(a)$ and at most k occurences of $D(b)$ at the right of $D(w_e)$, then there must be at least one occurrence of $D(a)$ and $D(b)$ before $D(w_e)$. Thus, $D(w_e) = R(w_e)$, otherwise the yield of v would not be of the form $a^*b^*\{0,1\}^*$. Moreover, $w_e = w_1$ since no $L(w_1)$ can appear after a $D(a)$ or a $D(b)$. □

For all $\alpha \in \mathcal{M}(G)$, let $Z_\alpha = \{u \in B^* \mid u = \alpha\}$ and $X_\alpha = \{u \in S^* \mid u = \alpha\}$. Using Lemma 3.4.1 and Lemma 3.4.2, we can write

$$\mathcal{T} = \bigcup_{i,j \leq k} \bigcup X_\gamma Z_{\alpha_1} L(b) \cdots Z_{\alpha_i} L(b) Z_{\alpha_{i+1}} L(a) \cdots Z_{\alpha_{i+j}} L(a) Z_{\alpha_{i+1+j}} , \qquad (3.2)$$

where the second union is taken over all $\gamma, \alpha_1, \ldots \alpha_{i+j+1} \in \mathcal{M}(G)$ such that $1\gamma\alpha_1 L(b) \cdots \alpha_i L(b) \alpha_{i+1} L(a) \cdots \alpha_{i+j} L(a) \alpha_{i+j+1} \in F$

Let $\phi : \{0,1\}^* \to \mathcal{M}(G)^*$ be the morphism that maps 0 into $R(0)$ and 1 into $R(1)$. Trivially, any monoid recognizing $L' = \phi(L)$, also recognizes L. We can express L' as follows.

$$L' = \bigcup_{i,j \leq k} \bigcup Z_{\alpha_1} \cdots Z_{\alpha_{i+1+j}} \qquad (3.3)$$

where the second union is taken over the same domain as in Equation 3.2.

Theorem 3.4.3 *Let* $L \subseteq \{0,1\}^*$ *be a regular language, let* T *be any linear tree language whose yield is* $L_0 = \{a^n b^n w \mid n \geq 0,\ w \in L\}$, *and let* G *be any finite groupoid linearly recognizing* T. *Then,* $\mathcal{M}(G)$ *is aperiodic (resp. solvable) only if the syntactic monoid of* L *is aperiodic (resp. solvable).*

Proof. The class of languages recognized by aperiodic (resp. solvable) monoid is closed under finite union and concatenation (see [56]). Hence, if $\mathcal{M}(G)$ is aperiodic (resp. solvable), then Equation 3.3 shows that L can be recognized by an aperiodic (resp. solvable) monoid. □

Corollary 3.4.4 *In the previous theorem, if the linear language L_0 is recognized by a finite groupoid G such that $\mathcal{M}(G)$ is aperiodic (resp. solvable), then the syntactic monoid of L is aperiodic (resp. solvable).*

Proof. If such a groupoid exists, then this groupoid recognizes a linear tree language whose yield is L_0. By the previous Theorem, the syntactic groupoid of L must be aperiodic (resp. solvable). □

Observe that Theorem 3.4.3 and Corollary 3.4.4 can be generalized to any groupoid whose multiplication monoid belongs to a variety that is closed under concatenation.

Chapter 4

Loops and Quasigroups

In this chapter, we will study the case of finite quasigroups which are those groupoids whose multiplication table forms a latin square, i.e. no row and no column of the multiplication table contains two identical elements.

The study of quasigroups has a long history (see [26]). The combinatorial properties of latin squares were investigated as early as the eighteenth century, and quasigroups were intensively studied between 1930 and 1950. In particular, a theory of loops has been developed which closely follows that of groups. Quasigroups have also been considered from the point of view of computational complexity. In [52] some subproblems of the graph isomorphism problem were investigated and graphs constructed from combinatorial structures were considered. In particular, the isomorphism problem for latin square graphs was proved to be in DTIME($n^{\log n}$). Miller also showed that the isomorphism problem for quasigroups is in DTIME($n^{\log n}$) and, recently, Wolf [82] proved that these problems are in DSPACE($\log^2(n)$).

4.1 Basic theory of loops

The purpose of this section is to present a short introduction to the algebraic theory of loops. This is motivated by the fact that the number of textbooks on this topic is really limited and that loop theory is little known. A large part of the material that appears here comes from papers written in the forties by A.A. Albert and R.H. Bruck. Their work contains an impressive amount

of information but is not easily accessible. Furthermore, there exists no single source exposing comprehensively the theorems on normality, loop decompositions, and loop extensions, along the lines of the analogous results in the theory of groups. This survey does not intend to be complete but at least it gives many theorems that we believe are fundamental for an understanding of the algebraic structure of loops.

A *quasigroup* Q is a groupoid satisfying right and left *cancellation laws* (i.e. for all $a, b \in Q$, $ax = b$ and $ya = b$ have one and only one solution). Thus the Cayley table of a finite quasigroup forms a latin square.

We note that if a finite quasigroup is associative then it is a group. This is proved in the following theorem.

Theorem 4.1.1 *If G is a finite associative quasigroup then G is a group.*

Proof. It suffices to show that G possesses an identity element. Let $a, l \in G$ be such that $la = a$. Since G is associative, for any $x \in G$ we have that $l(ax) = (la)x = ax$ showing that l is a left identity of G. Similarly, we show that G possesses a right identity $r \in G$. Finally, since $r = lr = l$ then, l is an identity for G. □

A *loop* is a quasigroup with an identity. We define subloops and homomorphisms between loops in the same manner as for groups. One observes that because any closed subset of a finite loop that contains the identity satisfies both cancellation laws (the associated sub-Cayley table is still a latin square), then it must be a subloop. In the sequel we will consider only finite loops.

4.1.1 Normality and homomorphisms

A subloop N of a loop L is called *normal* if it satisfies

$$xN = Nx \ , \ (Nx)y = N(xy) \ , \ y(xN) = (yx)N \tag{4.1}$$

for every $x, y \in L$. Equation 4.1 implies that $(xN)y = (Nx)y = N(xy) = (xy)N = x(yN) = x(Ny)$. Then we have

$$x(Ny) = (xN)y \tag{4.2}$$

Since N contains the identity, $x \in Nx$ for every x, and $y \in Nx$ implies $y = nx$ for some $n \in N$. Thus, $Ny = N(nx) = (Nn)x = Nx$, showing that any normal subloop partitions a loop into disjoint cosets. Furthermore, by the cancellation laws, each coset has cardinality $|N|$. Indeed, these cosets form a loop, under the operation $(Nx)(Ny) = N(xy)$, that is denoted by L/N. This is formalized in the following theorem.

Theorem 4.1.2 ([16]) *If N is a normal subloop of the loop L, then N defines a natural homomorphism $x \rightarrow Nx$ of L onto the quotient loop L/N.*

Proof. By Equations 4.1 and 4.2, we have $(Nx)(Ny) = ((Nx)N))y = (N(Nx))y = ((NN)x)y = (Nx)y = N(xy)$ and so $(Nx)(Ny) = N(xy)$ for every $x, y \in L$. It can be verified that L/N satisfies both cancellation laws, and the identity is N. □

Clearly, the cardinality of L/N is $|L|/|N|$. A loop L having no proper normal subloop except $\{1\}$ is called *simple*. Since the order of a normal subloop always divides the order of the loop, every loop of prime order is simple. The above theorem is also true in the reverse direction. That is, any loop homomorphism induces a normal subloop.

Theorem 4.1.3 ([16]) *The kernel K of a homomorphism $\varphi : L \rightarrow M$, where L and M are two loops, is a normal subloop of L.*

Proof. Since K is closed under multiplication, it is a subloop of L. If k is an element of K, then for any x in L there is a unique element a in L such that $xk = ax$. Hence $x\varphi = (a\varphi)(x\varphi)$ and $a\varphi = 1$ by the cancellation laws. Therefore a must be in K. Similarly we show that if $kx = xb$ for k in K then b is also in K. This shows that for any x in L, $xK = Kx$. Now if x, y, z are in L then by the cancellation laws there exist unique elements p, q, r, s of L such that $z = (px)y = q(xy) = x(yr) = (xy)s$. But if one of p, q, r, s is in K then $x\varphi = (x\varphi)(y\varphi)$ and thus, each of p, q, r, s is in K (still by the cancellation

laws). This proves that K satisfies Equation 4.1 and then is a normal subloop of L. □

The center Z of a loop L satisfies Equation 4.1, so Z is an abelian normal subloop of L. In fact any subloop of the center of a loop L is an abelian normal subloop of L. We will come back to the center of a loop in subsection 4.1.4 when we will talk about nilpotency.

In the rest of this subsection we will give some properties of normal subloops and loop homomorphisms.

Theorem 4.1.4 *Let N, K be subloops of a loop L with N normal. Let $h : L \to L/N$ be the natural homomorphism. Then the inverse image of $h(K)$ is KN.*

Proof. The inverse of $h(K)$ is the union of all cosets kN such that $k \in K$, that is KN. □

Corollary 4.1.5 *Let N and K be subloops of a loop L with N normal in L. Then N is a normal subloop of KN.* □

Corollary 4.1.6 *If N is a normal subloop of a loop L and K is a subloop of L such that $N \subseteq K \subseteq L$ then N is normal in K.* □

Theorem 4.1.7 *Let $\phi : L \to H$ be a loop homomorphism, and let $N \subseteq L$ be the kernel of ϕ. Then H and L/N are isomorphic.*

Proof. Observe first that all elements in a given coset of N have the same image in H. Moreover, if $a = bx$ and $\phi(a) = \phi(b)$, then $\phi(x) = 1$ and $a \in bN$. This shows that H and L/N have the same cardinality.

Let $\theta : H \to L/N$ be the bijection defined by $\theta(\phi(a)) = aN$. Then, since $ab \in abN$, we have $\theta(\phi(ab)) = (ab)N = (aN)(bN) = \theta(\phi(a))\theta(\phi(b))$, proving that θ is an isomorphism. □

Theorem 4.1.8 *Let L be a loop and H, N normal subloops of L with $H \subseteq N$. Then, $(L/H)/(N/H)$ is isomorphic to L/N.* □

Proof. Let $\alpha : L \to L/N$, $\theta : L \to L/H$ and $\phi : L/H \to (L/H)/(N/H)$ be the natural homomorphisms. Furthermore let $\psi = \phi\theta$. Since $\psi^{-1}(1) = \theta^{-1}\phi^{-1}(1) = \theta^{-1}(N/H) = N$ then the kernel of θ and the kernel of ψ are identical. But this implies that the image of α and ψ are isomorphic. □

Let L be a loop and N, K be subloops of L. Then the subloop generated by $N \cup K$ is called the *union* of K and N, and is denoted by $(K \cup N)$.

Theorem 4.1.9 *If N, K are subloops of a loop L with N normal, then $(K \cup N) = NK = KN$.*

Proof. We use here the proof of [16]. Let $B = (K \cup N)$. By corollary 4.1.6, N is a normal subloop of B. Let $\theta : B \to B/N$ be the natural homomorphism. Then, by Theorem 4.1.4 the inverse image of $\theta(K)$ is KN, a subloop of B containing both K and N. This implies that $B = KN$. □

In subsection 4.1.2 we will also prove that the union and the intersection of two normal subloops are normal subloops.

4.1.2 Multiplication group and inner mapping group

In this subsection we will see that with any finite loop we can associate two finite groups. These groups are very important in loop theory. They are used as a tool for proving many theorems, and also form a link between loop theory and group theory.

In [2], Albert consider the multiplication monoid of a quasigroup Q. In this case, the functions $R(a)$ and $L(a)$ are permutations on Q, and $\mathcal{M}(Q)$ is a group called the multiplication group of Q. On the other hand, if the multiplication monoid $\mathcal{M}$ of a groupoid G is a group, then G must be a quasigroup.

Theorem 4.1.10 *A finite groupoid G is a quasigroup if and only if $\mathcal{M}(G)$ is a group.*

Proof. We only need to show the *if* direction. Let $\mathcal{M}(G)$ be a group such that I is the identity permutation. Since G is finite, it suffices to show that for any $a, b \in G$ there exist $c, d \in G$ such that $ac = b$ and $da = b$. Let $c = bL^{-1}(a)$ and $d = bR^{-1}(a)$ where $L^{-1}(a)$ and $R^{-1}(a)$ are the respective inverses of $L(a)$ and $R(a)$. Then $ac = bL^{-1}(a)L(a) = bI = b$ and $da = bR^{-1}(a)R(a) = bI = b$, concluding the proof. □

In particular, if G contains an identity and $\mathcal{M}(G)$ is a group then G is a loop.

Theorem 4.1.11 ([2]) *Let L be a loop with center Z and $\mathcal{M}$ be its multiplication group with center $\mathcal{Z}$. Then Z is isomorphic to $\mathcal{Z}$ and $Z = 1\mathcal{Z}$.*

Proof. First we prove that $Z \subseteq 1\mathcal{Z}$. If $c \in Z$ then $R(c) = L(c)$. Furthermore, for all x and y in L we have $x(yc) = (xy)c$ and $c(xy) = (cx)y$. Equivalently, $R(c)L(x) = L(x)R(c)$ and $L(c)R(y) = R(y)L(c)$. Hence if $c \in Z$ then $R(c) = L(c) \in \mathcal{Z}$.

Next we prove that $1\mathcal{Z} \subseteq Z$. Suppose that $C \in \mathcal{Z}$ and let $c = 1C$. We will show that c is in the center of L. First we have $xc = cL(x) = 1CL(x) = 1L(x)C = xC$ showing that $C = R(c)$. Similarly $cx = cR(x) = 1CR(x) = 1R(x)C = xC$ showing that $C = L(c)$. Hence $R(c) = L(c)$ and so, we have $xc = cx$. Furthermore $R(c)L(x) = L(x)R(c)$ implies that $x(yc) = (xy)c$ while $L(c)R(y) = R(y)L(c)$ implies $c(xy) = (cx)y$. We thus have $xc = cx$, $c(xy) = (cx)y$ and $(xc)y = x(cy)$ from what it is easy to show that $(xc)y = x(cy)$ proving that $c \in Z$.

We have shown that $Z = 1\mathcal{Z}$. It remains to prove that Z and $\mathcal{Z}$ are isomorphic. First, by the above discussion, we know that if $C \in \mathcal{Z}$ and $c = 1C$ then $C = R(c) = L(c)$. This proves that Z and $\mathcal{Z}$ have the same cardinality. Now, let $\theta : Z \to \mathcal{Z}$ be defined by $\theta(c) = R(c)$. Clearly θ is one-to-one since, given $C \in \mathcal{Z}$, $c = 1C$ is uniquely defined. Thus θ is a one-to-one function between two sets of same cardinality, that is a bijection. Furthermore θ is an isomorphism since for any $C, D \in \mathcal{Z}$ we have $(1C)(1D) = 1CR(1D) = 1CD$. □

In [15], Bruck used a subgroup of the multiplication group called the *inner mapping group* as a very useful tool to decide, among other things, if a subloop is normal. Recall that a subloop N of a loop L is normal if it satisfies

$$xN = Nx \ , \ (Nx)y = N(xy) \ , \ y(xN) = (yx)N$$

This condition can be rewritten as

$$N = NR(x)L^{-1}(x) = NR(x)R(y)R^{-1}(xy) = NL(x)L(y)L^{-1}(yx)$$

for all $x, y \in L$. Let $\mathcal{J} = \mathcal{J}(L)$ be the subgroup of $\mathcal{M}(L)$ generated by all

$$T(x) = R(x)L^{-1}(x),$$

$$R(x,y) = R(x)R(y)R^{-1}(xy),$$

and

$$L(x,y) = L(x)L(y)L^{-1}(yx).$$

We call $\mathcal{J}$ the *inner mapping group* of L. The above discussion makes the next theorem clear.

Theorem 4.1.12 ([15]) *A subloop N of a loop L is normal iff $N\mathcal{J} = N$.* □

The next lemma will be useful to give another characterization of the inner mapping group and to prove the normality of certain subloops of a loop.

Lemma 4.1.13 ([15]) *Let L be a loop, $\mathcal{M}$ its multiplication group and $\mathcal{J}$ its inner mapping group. Then for any element $X \in \mathcal{M}$ there exists $U \in \mathcal{J}$ such that $X = UR(1X)$.*

Proof. Let $K = \{\alpha \in \mathcal{M} \mid \alpha \in \mathcal{J}R(1\alpha)\}$. We will proceed by proving that $\alpha P \in K$ for any $\alpha \in K$ and $P \in \mathcal{G} = \{R(x), L(x) \mid x \in L\}$. Since ,$\mathcal{G}$ is a generator of $\mathcal{M}$, this will imply that $K\mathcal{M} = K$ and that $K = \mathcal{M}$.

Observe first that $\mathcal{J}R(x) = \mathcal{J}T(x)L(x)$=$\mathcal{J}$L(x). Hence $K = \{\alpha \in \mathcal{M} \mid \alpha \in \mathcal{J}L(1\alpha)\}$. Let $\alpha \in \mathcal{J}R(t)$ where $1\alpha = t$. We have

$$\alpha R(x) \in \mathcal{J}R(t)R(x) = \mathcal{J}R(t,x)R(tx) = \mathcal{J}R(t,x)R(1\alpha R(x)) \subseteq K,$$

$$\alpha L(x) \in \mathcal{J}R(t)L(x) = \mathcal{J}T(t)L(t,x)L(xt) = \mathcal{J}T(t)L(t,x)L(1\alpha L(x)) \subseteq K.$$

□

Using the above lemma we can give another characterization of the inner mapping group.

Theorem 4.1.14 *The inner mapping group of a loop L is the subgroup of $\mathcal{M}(L)$ generated by all $\alpha \in \mathcal{M}$ such that $1\alpha = 1$ where 1 is the identity of L.*

Proof. Clearly $1U = 1$ for all $U \in \mathcal{J}$. Hence it suffices only to show that if $1X = 1$ for some $X \in \mathcal{M}$ then $X \in \mathcal{J}$. Let $\alpha \in \mathcal{M}$ be such that $1\alpha = 1$. Then by the preceding lemma $\alpha \in \mathcal{J}R(1\alpha) = \mathcal{J}R(1) = \mathcal{J}$, proving the theorem. □

Lemma 4.1.13 can also be used to prove the normality of certain subloops of a loop. A nonempty subset S of a loop L is called *self-conjugate* if $S\mathcal{J} = S$.

Theorem 4.1.15 *[16] Let H be a subloop of a loop L, and define $K = \{k \in H : k\mathcal{J} \subseteq H\}$. Then, K is the largest normal subloop of L contained in H.*

Proof. Since $(K\mathcal{J})\mathcal{J} \subseteq H$, K is a self-conjugate subset of L. Let $k \in K$ and $U \in \mathcal{J}$. Then, by Lemma 4.1.13, $R(k)U \in \mathcal{J}R(kU)$ so $(Kk)U = KR(k)U \subseteq K\mathcal{J}R(kU) \subseteq H(kU) \subseteq HH = H$ This proves that $KK = K$ and that K is a subloop of L. Furthermore, since $K\mathcal{J} = K$ then, K is also normal. To show that K is maximal, observe that if N is a normal subloop of L contained in H then, by Theorem 4.1.12, $N\mathcal{J} \subseteq N \subseteq H$ and thus, $N \subseteq K$. □

Corollary 4.1.16 *[16] Every self-conjugate subset of a loop L generates a normal subloop of L.*

Proof. Let S be a self-conjugate subset of L generating a subloop $H \subseteq L$. Defining K as in Theorem 4.1.15, we have $S\mathcal{J} \subseteq S \subseteq H$. This shows that $S \subseteq K$ and that $K = H$. Hence, by Theorem 4.1.15, H is a normal subloop of L. □

Theorem 4.1.17 *Let S be any subset of a loop L. Then, $\langle S\mathcal{J}\rangle$ is the smallest normal subloop of L that contains S.*

Proof. First, observe that $S\mathcal{J}$ is self-conjugate. Hence, by the above corollary, $\langle S\mathcal{J}\rangle$ is a normal subloop of L. Now, suppose that N is a normal subloop of L that contains S. Then, $S\mathcal{J} \subseteq N$ and so, $\langle S\mathcal{J}\rangle \subseteq N$. □

Given a loop L and a subset S of L, we call $\langle S\mathcal{J}\rangle$ the normal subloop of L generated by S. The next two theorems establish a relationship between the normal subloops of a loop L and the normal subgroups of $\mathcal{M}(L)$.

Theorem 4.1.18 *Let L be a loop and $\mathcal{M}$ its multiplication group. If $\mathcal{N}$ is a normal subgroup of $\mathcal{M}$ then $1\mathcal{N}$ is a normal subloop of L.*

Proof. Letting $N = 1\mathcal{N}$ we have $N\mathcal{J} = 1\mathcal{N}\mathcal{J} = 1\mathcal{J}\mathcal{N} = 1\mathcal{N} = N$. Hence, N is normal by theorem 4.1.12. □

Let N be any normal subloop of a loop L and define the following sets: $\mathcal{M}_N = \{U \in \mathcal{M} \mid \forall x \in L, xU \in xN\}$ and $\mathcal{J}_N = \mathcal{J} \cap \mathcal{M}_N$. So $\mathcal{M}_N$ is the set of mappings of $\mathcal{M}$ fixing the cosets generated by N. Observe that if $x \in L$ and $U \in \mathcal{M}_N$ then, by definition of $\mathcal{M}_N$ we have that $xU \in xN$, and so, $x\mathcal{M}_N \subseteq xN$. Furthermore, if $n \in N$, then, $R(n)$ is an element of $\mathcal{M}_N$, and $xn = xR(n) \in x\mathcal{M}_N$ implies that $xN \subseteq x\mathcal{M}_N$. Hence, we have that $xN = x\mathcal{M}_N$ for any $x \in L$ and in particular $N = 1\mathcal{M}_N$.

Theorem 4.1.19 ([3, 15]) *Let N be a normal subloop of a loop L then*

(i) $\mathcal{M}_N$ *is a normal subloop of* $\mathcal{M}(L)$ *and* $\mathcal{M}(L/N)$ *is isomorphic to* $\mathcal{M}(L)/\mathcal{M}_N$.

(ii) $\mathcal{J}_N$ *is a normal subloop of* $\mathcal{J}(L)$ *and* $\mathcal{J}(L/N)$ *is isomorphic to* $\mathcal{J}(L)/\mathcal{J}_N$.

Proof. (i) Let $\theta : L \to L/N$ be the natural homomorphism. Then it could be verified that $\phi : \mathcal{M}(L) \to \mathcal{M}(L/N)$, the function induced by $R(x)\phi = R(x\theta)$ and $L(x)\phi = L(x\theta)$, is a homomorphism such that $(x\alpha)\theta = (x\theta)(\alpha\phi)$ for all $x \in L$ and $\alpha \in \mathcal{M}(L)$. The kernel K of ϕ is the set of all $\alpha \in \mathcal{M}(L)$ such that $\alpha\theta = \theta$. Then $\kappa \in K$ iff $(x\kappa)\theta = x\theta$ for every $x \in L$ or equivalently $x\kappa \in xN$. But this implies that $K = \mathcal{M}_N$. Hence $\mathcal{M}_N$ is normal in $\mathcal{M}$ and $\mathcal{M}(L/N)$ is isomorphic to $\mathcal{M}/\mathcal{M}_N$.

(ii) Simply observe that $\phi^{-1}(1) \cap \mathcal{J} = \mathcal{J}_N$. □

We close this subsection with two theorems on the properties of normal subloops where the proofs make use of the multiplication group and the inner mapping group.

Theorem 4.1.20 *Let* K, N *be normal subloops of a loop* L. *Then* $K \cap N$ *is a normal subloop of* L. □

Proof. Let $a \in N \cap K$ and let $\mathcal{J}$ be the inner mapping group of L. Then $a\mathcal{J} \in N$ and $a\mathcal{J} \in K$ by Theorem 4.1.12. Hence $(N \cap K)\mathcal{J} \subseteq N \cap K$, proving the normality of $N \cap K$ again by Theorem 4.1.12. □

Theorem 4.1.21 *Let* K, N *be normal subloops of a loop* L. *Then* $(K \cup N)$ *is a normal subloop of* L. □

Proof. We know from group theory that the union of two normal subgroups of a group is still a normal subgroup. Hence $\mathcal{M}_K\mathcal{M}_N$ is a normal subgroup of $\mathcal{M}$. By Theorem 4.1.16 it is sufficient to show that $KN = 1\mathcal{M}_K\mathcal{M}_N$. In the observation preceding Theorem 4.1.19, we have seen that $N = 1\mathcal{M}_N$ and $K = 1\mathcal{M}_K$. Thus, $1\mathcal{M}_K\mathcal{M}_N = K\mathcal{M}_N \subseteq KN$ where the last inequality holds

by definition of $\mathcal{M}_N$. Moreover, we have $KN = \langle K \cup N \rangle$ by Theorem 4.1.9, and $R(1) \in \mathcal{M}_K \cap \mathcal{M}_N$. Thus, $K = 1\mathcal{M}_K R(1) \subseteq 1\mathcal{M}_K\mathcal{M}_N$ and $N = 1R(1)\mathcal{M}_N \subseteq 1\mathcal{M}_K\mathcal{M}_N$, concluding the proof. □

4.1.3 Commutators and associators

For any elements x, y and z of a loop L we define the *commutator* $[x, y]$ and the *associator* $[x, y, z]$ as the unique solution to the equations $xy = (yx)[x, y]$ and $(xy)z = (x(yz))[x, y, z]$. If N is a normal subloop of L then we denote by $S_{(N,L)}$ the subloop generated by all commutators and associators of the form $[n, x]$, $[x, n]$, $[n, x, y]$, $[x, n, y]$ and $[x, y, n]$ where $n \in N$ and $x, y \in L$.

Theorem 4.1.22 *For any normal subloop N of a loop L, $S_{(N,L)}$ is a subloop of N.*

Proof. Let $n \in N$ and $x, y \in L$. Then $(nx)y = (n(xy))[n, x, y]$. Since N is normal there exists $n_1, n_2 \in N$ such that $(nx)y = (xy)n_1$ and $(n(xy))[n, x, y] = (xy)(n_2[n, x, y])$. Hence $n_1 = n_2[n, x, y]$ and $[n, x, y] \in N$. A similar argument shows that $[x, n, y]$, $[x, y, n]$, $[n, x]$, $[x, n] \in N$. □

We define (N, L) to be the normal subloop generated by $S_{(N,L)}$. More specifically, $(N, L) = \langle S_{(N,L)}\mathcal{J} \rangle$ where $\mathcal{J}$ is the inner mapping group of L. When L is a group, it is not difficult to show that $S_{(N,L)} = (N, L)$. However, no proof is known (at least from the author) when L is a loop, with the notable exception where $N = L$. Let $S_{(L,L)}$ be denoted by L'.

Theorem 4.1.23 ([16]) *If L is a loop then L' is a normal subloop.*

Proof. We begin the proof by showing the following three properties of L':

1. $xy \in L'$ iff $yx \in L'$
2. $(xy)z \in L'$ iff $x(yz) \in L'$
3. $(xy)z \in L'$ iff $(ab)c \in L'$, for any permutation (a, b, c) of (x, y, z).

The two first properties follow from the fact that L' contains all commutators and associators. Let x, y, z be elements of L. Thus $(xy)z \in L' \leftrightarrow x(yz) \in L' \leftrightarrow (yz)x \in L'$, so

$$(xy)z \in L' \leftrightarrow (yz)x \in L' \tag{4.3}$$

Now let $k = [x, y]$. Then $(xy)z \in L' \leftrightarrow ((yx)k)z \in L' \leftrightarrow (z(yx))k \in L' \leftrightarrow z(yx) \in L' \leftrightarrow (yx)z \in L'$, so

$$(xy)z \in L' \leftrightarrow (yx)z \in L' \tag{4.4}$$

The third property is just a consequence of equations 4.3 and 4.4.

Now, in order to prove that L' is normal, we must show first that $xL' = L'x$. If x is a given element of L the equation $xa = bx$ induces a bijection $a \to b$ from L to itself. Choose any x' such that $x'x \in L'$. Then using the above properties we get $x'(xa) \in L' \leftrightarrow (x'x)a \in L' \leftrightarrow a \in L'$ and, similarly, $x'(bx) \in L' \leftrightarrow b \in L'$. Since $xa = bx$, we see that a and b are both or neither in L'. Thus $xL' = L'x$ for every $x \in L$.

It remains to show that $L'(xy) = (L'x)y$ and that $(xy)L' = x(yL')$. We will only prove the former equation, the proof of the last one being similar. Fix x, y in L and write $p = a(xy) = (bx)y$. If we choose any w such that $w(xy) \in L'$ then, using the above properties we have $pw \in L' \leftrightarrow (a(xy))w \in L' \leftrightarrow (w(xy))a \in L' \leftrightarrow a \in L'$. Similarly we deduce that $pw \in L' \leftrightarrow b \in L'$. Thus $a(xy) = (bx)y$ if and only if a, b are both in L', i.e. $L'(xy) = (L'x)y$. □

The next theorem plays an important role in the central nilpotency theory of loops.

Theorem 4.1.24 *Let N be a normal subloop of a loop L. Then $N/(N, L)$ is in the center of $L/(N, L)$.*

Proof. Let $h : L \to L/(N, L)$ be the natural homomorphism of L onto $L/(N, L)$. We want to prove that $h(N)$ is in the center of $L/(N, L)$. Let $n \in N$, $x, y \in L$, and let $m = h(n)$, $a = h(x)$ and $b = h(y)$. We have $h((nx)y) = (ma)b = (m(ab))h([n, x, y]) = m(ab)$. Similarly we find that

$(am))b = a(mb)$, $(ab)m = a(bm)$, and $ma = am$. Hence $h(n)$ is in the center of $L/(N, L)$. □

Theorem 4.1.25 *For any loop L, $L/(L, L)$ is an abelian group. Furthermore, if K is a normal subloop of L such that L/K is abelian then $(L, L) \subseteq K$.*

Proof. The first statement is a direct consequence of Theorem 4.1.24. Suppose that K is a normal subloop of L and that L/K is abelian. Let $\phi : L \to L/K$ be the natural homomorphism. Then, for any $x, y, z \in L$ we have $\phi([x, y]) = \phi([x, y, z]) = 1$, where 1 is the identity of L/K. Henceforth, $(L, L) \in \phi^{-1}(1) = K$. □

The subloop (L, L) is called the *commutator-associator subloop* of L. It will play a major role when we will discuss solvable loops.

In [15] Bruck defines (N, L) in a different manner. Let $\mathcal{J}$ be the inner mapping group of L and let $N(\mathcal{J})$ be the subloop of L generated by all elements of the form $nUL^{-1}(n)$ with $n \in N$ and $U \in \mathcal{J}$. Let $n \in N$, $U \in \mathcal{J}$ and $m = nUL^{-1}(n)$. Then $nU = mL(n) = nm$ and so, $m \in N$. Therefore, $N(\mathcal{J})$ is a subloop of N. In the rest of this subsection, we will prove that $N(\mathcal{J})$ is a normal subgroup of L and that $N(\mathcal{J}) = (N, L)$.

Lemma 4.1.26 ([15]) *Let H be a subloop of a loop L. If K is any subloop of L such that $H(\mathcal{J}) \subseteq K \subseteq H$ then K is a normal subloop of L*

Proof. Let x be any element of $K \subseteq H$, U any element of $\mathcal{J}$. Then $xUL^{-1}(n) = y \in H(\mathcal{J}) \subseteq K$ by hypothesis. Hence $xU = yL(x) = xy$ is in K. Hence $\mathcal{J}$ maps K into itself. But this means that K is normal. □

Theorem 4.1.27 *If N is subloop of L then $N(\mathcal{J})$ is a normal subloop of L.* □

Theorem 4.1.28 *Let N be a normal subloop of L. Then $(N, L) \subseteq N(\mathcal{J})$.*

Proof. We must show that $[n,x],[x,n],[n,x,y],[x,n,y]$ and $[x,y,n]$ belong to $N(\mathcal{J})$ for any $n \in N$ and $x,y \in L$. We will only do the proof for $[x,n,y]$, the other cases being similar. First, observe that if $n \in N$ and $x,y \in L$ then $(nx)y = (na)(xy)$ where $a = nR(x,y)L^{-1}(n) \in N(\mathcal{J})$, $x(yn) = (xy)(nb)$ where $b = nL(y,x)L^{-1}(n) \in N(\mathcal{J})$ and $nx = x(nc)$ where $c = nT(x)L^{-1}(n) \in N(\mathcal{J})$. Hence, since $N(\mathcal{J})$ is normal, there exists $a_1,\ldots,a_5 \in N(\mathcal{J})$ such that $x(ny) = x(y(na_1)) = (x(yn))a_2 = ((xy)n)a_3 = (x(yn))a_4 = (x(ny))a_5$. But $x(ny) = ((xn)y)[x,n,y]$ implies that $[x,n,y] = a_5 \in N(\mathcal{J})$.

□

Lemma 4.1.29 ([15] p.272) *Let N be a normal subloop of a loop L and Let $\mathcal{G}$ be any set of generators of $\mathcal{J}$. Then $N(\mathcal{J})$ is generated by all elements $nUL^{-1}(n)$ with $n \in N$ and $U \in \mathcal{G}$.* □

Theorem 4.1.30 *If N is a normal subloop of a loop L then $(N,L) = N(\mathcal{J})$.*

Proof. In view of the previous lemma and Theorem 4.1.28 it is sufficient to show that $nT_xL^{-1}(n)$, $nR(x,y)L^{-1}(n)$ and $nL(x,y)L^{-1}(n)$ belong to (N,L). The proof is similar to that of Theorem 4.1.28. □

4.1.4 Solvable and nilpotent loops

Among the most important normal subloops of a loop L, we have seen that there are the subloops (N,L) defined for any normal subloop N. As a particular case there is also the commutator-associator subloop of L. Finally, we have also introduced the center, an abelian normal subloop of L. These subloops play the same role in the theory of loops as their analogues in group theory. In particular they are used to define nilpotent and solvable loops.

Let L be a finite loop. We call a *normal series* a sequence

$$L = L_0 \supseteq L_1 \supseteq \cdots \supseteq L_k = \{1\} \tag{4.5}$$

where for all i such that $0 \leq i \leq k$, L_i is a normal subloop of L. Recall that by corollary 4.1.6, L_{i-1} is also normal in L_i. Furthermore when L_i/L_{i-1} is

simple for all i then (4.5) is called a *composition series.* It has been shown (see [3, 16]) that any two normal series of a loop have isomorphic refinements (Schreier Refinement Theorem). Hence, as a consequence, we have that all composition series of a loop are isomorphic (Jordan-Hölder Theorem).

A finite loop L is said to be *solvable* if the sequence

$$L = L^{(0)} \supseteq L^{(1)} \supseteq \cdots \supseteq L^{(i)} \supseteq \cdots$$

where for each i we have $L^{(i+1)} = (L^{(i)}, L^{(i)})$, terminates in the identity.

Theorem 4.1.31 *Every subloop and quotient loop of a solvable loop is solvable.*

Proof. Observe first that if N is a subloop of a loop L then (N, N) is a subloop of (L, L). Thus, if L is solvable then $N^{(i)} \subseteq L^{(i)}$ for all i, proving that N is solvable. Suppose now that $K = L/N$ where L is a solvable loop and N is a normal subloop of L. Let $\varphi : L \to K$ be the natural morphism. Then any commutator (resp. associator) of K is the homomorphic image of some commutator (resp. associator) in L. This means that $(K, K) \subseteq \varphi((L, L))$. Hence for any i we have $K^{(i)} \subseteq \varphi(L^{(i)})$, proving that K is solvable. □

Theorem 4.1.32 *A loop L is solvable if and only if it has a normal series*

$$L = L_0 \supseteq L_1 \supseteq \cdots \supseteq L_c = \{1\}$$

in which L_{i-1}/L_i is abelian for all i.

Proof. By Theorem 4.1.25 $L/(L, L)$ is an abelian group. Thus, any solvable loop possesses a normal series of the above form. For the other direction, observe that for any i, since L_i/L_{i+1} is abelian then, by Theorem 4.1.25, $(L_i, L_i) \subseteq L_{i+1}$. Hence for any i if $L^{(i)} \subseteq L_i$ then $L^{(i+1)} = (L^{(i)}, L^{(i)}) \subseteq (L_i, L_i) \subseteq L_{i+1}$. But $L = L^{(0)} = L_0$ implies that $L^{(i)} \subseteq L_i$ for all i. In particular $L^{(c)} \subseteq 1$, proving that L is solvable. □

Corollary 4.1.33 *A loop L is solvable if for some normal subloop N of L both N and L/N are solvable.*

Proof. If $L/N \supseteq L_1/N \supseteq \cdots \supseteq L_{r-1}/N \supseteq N/N$, and $N \supseteq N_1 \supseteq \cdots \supseteq N_{s-1} \supseteq 1$ are series satisfying the property of theorem 4.1.32 then $L \supseteq L_1 \supseteq \cdots \supseteq N \supseteq N_1 \supseteq \cdots \supseteq 1$ is a series satisfying the same property. □

We define a loop to be *nilpotent* if it has a normal series (4.5) such that L_i/L_{i-1} is in the center of L/L_{i-1} for all $1 < i \leq k$. Such a series is called a *central series*. One can observe that since L_i/L_{i-1} is abelian, any nilpotent loop is also solvable. For any nilpotent loop we define two canonical central series: The *lower central series* is the normal series

$$L = H_1 \supseteq H_2 \supseteq \cdots \supseteq H_m = \{1\} \tag{4.6}$$

where for all $i < m$, $H_{i+1} = (H_i, L)$. The *upper central series* is the central series

$$\{1\} = Z_0 \subseteq Z_1 \subseteq \cdots \subseteq Z_l = L \tag{4.7}$$

where for all $i > 0$, Z_i/Z_{i-1} is the center of L/Z_{i-1}.

Theorem 4.1.34 *Let L be a loop having lower and upper central series as in (4.6) and (4.7). If (4.5) is any central series of L then, $H_i \subseteq L_{m+1-i}$ and $L_i \subseteq Z_i$ for all i.*

Proof. The proof is identical to the case of groups (see [30] p.151). □

Hence the upper and the lower central series of a nilpotent loop have the same length. We call this length the *class* of the loop. Note that a loop of class 1 is an abelian group.

Next theorem relates the nature of a loop with that of its multiplication semigroup. The proofs can be found in [15] pp.280-282.

Theorem 4.1.35 *Let L be a loop, $\mathcal{M}$ its multiplication semigroup, and $\mathcal{J}$ its inner mapping group.*

1. *If $\mathcal{M}$ is nilpotent of class c then L is nilpotent of class not greater than c.*

2. *If L is nilpotent and has order g then $\mathcal{M}$ and $\mathcal{J}$ are solvable and their order divides some power of g.*

The above theorem cannot be extended by stating that L is nonsolvable whenever $\mathcal{M}$ is nonsolvable. In Subsection 4.9, we will construct a loop of order 10 that is solvable and whose multiplication group is nonsolvable. It remains to determine if L nonsolvable implies that $\mathcal{M}(G)$ is also nonsolvable.

Theorem 4.1.35 can however be improved in the case where the order of L is a power of some prime number. This kind of loop is called a *p-loop*. Bruck first mentions these loops in [15]. His first motivation was to show that any nilpotent loop can be decomposed into a direct product a finite p-loops. This result is well known to be true for groups, but he proved it to be false for general loops by giving explicit counter example of order 6. He observes also that since there exist loops of prime order having a non trivial subloop then, one cannot expect to develop a theory of p-loops comparable to what exists in the associative case. However, from the above results, we obtain the following corollary. ([15] p. 282).

Corollary 4.1.36 *If L is a nilpotent p-loop then $\mathcal{M}(L)$ and $\mathcal{J}(L)$ are nilpotent p-groups.* □

4.1.5 Isotopy

We have already introduced the notion of isotopy in Chapter 2. Most of the results stated here are simple restatements, in terms of loop, of previous ones.

Theorem 4.1.37 ([3] Thm. 2) *A loop L is isotopic to a group G if and only if L and G are isomorphic.*

Theorem 4.1.38 ([3]) *If L and H are isotopic loops then H is isomorphic to a principal isotope of L.*

Theorem 4.1.39 ([3]) *If (δ, η, ι) is a principal isotopy of a loop $(L, \cdot)$ onto a loop $(L, *)$ then there exist elements $f, g \in L$ such that $(\delta, \eta, \iota) = (R(f), L(g), \iota)$.*

Isotopy preserves some important properties of loops. We just state some of them without proof.

Theorem 4.1.40 ([3]) *1. Every loop isotopic to a loop $(L,\cdot)$ is isomorphic to a loop $(L,*)$ having precisely the same normal subloops as $(L,\cdot)$.*

2. Any loop isotopic to a simple loop is simple.

3. Isotopic loops have isomorphic multiplication groups.

4. Isotopic loops have isomorphic inner mapping groups.

5. Isotopic loops have isomorphic center.

As Albert mentions, the first part of the above theorem only says that if $(N,\cdot)$ is a normal subloop of $(L,\cdot)$ then $(N,*)$ is a normal subloop of $(L,*)$. But, it does not say that $(N,\cdot)$ and $(N,*)$ are isotopic at all.

4.1.6 Conjugated loops

With any loop $(L,\cdot)$ we associate five loops: $(L,/)$, $(L,\backslash)$, $(L,\circ)$, $(L,\oblong)$ and $(L,\diagdown)$. These six loops are said to be *conjugated*. The products of the conjugates of $(L,\cdot)$ are defined as followed. For all $a,b,c \in L$,

1. $a/b = c$ iff $c \cdot b = a$ (Right division)
2. $a\backslash b = c$ iff $a \cdot c = b$ (Left division)
3. $a \circ b = c$ iff $b \cdot a = c$ (Dual product)
4. $a \oblong b = c$ iff $b \cdot c = a$ (Dual right division)
5. $a \diagdown b = c$ iff $c \cdot a = b$ (Dual left division)

Some authors define a loop as a set L together with three binary operations $\cdot$, $/$ and $\backslash$ such that for all $x,y \in L$, the following conditions are satisfied.

(i) $x\backslash(x \cdot y) = x \cdot (x\backslash y) = y$

(ii) $(x \cdot y)/y = (x/y) \cdot y = x$

(iii) $x/x = y\backslash y$

Without condition (iii) we only get a quasigroup and it is not difficult to see that the two first conditions are equivalent to the definition of the conjugates $\backslash$ and $/$.

It is a simple exercise to prove that conjugation is an equivalence relation. Naturally it is possible for two of the above conjugates to be the same (e.g. if one of the conjugates is commutative). But this situation is subordinated to the following theorem.

Theorem 4.1.41 ([26]) *The number of distinct conjugates of a loop $(L, \cdot)$ is 1,2,3 or 6.*

Proof. Let R_1, R_2 and R_3 be the functions mapping $(L, \cdot)$ in $(L, /), (L\backslash)$ and (L, o) respectively. Then, one can verify that $\{R_1, R_2, R_3\}$ generates a group G isomorphic to the symmetric group S_3. Hence the number of distinct conjugates must be equal to some subgroup of S_3. □

Recall that the *exponent* of a group G is the smallest integer q such that $a^q = 1$ for all $a \in G$. The following lemma will be usefull later.

Theorem 4.1.42 *Let L be a loop and $\mathcal{M}$ its multiplication group. Then for any $a, b \in L$ we have $a/b = aR^{q-1}(b)$ and $a\backslash b = bL^{q-1}(a)$ where q is the exponent of $\mathcal{M}$.*

Proof. The proof is immediate using the fact that $a/b = aR^{-1}(b)$ and, since q is the exponent of $\mathcal{M}$, $R^{-1}(b) = R^{q-1}(b)$ (the case $a\backslash b$ is similar). □

4.1.7 Loop extensions

In this last subsection, we will see how, given two loops K and N, we can construct a loop L such that N is a normal subloop of L and K is isomorphic to L/N.

Let L be a loop, N a normal subloop of L, and L/N the quotient loop. We would like to define a product such that its application on N and L/N produces a loop isomorphic to L. Define the set $K = \{k_1, k_2, \ldots, k_m\}$ such that all $k_i \in L$, $k_1 = 1$, $m = |L/N|$, and $[k_1], [k_2], \ldots, [k_m]$ are the cosets generated by N. Then, any element $x \in L$ can be uniquely written as the product of an element k_i with an element $n_i \in N$. So, there is a bijection between the elements of L and those of $K \times N$.

If $k_i, k_j \in K$ and $n_i, n_j \in N$ then there is a unique $k_l \in K$ such that $[k_l] = [k_i][k_j]$. Furthermore $(k_i n_i)(k_j n_j) = k_l x$, where $x = ((k_i n_i)(k_j n_j))L^{-1}(k_l)$. Hence for any $k_i, k_j \in K$ we define the function $\phi_{k_i,k_j} : N \times N \to N$ by

$$\phi_{k_i,k_j}(n_i, n_j) = (n_i L(k_i))(n_j L(k_j)))L^{-1}(k_l)$$

where $k_l \in K$ is such that $[k_l] = [k_i][k_j]$. Then, the loop $(L/N \times N, *)$, defined by $([k_i], n_i) * ([k_j], n_j) = ([k_i][k_j], \phi_{k_i,k_j}(n_i, n_j))$, is isomorphic to L.

This idea can be used to construct an extension of two arbitrary loops. Let K and N be two loops. For each pair $k_1, k_2 \in K$ we associate a function $\phi_{k_1,k_2} : N \times N \to N$ such that the three following conditions are satisfied: First $\phi_{1,1} = N$, second $\phi_{k,1}$ is a loop with identity 1 for all $k \in K$, and finally 1 is a left identity of $\phi_{1,k}$ for all $k \in K$. With this setting we define the *extension* of K and N (called crossed extension by Albert [3]) as the the loop $(K \times N, *)$ define by

$$(k_1, n_1) * (k_2, n_2) = (k_1 k_2, \phi_{k_1,k_2}(n_1, n_2))$$

Theorem 4.1.43 ([3]) *A loop L is isomorphic to an extension of two loops K and N if and only if N is isomorphic to a normal subloop of L and K is isomorphic to L/N.* □

It seems that there is no loop analogue of the wreath product (see [36]). However, the direct product of two loops has been studied. We refer the interested reader to [16]. For more results on loops extension see [22].

4.2 Notation and definitions

In this section, we present the basic notation and definitions that will be used in the following. At the end of the section, we prove a lemma that will be useful thereafter.

If r is the root of a tree T, and if r_1 and r_2 are respectively the left and right child of r, then the subtrees of T rooted at r_1 and at r_2 are respectively called the *left* and the *right subtrees* of T.

We need to generalize the notion of special trees introduced in Chapter 2. Let Q be an alphabet and S a set of variables. A special tree T over $Q \cup S$ is a tree where each element of S appears exactly once as a label of a leaf.

As an example of application of the special trees, let T be a tree over an alphabet Q and let T_1 be any subtree of T. If we replace the subtree T_1 in T by a leaf labeled with X, then we obtain a special tree T_2 over $Q \cup \{X\}$. So, we have decomposed T into two trees, T_1 and T_2, and we have $T = T_2 \cdot^X T_1$.

We denote by $(T_1\ T_2)$ the tree T with left subtree T_1 and right subtree T_2. As an example, for variables X Y, $(X\ Y)$ represents the special tree whose yield is XY, and T can be represented by $((X\ Y) \cdot^X T_1) \cdot^Y T_2$.

In this thesis we will mostly use the single variable X. Thus, to simplify the notation we will write $\cdot$ instead of $\cdot^X$ when the context clearly indicates that only X is used. Observe that $\cdot^X$, for a fixed variable X, is an associative operation. Hence, for any special trees $T_1, \ldots, T_k$ over $Q \cup \{X\}$, the expression $T_1 \cdot T_2 \cdot \cdots \cdot T_k$ defines the same special tree no matter which parenthesization is used. This will be denoted by $\prod_{i=1}^{k} T_i$.

For any tree T, we define the *value* of T, denoted $v(T)$, to be the element of Q resulting from the evaluation of the yield of T using the induced parenthesization. If T is a special tree over $Q \cup \{X\}$ and T_1, T_2 are two trees over Q, then

$$v(T_1) = v(T_2) \Rightarrow v(T \cdot T_1) = v(T \cdot v(T_1)) = v(T \cdot T_2)\ . \qquad (4.8)$$

On the other hand, a recursive application of the cancellation laws shows that

$$v(T \cdot T_1) = v(T \cdot T_2) \Rightarrow v(T_1) = v(T_2) \ . \tag{4.9}$$

Two special trees are said to be *yield-equivalent* if they have the same yield.

In the rest of this section, we describe a way of modifying a tree without changing its yield or its value. This will be used many times in the following.

Let $w \in Q^+$, and let $T = (\prod_{i=1}^{k} T_i) \cdot R$ be a tree with yield w, where $k \geq |Q|$. For all i such that $1 \leq i \leq k$, pick any special tree $\overline{T_i}$ yield-equivalent to T_i. For any x, y such that $1 \leq x \leq y \leq k$, let $S(x,y) = \prod_{i=1}^{x-1} T_i \cdot \prod_{i=x}^{y} \overline{T_i} \cdot \prod_{i=y+1}^{k} T_i \cdot R$, ignoring the first term (the next to last term) whenever $x = 1$ ($y = k$). Observe that $S(x,y)$ is always yield-equivalent to T.

The following result is due to Hervé Caussinus.

Lemma 4.2.1 ([19]) *There exist two integers a, b such that $1 \leq a \leq b \leq k$ and such that $v(T) = v(S(a,b))$.*

Proof. Define $S(1,0) = T$. Since $|Q| \leq k$, there exist, by the pigeon-hole principle, two integers a, b such that $1 \leq a \leq b \leq k$ and such that

$$\begin{aligned} v(S(1,a-1)) &= v\left(\prod_{i=1}^{a-1} \overline{T_i} \cdot \prod_{i=a}^{b} T_i \cdot \prod_{i=b+1}^{k} T_i \cdot R\right) \\ &= v\left(\prod_{i=1}^{a-1} \overline{T_i} \cdot \prod_{i=a}^{b} \overline{T_i} \cdot \prod_{i=b+1}^{k} T_i \cdot R\right) \\ &= v(S(1,b)) \end{aligned}$$

From (4.9), we have that

$$v\left(\prod_{i=a}^{b} T_i \cdot \prod_{i=b+1}^{k} T_i \cdot R\right) = v\left(\prod_{i=a}^{b} \overline{T_i} \cdot \prod_{i=b+1}^{k} T_i \cdot R\right)$$

Finally, from (4.8), we have that

$$\begin{aligned} v(T) &= v\left(\prod_{i=1}^{a-1} T_i \cdot \prod_{i=a}^{b} T_i \cdot \prod_{i=b+1}^{k} T_i \cdot R\right) \\ &= v\left(\prod_{i=1}^{a-1} T_i \cdot \prod_{i=a}^{b} \overline{T_i} \cdot \prod_{i=b+1}^{k} T_i \cdot R\right) \\ &= v(S(a,b)) \end{aligned}$$

□

4.3 Languages recognized by quasigroups

In this section, we will define a restricted notion of recognition by finite groupoids, and we will show that any language recognized in this way is regular. Then, we will show that any language recognized by a finite quasigroup is also recognized in the restricted way, proving that it is regular.

Given a tree T and a path π in T, we define the *right-length* of π as its number of right edges. The *right-depth* of T is the maximum right-length of its paths. Then, for any integer k, the set of trees of right-depth $\leq k$ over a groupoid G corresponds precisely to the set RD_k of Section 2.5. Recall that any language recognized by a finite groupoid of constant right-depth is regular. We will use this result to prove the following theorem.

Theorem 4.3.1 *Any language recognized by a finite quasigroup is regular.*

Proof. We will show that any language recognized by a finite quasigroup Q is also recognized in constant right-depth by Q and hence is regular

Let Q be a quasigroup of order q, n a positive integer, and $w \in Q^n$. Let T be a tree with yield w and d the right-depth of T. Furthermore, for any set A, define the function $N_k : A^{(*)} \to \mathbb{N}$ such that $N_k(S)$ is the number of paths of right-length greater or equal to k in S.

We will show that if $d > 2q$, then there exists a tree T', yield equivalent to T, such that $v(T') = v(T)$ and $N_d(T') < N_d(T)$. The conclusion will follow from an iterated application of this fact.

Suppose that π is a path of right-length d in T. Without loss of generality, we can suppose that the first edge of π is a right edge. Otherwise, we just have to consider the maximal subtree of T having this property. Let $(n_0, m_0), (n_1, m_1), \ldots, (n_{2q}, m_{2q})$ be the first $2q+1$ right edges in π, and let Q_i be the subtree of T rooted at n_i. For all $0 \leq i \leq q-1$, define T_i to be the special tree over $Q \cup \{X\}$ obtained from Q_{2i} when Q_{2i+2} is replaced by

X, and let $R = Q_{2q}$. Clearly, we have $Q_{2i} = T_i \cdot Q_{2i+2}$ for every $i < q$, and $T = Q_0 = T_0 \cdot Q_2 = T_0 \cdot T_1 \cdot Q_4 = \cdots = \prod_{i=0}^{q-1} T_i \cdot Q_{2q}$. Hence, we have

$$T = \prod_{i=0}^{q-1} T_i \cdot R \ .$$

Our construction is such that the path to X in T_i contains precisely two right edges: (n_{2i}, m_{2i}) and (n_{2i+1}, m_{2i+1}). Thus, X is a leaf contained in the right subtree R_i of T_i, but it cannot be the leftmost leaf in R_i (see Fig. 1).

Let L_i be the left subtree of T_i, let f be the leftmost leaf in R_i, and let P_i be the special tree over $Q \cup \{X, Y\}$ obtained from R_i by substituting the variable Y for the leaf f. Thus, we have

$$T_i = (L_i \ R_i) = (L_i \ (P_i \cdot^Y f)) \ .$$

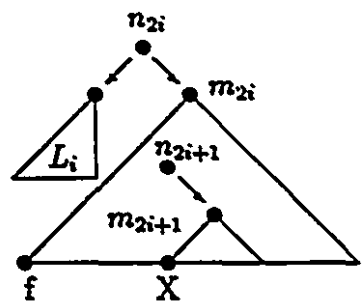

Figure 1 : T_i

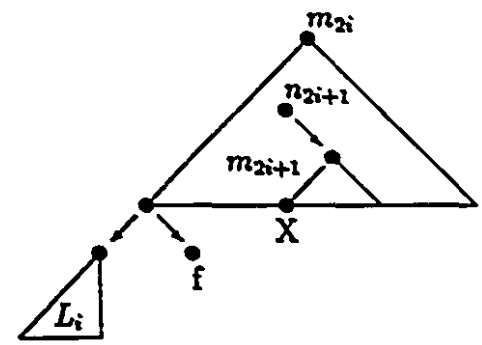

Figure 2 : $\overline{T_i}$

Now, let $\overline{T_i}$ (see Fig. 2) be the special tree defined by

$$\overline{T_i} = (P_i \cdot^Y (L_i \ f)) \ .$$

It should be clear that $\overline{T_i}$ is yield-equivalent to T_i. Thus, by Lemma 4.2.1, there are two integers $0 \leq a \leq b \leq q-1$ such that $v(T') = v(T)$ where T' is defined by

$$T' = \prod_{i=0}^{a-1} T_i \cdot \prod_{i=a}^{b} \overline{T_i} \cdot \prod_{i=b+1}^{q-1} T_i \cdot R \ .$$

Moreover, if d_i is the right-depth of T_i then $N_{d_i}(\overline{T}_i) \leq N_{d_i}(T_i)$. To see this, observe that the right-length of any path from the root to a leaf in L_i is unchanged. Also, the right-length of any path from the root to a leaf in R_i is decreased by one except for the path to f whose right-length remains the same.

Any path p, from the root of $T = \prod_{i=1}^{q-1} T_i \cdot R$ to a leaf w_i, has a corresponding path in T' that goes to the same leaf w_i. Only the segment of p passing through T_i will be modified if we replace T_i by $\overline{T}_i$, and, as we just observed, the right-length of this segment will not increase. On the contrary, the corresponding path of π in T' will have a strictly smaller right-length. Hence $N_d(T') < N_d(T)$, and this concludes the proof. □

4.4 Linear recognition

In this section, we will prove that any language linearly recognized by a finite quasigroup is regular. This result cannot be inferred from Theorem 4.3.1 because the transformation used in the proof does not preserve the "linearity" of the trees. Nevertheless, we will proceed in a similar manner: we will first define a restricted notion of linear recognition, and we will prove that any language recognized in this way by a finite groupoid is regular. Then, we will show the equivalence between linear recognition and its restricted version, in the context of finite quasigroups.

Let T be a linear tree and let m be a vertex in T. We say that an *alternation* occurs at m whenever m is a right (left) child and has a left (right) child that is not a leaf. The *number of alternations* in T is the number of vertices where an alternation occurs.

Let A be a finite alphabet. A language $L \subseteq A^*$ is said to be *recognized in constant alternations* by a groupoid G if there exist an alphabetic morphism $h : A^* \to G^*$, a subset $F \subseteq G$, and a constant k such that $x \in L$ if and only if there is a linear tree T with yield $h(x)$ and with at most k alternations such

that $v(T) \in F$.

Lemma 4.4.1 *If a language $L \subseteq A^*$ is recognized in constant alternations by a finite groupoid G, then L is regular.*

Proof. Suppose that L is recognized in $t \geq 0$ alternations by G with the accepting subset $F \subseteq G$. Without lost of generality we can suppose that $A \subseteq G$. The proof is by induction on t.

If $t = 0$ then $L = L_1 \cup L_2$ where

$$L_1 = \{x \in A^* \mid x \text{ left-to-right evaluates to some } g \in F\}$$

$$L_2 = \{x \in A^* \mid x \text{ right-to-left evaluates to some } g \in F\}$$

Since L_1 and L_2 are regular, L is also regular.

Suppose now that $t > 0$ and assume that the theorem is true for any $0 \leq s < t$. For any $a \in G$ define the language L_a as the set of $x \in A^*$ for which there exists a linear tree with yield x such that T evaluates to a in $t-1$ alternations. Moreover, we define the languages N_a and M_a as follows.

$$N_a = \{x \in A^* \mid ax \text{ left-to-right evaluates to some } g \in F\}$$

$$M_a = \{x \in A^* \mid xa \text{ right-to-left evaluates to some } g \in F\}$$

Clearly, N_a and M_a are regular. Furthermore, L_a is recognized by G in $t-1$ alternations. Hence, by the inductive hypothesis, L_a is regular. Moreover, L can be expressed as

$$L = \bigcup_{a \in G} L_a N_a \cup \bigcup_{a \in G} M_a L_a$$

This shows that L is regular and concludes the proof. □

Theorem 4.4.2 *Any language linearly recognized by a finite quasigroup is regular.*

Proof.

The proof is similar to that of Theorem 4.3.1. Let Q be a quasigroup of order q, and let L be linearly recognized by Q. By Lemma 4.4.1, it suffices to show that L is recognized in constant alternations by Q.

Let n be a positive integer, let $x \in Q^n$, and let T be a linear tree with yield x. For any set A define $\mathrm{Alt} : A^{\mathrm{LIN}} \to \mathbb{N}$ such that $\mathrm{Alt}(S)$ is the number of alternations in S.

It is sufficient to show that if $\mathrm{Alt}(T) > 3q$, then there exists a linear tree T' yield equivalent to T such that $\mathrm{Alt}(T') < \mathrm{Alt}(T)$ and $v(T') = v(T)$.

Suppose that $\mathrm{Alt}(T) > 3q$. Let n_0 be the root of T, $n_1, \ldots, n_{3q}$ be the first $3q$ vertices where an alternation occurs, and define Q_i to be the subtree of T rooted at n_i, for $i \leq 3q$. For $0 \leq i \leq q-1$, define T_i to be the linear special tree obtained from Q_{3i} by substituting the variable X for $Q_{3(i+1)}$, and let $R = Q_{3q}$. It should be clear that

$$T = \prod_{i=0}^{q-1} T_i \cdot R.$$

The above construction is such that each T_i has exactly 2 alternations (see Fig. 3). So, we can write

$$T_i = P_1 \cdot P_2 \cdot P_3,$$

where the special trees P_1, P_2, P_3 have no alternation.

Let $\overline{T}_i$ be the special tree (see Fig. 4) defined by

$$\overline{T}_i = P_1 \cdot P_3 \cdot P_2.$$

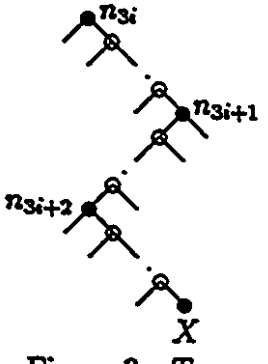

Figure 3 : T_i

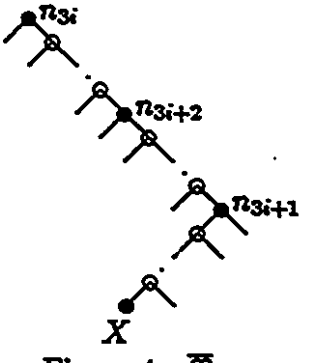

Figure 4 : $\overline{T}_i$

By lemma 4.2.1, there exist two positive integers, $0 \leq a \leq b \leq q-1$, such that $v(T') = v(T)$ where T' is defined by

$$T' = \prod_{i=0}^{a-1} T_i \cdot \prod_{i=a}^{b} \overline{T}_i \cdot \prod_{i=b+1}^{q-1} T_i \cdot R.$$

Furthermore, for $0 \leq i \leq q-1$, we have $\mathrm{Alt}(\overline{T}_i) < \mathrm{Alt}(T_i)$ (compare Fig. 3 and Fig. 4). This implies that $\mathrm{Alt}(T') < \mathrm{Alt}(T)$, proving the theorem. □

4.5 Parenthesization of logarithmic depth

As another application of Lemma 4.2.1, we can show that it is only necessary to consider parenthesizations of logarithmic depth in order to evaluate a word over a finite quasigroup. (In general this is not true since no word over a weakly linear groupoid can be evaluated to a nonzero element unless a linear evaluation tree be used.) This is formalized in the following theorem. The following proof is a simplification of a result from Hervé Caussinus.

Theorem 4.5.1 ([19]) *Let Q be a quasigroup of order q. For any $n > 0$, any $w \in Q^n$ and any T with yield w, there exists a yield equivalent tree S of depth smaller than $3q + \log_2 n$ such that $v(T) = v(S)$.*

Proof. Let n_0 be the root of T and suppose that T has a path of length $d \geq 3q + \log_2 n$. It is possible to find q nodes $n_1, \ldots, n_q$ along that path such that, for each $0 \leq i < q$, the portion from n_i to n_{i+1} has length exactly 3.

Let R be the subtree of T rooted at n_q. For $0 \leq i < q$, define the special tree T_i as the subtree of T rooted at n_i where the subtree rooted at n_{i+1} is replaced by the variable X. Hence, we have that $T = \prod_{i=0}^{q-1} T_i \cdot R$.

Suppose that the yield of T_i is of the form uXv. Pick two arbitrary trees U and V of minimal depth (i.e. smaller than $\log_2 n$) such that $\mathrm{yield}(U) = u$ and $\mathrm{yield}(V) = v$. Then, $\overline{T}_i = ((U\ X)\ V)$ is a special tree yield-equivalent to T_i. Observe that this transformation decreases the length of the path from n_i to X by one, while the depth of $\overline{T}_i$ is bounded above by $\log_2 n + 2$.

By Lemma 4.2.1, there exist two positive integers, $0 \leq a \leq b \leq q-1$, such that $v(T') = v(T)$ where T' is defined by

$$T' = \prod_{i=0}^{a-1} T_i \cdot \prod_{i=a}^{b} T_i \cdot \prod_{i=b+1}^{q-1} T_i \cdot R.$$

One can verify that the number of paths of length d is strictly less in T_i than in T_i. The conclusion follows from an iterated application of this argument. □

4.6 Regular languages recognized by quasigroups

We have shown in Section 4.3 that finite quasigroups only recognize regular languages. In this section we refine this result by showing that only open regular languages can be recognized by finite quasigroups.

A regular language over an alphabet A is said to be *open* (see [58]) if it is a finite union of languages of the form $L_0 a_1 L_1 \cdots a_k L_k$, where $k \geq 1$, $a_i \in A$, and $L_i \subseteq A^*$ is a language recognized by a finite group.

Lemma 4.6.1 *Any language $L \subseteq A^*$ of the form $L_0 \cdots L_k$, where L_i is recognized by a finite group, is open.*

Proof. For any $a \in A$, let $L_i a^{-1} = \{w \mid wa \in L_i\}$. Now, if L_i is recognized by a group G with the accepting set $F \subseteq G$, then $L_i a^{-1}$ is recognized by G using the same morphism and the accepting set $F' = \{g \in F \mid ga \in F\}$. Hence, L is a finite union of languages of the form $L_{i_1} a_{i_1}^{-1}\ a_{i_1} \cdots L_{i_m} a_{i_m}^{-1}\ a_{i_m}\ L_k$, where $m \geq 0$, $1 \leq i_1 < \cdots < i_m < k$ and $a_{i_j} \in A$. □

The proof of Lemma 4.4.1 shows that any language linearly recognized in constant alternation by a finite quasigroup Q is a finite union of languages of the form $L_0 \cdots L_k$, where L_i is left-to-right or right-to-left recognized by Q. So, each L_i is recognized by $\mathcal{M}(Q)$, which is a finite group, and by the previous lemma we have the following result.

Theorem 4.6.2 *Any language linearly recognized by a finite quasigroup is open.* □

This observation can be extended to general recognition.

Theorem 4.6.3 *Finite quasigroups recognize only open regular languages.*

Proof. We will use again the technique of Section 4.3.

Let Q be a finite quasigroup. We define a *comb* over Q recursively as follows. Any $a \in Q$ is a comb. If $a \in Q$ and $u \in Q^{(*)}$ is a comb then $w = (au)$ is also a comb. No other element of $Q^{(*)}$ is a comb.

Any tree $t \in Q^{(*)}$ can be decomposed into

$$s \cdot^{x_1} t_1 \cdots \cdot^{x_n} t_n \,, \tag{4.10}$$

where $n \geq 1$, $x_1, \ldots, x_n$ are variables, s is a special tree over $Q \cup \{x_1, \ldots, x_n\}$ such that each leaf is labeled with a variable, and t_i is a comb over Q. Let $\mathrm{comb}(t)$ be the smallest n for which such a decomposition exists.

We will show that, for any tree $t \in Q^{(*)}$, there exists a yield-equivalent tree $s \in Q^{(*)}$ such that $\mathrm{comb}(s)$ is bounded by a constant. By Lemma 4.6.1, this will prove the theorem because the set of combs in $Q^{(*)}$ that evaluates to a given element forms a language recognized by the multiplication group of Q. More precisely, we will show that for any tree $t \in Q^{(*)}$ such that $\mathrm{comb}(t) > 8^q$, where q is the order of Q, we can find a yield equivalent tree $t' \in Q^{(*)}$ such that $v(t) = v(t')$ and $\mathrm{comb}(t') < \mathrm{comb}(t)$.

Suppose that $t \in Q^{(*)}$ is such that $\mathrm{comb}(t) = n > 8^q$, and let t be decomposed as in Equation 4.10.

Since, s has more than 8^q leaves, it must possesses a path of length $k > 3q$. Let the nodes of this paths be $d_0, d_1, \ldots, d_{k-1}$, where d_0 is the root of s and d_{i+1} is the child of d_i.

For $0 \leq i \leq q$, let s_i be the tree rooted at d_{3i}. Moreover, for $0 \leq i < q$, let v_i be the special tree constructed from s_i by substituting the variable X for s_{i+1}. Hence we have

$$s = \prod_{i=0}^{q-1} v_i \cdot s_q$$

Let $x_{i_q}, \ldots, x_{i_q+j_q}$ be the leaves of s_q. Moreover, let $x_{i_1}, \cdots, x_{i_1+j_i}$ be the leaves at the left of X in v_i, and let $x_{i_2}, \cdots, x_{i_2+l_i}$ be the leaves at the right of X in v_i (one of these sequences can be empty but at least one of them has length ≥ 2). We can thus write

$$t = \prod_{i=0}^{q-1} z_i \cdot z_q$$

where

$$z_i = v_i \cdot^{x_{i_1}} t_{i_1} \cdots \cdot^{x_{i_1+j_i}} t_{i_1+j_i} \cdot^{x_{i_2}} t_{i_2} \cdots \cdot^{x_{i_2+l_i}} t_{i_2+l_i}, \quad i < q$$

and

$$z_q = s_q \cdot^{x_{i_q}} t_{i_q} \cdots \cdot^{x_{i_q+j_q}} t_{i_q+j_q}$$

Let $w_i = \text{yield}(t_i)$ and define u_{i_1} to be the comb whose yield is $w_{i_1} \cdots w_{i_1+j_1}$, and u_{i_2} the comb whose yield is $w_{i_2} \cdots w_{i_2+j_2}$. Then $\bar{z}_i = ((y_1 X) y_2) \cdot^{y_1} u_{i_1} \cdot^{y_2} u_{i_2}$ is yield equivalent to z_i.

By Lemma 4.2.1, there exist two integers a and b such that $v(t) = v(t')$, where

$$t' = \prod_{i=0}^{a-1} z_i \cdot \prod_{i=a}^{b} \bar{z}_i \cdot \prod_{i=b+1}^{q-1} z_i \cdot z_q$$

We observe that $\text{comb}(z_i) \geq 3$ while $\text{comb}(\bar{z}_i) \leq 2$. This implies that $\text{comb}(t') < \text{comb}(t)$, proving the theorem.

□

The above theorem has the following corollaries.

Corollary 4.6.4 *Let L be a language recognized by a finite quasigroup. Then, $\bar{L}$ is recognized by a quasigroup if and only if L is recognized by a finite group.*

Proof. The *if* part of the proof follows from the fact that the class of languages recognized by a finite group is closed under complementation. Moreover, it is shown in [57] that if both L and $\bar{L}$ are open, then they are recognized by a finite group. The conclusion follows from this result and Theorem 4.6.3.

□

Corollary 4.6.5 *The class of languages recognized by finite quasigroups is not closed under complementation.*

Proof. In the next section we will show that any cofinite language is recognized by a finite quasigroup. Such a language cannot be recognized by a group. Hence, by Theorem 4.6.4, no finite language can be recognized by a finite quasigroup. □

4.7 Weakly cancellative groupoids

A groupoid G is called *weakly cancellative* if for any $a, x, y \in G^0$, the two following properties are satisfied.

$$(ax = ay \neq 0) \Rightarrow (x = y)$$

$$(xa = ya \neq 0) \Rightarrow (x = y)$$

The Cayley table of a weakly cancellative groupoid is such that in each row and each column no nonzero element appears twice. Hence, the nonzero elements of such a groupoid forms a partially defined groupoid called *incomplete quasigroup*. This terminology is justified by the following lemma.

Lemma 4.7.1 ([29]) *An incomplete loop (quasigroup) containing n elements can be embedded in a loop (quasigroup) containing t elements, for any $t \geq 2n$.* □

We will also need the following result.

Lemma 4.7.2 *Let Q be a quasigroup and let $u, v, w \in Q^+$. Then, the cardinality of $Q(uwv)$ is at least as large as that of $Q(w)$.*

Proof. This is a direct consequence of the cancellation law. □

Weakly cancellative groupoids will be useful to prove that a language can be recognized by a quasigroup. This is a consequence of the following lemma.

Lemma 4.7.3 *Any language recognized by a weakly cancellative groupoid, with 0 in the accepting set, is also recognized by a quasigroup.*

Proof. Let G be a weakly cancellative groupoid, and let $L \subseteq G^*$ be a language recognized by G. Assume that 0 belongs to the accepting set. Let $B = G - \{0\}$, let $B^{(*)}$ be the free groupoid over the set B, and let β be the order of B. We also denote by B the incomplete loop induced by the elements of B in G.

We will define a sequence of incomplete loops B_i, for $i \geq 0$. Let $B_0 = B$ and define B_{i+1} from B_i as follows. All defined products in B_i are defined identically in B_{i+1}. Moreover, for any undefined product $a \cdot b$ in B_i, we define $a \cdot b = (ab)$ in B_{i+1}

Remark. *Observe that $(ab) \in B^{(*)}$ is a new element. Observe also that if c is an element of B_{i+1} that does not belong to B_i, then for any $x, y \in B$, $(x(yc))$ and $((xy)c)$ are two distinct elements of B_{i+3}. Similarly, $((cx)y)$ and $(c(xy))$ are two distinct elements of B_{i+3}. This and Lemma 4.7.2 imply that for any $u, v \in B^*$ such that $k = |u| + |v|$, $B_{k+1}((u(ab)v))$ contains at least k elements.*

Let $k = \beta + 2$ and let B_k be embedded in a finite loop H. We will argue that L is recognized by H with the accepting set containing all nonzero elements of the accepting set of G plus all elements not in B.

If $w \in B^*$ can be evaluated to a nonzero element in G, then w can be evaluated to the same element in H using the same parenthesization. This shows that if $w \in B^*$ is not accepted by B then it is not accepted by H. This also proves that if w is accepted by G but does not evaluate to 0, then it is accepted by H.

Suppose that w can be evaluated to 0 in G. Then, there exists a segment u of w of minimal length that can be evaluated to 0, i.e. $w = sut$, $0 \in G(u)$ and for any strict segment v of u, $0 \notin G(v)$. So, there exist $u_1, u_2 \in B^+$ and $a, b \in B$ such that $u = u_1 u_2$, $a \in G(u_1)$, $b \in G(u_2)$ and $ab = 0$ in G, but $a \neq 0$ and $b \neq 0$. This implies that w can be partially evaluated to $sabt$ both in G

and in H. Now, there are two possibilities. First, if s and t are small enough, then $s(ab)t$ can only be evaluated, in H, to an element in $B^{(*)} - B$: in this case H accepts w. Otherwise, by the above remark, $H(w)$ contains at least $\beta + 1$ different elements, and so, at least one of them is not in B. Thus, H accepts w if and only if G accepts w. □

Theorem 4.7.4 *Any cofinite language is recognized by a finite loop.*

Proof. Let A be a finite alphabet and let $L \subseteq A^*$ be cofinite. Let k be the smallest integer such that all $w \in A^*$ of length larger or equal to k are in L. Let $B = \cup_{i=0}^{k} A^i$ and $G = B \cup \{0\}$.

We define a product on G as follows. The absorbing element is 0, and for any $a, b \in B$, $a \cdot b = ab$ if $ab \in B$, otherwise $a \cdot b = 0$. Clearly, G is a weakly cancellative groupoid.

The partially defined loop G recognizes L by taking the accepting set to be 0 plus all elements in $B \cap L$. Finally, by Lemma 4.7.3, we can construct a finite loop, from G, that recognizes L. □

Observe that loops can recognize languages that are not cofinite and are not recognized by a group. A simple example is the set $\mathrm{OR} \subseteq \{0,1\}^*$, composed of all words that contain at least one 1. This language is recognized by U_1, the monoid defined by $0 \cdot 0 = 0$ and $0 \cdot 1 = 1 \cdot 0 = 1 \cdot 1 = 1$. Here, 0 is an identity and 1 is absorbing. Since U_1 is a weakly cancellative groupoid, the language OR can be recognized by a finite loop.

On the other hand, some very simple languages cannot be recognized by a loop (or even a quasigroup). This is the case for any finite set. To see this, let L be a finite language recognized by a loop B. Without loss of generality, we can suppose that $L \subseteq B^*$. Since B satisfies the cancellation laws, for any $w \in B^*$ not in L there exist $v \in B^+$ such that wv will be accepted by B. But, this contradicts the fact that L is finite.

Theorem 4.7.5 *No finite language can be recognized by a finite loop.* □

In Theorem 4.7.3, it is necessary that 0 belongs to the accepting set. Indeed, as the proof of Theorem 4.7.4 shows, any finite language can be recognized by a weakly associative groupoid. Moreover, nonregular languages can be recognized by such groupoids. For example, the 2-sided Dyck language D with two sets of parenthesis $\{a, \bar{a}\}$ and $\{b, \bar{b}\}$ can be recognized by the groupoid defined over the set $\{1, a, \bar{a}, b, \bar{b}, 0\}$, where 1 is the identity, 0 is absorbing, $a \cdot \bar{a} = \bar{a} \cdot a = b \cdot \bar{b} = \bar{b} \cdot b = 1$, and all other products yield 0. It suffices to take 1 as unique accepting element. Observe that D belongs to DSPACE($\log n$) but it is not known if it belongs to NC^1(see [49]).

4.8 Representing functions with expressions

Let G be a finite groupoid and $X_n = \{x_1, x_2, \ldots, x_n\}$ a finite set of variables. An element $w = w(x_1, \ldots, x_n) \in (G \cup X_n)^{(*)}$ is called an *expression* over G with n variables. If $w \in (G \cup X_n)^{(*)}$ and $v_1, \ldots v_n \in (G \cup X_m)^{(*)}$, then $w(v_1, \ldots, v_n) \in (G \cup X_m)^{(*)}$ is the expression obtained from w by replacing each variable x_i by the expression v_i. We associate expressions over $G^{(*)}$ with elements of G and expressions over $(G \cup X_n)^{(*)}$ with functions $G^n \to G$ in the obvious way. A function $f : A^n \to A$ is said to be *representable* over G if there exist an embedding $\theta : A \to G$ and an expression $w(x_1, \ldots, x_n)$ over G such that, for every $a_1, \ldots, a_n \in A$, $\theta(f(a_1, \ldots, a_n)) = w(\theta(a_1), \ldots, \theta(a_n))$.

In [51] Maurer and Rhodes proved that if G is a simple nonabelian group then any function $G^n \to G$ can be represented by an expression over G. In this section, we extend this theorem to the case of loops. As a corollary, we define a class of loops for which the problem of evaluating an expression is complete for NC^1. Just before, we give some basic results.

Lemma 4.8.1 *Let $(Q, \cdot)$ be a quasigroup. If a function can be represented over $(Q, /)$ or $(Q, \backslash)$, then it can be represented over $(Q, \cdot)$.*

Proof. This is a direct corollary of Lemma 4.1.42. □

As a special case of Lemma 2.6.4 we have the following result.

Lemma 4.8.2 *If L and H are isotopic loops then any function that can be represented over H can also be represented over L.*

The above lemma cannot be generalized to quasigroups. To see this, consider Z_4, the cyclic group of order four. It is well know that the AND function of two bits cannot be represented over any abelian group (e.g. see [72]). But we can find a quasigroup, isotopic to Z_4, on which the AND can be represented. Let $S = \{0,1,2,3\}$, let $\alpha : S \to S$ be the permutation $(0,1,3,2)$, and consider the quasigroup $Q = (S, \cdot)$ obtained from Z_4 using the isotopy $x \cdot y = \alpha(x + y)$. It is a simple exercise to verify that $AND(x,y) = (x \cdot y) \cdot (x \cdot y)$, for any $x, y \in \{0,1\}$.

Lemma 4.8.3 *Let L be a loop. The functions $A : L^3 \to L$ defined by $A(x,y,z) = [x,y,z]$ and $C : L^2 \to L$ defined by $C(x,y) = [x,y]$ can be represented over L.*

Proof. Observe that $[x,y,z] = ((xy)z)\backslash(x(yz))$ and $[x,y] = (xy)\backslash(yx)$. The conclusion follows from Lemma 4.8.1. □

We can now prove our generalization of the Maurer-Rhodes theorem. The proof is a straightforward adaptation for loops of a version for groups due to Howard Straubing [72].

Theorem 4.8.4 *If L is a finite simple loop that is not an abelian group then every function $f : L^n \to L$ can be represented over L.*

Proof. Let $g_1 \in L\backslash\{1\}$, where 1 is the identity of L, and let $g_2 \in L$. Since L is simple, then by Theorem 4.1.17 $\langle g_1 \mathcal{J} \rangle = L$ where $\mathcal{J}$ is the inner mapping group of L. In particular, there exists $u_{g_1,g_2}(x) \in (x\mathcal{J})^{(*)}$ such that $u_{g_1,g_2}(g_1) = g_2$ and $u_{g_1,g_2}(1) = 1$. Observe that any $U \in \mathcal{J}$ is the product of elements in $\{R(a), L(a) : a \in L\}$. Thus, xU is representable on L, and so is $u_{g_1,g_2}(x)$.

Because it is simple and not an abelian group, L is equal to its commutator-associator subloop. Thus, each element $h \in L$ can be written (assuming an

implicit parenthesization) as $h = \prod_{i=1}^{r} \delta_i$, where δ_i is a commutator $[g_i, h_i]$ or an associator $[f_i, g_i, h_i]$ of L. By Lemma 4.8.3, the function

$$\Delta_i(x,y) = \begin{cases} [x,y] & \text{if } \delta_i \text{ is a commutator} \\ [f_i,x,y] & \text{if } \delta_i \text{ is an associator} \end{cases}$$

can be represented over L, for all $1 \leq i \leq r$. Then

$$w_{2,h} = \prod_{i=1}^{r} \Delta_i(u_{h,g_i}(x), u_{h,h_i}(y))$$

can also be represented over L with $w_{2,h}(h,h) = h$ and $w_{2,h}(g,1) = w_{2,h}(1,g) = 1$, for all $g \in L$ ($w_{2,h}$ can be seen as representing the OR function of two bits).

For all $m \geq 2$, define the representable function

$$w_{m+1,h} = w_{2,h}(w_{m,h}, x_{m+1})$$

such that $w_{m,h}(h,\ldots,h) = h$ and $w_{m,h}(g_1,\ldots,g_m) = 1$ if $g_i = 1$ for some i.

Let $h,k \in L$, with $h \neq 1$, let k^λ be the unique element of L such that $k^\lambda k = 1$ and let $L\backslash\{k^\lambda\} = \{k_1,\ldots,k_t\}$. We can represent

$$z_{k,h} = w_{t,h}(u_{k_1k,h}(k_1x),\ldots,u_{k_tk,h}(k_tx))$$

such that $z_{k,h}(k) = h$ and $z_{k,h}(g) = 1$, for $g \neq k$.

Finally, let $\nu = (c_1,\ldots,c_n) \in L^n$ and let

$$v_{\nu,h} = w_{n,h}(z_{c_1,h}(x_1),\ldots,z_{c_n,h}(x_n)) \ .$$

Then, $v_{\nu,h}(\nu) = h$ and $v_{\nu,h}(\mu) = 1$, for $\mu \neq \nu$. Hence, we can represent any function $f : L^n \to L$ using the expression

$$\prod_{f(\nu)\neq 1} v_{\nu,f(\nu)}$$

(Any parenthesization can be used since at most one term in the product is different from the identity.) □

We observe in the above proof that the presence of commutativity is not as dramatic for loops as it is for groups. This is because in the Maurer-Rhodes theorem the crucial point is that the *OR* function of two bits can be

represented over any simple nonabelian group, and to achieve this we needed commutators. But in theorem 4.8.4 we can use associators when there is no commutator other than 1.

We define a *family* of expressions over a loop L as a set $S = \{W_1, W_2, \ldots\}$ where, for all $n \geq 0$, W_n is an expression over L with n variables. We say that a function $f : A^* \to A$ can be represented by S if, for all n, the restriction of f to A^n can be represented by W_n. The *length* of S is a function mapping n to the number of non-parenthesis symbols in W_n.

Corollary 4.8.5 *Let L be any simple loop that is not an abelian group. Then any function in* NC^1 *can be represented by a family of polynomial length expressions over L.*

Proof. This is a direct consequence of Theorem 4.8.4, since the OR of two bits and the negation of one bit can be represented over L. More explicitly, let h be any element of L different from the identity 1. Assume that h represent the boolean value 0 while the identity represent the value 1. Then, for any $x, y \in \{1, h\}$ we have

$$OR(x, y) = w_{2,h}(x, y)$$

Now, let h^λ be the unique element of L such that $h^\lambda h = 1$. Then we also have

$$NOT(x) = u_{h,h^\lambda}(x) \cdot h$$

Let t be the maximum between the lengths of the expressions OR and NOT. Then, any boolean formula of depth $k \log n$ can be represented by an expression over L of length at most $t^{3k \log n}$, which is polynomial. □

We now state a generalization of a theorem of Barrington [5] saying that the word problem over any nonsolvable group is complete for NC^1.

Theorem 4.8.6 *If G is a groupoid that contains a nonsolvable loop, then the problem of evaluating an expression over G is complete for* NC^1 *under* AC^0*-reductions.*

Proof. The problem of evaluating an expression over any fixed finite groupoid G is in NC^1. This is because the set of expressions over any finite groupoid that evaluate to some given element forms a parenthesis language, and therefore, belongs to NC^1 (see [17]). It remains to show that any function in NC^1 can be reduced to this problem when G contains a nonsolvable loop.

If L is a nonsolvable loop of G, then there exists a morphism $\varphi : L \to S$, where S is a nonabelian simple loop.

By Corollary 4.8.5, any function $f : \{0,1\}^* \to \{0,1\}$ in NC^1 is representable by a family of polynomial-length expressions over S. Let $w = w(x_1, \ldots, x_n)$ be such an expression for inputs of length n. Since we are dealing with Boolean functions, each variable x_i can only take two possible values $a, b \in S$. Suppose that a represents the value 0 and b the value 1. Choose any $s \in \varphi^{-1}(a)$, $t \in \varphi^{-1}(b)$, and define the mapping $\theta : \{0,1\} \to L$ by $\theta(0) = s$ and $\theta(1) = t$.

Finally, let $v(x_1, \ldots, x_n)$ be an expression over L defined from w by replacing each constant c in w by any element in $\varphi^{-1}(c)$. Then, for any $x_1 \cdots x_n \in \{0,1\}^n$, we have that $f(x_1, \ldots, x_n) = 1$ if and only if $v(\theta(x_1), \ldots, \theta(x_n)) \in \varphi^{-1}(b)$. Clearly, the reduction from f to v is a simple projection. □

4.9 Solvable loops

It is conjectured in [7] that the problem of evaluating a word over a solvable group is not complete for NC^1. However, we can construct a solvable loop of order 10 for which the problem of evaluating an expression is complete for NC^1.

Let Z_5 be the cyclic group of order five, and let G be the loop of order five defined by the following multiplication table.

	5	6	7	8	9
5	5	6	7	8	9
6	6	5	9	7	8
7	7	8	6	9	5
8	8	9	5	6	7
9	9	7	8	5	6

Since $8 \cdot 9 \neq 9 \cdot 8$ and $(6 \cdot 7) \cdot 8 \neq 6 \cdot (7 \cdot 8)$, G is neither commutative nor associative. Moreover, G is simple since it is of prime order. Hence, by Corollary 4.8.5, the problem of evaluating an expression over G is complete for NC^1. Now, define the loop $B = (B, \cdot)$ over the set $\{0, \ldots, 9\}$ using the multiplication table

	$0 \cdots 4$	$5 \cdots 9$
0 $\vdots$ 4	Z_5	G
5 $\vdots$ 9	G	Z_5

where a region labeled Z_5 (resp. G) corresponds to the multiplication table of Z_5 (resp. G). It should be clear that Z_5 is a normal subloop of B and that B/Z_5 is isomorphic to Z_2. Hence, B is a solvable loop.

Let $B' = (B, *)$ be the loop isotopic to B whose product is defined by $a * b = a \cdot \alpha(b)$, where α is the permutation $(0,5)(1,6)(2,7)(3,8)(4,9)$. The multiplication table of B' can be represented as follows (the identity is 5).

	$0 \cdots 4$	$5 \cdots 9$
0 $\vdots$ 4	G	Z_5
5 $\vdots$ 9	Z_5	G

Clearly, G is a subloop of B'. Thus, by Theorem 4.8.6, the problem of evaluating an expression over B' is complete for NC^1. Moreover, by Lemma 4.8.2,

any language that can be represented over B' can also be represented over B. We conclude that the problem of evaluating an expression over the solvable loop B is complete for NC^1.

Observe that the multiplication semigroup of B contains a nonsolvable simple subgroup. This is because $L(1)^{15} = (5,6)$ and $L(3)^2R(1)^{24} = (6,7,8,9,5)$ generate a group isomorphic to S_5. Thus, a solvable loop can have a nonsolvable multiplication group. Moreover, the problem of evaluating an expression over any such loop is complete for NC^1.

Theorem 4.9.1 *Let G be any finite groupoid with identity. If the multiplication semigroup $\mathcal{M}(G)$ contains a nonabelian simple group, then the problem of evaluating an expression over G is complete for* NC^1 *under* AC^0*-reductions.*

Proof. In [5] it is shown that the problem of evaluating a program over any nonabelian simple group D is complete for NC^1. Furthermore, we can suppose without loss of generality that on any input the program yields either the identity (and accepts) or some other fixed element (and rejects).

The proof follows from Theorem 2.6.6 □

Chapter 5

Growing Groupoids

5.1 Programs over growing groupoids

In this section, we generalize further the definition of recognition by programs. We will allow our model to use a different groupoid for each input length.

Definition 5.1.1 *Let $p : \mathbb{N} \to \mathbb{N}$ be a function and let $\mathcal{F} = (G_i)_{i \geq 0}$ be a family of groupoids such that G_n has order bounded above by $p(n)$, for each $n \geq 0$. A language $L \in A^*$ is said to be recognized by programs over $\mathcal{F}$ if for each $n \geq 0$ there exists a program P_n over G_n such that P_n accepts precisely those words in $L \cap A^n$. We also say that L is recognized by programs over groupoids of order p. Parenthesis programs over $\mathcal{F}$ are defined in the obvious way.*

In particular, if p is polynomial (resp. constant, exponential) then L is said to be recognized by polynomial (resp. constant, exponential) order programs over groupoids. Constant order programs are equivalent to programs over fixed groupoids as defined in Chapter 2.

Proposition 5.1.2 *Let k be any integer and $t : \mathbb{N} \to \mathbb{N}$ be any function. If $L \subseteq A^*$ is a language recognized by a family $P = (P_n)_{n \geq 0}$ of programs of length t over groupoids of order k, then L is recognized by a family $Q = (Q_n)_{n \geq 0}$ of programs of length t over a finite groupoid.*

Proof. Let M be the set of all pairs (G, F) such that G is a groupoid of order $k+1$ with an identity denoted 1, and F is a subset of G not containing

1. Let $(G_1, F_1), (G_2, F_2) \ldots, (G_m, F_m)$ be an enumeration of the elements of M and define $H = G_1 \times \cdots \times G_m$.

A program P_n over G_s with accepting set F_s can be simulated by a program Q_n over H as follows. The length of Q_n is the same as that of P_n. If the c^{th} instruction of P_n is $\langle i_c, f_c \rangle$, where $1 \leq i_c \leq n$ and $f_c : A \to G_s$, then the c^{th} instruction of Q_n is $\langle i_c, \phi_c \rangle$, where $\phi_c = (g_1^c, \ldots, g_m^c)$, $g_s^c = f_c$, and for all $j \neq s$, g_j^c is the constant function mapping each element of a to the identity 1. The accepting set of Q contains all elements of $\{(a_1, \ldots, a_m) \mid a_j \in G_j\}$ such that some a_j belongs to F_j. □

Unless otherwise specified, we will assume that all programs have polynomial length.

Program uniformity must be adapted when groupoids are growing. In particular the product of two elements must be computable and it must be decidable if a given element belongs to the accepting set.

Definition 5.1.3 *Let C be a complexity class. A family of programs $(P_n)_{n\geq 0}$ over $(G_n)_{n\geq 0}$ is said to be C-uniform whenever the three following conditions are satisfied.*

1. *On input (w, a, b) the problem of computing the product ab in $G_{|w|}$ belongs to C.*

2. *Moreover, on input (w, a), the problem of determining if a is in the accepting set of $P_{|w|}$ is in C.*

3. *On input (w, k), the problem of computing the k^{th} symbol of $P_{|w|}$ is in C, a symbol being either a parenthesis or an instruction. Moreover, on input w, the exact length of $P_{|w|}$ is computable in C.*

Remark. The last condition in the above definition implies that computing the element of $G_{|w|}$ produced by the k^{th} instruction can be done in C since the input alphabet is finite.

A first observation deals with L-uniform polynomial-order programs: they are no more powerful then constant order programs. Indeed, if each element of a groupoid G_n is represented with $O(\log n)$ symbols, then the word problem over G_n can be solved with a nondeterministic pushdown automaton using $O(\log n)$ cells on its auxiliary tape and working in polynomial time.

Non-uniform exponential-order programs are all powerful: any language whatsoever can be recognized by non-uniform exponential-order programs over cyclic groups. To see this let $L \in A^*$ and let G_n be the additive cyclic group of order $|A|^n$. For each $n \geq 0$ let P_n be a program over G_n such that P_n has length n and the i^{th} instruction looks at the i^{th} symbol w_i of the input and yields the number $w_i|A|^{i-1}$. So, L can be recognized by programs over $(G_n)_{n\geq 0}$ by taking the accepting set of G_n to be all those numbers for which a word in L is the $|A|$-ary representation.

However, if we restrict the programs to be uniform and if we use a finite accepting set, then programs over exponential-order groupoids become a non-trivial and interesting model of computation as we will see later. This situation can be compared with exponential-size semi-bounded Boolean circuits of logarithmic depth. As a non-uniform model of computation they can recognize any language, but if we restrict the direct connection language to be in P, then they recognize precisely those languages in NP (see [81]).

5.2 Machines versus programs

We examine in this section the relationship between uniform programs over family of groupoids and Turing machines. It is not surprising that programs over groupoids can be related with nondeterministic auxiliary pushdown automata. Interestingly, it also appears that linear recognition naturally corresponds with recognition by Turing machines. We now make these statement more precise.

For any function $f : \mathbb{N} \to \mathbb{N}$ and any complexity class C, we write $f \in C$ whenever the problem of computing f belongs to C.

5.2.1 Turing machines

In order to simulate Turing machines with programs, we use an approach similar to that of [59] by asking the machines to be oblivious, i.e. the head moves depend only on the length of the input, not on the input itself. Actually we will only need that Turing machines be *read-oblivious*, i.e. the behavior of the input head only is required to be oblivious.

The following lemma can be improved easily, but this version will be sufficient for our purpose.

Lemma 5.2.1 *Let $t(n) \in \Omega(n)$, $t(n) \in$ DTIME-SPACE$(t(n), \log t(n))$. Any TM M working in time $t(n)$ and space $s(n)$ can be simulated by an oblivious TM working in space $O(s(n) + \log t(n))$ and time $O(nt(n) \log n \log t(n))$.*

Proof. We construct a TM N that simulates M and stops after exactly $p(n)$ steps, for some $p(n) \in O(t(n) \log t(n))$. On input w of length n, N computes $t(n)$ on a special tape. This can be done in time $O(t(n))$ and space $O(\log t(n))$. Then, N starts the simulation, decrementing the number on its special tape after each move of M. The decrement can be done in time $2\lceil \log t(n) \rceil$. The machine stops when the special tape contains 0, and accepts if and only if M has already accepted. The total time is $O(t(n) \log t(n))$ and the space is $O(s(n) + \log t(n))$. After this step, we have a machine that takes the same time on any inputs of the same length.

A read-oblivious TM N' can simulates N as follows. The machine N' simulates N using an extra tape to move read-obliviously. To do this, N' successively scans its input from left to right and then from right to left using the extra tape to memorize the position of the input head of N. N' simulates each step of N in $2n \log n$ steps by scanning obliviously the input and modifying the extra tape which uses at most $\log n$ cells. Thus, the total space needed for this remains $O(s(n) + \log t(n))$ and the total time is $O(n \log nt(n) \log t(n))$. □

Proposition 5.2.2 *Let $t(n)$ be a function in* DTIME-SPACE$(t'(n), s'(n))$. *Any language $L \subseteq A^*$ recognized by a read-oblivious nondeterministic Turing machine M running in time $t(n)$ and space $s(n)$ is linearly recognized by a* DTIME-SPACE$(t'(n) + s(n),\ s'(n) + s(n) + \log t(n))$*-uniform family of programs of length $O(t(n))$ over groupoids of order $2^{O(s(n))}$.*

Proof. We adapt here an idea of [12]. We will assume that M scans its input from left to right and then from right to left $p(n)$ times, where $p(n)$ is a power of two such that $t(n) = 2np(n)$. (If M does not satisfy this property, we can use a Turing machine that simulates M and stopped after exactly $2np(n) \geq t(n)$ steps.) Assume furthermore that, from any state, M has at most two choices. Since the space is bounded by $s(n)$, the number of possible configurations is at most $d^{s(n)}$, for some $d > 0$.

Let X_n, Y_n and Z_n be three distinct copies of the set of configurations of M for inputs of length n. Define the set $B_n = X_n \cup Y_n \cup Z_n \cup A \cup \{0, t\}$. Let $a \in A$, $x, x_1, x_2 \in X_n$, $y \in Y_n$ and $z \in Z_n$, where x_1, x_2 are the configurations reached by M from x whenever M uses respectively the first and the second choices for a move after reading character a. Moreover, y is the copy of x_1 and z is the copy of x_2. We define a product on B_n as follows.

1. $tx = y$
2. $ya = x_1$
3. $xa = z$
4. $tz = x_2$
5. all other products yield 0

The first two lines in the above definition imply that $(tx)a = x_1$ while lines 3 and 4 imply that $t(xa) = x_2$. Hence, nondeterminism can be simulated by nonassociativity. More precisely, we have that the oblivious machine starts at

configuration x_0 and scans its input $w = a_1 \cdots a_n$ from left to right (and from right to left) $p(n)$ times before stopping. Then, the string

$$\phi(w) = t^{2np(n)} x_0 (w\tilde{w})^{p(n)} ,$$

where $\tilde{w}$ denotes the mirror image of w, evaluates to an accepting configuration in X_n if and only if M accepts w. Observe that the parenthesization must be linear.

ϕ is a projection that can be performed in the obvious way by a program P_n over B_n. To see that it is uniform, we must show that the three conditions in Definition 5.1.3 are satisfied. Using a reasonable encoding for the elements of B_n, it should be clear that taking the product of two elements and determining if a given element represents an accepting configuration can be done in time $O(s(n))$. The length of the program is $4np(n) + 1 = 2t(n) + 1$ which is computable, by assumption, in time $O(t'(n))$ and space $O(s'(n))$. Now, suppose that given (w, k) we want to compute the k^{th} instruction in $P_{|w|}$. Determining if $k \leq 2np(n)$ and if $k = 2np(n) + 1$, in which case the instruction yields respectively t and x_0, can be done in time $O(t'(n))$ and space $O(s'(n))$. Otherwise, we must determine the position of the input considered by the k^{th} instruction. This position i_k can be computed as follows. First compute $m = 1 + (k - 2np(n) - 2) \bmod 2p(n)$. Since $p(n)$ is a power of 2, both the subtraction and the modulus can be done in linear time (i.e. in time $\log t(n) \leq t'(n)$) and in space $O(\log t(n))$. Then, we have $i_k = m$ if $m < n$, and $i_k = n - m$ otherwise. This shows that $(P_n)_{n \geq 0}$ is DTIME-SPACE$(t'(n) + s(n), s'(n) + s(n) + \log t(n))$-uniform. □

When the machine is deterministic, there is no need for the symbol t in the above proof, and the groupoids can be redefined so that the programs are left-to-right.

Proposition 5.2.3 *Let $t(n)$ be a function in* DTIME-SPACE$(t'(n), s'(n))$. *Any language $L \subseteq A^*$ recognized by an oblivious deterministic Turing machine M running in time $t(n)$ and space $s(n)$ is left-to-right recognized by a*

DTIME-SPACE($t'(n)+s(n), s'(n)+s(n)+\log t(n)$)-*uniform family of programs of length* $O(t(n))$ *over groupoids of order* $2^{O(s(n))}$.

Proof. Redefine $B_n = X_n \cup A \cup \{0\}$. For any $a \in A$ and any $x, x' \in X_n$ such that x' is the configuration reached by M from x after reading symbol a, we define $xa = x'$; all other products yield 0.

Letting $\phi(w) = x_0(w\tilde{w})^{p(n)}$, we get a program that left-to-right evaluates to an accepting configuration if and only if M accepts w. The uniformity is shown as in the proof of Proposition 5.2.2 □

Proposition 5.2.4 *Any language linearly recognized by a* DTIME-SPACE($t(n)$, $s(n)$)*-uniform family of programs of length* $l(n)$ *over groupoids of order* $z(n)$ *is also recognized by a nondeterministic Turing machine in time* $O(l(n)t(n))$ *and space* $O(\log z(n) + s(n))$.

Proof. Let $(P_n)_{n\geq 0}$ be such a family of programs over groupoids $(G_n)_{n\geq 0}$. We will construct a Turing machine M that simulates these programs. Let $Y(i)$ be the element of G_n generated by the i^{th} instruction of P_n. On input w of length n, M does the following computation.

1. Compute $l(n)$ and guess a number k between 1 and $l(n)$.
2. Initialize two pointers $a \leftarrow k$ and $b \leftarrow k$.
3. $g \leftarrow Y(k)$
4. Iterate $l(n) - 1$ times the following step
5. Choose nondeterministically between

   (i) $a \leftarrow a - 1$
   $h \leftarrow Y(a)$
   $g \leftarrow hg$

(ii) $b \leftarrow b+1$

$h \leftarrow Y(b)$

$g \leftarrow gh$

6. Accept iff g belongs to the accepting set of P_n.

The space used is $O(\log z(n) + s(n))$ and the time is $O(l(n)t(n))$. □

As a consequence of Lemma 5.2.1 and Propositions 5.2.2 and 5.2.4 we have the following theorems.

Theorem 5.2.5 ([53]) NP *is equal to the class of languages linearly recognized by P-uniform programs over groupoids of exponential order.*

Theorem 5.2.6 NL *is equal to the class of languages linearly recognized by L-uniform programs over groupoids of polynomial order.*

Proposition 5.2.7 *Any language linearly recognized by a* DTIME-SPACE($t(n)$, $s(n)$)*-uniform family of parenthesized programs of length $l(n)$ over groupoids of order $z(n)$ is also recognized by a deterministic Turing machine in time $O(l(n)t(n))$ and space $O(s(n) + \log z(n))$. Moreover the machine is read-oblivious.*

Proof. We just have to adapt the proof of Proposition 5.2.4 for the case where $Y(i)$ can be an open or a closed parenthesis.

1. Compute $l(n)$.
2. Compute $Y(i)$ until a k is found such that both $Y(k)$ and $Y(k+1)$ belong to G_n.
3. $g \leftarrow Y(k)Y(k+1)$
4. Initialize two pointers $a \leftarrow k-2$ and $b \leftarrow k+2$.
5. Iterate $l(n)-1$ times the following step

6. Compute $Y(a)$ and $Y(b)$: exactly one of two following cases must happen

   If $Y(a)$ is an open parenthesis then

   $g \leftarrow gY(b)$; $a \leftarrow a-1$; $b \leftarrow b+2$

   If $Y(b)$ is a closed parenthesis then

   $g \leftarrow Y(a)b$; $a \leftarrow a-2$; $b \leftarrow b+1$

7. Accept iff g belongs to the acepting set of P_n.

The machine is oblivious because the parenthesization of each program is fixed. □

From Proposition 5.2.3 and Proposition 5.2.7 we immediately have the following theorems.

Theorem 5.2.8 P *is equal to the class of languages linearly (LTR) recognized by P-uniform parenthesized programs over groupoids of exponential order.*

Theorem 5.2.9 L *is equal to the class of languages linearly (LTR) recognized by L-uniform parenthesized programs over groupoids of polynomial-order.*

Theorem 5.2.8 can be improved in one direction.

Theorem 5.2.10 P *is equal to the class of languages recognized by P-uniform parenthesized programs over groupoids of exponential order.*

Proof. It suffices to show how P-uniform programs $(P_n)_{n\geq 0}$ over groupoid $(G_n)_{n\geq 0}$ can be simulated by a deterministic Turing machine M. On input w of length n, M begins by computing the length of P_n. Then, M computes the element of G_n generated by each instruction and writes this sequence on its tape. Finally M evaluates in a straightforward way this well-parenthesized expression. The time needed is polynomial since the expression has polynomial length, each product can be computed in polynomial time, and the machine can determine in polynomial time if the result belongs to the accepting set of P_n. □

5.2.2 Pushdown automata

Proposition 5.2.11 *Any language recognized by a* DTIME-SPACE($t(n), s(n)$)-*uniform family of programs of length* $l(n)$ *over groupoids of order* $z(n)$ *is also recognized by an AuxNPDA in time* $O(l(n)t(n) + l(n)\log z(n))$ *and space* $O(\log z(n) + \log l(n) + s(n))$.

Proof. On inputs of length n the AuxNPDA M simulates the program P_n as follows. First M computes $l(n)$, the length of P_n. Then, it makes $l(n)$ iterations of the following procedure. At iteration i, M computes the element $g \in G_n$ produced by the i^{th} instruction of P_n, and decides nondeterministically whether it pushes g on the stack or multiplies it with the element on the top of the stack. The machine accepts if after the $l(n)$ iterations the stack is empty and the element computed belongs to the accepting set of P_n.

Observe that pushing and popping an element takes time $O(\log z(n))$. Each iteration takes time $O(t(n) + \log z(n))$ and space $O(s(n) + \log z(n))$. Hence, the total time is $O(l(n)t(n) + l(n)\log z(n))$ and the space used is $O(\log z(n) + \log l(n) + s(n))$. □

Proposition 5.2.12 *Any language recognized by a* DTIME-SPACE($t(n), s(n)$)-*uniform family of parenthesized programs of length* $l(n)$ *over groupoids of order* $z(n)$ *is also recognized by an AuxDPDA in time* $O(l(n)t(n) + l(n)\log z(n))$ *and space* $O(\log z(n) + s(n))$.

Proof. The proof is similar to that of Proposition 5.2.11. The only difference is that at each iteration, the program symbol computed by the machine M is not necessarily an instruction, it can be an open or a closed parenthesis. Hence, M does not have to make any nondeterministic decision, it merely pushes an element whenever the next program's symbol is an instruction, and takes the product with the top element if it is a closed parenthesis. Observe that open parenthesis can be ignored since a program always yields a well-parenthesized expression. □

Proposition 5.2.13 *Let $t(n)$ be a function in* DTIME-SPACE$(t(n), s(n))$, *where $s(n) \geq \log n$. Any language recognized by a read-oblivious AuxNPDA in time $t(n)$ and space $s(n)$ is also recognized by* DTIME-SPACE$(t(n), s(n))$-*uniform programs of length $O(2^{O(s(n))}t(n))$ over groupoids of order $2^{O(s(n))}$.*

Proof. We will see the AuxNPDA M as a uniform family of 2-way oblivious NPDA $M_n = (Q_n, A, S, \delta_n, q_0, s_0)$ where Q_n is a set of states (corresponding to the state, the head positions and the content of the work tape of M), A is the input alphabet of M, S is the stack alphabet of M, q_0 is the initial state, s_0 is the initial stack symbol of M, and $\delta_n : Q_n \times S \times A \to Q_n \times (S \cup S^2 \cup \{\lambda\})$ is the transition function. Observe that the cardinality of Q_n is $2^{O(s(n))}$.

We can assume that initially M_n has s_0 on the top of its stack, it never pushes s_0 thereafter, its first move is a push move, and it accepts whenever it finds s_0 on the top of its stack. Assume also that there are three kinds of moves M_n can do: a pop move, a push move, or a null move (where the stack height remains unchanged).

The proposition will be proved in three steps.

Step 1. Let us fix the input length n and denote δ_n by δ. For each letter a of the input alphabet A, we define the following three sets.

$$\text{PUSH}_a = \{\langle p, s, q, t\rangle \mid (q, st) \in \delta(p, s, a)\}$$

$$\text{POP}_a = \{\langle p, s, q, \lambda\rangle \mid (q, \lambda) \in \delta(p, s, a)\}$$

$$\text{NULL}_a = \{[p, s, q, t] \mid (q, t) \in \delta(p, s, a)\}$$

Let $P_a = \text{PUSH}_a \cup \text{POP}_a \cup \text{NULL}_a$, let $R = \{\langle p, s, q, t\rangle, \langle p, s, q, \lambda\rangle, [p, s, q, t] \mid p, q \in Q_n, s, t \in S\}$, and let $P = R \cup \{0\}$ be a groupoid with the following product.

- 0 is absorbing
- $\langle p, s, q, t\rangle\langle q, t, r, \lambda\rangle = [p, s, r, s]$
- $[p, s, q, t]\langle q, t, r, \lambda\rangle = \langle p, s, r, \lambda\rangle$

- all other products yield 0.

The second product is motivated by the fact that a push move followed by a pop move is equivalent to a null move. The third product corresponds to the fact that a null move followed by a pop move is equivalent to a pop move.

Let $w \in A^*$ be an input of length n and consider a fixed computation of M_n on w. Suppose that at time i, M_n is at some state p, the top of the stack is s, and the input head scans the symbol $a \in A$. We define the function $f : \{1, \ldots, t(n)\} \to P$ as follows.

$$f(i) = \begin{cases} \langle p, s, q, t \rangle & \text{if at step } i, M_n \text{ pushes symbol } t \text{ on the stack and moves to state } q. \\ \langle p, s, q, \lambda \rangle & \text{if at step } i, M_n \text{ pops symbol } s \text{ off the stack and moves to state } q. \\ [p, s, q, t] & \text{if at step } i, M_n \text{ replaces symbol } s \text{ on the stack by } t \text{ and moves to state } q. \end{cases}$$

By assumption, we have $f(1) \in PUSH_a$ with $p = q_0$ and $s = s_0$, and we have $f(t(n)) \in \mathrm{POP}_a$.

Then, it is easy to verify that the given computation of M_n on w leads to an accepting configuration if and only if the string $f(1) \cdots f(t(n))$ can be evaluated to an element of the form $[q_0, s_0, p, s_0]$.

Step 2. This idea can be used to reduce the language recognized by M to a word problem over a groupoid that contains S. One difficulty is that if w is not fixed and if the computation path is not given, then the function f is not well defined. This can be solved by defining a groupoid $Q = 2^P$, where the product of two sets $S, T \in Q$ corresponds to the set of all elements resulting from the multiplication of one element in S and one element in T.

Define the function $g : A \to 2^P$ by $g(a) = P_a$, for all $a \in A$. Now, define the program D_n, for inputs of length n, as $v_1 v_2 \cdots v_{t(n)}$ where, for $j > 1$, the instructions are $v_j = \langle i_j, g \rangle$, i_j being the position of the input head of M at time j. Moreover, we set $v_1 = \langle 1, \alpha \rangle$, where $\alpha(a) = \langle q_0, s_0, p, s \rangle$ and $\delta(q_0, s_0, a) = (p, s)$. On input $w = a_1 \cdots a_n$, D_n yields a string $u = u_1 \cdots u_{t(n)}$ and accepts its input if and only if there exists a parenthesization of u resulting into a set in Q that contains an element in P of the form $[q_0, s_0, p, s_0]$. Hence,

w is accepted by D_n if and only if there exists a symbol f_{i,a_i} in each set u_i such that $f_{i_1,a_1} \cdots f_{i_{t(n)},a_{t(n)}}$ can be evaluated to $[q_0, s_0, p, s_0]$. This happens if and only if w is accepted by M.

Step 3. We must decrease the order of Q_n which is $2^{2^{O(s(n))}}$. We have defined elements of Q to be sets because we don't know at priori what element should be chosen in each u_i of the above programs. However, the nondeterministic selection of an element in a set $u \in Q$ can be done using nonassociativity in a smaller groupoid.

Let P' be a copy of P and define the following product on $G = P \cup P' \cup \{0, 1, c\}$, where $c, 0$ and 1 are new elements. There is an identity, which is 1, and 0 is an absorbing element. The product on P is defined as previously. We have $a'b' = b'$, $a'c = a$, $a'b = a$, for all $a', b' \in P'$ such that a' is the copy of $a \in P$. All other products yield 0.

Consider any ordering of the elements of P' and let $u \subseteq P'$. Let $y_u = y_1 \cdots y_{|P|}C$, where C is a new symbol and such that y_i is the i^{th} element of P' if this element belongs to u, otherwise $y_i = 1$.

Obviously, u contains some element $s' \in P'$ if and only if y_u can be evaluated to s.

It is also straightforward to check that $y_{u_1} \cdots y_{u_{t(n)}}$ evaluates to an element of the form $[q_0, s_0, p, s_0]$ if and only if w is accepted by M.

The length of the program is $|P|t(n) = 2^{O(s(n))}t(n)$ and the order of the groupoid is $2|P| + 3 = 2^{O(s(n))}$.

It remains to show that the program is DTIME-SPACE$(t(n), s(n))$-uniform. The only difficulty resides in determining what is the j^{th} instruction, given any j.

Let each element $v \in P'$ be encoded in binary with a string w' of the form $w' = psqtk$, where p, q are states of M_n, s is a stack symbol, t is a stack symbol or λ, and k says if v is a pop, a push, or a null move. Suppose that the respective length of p, s, q and t is the same for any element of P'. Suppose also that $l = |w'|$ and that 1^l represents no element of P'. Then, given w', one can compute in linear time p, s, q, t and k, plus the kind of move represented

by k (push, pop, null). Given any $a \in A$, it is also possible, in linear time, to determine if w' represents an element which is the copy of an element $w \in P_a$, since M_n has a constant number of possible moves from state p and top symbol s.

We can slightly modify the definition of y_u, for any $u \subseteq P'$, such that its length be 2^l: let $y_u = x_0 \cdots x_{2^l-1}$, where x_{2^l-1} represents the element c, x_i represents $s \in P'$ if the binary representation of i corresponds to an element $s \in u$, otherwise x_i represents 1. Observe that the length of the program is still $2^{O(s(n))}t(n)$ and the order of the groupoid is still $2^{O(s(n))}$.

Our program consists of a sequence of $t(n)$ blocks $Y_{u_1} \cdots Y_{u_{t(n)}}$ where each block contains 2^l instructions. Given j one can determine the number $b(j)$ of the block containing the j^{th} instruction and the position $p(j)$ of this instruction inside this block. Clearly, this can be done in linear time, i.e. in time $O(s(n))$.

We observe that for any $i \leq t(n)$, each instruction in a block looks at the same input position. Thus, in order to determine the position of the input considered by the j^{th} instruction, it only suffices to simulate M on any input of length n, using any nondeterministic choices and ignoring the stack. The desired position k corresponds to the location of the input head after $b(j)$ steps. This can be done in less than $t(n)$ steps, using at most $s(n)$ memory cells.

At this point we know that the j^{th} instruction has the form $\langle k, g \rangle$ and it remains to find what is the function $g : A \to P$. If the binary representation of $p(j)$ is 1^l, then g is the constant function $g(a) = c$. Otherwise, as we explain above, we can determine in linear time if the element v' represented by $p(j)$ is an element of P'. If v' does not represent such an element, then $g(a) = 1$ for all $a \in A$, otherwise we have $g(a) = v'$ if $v \in P_a$, and $g(a) = 1$ if $v \notin P_a$. □

Propositions 5.2.11 and 5.2.13 have the following immediate consequence.

Theorem 5.2.14 ([12]) SAC^1 *is equal to the the class of languages recognized by* L*-uniform programs over polynomial-order groupoids.*

Theorem 5.2.15 ([53]) NP *is equal to the the class of languages recognized by* P*-uniform programs over exponential-order groupoids.*

5.3 Tree-like circuits

Define a *subcircuit* S of a Boolean circuit C to be a subgraph of C that satisfies the following properties. The output gate of C is in S. Each AND gate in S has exactly the same children it has in C. Each OR gate in S has exactly one child chosen among those it has in C. Nothing else is in S.

A *subtree* of a Boolean circuit C is a tree obtained by expanding, in the obvious way, a subcircuit of C. Given some input for C, a subtree T is said to be accepting if it outputs 1, i.e. every input gate in T has value 1.

The *degree* of a gate in a Boolean circuit is defined recursively as follows. The degree of a constant is 0, the degree of a variable is 1, the degree of an OR gate is the maximal degree of its children, and the degree of an AND gate is the sum of the degrees of its children. The degree of a single-output Boolean circuit is defined as the degree of its output gate.

In [81] it is proved that NP (resp. LOGCFL) is equivalent to the class of languages recognized by P-uniform (resp. L-uniform) families of exponential (resp. polynomial) size circuit with polynomial degree.

It is useful to consider a further restriction on the Boolean circuits. A *skew circuit* is a circuit where among all the children of any AND gate, at most one of them is not an input gate. It is easy to verify that polynomial-depth skew circuits have polynomial degree.

It can be proved (see [81]) that L-uniform polynomial-size skew circuits recognize precisely the class of languages NL, while P-uniform exponential-size polynomial-depth skew-circuits correspond to NP.

Hence, we see that by restricting polynomial tree-size circuits to be skew, we make the related classes of complexity go from LOGCFL to NL, when the size is polynomial, while we still get NP, when the size is exponential.

Definition 5.3.1 *A building-block is a multiple-output depth-2 circuit with*

unbounded fan-in OR gates on the output level, each of which takes fan-in 2 AND gates as inputs. An input-block is a sequence of input gates. Input-blocks and building-blocks are generically referred to as blocks.

Definition 5.3.2 *A tree-like circuit is a semi-bounded Boolean circuit consisting of blocks connected together in the following manner. With each building-block B we exclusively associate two blocks B_1 and B_2, respectively called the left and right child of B. Each AND gate in B receives one input from a gate in B_1 and one input from a gate in B_2. There is a unique distinguished block, called the output-block, that feeds in no other block. In this way, the blocks have the structure of a binary tree where the root is the output-block, the leaves are the input-blocks, and the inner nodes are the building-blocks.*

We observe that any subtree in a tree-like circuit contains exactly one OR gate and one AND gate from each building-block, and one input gate from each input-block. The degree D of a tree-like circuit corresponds precisely to the number of input-blocks it contains, and so the total number of blocks is $2D - 1$.

The largest number of gates in a block of a tree-like circuit is called the *block-size* of the circuit. Observe that the number of gates in such a circuit is bounded above by the product of its degree and its block-size multiplied by 2. In particular, the languages recognized by L-uniform tree-like circuits of polynomial size are in SAC^1. The converse is given by the following proposition.

Proposition 5.3.3 *Any language recognized by a family of semi-bounded circuits with size s and depth d, is also recognized by a family of tree-like circuits having depth $2d$, block-size $2(s + sd)^2$ and degree 2^d.*

Proof. We exhibit a way of doing the transformation. The proof results from a three-step transformation of the circuit.

1. **All paths must have the same length.** This step is only needed to facilitate the construction. It suffices to insert before each input gate as

many OR gates as necessary to make the distance between this input gate to the root equal to d. This yields a circuit C_1 with at most $s + sd$ gates.

2. **The gates must alternate.** Starting from C_1, we want to construct a circuit C_2 such that the output is an OR gate and all gates on the first level (those fed by input gates) are AND gates. This can be done by adding at most $s + sd$ gates. Furthermore, we want that the type of the gates alternates between two consecutive levels. This is done by adding an OR gate (resp. AND gate) between two consecutive AND gates (resp. OR gates) The number of gates added in this way is bounded above by the number of pairs of gates. So, the total number of gates in C_2 is bounded by $2(s + sd)^2$, and the depth is at most $2d$.

3. **Transform the proofs into trees.** Consider the lowest level l of AND gates in C_2 (i.e. the level near the output). Duplicate the part of the circuit that is above l into two copies P_1 and P_2, and let each AND gate on level l have one input from P_1 and one from P_2. We repeat this process iteratively both in P_1 and in P_2 to get a tree-like circuit C_3. The degree of C_3 is at most 2^d, each block has no more than $2(s+sd)^2$ gates, and the depth is still $2d$.

□

Constructing a uniform tree-like circuit from a semi-bounded one, can be done by observing how the properties defining tree-like circuits can be translated for alternating Turing machines. Recall that in a semi-bounded ATM there are no two consecutive universal configurations along any computation path (see [81]).

Definition 5.3.4 *We define a tree-like ATM M to be a semi-bounded ATM satisfying the following properties.*

1. *There are three kind of states: universal, existential and reading. At a reading state, the machine reads a bit of the input and halts. A reading state can only be accessed from a universal state.*

2. *Any universal configuration of the machine has exactly two successors called the left and right successors. These two configurations are existential or reading.*

3. *The machine M has a special tape and starts by writing 1 on it. When M is at a universal state, it moves the head of its special tape one cell to the right and writes 0 or 1 depending only if it makes a left or a right move.*

4. *Let A and B be two universal configurations describing the same content on the special tape. Then, the left (resp. right) successor of A is reading if and only if the left (resp. right) successor of B is reading.*

If we construct a family of circuits from a tree-like ATM using the method of Ruzzo [63], then we naturally obtain a family of tree-like circuits. Indeed, all configurations with the same special tape content correspond to a gate in the same block of the circuit.

Proposition 5.3.5 *Let $t(n), s(n)$ and $z(n)$ be functions such that $z(n)$ belongs to* DTIME-SPACE$(z(n), \log z(n))$. *A semi-bounded ATM M running in time $t(n)$, space $s(n)$, and using $z(n)$ alternations can be simulated by a tree-like ATM running in time $O(t(n)+z^2(n))$, space $O(s(n)+z(n))$, and using $O(z(n))$ alternations.*

Proof. We must satisfy the four conditions of Definition 5.3.4.

The first two conditions are standard. The time and the space increase only by a multiplicative constant (see [62]). The number of alternations also increases by a multiplicative constant, because M is semi-bounded (this would not be the case otherwise).

The third condition has no influence on the time and the number of alternations. The space however can be influenced since the number of tape cells used is now bounded by the number of alternations.

These two transformations give us an ATM M_2 using time $O(t(n))$, space $O(s(n) + z(n))$ and alternations $O(z(n))$

A simple way of satisfying the last condition is to force the machine to have the same number of alternations on any computation path. We construct an ATM M_3 that simulates M_2 as follows. First M_3 marks $k = \lfloor \log z(n) \rfloor + 1$ cells on a special tape. By assumption, this can be done in time $O(z(n))$ and space $O(\log z(n))$ This tape will be used to count the number of alternations. Then, M_3 begins the simulation, incrementing its counter each time it enters a universal state. The increment can be done in exactly $2k \in O(z(n))$ steps by starting at the left end of the counter, moving to the right end and returning to the initial position. Since M_3 will have to do $z(n)$ such increments during the whole process, the total time spent on this task is $O(z^2(n))$. At the time the counter reaches 2^k, M_3 halts and rejects. Otherwise, before simulating a reading state, M_3 alternates between universal and existential states until the counter reaches its maximal value. Then, M_3 can simulate the reading state of M_2. The time needed is $O(t(n) + z^2(n))$, the space is $O(s(n) + z(n))$, and the number of alternations remains $O(z(n))$. □

Theorem 5.3.6 *A language is in* NP *if and only if it is recognized by a P-uniform family of tree-like circuits with exponential block-size and polynomial degree.*

Proof. In [81], it is proved that any language in NP can be recognized by a semi-bounded ATM M_1 working in polynomial time and making a logarithmic number of alternations. Proposition 5.3.5 shows that M_1 can be simulated by a tree-like ATM M_2 working in polynomial time and using $O(\log n)$ alternations. As we already mentioned, Ruzzo's simulation of ATM with Boolean circuits preserves the tree-like property. So, M_2 can be simulated by a P-uniform

family C of tree-like circuits with exponential block-size and $O(\log n)$ depth. Indeed, any logarithmic depth semi-bounded circuit have polynomial degree.

The other direction is a consequence of a result from [S1] saying that any language recognized by P-uniform semi-bounded circuits of exponential size and polynomial degree is in NP. □

Theorem 5.3.7 *A language is in* LOGCFL *if and only if it is recognized by a L-uniform family of tree-like circuits with polynomial block-size and polynomial degree.*

Proof. The proof is essentially identical to the previous one. It is based on the equivalence between LOGCFL and the class of languages recognized by polynomial-size polynomial-degree semi-bounded circuits. □

Observe that a skew circuit does not remain skew after we apply the transformation of Proposition 5.3.3. However, the proof can be adapted. To do this, we need to introduce the notion of skew alternating Turing machine.

Definition 5.3.8 *A skew alternating Turing machine (skew ATM) is an ATM satisfying condition 1 and 2 of Definition 5.3.4, and such that from any universal configuration all moves, except possibly one, lead to a reading state.*

Without loss of generality, we will assume that from any universal state of a skew ATM, a left move always leads to a reading state. We call *right computation path*, a computation path such that all moves originating from a universal state are right moves.

One can verify that Ruzzo's proof [63], concerning the simulation of ATM by uniform circuits, preserves the skew property. Unfortunately, the simulation of uniform circuits by ATM given in [63] does not preserve the skew property. Nevertheless, we can bypass this difficulty by using the following observation.

Lemma 5.3.9 NTIME-SPACE$(t(n), s(n)) =$ *skew*ATIME-SPACE$(t(n), s(n))$

Proof. ($\subseteq$) Let M be a nondeterministic Turing machine working in time $O(t(n))$ and space $O(s(n))$. Suppose without loss of generality that the states of M are partitioned between existential states and reading states. An alternating TM N can simulate M as follows. On existential states, N does the same thing as M. On reading states which are final, N simply branches universally to two identical reading states. On nonfinal reading states, N guesses the correct input symbol b. Then, it branches universally to both a reading state, to verify its guess, and to the state corresponding to a move made by M when reading symbol b.

($\supseteq$) Let us see how a nondeterministic machine M can simulate a skew alternating TM N. M behaves differently from N only at universal states. In this case, M simulates sequentially the two possible moves. It first begins with a move leading to a reading state and rejects if this computation is rejecting. Since N always halts after being in a reading state, this part of the simulation can be done in constant time using only deterministic states. If M has not rejected, it continues its computation by simulating the second move. If M concludes the simulation of N without rejecting, then it accepts. □

Proposition 5.3.10 *Let $s(n) \in \Omega(\log n)$, $t(n) \in \Omega(n)$, and $s(n) \leq t(n)$. A language is recognized by a skew ATM in time $t^{O(1)}(n)$ and space $s(n)$ if and only if it is recognized by a DTIME$(\log t(n))$-uniform family of skew circuits of size $O(2^{s(n)})$ and depth $t^{O(1)}(n)$.*

Proof. It is proved in [81] that for $s(n) \in \Omega(\log n)$, $t(n) \in \Omega(n)$, and $s(n) \leq t(n)$, NTIME-SPACE$(t^{O(1)}(n), s(n))$ is equal to the class of languages recognized by DTIME$(s(n))$-uniform skew circuits of size $O(2^{s(n)})$ and depth $t^{O(1)}(n)$. The proof follows from this and Lemma 5.3.9. □

We must modify slightly the definition of tree-like ATM in the context of skew ATM. Actually, only the third condition needs to be changed. This is because, in order to access all its inputs, a skew circuit must have at least

linear depth. Hence, applying Proposition 5.3.5 on a skew ATM using $\log n$ space would result in a tree-like ATM using at least linear space.

Definition 5.3.11 *A skew ATM M is tree-like if it satisfies conditions 1,2 and 4 of Definition 5.3.4, and if it satisfies the following condition.*

3'. The machine M has a special tape used as a counter initialized to 0. Each time M moves to a universal state, it increments its counter.

The motivation of this definition is given by the following proposition.

Proposition 5.3.12 *Let $z(n)$ be in* DTIME-SPACE$(z(n), \log z(n))$. *Any skew ATM M using time $t(n)$, space $s(n)$, and alternation $z(n)$ can be simulated by a tree-like skew ATM N using time $O(t(n) \log z(n))$ and space $O(s(n) + \log z(n))$*

Proof. The first two conditions of Definition 5.3.4 are already satisfied and the third one causes no problem. It is only necessary to show how we can satisfy the last condition while preserving the skewness of the ATM.

The idea is similar to that used in Proposition 5.3.5. We construct N such that any right computation path has the same number of alternations. This is simply done by using the special tape to count in binary the number of alternations. This takes space $O(\log z(n))$. Each increment takes time $O(\log z(n))$. The total space is thus $O(s(n)+\log z(n))$ and the time is increased by a factor of $O(\log z(n))$.

The other difference with Proposition 5.3.5 is that the added universal configurations branch to an accepting configuration and to an existential configuration. □

Proposition 5.3.13 *Any DTIME($\log s(n)$)-uniform family C of skew circuits of size $s(n)$ and depth $d(n)$ can be simulated by a DTIME($\log s(n)$)-uniform family of tree-like skew circuits C'_n of size $(s(n)d(n))^{O(1)}$ and depth $d^{O(1)}(n)$.*

Proof. From Proposition 5.3.10, we have that C can be simulated by a skew ATM M running in time $O(d^{O(1)}(n))$ and space $O(\log s(n))$.

M can itself be simulated by a tree-like skew ATM M' using time $O(d^{O(1)}(n))$ and space $O(\log s(n) + \log d(n))$, by Proposition 5.3.12.

Finally, we can simulate M' with a uniform family C' of circuits of size $(s(n)d(n))^{O(1)}$ and depth $d^{O(1)}(n)$, using the construction of Ruzzo [63]. □

In [81], it is shown that NL (resp. NP) corresponds to the class of languages recognized by uniform skew circuits of polynomial depth and polynomial (resp. exponential) size. This and Proposition 5.3.13 give the following theorems.

Theorem 5.3.14 *A language is in* NL *if and only if it recognized by a family of uniform tree-like skew circuits with polynomial block-size and polynomial degree.*

Theorem 5.3.15 *A language is in* NP *if and only if it is recognized by a family of uniform tree-like skew circuits with exponential block-size and polynomial degree.*

5.4 Construction of a family of groupoids

5.4.1 Groupoids G_m

We observe that a proof in a tree-like circuit consists in a consistent selection of one OR-gate and one AND-gate in each block. Let us examine what is meant by consistent selection.

Consider any numbering of the gates of C. With any AND-gate g we associate triples of the form (a, c, b), where c is the number assigned to an OR-gate using g as input. If the left child of g is an input gate labeled with the variable x_1 then, $a = x_1$, otherwise a is the number assigned to the left child of g. The number b is defined similarly using the right child. Let B be a block with left child B_1 and right child B_2. Then, a consistent selection of gates implies that if an AND-gate of the form (a, c, b) has been selected in B, then we must choose a gate of the form (s, a, t) in B_1 and a gate of the form (u, b, v) in B_2.

Let C_m be a tree-like circuit whose gates are numbered such that (1) any two distinct gates have distinct numbers, (2) the output gate has number m, (3) no gate has number 0, and (4) if a gate in C_m has number k then the number assigned to any of its children is strictly smaller than k. Such a numbering will be called a *normal numbering bounded by* m. Furthermore, we will represent the input bits by 0 and m instead of 0 and 1. So, an input x to the circuit is accepted if the circuit outputs m and it is rejected if the circuit outputs 0.

We now describe the construction of a family of groupoids $(G_n)_{n\geq 1}$ such that the problem of evaluating C_m is reducible to the word problem over G_m. The above setting indicates that G_m can be defined over the set

$$G_m = \{(a, c, b) : 0 \leq a, b, c \leq m\}.$$

We will define a complete order relation between the triples in G_m. For any two triples (a, c, b) and (e, d, f) we write $(a, c, b) \leq (e, d, f)$ if and only if $am^2 + bm + c \leq em^2 + fm + d$. The product in G_m is defined as follows.

1. $(m, a, m)(a, c, b) = (m, c, b)$
2. $(a, c, b)(m, b, m) = (a, c, m)$
3. $(a, c, b)(e, d, f) = \begin{cases} (a, c, b) & \text{if } (e, d, f) \leq (a, c, b) \\ (e, d, f) & \text{otherwise} \end{cases}$

   for all cases not covered by 1. and 2.

5.4.2 Characterization of LOGCFL

Suppose that circuit C_n has n inputs $x_1, \ldots, x_n$. We will describe a way to encode C_n into a string $w = w(x_1, \ldots, x_n) \in (G_m \cup \{(x_i, c, x_i) : i \leq n, c \leq m\})^*$ such that for any $v = a_1 \cdots a_n \in \{0, m\}^n$, v is accepted by C_m if and only if $w(a_1, \ldots, a_n)$ can be evaluated to (m, m, m). We will associate the AND-gates of C_m with triples in G_m as explained above.

Each block B_k in C_m can be represented by a sequence of triples corresponding to the set of AND-gates contained in B_k. The sequences corresponding

to an input block that is a left child will be in decreasing order. All other sequences are in increasing order.

Let the blocks of C_m be $B_1, \ldots B_t$ with B_t the output block. With each block B_k we associate a sequence of blocks denoted β_k. If B_k is an input-block then $\beta_k = B_k$. If B_i and B_j are respectively the left and right child of B_k then $\beta_k = \beta_i B_k \beta_j$. Our string w will simply be the sequence β_t where each B_k is represented by a sequence of triples as explained above.

We must now prove our claim that for any $y = a_1 \cdots a_n \in \{0, m\}^n$, y is accepted by C_m if and only if $(m, m, m) \in G_m(w)$ where $w = w(a_1, \ldots, a_n)$.

We first show by induction on the depth of a gate c that if w can be partially evaluated to a word $u(m, c, m)v$, then the subcircuit rooted at gate c evaluates to 1.

Recall that w is not any word in G_m^*: it is the result of a reduction and has the structure discussed above.

Suppose first that c belongs to an input-block. Then, it should be clear that the first two rules in the definition of the product do not need to be used. In other words, (m, c, m) is a symbol in w, and gate c is an input-gate that evaluates to 1.

Suppose that c belongs to a block B which is not an input-block. There are two possibilities depending on whether both children of B are building-blocks or not (by assumption we know that at least one of them is a building-block). Suppose that the first possibility occurs (the second one is treated similarly). In this case, the only way to get element (m, c, m) is to use at some step Product 1 on $(m, a, m)(a, c, b)$ and then Product 2 on $(m, c, b)(m, b, m)$. Then, w can be partially evaluated to $w_1(m, a, m)w_2(a, c, b)w_3(m, b, m)w_4$ where a and b belong respectively to the left and right child of B. By inductive assumption both gates a and b evaluate to 1. Consequently, the AND-gate associated with (a, c, b) evaluates to 1 and so does the OR-gate c. This proves one direction.

Suppose now that C_m contains an accepting subtree, i.e. there exists a consistent selection of one AND-gate and one OR-gate in each building-block and a consistent selection of one accepting input-gate per input-block. We

will prove by induction on the depth of the block-tree that w can be evaluated to (m, c, m), where c is the number of an OR-gate in the output-block that evaluates to 1.

Suppose first that the 'block-depth' of C_n is 2. Then, $w = usv$ where s is the encoding of the output-block, u is the encoding of its left child, and v is the encoding of its right child. Observe that both u and v are input blocks. Suppose that the correct triples to choose in u, s and v are respectively (m, a, m), (a, c, b) and (m, b, m). Since u is a list of triples in reverse order then, multiplying (m, a, m) to the right in u yields (m, a, m). Similarly, multiplying (a, c, b) to the left in s yields (a, c, b). After this partial evaluation of w we get a string of the form $u'(m, a, m)(a, c, b)s'v$ that can be reduced further to $u'(m, c, b)s'v$. Observe that because we use a normal numbering for nodes of C_m, (m, c, b) is larger than any triples in s', and thus the last string can be reduced to $u'(m, c, b)v$. We continue the evaluation by multiplying (m, b, m) to the left in v, and we obtain the string $u'(m, c, b)(m, b, m)v'$ that can be reduced to $u'(m, c, m)v'$. Observing that (m, c, m) is larger than any element in u' and v' we obtain the single triple (m, c, m). This proves the basis of our induction.

For the induction step, let $w = usv$ where s is the output-block, u and v the left and right block-subtrees of s. Let (a, c, b) be the correct triple to choose in s. This means that the gate numbered a (resp. b) in u (resp. v) is the root of an accepting subtree in u (resp. v). Suppose inductively that u (resp. v) can be evaluated to (m, a, m) (resp. (m, b, m)). Thus, w can be partially evaluated to $(m, a, m)s(m, b, m)$. Multiplying (a, c, b) to the left in s gives $(m, a, m)(a, c, b)s'(m, b, m)$ that can be reduced to $(m, c, b)s'(m, b, m)$. Finally, (m, c, b) being larger than any triple in s', we can continue the evaluation to obtain $(m, c, b)(m, b, m) = (m, c, m)$.

We have proved the following theorem.

Theorem 5.4.1 *Any language recognized by a family of tree-like circuits with block-size s and degree d is also recognized by a family of programs of length*

$s(2d-1)$ *over groupoids of order* s^3.

Corollary 5.4.2 *Let* $(G_n)_{n\geq 0}$ *be the family of groupoids constructed above. Any language in (non-uniform)* SAC^1 *is recognized by a family of programs over* $(G_{p(n)})_{n\geq 0}$, *where* $p(n)$ *is polynomial.*

5.4.3 Nondeterministic logarithmic space

In this subsection we will show that there exists a family of groupoids $(D_n)_{n\geq 1}$ whose linear word problem is complete for NL. This family will have the property that for every n, D_n is isomorphic to a subgroupoid of G_n defined in the previous section.

Define D_m as the set

$$D_m = \{(a,b) : 0 \leq a,b \leq m\}.$$

The product of D_m is defined by

1. $(m,a)(a,c) = (m,c)$
2. $(a,c)(e,d) = \begin{cases} (a,c) & \text{if } (e,d,m) \leq (a,c,m) \\ (e,d) & \text{otherwise} \end{cases}$

   for all cases not covered by 1.

It is easy to verify that the mapping $(a,c) \rightarrow (a,c,m)$ is an isomorphism from D_m to a subgroupoid of G_m.

We will show how to reduce the accessibility problem over a directed graph with m nodes to the linear word problem over D_m. Let P be a directed graph of size m. Suppose that the vertices of P are labeled with a number lower or equal to m. Suppose furthermore that any edge (a,b) in P is such that $a < b$. The problem is to determine if there is a path from 1 to m. It is well known that this problem is complete for NL under a logspace reduction (see [25]).

Let $w = (m,1)w_1 \cdots w_{m-1}$ where w_i is a list of the edges emerging from vertex i. All lists are in increasing order.

A simple argument similar to that used in the previous section shows that w can be evaluated to (m,m) if an only if there is a path in P from m to itself.

Moreover, we can restrict the parenthesization to be of right-depth 2 without loss of generality.

The proof of Lemma 2.5.3 shows that any language recognized in constant right-depth by a L-uniform family of programs over polynomial-order groupoids belongs to NL. Since the above program is clearly L-uniform, we have the following result.

Theorem 5.4.3 NL *is equal to the class of languages* RD_2*-recognized by* L*-uniform programs over* $(D_{p(n)})_{n\geq 0}$*, for some polynomial* $p(n)$.

5.4.4 Deterministic logarithmic space

The family of groupoid $(D_n)_{n\geq 1}$ defined in the previous subsection can be used to capture L. In order to do this, we will reduce the 1GAP problem (i.e. the accessibility problem on directed graph of outdegree 1) to the problem of evaluating a string $w \in D_m^*$ from left to right.

Let P be a directed graph with outdegree 1. The conditions on P and the reduction to a word w are identical to what was defined in the previous subsection. The resulting string w is such that each w_i contains at most one vertex. One can verify that there is a path from m to itself if and only if (m, m) is the element obtained by evaluating w from left to right.

Theorem 5.4.4 L *is equal to the class of languages left-to-right recognized by* L*-uniform programs over* $(D_{p(n)})_{n\geq 0}$*, for some polynomial* $p(n)$.

5.4.5 Bounded circuits of logarithmic depth

We will show that any problem in NC^1 is reducible to the problem of evaluating from left to right a word over $D_1 1$.

Let $x = x_1 \cdots x_n$ be a word over the permutation group S_5. We know that the problem of determining if x maps 5 on itself is complete for NC^1.

Assume that the length of x is even. Fix i, and for, $1 \leq j \leq 5$, let y_j be the image of j under the permutation x_i. Represent x_i as the sequence

$(1, 5 + y_1) \cdots (5, 5 + y_5)$, if i is odd, and $(6, y_1) \cdots (10, y_5)$, if i is even, and let $w = (11, 5)x(5, 11) \in D_1 1^*$.

One can verify that x maps 11 on itself if and only if w left-to-right evaluates to $(11, 11)$.

This and Barrington's theorem [5] yield the following result.

Theorem 5.4.5 NC^1 *is equal to the class of languages left-to-right recognized by* DLOGTIME-*uniform programs over* $D_1 1$. □

5.5 Clean circuits

Definition 5.5.1 *A clean circuit is a tree-like circuit that, on any input* $x_1 \cdots x_n$, *has at most one AND-gate and one OR-gate that evaluates to 1 in each building block. Moreover, every input-block contains exactly two input gates looking at the same input position* i. *One of the gate is said to be positive and outputs* x_i, *the other is said to be negative and outputs* $\bar{x}_i$.

Lemma 5.5.2 *For any AND-gate* g *in a clean circuit, either the fan-out of* g *is 1 or this gate never evaluates to 1. Moreover, for any two OR-gates* g_1 *and* g_2, *there is at most one AND-gate that takes its inputs from both* g_1 *and* g_2.

Proof. For the first part, simply observe that if two OR-gates in a block receive input from the same AND-gate which evaluates to 1 on some input, then both OR-gates evaluate to 1, and the circuit is not clean. The second part is a direct consequence of the definition. □

It it known from [81] that P-uniform circuits of exponential size and polynomial degree, and in particular tree-like circuits of exponential block-size and polynomial degree, can be simulated by nondeterministic Turing machine running in polynomial time. Later in this section, we will show that when we further restrict the circuits to be clean, the class of languages recognized corresponds precisely to P. However, to achieve this we must use a stronger

notion of uniformity. This is because, given any gate number a, a nondeterministic machine can, in polynomial time, guess a gate number b and verify that a is effectively a child of b. Nondeterminism seems to be essential here because the number of gates in the circuit is exponential.

Let $(C_n)_{n\geq 0}$ be a family of clean circuits. Consider any numbering of the blocks of C_n, for any integer n. For each circuit C_n, we define the following functions.

1. **root**(i) is true iff i is the number of the output-block of C_n.
2. **leaf**(i) is true iff i is the number of an input-block in C_n.
3. **left**(i) is the number of the left child of the block number i.
4. **right**(i) is the number of the right child of the block number i.
5. **parent**(i) is the number of the parent of the block number i.
6. **pos**(i) is the number of the positive input gate in input block i.
7. **neg**(i) is the number of the negative input gate in input block i.

Given a complexity class C, We define a family $(C_n)_{n\geq 0}$ of clean circuits to be C-uniform if the following conditions are satisfied.

- The direct connection language is in C.
- For any integer n, the problem of computing any of the functions **root**, **leaf**, **left**, **right**, **parent**, **neg**, and **pos** is in C.

Clean circuits are closely related with parenthesized programs over families of groupoids. We begin by showing this in the nonuniform setting.

Proposition 5.5.3 *Any language over the alphabet* $\{0,1\}$ *(linearly) recognized by parenthesized programs of length* $h(n)$ *over groupoids of order* $f(n)$ *is also recognized by clean (skew) circuits of degree* $h(n)$ *and block-size* $f^2(n)+f(n)$.

Proof. Let P_n be a parenthesized program over a groupoid G_n, where n is the length of the input considered by P_n. We associate with each well-parenthesized sub-program of P_n a block in a tree-like circuit. With each single instruction, we associate an input block. More precisely, a single instruction $\langle i, \alpha \rangle$, where $1 \leq i \leq n$ and $\alpha : \{0,1\} \rightarrow G_n$, corresponds to an input block consisting of 2 input gates testing if the element generated by the instruction is $\alpha(0)$ or $\alpha(1)$, respectively. Clearly only one input-gate can be evaluated to 1 in each input block.

Let $D_n = (A_n B_n)$ be a subprogram of P_n, where A_n and B_n are parenthesized programs over G_n. We associate with D_n a building block that contains $f(n)$ OR-gates. Each OR-gate tests whether D_n evaluates to some element of G_n. Such a gate, testing for example if D_n evaluates to $g \in G_n$, has children which are AND-gates testing if A_n evaluates to g_1 and B_n evaluates to g_2, for all $g_1 g_2 = g$. The input of these AND-gates are the appropriate gates of the blocks respectively associated with A_n and B_n.

Since each block in the circuit is uniquely associated with a parenthesized subprogram, and since each subprogram evaluates to a unique element, only one OR-gate and one AND-gate can evaluate to 1 in each building-block

Observe that this construction yields skew circuits whenever the programs are linearly parenthesized. □

Proposition 5.5.4 *Any language recognized by a family of clean circuits of degree $d(n)$ and block-size $s(n)$ is also recognized by a family of parenthesized programs of length $6d(n) - 2$ over groupoids of order $s(n) + 2$.*

Proof. Let C_n be such a circuit. By Lemma 5.5.2, there is at most one AND-gate g that receives its input from any two OR-gates a and b. Moreover, the fan-out of any AND-gate being 1, there is at most one OR-gate c that has g as input. This motivates the definition of a product on $B_n = \{1, \ldots, s(n)\} \cup \{O, I\}$, where O is absorbing and I is the identity, and such that $a \cdot b = c$, if c exists, otherwise $a \cdot b = O$. (a, b, c defined as above).

For each input gate g in C_n, define the instruction $V_g = \langle i_g, f_g \rangle$, where i_g is the position of the bit being considered by g and f_g is the function that yields g whenever gate g evaluates to 1, and I otherwise.

For each input block B with g_1, g_2 as input gates, define the parenthesized subprogram $P_B = (V_{g_1} V_{g_2})$.

The block structure of C_n induces a well parenthesized representation of the input blocks $B_1, \ldots, B_{d(n)}$ in a natural way. This leads to the definition of a parenthesized program P_n whose yield (recall that P_n can be seen as a tree) is $P_{B_1} \cdots P_{B_{d(n)}}$ and that evaluates to the number of an AND-gate in the output block of C_n if and only if C_n accepts its input. □

Theorem 5.5.5 P *is equal to the class of languages recognized by a* P*-uniform family of clean (skew) circuits with exponential block-size and polynomial degree.*

The proof of the above theorem follows from Theorem 5.2.8, Theorem 5.2.10, and the two following propositions.

Proposition 5.5.6 *Any language (linearly) recognized by P-uniform parenthesized polynomial-length programs over groupoids of exponential-order is also recognized by* P*-uniform clean (skew) circuits of polynomial degree and exponential block-size.*

Proof. We must show that if the programs are P-uniform, the construction of Proposition 5.5.3 yields a family of P-uniform circuits. Let $h(n)$ be the length of the programs.

In order to prove this, we must set a numbering of the block of the circuits. First we label each block with a pair of integers (i, j) saying that this block considers the subprogram lying between positions i and j. Then, we choose a numbering that encodes efficiently those labels. An input block is thus labeled with a pair of the form (i, i), and it can be verified if such a pair actually corresponds to an input block by checking if the i^{th} symbol of the program

is an instruction: this can be done in polynomial time. Moreover, the output block is labeled with the pair $(1, h(n))$. Hence, verifying that a pair labels the output block can be done in polynomial time.

Given a pair (i, j), one can determine if there is a well parenthesized subprogram starting at position i and finishing at position j simply by counting and comparing the number of closed an open parenthesis. This can be done in polynomial time. To find the left child of a block labeled (i, j) it suffices to compute the unique $k < j$ such that $(i+1, k)$ corresponds to a well-parenthesized subprogram. This can be done by trying all the possible values for k between $i+1$ and $j-1$. We proceed similarly to find the right child of a block. To find the parent of a block labeled (i, j), it suffices to compute the unique integer k such that either $(k, j+1)$ or $(i-1, k)$ corresponds to a well-parenthesized subprogram. All these computations can be done in polynomial time.

We must also find a numbering of the gates of the circuits. First we label each OR gate with a tuple (a, b, g, OR) meaning that this gate checks if the subprogram located between position a and b yields g. An AND gate is labeled with a tuple $(a, b, g_1, g_2, \text{AND})$, meaning that this gate checks if the subprogram $D_n = (A_n B_n)$, located between positions a and b, evaluates to $g = g_1 g_2$ and receives input from OR gates checking if A_n evaluates to g_1 and B_n evaluates to g_2. A positive (resp. negative) input gate is labeled with a pair (i, g), meaning that the i^{th} symbol of the programs is an instruction (i, v), and $v(1) = g$ (resp. $v(0) = g$). Choosing a numbering that encodes these labels, the direct connection can be recognized in polynomial time. One difficulty occurs when we need to verify that a and b actually bound a well-parenthesized subprogram, but as we already mentioned, this can be determined in polynomial time.

Finally, given the number m of an input block, one can compute its negative and positive gates in polynomial time. Indeed, if m encodes the pair (i, i), then it suffices to compute the i^{th} symbol which is an instruction (j, v). The positive gate has the number that encodes the pair $(i, v(1))$ and the negative gate has the number that encodes $(i, v(0))$. □

Proposition 5.5.7 *Any language recognized by a* P*-uniform family of clean circuits of polynomial degree and exponential block-size is also recognized by a family of polynomial-length parenthesized programs over groupoids of exponential order.*

Proof. As in Proposition 5.5.6, it suffices to show that the construction yields a P-uniform family of programs.

We will however make a small modification in the construction. Recall that each subprogram P_{B_i} contains exactly four symbols. We modify P_n into a program Q_n by replacing each parenthesis by four identical parenthesis. We do this because given a position i in Q_n, the fact that the i^{th} symbol is in some P_{B_i} will be independent of the two weaker bits of i, it will depend only on the number $j = \lfloor i/4 \rfloor$. In other words, all positions beginning by j are in the same subprogram P_{B_i}, are open parenthesis, or are closed parenthesis.

This can be checked in polynomial time using the following algorithm. Consider a numbering of the block of the circuit and let m be the number associated with the output block.

```
x ← m
Repeat
 If leaf(x) then
  write 'subprogram'
  y ← parent(x)
  While x = right(y) do
   write 'closed parenthesis'
   If root(y) then stop
   x ← y
   y ← parent(x)
 Else
  write 'open parenthesis'
  x ← left(x)
```

The above algorithm makes a depth-first search in the tree composed of all blocks of the circuits. It writes a sequence of messages, where the y^{th} message indicates if the y^{th} batch of four symbols in the program are open parenthesis, closed parenthesis, or a subprogram P_{B_i}. Since the degree of the circuits is polynomial and since all calls to any of the functions **root**, **leaf**, **right**, **left**, and **parent** take polynomial time, the total time taken by the algorithm is polynomial.

Now, computing the i^{th} symbol can be done in polynomial time simply by computing $j = \lfloor i/4 \rfloor$ and by checking what is the j^{th} message written by the algorithm: if the message is 'open parenthesis' or 'close parenthesis' then the i^{th} symbol is the appropriate parenthesis, if the message is 'subprogram' then we only have to look at the 2 weaker bits of i to determine if the symbol is an instruction or not. If the symbol is an instruction, then we can find it in polynomial time by computing first the block number associated with the subprogram using the above algorithm. Then, we can compute the number m_1 of the negative gate and the number m_2 of the positive gate in this input block. If the labels of m_1 and m_2 are respectively (i, g_1) and (i, g_2), then the instruction is (i, v), where $v(1) = g_1$ and $v(0) = g_2$. This concludes the proof. □

Theorem 5.5.8 L *is equal to the class of languages recognized by a L-uniform family of clean skew circuits with polynomial block-size and polynomial degree.*

Proof. Observe that the proof of Proposition 5.5.7 and that of Proposition 5.5.6 remains valid in the context of L-uniformity. This is a consequence of the fact that the depth-first search in a rooted undirected tree can be done in logarithmic space in terms of the number of nodes (see [25]). Thus, the result follows form these two propositions and Proposition 5.2.9. □

5.6 The missing class

Clean circuits can thus be viewed as a deterministic version of tree-like (and thus semi-bounded) circuits.

Table 5.1 describes the equivalence between different types of tree-like circuits of polynomial degree and programs of polynomial length, and gives the relation with some important complexity classes.

Class		Circuits	Block-size	Prog	Order
NC^1	=	clean	cst	det	cst
L	=	clean skew	poly	det lin	poly
NL	=	tree-like skew	poly	lin	poly
LOGDCFL	$\supseteq$	clean	poly	det	poly
LOGCFL	=	tree-like	poly	gen	cst or poly
P	=	clean	exp	det lin or det gen	exp
NP	=	tree-like	exp	lin or gen	exp

Table 5.1: Relationship between different complexity classes, polynomial-length programs and polynomial-degree tree-like circuits

It is remarkable that only LOGDCFL is not exactly characterized. One would be tempted to conjecture that the only inequality in Table 5.1 can be replaced by an equality. However, things are not so clear, as the following discussion shows.

Let P_n be a program (for input of length n) over a groupoid G_n. For any input w of length n accepted by P_n, let the depth of w be the depth of the smallest evaluation tree that can be used to accept w. Define the depth of P_n as the maximum between the depth of all w accepted by P_n. We also define the depth of a family of programs $(P_n)_{n\geq 0}$ as a function mapping n to the depth of P_n.

In Section 5.4, we have constructed a family of groupoids over which

polynomial-length programs recognize precisely those languages in SAC^1. We observe that we only need to use trees of logarithmic depth. Hence logarithmic-depth programs over polynomial-order groupoids are as powerful as general programs.

It is not known if this situation remains true for parenthesized programs. As a starting point for an answer, we have the following result.

A PRAM is said to satisfy the OROW condition if for any register there is at most one processor that writes in it and at most one processor that can read in it (see [61]). We denote by OROW-PRAM($d(n)$) the class of languages recognized in $O(d(n))$ steps by a PRAM satisfying the OROW condition.

Proposition 5.6.1 *Any language $L \subseteq A^*$ recognized by* L-*uniform parenthesized programs of depth $d(n)$ over polynomial-order groupoids belongs to* OROW-PRAM($d(n)$).

Proof. Let n be any integer, let P_n be the program recognizing $L \cap A^n$. Suppose that P_n is defined over a groupoid G_n of order $p(n)$. We will construct an OROW-PRAM that works in time $O(d(n))$, uses a polynomial number of processors, and recognizes L.

Observe first that the content of a register can be propagated into a polynomial number of registers in logarithmic time and in a way that satisfies the OROW property. It suffices to use a polynomial number of processors connected together in the manner of a binary tree. We will implicitly use this idea in what follows.

Let us see P_n as a tree. With each node of P_n, we associate a distinct register.

A register associated with a leaf, i.e. an instruction, will hold the element of G_n generated by this instruction. Since this element can be found in logspace and since $L \subseteq$ OROW-PRAM($\log n$) (see [61]), this can also be done by a polynomial number of processors using their own registers, satisfying the OROW condition, and using $O(\log n)$ steps.

Let N be an internal node of P_n, and let N_1 and N_2 be the children of N. The register R associated with N will hold the result of the product of the two elements contained in the registers associated with N_1 and N_2. This is done as follows. We exclusively associate with R a processor P and $p^2(n)$ special registers that hold the multiplication table of G_n. To do so, each special register is assigned with a distinct pair (a, b), where $a, b \leq p(n)$. By assumption the product ab can be done in logspace. This means that $p^2(n)$ sets of polynomial number of processors can be used, while satisfying the OROW condition, to write into these $p^2(n)$ registers the multiplication table of P_n (observe that this can be done in parallel for each node N). Then, processor P simply reads the content of the registers associated with N_1 and N_2, say a and b, and copies the content of the register associated with the pair (a, b) in register R

At this point, the register associated with the root of P_n contains the element $g \in G_n$ to which P_n evaluates on input w. It remains to determine if g belongs to the accepting set. By assumption, this can be done in logspace and so by a polynomial number of processors working in logarithmic time and satisfying the OROW condition.

The time used by the machine is $O(\log n) + O(d(n))$ which is equal to $O(d(n))$. □

Results from [30] and [27] suggest that LOGDCFL could be more powerful than OROW-PRAM($\log n$).(Recall that Dymond and Ruzzo proved that LOGDCFL corresponds precisely to CROW-PRAM($\log n$).) Hence, if LOGDCFL is effectively equal to the class of languages recognized by L-uniform parenthesized programs over polynomial groupoids, then it would be possible that this last class could not be restricted further to logarithmic-depth programs. On the other hand, if any parenthesized programs can be restricted to have logarithmic depth, then the class of languages recognized by this model could be strictly contained in LOGDCFL. We leave this question as an open problem.

Chapter 6

Conclusion

There are many reasons for continuing investigation of finite groupoids. For example, algebra has become one of the most useful mathematical tool in computational complexity (e.g. the polynomial method for proving circuit lower bounds [11] or the algebraic approach for understanding the inner structure of NC^1 [8]). So, it is just natural to try to understand such a fundamental algebraic structure from a complexity point of view. Also, finite groupoids correspond to pushdown automata, as finite semigroups are related to finite automata. Using groupoids for the study of context-free languages could be fruitful. This approach is particularly attractive when we consider the importance of semigroup theory in the study of regular languages (see [28, 56, 46]). Another reason is the very close connection between cellular automata and finite groupoids that has been observed recently. In [55] it is shown how any finite groupoid can be seen as an infinite one-dimensional cellular automaton where each cell changes its state according to its current state and that of its left neighbour. Then, cellular automata having periodic behavior are shown to correspond to particular varieties of finite groupoids. Let us also mention that programs over groupoids are useful computational models that can be used to capture many important complexity classes.

Semigroup theory has been so important for understanding the structure of finite automata that it would be interesting to define a notion of transformation groupoids of pushdown automata that would extend the transformation monoids of finite automata. Observe that a finite groupoid G recognizing a

language L can be minimized, in a unique way, by using the syntactic groupoid of the tree language recognized by G with the same accepting subset.

We have seen that the algebraic structure of finite groupoids is significant. It was known that the syntactic groupoid G_T of a tree language T is unique and divides any groupoid that recognizes T. This has many consequences. For example, a commutative groupoid cannot recognize a tree language whose syntactic groupoid is not itself commutative. As an other example, if the multiplication monoid of G_T is nonsolvable, then no groupoid with a solvable multiplication monoid can recognize T, by Proposition 2.1.6. Moreover, we showed that the structure of the multiplication monoid influences the kind of languages that can be linearly recognized. It appears however that the situation could be different in the context of recognition by programs, where groupoids having aperiodic multiplication monoid seem able to recognize SAC^1 and, in particular, any context-free language [18, 10]

Some classes of groupoids would deserve to be investigated further. This is the case of the Lie groupoids and the one-sided groupoids introduced in chapter 2. In particular, it would be interesting to determine the exact complexity of the word problem over one-sided groupoids. Is it complete for LOGDCFL? It would be useful to find algebraic properties that would permit to recognize only the deterministic context-free languages.

One of the most important contribution of this thesis concerns finite quasigroups and the fact that they are no more powerful than finite semigroups. We have seen that any language recognized or linearly recognized by a finite quasigroup is open. It remains to determine if the converse is also true. Indeed, this depends on whether or not languages recognized by quasigroups are closed under concatenation. Another interesting question is if recognition and linear recognition by finite quasigroups are equivalent.

We said nothing about families of quasigroups. We would very much like to know what class of languages is recognized by polynomial length programs over polynomial order quasigroups. This question, restricted to groups, has not been settled yet. One easily shows that polynomial length programs over

polynomial order groups recognize only languages in L, but we do not know if all languages in L can be recognized by such programs.

It would be important to find many simple examples of finite groupoids whose word problems are complete for complexity classes like L, NL, SAC^1, and LOGDCFL: many, because comparing them together could give some hint towards understanding their structure; simple, because in order to be useful they must be easily analyzable. In chapter 5, we gave a simple family of groupoids $(G_n)_{n\geq 0}$ with which we can capture many complexity classes. This can be used to reformulate some problems in computational complexity. For example, proving that no polynomial length left-to-right programs over $G_1$1 can simulate some polynomial length programs over $(G_n)_{n\geq 0}$ would imply that $NC^1 \neq SAC^1$.

We hope that the definition of clean circuits introduced in this thesis will be useful for future research. In [81], it is shown how circuits of polynomial degree are related to nondeterministic complexity classes. In this sense, clean circuits correspond to deterministic classes. At least, this is true for P and L. However, these results are not completely satisfactory since we have not been able to characterize LOGDCFL in this way. Actually, it seems that clean circuits of polynomial size and polynomial degree would correspond more closely to OROW-PRAM's using a polynomial number of processors and running in polynomial time.

Bibliography

[1] M Ajtai, Σ_1^1-*formulae on finite structures*, Annals of Pure and Applied Logic, 24 pp.1-48, 1983.

[2] A.A. Albert, Quasigroups. I, *Trans. Amer. Math. Soc.*, Vol. 54 (1943) pp.507-519.

[3] A.A. Albert, Quasigroups. II, *Trans. Amer. Math. Soc.*, Vol. 55 (1944) pp.401-419.

[4] J.L. Balcázar, J. Díaz and J. Gabarró, *Structural Complexity I*, Springer-Verlag 1988.

[5] D.A. Barrington, *Bounded-Width Polynomial-Size Branching Programs Recognize Exactly those Languages in* NC^1, *JCSS* **38**, 1 (1989), pp. 150-164.

[6] D. Mix Barrington, N. Immerman and H. Straubing, *On Uniformity within* NC^1, *JCSS* **41**, 3 (1990), pp. 274-306. Also University of Massachusetts COINS Tech. Rep. 1989-88.

[7] D.A. Barrington, H. Straubing and D. Thérien, *Non-Uniform Automata Over Groups*, Information and Computation 89 (1990) pp.109-132.

[8] D. Barrington and D. Thérien, *Finite Monoids and the Fine Structure of* NC^1, *JACM* **35**, 4 (1988), pp. 941-952.

[9] M. Beaudry, Word Problems on quasipolynomial-size Groupoids, Manuscript 1993.

[10] M. Beaudry, Languages recognized by finite aperiodic groupoids, Technical Report 162, Université de Sherbrooke, August 1995.

[11] R. Beigel, *The Polynomial Method in Circuit Complexity*, in Proc. of the 8th annual Structure in Complexity Theory, 1993, pp.82-95.

[12] F. Bédard, F. Lemieux and P.McKenzie, Extensions to Barrington's M-program model, *TCS* **107** (1993), pp. 31-61.

[13] J. Berstel, *Transduction and Context-free languages*, Teubner, 1979.

[14] A. Borodin, *On Relating Time and Space to Size and Depth*, SIAM J. on Computing 6, 4 (1977), pp. 733-744.

[15] R.H. Bruck, Contributions to the Theory of Loops, *Trans. AMS*, (60) 1946 pp.245-354.

[16] R.H. Bruck, *A Survey of Binary Systems*, Springer-Verlag, 1966.

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

[18] H. Caussinus, *Contribution à l'étude du non-déterminisme restreint*, Ph.D. Thesis, Université de Montréal, 1996.

[19] H. Caussinus and F. Lemieux, *The Complexity of Computing over Quasigroups*, In the Proceedings of the 14th annual FST&TCS Conference, pp.36-47, 1994.

[20] A.K. Chandra, L.J. Kozen and L.Stockmeyer, *Alternation*, J. of the ACM **28** pp.114-133, 1981.

[21] A. Chandra, L. Stockmeyer and U. Vishkin, *Constant Depth Reducibility*, SIAM J. on Computing **13**, 2 (1984), pp. 423-439.

[22] O. Chein, H.O. Pfugfelder, and J.D.H. Smith, *Quasigroups and Loops: Theory and Applications*, Helderman Verlag Berlin, 1990.

[23] S.A. Cook, *Characterization of pushdown machines in terms of time-bounded computers*, *JACM* 18, 1 (1971), pp. 4-18.

[24] S.A. Cook, *A Taxonomy of Problems with Fast Parallel Algorithms*, *Information and Computation* 64 (1985), pp. 2-22.

[25] S.A. Cook and P. McKenzie, *Problems Complete for Deterministic Logarithmic Space*, J. of Algorithms 8 (1987), pp. 385-394.

[26] J. Dénes and A.D. Keedwell, *Latin Squares and their Applications*, English University Press, 1974.

[27] P.W. Dymond and W.L. Ruzzo, *Parallel RAMs with Owned Global Memory and Deterministic Context-Free Languages Recognition*, Proc. 13th International Colloquium on Automata, Languages and programming, 1986, pp.95-104.

[28] S. Eilenberg, *Automata, Languages and Machines*, Academic Press, Vol. B, (1976).

[29] T. Evans, *Embedding Incomplete Latin Squares*, Amer. Math. Monthly, 67 pp.958-961, 1960.

[30] F.E. Fich and A. Wigderson, *Toward Understanding Exclusive Read*, SIAM J. Comput., 19 4, 1990, pp.718-727.

[31] M.L. Furst, J.B. Saxe and M. Sipser, *Parity, Circuits, and the Polynomial-Time Hierarchy*, *Proc. of the 22nd IEEE Symp. on the Foundations of Computer Science* (1981), pp. 260-270. Journal version Math. Systems Theory 17 (1984), pp. 13-27.

[32] Z. Galil, *Two way deterministic pushdown automaton languages and some open problems in the theory of computation*, Proc. 15th IEEE Symp. Switching and Automata Theory, pp.170-177, 1974.

[33] M.R. Garey and D.S. Johson, *Computers and Intractability: A Guide to the* NP-*Completeness*, Freeman, 1979.

[34] F. Gecseg and M. Steinby, *Tree Automata*, Akademiai Kiado, Budapest, 1986.

[35] S. Greibach, *The Hardest Context-Free Language*, SIAM J. on Computing 2, 4 (1973), pp. 304-310.

[36] P. Hall, *The theory of groups*, Macmillan, 1959

[37] J. Hartmanis, *On the Non-Determinancy in Simple Computing Devices*, Acta Informatica 1,pp.336-344, 1972.

[38] J. Hartmanis, *Gödel, von Neumann and the* P =?NP *problem*, Bulletin of the EATCS, (38) pp.101-107, 1989.

[39] A. Hajnal, W. Maass, P. Pudlák, M. Szegedy, and G.Turán, *Threshold circuits of bounded depth*, In FOCS 1987 pp.99-110.

[40] M.A. Harrison, *Introduction to Formal Langage Theory*, Addison-Wesley (1978).

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

[42] N. Immerman and S.Landau, *The Complexity of Iterated Multiplication*, Proc. of the 4th Annual Conf. on the Structure in Complexity Theory, IEEE Computer Society Press (1989), pp. 104-111.

[43] E. Jurvanen, A. Potthoff and W. Thomas, *Tree Languages Recognizable by Regular Frontier Check*, Technical Report #9311, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, June 1993.

[44] K.M. Kapp, *Green's Lemma for Groupoids*, Rocky Mountain Journal of Mathematics, Vol.1, No.3, Summer 1971.

[45] R. Ladner, *The circuit value problem is logspace complete for* P, SIGACT News, 7 pp.18-20, 1975.

[46] G. Lallement, *Semigroups and combinatorial applications*, Wiley, 1979.

[47] C.Y. Lee, *Representation of switching functions by binary decision programs*, Bell Systems Tech. J., 38 pp.985-999, 1959.

[48] F. Lemieux, *Complexité, langages hors-contextes et structures algebriques non-associatives*, Masters Thesis, Université de Montréal, 1990.

[49] R. Lipton and Y. Zalcstein, *Word Problems Solvable in Logspace*, J. ACM 24, 3 (1977), pp.522-526.

[50] P. McKenzie, Personnal communication.

[51] W.D. Maurer and J. Rhodes, A Property of Finite Simple Non-abelian Groups, *Proc. AMS* 16 (1965), 552-554.

[52] G.L. Miller, On the $n^{\log n}$ Isomorphic Technique, *Proc. of the 10th ACM Symp. on the Theory of Computing* (1978), pp. 51-58.

[53] A. Muscholl, Characterizations of LOG, LOGDCFL and NP based on groupoid programs, Manuscript, 1992.

[54] H.O. Pfugfelder, *Quasigroups and Loops: Introduction*, Heldermann Verlag, 1990.

[55] J. Pedersen, *Cellular Automata as Algebraic systems*, Complex System 6 (1992) pp.237-250.

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

[57] J.-E. Pin, *Finite Group Topology and p-adic Topologic for Free Monoids.* ICALP 1985.

[58] J.-E. Pin, *BG = PG: A Success Story*, Proc. of Intern. Conf. on Groups, Semigroups, and Formal Languages, York 1993, Kluwer Publisher.

[59] N. Pippenger and M. J. Fischer, *Relations Among Complexity Measures*, *JACM* 26, 2 (1979), pp. 361-381.

[60] A.L. Rosenberg, *A Machine Realization of the Lin ar Context-Free Languages*, *Information and Computation* 10 (1967), pp. 175-188.

[61] P. Rossmanith, *The Owner Concepts for PRAMs*, Proc. 8th Annual Symp. on Theoretical Aspects of Computer Science, 1991.

[62] W. L. Ruzzo, *Tree-Size Bounded Alternation*, J. Computer and Systems Science 21, (1980) pp. 218-235.

[63] W. Ruzzo, *On uniform circuit complexity, JCSS* 22, 3 (1981), pp. 365-383.

[64] J.E. Savage, *The complexity of computing*, Wiley, 1976.

[65] M.P. Schützenberger, On Finite Monoids having only trivial subgroups, *Information and Control* 8 (1965), pp. 190-194.

[66] I. Simon, *Piecewise testable Events*, Proc. 2nd GI Conf., LNCS 33, Springer, pp. 214-222, 1975.

[67] C.E. Shannon, *A Symbolic analysis of relay and switching circuits*, Trans. AIEE 57, pp.713-723, 1938.

[68] C.E. Shannon, *The synthesis of two-terminal switching circuits*, Bell Systems Technical Journal 28, pp.59-98, 1949.

[69] R. Smolensky, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, In Proc. 19th Ann. ACM Symp. Theor. Comput., pp77-82, 1987.

[Su75] I.H. Sudburough, *A Note on Tape-Bounded Complexity Classes and Linear Context-Free Langages*, *JACM* 22, 4 (1975), pp. 499-500.

[70] P.M. Spira, *On time hardware complexity tradeofs for Boolean functions*, Proceeding of the 4th Hawaii International Symposium on System Sciences, pp.525-527, 1971.

[71] M. Steinby, *A Theory of Tree Languages Varieties*, in Tree Automata and Languages (ed. M. Nivat and A Podelski), Elsevier, 1992.

[72] H. Straubing, Representing Functions by Words over Finite Semigroups, Université de Montréal, Technical Report #838, 1992.

[73] H. Straubing, *Finite Automata, Formal logic, and Circuit Complexity*, Birkhäuser, 1994.

[74] L.J. Stockmeyer and U. Vishkin, *Simulation of parallel random access machines by circuits*, SIAM J. Compt. Vol.13 (2), 1984.

[75] I. Sudburough, *On the Tape Complexity of Deterministic Context-Free Languages*, *JACM* **25**, 3 (1978), pp. 405-414.

[76] D. Thérien, Classification of Finite Monoids: The Language Approach, *TCS* **14** (1981), pp. 195-208.

[77] D. Thérien, Subword Counting and Nilpotent Groups, in Combinatorics on Words: Progress and perspectives, Academic Press, 1983.

[78] W. Thomas, *Logical aspects in the study of tree languages*, 9th Coll. on trees in algebra and in programming, Cambridge University Press, pp. 270-280, 1984.

[79] L.G. Valiant, *Reducibility by Algebraic Projection*, L'enseignement Mathématique, **28**, 3-4 (1982), pp.253-268 *JCSS* **10**, 2 (1975), pp. 308-315.

[80] H. Venkateswaran, *Properties that Characterize* LOGCFL, *Proc. of the 19th ACM Symp. on the Theory of Computing* (1987), pp. 141-150.

[81] H. Venkateswaran, *Circuit definitions of nondeterministic complexity classes*, Proceedings of the 8th annual FST&TCS Conference, 1988.

[82] M.J. Wolf, Nondeterministic Circuits, Space Complexity and Quasigroups, *TCS* **125** (1994), pp. 295-314.