UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

ÉQUIVALENCE DES GRAMMAIRES DE FONCTION SIMPLE

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

CÉDRIC BASTIEN

DÉCEMBRE 2006

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce mémoire intitulé :

ÉQUIVALENCE DES GRAMMAIRES DE FONCTION SIMPLE

présenté par
Cédric Bastien

pour l'obtention du grade de maître ès science (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Dr. Jurek Czyzowicz .... Directeur de recherche
Dr. Wojciech Fraczak .... Codirecteur de recherche
Dr. Kamel Adi .... Président du jury
Dr. Andrzej Pelc .... Membre du jury

Mémoire accepté le : 1 décembre 2006

Remerciements

Merci à Jurek Czyzowicz pour m'avoir transmis le désir d'entreprendre des études supérieures et m'avoir donné le goût à la recherche.

Merci à Wojciech Fraczak pour m'avoir encouragé et aidé tout au long de la préparation et de la rédaction de ce mémoire.

Merci à Didier Caucal pour s'être généreusement offert de réviser ce travail et pour ses nombreux conseils.

Merci à Feliks Welfeld et à toute l'équipe de IDT Canada Inc. pour m'avoir fourni l'équipement et le support nécessaires à la mise en oeuvre de la partie expérimentale de ce travail.

Merci au CRSNG pour avoir financé ce travail de recherche.

Enfin, merci à ma famille et à mes amis pour les nombreux sourires qu'ils m'apportent jour après jour.

Table des matières

Remerciements **i**

Liste des figures **v**

Liste des tableaux **vi**

Résumé **vii**

1 Introduction **1**

1.1 Classification de paquets sur les réseaux 1

1.2 Problème de l'équivalence des grammaires de fonction simple 3

1.3 Résultats 4

2 Revue de la littérature **6**

2.1 Langages et le problème de la décidabilité 6

2.1.1 Langages simples 7

2.1.2 Autres classes de langages 8

2.2 Transductions 8

2.2.1 Fonctions simples 9

2.2.2 Autres classes de transductions 9

3 Terminologie et notions élémentaires **11**

3.1 Symboles, mots et langages 11

3.2 Opérations sur les mots et langages 12

4 Équivalence des grammaires simples **13**

4.1 Grammaires simples 13

4.2 Quotient d'un langage simple 15

4.3 Fonction de décomposition 18

4.4 Comparaison de chaînes compressées 20

4.5 Algorithme d'équivalence des grammaires simples 21

4.5.1 Preuve d'exactitude et complexité 22

4.6 Exemples d'exécution de ÉQUIVALENCEGS 26

5 Équivalence des grammaires de fonction simple **29**

5.1 Grammaires de fonction simple 29

5.2 Groupe libre sur l'alphabet Ω 31

5.3 Fonction de décomposition 33

5.4 Relation de conjugaison 34

5.5 Quotient d'une fonction simple 38

5.6 Procédure `extraire_equation` 39

5.6.1 Preuves de terminaison et d'exactitude 42

5.7 Algorithme d'équivalence des grammaires de fonction simple 45

5.7.1 Relation de type *self-proving* 45

5.7.2 Description de l'algorithme ÉQUIVALENCEGFS 46

5.7.3 Preuve d'exactitude et complexité 47

5.8 Exemple d'exécution de ÉQUIVALENCEGFS 50

6 Implémentation des algorithmes d'équivalence des grammaires simples 53

6.1 Description des algorithmes implémentés 53

6.2 Méthodologie 54

6.3 Résultats 56

7 Conclusion 59

Bibliographie 61

Liste des figures

1.1 Illustration d'un classificateur utilisé pour filtrer l'accès à un réseau. . . . 2

1.2 Illustration d'un classificateur utilisé pour implémenter une fonction de routage. 3

4.1 Arbre de dérivation (Exemple 4.1) 15

4.2 Arbre de dérivation d'un mot le plus court de $L_G(A)$ (Exemple 4.2) . . . 17

4.3 Algorithme ÉquivalenceGS. 23

4.4 Exemple d'exécution de ÉquivalenceGS démontrant $L_G(V) = L_G(Z)$. 27

4.5 Exemple d'exécution de ÉquivalenceGS démontrant $L_G(V) \neq L_G(Z)$. 28

5.1 Procédure `extraire_equation`. 41

5.2 Algorithme ÉquivalenceGFS. 48

5.3 Exemple d'exécution de ÉquivalenceGFS démontrant $F_G(S) = F_G(T)$. 51

6.1 Résultats de l'exécution des algorithmes d'équivalence des grammaires simples. 57

6.2 Temps d'exécution de l'algorithme alg3 sur des grammaires simples générées aléatoirement. 58

Liste des tableaux

6.1 Caractéristiques des trois catégories de grammaires simples du jeu d'essais. 56

Résumé

Une *grammaire simple* est une grammaire LL(1) en forme normale de Greibach. Si l'alphabet d'une grammaire simple est l'union de deux alphabets, un alphabet d'entrée Σ et un alphabet de sortie Ω, la grammaire est une grammaire de fonction simple. Une *fonction simple* est une application partielle $F : \Sigma^* \to \Omega^*$ définie à partir d'un symbole non-terminal d'une grammaire de fonction simple. Étant donnés deux non-terminaux A et B d'une grammaire de fonction simple, le *problème d'équivalence des grammaires de fonction simple* consiste à déterminer si les fonctions simples définies par A et B sont les mêmes.

Les fonctions simples sont utilisées dans une technologie de classification de paquets sur les réseaux pour représenter efficacement les règles de classification. Comme la taille des règles peut être considérable, il importe de pouvoir identifier les facteurs qui sont communs à plusieurs règles afin d'éviter de les dupliquer inutilement. Cette tâche nécessite l'utilisation d'un algorithme d'équivalence des grammaires de fonction simple.

Nous présentons dans ce mémoire un algorithme permettant de résoudre efficacement le problème d'équivalence des grammaires de fonction simple. Cet algorithme fonctionne en temps polynomial par rapport à n, la taille de la description de la grammaire d'entrée, et $||G||$, la longueur maximale d'un mot le plus court pouvant être engendré par un non-terminal de la grammaire. C'est le premier algorithme permettant de résoudre ce problème. Nous présentons également un algorithme permettant de résoudre le problème d'équivalence des grammaires simples. Cet algorithme fonctionne en temps $O(n^7 \log^2 n)$, où n est la taille de la description de la grammaire d'entrée, améliorant la borne supérieure précédente qui était $O(n^{13})$. Nous présentons finalement une étude comparant les performances pratiques de notre algorithme d'équivalence des grammaires simples avec celles des deux principaux algorithmes déjà existants pour résoudre ce problème.

Abstract

A *simple grammar* is a LL(1) grammar in Greibach normal form. If the alphabet of a simple grammar is the union of two alphabets, an input alphabet Σ and an output alphabet Ω, the grammar is a *simple function grammar*. A *simple function* is a partial mapping $F : \Sigma^* \to \Omega^*$ defined from a non-terminal of a simple function grammar. Given two non-terminals A and B of a simple function grammar, the *simple function grammar equivalence problem* consists in deciding whether the simple functions defined by A and B are equal.

Simple functions are used in a network packet classification technology to efficiently implement the classification rules. In order to manage large sets of those classification rules in memory, it is useful to be able to identify the common factors shared by multiple rules, in order to avoid their duplication in the system. This task is done using a simple function grammar equivalence algorithm.

We present in this work an efficient algorithm solving the simple function grammar equivalence problem. The algorithm works in time polynomial with respect to n, the size of the description of the grammar, and $||G||$, the maximal length of a shortest word generated by a non-terminal of the grammar. It is the first algorithm solving this problem. We also present an algorithm solving the simple grammar equivalence problem. This algorithm works in time $O(n^7 \cdot \log^2 n)$, where n is the size of the description of the grammar, improving on the previous upper bound which was $O(n^{13})$. Finally, we present the results of an experimental study comparing the practical performances of our simple grammar equivalence algorithm with those of the two best-known algorithms for this problem.

Chapitre 1

Introduction

1.1 Classification de paquets sur les réseaux

Les technologies de l'information font partie intégrante de l'informatique moderne. Par le biais des réseaux, une grande quantité d'information peut être rapidement transmise entre les systèmes informatiques. Les protocoles de communication généralement utilisés pour transférer l'information d'un système à un autre permettent la présence d'agents intermédiaires qui peuvent effectuer certaines tâches sur les paquets qui sont échangés. Ces tâches peuvent aller du routage des paquets qui circulent, au filtrage de l'information qu'ils contiennent, jusqu'à l'extraction de statistiques basées sur le contenu de ceux-ci. Compte tenu de la quantité importante d'information qui peut circuler sur les réseaux et de la rapidité avec laquelle l'information y est transmise, il importe que les technologies exécutant ces tâches soient très efficaces.

Les *machines à états de concaténation* (MEC) ont été introduites récemment [8] dans le but d'apporter une solution pratique à ce problème. Une MEC permet d'effectuer une lecture linéaire d'un flux d'information et, pour chaque paquet rencontré, d'émettre sur le champs une valeur particulière correspondant au résultat du travail effectué sur ce paquet par la machine. Plus concrètement, les MEC sont des machines à états constituées de trois types d'états : des états *standards*[1] à partir desquels un symbole lu en entrée détermine le prochain état à atteindre, des états *finaux* auxquels sont associés des suites de symboles de sortie et des *états de concaténation*, représentant la concaténation de

[1]Ces états sont appelés *switch states*, en anglais.

deux états standards ou finaux e_1 et e_2 de la machine. Lorsqu'une MEC atteint un état de concaténation, l'exécution se poursuit à partir de l'état e_1 et l'état e_2 est placé sur une pile. Lorsque la machine atteint un état final, la machine produit en sortie la suite de symboles associée à cet état, puis l'exécution se poursuit à partir de l'état situé sur le dessus de la pile et celui-ci est enlevé de la pile. L'exécution se termine lorsque la machine atteint un état final et que la pile est vide. La concaténation des séquences de symboles produites en sortie durant l'exécution constitue le mot de sortie associé au mot reçu en entrée.

Puisque pour une machine donnée, il faut un temps constant pour changer d'état suite à la lecture d'un symbole d'entrée, le temps nécessaire à la transformation d'un mot d'entrée en un mot de sortie dépend directement de la longueur des mots manipulés. Lorsqu'elles sont implémentées au niveau matériel, les MEC permettent d'effectuer certaines tâches sur les réseaux de manière très efficace. Les MEC possèdent aussi d'autres avantages. Elles sont en mesure de représenter en forme compressée certains automates finis. Elles sont également en mesure de vérifier certaines propriétés sur les mots d'entrée qui ne peuvent être vérifiées par un automate fini. Par exemple, une MEC peut vérifier si les parenthèses d'une expression sont correctement imbriquées, ce qu'un automate fini n'est pas en mesure de faire.

Les machines à états de concaténation sont utilisées à IDT Canada Inc. dans une technologie de filtrage et de classification de paquets sur les réseaux implémentée au niveau matériel. La tâche la plus simple pouvant être effectuée par ces classificateurs est de déterminer, pour chaque paquet reçu en entrée, si le paquet doit être accepté ou rejeté. Par exemple, un classificateur de ce type peut être utilisé à l'entrée d'un réseau pour analyser le contenu des paquets TCP/IP reçus et ne donner accès au réseau qu'à ceux pour lesquels l'adresse IP de la source appartient à une classe d'adresses autorisées (figure 1.1).

Fig. 1.1 – Illustration d'un classificateur utilisé pour filtrer l'accès à un réseau.

Une tâche plus générale pouvant également être accomplie par les classificateurs est de produire en sortie une valeur particulière qui est entièrement déterminée à partir de la valeur des champs du paquet reçu en entrée. Un classificateur de ce type peut être utilisé par exemple pour implémenter une fonction de routage. Dans ce cas, la valeur produite en sortie par le classificateur est l'adresse où doit être transféré le paquet reçu en entrée (figure 1.2).

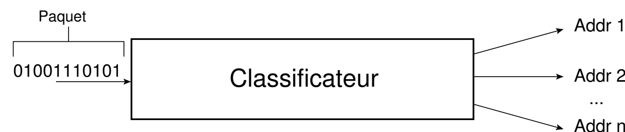

Fig. 1.2 – Illustration d'un classificateur utilisé pour implémenter une fonction de routage.

Dans la technologie de IDT Canada Inc., les règles de classification sont décrites formellement par ce que l'on appelle des *grammaires de fonction simple*. Les grammaires de fonction simple sont équivalentes aux machines à états de concaténation en ce sens que les relations entre les mots d'entrée et les mots de sortie pouvant être implémentées par une MEC sont les mêmes que celles pouvant être décrites par une grammaire de fonction simple. Ces relations sont appelées des *fonctions simples*.

1.2 Problème de l'équivalence des grammaires de fonction simple

En pratique, une MEC peut implémenter une grande quantité de règles de classification et celles-ci sont souvent ajoutées au système de manière incrémentale. Or, il peut arriver que différentes règles du système possèdent certaines parties communes. En effet, si plusieurs règles s'appliquent à des paquets possédant un même format, il est probable que certaines de ces règles définissent un comportement identique pour certain champs de ces paquets. Afin de limiter l'espace mémoire utilisé par une MEC pour représenter l'ensemble des règles du système, nous souhaitons que ces sections ne soient implémentées qu'une seule fois par la machine. Une manière d'identifier les sections qui sont communes

à plusieurs règles est de décomposer les règles en sous-règles primitives, puis de comparer ces sous-règles au moyen d'un algorithme d'*équivalence des grammaires de fonction simple*.

Le problème d'équivalence des grammaires de fonction simple consiste à déterminer, pour deux grammaires de fonction simple données, si les fonctions simples décrites par les deux grammaires sont identiques. Un algorithme permettant de résoudre ce problème doit donc exécuter deux tâches : d'une part, il doit vérifier que les fonctions décrites par les deux grammaires ont le même domaine et d'autre part, il doit vérifier, pour chaque mot appartenant au domaine des fonctions, que les mots produits en sortie par les deux grammaires sont les mêmes.

La première partie de ce travail est consacrée à résoudre la première de ces deux tâches. Comme les valeurs de sortie sont ignorées à cette étape, nous pouvons considérer que les grammaires de fonction simple n'associent aucun mot de sortie aux mots appartenant au domaine. Les grammaires ne décrivant que le domaine des fonctions simples sont appelées des *grammaires simples*. Le problème considéré dans la première partie du travail est donc celui de l'*équivalence des grammaires simples*. La seconde partie du travail est consacrée au problème de l'équivalence des grammaires de fonction simple, dans son ensemble.

1.3 Résultats

Nous présentons dans un premier temps un algorithme permettant de résoudre le problème d'équivalence des grammaires simples en temps $O(n^7 log^2 n)$, où n est la taille de la représentation textuelle de la grammaire, améliorant la borne supérieure précédente qui était de $O(n^{13})$ (c.f. [12]).

Nous présentons également un algorithme solutionnant le problème d'équivalence des grammaires de fonction simple. Cet algorithme s'exécute en temps polynomial par rapport à n, la taille de la représentation textuelle de la grammaire, et $||G||$, la longueur maximale d'un mot le plus court engendré par un symbole non-terminal de la grammaire, et en temps exponentiel si l'on ne considère que la taille de la grammaire. C'est le premier algorithme permettant de résoudre ce problème. Cet algorithme est une extension et une généralisation de l'algorithme d'équivalence des grammaires simples présenté dans [5].

La clé de notre algorithme pour résoudre le problème d'équivalence des grammaires de fonction simple est l'utilisation d'un groupe libre défini sur les symboles de l'alphabet de sortie et d'une propriété des équations de conjugaison dans les groupes libres.

Nous présentons finalement une étude comparant les performances pratiques de notre algorithme d'équivalence des grammaires simples avec celles des deux principaux algorithmes déjà existant pour résoudre ce problème, c.-à-d. [5] et [12]. Les résultats de cette expérimentation démontrent que, dans le contexte pratique considéré, notre algorithme ainsi que celui présenté dans [5] peuvent tous les deux être utilisés pour résoudre le problème d'équivalence des grammaires simples tandis que l'algorithme de [12] est inefficace pour résoudre ce problème dans le même contexte.

Chapitre 2

Revue de la littérature

2.1 Langages et le problème de la décidabilité

Les langages et les relations sur les langages, appelées *transductions*, sont étudiés depuis de nombreuses années. L'une des notions théoriques qui a été le plus souvent considérée par les chercheurs dans ce domaine est l'aspect de la décidabilité d'un problème. Un problème est dit *décidable* s'il peut être résolu par un algorithme s'exécutant sur une machine de Turing. Puisque la machine de Turing est acceptée par la communauté scientifique en tant que modèle d'ordinateur, la décidabilité est considérée comme un moyen déterminant ce qu'il est possible de réaliser par les ordinateurs contemporains. Une question indécidable célèbre est connue sous le nom de *problème d'arrêt* : étant donné un programme, déterminer s'il existe un ensemble de données pour lequel ce programme ne s'arrête pas. Le fait que ce problème soit indécidable a des conséquences importantes car elle rend impossible la construction de systèmes de diagnostiques.

Plusieurs des problèmes indécidables que l'on connaît concernent le domaine de la théorie des langages. Par exemple, nous savons que le problème d'équivalence des *langages hors-contextes* est indécidable. Ce même problème est décidable, cependant, pour certaines sous-classes des langages hors-contextes. C'est pourquoi de nombreux travaux au sujet de ces classes de langages ont été effectués par les chercheurs et les résultats obtenus ont permis de développer une variété d'applications. Par exemple, la classe des *visibly pushdown languages*, qui a été récemment définie par R. Alur et P. Madhusudan [1] s'avère être particulièrement utile à décrire et traiter des fichiers XML [23], tandis

que les *weighted transducers* [20] trouvent des applications au niveau de technologies de reconnaissance et de traitement de la voix [22]. De même, la classe des langages simples, que nous considérons dans ce mémoire, à été initialement étudiée par les chercheurs du point de vue de la décidabilité de son problème d'équivalence. Comme cette classe de langages possède des propriétés qui sont intéressantes d'un point de vue pratique, les scientifiques ont par la suite cherché des algorithmes efficaces solutionnant le problème d'équivalence pour cette classe.

2.1.1 Langages simples

Les langages simples ont été introduits par A. J. Korenjak et J. E. Hopcroft [17] comme la classe des langages pouvant être engendrés par une *grammaire simple*, ou de manière équivalente, comme les langages pouvant être acceptés par les *automates à piles déterministes à 1 état*. Ces auteurs présentent dans [17] un algorithme d'équivalence des grammaires simples dont la complexité est doublement exponentielle par rapport à la taille de la description de la grammaire d'entrée. Ce résultat a permis de démontrer que le problème d'équivalence est décidable pour la classe des langages simples.

Cet algorithme a été repris par plusieurs auteurs, notamment D. Wood [32], M. A. Harrison [11] et B. Courcelle [6], sans toutefois qu'une amélioration ne soit apportée à la borne supérieure sur la complexité de l'algorithme. D. Caucal [5] a lui aussi repris cette méthode en proposant un algorithme similaire mais dont la complexité est $O(n^3 \cdot ||G||)$, où n est la taille de la description de la grammaire et $||G||$ est la longueur maximale d'un mot le plus court pouvant être engendré par un non-terminal de la grammaire. Ainsi, l'algorithme est polynomial par rapport à n et $||G||$, mais est de complexité exponentielle si l'on ne considère que le paramètre n, car $||G||$ peut être exponentiel par rapport à n.

Y. Hirshfeld, M. Jerrum et F. Moller [12] ont ensuite proposé un algorithme qui résout le problème de l'équivalence des grammaires simples en temps $O(n^{13})$. C'est le premier algorithme à s'exécuter en temps polynomial par rapport à n seulement. Les auteurs utilisent des techniques élaborées de compression de chaînes de caractères et de recherche sur des chaînes compressées pour obtenir un algorithme de complexité polynomiale.

L'algorithme que nous présentons dans ce travail (voir également [3, 4]) pour le problème d'équivalence des grammaires simples est une extension des algorithmes présentés dans [5] et [12]. Notre algorithme s'exécute en temps $O(n^7 \log^2 n)$. Récemment, la borne

supérieure sur le temps nécessaire pour résoudre ce problème à été réduite à $O(n^6 \log^2 n)$ par S. Lasota et W. Rytter [18]. Ces auteurs décrivent dans leur travail un algorithme permettant de résoudre en temps $O(n^5 \log^2 n)$ le problème consistant à trouver la première paire de non-terminaux qui diffèrent entre deux chaines de caractères représentées de manière compressées, améliorant la borne supérieure précédente pour ce problème qui était $O(n^6 \log^2 n)$. Comme ce problème est au coeur dans notre solution au problème d'équivalence des grammaires simples, la complexité de notre algorithme s'en trouve également réduite.

Il est intéressant de noter, par ailleurs, que le problème consistant à déterminer si un langage simple est inclus dans un autre langage simple est indécidable, comme l'a démontré E. P. Friedman [10].

2.1.2 Autres classes de langages

Plusieurs résultats concernant le problème d'équivalence pour d'autres classes de langages sont également connus. L'un des plus importants concerne la classe des langages pouvant être reconnus par un automate à pile déterministe. Le problème de la décidabilité de l'équivalence pour cette classe de langages est resté ouvert pendant plus de trente ans, avant que G. Sénizergues [25] parvienne à démontrer, en 1997, que le problème est décidable. Depuis, des preuves alternatives de décidabilité de ce problème ont été présentées, notamment par C. Stirling [27, 28].

Des résultats concernant le problème d'équivalence pour d'autres classes d'automates peuvent aussi être trouvés, par exemple, dans [21, 24, 29].

2.2 Transductions

Le problème d'équivalence a aussi été étudié pour différentes classes de transductions. Comme pour les langages, l'aspect de la décidabilité du problème d'équivalence pour les transductions est très important d'un point de vue pratique. Le problème d'équivalence est donc l'un des problèmes les plus souvent étudiés au sujet des transductions et des algorithmes d'équivalence efficaces sont recherchés lorsque ce problème est décidable pour une classe particulière.

2.2.1 Fonctions simples

Nous sommes intéressés dans ce mémoire par une classe de transductions appelées *fonctions simples*. Les fonctions simples peuvent être vue comme une extension des langages simples et des transducteurs finis déterministes. Ce sont les transductions pouvant être décrites au moyen d'une machine à états de concaténation ou, de manière équivalente, à l'aide d'une grammaire de fonction simple. Ces concepts ont été formellement introduits par W. Debski et W. Fraczak [8], comme modèle mathématique décrivant une technologie de filtrage et de classification de paquets sur les réseaux appelée PAX.port et développée à IDT Canada Inc. D'un point de vue plus classique, les fonctions simples correspondent aux transductions pouvant être décrites par les *transducteurs à pile déterministes à 1 état*. Cette classe de transducteurs n'a jamais été étudiée directement auparavant et aucun algorithme n'est connu pour résoudre le problème d'équivalence pour celle-ci. L'algorithme que nous présentons dans ce mémoire (voir aussi [2]) est donc le premier à résoudre directement ce problème. Certains travaux existent cependant au sujet de classes de transducteurs possédant certaines propriétés en commun avec les fonctions simples.

2.2.2 Autres classes de transductions

Nous savons d'après les résultats de G. Sénizergues [26] que le problème d'équivalence est décidable pour la classe des transducteurs à pile déterministes. Comme cette classe de transducteurs est strictement plus large que celle des transducteurs à pile déterministes à 1 état et qu'il est possible de transformer toute grammaire de fonction simple en un transducteur à pile déterministe à 1 état décrivant la même fonction, nous savons que le problème d'équivalence des grammaires de fonction simple est décidable. Le résultat de G. Sénizergues consiste à appliquer en parallèle deux méthodes de demi-décision, permettant de détecter respectivement l'équivalence et la non-équivalence des transducteurs d'entrée. Il est cependant impossible d'établir une borne supérieure sur le temps d'exécution d'une telle solution.

Un algorithme a été présenté par E. Tomita et K. Seino [30] pour résoudre le problème d'équivalence des transducteurs à pile déterministes de type temps-réel acceptant par la pile vide. On dit qu'un transducteur est de type *temps-réel* s'il lit un symbole

d'entrée à chaque étape de son exécution. Or, la classe des transducteurs correspondant aux MEC ne sont pas de type temps-réel. L'algorithme de E. Tomita et K. Seino ne s'applique donc pas au problème d'équivalence des grammaires de fonction simple. Bien que cet algorithme soit de complexité doublement exponentielle par rapport à la taille des transducteurs d'entrée, il demeure le meilleur algorithme que l'on connaisse pour résoudre le problème d'équivalence pour une classe de transducteurs possédant des propriétés similaires aux transducteurs décrivant les fonctions simples.

D'autres chercheurs se sont intéressés au problème d'équivalence des transducteurs, par exemple pour certaines classes de transducteurs à pile déterministes [15] et des classes de transducteurs finis [9, 14, 31].

Chapitre 3

Terminologie et notions élémentaires

3.1 Symboles, mots et langages

Nous présentons tout d'abord les principales notions qui seront nécessaires à la définition formelle du problème d'équivalence des grammaires simples et des grammaires de fonction simple et des algorithmes solutionnant ces problèmes. Les notations et la terminologie utilisées sont inspirées de celles que l'on retrouve dans le livre de J. E. Hopcroft et J. D. Ullman [13].

L'unité élémentaire atomique de la théorie des langages formels est le *symbole*, généralement représenté par des lettres minuscules du début de l'alphabet (ex : a, b, c, ...). Les symboles peuvent être regroupés en ensembles finis, appelés *alphabets*, qui sont représentés par des lettres grecques majuscules.

Exemple 3.1. $\Sigma = \{a, b, c\}$ *est l'alphabet composé des symboles a, b et c.*

Étant donné un alphabet Σ, une suite finie de symboles $a_1a_2 \ldots a_n$, avec $a_i \in \Sigma$ pour tout $i \in [1, n]$, s'appelle un *mot* sur l'alphabet Σ. Les mots sont souvent représentés par des lettres minuscules de la fin de l'alphabet (ex : w, x, y, z). Soit un mot $w = a_1a_2 \ldots a_n$, avec $a_i \in \Sigma$ pour tout $i \in [1, n]$, alors la longueur de w, notée $|w|$, est n. En général, nous ne faisons pas de réelle distinction entre un symbole unique et un mot de longueur 1. Nous admettons aussi un mot de longueur 0, noté ε, que l'on appelle le *mot vide*.

On appelle *langage* un ensemble de mots sur un alphabet Σ. Les langages sont généralement représentés par la lettre L. Nous traitons sans distinction un mot w et le langage $\{w\}$ qui le contient lorsque cela ne cause aucune confusion.

Exemple 3.2. *$L = \{abbc, cb, \varepsilon\}$ est un langage sur l'alphabet $\Sigma = \{a, b, c\}$ contenant trois mots : abbc, cb et le mot vide.*

3.2 Opérations sur les mots et langages

Nous définissons l'opération de concaténation sur les mots, notée $\cdot$, comme suit : soient $w_1 = a_1 \dots a_m$, et $w_2 = b_1 \dots b_n$, alors $w_1 \cdot w_2 \stackrel{\text{def}}{=} a_1 \dots a_m b_1 \dots b_n$. En général, nous représentons la concaténation de deux mots w_1 et w_2 directement par $w_1 w_2$ plutôt que $w_1 \cdot w_2$. Nous étendons l'opération de concaténation à l'ensemble des langages : soient deux langages L_1 et L_2, alors $L_1 L_2 \stackrel{\text{def}}{=} \{w_1 w_2 \mid w_1 \in L_1 \text{ et } w_2 \in L_2\}$. Nous définissons également la *puissance* d'un langage, de manière récursive comme suit :

$$L^n \stackrel{\text{def}}{=} \begin{cases} LL^{n-1} & \text{si } n > 0, \\ \{\varepsilon\} & \text{si } n = 0. \end{cases}$$

Nous définissons l'opérateur de fermeture de Kleene * sur les langages, par :

$$L^* \stackrel{\text{def}}{=} \bigcup_{n \geq 0} L^n$$

et l'opérateur $+$ sur les langages par : $L^+ \stackrel{\text{def}}{=} L^* \setminus \{\varepsilon\}$.

Exemple 3.3. *Soit $L = \{ab, c\}$, alors $L^* = \{\varepsilon, ab, c, abab, abc, cab, cc, ababab, \dots\}$. Aussi, pour un alphabet $\Sigma = \{a, b\}$, Σ^* contient tous les mots sur a et b, incluant le mot vide, et Σ^+ contient tous les mots sur a et b, à l'exception du mot vide.*

Chapitre 4

Équivalence des grammaires simples

4.1 Grammaires simples

Nous présentons dans ce chapitre un premier résultat : un algorithme d'équivalence des grammaires simples. Nous débutons, dans cette section, par une présentation formelle des principaux concepts associés aux grammaires simples et aux langages simples.

DÉFINITION 4.1. *Une* grammaire simple *est un triplet* $G = (\Sigma, N, P)$ *où :*

- Σ *est un ensemble fini de symboles* terminaux*,*
- N *est un ensemble fini de symboles* non-terminaux*,*
- $P \subset N \times \Sigma \times N^*$ *est un ensemble fini de* règles de production*, tel que pour tous* $A \in N, a \in \Sigma$ *et* $\alpha, \beta \in N^*$ *:*

$$((A, a, \alpha) \in P \quad et \quad (A, a, \beta) \in P) \quad \Longrightarrow \quad \alpha = \beta.$$

Nous notons généralement les mots sur $(\Sigma \cup N)^*$ par des lettres grecques minuscules (ex : α, β, γ) et les symboles non-terminaux de N par des lettres majuscules (ex : $A, B, C \ldots$). Les règles de production $(A, a, \alpha) \in P$ sont notées $A \rightarrow a\alpha$. Nous notons par $|G|$ la taille de la représentation de la grammaire G, c.-à-d.

$$|G| \stackrel{\text{def}}{=} \Big(\sum_{(A,a,\alpha) \in P} |Aa\alpha| \Big) + |\Sigma| + |N|.$$

Les éléments de P définissent pour chaque $a \in \Sigma$ une relation $\xrightarrow{a} \subset N^+ \times N^*$, comme suit :

$$(A\alpha, \gamma\alpha) \in \xrightarrow{a} \quad \text{ssi} \quad (A, a, \gamma) \in P \text{ et } \alpha \in N^*.$$

Pour tout $w = a_1 \dots a_n$, avec $a_i \in \Sigma$ pour tout $i \in [1, n]$, nous définissons la relation de *dérivation* $\xrightarrow{w} \subset N^* \times N^*$ comme suit :

$(\alpha_0, \alpha_n) \in \xrightarrow{w}$ ssi :

- $\alpha_0 = \alpha_n$ et $w = \varepsilon$, ou
- $\exists\ \alpha_1, \dots, \alpha_{n-1}$ tels que $(\alpha_{i-1}, \alpha_i) \in \xrightarrow{a_i}$, pour tout $i \in [1, n]$.

En général, nous écrivons $\alpha \xrightarrow{w} \beta$ plutôt que $(\alpha, \beta) \in \xrightarrow{w}$. Nous disons que β est *dérivé* de α sur le mot w si $\alpha \xrightarrow{w} \beta$.

DÉFINITION 4.2. *Soit $G = (\Sigma, N, P)$ une grammaire simple. Pour tout $\alpha \in N^*$, nous définissons*

$$L_G(\alpha) = \{w \in \Sigma^* \mid \alpha \xrightarrow{w} \varepsilon\},$$

c.-à-d. l'ensemble des mots terminaux w pour lesquels α dérive sur le mot vide. $L_G(\alpha)$ est appelé un langage simple*. Lorsque G est clair selon le contexte, nous écrivons simplement $L(\alpha)$ plutôt que $L_G(\alpha)$.*

Nous notons par $||\alpha||$ la longueur minimale d'un mot de $L(\alpha)$, c.-à-d. :

$$||\alpha|| \stackrel{\texttt{def}}{=} \texttt{min}\{|w| \mid w \in L(\alpha)\}.$$

EXEMPLE 4.1. *Soit $G = (\{a, b\}, \{X, Y\}, P)$ une grammaire simple où $P = \{X \to aXY, X \to bY, Y \to b\}$. Le langage simple engendré par le non-terminal X est :*

$$L(X) = \{a^n b^{n+2} \mid n \geq 0\}.$$

Par exemple, $aabbbb \in L(X)$ car :

$$X \xrightarrow{a} XY \xrightarrow{a} XYY \xrightarrow{b} YYY \xrightarrow{b} YY \xrightarrow{b} Y \xrightarrow{b} \varepsilon.$$

Une dérivation peut également être représentée par un arbre de dérivation*, tel qu'illustré à la figure 4.1.*

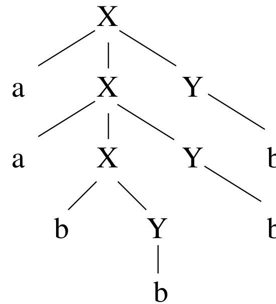

Fig. 4.1 – Arbre de dérivation du mot *aabbbb* (exemple 4.1).

Soit $G = (\Sigma, N, P)$ une grammaire simple. Deux non-terminaux $A, B \in N$ sont dits *équivalents* dans G, noté $A \equiv_G B$, ssi $L_G(A) = L_G(B)$. Le *problème d'équivalence des grammaires simples* est défini en terme d'entrée/sortie, comme suit :

ENTRÉE : Une grammaire simple $G = (\Sigma, N, P)$ et deux non-terminaux $A, B \in N$;
SORTIE : vrai si $A \equiv_G B$, faux sinon.

4.2 Quotient d'un langage simple

Tous les algorithmes que l'on connaît pour résoudre le problème d'équivalence des grammaires simples utilisent le fait que l'on peut calculer le *quotient* d'un langage simple par un autre, en supposant que celui-ci existe. Plus précisément, soit A, B deux non-terminaux d'une grammaire simple $G = (\Sigma, N, P)$ tels que $L(A) = L(B) \cdot L$ pour un certain $L \subseteq \Sigma^*$. Alors le quotient de $L(A)$ par $L(B)$ est L.

Un langage L est appelé un *langage préfixe* ssi L ne contient pas deux mots distincts tels que l'un est préfixe de l'autre. Comme les langages simples sont des langages préfixes [17], le quotient de $L(A)$ par $L(B)$ peut être obtenu par une dérivation de A sur n'importe quel mot $w \in L(B)$, c.-à-d. que s'il existe L tel que $L(A) = L(B) \cdot L$, alors $A \xrightarrow{w} \gamma$, avec $\gamma \in N^*$, implique $L(\gamma) = L$.

Comme $||B||$ peut être exponentiel par rapport à $|G|$, il n'est pas toujours possible de choisir un mot w de $L(B)$ dont la longueur n'est pas exponentielle par rapport à $|G|$. Cette méthode par dérivation directe prend donc un temps exponentiel en pire cas. Il est possible d'améliorer cette complexité lorsque l'on connaît au préalable la valeur de $||A||$ pour tout $A \in G$.

Lemme 4.1. *Soit $G = (\Sigma, N, P)$ une grammaire simple telle que $|G| = n$. Il est possible de calculer $||A||$ pour tout $A \in N$ en temps $O(n \log n)$.*

Démonstration. Trouver $||A||$ pour tout $A \in N$ correspond au problème consistant à trouver les chemins les plus courts dans un graphe de type *et/ou*, à partir d'une source unique. Ce problème peut être résolu en temps $O(n \log n)$ en utilisant l'algorithme de Dijkstra. □

À partir de ces valeurs, il est facile de trouver en temps $O(n)$ une règle $(A, a, \alpha) \in P$, pour chaque non-terminal $A \in N$, permettant de dériver un mot le plus court de $L(A)$, ce qui nous donne, implicitement, l'arbre de dérivation d'un mot le plus court pour chaque non-terminal $A \in N$.

Lemme 4.2. *Soient $G = (\Sigma, N, P)$ une grammaire simple telle que $|G| = n$ et $A, B \in N$ tels que $L(A) = L(B) \cdot L$ pour un certain $L \subseteq \Sigma^*$. Nous pouvons calculer $\gamma \in N^*$ tel que $L(\gamma) = L$ en temps $O(n)$. Il est garanti que $|\gamma| \leq n$.*

Démonstration. Considérons l'arbre de dérivation d'un mot le plus court w de $L(A)$. Pour trouver γ, il suffit de trouver, dans l'arbre de dérivation de w, le chemin allant de la racine jusqu'à la feuille correspondant au symbole situé à la position $||B||$ dans w. Les non-terminaux situés aux racines des sous-arbres qui sont à droite de ce chemin forment le mot γ. Comme w est un mot le plus court, aucun chemin dans l'arbre de dérivation ne contient deux occurrences d'un même non-terminal : la profondeur de l'arbre est donc inférieure ou égale à $|N|$. Ainsi, $|\gamma| \leq n$, car nous ne considérons qu'une seule règle de production pour chaque non-terminal présent sur le chemin, et γ peut être calculé en temps $O(n)$. □

Le résultat de la méthode présentée dans la preuve du lemme 4.2 pour calculer le quotient de A par B sera noté $\texttt{quot}(A, B)$. Notez que $\texttt{quot}(A, B)$ donne un résultat dès que $||A|| \geq ||B||$, sans vérifier s'il existe réellement $L \in \Sigma^*$ tel que $L(A) = L(B) \cdot L$. Ainsi, $L(A) = L(B) \cdot L(\texttt{quot}(A, B))$ ssi $L(B)$ est un diviseur gauche de $L(A)$.

Exemple 4.2. *Soit* $G = (\{a, b\}, \{A, B, C, D, E\}, P)$*, où :*

$$\begin{aligned} P = \{ \; & A \to aCEEDCB, && C \to bED, \\ & A \to bEEBDC, && D \to aEED, \\ & B \to aEEB, && D \to b, \\ & B \to bC, && E \to b \;\; \}. \end{aligned}$$

Nous allons calculer la valeur de **quot**(A, B)*. Nous calculons tout d'abord* $||A|| = 11$*,* $||B|| = 4$*,* $||C|| = 3$*,* $||D|| = 1$ *et* $||E|| = 1$*. À partir de ces valeurs, nous trouvons, dans l'arbre de dérivation d'un mot le plus court* $w \in L_G(A)$*, le chemin partant de la racine jusqu'à la feuille correspondant au symbole terminal à la position* $||B|| = 4$ *de* w*, tel qu'illustré à la figure 4.2.*

Fig. 4.2 – Arbre de dérivation d'un mot le plus court $w \in L_G(A)$. Le chemin partant de la racine jusqu'à la feuille correspondant au symbole terminal à la position $||B|| = 4$ de w est représenté en gras. La séquence des symboles encerclés constitue le résultat de quot(A, B).

Le résultat de **quot**(A, B) *est formé de la séquence de non-terminaux situés aux racines des sous-arbres qui sont à droite de ce chemin, c.à-d.* **quot**$(A, B) = CDC$*. Notez que comme nous connaissons les valeurs* $||X||$ *pour tout* $X \in N$*, il n'est jamais nécessaire de représenter explicitement les sous-arbres qui sont représentés par des traits pointillés à la figure 4.2.*

4.3 Fonction de décomposition

Notre algorithme d'équivalence des grammaires simples utilise également le concept de *fonction de décomposition*.

Définition 4.3. *Soit N un alphabet. Une* fonction de décomposition *est une application $\mathcal{D} : N \to N^+$ pour laquelle il existe un ordre total sur les symboles de N, tel que pour tout $A \in N$, $\mathcal{D}(A) = A$ ou $A > B$ pour tous les symboles B présents dans le mot $\mathcal{D}(A)$.*

Nous étendons la définition d'une fonction de décomposition $\mathcal{D} : N \to N^+$ en une fonction $\mathcal{D} : N^* \to N^*$, telle que $\mathcal{D}(\varepsilon) \stackrel{\text{def}}{=} \varepsilon$ et $\mathcal{D}(A_1A_2 \dots A_n) \stackrel{\text{def}}{=} \mathcal{D}(A_1)\mathcal{D}(A_2) \dots \mathcal{D}(A_n)$, où $A_i \in N$ pour tout $i \in [1, n]$. Une fonction de décomposition $\mathcal{D} : N^* \to N^*$ est donc entièrement définie à partir des valeurs de $\mathcal{D}(A)$ pour chaque $A \in N$, mais son application s'étend aux mots sur N^*.

Nous définissons $\mathcal{D}^k$, pour tout $k \geq 1$, comme suit :

$$\mathcal{D}^k \stackrel{\text{def}}{=} \begin{cases} \mathcal{D} & \text{si } k = 1, \\ \mathcal{D} \circ \mathcal{D}^{k-1} & \text{si } k > 1 \end{cases}$$

où $\circ$ dénote la composition de fonctions. Nous notons $\mathcal{D}^{|N|}$ par $\mathcal{D}^*$ car $\mathcal{D}^{|N|} = \mathcal{D}^{|N|+1}$.

Exemple 4.3. *Soient $N = \{A, B, C, D\}$ et $\mathcal{D} : N^* \to N^*$ une fonction de décomposition définie par $\mathcal{D}(A) = BCCBD$, $\mathcal{D}(B) = CD$, $\mathcal{D}(C) = C$ et $\mathcal{D}(D) = D$, alors :*

$$\mathcal{D}^*(ADB) = CDCCCDDDCD.$$

Dans la construction de notre algorithme, nous sommes intéressé par un type particulier de fonctions de décomposition, appelé *self-proving*.

Définition 4.4. *Soit $G = (\Sigma, N, P)$ une grammaire simple. Une fonction de décomposition $\mathcal{D} : N^* \to N^*$ est dite* self-proving *si pour chaque $A \in N$, nous avons :*
- *Si $A \xrightarrow{a} \alpha$, alors il existe β tel que $\mathcal{D}(A) \xrightarrow{a} \beta$ et $\mathcal{D}^*(\alpha) = \mathcal{D}^*(\beta)$, et*
- *Si $\mathcal{D}(A) \xrightarrow{a} \beta$, alors il existe α tel que $A \xrightarrow{a} \alpha$ et $\mathcal{D}^*(\alpha) = \mathcal{D}^*(\beta)$,*

où $a \in \Sigma$ et $\alpha, \beta \in N^*$.

Les fonctions de décomposition de type *self-proving* ont été introduites par B. Courcelle [6], et utilisées dans les algorithmes d'équivalence des grammaire simples présentés par D. Caucal [5] et par Y. Hirshfeld, M. Jerrum et F. Moller [12]. L'utilité de ces fonctions dans le contexte de l'équivalence des grammaires simples s'explique par le lemme suivant.

LEMME 4.3. *Soient $G = (\Sigma, N, P)$ une grammaire simple et $\mathcal{D} : N^* \to N^*$ une fonction de décomposition* self-proving. *Alors, pour tout $\alpha \in N^*$, $L_G(\alpha) = L_G(\mathcal{D}(\alpha))$.*

Démonstration. Pour tous $n \geq 0$ et $\alpha \in N^*$, nous définissons :

$$L_n(\alpha) \stackrel{\text{def}}{=} \{w \in L_G(\alpha) \mid |w| \leq n\}.$$

Nous démontrons, par induction sur n, que si $\mathcal{D}$ est une fonction de décomposition *self-proving* en G, alors pour tous $\alpha \in N^*$ et $n \geq 0$, $L_n(\alpha) = L_n(\mathcal{D}(\alpha))$.

Si $n = 0$: $L_0(\alpha) = \{\varepsilon\}$ si $\alpha = \varepsilon$, sinon $L_0(\alpha) = \emptyset$. Or, $\mathcal{D}(\alpha) = \varepsilon$ ssi $\alpha = \varepsilon$, par définition d'une fonction de décomposition, donc $L_0(\alpha) = L_0(\mathcal{D}(\alpha))$ pour tout $\alpha \in N^*$.

Supposons que pour tout $k \in [0, n-1]$, $L_k(\alpha) = L_k(\mathcal{D}(\alpha))$ pour tout $\alpha \in N^*$. Nous allons démontrer, par contradiction, que $L_n(\alpha) = L_n(\mathcal{D}(\alpha))$ pour tout $\alpha \in N^*$. Supposons qu'il existe un mot $\alpha \in N^*$ tel que $L_n(\alpha) \neq L_n(\mathcal{D}(\alpha))$. Alors il existe $w \in \Sigma^*$ tel que $w \in L_n(\alpha)$ et $w \notin L_n(\mathcal{D}(\alpha))$, ou $w \notin L_n(\alpha)$ et $w \in L_n(\mathcal{D}(\alpha))$. Supposons, sans perte de généralité, que nous sommes en présence du premier cas, c.-à-d. $w \in L_n(\alpha)$ et $w \notin L_n(\mathcal{D}(\alpha))$.

Nous avons $|w| = n > 0$. Soient $a \in \Sigma$ et $w' \in \Sigma^*$ tels que $w = aw'$. Comme $\mathcal{D}$ est *self-proving* et que $\alpha \xrightarrow{a} \beta_1$ pour un certain $\beta_1 \in N^*$, nous avons $\mathcal{D}(\alpha) \xrightarrow{a} \beta_2$ pour un certain $\beta_2 \in N^*$ et $\mathcal{D}^*(\beta_1) = \mathcal{D}^*(\beta_2)$. De plus, $w' \in L_{n-1}(\beta_1)$ et $w' \notin L_{n-1}(\beta_2)$.

Par hypothèse d'induction, $L_{n-1}(\gamma) = L_{n-1}(\mathcal{D}(\gamma))$ pour tout $\gamma \in N^*$, donc $L_{n-1}(\beta_1) = L_{n-1}(\mathcal{D}^*(\beta_1)) = L_{n-1}(\mathcal{D}^*(\beta_2)) = L_{n-1}(\beta_2)$. Mais, $w' \in L_{n-1}(\beta_1)$, donc $w' \in L_{n-1}(\beta_2)$, ce qui est une contradiction. □

Une application directe du lemme 4.3 est que si $\mathcal{D}$ est une fonction de décomposition de type *self-proving*, alors pour tous $\alpha_1, \alpha_2 \in N^*$, $\mathcal{D}^*(\alpha_1) = \mathcal{D}^*(\alpha_2)$ implique $\alpha_1 \equiv_G \alpha_2$.

4.4 Comparaison de chaînes compressées

Pour vérifier qu'une fonction de décomposition $\mathcal{D} : N^* \to N^*$ est *self-proving*, il est nécessaire de vérifier si $\mathcal{D}^*(\alpha) = \mathcal{D}^*(\beta)$ pour certains mots $\alpha, \beta \in N^*$. Cette vérification peut être effectuée directement, en représentant explicitement les chaînes $\mathcal{D}^*(\alpha)$ et $\mathcal{D}^*(\beta)$ et en les comparant symbole par symbole. Cependant, la longueur d'un mot $\mathcal{D}^*(\alpha)$ peut être exponentielle par rapport à la taille de l'alphabet sur lequel $\mathcal{D}$ est défini.

Une approche pour résoudre ce problème en temps polynomial est de comparer les chaînes de caractères en les conservant sous leur forme compressée, c.à-d. en utilisant directement les valeurs de $\mathcal{D}$, α et β, sans calculer explicitement $\mathcal{D}^*(\alpha)$ et $\mathcal{D}^*(\beta)$. Cette méthode, qui a été utilisée également par Y. Hirshfeld, M. Jerrum et F. Moller [12] dans leur solution polynomiale au problème d'équivalence des grammaires simples, utilise des techniques élaborées qui ont été développées dans le contexte de la recherche sur la comparaison des chaînes de caractères compressées.

Notre algorithme considère une version étendue de ce problème : lorsque $\mathcal{D}^*(\alpha) \neq \mathcal{D}^*(\beta)$, nous désirons également connaître la première paire de symboles, à partir du début des mots, qui témoigne de l'inégalité de $\mathcal{D}^*(\alpha)$ et $\mathcal{D}^*(\beta)$. Plus précisément, le problème que nous cherchons à résoudre est le suivant :

Entrée : deux mots $\alpha, \beta \in N^*$ et une fonction de décomposition $\mathcal{D} : N^* \to N^*$;

Sortie :
- NIL, si $\mathcal{D}^*(\alpha) = \mathcal{D}^*(\beta)$;
- ÉCHEC, si $\mathcal{D}^*(\alpha) \neq \mathcal{D}^*(\beta)$ et l'un de $\mathcal{D}^*(\alpha)$, $\mathcal{D}^*(\beta)$ est préfixe de l'autre ;
- $(A, B) \in N \times N$, sinon, où A et B sont les premiers symboles apparaissant à la même position dans $\mathcal{D}^*(\alpha)$ et $\mathcal{D}^*(\beta)$, respectivement, qui diffèrent.

Nous notons par `First-MP`$(\alpha, \beta, \mathcal{D})$ un algorithme solutionnant ce problème[1]. Nous supposons que `First-MP` retourne des paires (A, B) ordonnées, c.-à-d. telles que $||A|| \geq ||B||$.

Une fonction de décomposition $\mathcal{D} : N^* \to N^*$ est dite *binaire* si $|\mathcal{D}(A)| \leq 2$ pour tout $A \in N$.

[1]La notation `First-MP` provient de l'anglais : *First Mismatch Pair*.

LEMME 4.4. *Soient $\mathcal{D} : N^* \to N^*$ une fonction de décomposition binaire et des mots $\alpha, \beta \in N^*$ tels que la longueur de α et β est au plus $O(|N|)$. Alors, le problème* **`First-MP`**$(\alpha, \beta, \mathcal{D})$ *peut être résolu en temps $O(k^2 \cdot h^2)$, où $k = |N|$ et :*

$$h \stackrel{\text{def}}{=} \min\{k \geq 1 \mid \mathcal{D}^k = \mathcal{D}^{k+1}\}$$

est la profondeur de la fonction de décomposition.

Démonstration. Ce résultat est obtenu en utilisant l'algorithme présenté dans [19]. □

4.5 Algorithme d'équivalence des grammaires simples

Nous décrivons maintenant le fonctionnement de notre algorithme d'équivalence des grammaires simples. Soient $G = (\Sigma, N, P)$ une grammaire simple et deux non-terminaux $X, Y \in N$. L'algorithme ÉQUIVALENCEGS cherche à déterminer si $X \equiv_G Y$. L'idée de l'algorithme est de construire une fonction de décomposition $\mathcal{D} : N^* \to N^*$ de type *self-proving* telle que $\mathcal{D}^*(X) = \mathcal{D}^*(Y)$, ce qui démontrera, d'après le lemme 4.3, que $X \equiv_G Y$, ou de trouver en cours d'exécution une preuve nous permettant de réfuter l'équivalence des symboles d'entrée.

Nous fixons tout d'abord un ordre total $A_1 < A_2 < \cdots < A_n$ sur les éléments de N, tel que $i < j$ implique $||A_i|| \leq ||A_j||$ et nous initialisons la fonction de décomposition $\mathcal{D}$ avec $\mathcal{D}(A) = A$ pour tout $A \in N$. Nous conservons également un ensemble $Q \subset N^* \times N^*$ de paires de mots, initialisé à $Q = \{(X, Y)\}$, c.-à-d. que Q contient initialement la paire de non-terminaux d'entrée de l'algorithme.

En cours d'exécution, nous ajoutons à Q toutes les paires (β_1, β_2) pour lesquelles nous devons vérifier si $\mathcal{D}^*(\beta_1) = \mathcal{D}^*(\beta_2)$. Cette vérification est effectuée par la boucle principale de l'algorithme. À chaque itération, `First-MP`$(\beta_1, \beta_2, \mathcal{D})$ est exécutée sur une paire (β_1, β_2) choisie arbitrairement dans Q. Si `First-MP` retourne (A, B), nous effectuons une mise à jour de $\mathcal{D}$ en assignant $\mathcal{D}(A) := B \cdot \texttt{quot}(A, B)$ de manière à résoudre le conflit. Nous conservons la paire (β_1, β_2) dans Q, et continuons itérativement à exécuter `First-MP`$(\beta_1, \beta_2, \mathcal{D})$ et à effectuer des mises à jour de $\mathcal{D}$, jusqu'à ce que `First-MP` retourne NIL (c.-à-d. jusqu'à ce que $\mathcal{D}^*(\beta_1) = \mathcal{D}^*(\beta_2)$), ou que `First-MP` retourne ÉCHEC, ce qui constituerait alors une preuve de la non-équivalence des mots d'entrée.

Comme nous désirons que $\mathcal{D}$ soit *self-proving*, nous devons également vérifier, lorsque nous faisons une modification de $\mathcal{D}(A)$ pour un non-terminal $A \in N$, que pour tout $a \in \Sigma$:

1. Si $A \xrightarrow{a} \beta_1'$, alors il existe β_2' tel que $\mathcal{D}(A) \xrightarrow{a} \beta_2'$,
2. Si $\mathcal{D}(A) \xrightarrow{a} \beta_2'$, alors il existe β_1' tel que $A \xrightarrow{a} \beta_1'$, et
3. Si nous trouvons, aux points 1 et 2, que A et $\mathcal{D}(A)$ ont des dérivations sur a, alors $\mathcal{D}^*(\beta_1') = \mathcal{D}^*(\beta_2')$,

où $\beta_1', \beta_2' \in N^*$.

Les deux premières vérifications sont effectuées immédiatement pour chaque $a \in \Sigma$, et l'algorithme retourne FAUX si l'une de ces vérifications échoue. Si elles réussissent, nous ajoutons à l'ensemble Q toutes les paires (β_1', β_2') obtenues, afin de vérifier ultérieurement si $\mathcal{D}^*(\beta_1') = \mathcal{D}^*(\beta_2')$ pour chacune d'elles.

Si Q devient vide, nous savons que la fonction de décomposition $\mathcal{D}$ qui a été construite par l'algorithme est *self-proving*. Comme la paire de symboles d'entrée (X, Y) est initialement dans Q, nous savons également que $\mathcal{D}^*(X) = \mathcal{D}^*(Y)$, donc $X \equiv_G Y$.

L'algorithme ÉQUIVALENCEGS est présenté à la figure 4.3. Pour tous $\alpha \in N^+$ et $a \in \Sigma$, nous notons par $\alpha \xrightarrow{a}$ le fait qu'il existe $\beta \in N^*$ tel que $\alpha \xrightarrow{a} \beta$ et par $\alpha \not\xrightarrow{a}$ le fait qu'il n'en existe pas. Nous écrivons $(\alpha, \beta) \xrightarrow{a} (\alpha', \beta')$ pour dire que $\alpha \xrightarrow{a} \alpha'$ et $\beta \xrightarrow{a} \beta'$.

4.5.1 Preuve d'exactitude et complexité

Nous procédons à l'analyse de la complexité de l'algorithme ÉQUIVALENCEGS. Pour en simplifier l'analyse, nous considérons que G est en forme normale binaire de Greibach (noté `FNG2`), c.-à-d. que pour chaque $(A, a, \alpha) \in P$ nous avons $\alpha \in \{\varepsilon\} \cup N \cup N^2$.

LEMME 4.5. *Pour chaque grammaire simple G, il existe une grammaire simple équivalente G' en **FNG2** avec $O(|G|)$ symboles non-terminaux. G' peut être construite à partir de G en temps $O(|G|)$.*

Pour toute grammaire simple $G = (\Sigma, N, P)$ en `FNG2`, $|G|$ et $|N|$ sont de même ordre. Ainsi, par la taille d'une grammaire simple $G = (\Sigma, N, P)$, nous signifierons dorénavant $n = |N|$.

Entrée : Une grammaire simple $G = (\Sigma, N, P)$ et des non-terminaux $X, Y \in N$;
Sortie : VRAI si $X \equiv_G Y$, FAUX sinon.
  Initialisation :
    $Q := \{(X, Y)\}$;
    for each $A \in N$ **do** $\mathcal{D}(A) := A$;
  while Q n'est pas vide **do**
    (β_1, β_2) := un élément de Q ;
    switch (`First-MP`$(\beta_1, \beta_2, \mathcal{D})$) **do**
      case NIL : enlever (β_1, β_2) de Q ;
      case ÉCHEC : **return** FAUX ;
      case (A, B) : — Nous supposons $A \geq B$.
        $\gamma := quot(A, B)$;
        $\mathcal{D}(A) := B\gamma$;
        for each $a \in \Sigma$ **do**
          if $(A, B\gamma) \xrightarrow{a} (\beta'_1, \beta'_2)$ **then** ajouter (β'_1, β'_2) à Q ;
          if $(A \xrightarrow{a}$ et $B \stackrel{a}{\not\to})$ ou $(A \stackrel{a}{\not\to}$ et $B \xrightarrow{a})$ **then return** FAUX ;
  return VRAI ;

FIG. 4.3 – Algorithme ÉQUIVALENCEGS.

LEMME 4.6. *Le nombre d'itérations de la boucle **while** de l'algorithme* ÉQUIVALENCEGS *est $O(n)$.*

Démonstration. Une mise à jour de $\mathcal{D}$ est effectuée pour un symbole $A \in N$, seulement si A est présent dans le mot $\mathcal{D}^*(\beta_1)$ pour un certain $\beta_1 \in N^*$, c.-à-d. seulement si $\mathcal{D}(A) = A$, et la nouvelle valeur assignée à $\mathcal{D}(A)$ diffère nécessairement de A. Ainsi, la valeur de $\mathcal{D}(A)$ est modifiée au plus une fois pour chaque $A \in N$.

Après avoir effectué au plus $n - 1$ mises à jour de $\mathcal{D}$, `First-MP`$(\beta_1, \beta_2, \mathcal{D})$ doit nécessairement retourner NIL (dans ce cas (β_1, β_2) est enlevé de Q) ou ÉCHEC (dans ce cas l'algorithme se termine).

Le nombre maximal d'itérations de l'algorithme est donc égal à la somme du nombre de mises à jour de $\mathcal{D}$ qui sont effectuées (au plus $n - 1$) et du nombre total d'éléments ajoutés dans Q au cours de l'exécution de l'algorithme. Q ne contient initialement qu'un seul élément, et nous ajoutons des éléments à Q seulement lorsque $\mathcal{D}$ est modifié (au plus $|\Sigma|$ éléments sont ajoutés chaque fois). Le nombre d'itérations de l'algorithme est donc $O(n)$. □

Corollaire 4.1. *L'algorithme* ÉquivalenceGS *fonctionne en temps $O(n \cdot F(n))$, où $F(n)$ est la complexité de l'algorithme* **`First-MP`**.

Démonstration. Le résultat découle directement des lemmes 4.1, 4.2 et 4.6. □

Lemme 4.7. *Chaque instance de* **`First-MP`**$(\beta_1, \beta_2, \mathcal{D})$ *dans l'algorithme* ÉquivalenceGS *peut être répondue en temps $O(n^6 \log^2 n)$.*

Démonstration. Soient N un alphabet et $\mathcal{D} : N^* \times N^*$ la fonction de décomposition construite par ÉquivalenceGS. D'après le lemme 4.2, $|\mathcal{D}(A)| \leq n$ pour tout $A \in N$. Nous pouvons construire une fonction de décomposition binaire $\mathcal{D}_2$ telle que $\mathcal{D}_2^* = \mathcal{D}^*$ contenant $k \leq n^2$ non-terminaux et ayant une profondeur $h = O(n \log n)$.

La construction de $\mathcal{D}_2$ à partir de $\mathcal{D}$ est effectuée de manière similaire à une transformation balancée en forme normale de Chomsky. Si $\mathcal{D}(A) = B_1 B_2 \ldots B_n$, nous introduisons $n - 2$ nouveaux non-terminaux auxiliaires de manière à construire un arbre binaire balancé engendrant $B_1 B_2 \ldots B_n$ à partir de A. Pour transformer $\mathcal{D}$ en $\mathcal{D}_2$, il faut introduire $O(n)$ nouveaux non-terminaux pour chaque non-terminal de N, le nombre total de non-terminaux dans $\mathcal{D}_2$ est donc $O(n^2)$, c.-à-d. k est dans $O(n^2)$. Cependant, la profondeur de $\mathcal{D}$ est changée seulement de manière logarithmique pour former $\mathcal{D}_2$, car pour chaque chemin allant de haut en bas dans l'arbre de décomposition de $\mathcal{D}_2^*(A)$ avec $A \in N$, nous avons au maximum n non-terminaux appartenant à N et au plus $O(n \log n)$ non-terminaux auxiliaires, c.-à-d. que h est dans $O(n \log n)$.

Le résultat découle maintenant du lemme 4.4. □

Lemme 4.8. *L'algorithme* ÉquivalenceGS *est correct.*

Démonstration. Nous savons, d'après le lemme 4.6 que l'algorithme se termine toujours. Pour démontrer que l'algorithme est correct, nous vérifions tout d'abord que si l'algorithme retourne vrai, alors $X \equiv_G Y$.

D'après le lemme 4.3, il suffit de démontrer qu'à la fin de l'exécution, $\mathcal{D}$ est une fonction de décomposition de type *self-proving* et $\mathcal{D}^*(X) = \mathcal{D}^*(Y)$. La deuxième condition est immédiate, car :

1. Q contient initialement (X, Y),
2. un élément (β_1, β_2) n'est enlevé de Q que si $\mathcal{D}^*(\beta_1) = \mathcal{D}^*(\beta_2)$, et

3. l'algorithme ne retourne VRAI que si Q est vide.

Si une modification de $\mathcal{D}$ est effectuée après avoir vérifié $\mathcal{D}^*(\beta_1) = \mathcal{D}^*(\beta_2)$, l'égalité demeure vraie car nous ne modifions la valeur de $\mathcal{D}(A)$ pour un $A \in N$ que si $\mathcal{D}(A) = A$. Ainsi, $\mathcal{D}^*(X) = \mathcal{D}^*(Y)$ à la fin de l'exécution de l'algorithme.

Nous vérifions maintenant que $\mathcal{D}$ est nécessairement une fonction de décomposition de type *self-proving* à la fin de l'exécution. Tout d'abord, remarquez que si $\mathcal{D}(A) = A$ pour un certain $A \in N$, les conditions pour que $\mathcal{D}$ soit *self-proving* sont respectées pour ce symbole. Aussi, chaque fois que nous assignons $\mathcal{D}(A) = \gamma$ pour un symbole $A \in N$, γ ne contient que des symboles $B \in N$ tels que $B < A$, donc $\mathcal{D}$ est bien une fonction de décomposition. L'algorithme vérifie également, à chaque modification de $\mathcal{D}(A)$, que les conditions pour que $\mathcal{D}$ soit *self-proving* sont respectées par A et par la nouvelle valeur de $\mathcal{D}(A)$. Donc, $\mathcal{D}$ est nécessairement une fonction de décomposition de type *self-proving* à la fin de l'exécution.

Nous démontrons maintenant que si $X \equiv_G Y$, alors l'algorithme retourne VRAI. Supposons qu'au début d'une itération de l'algorithme :

1. toutes les paires $(\beta_1, \beta_2) \in Q$ sont telles que $\beta_1 \equiv_G \beta_2$ et,
2. $A \equiv_G \mathcal{D}(A)$ pour tout $A \in N$.

Nous allons démontrer que FAUX ne peut pas être retourné par l'algorithme au cours de l'itération, et qu'à la fin de l'itération, ces deux conditions tiennent toujours.

Notons tout d'abord que pour tout $(\beta_1, \beta_2) \in Q$, `First-MP`$(\beta_1, \beta_2, \mathcal{D})$ ne peut pas retourner ÉCHEC. Si `First-MP` retourne NIL, la paire (β_1, β_2) est enlevée de Q, l'itération se termine et les conditions 1 et 2 sont toujours respectées. Si `First-MP`$(\beta_1, \beta_2, \mathcal{D})$ retourne (A, B), alors $L(B)$ doit être un diviseur gauche de $L(A)$, car nous avons supposé $\beta_1 \equiv_G \beta_2$ et $A \equiv_G \mathcal{D}(A)$ pour tout $A \in N$, donc $A \equiv_G B\texttt{quot}(A, B)$. Après avoir assigné $\mathcal{D}(A) := B\texttt{quot}(A, B)$, $\mathcal{D}$ respecte toujours la condition 2. Comme $L(B)$ est un diviseur gauche de $L(A)$, il est impossible de trouver un $a \in \Sigma$ tel que ($A \xrightarrow{a}$ et $B \stackrel{a}{\not\rightarrow}$) ou ($A \stackrel{a}{\not\rightarrow}$ et $B \xrightarrow{a}$), donc l'algorithme ne peut retourner FAUX à cet endroit, et les paires (β_1', β_2') ajoutées à Q sont telles que $\beta_1' \equiv_G \beta_2'$. Ainsi, la valeur FAUX ne peut pas être retournée au cours de l'itération et les deux conditions sont nécessairement respectées à la fin de celle-ci.

Si $X \equiv_G Y$, alors les conditions 1 et 2 sont nécessairement respectées au départ de l'algorithme. Elles seront donc conservées tout au long de l'exécution de l'algorithme

et celui-ci ne retournera pas la valeur FAUX. Comme l'algorithme se termine toujours, l'ensemble Q sera éventuellement vide et l'algorithme retournera VRAI.

Nous avons démontré que $X \equiv_G Y$ ssi l'algorithme retourne VRAI et l'algorithme se termine toujours d'après le lemme 4.6, l'algorithme ÉQUIVALENCEGS est donc correct.

□

THÉORÈME 4.1. *L'algorithme* ÉQUIVALENCEGS(X, Y, G) *décide de l'équivalence de deux non-terminaux* X *et* Y *d'une grammaire simple* G *en temps* $O(n^7 \log^2 n)$*, où* $n = |G|$*.*

Démonstration. Le résultat est une conséquence directe du corollaire 4.1 et des lemmes 4.7 et 4.8. □

4.6 Exemples d'exécution de ÉquivalenceGS

Nous présentons un exemple de l'exécution de l'algorithme ÉQUIVALENCEGS pour un cas où les symboles reçus en entrée par l'algorithme sont équivalents.

EXEMPLE 4.4. *Soit* $G = (\{a, b\}, \{X, Y, Z, V\}, \{X \rightarrow a, X \rightarrow bXY, Y \rightarrow aX, Y \rightarrow bYY, Z \rightarrow aXX, Z \rightarrow bYXY, V \rightarrow aY, V \rightarrow bYZ\})$*. Nous considérons l'exécution d'un appel de* ÉQUIVALENCEGS(V, Z, G)*.*

Dans la phase d'initialisation, nous calculons la longueur d'un mot le plus court pouvant être engendré par chaque non-terminal, c.-à-d. $||X|| = 1$*,* $||Y|| = 2$*,* $||Z|| = ||V|| = 3$*, et nous fixons un ordre total sur les non-terminaux de la grammaire qui doit être compatible avec ces valeurs, par exemple* $X < Y < Z < V$*.*

Nous initialisons Q *à* $\{(V, Z)\}$ *et la fonction de décomposition* $\mathcal{D}$ *avec* $\mathcal{D}(A) := A$*, pour chaque non-terminal* A *de la grammaire.*

Nous débutons la boucle ***while*** *en choisissant un élément de* Q*, c.-à-d.* $(\beta_1, \beta_2) = (V, Z)$*, et en appelant* `First-MP`$(\beta_1, \beta_2, \mathcal{D})$*. L'algorithme présenté dans [19] doit être utilisé pour effectuer cette opération. Cependant, pour simplifier la présentation, nous expliciterons ici les valeurs de* $\mathcal{D}^*(\beta_1)$ *et* $\mathcal{D}^*(\beta_2)$*.*

Dans notre cas `First-MP`$(\beta_1, \beta_2, \mathcal{D}) = (V, Z)$*, car* $\mathcal{D}^*(\beta_1) = V$ *et* $\mathcal{D}^*(\beta_2) = Z$*.*

*La prochaine étape consiste à calculer le quotient du premier symbole, V, par le second symbole, Z. Comme $||V|| = ||Z|| = 3$, **`quot`**$(V, Z) = \varepsilon$. Se référer à la démonstration du lemme 4.2 et à l'exemple 4.2 pour plus de détails sur l'opération **`quot`**.*

*En utilisant les valeurs calculées par **`First-MP`** et par **`quot`**, l'algorithme met à jour la fonction de décomposition $\mathcal{D}$ en assignant $\mathcal{D}(V) := Z\varepsilon = Z$.*

*Comme dernière étape de la boucle **while**, l'algorithme calcule les dérivations sur les mêmes symboles terminaux, en parallèle, pour V et pour sa nouvelle valeur de décomposition Z, c.-à-d., $(V, Z) \xrightarrow{a} (Y, XX)$ et $(V, Z) \xrightarrow{b} (YZ, YXY)$. Les paires résultantes (Y, XX) et (YZ, YXY) sont ajoutées à Q.*

La trace complète de l'exécution de l'algorithme est présentée à la Fig. 4.4. Les éléments soulignés dans l'ensemble Q sont les paires (β_1, β_2) considérées par l'algorithme à chaque itération.

Valeurs au début de l'itération					Appels de fonction			
Q	$\mathcal{D}(X)$	$\mathcal{D}(Y)$	$\mathcal{D}(Z)$	$\mathcal{D}(V)$	$\mathcal{D}^*(\beta_1)$	$\mathcal{D}^*(\beta_2)$	`First-MP`	*quot*
$\underline{(V,Z)}$	X	Y	Z	V	V	Z	(V,Z)	ε
$\underline{(V,Z)}$, (Y,XX), (YZ,YXY)	X	Y	Z	Z	Z	Z	NIL	—
$\underline{(Y,XX)}$, (YZ,YXY)	X	Y	Z	Z	Y	XX	(Y,X)	X
$\underline{(Y,XX)}$, (YZ,YXY), (X,X), (YY,XYX)	X	XX	Z	Z	XX	XX	NIL	—
$\underline{(YZ,YXY)}$, (X,X), (YY,XYX)	X	XX	Z	Z	XXZ	$XXXXX$	(Z,X)	XX
$\underline{(YZ,YXY)}$, (X,X), (YY,XYX), (XX,XX), $(YXY,XYXX)$	X	XX	XXX	Z	$XXXXX$	$XXXXX$	NIL	—
$\underline{(X,X)}$, (YY,XYX), (XX,XX), $(YXY,XYXX)$	X	XX	XXX	Z	X	X	NIL	—
$\underline{(YY,XYX)}$, (XX,XX), $(YXY,XYXX)$	X	XX	XXX	Z	$XXXX$	$XXXX$	NIL	—
$\underline{(XX,XX)}$, $(YXY,XYXX)$	X	XX	XXX	Z	XX	XX	NIL	—
$\underline{(YXY,XYXX)}$	X	XX	XXX	Z	$XXXXX$	$XXXXX$	NIL	—
vide	X	XX	XXX	Z				

FIG. 4.4 – Exemple d'exécution de ÉQUIVALENCEGS démontrant $L_G(V) = L_G(Z)$.

*Après dix itérations de la boucle **while**, l'ensemble Q devient vide, nous en concluons que les non-terminaux V et Z sont équivalents.*

Nous présentons maintenant un exemple d'exécution de l'algorithme ÉQUIVALENCEGS dans un cas où les symboles d'entrée ne sont pas équivalents.

EXEMPLE 4.5. *Nous considérons l'exécution de l'algorithme* ÉQUIVALENCEGS(V, Z, G), *où* $G = (\{a, b\}, \{X, Y, Z, V\}, \{X \rightarrow a, X \rightarrow bZX, Y \rightarrow aX, Y \rightarrow bXYX, Z \rightarrow aY, Z \rightarrow bXYZ, V \rightarrow aXX, V \rightarrow bZV\})$.

Dans la phase d'initialisation, nous calculons la longueur d'un mot le plus court pouvant être engendré par chaque non-terminal, c.-à-d. $||X|| = 1$, $||Y|| = 2$, $||Z|| = ||V|| = 3$, *et nous fixons un ordre total sur les non-terminaux de la grammaire qui doit être compatible avec ces valeurs, par exemple* $X < Y < Z < V$.

La trace complète de l'exécution de la boucle ***while*** *de l'algorithme est présentée à la Fig. 4.5. Les éléments soulignés dans l'ensemble* Q *sont les paires* (β_1, β_2) *considérées par l'algorithme à chaque itération.*

Valeurs au début de l'itération					Appels de fonction			
Q	$\mathcal{D}(X)$	$\mathcal{D}(Y)$	$\mathcal{D}(Z)$	$\mathcal{D}(V)$	$\mathcal{D}^*(\beta_1)$	$\mathcal{D}^*(\beta_2)$	`First-MP`	*quot*
$\underline{(V,Z)}$	X	Y	Z	V	V	Z	(V,Z)	ϵ
$\underline{(V,Z)}$, (XX,Y), (ZV,XYZ)	X	Y	Z	Z	Z	Z	NIL	—
$\underline{(XX,Y)}$, (ZV,XYZ)	X	Y	Z	Z	XX	Y	(Y,X)	X
$\underline{(XX,Y)}$, (ZV,XYZ), (X,X), (XYX,ZXX)	X	XX	Z	Z	XX	XX	NIL	—
$\underline{(ZV,XYZ)}$, (X,X), (XYX,ZXX)	X	XX	Z	Z	ZZ	$XXXZ$	(Z,X)	Y
$\underline{(ZV,XYZ)}$, (X,X), (XYX,ZXX), (Y,Y), (XYZ,ZXY)	X	XX	XY	Z	$XXXXXX$	$XXXXXX$	NIL	—
$\underline{(X,X)}$, (XYX,ZXX), (Y,Y), (XYZ,ZXY)	X	XX	XY	Z	X	X	NIL	—
$\underline{(XYX,ZXX)}$, (Y,Y), (XYZ,ZXY)	X	XX	XY	Z	$XXXX$	$XXXXX$	ÉCHEC	

FIG. 4.5 – Exemple d'exécution de ÉQUIVALENCEGS démontrant $L_G(V) \neq L_G(Z)$.

Le dernier appel de ***`First-MP`*** *a retourné échec, nous en concluons que les non-terminaux* V *et* Z *ne sont pas équivalents.*

Chapitre 5

Équivalence des grammaires de fonction simple

5.1 Grammaires de fonction simple

Nous présentons dans ce chapitre un algorithme permettant de résoudre le problème d'équivalence des grammaires de fonction simple. Cet algorithme est une généralisation de l'algorithme ÉQUIVALENCEGS présenté au chapitre 4, accomplissant la tâche supplémentaire de comparer les mots produits en sortie pour chaque mot d'entrée engendré par les non-terminaux de la grammaire.

DÉFINITION 5.1. *Une* grammaire de fonction simple *est un 4-uplet* $G = (\Sigma, \Omega, N, P)$ *où :*

- Σ *est un ensemble fini de symboles d'entrée, appelé* alphabet d'entrée*,*
- Ω *est un ensemble fini de symboles de sortie, disjoint de* Σ*, appelé* alphabet de sortie*,*
- N *est un ensemble fini de symboles* non-terminaux*,*
- $P \subset N \times \Sigma \times (\Omega \cup N)^*$ *est un ensemble fini de* règles de productions*,*

tel que pour tous $a \in \Sigma$ *et* $\alpha, \beta \in (\Omega \cup N)^*$ *:*

$$((A, a, \alpha) \in P \quad et \quad (A, a, \beta) \in P) \quad \Longrightarrow \quad \alpha = \beta.$$

Une grammaire de fonction simple telle que $\Omega = \emptyset$ n'est rien d'autre qu'une grammaire simple, c.-à-d. que le domaine d'une fonction simple est un langage simple. Ainsi,

la plupart des définitions et notations concernant les grammaires simples s'étendent naturellement aux grammaires de fonction simple. Nous définissons pour chaque $a \in \Sigma$ la relation $\stackrel{a}{\rightarrow} \subset \Omega^* N(\Omega \cup N)^* \times (\Omega \cup N)^*$:

$$uA\alpha \stackrel{a}{\rightarrow} u\gamma\alpha \quad \text{ssi} \quad (A, a, \gamma) \in P, u \in \Omega^* \text{ et } \alpha \in (\Omega \cup N)^*,$$

et pour tout $w = a_1 \dots a_n$, avec $a_i \in \Sigma$ pour tout $i \in [1, n]$, la relation de *dérivation* $\stackrel{w}{\rightarrow} \subset (\Omega \cup N)^* \times (\Omega \cup N)^*$:

$\alpha_0 \stackrel{w}{\rightarrow} \alpha_n$ ssi :

- $\alpha_0 = \alpha_n$ et $w = \varepsilon$, ou
- $\exists\ \alpha_1, \dots, \alpha_{n-1} \in (\Omega \cup N)^*$ tels que $\alpha_{i-1} \stackrel{a_i}{\rightarrow} \alpha_i$, pour tout $i \in [1, n]$.

Soient $w \in \Sigma^*$ et $\alpha, \beta \in (\Omega \cup N)^*$, nous écrivons :

$$\beta = \texttt{derive}(\alpha, w) \Leftrightarrow \alpha \stackrel{w}{\rightarrow} \beta.$$

S'il n'existe aucun β tel que $\alpha \stackrel{w}{\rightarrow} \beta$, alors $\texttt{derive}(\alpha, w) \stackrel{\texttt{def}}{=} \bot$.

DÉFINITION 5.2. *Soit $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple. Pour tout $\alpha \in (\Omega \cup N)^*$, nous définissons la relation $F_G(\alpha)$ comme suit :*

$$F_G(\alpha) \stackrel{\texttt{def}}{=} \{(w, u) \in \Sigma^* \times \Omega^* \mid u = \texttt{derive}(\alpha, w) \text{ et } u \neq \bot\}.$$

La relation $F_G(\alpha)$ est appelée une fonction simple. *Nous écrivons $F_G(\alpha)(w) = u$ ssi $(w, u) \in F_G(\alpha)$.*

Nous notons par $||\alpha||$ la longueur minimale d'un mot $w \in \Sigma^*$ tel que $F_G(\alpha)(w)$ est défini. Nous notons par $||G||$ la longueur maximale d'un mot le plus court engendré par un non-terminal de la grammaire G, c.-à-d. : $||G|| \stackrel{\texttt{def}}{=} \texttt{max}\{||A|| \mid A \in N\}$. Nous notons par $\texttt{min}\ F_G(A)$ l'unique élément $(w, u) \in F_G(A)$ tel que w est de longueur minimale et lexicographiquement minimal.

Les fonctions simples constituent un monoïde avec l'opérateur de concaténation défini, pour deux fonctions simples f et g, par :

$$fg \stackrel{\texttt{def}}{=} \{(x_1 x_2, y_1 y_2) \mid (x_1, y_1) \in f, (x_2, y_2) \in g\},$$

avec la fonction $\{(\varepsilon,\varepsilon)\}$ comme élément neutre et $\perp \stackrel{\text{def}}{=} \{\}$ agissant comme zéro. Se référer à [8] pour plus de détails concernant les fonctions simples vues comme un monoïde. En général, nous écrivons w et u plutôt que $\{(w,\varepsilon)\}$ et $\{(\varepsilon,u)\}$, avec $w \in \Sigma^*$ et $u \in \Omega^*$, respectivement. En particulier, nous écrivons ε plutôt que $\{(\varepsilon,\varepsilon)\}$.

Soient $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple et $\alpha, \beta \in (\Omega \cup N)^*$. Alors α et β sont dits équivalents dans G, noté $\alpha \equiv_G \beta$, ssi $F_G(\alpha) = F_G(\beta)$. Le *problème d'équivalence des grammaires de fonction simple* est défini comme suit :

ENTRÉE : Une grammaire de fonction simple $G = (\Sigma, \Omega, N, P)$ et deux non-terminaux $A, B \in N$;
SORTIE : VRAI si $A \equiv_G B$, FAUX sinon.

EXEMPLE 5.1. *Soit la grammaire de fonction simple* $G = (\{0,1\}, \{a,b\}, \{S_1, S_2, A_1, A_2\}, P)$ *où :*

$$\begin{array}{lllll} P & = & \{ & S_1 \to 0aS_1A_1b, & S_2 \to 0aS_2A_2, \\ & & & S_1 \to 1, & S_2 \to 1, \\ & & & A_1 \to 1, & A_2 \to 1b \quad \}. \end{array}$$

Alors $S_1 \equiv_G S_2$ *car* $F_G(S_1) = F_G(S_2) = \{(0^n1^{n+1}, a^nb^n) \mid n \geq 0\}$.

5.2 Groupe libre sur l'alphabet Ω

Soit $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple. Dans le cadre de notre algorithme, nous sommes parfois intéressés à représenter le fait que toutes les dérivations d'un mot $\alpha \in (\Omega \cup N)^*$ doivent engendrer une même suite initiale ou finale de symboles de sortie. Par exemple, nous pouvons savoir que la fonction simple définie à partir d'un non-terminal A est la même que celle définie à partir d'un non-terminal B, sauf pour un préfixe $u \in \Omega^*$ qui doit être éliminé de chaque mot de sortie de B. Alors, nous écrivons $uA \equiv_G B$, ou de manière équivalente $A \equiv_G u^{-1}B$.

Nous formalisons cette notation en introduisant un groupe libre engendré à partir des symboles de l'alphabet de sortie Ω. Soit $\overline{\Omega} \stackrel{\text{def}}{=} \{\overline{a} \mid a \in \Omega\}$ tel que pour tout $a \in \Omega$, $\overline{a} \in \overline{\Omega}$ est l'inverse de a, c.-à-d. $\overline{a}a = a\overline{a} = \varepsilon$. L'ensemble $\Omega^{\otimes} \stackrel{\text{def}}{=} (\Omega \cup \overline{\Omega})^*$ muni de l'opération de concaténation constitue un groupe libre, avec ε comme élément neutre. Nous avons $a^{-1} = \overline{a}$ pour tout $a \in \Omega$ et $(xy)^{-1} = y^{-1}x^{-1}$ pour tous $x, y \in \Omega^{\otimes}$. Par

exemple, $(ab\bar{c})^{-1} = c\bar{b}\bar{a}$ et $bc(abc)^{-1} = \bar{a}$. Les notions de dérivation et de fonction $F_G(\alpha)$ s'étendent naturellement pour prendre en compte l'alphabet $\overline{\Omega}$ en les redéfinissant sur la grammaire $G = (\Sigma, \Omega \cup \overline{\Omega}, N, P)$.

Étant donnés deux mots $u, v \in \Omega^{\otimes}$, $u = v$ n'implique pas nécessairement que u et v sont identiques en tant que chaînes de caractères. Par exemple, $ab\bar{b}ba = \bar{b}baba$. Lorsque nous voulons préciser que deux mots $u, v \in \Omega^{\otimes}$ sont des chaînes identiques, nous écrivons $u \overset{\texttt{id}}{=} v$.

Pour chaque mot $u \in \Omega^{\otimes}$, il existe un unique mot $v \in \Omega^{\otimes}$ de longueur minimale tel que $u = v$. Nous notons par $\texttt{reduire}(u)$ le mot de longueur minimale qui est égal à u. Nous disons qu'un mot u est *réduit*, si $\texttt{reduire}(u) \overset{\texttt{id}}{=} u$. Pour tout les mots $u \in \Omega^{\otimes}$, le mot $\texttt{reduire}(u)$ peut être calculé en temps linéaire par rapport à la longueur du mot u : ce mot ne contient aucune occurrence d'un sous-mot aa^{-1} pour tout $a \in \Omega \cup \overline{\Omega}$. Par exemple, $\texttt{reduire}(a\bar{b}a\bar{a}bb\bar{a}b) \overset{\texttt{id}}{=} ab\bar{a}b$. Aussi, $baa\bar{b}$ est un mot réduit car $\texttt{reduire}(baa\bar{b}) \overset{\texttt{id}}{=} baa\bar{b}$.

Un mot réduit $u = a_1 a_2 \dots a_n \in \Omega^{\otimes}$, où $a_i \in \Omega \cup \overline{\Omega}$ pour tout $i \in [1, n]$, est dit *cycliquement réduit* si $a_1 \neq a_n^{-1}$. Lorsque nous désirons mettre l'emphase sur le fait qu'un mot u est cycliquement réduit, nous dénotons u par $\hat{u}$. Pour tout mot réduit $u \in \Omega^{\otimes}$, il existe des mots réduits $u_1, \hat{u}_2 \in \Omega^{\otimes}$ uniques tels que $u \overset{\texttt{id}}{=} u_1 \hat{u}_2 u_1^{-1}$ et $|\hat{u}_2| > 0$. Nous appelons cette représentation de u la *décomposition cyclique* de u. La décomposition cyclique d'un mot réduit u se calcule en temps $O(|u|)$ en trouvant le plus long préfixe de u tel que son inverse est un suffixe de u.

Soit $x \in \Omega^{\otimes}$ et $k \in \mathbb{Z}$. Nous définissons le mot x^k comme suit :

$$x^k \overset{\texttt{def}}{=} \begin{cases} x \cdot x^{k-1} & \text{si } k > 0, \\ \varepsilon & \text{si } k = 0, \\ (x^{-1})^{-k} & \text{si } k < 0. \end{cases}$$

Nous disons qu'un mot $u \in \Omega^{\otimes}$ n'est pas *primitif* s'il existe $x \in \Omega^{\otimes}$ et $k > 1$ tels que $u = x^k$, sinon u est *primitif*. Pour tout $u \in \Omega^{\otimes} \setminus \{\varepsilon\}$, il existe un unique mot réduit $x \in \Omega^{\otimes}$, noté $\texttt{racine}(u)$, et un unique $k > 0$, noté $\texttt{puissance}(u)$, tels que $u = x^k$.

Exemple 5.2. *Soit le mot $u = a\bar{b}abbabbab\bar{a}abb\bar{a} \in \Omega^{\otimes}$. Alors **racine**$(u) = a\bar{b}abbb\bar{a}$ et **puissance**$(u) = 3$.*

LEMME 5.1. *Soit $u \in \Omega^{\otimes}$. Il existe un algorithme permettant de calculer* $\texttt{racine}(u)$ *et* $\texttt{puissance}(u)$ *en temps $O(|u|)$.*

Démonstration. Nous supposons que u est donné sous sa forme réduite. Soient $u_1, \hat{u}_2 \in \Omega^{\otimes}$ tels que $u = u_1 \hat{u}_2 u_1^{-1}$ est la décomposition cyclique de u. Notez que u est une puissance d'un mot primitif $x \in \Omega^{\otimes}$ ssi $\hat{u}_2$ est une puissance d'un mot primitif $y \in \Omega^{\otimes}$ tel que $x = u_1 y u_1^{-1}$. Comme $\hat{u}_2$ est cycliquement réduit, $\texttt{reduire}(y^k) \overset{\texttt{id}}{=} y^k$ pour tout $k \geq 1$ et $\hat{u}_2$ peut être traité comme un mot dans un monoïde libre engendré par $\Omega \cup \overline{\Omega}$. Dans ce contexte, calculer $\texttt{racine}(\hat{u}_2)$ et $\texttt{puissance}(\hat{u}_2)$ peut être effectué en temps linéaire en trouvant les occurrences de $\hat{u}_2$ dans le mot $\hat{u}_2\hat{u}_2$ (voir [7] pour plus de détails). Les valeurs de $\texttt{racine}(u)$ et $\texttt{puissance}(u)$ s'obtiennent ensuite à partir des valeurs de $\texttt{racine}(\hat{u}_2)$ et $\texttt{puissance}(\hat{u}_2)$. □

5.3 Fonction de décomposition

Notre algorithme d'équivalence utilise le concept de fonction de décomposition.

DÉFINITION 5.3. *Soit $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple. Une* fonction de décomposition *est une application $\mathcal{D} : N \to (\Omega \cup \overline{\Omega})^* N (\Omega \cup \overline{\Omega} \cup N)^*$ pour laquelle il existe un ordre total sur les symboles de N, tel que pour tout $A \in N$, $\mathcal{D}(A) = A$ ou $A > B$ pour tous les symboles $B \in N$ présent dans le mot $\mathcal{D}(A)$.*

Une paire $(A, \alpha) \in N \times (\Omega \cup \overline{\Omega})^* N (\Omega \cup \overline{\Omega} \cup N)^*$, telle que $\alpha = A$ ou $A > B$ pour tous les symboles $B \in N$ présents dans le mot α est appelée une *équation de décomposition*.

Nous étendons la définition d'une fonction de décomposition en une fonction $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$, telle que $\mathcal{D}(\varepsilon) \overset{\texttt{def}}{=} \varepsilon$, $\mathcal{D}(a) \overset{\texttt{def}}{=} a$ pour tout $a \in \Omega \cup \overline{\Omega}$ et $\mathcal{D}(A_1 A_2 \ldots A_n) \overset{\texttt{def}}{=} \mathcal{D}(A_1)\mathcal{D}(A_2)\ldots\mathcal{D}(A_n)$, où $A_i \in (\Omega \cup \overline{\Omega} \cup N)$ pour tout $i \in [1, n]$. Une fonction de décomposition $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$ est donc entièrement définie à partir des valeurs de $\mathcal{D}(A)$ pour chaque $A \in N$, mais son application s'étend aux mots sur $(\Omega \cup \overline{\Omega} \cup N)^*$.

Nous définissons $\mathcal{D}^k$, pour tout $k \geq 1$, comme suit :

$$\mathcal{D}^k \overset{\texttt{def}}{=} \begin{cases} \mathcal{D} & \text{si } k = 1, \\ \mathcal{D} \circ \mathcal{D}^{k-1} & \text{si } k > 1 \end{cases}$$

où $\circ$ dénote la composition de fonctions. Nous notons $\mathcal{D}^{|N|}$ par $\mathcal{D}^*$ car $\mathcal{D}^{|N|} = \mathcal{D}^{|N|+1}$.

Proposition 5.1. *Soient $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple et $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$ une fonction de décomposition. Si $F_G(\alpha) = F_G(\mathcal{D}(\alpha))$ pour tout $\alpha \in (\Omega \cup \overline{\Omega} \cup N)^*$, alors $F_G(\alpha) = F_G(\mathcal{D}^*(\alpha))$ pour tout $\alpha \in (\Omega \cup \overline{\Omega} \cup N)^*$.*

Démonstration. Le résultat est une conséquence directe de la transitivité de l'égalité. □

Dans le cadre de notre algorithme d'équivalence, nous cherchons à construire une fonction de décomposition telle que $F_G(A) = F_G(\mathcal{D}(A))$ pour tout $A \in N$. Son rôle est de transformer les mots sur l'alphabet $\Omega \cup \overline{\Omega} \cup N$ en des mots équivalents sur l'alphabet $\Omega \cup \overline{\Omega} \cup N'$, où $N' = \{A \in N \mid \mathcal{D}(A) = A\} \subseteq N$. Éventuellement, s'il n'existe qu'un seul $A \in N$ tel que $\mathcal{D}(A) = A$, alors pour tous les mots $\alpha \in (\Omega \cup \overline{\Omega} \cup N)^*$, $\mathcal{D}^*(\alpha) \in (\Omega \cup \overline{\Omega} \cup \{A\})^*$, c.-à-d. que le seul non-terminal pouvant apparaître dans un mot $\mathcal{D}^*(\alpha)$ est A.

5.4 Relation de conjugaison

Notre algorithme d'équivalence manipule également un autre type d'équations, appelées *équations de conjugaison*. Soit $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple. Une équation de conjugaison est une équation du type $u_1X \equiv_G Xu_2$ où $u_1, u_2 \in \Omega^\otimes$ et $X \in N$, notée par un triplet (u_1, X, u_2). Nous appelons *relation de conjugaison* un ensemble $\mathcal{C}$ d'équations de conjugaisons.

Le lemme 5.3 décrit une propriété des équations de conjugaison dans les groupes libres. Avant de présenter cette propriété, nous énonçons un résultat connu concernant les équations de commutativité dans les groupes libres.

Lemme 5.2. *Soit $u, v \in \Omega^\otimes$ tels que $uv = vu$. Alors il existe $x \in \Omega^\otimes$ et $k_1, k_2 \in \mathbb{Z}$ tels que $u = x^{k_1}$ et $v = x^{k_2}$.*

Démonstration. Voir [16, Chapitre 1, Lemme 2]. □

LEMME 5.3. *Soit* $X \subseteq \Omega^{\otimes}$ *tel que* $|X| \geq 2$ *et soient* $u_1, v_1 \in \Omega^{\otimes} \setminus \{\varepsilon\}$ *tels que* $u_1 w = w v_1$ *pour tout* $w \in X$*. Alors, pour chaque* $u_2, v_2 \in \Omega^{\otimes}$*,* $u_2 w = w v_2$ *pour tout* $w \in X$ *ssi*

– $\boldsymbol{puissance}(u_1) = \boldsymbol{puissance}(v_1)$,
– $\boldsymbol{puissance}(u_2) = \boldsymbol{puissance}(v_2)$,
– $(\boldsymbol{racine}(u_1), \boldsymbol{racine}(v_1)) \in \{\, (\boldsymbol{racine}(u_2), \boldsymbol{racine}(v_2)),$
$(\boldsymbol{racine}(u_2)^{-1}, \boldsymbol{racine}(v_2)^{-1}) \,\}$.

Démonstration.
$\Longrightarrow$: Soit $w_1, w_2 \in X$ tels que $w_1 \neq w_2$. Alors :

$$\begin{aligned} u_1 w_1 = w_1 v_1 \quad \text{et} \quad u_1 w_2 = w_2 v_1 \quad &\Longrightarrow \quad w_1^{-1} u_1 w_1 = v_1 \quad \text{et} \quad w_2^{-1} u_1 w_2 = v_1 \\ &\Longrightarrow \quad w_1^{-1} u_1 w_1 = w_2^{-1} u_1 w_2 \\ &\Longrightarrow \quad w_2 w_1^{-1} u_1 = u_1 w_2 w_1^{-1} \end{aligned}$$

D'après le lemme 5.2, il existe $x \in \Omega^{\otimes}$ et $k_1, k_2 \in \mathbb{Z}$ tels que $w_2 w_1^{-1} = x^{k_1}$ et $u_1 = x^{k_2}$. Nous pouvons supposer, sans perte de généralité, que x est primitif (sinon il suffit de trouver la racine de x et de modifier les puissances en conséquence). Alors :

$$v_1 = w_1^{-1} x^{k_2} w_1 = (w_1^{-1} x w_1)^{k_2} \quad \text{et} \quad w_2 = x^{k_1} w_1.$$

Par hypothèse, nous avons :

$$\begin{aligned} & u_2 w_1 = w_1 v_2 && \Longrightarrow && v_2 = w_1^{-1} u_2 w_1 \\ \text{et} \quad & u_2 w_2 = w_2 v_2 && \Longrightarrow && u_2 x^{k_1} w_1 = x^{k_1} w_1 v_2 \\ & && \Longrightarrow && u_2 x^{k_1} w_1 = x^{k_1} w_1 w_1^{-1} u_2 w_1 \\ & && \Longrightarrow && u_2 x^{k_1} = x^{k_1} u_2 \end{aligned}$$

D'après le lemme 5.2, et comme x est primitif, il existe $k_3 \in \mathbb{Z}$ tel que $u_2 = x^{k_3}$. Donc, $v_2 = w_1^{-1} x^{k_3} w_1 = (w_1^{-1} x w_1)^{k_3}$.

En s'assurant d'avoir k_2 et k_3 positifs, inversant les racines au besoin, nous trouvons que les puissances et racines des mots u_1, u_2, v_1, v_2 respectent l'énoncé du lemme.

$\Longleftarrow$: Soient $r_1, r_2 \in \Omega^{\otimes}$ des mots réduits et $k_1, k_2 > 0$ tels que :

- $\texttt{puissance}(u_1) = \texttt{puissance}(v_1) = k_1$,
- $\texttt{puissance}(u_2) = \texttt{puissance}(v_2) = k_2$,
- $(\texttt{racine}(u_1), \texttt{racine}(v_1)) = (r_1, r_2) \in \{ (\texttt{racine}(u_2), \texttt{racine}(v_2)), (\texttt{racine}(u_2)^{-1}, \texttt{racine}(v_2)^{-1}) \}$.

Soit $w \in X$. Par hypothèse :

$$\begin{aligned} u_1 w = w v_1 \quad & \Longrightarrow \quad r_1^{k_1} w = w r_2^{k_1} \\ & \Longrightarrow \quad r_1^{k_1} = w r_2^{k_1} w^{-1} \\ & \Longrightarrow \quad r_1^{k_1} = (w r_2 w^{-1})^{k_1}. \end{aligned}$$

Soit $z \in \Omega^{\otimes}$ tel que $\texttt{reduire}(w r_2 w^{-1}) \overset{\texttt{id}}{=} z$. Donc $r_1^{k_1} = z^{k_1}$. Soient $a_1, \hat{a}_2, b_1, \hat{b}_2 \in \Omega^{\otimes}$ tels que $r_1 = a_1 \hat{a}_2 a_1^{-1}$ et $z = b_1 \hat{b}_2 b_1^{-1}$ sont les décompositions cycliques de r_1 et z, respectivement. Donc :

$$(a_1 \hat{a}_2 a_1^{-1})^{k_1} = (b_1 \hat{b}_2 b_1^{-1})^{k_1} \quad \Longrightarrow \quad a_1 \hat{a}_2^{k_1} a_1^{-1} \overset{\texttt{id}}{=} b_1 \hat{b}_2^{k_1} b_1^{-1}.$$

Il y a 3 cas possibles selon les longueurs relatives de $\hat{a}_2$ et $\hat{b}_2$.

1. Supposons que $|\hat{a}_2| = |\hat{b}_2|$. Alors, $a_1 \overset{\texttt{id}}{=} b_1$ et $\hat{a}_2 \overset{\texttt{id}}{=} \hat{b}_2$, donc $r_1 = z$.
2. Supposons que $|\hat{a}_2| < |\hat{b}_2|$. Alors $|a_1| > |b_1|$, donc b_1 est un préfixe de a_1 et il existe $y \in \Omega^{\otimes}$ tel que $a_1 \overset{\texttt{id}}{=} b_1 y$ et $|y| > 0$. Ainsi,

$$\begin{aligned} a_1 \hat{a}_2^{k_1} a_1^{-1} \overset{\texttt{id}}{=} b_1 \hat{b}_2^{k_1} b_1^{-1} \quad & \Longrightarrow \quad b_1 y \hat{a}_2^{k_1} y^{-1} b_1^{-1} \overset{\texttt{id}}{=} b_1 \hat{b}_2^{k_1} b_1^{-1} \\ & \Longrightarrow \quad y \hat{a}_2^{k_1} y^{-1} \overset{\texttt{id}}{=} \hat{b}_2^{k_1}. \end{aligned}$$

   Le premier et le dernier symbole de $\hat{b}_2$ sont donc inverses l'un de l'autre, ce qui contredit le fait que $\hat{b}_2$ est un mot cycliquement réduit.
3. Supposons que $|\hat{a}_2| > |\hat{b}_2|$. Ce cas est symétrique au cas 2.

Le seul cas possible est le cas 1, donc $r_1 = z = w r_2 w^{-1}$.

Par induction sur k, nous pouvons démontrer que $r_1^k = (wr_2w^{-1})^k$ pour tout $k \in \mathbb{Z}$. En particulier,

$$\begin{array}{rlcl} & r_1^{k_2} = (wr_2w^{-1})^{k_2} & \Longrightarrow & r_1^{k_2} = wr_2^{k_2}w^{-1} \\ & & \Longrightarrow & r_1^{k_2}w = wr_2^{k_2} \\ et & r_1^{-k_2} = (wr_2w^{-1})^{-k_2} & \Longrightarrow & r_1^{-k_2} = wr_2^{-k_2}w^{-1} \\ & & \Longrightarrow & (r_1^{-1})^{k_2} = w(r_2^{-1})^{k_2}w^{-1} \\ & & \Longrightarrow & (r_1^{-1})^{k_2}w = w(r_2^{-1})^{k_2} \end{array}$$

donc $u_2w = wu_2$. □

D'après le lemme 5.3, il est suffisant de conserver dans $\mathcal{C}$ au plus une équation (u_1, A, u_2) pour chaque non-terminal $A \in N$. Par simplicité, les équations que nous conserverons dans $\mathcal{C}$ seront telles que u_1 et u_2 sont réduits et primitifs.

Le lemme 5.3 s'applique à un non-terminal seulement si celui-ci engendre au moins deux mots en sortie. Il est donc nécessaire de pouvoir identifier l'ensemble $\texttt{SingleOut}(G)$ des non-terminaux de G ne produisant qu'une valeur de sortie, c.-à-d. :

$$\texttt{SingleOut}(G) \stackrel{\text{def}}{=} \{A \in N \mid \forall (w, u), (w', u') \in F_G(A), u = u'\}.$$

Lemme 5.4. *Soit $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple. Il est possible de calculer $\texttt{SingleOut}(G)$ en temps $O(n^3 \cdot ||G||)$, où $n = |G|$.*

Démonstration. Nous associons à chaque $A \in N$ un ensemble $\texttt{sortie}(A)$, initialement vide. Nous ajouterons un mot de sortie w à cet ensemble dès que nous trouverons que w peut être engendré en sortie par A. Nous associons également l'ensemble $\texttt{sortie}(a)$ à chaque $a \in \Omega$, initialisé à $\texttt{sortie}(a) = \{a\}$.

Nous supposons que chaque non-terminal de la grammaire engendre au moins un mot. Nous parcourons itérativement l'ensemble des règles de production de la grammaire dans le but d'ajouter jusqu'à deux mots de sortie à chaque ensemble $\texttt{sortie}(A)$. Pour chaque règle de production $(A, a, A_1A_2 \ldots A_n) \in P$ où $A \in N$, $a \in \Sigma$ et $A_1, A_2, \ldots, A_n \in \Omega \cup N$, si $|\texttt{sortie}(A_i)| = 1$ pour chaque $i \in [1, n]$, nous ajoutons le mot $w_1w_2 \ldots w_n$ à l'ensemble $\texttt{sortie}(A)$, c.-à-d. $\texttt{sortie}(A) := \texttt{sortie}(A) \cup \{w_1w_2 \ldots w_n\}$ où w_i est l'unique élément de $\texttt{sortie}(A_i)$. Nous éliminons un non-terminal A de la grammaire dès que $|\texttt{sortie}(A)| = 2$, et nous éliminons, récursivement, tous les non-terminaux B pour

lesquels il existe $(B, b, \beta) \in P$ tel que β contient un symbole éliminé. Nous terminons l'exécution de cette procédure lorsque que les ensembles $\texttt{sortie}(A)$ se stabilisent, c.-à-d. dès qu'une itération entière s'effectue sans apporter de modification aux ensembles $\texttt{sortie}(A)$, pour chaque A qui n'a pas été éliminé de la grammaire. L'ensemble des symboles qui sont toujours présents dans la grammaire à la fin de l'exécution correspond à $\texttt{SingleOut}(G)$.

Au plus $|N| \leq |G|$ itérations sont effectuées, car nous débutons une nouvelle itération seulement si nous ajoutons lors de l'itération précédente un mot à un ensemble $\texttt{sortie}(A)$ qui ne contenait auparavant aucun élément. À chaque itération, nous parcourons l'ensemble de la grammaire dont la taille est $|G|$ et engendrons un mot pour chaque règle de production de la grammaire. La longueur des mots engendrés est $O(n \cdot ||G||)$ et au plus deux mots différents sont engendrés pour chaque non-terminal. Chaque non-terminal est éliminé au plus une fois, et pour chaque symbole éliminé, nous parcourons la grammaire une seule fois afin d'identifier les nouveaux symboles à éliminer. Cette procédure s'exécute donc en temps $O(n^3 \cdot ||G||)$. □

5.5 Quotient d'une fonction simple

Notre algorithme d'équivalence manipule des paires de mots $\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$ pour lesquelles nous cherchons à démontrer l'équivalence. Si α_1 ou α_2 ne contient que des symboles de l'alphabet de sortie, l'équivalence de α_1 et α_2 peut être vérifiée directement.

Sinon, les mots α_1 et α_2 sont sous la forme $u_1 A \alpha_1'$ et $u_2 B \alpha_2'$, respectivement, où $u_1, u_2 \in \Omega^\otimes$, $A, B \in N$ et $\alpha_1', \alpha_2' \in (\Omega \cup \overline{\Omega} \cup N)^*$. Pour chaque mot $\alpha = uA\alpha'$ où $u \in \Omega^\otimes$, $A \in N$ et $\alpha' \in (\Omega \cup \overline{\Omega} \cup N)^*$, nous notons u, A et α' par $\texttt{OutPref}(\alpha)$, $\texttt{PremierNT}(\alpha)$ et $\texttt{Suffixe}(\alpha)$, respectivement. En supposant $||A|| \geq ||B||$ et $F_G(\alpha_1) = F_G(\alpha_2)$, nous savons que $F_G(u_2 B)$ doit être un diviseur gauche de $F_G(u_1 A)$, c.-à-d. qu'il doit exister $\gamma \in (\Omega \cup \overline{\Omega} \cup N)^*$ tel que $F_G(u_1 A) = F_G(u_2 B) F_G(\gamma)$. Nous pouvons obtenir le mot γ en *divisant* $u_1 A$ à gauche par n'importe quel élément $(w_B, u_2 u_B) \in F_G(u_2 B)$. Pour ce faire, nous dérivons *explicitement* le mot d'entrée w_B à partir de $u_1 A$ et divisons *implicitement* $u_1 A$ par le mot de sortie $u_2 u_B$ en concaténant l'inverse de $u_2 u_B$ à gauche de $u_1 A$, c.-à-d. :

$$\texttt{quot}(u_1 A, u_2 B) \stackrel{\texttt{def}}{=} u_B^{-1} u_2^{-1} \texttt{derive}(u_1 A, w_B),$$

où $u_1, u_2 \in \Omega^{\otimes}$, $A, B \in N$, $||A|| \geq ||B||$ et $(w_B, u_B) = \texttt{min}\ F_G(B)$.

Proposition 5.2. *Soient $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple, $u_1, u_2 \in \Omega^{\otimes}$, $A, B \in N$ tels que $||A|| \geq ||B||$ et $\alpha, \beta \in (\Omega \cup \overline{\Omega} \cup N)^*$. Alors $F_G(u_1 A\alpha) = F_G(u_2 B\beta)$ ssi $F_G(u_1 A) = F_G(u_2 B\gamma)$ et $F_G(\gamma\alpha) = F_G(\beta)$, où $\gamma = \texttt{quot}(u_1 A, u_2 B)$.*

Démonstration. Le résultat est une conséquence du fait que le monoïde des fonctions simples est simplifiable à gauche. Deux cas sont possibles :

- γ est tel que $F_G(u_1 A) = F_G(u_2 B\gamma)$, c.à-d. $F_G(u_2 B)$ est un diviseur gauche de $F_G(u_1 A)$. Dans ce cas la validité de l'énoncé est directe.
- Si $F_G(u_1 A) \neq F_G(u_2 B\gamma)$, alors $(F_G(u_2 B))^{-1} F_G(u_1 A)$ n'est pas défini, donc $F_G(u_1 A\alpha)$ ne peut pas être égal à $F_G(u_2 B\beta)$.

□

L'opération `quot` permet de remplacer une équation $u_1 A\alpha'_1 \equiv_G u_2 B\alpha'_2$ par deux équations $u_1 A \equiv_G u_2 B\gamma$ et $\gamma\alpha'_1 \equiv_G \alpha'_2$, où $\gamma = \texttt{quot}(u_1 A, u_2 B)$. Si $A \neq B$, l'équation $u_1 A \equiv_G u_2 B\gamma$ correspond à une équation de décomposition $\mathcal{D}(A) = u_1^{-1} u_2 B\gamma$. Si $A = B$, alors γ ne contient que des symboles de sortie et l'équation $u_1 A \equiv_G u_2 B\gamma$ correspond plutôt à une *équation de conjugaison* $u_2^{-1} u_1 A \equiv_G A\gamma$.

5.6 Procédure `extraire_equation`

Nous présentons une procédure, appelée `extraire_equation`, qui est utilisée par l'algorithme d'équivalence des grammaires de fonction simple que nous présentons à la section 5.7. Cette procédure remplie un rôle similaire à celui de la procédure `First-MP` utilisée dans l'algorithme d'équivalence des grammaires simples présenté au chapitre 4. Elle est utilisée par notre algorithme d'équivalence des grammaires de fonction simple pour découvrir les équations de conjugaison et les équations de décomposition qui doivent être ajoutées, respectivement, à la relation de conjugaison et à la fonction de décomposition que l'algorithme construit.

Soit $G = (\Sigma, \Omega, N, P)$ une grammaire de fonction simple. La procédure reçoit en entrée des mots $\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$, une relation de conjugaison $\mathcal{C} \subset \Omega^{\otimes} \times N \times \Omega^{\otimes}$, et une fonction de décomposition $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \rightarrow (\Omega \cup \overline{\Omega} \cup N)^*$. Elle produit en sortie l'une

des quatre valeurs suivantes : NIL, ÉCHEC, une équation de conjugaison (u_1, A, u_2) ou une équation de décomposition (A, α). Soit $E(\mathcal{C}, \mathcal{D}) \stackrel{\mathsf{def}}{=} \{(v_1 A, A v_2) \mid (v_1, A, v_2) \in \mathcal{C}\} \cup \mathcal{D}$.

La relation entre les valeurs d'entrée de la procédure et les valeurs qui sont produites en sortie respecte les propriétés suivantes :

1. Si $F_G(\beta_1) = F_G(\beta_2)$ pour tout $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$ et $F_G(\alpha_1) = F_G(\alpha_2)$, alors :
   (a) `extraire_equation` ne retourne pas ÉCHEC,
   (b) si `extraire_equation` retourne une équation de conjugaison (u_1, A, u_2), alors $F_G(u_1 A) = F_G(A u_2)$,
   (c) si `extraire_equation` retourne une équation de décomposition (A, α), alors $F_G(A) = F_G(\alpha)$,
2. Si $F_G(\beta_1) = F_G(\beta_2)$ pour tout $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$, alors `extraire_equation` retourne NIL seulement si $F_G(\alpha_1) = F_G(\alpha_2)$.

La procédure `extraire_equation` que nous utilisons dans notre algorithme d'équivalence des grammaires de fonction simple est présentée à la Fig 5.1. Tous les mots manipulés par cette procédure sont en tout temps conservés sous leur forme réduite. La procédure fonctionne en itérant une boucle `while`. Si α_1 ou α_2 est égal à $\perp$ au début de l'itération, la procédure retourne NIL ou ÉCHEC, selon que $\alpha_1 = \alpha_2$ ou $\alpha_1 \neq \alpha_2$. Sinon, nous décomposons les mots α_1 et α_2 avec $\mathcal{D}$ et nous éliminons le préfixe commun à $\mathcal{D}^*(\alpha_1)$ et $\mathcal{D}^*(\alpha_2)$. Soient α_1, α_2 les mots ainsi obtenus. Si α_1 et α_2 sont vides, la procédure retourne NIL. Sinon, si α_1 ou α_2 ne contient que des symboles de l'alphabet de sortie, la procédure retourne ÉCHEC.

Sinon $\alpha_1 = u_1 A \alpha_1'$ et $\alpha_2 = u_2 B \alpha_2'$, où $u_1, u_2 \in \Omega^{\otimes}$, $A, B \in N$ et $\alpha_1', \alpha_2' \in (\Omega \cup \overline{\Omega} \cup N)^*$. Nous procédons à la division de $u_1 A \alpha_1'$ et $u_2 B \alpha_2'$ en calculant $\gamma = \texttt{quot}(u_1 A, u_2 B)$, de manière à obtenir deux paires de mots $(u_1 A, u_2 B \gamma)$ et $(\gamma \alpha_1', \alpha_2')$. Si $\gamma = \perp$, la procédure retourne ÉCHEC.

Sinon, il y a deux cas selon les valeurs relatives de A et B :

1. Si $A = B$, et $A \in \texttt{SingleOut}(G)$, nous savons que $u_1 A \equiv_G u_2 B \gamma$. Nous assignons $\alpha_1 := \gamma \alpha_1'$ et $\alpha_2 := \alpha_2'$ et débutons une nouvelle itération. Si $A \notin \texttt{SingleOut}(G)$, nous vérifions s'il existe déjà une équation de conjugaison pour le symbole A dans la relation de conjugaison $\mathcal{C}$. Si oui, nous vérifions que les racines et puissances de

$u_2^{-1}u_1$ et γ sont compatibles avec les valeurs présentes dans la relation de conjugaison $\mathcal{C}$. Si les valeurs sont compatibles, nous assignons $\alpha_1 := \gamma\alpha'_1$ et $\alpha_2 := \alpha'_2$ et débutons une nouvelle itération. Sinon nous avons trouvé une preuve que les non-terminaux reçus en entrée ne sont pas équivalents et la procédure retourne ÉCHEC. S'il n'y a pas déjà une équation de conjugaison dans $\mathcal{C}$ pour le symbole A, la procédure retourne l'équation de conjugaison $(u_2^{-1}u_1, A, \gamma)$.

2. Si $A \neq B$, la procédure retourne l'équation de décomposition $(A, u_1^{-1}u_2B\gamma)$.

Procédure `extraire_equation`$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D})$ $\{\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega} \cup N)^*\}$
  while(TRUE) **do**
    If $\alpha_1 = \bot$ et $\alpha_2 = \bot$ **then return** NIL ;
    If $\alpha_1 = \bot$ ou $\alpha_2 = \bot$ **then return** ÉCHEC ;
    $\alpha_1 := \mathcal{D}^*(\alpha_1)$, $\alpha_2 := \mathcal{D}^*(\alpha_2)$ — *Décomposition de α_1 et α_2 par $\mathcal{D}$.*
    Simplifier (α_1, α_2) en éliminant le préfixe commun.
    If $\alpha_1 = \alpha_2 = \varepsilon$ **then return** NIL ;
    If α_1 ou α_2 est élément de $\Omega^{\otimes}$ **then return** ÉCHEC ;

      — **Commentaire** *: À cette étape α_1 et α_2 contiennent au moins un non-terminal, c.-à-d., $\alpha_1 = u_1A\alpha'_1$ et $\alpha_2 = u_2B\alpha'_2$, et ils diffèrent syntaxiquement sur le premier symbole.*

    $u_1 :=$ `OutPref`(α_1), $A :=$ `PremierNT`(α_1), $\alpha'_1 :=$ `Suffixe`(α_1)
    $u_2 :=$ `OutPref`(α_2), $B :=$ `PremierNT`(α_2), $\alpha'_2 :=$ `Suffixe`(α_2)
      — **Commentaire** *: Sans perte de généralité, nous supposons $A \geq B$.*
    $\gamma :=$ `quot`(u_1A, u_2B)
    If $\gamma = \bot$ **then return** ÉCHEC ;
    If $A = B$ **then** :
      If $A \in$ `SingleOut`(G) **then** $\alpha_1 := \gamma\alpha'_1$, $\alpha_2 := \alpha'_2$ et débuter une nouvelle itération.
      If `puissance`$(u_1^{-1}u_2) \neq$ `puissance`(γ) **then return** ÉCHEC ;
      $x :=$ `racine`$(u_1^{-1}u_2)$, $y :=$ `racine`(γ) ;
        — **Commentaire** *: L'équation $u_1A\gamma = u_2A$ correspond à l'équation de conjugaison $u_1^{-1}u_2A = A\gamma$, et, d'après le lemme 5.3, à $xA \stackrel{\mathtt{id}}{=} Ay$.*

      If (r_1A, Ar_2) est élément de $\mathcal{C}$ **then**
        If $(x = r_1$ et $y = r_2)$ ou $(x = r_1^{-1}$ et $y = r_2^{-1})$ **then**
          $\alpha_1 := \gamma\alpha'_1$, $\alpha_2 := \alpha'_2$ et débuter une nouvelle itération
        else return ÉCHEC.
      return (x, A, y).
    else return $(A, u_1^{-1}u_2B\gamma)$.
  end {de la boucle while}

FIG. 5.1 – Procédure `extraire_equation`.

5.6.1 Preuves de terminaison et d'exactitude

Nous démontrons que la procédure `extraire_equation` se termine toujours et qu'elle respecte les deux propriétés énoncées.

LEMME 5.5. *Le nombre d'itérations de la boucle* ***while*** *de la procédure* `extraire_equation` *est* $O(min\{||\alpha_1||, ||\alpha_2||\})$.

Démonstration. À chaque itération de la procédure `extraire_equation`, $||\alpha_1||$ et $||\alpha_2||$ diminuent, donc après au plus $\texttt{min}\{||\alpha_1||, ||\alpha_2||\}$ itérations, $||\alpha_1|| = 0$ ou $||\alpha_2|| = 0$, c.-à-d. α_1 ou α_2 ne contient que des symboles de sortie, et la procédure se termine. □

LEMME 5.6. *Soient une grammaire de fonction simple* $G = (\Sigma, \Omega, N, P)$, *une relation de conjugaison* $\mathcal{C} \subset \Omega^{\otimes} \times N \times \Omega^{\otimes}$, *une fonction de décomposition* $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$ *et* $\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$, *les paramètres d'entrée d'un appel de la procédure* `extraire_equation`. *Si* $F_G(\beta_1) = F_G(\beta_2)$ *pour tout* $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$ *et* $F_G(\alpha_1) = F_G(\alpha_2)$, *alors :*

1. `extraire_equation` *ne retourne pas* ÉCHEC,
2. *si* `extraire_equation` *retourne une équation de conjugaison* (u_1, A, u_2), *alors* $F_G(u_1A) = F_G(Au_2)$,
3. *si* `extraire_equation` *retourne une équation de décomposition* (A, α), *alors* $F_G(A) = F_G(\alpha)$.

Démonstration. Supposons qu'au début d'une itération de la procédure `extraire_equation` :

1. $F_G(\alpha_1) = F_G(\alpha_2)$, et
2. $F_G(\beta_1) = F_G(\beta_2)$ pour tout $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$.

En traçant une itération de la procédure, nous allons démontrer que si une valeur est retournée par la procédure au cours de l'itération, celle-ci respecte les trois conditions de l'énoncé du lemme et que si aucune valeur n'est retournée, alors $F_G(\alpha_1) = F_G(\alpha_2)$ au début de l'itération suivante.

1. Si $(\alpha_1, \alpha_2) = (\perp, \perp)$, la procédure retourne NIL.
2. À cette étape, $\alpha_1 \neq \perp$ et $\alpha_2 \neq \perp$ car $F_G(\alpha_1) = F_G(\alpha_2)$: ÉCHEC n'est pas retourné.
3. D'après la proposition 5.1, $F_G(\alpha_1) = F_G(\mathcal{D}^*(\alpha_1))$ et $F_G(\alpha_2) = F_G(\mathcal{D}^*(\alpha_2))$ donc $F_G(\mathcal{D}^*(\alpha_1)) = F_G(\mathcal{D}^*(\alpha_2))$.
4. L'élimination du préfixe commun préserve l'équivalence pour les mots résultants.

5. Si $\alpha_1 = \alpha_2 = \varepsilon$, la procédure retourne NIL.
6. Comme $F_G(\alpha_1) = F_G(\alpha_2)$, α_1 et α_2 doivent contenir au moins un symbole variable rendu à ce point. La valeur ÉCHEC n'est donc pas retournée, $\alpha_1 = u_1 A \alpha_1'$ et $\alpha_2 = u_2 B \alpha_2'$.
7. D'après la proposition 5.2, suite à l'application de l'opération $\texttt{quot}(u_1 A, u_2 B) = \gamma$, $F_G(u_1 A) = F_G(u_2 B \gamma)$ et $F_G(\alpha_1' \gamma) = F_G(\alpha_2')$, et $\gamma \neq \bot$.
8. Si $A = B$:
   (a) Si $A \in \texttt{SingleOut}(G)$, l'itération se termine et l'invariant est respecté car $F_G(\gamma \alpha_1') = F_G(\alpha_2')$.
   (b) Sinon, comme $F_G(u_2^{-1} u_1 A) = F_G(A\gamma)$, ÉCHEC n'est pas retourné car nous avons $\texttt{puissance}(u_2^{-1} u_1) = \texttt{puissance}(\gamma)$ d'après le lemme 5.3.
   (c) S'il existe déjà une paire $(r_1 A, A r_2) \in \mathcal{C}$, alors d'après le lemme 5.3 et comme $F_G(u_2^{-1} u_1 A) = F_G(A\gamma)$ et $F_G(r_1 A) = F_G(A r_2)$, $(x, y) \in \{(r_1, r_2), (r_1^{-1}, r_2^{-1})\}$. Donc ÉCHEC n'est pas retourné, l'itération se termine et l'invariant est respecté car $F_G(\gamma \alpha_1') = F_G(\alpha_2')$.
   (d) Sinon, l'algorithme retourne $(u_2^{-1} u_1, A, \gamma)$, et $F_G(u_2^{-1} u_1 A) = F_G(A\gamma)$.
9. Sinon, l'algorithme retourne $(A, u_1^{-1} u_2 B \gamma)$, et $F_G(A) = F_G(u_1^{-1} u_2 B \gamma)$.

Comme la procédure `extraire_equation` se termine toujours, et que la validité des équations est un invariant à chaque itération, la procédure doit éventuellement retourner NIL, une équation de conjugaison $(u_2^{-1} u_1, A, \gamma)$ telle que $F_G(u_2^{-1} u_1 A) = F_G(A\gamma)$ ou une équation de décomposition $(A, u_1^{-1} u_2 B \gamma)$ telle que $F_G(A) = F_G(u_1^{-1} u_2 B \gamma)$. □

Pour tout $n \geq 0$ et $\alpha \in (\Omega \cup \overline{\Omega} \cup N)^*$, nous définissons :

$$F_n(\alpha) \stackrel{\text{def}}{=} \{(w, u) \in F_G(\alpha) \mid |w| \leq n\}.$$

Nous démontrons le résultat suivant, qui est une condition plus forte que la propriété 2 de la procédure `extraire_equation`.

LEMME 5.7. *Soient une grammaire de fonction simple $G = (\Sigma, \Omega, N, P)$, une relation de conjugaison $\mathcal{C} \subset \Omega^{\otimes} \times N \times \Omega^{\otimes}$, une fonction de décomposition $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$ et $\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$, les paramètres d'entrée d'un appel de la procédure* ***`extraire_equation`***. *Soit $n \geq 0$. Si $F_n(\beta_1) = F_n(\beta_2)$ pour tout $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$ et* ***`extraire_equation`***$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D}) =$ NIL, *alors $F_n(\alpha_1) = F_n(\alpha_2)$.*

Démonstration. Par induction sur $||\alpha_1||$. Si $||\alpha_1|| = 0$, alors α_1 ne contient que des symboles de sortie. Comme `extraire_equation` retourne NIL, $\mathcal{D}^*(\alpha_1) = \mathcal{D}^*(\alpha_2)$ donc $F_n(\alpha_1) = F_n(\alpha_2)$.

Supposons que $F_n(\beta_1) = F_n(\beta_2)$ pour tous $\beta_1, \beta_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$ tels que :

1. $||\beta_1|| < k$ et
2. `extraire_equation`$(\beta_1, \beta_2, \mathcal{C}, \mathcal{D}) =$ NIL.

Soient $\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$ tels que $||\alpha_1|| = k$ et `extraire_equation`$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D}) =$ NIL.

Comme $F_n(\alpha) = F_n(\mathcal{D}(\alpha))$ pour tout $\alpha \in (\Omega \cup \overline{\Omega} \cup N)^*$, nous avons $F_n(\alpha_1) = F_n(\mathcal{D}^*(\alpha_1))$ et $F_n(\alpha_2) = F_n(\mathcal{D}^*(\alpha_2))$. Soient α_1, α_2 les mots résultant de l'élimination du préfixe commun des mots décomposés. Si $\alpha_1 = \alpha_2 = \varepsilon$, le résultat est immédiat. Sinon, $\alpha_1 = u_1 A \alpha_1'$ et $\alpha_2 = u_2 B \alpha_2'$, car nous supposons que la procédure retourne NIL, et $A = B$, pour la même raison. Si $A \in$ `SingleOut`(G), alors $F_n(u_1 A) = F_n(u_2 B \gamma)$, d'après la définition du quotient. Par induction, $F_n(\gamma \alpha_1') = F_n(\alpha_2')$ car $||\gamma \alpha_1'|| < k$ et débuter une nouvelle itération équivaut à un appel récursif de la procédure, et nous savons par hypothèse que la procédure retourne NIL, donc $F_n(\alpha_1) = F_n(\alpha_2)$ dans ce cas.

Sinon, comme la procédure retourne NIL et $F_n(v_1 B) = F_n(B v_2)$ pour tout $(v_1, B, v_2) \in \mathcal{C}$, il découle du lemme 5.3 que $F_n(u_1 A) = F_n(A u_2)$, et par induction, $F_n(\gamma \alpha_1') = F_n(\alpha_2')$, donc $F_n(\alpha_1) = F_n(\alpha_2)$. □

PROPOSITION 5.3. *La procédure* ***`extraire_equation`*** *retourne une équation de conjugaison (u_1, A, u_2) seulement s'il n'existe aucun $v_1, v_2 \in \Omega^{\otimes}$ tels que $(v_1, A, v_2) \in \mathcal{C}$ et retourne une équation de décomposition (A, α) seulement si $\mathcal{D}(A) = A$ et $\alpha \neq A$.*

Démonstration. La procédure vérifie qu'il n'existe aucun $v_1, v_2 \in \Omega^{\otimes}$ tels que $(v_1, A, v_2) \in \mathcal{C}$ avant de retourner une équation de conjugaison (u_1, A, u_2), donc la première condition est respectée.

Le mot $\mathcal{D}^*(\alpha_1)$ ne contient un symbole A que si $\mathcal{D}(A) = A$. De plus, la procédure vérifie que $A \neq B$ avant de retourner une équation de conjugaison (A, α), donc $\alpha = u_2 B \gamma \neq A$, donc $\alpha = u_2 B \gamma \neq A$, ce qui démontre la deuxième condition. □

5.7 Algorithme d'équivalence des grammaires de fonction simple

Nous présentons maintenant notre algorithme d'équivalence des grammaires de fonction simple, appelé ÉQUIVALENCEGFS. Cet algorithme construit une relation *self-proving* à partir d'une relation de conjugaison $\mathcal{C}$ et d'une fonction de décomposition $\mathcal{D}$, et utilise la procédure `extraire_equation` pour extraire les équations permettant de construire $\mathcal{C}$ et $\mathcal{D}$ et pour démontrer l'équivalence des non-terminaux reçus en entrée.

5.7.1 Relation de type *self-proving*

Nous définissons tout d'abord la notion de relation *self-proving* qui est utilisée par notre algorithme.

DÉFINITION 5.4. *Soient une grammaire de fonction simple $G = (\Sigma, \Omega, N, P)$, une relation de conjugaison $\mathcal{C} \subset \Omega^{\otimes} \times N \times \Omega^{\otimes}$ et une fonction de décomposition $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$.*

Nous disons que $E(\mathcal{C}, \mathcal{D})$ est self-proving *si pour tout $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$:*

- *Si $\alpha_1 \xrightarrow{a} \beta_1$, alors il existe β_2 tel que $\alpha_2 \xrightarrow{a} \beta_2$ et `extraire_equation`$(\beta_1, \beta_2, \mathcal{C}, \mathcal{D}) =$ NIL,*
- *Si $\alpha_2 \xrightarrow{a} \beta_2$, alors il existe β_1 tel que $\alpha_1 \xrightarrow{a} \beta_1$ et `extraire_equation`$(\beta_1, \beta_2, \mathcal{C}, \mathcal{D}) =$ NIL,*

où $a \in \Sigma$, $\beta_1, \beta_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$.

Nous démontrons un résultat concernant les relations de type *self-proving* qui est analogue à celui présenté au sujet des fonctions de décomposition de type *self-proving* pour les grammaires simples (lemme 4.3).

LEMME 5.8. *Soient une grammaire de fonction simple $G = (\Sigma, \Omega, N, P)$, une relation de conjugaison $\mathcal{C} \subset \Omega^{\otimes} \times N \times \Omega^{\otimes}$ et une fonction de décomposition $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$ tels que $E(\mathcal{C}, \mathcal{D})$ est* self-proving*. Alors $F_G(\alpha_1) = F_G(\alpha_2)$ pour tout $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$.*

Démonstration. Nous démontrons, par induction sur n, que si $E(\mathcal{C}, \mathcal{D})$ est *self-proving*, alors pour tout $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$ et $n \geq 0$, $F_n(\alpha_1) = F_n(\alpha_2)$.

Si $n = 0$: il y a 3 possibilités selon le format de α_1 et α_2 :

1. Si $\alpha_1, \alpha_2 \in (\Omega \cup \overline{\Omega})^* N (\Omega \cup \overline{\Omega} \cup N)^*$, alors $F_0(\alpha_1) = F_0(\alpha_2) = \emptyset$.
2. Si α_1 ou α_2 est un élément de $\Omega^{\otimes}$, mais pas les deux, alors $(\alpha_1, \alpha_2) \notin E(\mathcal{C}, \mathcal{D})$.
3. Si α_1, α_2 sont tous les deux éléments de $\Omega^{\otimes}$, alors $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$ ssi $\alpha_1 = \alpha_2$, donc $F_0(\alpha_1) = F_0(\alpha_2)$.

Ainsi, $F_0(\alpha_1) = F_0(\alpha_2)$ pour tout $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$.

Supposons que pour tout $k \in [0, n-1]$, $F_k(\alpha_1) = F_k(\alpha_2)$ pour tout $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$. Nous allons démontrer, par contradiction, que cette égalité tient également pour $k = n$.

Supposons qu'il existe $(\alpha_1, \alpha_2) \in E(\mathcal{C}, \mathcal{D})$ tel que $F_n(\alpha_1) \neq F_n(\alpha_2)$. Alors il existe $(w, u) \in \Sigma^* \times \Omega^*$ tel que $(w, u) \in F_n(\alpha_1)$ et $(w, u) \notin F_n(\alpha_2)$, ou $(w, u) \notin F_n(\alpha_1)$ et $(w, u) \in F_n(\alpha_2)$. Supposons, sans perte de généralité, que nous sommes en présence du premier cas, c.-à-d. $(w, u) \in F_n(\alpha_1)$ et $(w, u) \notin F_n(\alpha_2)$.

Nous avons $|w| = n > 0$. Soient $a \in \Sigma$ et $w' \in \Sigma^*$ tels que $w = aw'$. Comme $E(\mathcal{C}, \mathcal{D})$ est *self-proving* et que $\alpha_1 \xrightarrow{a} \beta_1$ pour un certain $\beta_1 \in (\Omega \cup \overline{\Omega} \cup N)^*$, alors $\alpha_2 \xrightarrow{a} \beta_2$ pour un certain $\beta_2 \in (\Omega \cup \overline{\Omega} \cup N)^*$ et $\texttt{extraire_equation}(\beta_1, \beta_2, \mathcal{C}, \mathcal{D}) = \textsc{nil}$. De plus, $(w', u) \in F_{n-1}(\beta_1)$ et $(w', u) \notin F_{n-1}(\beta_2)$.

Par hypothèse d'induction, $F_{n-1}(\gamma_1) = F_{n-1}(\gamma_2)$ pour tout $(\gamma_1, \gamma_2) \in E(\mathcal{C}, \mathcal{D})$. Or, comme $\texttt{extraire_equation}(\beta_1, \beta_2, \mathcal{C}, \mathcal{D}) = \textsc{nil}$, nous avons $F_{n-1}(\beta_1) = F_{n-1}(\beta_2)$ d'après le lemme 5.7. Mais $(w', u) \in F_{n-1}(\beta_1)$, donc $(w', u) \in F_{n-1}(\beta_2)$, ce qui est une contradiction. □

5.7.2 Description de l'algorithme ÉquivalenceGFS

Notre algorithme d'équivalence des grammaires de fonction simple ÉQUIVALENCEGFS consiste à construire une relation de conjugaison $\mathcal{C}$ et une fonction de décomposition $\mathcal{D}$, telles que $E(\mathcal{C}, \mathcal{D})$ est *self-proving*, nous permettant de démontrer l'équivalence des non-terminaux $X, Y \in N$ reçus en entrée.

La relation de conjugaison $\mathcal{C} \subset \Omega^{\otimes} \times N \times \Omega^{\otimes}$ est initialisée à $\mathcal{C} = \emptyset$. La fonction de décomposition $\mathcal{D} : (\Omega \cup \overline{\Omega} \cup N)^* \to (\Omega \cup \overline{\Omega} \cup N)^*$ est initialisée à $\mathcal{D}(A) := A$ pour tout $A \in N$. Notez que $E(\mathcal{C}, \mathcal{D})$ est initialement *self-proving*. Un ordre total sur les éléments de N est établi, de sorte que pour tous $A, B \in N$, $A > B$ implique $||A|| \geq ||B||$. Tous les mots manipulés par l'algorithme sont conservés, à tout moment, sous leur forme réduite.

L'algorithme conserve un ensemble $Q \subset (\Omega \cup \overline{\Omega} \cup N)^* \times (\Omega \cup \overline{\Omega} \cup N)^*$ de paires de mots (α_1, α_2) pour lesquels nous devons vérifier si `extraire_equation`$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D}) =$ NIL. L'ensemble Q contient initialement la paire de non-terminaux (X, Y) reçue en entrée de l'algorithme. L'algorithme fonctionne en itérant une boucle `while`. À chaque itération, nous choisissons arbitrairement une paire (α_1, α_2) pour laquelle nous appelons `extraire_equation`$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D})$.

Si `extraire_equation` retourne NIL nous avons démontré l'équivalence de α_1 et α_2 (en supposant $E(\mathcal{C}, \mathcal{D})$ *self-proving*) et l'itération se termine. Si `extraire_equation` retourne ÉCHEC, nous avons trouvé une preuve réfutant l'équivalence de la paire de non-terminaux d'entrée, et l'algorithme retourne FAUX. Si `extraire_equation` retourne une équation de conjugaison (u_1, A, u_2), nous ajoutons cette équation à $\mathcal{C}$, puis nous appliquons pour chaque $a \in \Sigma$ une étape de dérivation en parallèle sur $u_1 A$ et $A u_2$, et ajoutons les paires résultantes à Q, pour que l'équivalence des mots de chacune de ces paires soit vérifiée ultérieurement, tel qu'exigé par la définition d'une relation *self-proving*, et l'itération se termine. Si `extraire_equation` retourne une équation de décomposition (A, α), nous mettons à jour $\mathcal{D}$ avec $\mathcal{D}(A) := \alpha$, puis nous appliquons pour chaque $a \in \Sigma$ une étape de dérivation en parallèle sur A et α, et ajoutons les paires résultantes à Q, et l'itération se termine.

Nous exécutons cette procédure itérativement sur les éléments de Q, jusqu'à ce que Q devienne vide, ce qui démontrera l'équivalence des non-terminaux d'entrée, ou jusqu'à trouver une preuve réfutant l'équivalence des non-terminaux d'entrée.

L'algorithme ÉQUIVALENCEGFS est présenté à la Fig 5.2.

5.7.3 Preuve d'exactitude et complexité

Nous démontrons que l'algorithme ÉQUIVALENCEGFS est correct en démontrant les faits suivants, où $X, Y \in N$ sont les symboles d'entrée de l'algorithme :

1. L'algorithme se termine toujours,
2. Si $F_G(X) = F_G(Y)$, alors l'algorithme retourne VRAI,
3. Si l'algorithme retourne VRAI, alors $F_G(X) = F_G(Y)$.

LEMME 5.9. *L'algorithme* ÉQUIVALENCEGFS *s'exécute en temps polynomial par rapport à n et $||G||$, où $n = |G|$.*

Entrée : Une grammaire de fonction simple $G = (\Sigma, \Omega, N, P)$ et $X, Y \in N$;
Sortie : VRAI si $X \equiv_G Y$, FAUX sinon.

```
Initialisation :
    Q := {(X, Y)} ;
    C := ∅ ;
    for each A ∈ N do D(A) := A ;
while Q n'est pas vide do
    (α1, α2) := un élément choisi dans Q ;
    switch (extraire_equation(α1, α2, C, D)) do
        case NIL : enlever (α1, α2) de l'ensemble Q ;
        case ÉCHEC : return FAUX ;
        case (u1, A, u2) :      — Équation de conjugaison
            C := C ∪ {(u1, A, u2)};
            for each a ∈ Σ do
                β1 := derive(u1A, a), β2 := derive(Au2, a) ;
                Ajouter (β1, β2) à Q ;
        case (A, α) :      — Équation de décomposition
            D(A) := α ;
            for each a ∈ Σ do
                β1 := derive(A, a), β2 := derive(α, a) ;
                Ajouter (β1, β2) à Q ;
return VRAI ;
```

FIG. 5.2 – Algorithme ÉQUIVALENCEGFS.

Démonstration. Soit $k \stackrel{\text{def}}{=} \texttt{max}\{|\alpha| \mid (A, a, \alpha) \in P\}$, c.à-d. la longueur de la plus longue règle de production dans P. Alors, pour tout $A \in N$, $(w, u) \in F_G(A)$ implique $|u| \leq k|w|$. Remarquez que pour tout $(\alpha, \beta) \in Q$, $\texttt{min}(||\alpha||, ||\beta||) \leq k \cdot ||G||$.

La phase préliminaire (calculer `min` $F_G(A)$ pour tout $A \in N$, et calculer $\texttt{SingleOut}(G)$) prends un temps $O(n^3 \cdot ||G||)$ (lemme 5.4). La proposition 5.3 implique que le nombre d'insertions dans $\mathcal{C}$ et $\mathcal{D}$ est $O(|N|)$, l'ensemble Q peut donc contenir au plus $|N| \cdot |\Sigma| + 1 \leq |G| \cdot |\Sigma| + 1$ paires (α, β). Le nombre d'itérations de la boucle `while` est donc $O(|G|)$. Le nombre d'itérations de la procédure `extraire_equation` à chaque appel est $O(|G| \cdot ||G||)$ (lemme 5.5). Toutes les opérations de la procédure prennent un temps qui est proportionnel à la taille des mots manipulés (lemme 5.1). La taille des mots manipulés étant polynomiale par rapport à $|G|$ et $||G||$, nous obtenons que le temps d'exécution de l'algorithme ÉQUIVALENCEGFS est polynomial par rapport à $|G|$ et $||G||$. □

LEMME 5.10. *Soient une grammaire de fonction simple $G = (\Sigma, \Omega, N, P)$ et $X, Y \in N$. Si $F_G(X) = F_G(Y)$, alors* ÉQUIVALENCEGFS(X, Y, G) *retourne* VRAI.

Démonstration. Supposons qu'au début d'une itération de l'algorithme ÉQUIVALENCEGFS :

1. $F_G(\alpha_1) = F_G(\alpha_2)$ pour tout $(\alpha_1, \alpha_2) \in Q$,
2. $F_G(\beta_1) = F_G(\beta_2)$ pour tout $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$.

En traçant une itération de l'algorithme, nous allons démontrer que FAUX ne peut pas être retourné au cours de l'itération, et qu'à la fin de l'itération, ces deux conditions tiennent toujours. Soit (α_1, α_2), l'élément retiré de Q au début de l'itération.

D'après le lemme 5.6, $\texttt{extraire_equation}(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D})$ ne peut pas retourner ÉCHEC. S'il retourne NIL, l'itération se termine et l'invariant est respecté. S'il retourne une équation de conjugaison (u_1, A, u_2), alors $F_G(u_1 A) = F_G(A u_2)$ d'après le lemme 5.6 et l'ajout de (u_1, A, u_2) à $\mathcal{C}$ respecte la condition 2 de l'invariant. S'il retourne une équation de décomposition (A, α), alors $F_G(A) = F_G(\alpha)$ d'après le lemme 5.6 et l'ajout de (A, α) à $\mathcal{D}$ respecte la condition 2 de l'invariant. Dans les deux cas, les paires (β_1, β_2) qui sont ensuite ajoutées à Q respectent la condition 1 de l'invariant et l'itération se termine.

Lorsque la paire d'entrée (X, Y) de l'algorithme ÉQUIVALENCEGFS est telle que $F_G(X) = F_G(Y)$, les deux conditions de l'invariant sont respectées au début de l'exécution de l'algorithme. Les conditions doivent donc tenir tout au long de l'exécution et

la valeur FAUX ne peut jamais être retournée. Comme l'algorithme se termine toujours d'après le lemme 5.9, l'algorithme retourne éventuellement la valeur VRAI. □

LEMME 5.11. *Soient* $G = (\Sigma, \Omega, N, P)$ *une grammaire de fonction simple et* $X, Y \in N$. *Si* ÉQUIVALENCEGFS(X, Y, G) *retourne* VRAI, *alors* $F_G(X) = F_G(Y)$.

Démonstration. Prenons $\mathcal{C}$ et $\mathcal{D}$ à la fin d'une exécution où ÉQUIVALENCEGFS$(X, Y, G) =$ VRAI. Alors `extraire_equation`$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D})$ doit retourner NIL pour tous les éléments (α_1, α_2) qui ont été ajoutés à l'ensemble Q au cours de l'exécution de l'algorithme. Donc $E(\mathcal{C}, \mathcal{D})$ est *self-proving* et, d'après le lemme 5.8, $F_G(\beta_1) = F_G(\beta_2)$ pour tout $(\beta_1, \beta_2) \in E(\mathcal{C}, \mathcal{D})$.

Q contient initialement la paire d'entrée (X, Y), donc `extraire_equation`$(X, Y, \mathcal{C}, \mathcal{D}) =$ NIL. Comme $E(\mathcal{C}, \mathcal{D})$ est *self-proving*, il résulte du lemme 5.7 que $F_G(X) = F_G(Y)$. □

Cette série de lemmes nous donne directement le théorème suivant.

THÉORÈME 5.1. *L'algorithme* ÉQUIVALENCEGFS(A, B, G) *décide de l'équivalence de deux non-terminaux* A *et* B *d'une grammaire de fonction simple* G *en temps polynomial par rapport à* n *et* $||G||$, *où* $n = |G|$.

5.8 Exemple d'exécution de ÉquivalenceGFS

Nous présentons un exemple de l'exécution de l'algorithme ÉQUIVALENCEGFS.

EXEMPLE 5.3. *Soit* $G = (\Sigma, \Omega, N, P) = (\{0, 1\}, \{a, b\}, \{S, T, X, Y, Z\}, \{S \to 1bZbaab,\ T \to 1YbaaY,\ X \to 0abba,\ X \to 1aYba, Y \to 0bbaXab,\ Y \to 1bZbaab,\ Z \to 0baXaY,\ Z \to 1ZbaaY\})$ *une grammaire de fonction simple.*

Nous considérons l'exécution d'un appel de ÉQUIVALENCEGFS(S, T, G).

Nous calculons tout d'abord $\min F_G(A)$, *pour tout* $A \in \{S, T, X, Y, Z\}$ *:*
$(w_S, u_S) = (w_T, u_T) = (10000, bbaabbaabbaabbaabbaab)$,
$(w_X, u_X) = (0, abba)$, $(w_Y, u_Y) = (00, bbaabbaab)$, *et*
$(w_Z, u_Z) = (0000, baabbaabbaabbaab)$.

Nous spécifions $X < Y < Z < S < T$ *car* $||X|| = 1$, $||Y|| = 2$, $||Z|| = 4$, $||S|| = ||T|| = 5$. *Nous calculons* ***`SingleOut`***(G), *qui dans notre cas est vide.*

L'étape d'initialisation assigne $Q = \{(S, T)\}$, $\mathcal{C} = \{\}$, *et* $\mathcal{D}(A) := A$ *pour tout* $A \in N$.

La première itération de la boucle **while** *enlève tout d'abord une paire* $(\alpha_1, \alpha_2) = (S, T)$ *de l'ensemble* Q*, puis un appel de* **extraire_equation**$(\alpha_1, \alpha_2, \mathcal{C}, \mathcal{D})$ *est effectué.*

La paire est décomposée par la procédure au moyen de $\mathcal{D}$*, ce qui ne change pas la paire car* $\mathcal{D}(A) = A$ *pour tout* $A \in N$*. Le plus long préfixe commun de* S *et* T *est ensuite enlevé, ce qui ne change pas les mots car* S *et* T *ne partagent aucun préfixe commun.* S *et* T *sont de la forme :* $u_1.A.\alpha' = \varepsilon.S.\varepsilon$ *et* $u_2.B.\alpha'_2 = \varepsilon.T.\varepsilon$*. Ensuite, comme* $S < T$ *et*

$$(w_S, u_S) = (10000, bbaabbaabbaabbaabbaab),$$

nous calculons

$$\gamma = \textbf{\textit{quot}}(T, S) = (bbaabbaabbaabbaabbaab)^{-1}\textbf{\textit{derive}}(T, 10000) = \varepsilon.$$

#	Q	(α_1, α_2)	simplification de (α_1, α_2)	γ	$\mathcal{C}$	$\mathcal{D}$
1	(S, T)	(S, T)	$(\varepsilon.S.\varepsilon, \varepsilon.T.\varepsilon)$	ε		(T, S)
2	(S, T), $(\bot, \bot)$, $(YbaaY, bZbaab)$	(S, T)	$(\varepsilon, \varepsilon)$			
3	$(\bot, \bot)$, $(YbaaY, bZbaab)$	$(\bot, \bot)$				
4	$(YbaaY, bZbaab)$	$(YbaaY, bZbaab)$	$(\varepsilon.Y.baaY, b.Z.baab)$	$\bar{b}Y$		$(Z, \bar{b}Y\bar{b}Y)$
5	$(YbaaY, bZbaab)$, $(baXaY, baXaY)$, $(ZbaaY, ZbaaY)$	$(YbaaY, bZbaab)$	$(baa.Y.\varepsilon, \bar{b}.Y.baab)$	$baab$	$(bbaa, Y, baab)$	
6	$(YbaaY, bZbaab)$, $(baXaY, baXaY)$, $(ZbaaY, ZbaaY)$, $(bbaabbaXab, bbaXabbaab)$, $(bbaabZbaab, bZbaabbaab)$	$(YbaaY, bZbaab)$	$(baa.Y.\varepsilon, \bar{b}.Y.baab)$	$baab$	$(bbaa, Y, baab)$	
		$(baab, baab)$	$(\varepsilon, \varepsilon)$			
7	$(baXaY, baXaY)$, $(ZbaaY, ZbaaY)$, $(bbaabbaXab, bbaXabbaab)$, $(bbaabZbaab, bZbaabbaab)$	$(baXaY, baXaY)$	$(\varepsilon, \varepsilon)$			
8	$(ZbaaY, ZbaaY)$, $(bbaabbaXab, bbaXabbaab)$, $(bbaabZbaab, bZbaabbaab)$	$(ZbaaY, ZbaaY)$	$(\varepsilon, \varepsilon)$			
9	$(bbaabbaXab, bbaXabbaab)$, $(bbaabZbaab, bZbaabbaab)$	$(bbaabbaXab, bbaXabbaab)$	$(abba.X.ab, \varepsilon.X.abbaab)$	$abba$	$(abba, X, abba)$	
10	$(bbaabbaXab, bbaXabbaab)$, $(bbaabZbaab, bZbaabbaab)$, $(abbaabba, abbaabba)$, $(abbaaYba, aYbaabba)$	$(bbaabbaXab, bbaXabbaab)$	$(abba.X.ab, b.X.abbaab)$	$abba$	$(abba, X, abba)$	
		$(abbaab, abbaab)$	$(\varepsilon, \varepsilon)$			
11	$(bbaabZbaab, bZbaabbaab)$, $(abbaabba, abbaabba)$, $(abbaaYba, aYbaabba)$	$(bbaabZbaab, bZbaabbaab)$	$(bbaa.Y.\bar{b}Ybaab, \varepsilon.Y.\bar{b}Ybaabbaab)$	$baab$	$(bbaa, Y, baab)$	
		$(baab\bar{b}Ybaab, \bar{b}Ybaabbaab)$	$(baa.Y.baab, \bar{b}.Y.baabbaab)$	$baab$	$(bbaa, Y, baab)$	
		$(baabbaab, baabbaab)$	$(\varepsilon, \varepsilon)$			
12	$(abbaabba, abbaabba)$, $(abbaaYba, aYbaabba)$	$(abbaabba, abbaabba)$	$(\varepsilon, \varepsilon)$			
13	$(abbaaYba, aYbaabba)$	$(abbaaYba, aYbaabba)$	$(bbaa.Y.ba, \varepsilon.Y.baabba)$	$baab$	$(bbaa, Y, baab)$	
		$(baabba, baabba)$	$(\varepsilon, \varepsilon)$			
14	*vide*	Retourne NIL				

FIG. 5.3 – Exemple d'exécution de ÉQUIVALENCEGFS démontrant $F_G(S) = F_G(T)$.

Comme le premier non-terminal de S et T est différent (c.à-d. $S < T$), la procédure **`extraire_equation`** *retourne* (T, S). *Nous mettons* $\mathcal{D}$ *à jour avec* $\mathcal{D}(T) := S$. *Nous calculons également* **`derive`**$(T, 0) = \bot$, **`derive`**$(S, 0) = \bot$, **`derive`**$(T, 1) = YbaaY$ *et* **`derive`**$(S, 1) = bZbaab$ *et ajoutons les paires* $(\bot, \bot)$ *et* $(YbaaY, bZbaab)$ *à l'ensemble* Q, *ce qui complète l'itération.*

La trace complète de l'exécution de l'algorithme est présentée à la Fig 5.3. La colonne # numérote les itérations de la boucle ***while*** *de l'algorithme* ÉquivalenceGFS. *La colonne* Q *contient l'état de* Q *au début de chaque itération de la boucle* ***while*** *de* ÉquivalenceGFS. *La colonne* (α_1, α_2) *contient les valeurs associées à* α_1 *et* α_2 *au début de chaque itération de la boucle* ***while*** *de la procédure* **`extraire_equation`**. *La colonne* "simplification de (α_1, α_2)" *correspond au résultat de la décomposition de* α_1 *et* α_2 *par* $\mathcal{D}$ *suivi de l'élimination du plus long préfixe commun : le résultat est écrit sous la forme* $(u_1.A.\alpha'_1, u_2.B.\alpha'_2)$. *La colonne* $\mathcal{C}$ *contient les équations de conjugaison ajoutées par l'algorithme* ÉquivalenceGFS *(en noir) ou vérifiées par la procédure* **`extraire_equation`** *(en gris) au cours de l'itération. La colonne* $\mathcal{D}$ *contient les mises-à-jour de* $\mathcal{D}$ *effectuées par l'algorithme* ÉquivalenceGFS.

Chapitre 6

Implémentation des algorithmes d'équivalence des grammaires simples

6.1 Description des algorithmes implémentés

Nous présentons dans ce chapitre les résultats de l'exécution d'une implémentation de notre algorithme ÉQUIVALENCEGS, que nous appellerons ALG1. À titre comparatif, nous avons également implémentés les algorithmes d'équivalence des grammaires simples présentés dans [5] et [12], que nous appellerons respectivement ALG2 et ALG3. Ces trois algorithmes ont été exécutés sur le même jeu d'essais, provenant de grammaires simples utilisées en pratique à IDT Canada Inc. dans une technologie de classification de paquets.

Les trois algorithmes que nous avons implémentés se différencient sur deux principaux aspects : la méthode de construction de la fonction de décomposition $\mathcal{D}$ de type *self-proving* et la méthode utilisée pour découvrir la première paire de symboles non-terminaux qui diffère dans les chaînes décomposées par $\mathcal{D}$.

Les algorithmes ALG1 et ALG2 construisent la fonction de décomposition de type *self-proving* directement à partir des non-terminaux reçus en entrée de l'algorithme. Une seule fonction est construite, permettant de démontrer l'équivalence pour cette paire de non-terminaux. L'algorithme ALG3 procède de manière différente. Il construit tout d'abord un ensemble $\mathcal{S}$ de tous les candidats potentiels de décompositions, c.-à-d. que

pour chaque paire $A, B \in N$ telle que $||A|| \geq ||B||$, il calcule $(A, B \cdot \texttt{quot}(A, B))$. Il tente ensuite de démontrer que cet ensemble est *self-proving* en construisant pour chaque $(A, \alpha) \in \mathcal{S}$ une fonction de décomposition de type *self-proving* $\mathcal{D} \subset S$ telle que $\mathcal{D}^*(\beta_1) \equiv_G \mathcal{D}^*(\beta_2)$ pour chaque $\beta_1, \beta_2 \in N^*$ tels que $A \xrightarrow{a} \beta_1$ et $\alpha \xrightarrow{a} \beta_2$ pour un certain $a \in \Sigma$. S'il est impossible de construire une telle fonction de décomposition pour une paire de $\mathcal{S}$, alors cette paire est éliminée de $\mathcal{S}$ et le processus est repris du début, jusqu'à l'obtention d'un ensemble $\mathcal{S}$ où toutes les paires satisfont ce critère. L'ensemble $\mathcal{S}$ résultant contient toutes les paires $(A, B \cdot \texttt{quot}(A, B))$ telles que $A \equiv_G B \cdot \texttt{quot}(A, B)$. En particulier, il contient toutes les paires de symboles $(A, B) \in N \times N$ telles que $A \equiv_G B$. Cet algorithme permet donc de trouver toutes les paires de non-terminaux équivalents d'une grammaire simple, plutôt que de tester l'équivalence seulement pour une paire donnée comme le font ALG1 et ALG2.

La seconde différence réside dans l'approche utilisée pour implémenter l'algorithme `First-MP` consistant à trouver la première paire de symboles qui diffère entre les mots $\mathcal{D}^*(\alpha_1)$ et $\mathcal{D}^*(\alpha_2)$, pour des mots $\alpha_1, \alpha_2 \in N^*$ et une fonction de décomposition $\mathcal{D} : N^* \to N^*$ donnés. L'algorithme ALG2 décompose explicitement les mots $\mathcal{D}^*(\alpha_1)$ et $\mathcal{D}^*(\alpha_2)$ et compare deux à deux les symboles des mots décomposés. Comme un mot décomposé par $\mathcal{D}$ peut avoir une longueur qui est exponentielle par rapport à $|N|$, cette approche prend un temps exponentiel en pire cas. Les algorithmes ALG1 et ALG3 utilisent plutôt des algorithmes élaborés permettant de résoudre ce problème en temps polynomial, en utilisant des techniques de programmation dynamique.

6.2 Méthodologie

L'algorithme ALG3 permet de trouver toutes les paires de non-terminaux d'une grammaire simple qui sont équivalents, tandis que les algorithmes ALG1 et ALG2 ne vérifient l'équivalence que pour une paire de non-terminaux donnée. Afin de pouvoir comparer les performances de ALG1 et ALG2 avec celles de ALG3, nous appelons itérativement les deux premiers algorithmes sur l'ensemble des paires de non-terminaux $A, B \in N$, ce qui augmente la complexité des algorithmes ALG1 et ALG2 d'un facteur $O(|N|^2)$. La tâche accomplie par les trois algorithmes est donc de découvrir l'ensemble des paires de non-terminaux d'une grammaire qui sont équivalents.

Nous avons implémenté soigneusement les trois algorithmes en apportant certaines améliorations lorsque possible, en respectant toutefois la nature des algorithmes et sans modifier leur complexité théorique en pire cas. Notamment, certaines améliorations ont été rendues possibles par le fait que nous appelons successivement les algorithmes ALG1 et ALG2 sur des paires de non-terminaux d'une même grammaire. Certaines informations qui sont calculées lors d'un appel de l'algorithme peuvent parfois être réutilisées au cours des appels subséquents, en particulier l'information calculée par l'algorithme implémentant la fonction `First-MP` qui est utilisé par ALG1. Cet algorithme, qui est décrit dans [19], construit un tableau de programmation dynamique en cours d'exécution. Or, nous savons que certaines valeurs de ce tableau peuvent être encore valides lors d'un appel subséquent de ALG1 sur différents non-terminaux d'une même grammaire simple. Lorsque cela est possible, nous réutilisons ces valeurs sans les calculer de nouveau. De même, lorsque qu'un appel de ALG1 et ALG2 retourne VRAI, la fonction de décomposition $\mathcal{D}$ qui a été bâtie par les algorithmes est *self-proving*, elle peut donc être réutilisée au cours des appels subséquents de ces algorithmes pour une même grammaire simple. Aussi, nous utilisons une approche par *lazy evaluation* pour construire le tableau de programmation dynamique de l'algorithme `First-MP` utilisé dans ALG1, afin d'éviter de calculer inutilement les valeurs qui ne sont pas requises.

Une autre amélioration apportée aux algorithmes ALG1 et ALG2 est que nous ne vérifions l'équivalence de deux symboles non-terminaux A et B seulement si $||A|| = ||B||$. Ceci permet d'éviter d'appeler les algorithmes d'équivalence sur des paires de symboles pour lesquels la non-équivalence peut être découverte de manière triviale.

Le jeu d'essais sur lequel les trois algorithmes ont été testés est un échantillon des grammaires simples utilisées à IDT Canada Inc. dans une technologie de classification de paquets. Les grammaires simples qui ont été choisies proviennent des trois catégories suivantes :

A. Chaque grammaire de cette catégorie définie un paquet HTTP utilisant le protocole TCP, décrivant des contraintes associant :
   - des adresses IP de source et de destination
   - des en-têtes HTTP définies par le biais d'expressions régulières similaires à :
     ```
     [^\r\n]*/index\.html HTTP/[0-9]+\.[0-9]+\r\n.*[H|h]ost:idt\.com\r\n.*...
     ```

B. Les grammaires de cette catégorie décrivent des règles pour une technologie de Sun permettant l'équilibre de charge (*load balancing*) pour les couches 4 à 7. L'analyse

	A			B			C		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Nombre de non-terminaux	2878	11784	29520	1852	2568	3478	1122	4555	5765
Nombre de règles de production	4972	19025	49735	2418	3415	4944	1707	5254	6789
Longueur moyenne d'un mot le plus court	180	216	317	62	218	280	201	281	307
Longueur maximale d'un mot le plus court	374	622	1064	624	773	824	680	680	680

TAB. 6.1 – Caractéristiques des trois catégories de grammaires simples du jeu d'essais.

est basée sur les protocoles IP et les ports TCP/UDP (couche 4) ou sur les URLs, les *cookies* et les scripts CGI (couche 7).

C. Les grammaires de cette catégorie décrivent des règles démontrant les capacités de PAX.port (un co-processeur programmable effectuant la classification de paquets à la vitesse de transmission des données) utilisé comme coupe-feu (couche 3).

Toutes les grammaires du jeu d'essais possèdent un alphabet terminal binaire. D'autres caractéristiques de ces grammaires sont décrites dans le tableau 6.1.

6.3 Résultats

Nous comparons les performances des trois algorithmes en les exécutant sur divers exemples issus des trois catégories de grammaires simples. Nous mesurons pour chacun le temps pris pour construire une grammaire simple réduite, c.-à-d. une grammaire simple équivalente à la grammaire d'entrée mais qui ne contient aucune paire de non-terminaux équivalents. Les résultats, présentés à la figure 6.1, sont regroupés par catégorie de grammaires et ordonnés selon le nombre de non-terminaux dans chaque grammaire, chaque colonne représentant une exécution particulière.

L'algorithme ALG2 offre de bonnes performances pour tous les tests effectués. Le temps nécessaire pour exécuter cet algorithme sur différentes grammaires de tailles similaires varie très peu, ce qui fait de cet algorithme une bonne solution pratique au problème de l'équivalence des grammaires simples.

L'algorithme ALG1 offre lui aussi de bonnes performances pour la plupart des grammaires considérées, bien que le temps d'exécution pour certaines grammaires soit nettement plus élevé que le temps requis par ALG2 pour ces mêmes grammaires. Les algorithmes ALG1 et ALG2 ont tous les deux été en mesure de compléter leur exécution, pour toutes les grammaires du jeu d'essais, à l'intérieur du temps alloué.

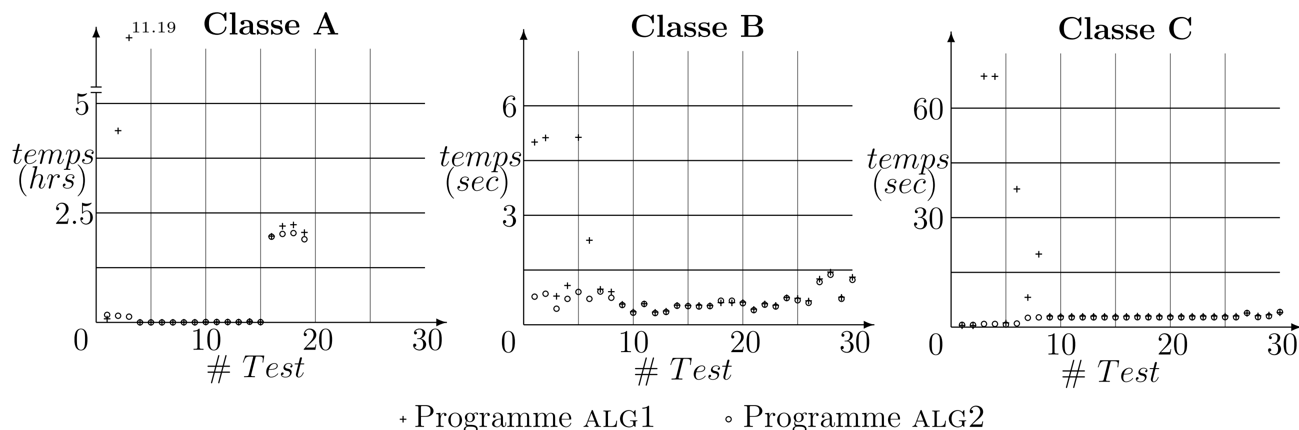

FIG. 6.1 – Résultats expérimentaux de l'exécution des algorithmes d'équivalence des grammaires simples. Chaque colonne représente une grammaire différente, les grammaires étant ordonnées de gauche à droite par rapport au nombre de non-terminaux.

L'algorithme ALG1 nécessite en théorie un espace mémoire de taille $O(n^4)$ car le tableau de programmation dynamique construit par l'algorithme `First-MP` utilisé est de taille $O(n^4)$. Cependant, les améliorations apportées à l'algorithme, notamment l'utilisation de l'approche par *lazy memory allocation*, c.-à-d. que nous calculons uniquement les valeurs nécessaires du tableau de programmation dynamique et n'allouons que l'espace mémoire nécessaire pour conserver ces valeurs, permet de réduire considérablement l'espace mémoire utilisé par cet algorithme pour les grammaires considérées.

L'algorithme ALG3 n'est pas efficace car il n'a pas été en mesure de produire un résultat à l'intérieur du temps alloué, même pour les plus petites grammaires. Notez que cet algorithme construit également un tableau de programmation dynamique, dont la taille est $O(n^6)$, pour résoudre le problème `First-MP`. Cependant, la structure de ce tableau étant plus complexe que celui qui est construit par l'algorithme ALG1, il ne nous a pas été possible d'utiliser une approche par *lazy evaluation* pour cet algorithme. De ce point de vue, nous considérons que l'algorithme ALG3 est relativement plus difficile à implémenter.

Afin d'évaluer tout de même le comportement de ALG3, nous avons généré différentes grammaires simples, faisant varier le nombre de non-terminaux de 10 à 2500. Il est important de noter que ces grammaires ont été générées aléatoirement.

Le tableau de la figure 6.2 présente les propriétés des grammaires qui ont été générées. Chaque ligne décrit un ensemble particulier de grammaires, au moyen d'une progression arithmétique représentant le nombre de non-terminaux dans chaque grammaire de

l'ensemble. La colonne *Répétitions* spécifie le nombre de grammaires ayant été générées pour chaque valeur de la progression arithmétique. La figure 6.2 présente, pour chaque taille de grammaire, la moyenne du temps d'exécution. Ces moyennes sont ordonnées selon le nombre de non-terminaux des grammaires correspondantes.

Min	**Raison**	**Max**	**Répétitions**
10	10	90	10
100	50	950	10
1000	250	2000	3
2500	− − −	2500	1

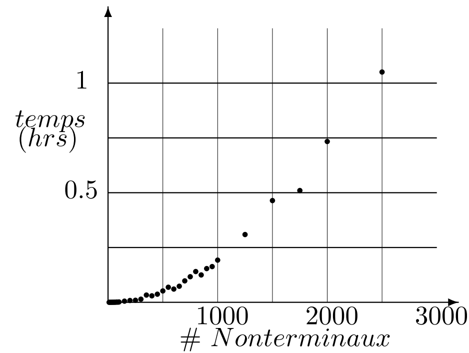

Fig. 6.2 – Temps d'exécution de l'algorithme ALG3 sur des grammaires simples générées aléatoirement.

Il n'était pas possible d'exécuter l'algorithme sur des grammaires possédant plus de 2500 non-terminaux.

Chapitre 7

Conclusion

Nous avons présenté dans ce travail un algorithme d'équivalence des grammaires simples (ALG1) dont la complexité est $O(n^7 \log^2 n)$, améliorant la borne supérieure sur le temps d'exécution des meilleurs algorithmes permettant de résoudre ce problème qui était $O(n^3 \cdot ||G||)$ (ALG2, [5]) et $O(n^{13})$ (ALG3, [12]), où n est la taille de la représentation la grammaire simple et $||G||$ est la longueur maximale d'un mot le plus court pouvant être engendrés par un non-terminal de la grammaire. Comme $||G||$ peut être exponentiel par rapport à n, l'algorithme ALG2 est de complexité exponentielle par rapport à n, tandis que les deux autres algorithmes fonctionnent en temps polynomial.

Nous avons implémenté ces trois algorithmes et les résultats sur le jeu d'essais considéré démontrent que les algorithmes ALG1 et ALG2 peuvent tous les deux être utilisés pour résoudre en pratique le problème d'équivalence des grammaires simples. L'algorithme ALG2 présente un comportement plus stable que celui de l'algorithme ALG1 que nous avons développé, bien que les performances des deux algorithmes soient comparables dans la plupart des cas. Dans le contexte considéré, le comportement de l'algorithme ALG2 est donc celui d'un algorithme polynomial et non d'un algorithme exponentiel. L'implémentation de l'algorithme ALG2 est également beaucoup plus simple que celle de ALG1 car l'approche par programmation dynamique utilisée par ALG1 pour résoudre le problème `First-MP` est beaucoup plus complexe que l'approche directe utilisée par ALG2 pour résoudre ce même problème.

Il n'est pas difficile cependant de définir une classe de grammaires simples pour laquelle le temps d'exécution de l'algorithme ALG2 augmente de manière exponentielle

par rapport à la taille des grammaires tandis que l'algorithme ALG1 s'exécute presque instantanément. Une approche intéressante serait donc d'utiliser un algorithme hybride, ayant le comportement de ALG2 lorsque la valeur du paramètre $||G||$ est relativement petite par rapport à n, mais qui s'exécute comme ALG1 dans les cas particuliers où l'algorithme ALG2 n'est pas efficace.

Nos résultats démontrent également que l'algorithme ALG3 n'est pas applicable en pratique, sauf pour des grammaires possédant un très petit nombre de symboles non-terminaux. Il est important de noter cependant que l'algorithme ALG3 permet de résoudre un problème plus général que celui de l'équivalence des grammaires simples. En effet, les grammaires qui sont considérées par cet algorithme correspondent aux grammaires en forme normale de Greibach qui ne sont pas nécessairement déterministes. De plus, cet algorithme a été conçu dans le but de démontrer que certains problèmes théoriques concernant les langages peuvent être résolus en temps polynomial. Les auteurs ne considèrent aucunement l'aspect pratique de leur solution.

Nous avons finalement présenté un algorithme permettant de résoudre le problème d'équivalence des grammaires de fonction simple. C'est le premier algorithme permettant de résoudre ce problème. Notre algorithme est une généralisation de l'algorithme d'équivalence des grammaires simples présenté dans la première partie du travail. Notre approche pour résoudre ce problème plus général consiste à utiliser une propriété des équations de conjugaison sur un groupe libre engendré par les symboles de l'alphabet de sortie de la grammaire ce qui nous permet de comparer également les mots produits en sortie pour chaque mot d'entrée. Notre algorithme d'équivalence des grammaires de fonction simple est de complexité polynomiale par rapport à n et $||G||$ et de complexité exponentielle si l'on ne considère que le paramètre n. Nous croyons que le comportement de notre algorithme d'équivalence des grammaires de fonction simple est similaire à celui de ALG2 pour le problème d'équivalence des grammaire simple, compte tenu des similarités dans la structure de ces deux algorithmes. Un problème qui demeure ouvert est celui de la conception d'un algorithme d'équivalence des grammaires de fonction simple dont la complexité est polynomiale par rapport à n, comme nous l'avons fait dans ce travail pour le problème d'équivalence des grammaires simples.

Bibliographie

[1] Alur, R., and Madhusudan, P. Visibly pushdown languages. In *Proceedings of STOC'04* (2004), pp. 202–211.

[2] Bastien, C., Czyzowicz, J., Fraczak, W., and Rytter, W. Equivalence of functions represented by simple context-free grammars with output. In *Developments in Language Theory* (2006), pp. 71–82.

[3] Bastien, C., Czyzowicz, J., Fraczak, W., and Rytter, W. Prime normal form and equivalence of simple grammars. *Theoretical Computer Science 363*, 2 (Octobre 2006), 124–134.

[4] Bastien, C., Czyzowicz, J., Fraczak, W., and Rytter, W. Reducing simple grammars : Exponential against highly-polynomial time in practice. In *Pre-Proceedings of CIAA Conference* (Taipei, Taiwan, Août 2006).

[5] Caucal, D. A fast algorithm to decide on simple grammars equivalence. In *Optimal Algorithms*, vol. 401 of *LNCS*. Springer, 1989, pp. 66–85.

[6] Courcelle, B. An axiomatic approach to the Korenjak-Hopcroft algorithms. *Mathematical Systems Theory 16*, 3 (1983), 191–231.

[7] Crochemore, M., and Rytter, W. *Text Algorithms*. Oxford University Press, New York, 1994.

[8] Debski, W., and Fraczak, W. Concatenation state machines and simple functions. In *Implementation and Application of Automata, CIAA'04*, vol. 3317 of *LNCS*. Springer, 2005, pp. 113–124.

[9] Demers, A., Keleman, C., and Reusch, B. On some decidable properties of finite-state translations. *Acta Informatica 17* (1982), 349–364.

[10] Friedman, E. P. The inclusion problem for simple languages. *Theoretical Computer Science 1*, 4 (1976), 297–316.

[11] Harrison, M. A. *Introduction to formal language theory.* Addison Wesley, 1978.

[12] Hirshfeld, Y., Jerrum, M., and Moller, F. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science 158*, 1–2 (1996), 143–159.

[13] Hopcroft, J., and Ullman, J. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] Ibarra, O. H. The unsolvability of the equivalence problem for ε-free NGSM's with unary input (output) alphabet and applications. *SIAM Journal of Computing 7*, 4 (1978), 524–532.

[15] Ibarra, O. H., and Rosier, L. E. On the decidability of equivalence for deterministic pushdown transducers. *Information Processing Letters 13*, 3 (1981), 89–93.

[16] Johnson, D. L. *Presentations of groups.* Cambridge University Press, 1990.

[17] Korenjak, A. J., and Hopcroft, J. E. Simple deterministic languages. In *Proc. IEEE 7th Annual Symposium on Switching and Automata Theory* (1966), IEEE Symposium on Foundations of Computer Science, pp. 36–46.

[18] Lasota, S., and Rytter, W. Faster algorithm for bisimulation equivalence of normed context-free processes. In *Proc. MFCS'06* (2006), vol. 4162 of *LNCS*, Springer-Verlag, pp. 646–657.

[19] Miyazaki, M., Shinohara, A., and Takeda, M. An improved pattern matching for strings in terms of straight-line programs. *Journal of Discrete Algorithms 1*, 1 (2000), 187–204.

[20] Mohri, M. Weighted finite-state transducer algorithms : An overview. In *Formal Languages and Applications*, vol. 148 of *Studies in Fuzziness and Soft Computing.* Springer, Berlin, 2004, pp. 551–564.

[21] Oyamaguchi, M. The decidability of equivalence for real-time DPDAs. *Journal of the ACM 34*, 3 (1987), 731–760.

[22] Pereira, F., and Riley, M. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing.* MIT Press, 1997, pp. 431–453.

[23] Pitcher, C. Visibly pushdown expression effects for xml stream processing. In *Proc. Programming Language Technologies for XML (PLAN-X)* (Janvier 2005).

[24] ROMANOVSKII, V. The equivalence problem for real-time deterministic pushdown automata. *Kibernetika 2* (1986), 13–23.

[25] SÉNIZERGUES, G. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP'97*, vol. 1256 of *LNCS*. Springer, 1997.

[26] SÉNIZERGUES, G. $T(A) = T(B)$? Tech. Rep. 1209-99, LaBRI, France, 1999.

[27] STIRLING, C. Decidability of DPDA equivalence. *Theoretical Computer Science 255*, 1-2 (2001), 1–31.

[28] STIRLING, C. Deciding DPDA equivalence is primitive recursive. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)* (2002), vol. 2380 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 821–832.

[29] TOMITA, E. A direct branching algorithm for checking the equivalence of some classes of deterministic pushdown automata. *Information and Control 52* (1982), 187–238.

[30] TOMITA, E., AND SEINO, K. A direct branching algorithm for checking the equivalence of two deterministic pushdown transducers, one of which is real-time strict. *Theoretical Computer Science 64*, 1 (Avril 1989), 39–53.

[31] TURAKAINEN, P. The undecidability of some equivalence problems concerning ngsm's and finite substitutions. *Theoretical Computer Science 174* (1997), 269–274.

[32] WOOD, D. Some remarks on the KH algorithm for s-grammars. *BIT Numerical Mathmatics 13*, 4 (1973), 476–489.