Machines d'Eilenberg Effectives

Benoit Razet

To cite this version:

Benoit Razet. Machines d'Eilenberg Effectives. Informatique [cs]. Université Paris-Diderot - Paris VII, 2009. Français. NNT : . tel-00463049

HAL Id: tel-00463049

https://theses.hal.science/tel-00463049

Submitted on 11 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ PARIS. DIDEROT (Paris 7)

ÉCOLE DOCTORALE : Sciences mathématiques de Paris Centre

DOCTORAT

Informatique

BENOÎT RAZET

MACHINES D'EILENBERG EFFECTIVES

Thèse dirigée par Gérard HUET

Soutenue le 26 novembre 2009

JURY

MM.	**Jean-Marc Champarnaud**	Rapporteur
	Jean-Christophe Filliâtre	Rapporteur
	Christine Paulin-Mohring	Rapporteur
	Gérard Berry	Examinateur
	Roberto Di Cosmo	Examinateur
	Aarne Ranta	Examinateur
	Gérard Huet	Directeur de thèse

Table des matières

Introduction **9**

I Langages Réguliers, Automates Finis et Axiomatisations **15**

1 Théories des langages réguliers et des automates finis 15
1.1 Semigroupe, monoïde libre, mot, langage 15
1.2 Expressions régulières et langages réguliers 16
1.3 Automates finis et langages reconnaissables 17
1.4 Théorème de Kleene 18
2 Quelques axiomatisations des algèbres de Kleene 19
2.1 Axiomatisation des algèbres de Kleene par Kozen 20
2.2 Axiomatisation des algèbres d'actions de Pratt 24
2.3 Exemples 27
2.4 Compléments 30
3 Conclusion 32

II Simulation des Machines d'Eilenberg **35**

1 Machines relationnelles 35
2 Programmation fonctionnelle en ML 37
3 Relations calculables et flux 39
4 Machines d'Eilenberg 44
4.1 Définition du noyau 44
4.2 Modularité 45
4.3 Interfaces 45
4.4 Généralité du modèle 45
5 Machines d'Eilenberg finies 48
5.1 Définitions 48
5.2 Automates finis et transducteurs 49
5.3 Un moteur réactif de simulation 50
5.4 Formalisation 52
5.5 Formalisation du moteur réactif en Coq 56
5.6 Moteur réactif optimisé 58
5.7 Exemple : un automate à pile pour la grammaire du λ-calcul 59
5.8 Applications 61
6 Machines d'Eilenberg effectives dans un cadre général de calculabilité 62
6.1 Moteur réactif avec stratégies 62
6.2 Moteur semi-réactif avec stratégies 68
7 Conclusion 71

III Algorithmes Fonctionnels de Synthèse d'Automates **73**
1 Structures primitives et algorithme de Thompson 73
1.1 Expressions régulières 73
1.2 Automates finis non-déterministes 74
1.3 Algorithme de Thompson 75
2 Réduire les expressions 76
2.1 Expressions régulières décorées 76
2.2 Associativité de la concaténation 78
2.3 ϵ-réduction 79
2.4 Forme normale d'étoile 80
2.5 Expression régulière normalisée 82
3 Dériver les expressions régulières 82
3.1 Dérivées de Brzozowski et méthode générale 82
3.2 Dérivées d'expressions régulières linéaires 84
3.3 Dérivées partielles 86
3.4 Dérivées canoniques 87
4 Synthèse de l'automate de positions 91
5 Synthèse de l'automate de continuations 93
6 Synthèse de l'automate d'équations 97
7 Performances des algorithmes 101
7.1 Temps de calcul des algorithmes 101
7.2 Nombre de transitions-états de l'automate 101
7.3 Nombre d'états de l'automate 102
7.4 Valider l'implémentation 104
8 Remarques 104
9 Conclusion 105

Conclusion **107**

Résumé

La théorie des automates est apparue pour résoudre des problèmes aussi bien pratiques que théoriques, et ceci dès le début de l'informatique. Désormais, les automates font partie des notions fondamentales de l'informatique, et se retrouvent dans la plupart des logiciels. En 1974, Samuel Eilenberg proposa un modèle de calcul qui unifie la plupart des automates (transducteurs, automates à pile et machines de Turing) et qui a une propriété de modularité intéressante au vu d'applications reposant sur différentes couches d'automates ; comme cela peut être le cas en linguistique computationnelle. Nous proposons d'étudier les techniques permettant d'avoir des machines d'Eilenberg effectives. Cette étude commence par la modélisation de relations calculables à base de flux, puis continue avec l'étude de la simulation des machines d'Eilenberg définies avec ces relations. Le simulateur est un programme fonctionnel énumérant progressivement les solutions, en explorant un espace de recherche selon différentes stratégies. Nous introduisons, en particulier, la notion de machine d'Eilenberg finie pour laquelle nous fournissons une preuve formelle de correction de la simulation. Les relations sont une première composante des machines d'Eilenberg, la deuxième composante étant son contrôle, qui est défini par un automate fini. Dans ce contexte, on peut utiliser une expression régulière comme syntaxe pour décrire la composante de contrôle d'une machine d'Eilenberg. Récemment, un ensemble de travaux exploitant la notion de dérivées de Brzozowski, a été la source d'algorithmes efficaces de synthèse d'automates non-déterministes à partir d'expressions régulières. Nous faisons l'état de l'art de ces algorithmes, tout en donnant une implémentation efficace en OCaml permettant de les comparer les uns aux autres.

Remerciements

Gérard Huet est certainement la personne à qui je dois le plus dans cette histoire. J'ai senti des encouragements de sa part et cela dès mon stage de Master. Je lui suis reconnaissant de m'avoir fait prendre du retard dans la préparation de milliers de pains et de m'avoir permis de faire une thèse sous sa direction. Sa disponibilité, ses conseils, son encadrement, les innombrables relectures qu'il a pu faire de tout ce que j'ai écrit, ce qu'il m'a fait découvrir en sciences et ailleurs, tout cela a fait que cette thèse aura été une aventure passionnante. Je lui suis reconnaissant pour sa bienveillance et pour son soutien permanent, cela a été décisif pour que je mène à bien ce travail.

Je remercie Jean-Marc Champarnaud et Jean-Christophe Filliâtre de l'attention toute particulière avec laquelle ils ont relu le manuscrit. Leurs commentaires et corrections m'ont été très précieux et m'ont permis d'améliorer grandement la qualité du manuscrit. Je remercie les autres membres du jury, Gérard Berry, Roberto Di Cosmo, Christine Paulin-Mohring et Aarne Ranta qui m'ont fait l'honneur de leur participation et qui m'ont apporté des idées nouvelles sur mon travail.

Jean-Baptiste Tristan a joué un rôle important pendant ma thèse, c'est certainement celui qui, en dehors des personnes figurant sur la page principale, connaît le mieux mon sujet. Je le remercie pour son écoute bien sûr mais surtout pour tous les bons moments que nous avons passés ensemble au boulot et en dehors. C'est l'occasion aussi pour moi de remercier sa famille qui m'a très chaleureusement accueillie en vacances avec eux et qui m'a permis de rédiger une partie de ma thèse dans la bonne humeur.

Je remercie Xavier Leroy pour son accueil au sein de l'équipe Gallium à l'INRIA Rocquencourt, je me suis vraiment senti comme un membre de l'équipe avec laquelle j'ai pu partager tous les moments de la vie de labo, en particulier les pauses café si importantes. Je remercie Zaynah Dargaye qui m'a supporté dans le même bureau lorsque je faisais des grimaces suite à la lecture d'articles bien trop compliqués, je la remercie pour sa bonne humeur, ses encouragements et son soutien. Je remercie Nicolas Pouillard pour sa disponibilité et son aide à me sortir de problèmes qui me laissaient pantois, je le remercie aussi pour sa curiosité et son écoute. Je remercie Nelly Maloisel et Stéphanie Aubin pour leur bonne humeur et leur aide lorsqu'il s'agit de batailler avec la paperasse. Je remercie aussi les personnes avec qui j'ai partagé de bons moments à l'INRIA, qu'ils soient du projet Gallium, du projet Moscova, du club de dessin ou encore du club de jonglage : Alain Frisch, Alexandre Pilkiewicz, Arthur Charguéraud, Benoit Montagu, Berke Durak, Boris Yakobowski, Damien Doligez, Denise Maurice, Didier Rémy, Francesco Zappa Nardelli, François Pottier, Isabelle Cabrera, Jade Alglave, Jean-Jacques Lévy, Keiko Nakata, Luc Maranget, Michel Mauny, Paolo Herms, Patrick Meumeu, Sandrine Blazy, Shaoyi Yin, Tahina Ramananandro et Yann Régis-Gianas.

Je remercie Pierre Chabanier, qui m'a taillé sur mesure un programme que j'ai eu le plaisir d'utiliser pour produire le présent manuscrit. Je remercie Matthieu Sozeau pour son aide précieuse en Coq.

Je remercie Yves Bertot ainsi que l'ensemble du projet Marelle pour leur accueil à l'IN-

RIA Sophia-Antipolis, cela a été l'occasion pour moi de travailler sur d'autres thématiques, de découvrir cette belle région du Sud, et de finir ma thèse tout en restant dans le milieu académique idéal.

La répétition pour la soutenance de thèse est une étape importante puisque c'est vraiment la dernière ligne droite, je tiens donc à remercier ceux qui ont assisté à ces multiples répétitions, tout d'abord le projet Marelle à Sophia, puis Florence Plateau et Louis Mandel à Paris, puis Arthur Charguéraud, Tahina Ramananandro et Gérard Huet à Rocquencourt, et pour finir ma maman et Yann Hendel chez moi.

Pour finir, je tiens à remercier très chaleureusement mes amis, mon frère Thomas ainsi que mes parents qui m'ont suivi et encouragé tout au long de cette aventure et devant qui j'ai eu l'honneur et le plaisir de présenter mes travaux lors de la soutenance.

Introduction

Les premiers ordinateurs sont apparus au vingtième siècle. Ce sont des machines d'une grande complexité et pour lesquelles il est utile d'avoir des modèles plus simples et plus intuitifs qui rendent compte de leur capacité. Parmi les pionniers de l'informatique on peut mentionner Alan Turing qui a proposé un modèle mathématique assez simple rendant compte de ces machines complexes, on parle désormais des *machines de Turing* et c'est le modèle de référence. Voyons les différentes notions qui sont mises en jeu dans une machine de Turing. Une machine de Turing est une machine à calculer, elle prend donc une donnée en entrée et produit un résultat de sortie. Elle se compose d'un programme et d'une mémoire. Le programme sert à décrire les séquences d'actions qui peuvent opérer sur la mémoire. Le protocole pour exécuter un calcul est le suivant : il faut préparer la mémoire de manière à ce qu'elle contienne la donnée d'entrée, puis on lance l'exécution de la machine qui modifie la mémoire, et enfin si la machine s'arrête alors on récupère en mémoire le résultat de sortie. La mémoire est modélisée par un ruban (de longueur non bornée) sur lequel est écrit une suite de valeurs binaires (0 ou 1) qu'on appelle des bits. Toute information (par exemple, la donnée d'entrée) doit donc être encodée comme une suite de bits sur ce ruban. Durant son exécution, la machine connaît à chaque instant sa position dans le programme (pour connaître la prochaine action potentielle) et sa position sur le ruban (pour connaître le prochain bit sur le ruban). Les actions qui peuvent avoir lieu sur le ruban sont des opérations de lecture, d'écriture et de déplacement. Tout calcul mécanique peut être effectué par une machine de Turing[1]. Nous avons vu que la structure de la mémoire, c'est-à-dire un ruban, est assez simple. La structure du programme est quant à elle plus complexe, elle est représentée par un graphe fini étiqueté par des actions. Lorsque les actions sont libres de toute interprétation sur la bande, le programme est décrit par un *automate fini*, structure fondamentale et incontournable de l'informatique.

Programmation. En théorie, n'importe quel calcul effectif peut être le résultat d'une machine de Turing, cependant le modèle de calcul correspondant reste très idéalisé et éloigné par rapport à la pratique algorithmique réelle des programmeurs. Par exemple, c'est un très bon exercice d'exprimer un algorithme d'addition d'entiers comme une machine de Turing mais il est impensable d'écrire des algorithmes à peine plus compliqués. Pour exprimer des algorithmes on utilise plutôt des langages de programmation, dont certains sont généralistes et d'autres plus spécialisés pour certaines tâches. Dans le langage de programmation généraliste OCaml [LDGV09], le programmeur écrit ses algorithmes de manière structurée en utilisant des constructions de haut niveau. Les algorithmes sont définis de manière récursive sur des structures de données algébriques. Les structures de données peuvent être abstraites à l'aide de paramètres de type et dans ce cas on parle de polymorphisme. On dit qu'OCaml est un langage de *programmation fonctionnelle* parce que les fonctions ont le même statut que des valeurs plus rudimentaires telles que les entiers. On dit que les fonctions sont d'ordre supérieur parce qu'elles peuvent prendre en paramètre d'autres fonctions. Les structures de données algébriques,

[1] Il s'agit de la thèse de Church.

la récursion, le polymorphisme et l'ordre supérieur sont des atouts indéniables du langage de programmation OCaml qui permettent d'exprimer des algorithmes concis et proches des définitions mathématiques. OCaml est muni d'un système de typage des données qui permet de rejeter statiquement (à la phase de compilation) des programmes dont l'exécution pourrait être erronée, et pour les programmes valides on obtient en contrepartie des garanties de sûreté sur leur exécution. Les types des programmes sont déterminés automatiquement. Cette approche de typage peut être poussée bien plus loin avec des systèmes de types permettant d'exprimer de réelles propriétés mathématiques sur les algorithmes. L'assistant de preuves formelles Coq [Tea09] permet de faire des spécifications mathématiques formelles en se basant sur la théorie des types. En Coq, on peut aussi exprimer des algorithmes, spécifier des propriétés et vérifier que les algorithmes satisfont ces spécifications. Le système Coq est muni d'une technique d'*extraction de programme* vers OCaml, cela rend l'approche de développements formels d'algorithmes très effective. Dans cette thèse nous exprimerons nos algorithmes en OCaml, dont certains ont été formellement vérifiés en Coq.

Automates finis, expressions régulières et langages réguliers. Étant donné un ensemble fini de symboles qu'on appelle *alphabet*, un automate fini est un graphe fini orienté et étiqueté par des symboles. La structure de graphe permet de décrire des chemins auxquels on associe des séquences de symboles. Si on définit un *mot* comme une séquence de symboles et un *langage* comme un ensemble de mots alors à tout automate fini on associe un langage. Les automates finis permettent donc de définir des langages de manière formelle. Plus généralement, l'étude des langages sous cet aspect très formel d'ensembles de mots est appelée la *théorie des langages formels*.

L'étude de la classe des langages qui sont reconnaissables par des automates finis a débuté en 1956 par un article fondateur de Stephen C. Kleene [Kle56]. Le théorème fondamental de Kleene énonce l'équivalence de la classe des langages *reconnaissables* (qu'on peut définir par automate fini) et ceux qu'on peut définir par une *expression régulière*. Ce théorème permet de bien identifier cette classe de langages qu'on appelle les *langages réguliers*. Les expressions régulières sont des expressions algébriques définies récursivement à partir de symboles et des trois opérations suivantes : union, concaténation et étoile de Kleene. Ces opérations sont suffisantes pour définir tous les langages qu'on peut reconnaître par un automate fini et l'avantage des expressions régulières sur les automates est qu'elles sont plus pratiques à manipuler parce que plus algébriques. On peut aussi ajouter d'autres opérateurs aux expressions régulières, en particulier ceux qui correspondent à des opérations de fermeture sur des langages réguliers tels que les opérateurs booléens d'intersection et de complément.

Le théorème de Kleene permet d'étudier les langages réguliers soit sous un aspect calculatoire avec les automates, soit sous un aspect algébrique avec les expressions régulières. De nombreuses questions fondamentales se sont posées sur ces objets dont des questions de décidabilité. En particulier la décidabilité de l'égalité a été résolue de manière effective par Michael Rabin et Dana Scott en 1959 dans leur célèbre[2] article [RS59]. Un autre résultat important du domaine est la preuve d'existence et d'unicité d'un automate minimal pour tout automate fini. Cela permet aussi de décider l'égalité de deux langages en calculant les automates minimaux associés et en les comparant, mais cela permet surtout de donner une forme canonique pour un langage régulier en terme d'automates.

On peut se poser une question similaire du côté algébrique : existe-t-il un système d'équations entre expressions régulières qui permette de rendre compte de leur égalité (en termes des langages qu'elles dénotent) ? Bien que certaines égalités soient triviales (commutativité de l'union, associativité de la concaténation), la question de trouver une telle axiomatisation complète a été posée comme un problème ouvert par Kleene en 1956. Le premier système complet a été donné

[2]Le prix Turing a été attribué à Rabin et Scott en 1976 pour cet article.

par Arto Salomaa en 1966 [Sal66]. Le mathématicien John Horton Conway s'intéressa au sujet dans un livre [Con71] où ces questions sont discutées abondamment. Le système d'équations le plus communément admis est celui proposé par Dexter Kozen [Koz94]. Ces travaux sont restés longtemps cantonnés à un milieu d'algébristes alors que leur portée devrait toucher de nombreux domaines de l'informatique. Nous présenterons donc un état de l'art sur ce sujet.

Les automates sont vraiment un outil fondamental en informatique tant sur le plan théorique que sur le plan pratique, on peut dire qu'ils sont omniprésents ; on les retrouve notamment dans les domaines suivants : description de circuits, langage de programmation (compilation), vérification (*model checking*), logique (avec la logique monadique de second ordre), algèbre (algèbres de Kleene), recherche de motif dans des textes et même dans les systèmes d'exploitation. En effet, le système d'exploitation Unix a été développé par des spécialistes de la théorie des automates tel que Ken Thompson. Au cœur d'Unix on trouve une notion de flux de caractères qui permet de combiner des commandes successivement à l'aide de la commande pipe `|`. Un autre exemple important d'utilisation des automates finis est illustré par la commande `grep` qui effectue la recherche de motif dans un texte. Cette fonctionnalité se retrouve implémentée d'une manière ou d'une autre dans de nombreux logiciels. Le succès de cette fonctionnalité a fait que les expressions régulières sont devenues un paradigme pour les programmes de manipulation de textes tels que `sed`, mais aussi pour les langages de script tels que Perl, Python et Ruby. Dans ces outils, les automates ne font pas seulement que reconnaître si un mot appartient au langage mais ils traduisent un mot en même temps qu'ils le reconnaisse ; on appelle de tels automates des *transducteurs*.

Cette omniprésence des expressions régulières a fait que les techniques pour les compiler en automates n'ont cessé de progresser. Parmi les articles historiques on peut mentionner ceux de McNaughton et Yamada [MY60] et de Glushkov [Glu61]. Ce sont les premiers articles détaillant des procédures effectives pour synthétiser des automates à partir d'expressions régulières. On peut aussi mentionner l'article de Thompson [Tho68] qui est à l'origine de la commande `grep`. Dans cette thèse nous présenterons des algorithmes issus de la technique des dérivées de Brzozowski [Brz64]. Les algorithmes issus de cette technique exploitent une notion de dérivation sur les expressions régulières. Il en résulte des algorithmes efficaces tant en temps d'exécution qu'en taille de l'automate produit. Nous montrerons que cette famille d'algorithmes, issus de travaux récents, s'implémente dans le langage de programmation OCaml très efficacement.

Linguistique computationnelle. En 1963, le linguiste Noam Chomsky et le mathématicien Marcel-Paul Schützenberger publient un article [CS63] qui établit de nouveaux liens entre la théorie des langages et la théorie des automates en étudiant la classe de langages qui correspondent à la syntaxe des langages de programmation. L'exemple caractéristique de motif appartenant à cette classe de langages est l'emboîtement d'expressions bien parenthésées, par exemple `(()())()`. Ce motif typique est présent dans tous les langages de programmation modernes. Cependant il n'est pas capturable par les expressions régulières et il faut des automates plus puissants que les automates finis pour le reconnaître ; ce sont les automates à pile. On arrive à dégager clairement une classe de langages plus grande que les langages réguliers contenant ce type de motif, on les appelle les *langages hors-contexte*. À partir des travaux de Chomsky et Schützenberger il s'est dégagé une méthode générique adéquate à l'analyse des langages de programmation. En effet, l'algorithme d'analyse se décrit en deux niveaux successifs : le lexique et la syntaxe. Concernant le lexique, défini le plus souvent à l'aide d'expressions régulières, la puissance des automates finis est suffisante. Concernant la syntaxe, on la définit à l'aide d'une grammaire hors-contexte qui est traduite en automate à pile. Ces deux programmes ont besoin d'une interface pour communiquer l'un avec l'autre à la manière d'un client-serveur. Cette construction de l'algorithme d'analyse en deux couches est parfaitement adaptée aux langages de

programmation cependant elle échoue pour l'analyse des langues naturelles. Dans le cas des langages de programmation, les automates servant à l'analyse doivent être déterministes. On entend par déterminisme que les calculs sont séquentiels et ne peuvent mener qu'à un seul résultat. Plus généralement on peut décrire des systèmes d'analyse qui sont non-déterministes, c'est-à-dire dont le recherche de plusieurs résultats se fait en explorant un espace de recherche multi-directionnel. Nous proposons dans cette thèse de simuler un modèle de calcul non-déterministe dont la modularité intrinsèque permet de définir des algorithmes d'analyse en couches à la manière du lexique/syntaxe.

Concernant la modélisation des langues naturelles, c'est-à-dire les langues parlées par l'homme, il se trouve que certains processus humains de construction de phrases sont décrits de manière algorithmique. À l'époque des débuts de l'*intelligence artificielle* ceci a donné l'impulsion à de nombreux projets cherchant à mécaniser les langues naturelles. À la même époque, plusieurs formalismes mathématiques pour modéliser les langues sont apparus ; dans ce domaine Chomsky était un pionnier. Rapidement certains ont vu à travers ces formalismes un moyen de réaliser des outils de *traduction automatique*, c'est-à-dire des logiciels pour traduire automatiquement un texte exprimé dans une langue vers une autre langue, par exemple de l'anglais vers le russe. Quelques décennies plus tard, la traduction automatique a péniblement progressé et le nombre de projets sur cette thématique a chuté vers le milieu des années 70 [Gro72] parce qu'il s'est avéré que le sujet était bien plus difficile que prévu. Quoi qu'il en soit, la traduction automatique a été à l'origine de nombreux travaux se situant à l'interface de la linguistique et de l'informatique, on les regroupe désormais sous le nom de *linguistique computationnelle*. Les enjeux autour de cette discipline sont majeurs, surtout à notre époque où l'on a très facilement accès à une multitude de documents rédigés dans une grande variété de langues.

Il existe de nombreux outils pour la linguistique computationnelle qu'on peut retrouver sous forme de boîte à outils qui peuvent être développées par des entreprises mais aussi dans le milieu académique dans le monde entier. Les automates et les transducteurs ont beaucoup d'applications en linguistique computationnelle, on pourra se référer aux références suivantes [KK94, Kar00, Moh97, RS95, RS97, Spr92]. Notre intérêt s'est concentré sur la boîte à outils *Zen* [Hue05, Hue02] développée par Gérard Huet à l'INRIA Paris-Rocquencourt. Elle contient des structures de données à base d'automates pour créer des lexiques. Les lexiques peuvent être annotés par des informations de flexions et dans ce cas l'automate peut servir de transducteur. À l'aide d'un simulateur, facilement modifiable, appelé *moteur réactif*, les lexiques sont utilisés soit en reconnaissance, soit en analyse ou encore en synthèse. Par exemple, étant donnés une phrase et un lexique, le moteur réactif anime l'automate associé au lexique afin de décomposer la phrase en une séquence de mots appartenant au lexique. Une particularité de la boîte à outils est d'être développée en OCaml de manière applicative, avec un souci de présenter les structures de données et les algorithmes avec clarté et concision selon une méthodologie de programmation vue comme une œuvre littéraire (en anglais *literate programming*)[3].

La boîte à outils Zen est issue d'outils spécifiques pour l'analyse du sanskrit. Le sanskrit est une langue ancienne indienne qui a été complètement formalisée par Panini au 4e siècle av. J.C. de manière rigoureuse en s'appuyant sur la phonologie et en utilisant des règles grammaticales, le tout décrit de manière exhaustive et méticuleuse. Ce travail est très précurseur puisque ce n'est que deux mille ans plus tard que les langues commenceront à être présentées avec une telle précision. Le logiciel correspondant est accessible sous forme d'applications web à l'adresse `sanskrit.inria.fr`. Il comprend un dictionnaire sanskrit-français, mais aussi des outils complètement mécanisés de conjugaison, de déclinaison et d'analyse. En sanskrit, il y a une correspondance exacte entre la phrase écrite et la phrase telle que prononcée oralement ;

[3]Les algorithmes ne doivent pas être considérés comme des *"détails techniques"* qu'il faut nécessairement dissimuler.

l'alphabet est donc étroitement relié à l'ensemble des phonèmes, et dans la forme écrite de la phrase on ne retrouve pas de marqueur de séparation entre les mots (comme l'espace). Une des difficultés est de rendre compte du phénomène de liaison, qui est une transformation se produisant à la frontière commune de deux mots qui se suivent, on appelle ce phénomène de liaison le *sandhi*. Pour faire l'analyse d'une phrase sanskrite il faut d'abord trouver les mots dans la phrase et donc la première étape incontournable est de défaire le sandhi. Pour ce type d'analyse les ambiguïtés sont nombreuses et pour être efficace il faut simultanément segmenter la phrase en mots tout en inversant le sandhi. Il est important de noter que contrairement à l'analyse des langages de programmation qui se fait avec des algorithmes déterministes (de préférence), l'analyse des langues naturelles requiert des analyses non-déterministes à cause de l'ambiguïté inhérente à la langue. L'analyse morphologique du sanskrit peut être décrite par un transducteur modulaire [HR06] qui correspond à deux niveaux d'automates, le premier qui contrôle les séquences d'exploration dans les lexiques et le deuxième qui réagit effectivement sur les lexiques qui sont annotés des transductions de sandhi. Ces deux types de contrôle sont tous deux décrits par des automates finis avec un soin particulier pour la gestion du non-déterminisme. Dans cette thèse nous présentons un modèle de calcul modulaire pour exprimer facilement ce type d'algorithme d'analyse.

Machines d'Eilenberg. Samuel Eilenberg était un mathématicien qui s'intéressa aussi à l'informatique théorique et en particulier à la théorie des automates. Sa contribution majeure à la théorie des automates est le livre *"Automata, Languages and Machines"* [Eil74]. Les notions sont présentées de manière algébrique en insistant sur une présentation constructive des preuves des théorèmes afin d'en déduire des algorithmes. Une des ambitions affichées par Eilenberg dans son livre était de proposer un cadre unifié pour traiter tous les niveaux de la hiérarchie de Chomsky, c'est-à-dire les différents niveaux de calculabilité : régulier, hors-contexte, sensible au contexte et récursivement énumérable. Tous ces niveaux seraient traités dans quatre volumes ; au final seulement deux volumes seront terminés. Cependant dès le premier volume il présenta son modèle de calcul général des X-machines qui lui aurait été utile pour son œuvre complète. Une X-machine est un automate fini étiqueté par des relations binaires sur un ensemble X. L'automate définit les séquences possibles de composition de relations. En faisant l'union des compositions de relations on obtient une relation particulière qu'on appelle la relation caractéristique de la machine. Les actions de l'automate sont des relations binaires sur X. Ce modèle de calcul est non-déterministe pour deux raisons : d'une part avec le contrôle (automate non-déterministe) et d'autre part avec les données (relations). Les données manipulées par ces machines sont abstraites par un ensemble arbitraire X qu'on peut instancier par différentes structures de données selon les besoins : mots, bandes, piles, compteur, *etc...* Grâce à la puissance des relations et à l'abstraction du domaine X, une machine d'Eilenberg peut faire des calculs arbitraires et en particulier modéliser les automates finis, les transducteurs, les automates à pile et même les machines de Turing. Ceci constitue une première motivation pour étudier ce modèle de calcul. La seconde motivation tient au fait que le modèle permet de décrire des algorithmes d'analyse en tant que plusieurs niveaux d'automates à la manière de l'analyse lexique/syntaxe des langages de programmation mais dans un cadre plus général de non-déterminisme. Cependant Eilenberg a exprimé son modèle en utilisant les relations trop générales de la théorie des ensembles, et ces dernières ne conviennent pas aux calculs effectifs. Nous proposons donc d'étudier une version *effective* des machines d'Eilenberg. Nous donnerons des techniques pour simuler les machines d'Eilenberg effectives. Les différentes simulations correspondent à des optimisations dans l'énumération des solutions selon certaines propriétés que peuvent vérifier la machine. Par exemple nous insisterons sur les *machines d'Eilenberg finies* [Raz08a, Raz08b] pour lesquelles une recherche de solution en utilisant une stratégie de parcours en profondeur d'abord est possible. Ce moteur réactif est exprimé par

un algorithme purement applicatif sur lequel on peut effectuer des preuves formelles de correction et complétude de l'énumération des solutions. Ensuite nous proposerons une paramétrisation du moteur réactif avec des stratégies. Une des stratégies définira un simulateur pour des machines d'Eilenberg effectives générales.

Plan.
Le chapitre I présente les notions fondamentales de languages réguliers, d'automates finis et d'expressions régulières. Nous fournirons aussi dans ce chapitre un état de l'art des différentes présentations axiomatiques des algèbres de Kleene.

Le chapitre II est dédié à l'étude des machines d'Eilenberg qui sont d'abord présentées à la manière d'Eilenberg c'est-à-dire en théorie des ensembles. Puis nous préciserons ce modèle dans un cadre plus effectif. Pour cela, nous discuterons des relations calculables, ensuite de simulation des machines d'Eilenberg finies puis de la simulation de machines d'Eilenberg effectives.

Le chapitre III présente les implémentations des algorithmes efficaces de synthèse d'automates à partir d'expressions régulières au vu des travaux récents.

Nous voulons montrer dans cette thèse que bien que le modèle de calcul d'Eilenberg soit fondamental d'un point de vue théorique, il est aussi très effectif en pratique. Il a été introduit à un moment où son implémentation aurait été difficile et nous montrons ici qu'il est désormais possible de le rendre effectif à l'aide d'un langage de programmation généraliste moderne de haut niveau. Les programmes présentés dans cette thèse permettent d'une part d'interpréter les machines d'Eilenberg effectives (chapitre II) et d'autre part d'utiliser les expressions régulières comme syntaxe pour définir de telles machines (chapitre III).

Chapitre I

Langages Réguliers, Automates Finis et Axiomatisations

Ce chapitre contient les définitions et notions usuelles de la théorie des automates qui seront nécessaires pour la suite du document. Dans une première partie nous présenterons les langages réguliers, les expressions régulières, les automates finis et nous finirons en rappelant le théorème de Kleene. Dans une seconde partie d'algèbre, nous présenterons deux axiomatisations de l'égalité pour des structures algébriques relatives aux langages réguliers et expressions régulières.

1 Théories des langages réguliers et des automates finis

1.1 Semigroupe, monoïde libre, mot, langage

Un *semigroupe* $(M, \cdot)$ est un ensemble M muni d'une opération binaire $\cdot : M \times M \to M$ appelée le *produit* et notée $m \cdot n$ pour m et n éléments de M ; on pourra omettre l'opérateur lorsque le contexte le permettra et noter plus simplement mn. Un semigroupe satisfait l'axiome d'associativité du produit :

$$\forall m_1, m_2, m_3 \in M,\ (m_1 m_2) m_3 = m_1 (m_2 m_3), \tag{1}$$

Un *monoïde* $(M, \cdot, 1_M)$ est un semigroupe avec un élément remarquable $1_M \in M$ qui a la propriété d'être un *élément neutre* pour le produit :

$$\forall m \in M,\ 1_M m = m = m 1_M. \tag{2}$$

En utilisant cet axiome (2) on peut prouver l'unicité de 1_M, on parlera donc de *l*'élément neutre du produit de monoïde.

On considère maintenant un monoïde particulier qui rend compte de la structure des mots. Il s'agit du *monoïde libre* sur un ensemble Σ, c'est-à-dire le monoïde *engendré par* Σ, on le note Σ^*. Lorsque Σ est un ensemble fini, on parle d'*alphabet* et ses éléments sont appelés des *symboles* qu'on notera a, b, c, *etc.* On définit un *mot* comme un élément du monoïde libre Σ^* dont le mot vide ϵ (mot de longueur nulle) est l'élément neutre. On utilisera couramment l'isomorphisme entre un mot de longueur n et la fonction f de $[1..n]$ dans Σ telle que $f(i) = a_i$ pout tout i dans $[1..n]$. La propriété fondamentale du monoïde libre est de pouvoir identifier les mots comme séquences de symboles : soient deux mots $a_1 \cdots a_n$ et $b_1 \cdots b_m$, si $a_1 \cdots a_n = b_1 \cdots b_m$ alors $n = m$ et $\forall i \in \mathbb{N}, 0 < i \leq n \Rightarrow a_i = b_i$.

Un *langage* sur l'alphabet Σ est un sous-ensemble de Σ^*. Par exemple, le langage contenant uniquement le mot vide est $\{\epsilon\}$. Le langage $L = \{xy \mid x \in \Sigma, y \in \Sigma\}$ est le langage des mots de deux lettres sur Σ, c'est un langage fini. On note a^n le mot de longueur n fait exclusivement de a. Le langage $\{a^n b \mid n \in \mathbb{N}\}$ est le langage qui contient tous les mots commençant par un nombre arbitraire de a et terminant par b. Le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ est le langage dont les mots sont décomposables en deux sous-mots de même taille construits respectivement avec a et b.

1.2 Expressions régulières et langages réguliers

Les *expressions régulières*[1] sont des expressions de l'algèbre libre sur la signature $(\Sigma, +, \cdot, ^*, 0, 1)$ où Σ est un alphabet, les opérateurs $+$ et $\cdot$ sont des opérateurs binaires, l'opérateur * est unaire et 0 et 1 sont des constantes. Par exemple $a \cdot ((b+c)^* \cdot c)$ est une expression régulière.

On peut donner des interprétations différentes aux valeurs et opérations associées aux expressions régulières. L'interprétation qui nous intéresse ici est celle qui associe à toute expression régulière un langage sur l'alphabet Σ.

Une expression régulière E définit un langage $L(E)$ en interprétant les opérateurs $+, \cdot$ et * des expressions régulières par l'union, la concaténation et l'itération (étoile de Kleene). L'expression 0 représente le langage vide $\emptyset$. L'expression 1 représente le langage $L(1)$ égal à $\{\epsilon\}$. Pour tout symbole a, le langage $L(a)$ est le langage qui contient uniquement le mot fait du seul symbole a, c'est-à-dire $\{a\}$. Si E et F sont deux expressions régulières, l'expression $E+F$ définit le langage $L(E+F)$ comme l'*union* des langages $L(E)$ et $L(F)$, c'est-à-dire $L(E+F) = L(E) \cup L(F)$. L'expression $E \cdot F$ définit le langage $L(E \cdot F)$ comme le produit de *concaténation* des langages $L(E)$ et $L(F)$, c'est-à-dire $L(E \cdot F) = L(E) \cdot L(F) = \{ww' \mid w \in L(E) \wedge w' \in L(F)\}$. L'expression E^* définit le langage $L(E^*)$ comme l'union infinie des itérations de concaténations de $L(E)$, c'est-à-dire $L(E^*) = \bigcup_{n \in \mathbb{N}} L(E)^n$ avec le langage $L(E)^n$ défini par récurrence sur n de la manière suivante : $L(E)^0 = L(1) = \{\epsilon\}$ et pour tout entier i, $L(E)^{i+1} = L(E) \cdot L(E)^i$.

$$L(0) = \emptyset$$

$$L(1) = \{\epsilon\}$$

$$L(E+F) = L(E) \cup L(F)$$

$$L(E \cdot F) = L(E) \cdot L(F) = \{ww' \mid w \in L(E) \wedge w' \in L(F)\}$$

$$L(E^*) = \bigcup_{n \in \mathbb{N}} L(E)^n$$

Soit E une expression régulière, on définit la *taille* de E, notée $|E|$, par le nombre d'opérateurs, de constantes et de symboles qui composent E. On définit la *largeur* de E, notée $||E||$, comme le nombre d'occurrences de symboles apparaissant dans E.

On appelle *langage régulier* sur Σ un langage $\mathcal{L}$ définissable par une expression régulière E $(\exists E, \mathcal{L} = L(E))$. La classe des langages réguliers sur un alphabet Σ sera notée $Reg(\Sigma)$.

Les langages réguliers sont fermés par d'autres opérations. Tout d'abord, les langages réguliers sont fermés par les opérations de l'algèbre booléenne. Le *complément* noté L_1^-, l'*intersection* notée $L_1 \cap L_2$ et le *ou exclusif* noté $L_1 \oplus L_2$ sont définis par :

[1] Dans la tradition algébriste française on parle d'*expressions rationnelles*, nous utiliserons dans cette thèse la terminologie anglo-saxonne originelle.

– $L_1^- = \{w \mid w \notin L_1\}$
– $L_1 \cap L_2 = \{w \mid w \in L_1 \land w \in L_2\}$
– $L_1 \oplus L_2 = \{w \mid (w \in L_1 \land w \notin L_2) \lor (w \notin L_1 \land w \in L_2)\}$

En utilisant l'opération de complément et la constante 0 on peut retrouver l'ensemble de tous les mots $\Sigma^* = L(0^-)$. On considère aussi les opérations de *dérivée à gauche* et de *dérivée à droite*, notées respectivement $L_1 \setminus L_2$ et $L_2 \mathbin{/} L_1$, ainsi que l'opération de *renversement* notée $\widetilde{L_1}$. Ces opérations sont définies de la manière suivante :

– $L_1 \setminus L_2 = \{v \mid \exists u, u \in L_1 \land uv \in L_2\}$
– $L_2 \mathbin{/} L_1 = \{v \mid \exists u, u \in L_1 \land vu \in L_2\}$
– $\widetilde{L_1} = \{\widetilde{w} \mid w \in L_1\}$ avec $\widetilde{w} = a_n \cdots a_1$ si $w = a_1 \cdots a_n$ et $\widetilde{\epsilon} = \epsilon$

Les langages réguliers sont aussi fermés par les opérations de *résiduation à droite* et de *résiduation à gauche*, notée respectivement $L_1 \rightarrow L_2$ et $L_2 \leftarrow L_1$ et définies de la manière suivante :

– $L_1 \rightarrow L_2 = \{v \mid \forall u, u \in L_1 \Rightarrow uv \in L_2\}$
– $L_2 \leftarrow L_1 = \{v \mid \forall u, u \in L_1 \Rightarrow vu \in L_2\}$

On peut définir la résiduation à partir de la dérivée, $L_1 \rightarrow L_2 = (L_1 \setminus L_2^-)^-$ et pour cette raison la résiduation joue un rôle d'opération duale par rapport à la dérivée. Les résiduations sont des opérations moins habituelles mais elles jouent un rôle essentiel pour les questions d'axiomatisation, dont nous parlerons dans la suite.

1.3 Automates finis et langages reconnaissables

Un *automate fini* $\mathcal{A}$ est défini par un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ avec Q un ensemble fini d'états, Σ un alphabet fini, une table de transitions $\delta \subseteq (Q \times \Sigma \times Q)$, un ensemble d'états *initiaux* $I \subseteq Q$ et un ensemble d'états *acceptants* $F \subseteq Q$.

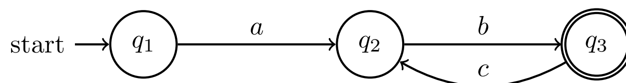
Une *transition* de l'automate est un élément $(q, a, q') \in \delta$, qu'on note aussi $q \xrightarrow{a} q'$, on dit que a est l'*étiquette* de la transition. Un *chemin* dans l'automate $\mathcal{A}$ est une séquence finie de transitions consécutives $(q_1, a_1, q_2)(q_2, a_2, q_3) \cdots (q_n, a_n, q_{n+1})$, noté aussi $q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} q_{n+1}$, avec n la longueur du chemin. On appelle *étiquette* de ce chemin le mot $w = a_1 \cdots a_n$. Un chemin peut être de longueur nulle et son étiquette est alors le mot vide ϵ. Un mot $w = a_1 \cdots a_n$ est dit *reconnu* par l'automate $\mathcal{A}$ s'il existe un chemin étiqueté par w avec $q_1 \in I$ et $q_{n+1} \in F$. L'automate reconnaît ϵ s'il existe un état initial qui est aussi un état acceptant. Le langage reconnu par l'automate $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$, on le note $L(\mathcal{A})$.

L'automate est dit *déterministe* lorsque l'ensemble des états initiaux est un singleton, $|I| = 1$, et lorsque la table de transitions peut être interprétée par une fonction de transitions, $\forall q \in Q, \forall a \in \Sigma, |\{q' \mid \delta(q, a, q')\}| \leq 1$. Lorsque l'automate $\mathcal{A}$ est déterministe on parle de DFA (*deterministic finite automaton*) sinon dans le cas plus général on dit que c'est un NFA (*non-deterministic finite automaton*). Les automates finis ont un graphe de transitions étiquetées par des symboles, cependant on peut assez directement étendre les définitions précédentes pour avoir des automates étiquetés par des mots $(Q, \Sigma^*, \delta, I, F)$. Pour ce type d'automate, un chemin est de la forme $q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_n} q_{n+1}$, et son étiquette est le mot $w_1 \cdots w_n$. Un automate non-déterministe à *transitions spontanées*, noté ϵ-NFA, est un automate dont les transitions sont soit étiquetées par des symboles soit par le mot vide ϵ, $\delta \subseteq (Q \times \{\Sigma \cup \epsilon\} \times Q)$. On parle de transition spontanée (ϵ-move en anglais) lorsque la transition est étiquetée par le mot vide ϵ.

On appelle *langage reconnaissable* sur Σ un langage $\mathcal{L}$ reconnu par un automate fini $\mathcal{A}$ ($\exists \mathcal{A}, \mathcal{L} = L(\mathcal{A})$). La classe des langages reconnaissables sur un alphabet Σ sera notée $Rec(\Sigma)$.

Les automates sont définis de manière formelle avec des constructions ensemblistes, cependant on peut les représenter par leur graphe pour aider l'intuition. Par exemple, l'automate

défini par $(\{1, 2, 3\},\ \{a, b, c\},\ \{(1, a, 2), (2, b, 3), (3, c, 2)\}, \{1\}, \{3\})$ sera représenté de la manière suivante :



Les états sont représentés par des cercles. Une transition $(q, a, q') \in \delta$ est représentée par une flèche ayant pour origine q, pour destination q' et qui est annotée par l'étiquette a. Les états initiaux sont indiqués par une flèche entrante, les états acceptants sont délimités par des cercles en double-trait.

1.4 Théorème de Kleene

Le théorème de Kleene [Kle56] est un théorème fondamental de la théorie des automates. Il établit l'équivalence de la classe des langages réguliers $Reg(\Sigma)$ et des langages reconnaissables $Rec(\Sigma)$. C'est-à-dire qu'un langage reconnu par un automate fini peut aussi être représenté par une expression régulière et *vice versa*.

Théorème I.1 (Kleene [Kle56])**.** Soit $\mathcal{L}$ un langage,

$$\exists E,\ L(E) = \mathcal{L} \quad \Leftrightarrow \quad \exists \mathcal{A},\ L(\mathcal{A}) = \mathcal{L}.$$

Grâce au théorème de Kleene, les propriétés des langages réguliers peuvent être abordées sous deux aspects, soit syntaxiquement avec les expressions régulières, soit de manière calculatoire par les automates. Par exemple, les propriétés de fermeture des langages réguliers par les opérations de l'algèbre booléenne sont facilement prouvables en passant par les automates. Pour cela il faut savoir que tout NFA ou ϵ-NFA peut être traduit en automate fini déterministe (DFA) complet (dont la fonction de transition est totale). Il est facile de construire le complément d'un langage et l'intersection de deux langages à partir d'automates déterministes complets. Le théorème de Kleene fut le premier grand théorème du domaine. Un autre théorème important est celui qui prouve l'existence et l'unicité (à renommage des états près) de l'automate déterministe *minimal* (en le nombre d'états). Cela permet de décider l'équivalence de deux langages réguliers en calculant leur automate minimal et en les comparant. Ce résultat est issu des travaux de Rabin et Scott de 1959 [RS59] qui contient de nombreux autres résultats fondamentaux.

Dans le processus de reconnaissance d'un mot par un automate, décrit jusqu'à présent, il n'est pas question de compter la multiplicité avec laquelle un mot est reconnu. De la même manière, les opérations ensemblistes des langages réguliers ne gardent pas trace des multiplicités qui pourrait être capturées par des opérations définies sur des multi-ensembles. Cette extension aux *multiplicités* pour les automates comme pour les langages réguliers a été un sujet de recherches actif, exploités en parallèle avec la théorie des automates. Ces multiplicités se généralisent encore plus lorsqu'on les considère comme définies sur un semi-anneau abstrait. Cette généralisation a l'avantage de capturer l'extension des automates et des langages aux probabilités. Tout cela a fait l'objet d'études sous l'aspect de mathématique de *séries rationnelles*. Tout d'abord le théorème de Kleene a été étendu pour rendre compte des multiplicités par Schützenberger en 1961 [Sch61], on parle du théorème de Kleene-Schützenberger. Dans son livre sur les automates, Eilenberg [Eil74] fut le premier à inclure de manière quasi systématique les multiplicités au cœur des notions présentées. Sinon, l'ouvrage de référence sur les séries rationnelles est celui de Berstel et Reutenauer [BR88]. Le livre de Sakarovitch paru en 2003 [Sak03] est l'ouvrage de référence le plus récent présentant l'état de l'art de la théorie des automates.

Dans le théorème de Kleene, l'une des directions de l'équivalence indique qu'on peut utiliser les expressions régulières comme une syntaxe pour définir des automates. Dans le chapitre II, nous étudierons le modèle de calcul des machines d'Eilenberg qui sont définies à partir d'automates. On pourra donc utiliser les expressions régulières comme syntaxe pour décrire ces automates. Pour cette raison, dans le chapitre III nous présenterons différents algorithmes pour synthétiser des automates à partir d'expressions régulières. Avant de passer aux chapitres suivants nous dédions une section au problème de l'axiomatisation de l'égalité des langages réguliers, qui dans un cadre plus général forment une algèbre de Kleene.

2 Quelques axiomatisations des algèbres de Kleene

Certaines expressions régulières différentes en syntaxe peuvent dénoter des langages identiques. Par exemple, pour toutes expressions régulières E et F on a que $L(E+F) = L(F+E)$; il s'agit de la commutativité de la somme (ou de l'union) de deux langages. On a aussi l'associativité et l'idempotence de la somme. Concernant le produit, on a l'associativité et le fait que 0 est un élément absorbant. Toutes ces propriétés simples peuvent être mises en axiomes pour rendre compte de l'égalité des langages associés aux expressions régulières. On se pose ici la question de savoir quels sont les axiomes qui permettraient de trouver toutes les égalités entre expressions régulières ? Ces axiomes existent-ils en nombre fini ? S'ils existent, de quelle nature sont-ils ? La littérature est riche sur ce sujet qui n'est souvent que peu détaillé dans les livres de référence de la théorie des automates. Nous nous permettons dans cette thèse de faire un bref état de l'art du domaine en présentant les résultats principaux.

Une *algèbre de Kleene* est une structure $\mathcal{A} = (A, +, \cdot, ^*, 0, 1)$, avec A un ensemble muni d'opérations de même type que les expressions régulières et qui satisfait un ensemble d'axiomes d'égalité. Les algèbres de Kleene apparaissent dans de nombreux domaines tels que la sémantique et la logique des programmes, la théorie des automates et des langages, l'analyse d'algorithmes ou encore l'algèbre des relations. Les axiomes seront formulés dans un sous-ensemble de la logique du premier ordre avec égalité. L'axiomatisation est faite pour être valide et complète dans le modèle des langages réguliers, qu'on appelle le *modèle standard* et qui aura un rôle privilégié dans la suite.

Depuis les travaux initiaux de Kleene [Kle56] en 1956, la question de trouver une *axiomatisation* pour cette algèbre a suscité un grand intérêt. Par exemple on souhaiterait prouver que $((a+1)\cdot b^*)^* = (a+b)^*$ à l'aide d'un ensemble d'axiomes bien compris. On dit qu'une axiomatisation est *complète* lorsque celle-ci rend compte de toutes les égalités pour le modèle des langages réguliers. On dit qu'une axiomatisation est *finie* lorsqu'elle possède un nombre fini d'axiomes. On dit qu'une axiomatisation est *équationnelle* lorsque tous les axiomes sont de la forme $E = F$ pour E et F deux expressions régulières dont les symboles sont interprétés par des variables universellement quantifiées ; par exemple l'axiome de commutativité de l'opérateur $+$, noté $a+b = b+a$, doit être compris comme

$$\forall a\ b,\ a+b = b+a.$$

On dit qu'une axiomatisation est *quasi-équationnelle* lorsque les axiomes sont soit équationnels soit des équations conditionnelles (clauses de Horn) de la forme

$$(E_1 = {E_1}' \wedge \cdots \wedge E_n = {E_n}') \Rightarrow F = F'.$$

On notera donc E, F, E', *etc.* les expressions d'une algèbre de Kleene alors que dans les formules des théories axiomatiques on notera les variables universellement quantifiées par a, b, c, *etc.* Cela

peut paraître confus dans un premier temps, en particulier parce que les variables sont notées comme les symboles du modèle standard[2].

Récapitulons les principaux résultats concernant le problème d'axiomatisation des algèbres de Kleene. Redko [Red64] a montré en 1964 qu'il n'existait pas d'axiomatisation finie purement équationnelle. C'est Arto Salomaa [Sal66] qui proposa le premier en 1966 une axiomatisation finie complète. John Horton Conway contribua au sujet en 1971 avec son livre *Regular Algebra and Finite Machines* [Con71] en présentant les résultats connus à cette époque avec un regard original. Il proposa une preuve plus simple que celle de Redko pour la non finitude d'une axiomatisation équationnelle. Il proposa aussi un ensemble infini d'axiomes équationnels, et il utilisa aussi beaucoup dans ses développements le fait que les matrices sont munies d'une structure d'algèbre de Kleene. Cette correspondance est importante parce qu'elle permet d'utiliser la théorie des automates puisque les automates peuvent être vus comme des matrices. En s'inspirant des travaux de Conway, c'est Dexter Kozen [Koz94] qui proposera l'axiomatisation la plus communément utilisée : c'est une axiomatisation finie quasi-équationnelle, contrairement à celle de Salomaa dont l'un des axiomes n'est pas exactement une équation conditionnelle. Kozen prouva la complétude de son axiomatisation pour le modèle standard. Vaughan Pratt [Pra90] proposa en 1990 une axiomatisation finie purement équationnelle des algèbres de Kleene enrichies des deux opérateurs de résiduation ; il appelle une telle algèbre une *algèbre d'actions*. La théorie équationnelle associée aux algèbres d'actions est une extension conservative de la théorie de Kozen. Autant les algèbres de Kleene sont conçues pour rendre compte du modèle standard (interprétation sur les langages réguliers), autant les algèbres d'actions s'inspirent du *modèle relationnel*. Le modèle relationnel a pour éléments les relations binaires sur un ensemble D. La somme correspond à l'union de deux relations, le produit correspond à la composition de deux relations et l'étoile correspond à la traditionnelle fermeture réflexive transitive des relations. L'élément 0 est la relation vide et 1 est la relation identité.

Dans la suite nous rappelons les axiomatisations de Kozen et de Pratt. Puis nous donnerons des exemples d'algèbres de Kleene qui permettent de discuter l'indépendance des axiomes. Ensuite, dans un paragraphe complémentaire nous évoquerons l'axiomatisation équationnelle finie de Yanov [Yan62] qui est complète pour le modèle des langages de Yanov (langages réguliers clos par sous-mot) ; puis nous discuterons d'une axiomatisation pour une variante des algèbres de Kleene, à savoir les +-algèbres. Dans une +-algèbre l'étoile de Kleene, fermeture réflexive transitive du produit, est remplacée par la fermeture transitive (une opération plus *primitive*).

2.1 Axiomatisation des algèbres de Kleene par Kozen

Nous présentons l'axiomatisation de Kozen des algèbres de Kleene et rappelons le théorème de complétude de cette axiomatisation pour le modèle standard. L'axiomatisation est finie avec des axiomes équationnels et des équations conditionnelles simples de la forme $E = E' \Rightarrow F = F'$, c'est-à-dire que les équations conditionnelles n'ont qu'une seule prémisse. L'axiomatisation est rappelée en figure 1. Elle indique que $(A, +, 0)$ est un monoïde commutatif et idempotent. $(A, \cdot, 1)$ est un monoïde. La somme distribue sur le produit et 0 est un élément absorbant pour le produit. Il reste quatre autres axiomes relatifs à l'étoile de Kleene (S1, S2, S3 et S4). Deux d'entre eux indiquent que l'étoile est la fermeture réflexive et transitive du produit, et les deux autres sont des équations de point fixe. Seules ces deux équations de point fixe sont présentées sous forme d'équations conditionnelles, les autres axiomes sont purement équationnels. L'axiomatisation utilise la relation $\leq$ qui est simplement une notation pour $a \leq b \stackrel{\text{def}}{=} a + b = b$. Dans le modèle

[2]Nous avons choisi de suivre la notation des articles de Kozen et de Pratt.

$$a + (b + c) = (a + b) + c \quad \text{(Ax1)}$$
$$a + b = b + a \quad \text{(Ax2)}$$
$$a + 0 = a \quad \text{(Ax3)}$$
$$a + a = a \quad \text{(Ax4)}$$
$$a(bc) = (ab)c \quad \text{(Ax5)}$$
$$1a = a \quad \text{(Ax6)}$$
$$a1 = a \quad \text{(Ax7)}$$
$$a(b + c) = ab + ac \quad \text{(Ax8)}$$
$$(a + b)c = ac + bc \quad \text{(Ax9)}$$
$$0a = 0 \quad \text{(Ax10)}$$
$$a0 = 0 \quad \text{(Ax11)}$$

$$1 + aa^* \leq a^* \quad \text{(S1)}$$
$$1 + a^*a \leq a^* \quad \text{(S2)}$$
$$ab \leq b \Rightarrow a^*b \leq b \quad \text{(S3)}$$
$$ba \leq b \Rightarrow ba^* \leq b \quad \text{(S4)}$$

FIG. 1 – L'axiomatisation des algèbres de Kleene à la Kozen

standard, la relation $\leq$ correspond à l'inclusion ensembliste des langages réguliers. La relation $\leq$ est un préordre.

En guise d'échauffement, nous montrons quelques lemmes faciles. On commence par l'associativité de la relation $\leq$ avec le lemme suivant :

Lemme 1. Pour toutes expressions régulières a, b et c, on a

$$a \leq b \wedge b \leq c \Rightarrow a \leq c.$$

Démonstration.

$a + b = b$	hypothèse
$b + c = c$	hypothèse
$(a + b) + c = b + c$	
$a + (b + c) = b + c$	axiome Ax1
$a + c = c$	par réécriture de la 2e hypothèse

□

Montrons que la relation $\leq$ est antisymétrique dans le lemme suivant :

Lemme 2. Pour toutes expressions régulières a et b, on a

$$a = b \text{ si et seulement si } a \leq b \text{ et } b \leq a.$$

Démonstration. On démontre l'équivalence en prouvant les deux implications :

1. $\Rightarrow$. On suppose que $a = b$ alors $a + b = b + b$ et avec l'idempotence du + (Ax4), on obtient $a + b = b$ donc $a \leq b$. L'autre inégalité s'obtient de la même manière par symétrie.
2. $\Leftarrow$. On suppose que $a \leq b$ et $b \leq a$. De manière indépendante, en utilisant la commutativité du + (Ax2) on a $a + b = b + a$ et alors en utilisant les hypothèses on obtient $b = a$.

□

La relation $\leq$ est monotone pour les opérateurs $+$, $\cdot$ et *.

Lemme 3 (monotonicité)**.**

$$a \leq b \Rightarrow a + c \leq b + c \quad (3)$$
$$a \leq b \Rightarrow a \cdot c \leq b \cdot c \quad (4)$$
$$a \leq b \Rightarrow a^* \leq b^* \quad (5)$$

Démonstration.

– $a \leq b \Rightarrow a + c \leq b + c$

$a + b = b$	définition de $\leq$
$(a + b) + c = b + c$	
$(a + b) + (c + c) = b + c$	idempotence de $+$
$(a + c) + (b + c) = b + c$	associativité-commutativité de $+$
$a + c \leq b + c$	définition de $\leq$

– $a \leq b \Rightarrow a \cdot c \leq b \cdot c$

$a + b = b$	définition de $\leq$
$(a + b) \cdot c = b \cdot c$	
$ac + bc = bc$	distributivité
$ac \leq bc$	

– $a \leq b \Rightarrow a^* \leq b^*$

	$a \leq b$	définition de $\leq$
	$ab^* \leq bb^*$	monotonicité de $\cdot$
or	$bb^* \leq b^*$	axiome S1
donc	$ab^* \leq b^*$	transitivité de $\leq$
	$a^*b^* \leq b^*$	axiome S3
or	$1 \leq b^*$	axiome S1
donc	$a^* \leq a^*b^*$	monotonicité de $\cdot$ et neutralité de 1
	$a^* \leq b^*$	transitivité de $\leq$

□

Exemple 1. Montrons que dans une algèbre de Kleene on a $0^* = 1$.
Pour cela établissons l'inégalité dans les deux sens et on conclut par antisymétrie de $\leq$:

1. $1 \leq 0^*$. Cela découle de l'inégalité S1.
2. $0^* \leq 1$. Tout d'abord on peut établir que $0 \cdot 1 \leq 0 \leq 1$, et ensuite, en utilisant l'axiome d'équation conditionnelle S3 on en déduit que $0^* \cdot 1 \leq 1$ et on conclut que $0^* \leq 1$.

Voici deux lemmes utiles pour la suite.

Lemme 4. Dans toute algèbre de Kleene on a

$$ab^* \leq b^* \Rightarrow a^* \leq b^*.$$

Démonstration.

	$ab^* \leq b^*$	par hypothèse
	$a^*b^* \leq b^*$	axiome S3
or	$1 \leq b^*$	axiome S1
	$a^*1 \leq a^*b^*$	monotonicité de $\cdot$
	$a^* \leq b^*$	neutralité et transitivité

□

Lemme 5. Dans toute algèbre de Kleene on a

$$a^*a^* \leq a^*.$$

Démonstration.

	$1 + aa^* \leq a^*$	axiome S1
or	$aa^* \leq 1 + aa^*$	
	$aa^* \leq a^*$	transitivité de $\leq$
donc	$a^*a^* \leq a^*$	axiome S3

□

Le prochain exemple établit une égalité calculée par une transformation d'expression régulière qui correspond à la *forme normale d'étoile* ; nous détaillerons cette transformation plus loin dans cette thèse, prouvons maintenant que l'égalité est dérivable à partir des axiomes.

Exemple 2. En notant $b^\epsilon \stackrel{\text{def}}{=} (b+1)$, dans toute algèbre de Kleene on a

$$(a^*b^\epsilon)^* = (a + b^\epsilon)^*.$$

On remarque qu'une étoile disparaît et qu'un produit est transformé en somme. Pour prouver cette égalité, établissons l'inégalité dans les deux sens et utilisons le lemme 2.

- $(a^*b^\epsilon)^* \leq (a + b^\epsilon)^*$:

	$(a + b^\epsilon)^*(a + b^\epsilon)^* \leq a + b^\epsilon$	lemme 5
	$(a + b^\epsilon)^*(a + b^\epsilon)^*(a + b^\epsilon)^* \leq (a + b^\epsilon)^*$	lemme 5
or	$a \leq a + b^\epsilon$	
et	$b^\epsilon \leq a + b^\epsilon$	
donc on a	$a^*b^\epsilon(a + b^\epsilon)^* \leq (a + b^\epsilon)^*$	monotonicité
finalement	$(a^*b^\epsilon)^* \leq (a + b^\epsilon)^*$	lemme 4

- $(a + b^\epsilon)^* \leq (a^*b^\epsilon)^*$: On a la suite d'inéquations suivante :

	$a + b^\epsilon \leq ab + a + b^\epsilon = (a+1)b^\epsilon$	
et aussi	$(a+1)b^\epsilon \leq a^*b^\epsilon$	monotonicité et axiome S1
	$a + b^\epsilon \leq a^*b^\epsilon$	par transitivité de $\leq$
et donc	$(a + b^\epsilon)^* \leq (a^*b^\epsilon)^*$	monotonicité de l'étoile

On notera $\vdash E = F$ lorsqu'on peut dériver par déduction quasi-équationnelle de l'axiomatisation l'égalité $E = F$.

Théorème I.2 (Kozen [Koz94])**.** L'axiomatisation des algèbres de Kleene est *complète* pour l'égalité dans le modèle des langages réguliers.

$$\vdash E = F \text{ si et seulement si } L(E) = L(F).$$

Démonstration. Ce théorème est le sujet de l'article de Kozen *"A completeness theorem for Kleene algebras and the regular events"* [Koz94]. Kozen prouve le théorème en recodant des opérations classiques sur les automates sous une forme algébrique en utilisant les matrices : on sait que deux expressions régulières sont égales si et seulement si les automates minimaux associés sont égaux. On peut prouver en utilisant les axiomes qu'une expression régulière et son automate minimal associé sont équivalents. Pour ce faire, on utilise une représentation matricielle des automates avec des coefficients qui sont des expressions régulières. Cette structure algébrique de matrices munie d'opérations adéquates forme elle-même une algèbre de Kleene. On peut donc utiliser l'axiomatisation pour valider les passes successives qui permettent de construire l'automate minimal ; à savoir, la traduction d'expressions en ϵ-NFA (algorithme de Thompson [Tho68]), la déterminisation et pour finir la minimisation. □

Dans de récents travaux, Braibant et Pous [BP09] ont présenté une formalisation complète dans l'assistant de preuves Coq du théorème précédent. Ce développement fournit une procédure de décision pour prouver de manière automatique que deux éléments d'une algèbre de Kleene sont équivalents selon les axiomes.

2.2 Axiomatisation des algèbres d'actions de Pratt

Une *algèbre d'actions* est une algèbre $\mathcal{A} = (A, +, 0, \cdot, 1, \rightarrow, \leftarrow, ^*)$ telle que $(A, +, 0)$ et $(A, \cdot, 1)$ sont des monoïdes, l'opération $+$ étant commutative et idempotente, et satisfaisant les axiomes suivants :

$$a \leq c \leftarrow b \quad \overset{L}{\Leftrightarrow} \quad ab \leq c \quad \overset{R}{\Leftrightarrow} \quad b \leq a \rightarrow c \tag{ACT1}$$

$$1 + a^*a^* + a \leq a^* \tag{ACT2}$$

$$1 + bb + a \leq b \quad \Rightarrow \quad a^* \leq b \tag{ACT3}$$

Les équivalences de ACT1 axiomatisent les opérations de résiduation $\leftarrow$ et $\rightarrow$; ACT1 comprend deux axiomes d'équivalence, l'équivalence ACT1L axiomatise la résiduation à gauche et l'équivalence ACT1R la résiduation à droite. Les axiomes ACT2 et ACT3 sont de même nature que les quatre axiomes pour l'étoile (S1, S2, S3 et S4) de l'axiomatisation de Kozen, avec une volonté de les fusionner deux à deux.

Le modèle standard et le modèle relationnel sont aussi des modèles valides pour les algèbres d'actions. En effet sur les langages on choisit les opérations de résiduation :
- $L_1 \rightarrow L_2 = \{v \mid \forall u, u \in L_1 \Rightarrow uv \in L_2\}$,
- $L_2 \leftarrow L_1 = \{v \mid \forall u, u \in L_1 \Rightarrow vu \in L_2\}$.

Et pour le modèle relationnel on choisit les opérations de résiduation suivantes :
- $R \rightarrow S = \{(v, w) \mid \forall u \in D, uRv \Rightarrow uSw\}$,
- $S \leftarrow R = \{(u, v) \mid \forall w \in D, vRw \Rightarrow uSw\}$.

Pour l'instant une algèbre d'actions se présente sous la forme d'une théorie quasi-équationnelle, nous allons voir qu'on peut la présenter sous forme purement équationnelle. C'est l'avantage qu'ont les algèbres d'actions par rapport aux algèbres de Kleene : l'ajout des opérations de résiduation permet de changer la présentation axiomatique de la théorie.

Tout d'abord on remarque que les axiomes de distibutivité sont absents de l'axiomatisation. En effet nous allons montrer qu'on peut déduire la distributivité de la multiplication sur l'addition à partir des axiomes ACT1L et ACT1R. On utilisera le principe suivant :

$$a \leq c \Leftrightarrow b \leq c \text{ implique } a = b.$$

Ainsi pour prouver la distributivité à gauche $a(b+c) = ab+ac$ on fait le raisonnement suivant :

$$\begin{aligned} a(b+c) \leq d &\Leftrightarrow (b+c) \leq a \to d && \text{par l'axiome ACT1L} \\ &\Leftrightarrow b \leq a \to d \ \wedge \ c \leq a \to d \\ &\Leftrightarrow ab \leq d \ \wedge \ ac \leq d \\ &\Leftrightarrow ab + ac \leq d \end{aligned}$$

On effectue un raisonnement similaire en utilisant l'opération $\leftarrow$ pour prouver la distributivité à droite $(a+b)c = ac+bc$. Maintenant nous allons montrer plusieurs propriétés importantes :

	$b \leq a \to ab$	ACT1R avec $c = ab$	(ACT6)
	$a(a \to b) \leq b$	ACT1R avec $b = a \to c$	(ACT7)
	$a(a \to b)(b \to c) \leq b(b \to c)$	multiplier par $(b \to c)$	
or	$b(b \to c) \leq c$	instance de ACT7	
donc	$a(a \to b)(b \to c) \leq c$	transitivité de $\leq$	
	$(a \to b)(b \to c) \leq a \to c$		
cas particulier	$(a \to a)(a \to a) \leq a \to a$		(6)
Reflexivité	$1 \leq a \to a$		(7)

En utlisant la clause de Horn ACT3 pour laquelle on substitue à a et à b l'expression $a \to a$ et en utilisant 6 et 7 on obtient l'équation suivante :

$$(a \to a)^* \leq a \to a \qquad \text{(ACT13)}$$

Pratt appelle l'équation ACT13 la *récursion pure* (en anglais, *pure induction*). Cet axiome peut paraître surprenant parce qu'il indique qu'une expression étoilée est plus petite que cette expression sans étoile. Nous vérifierons cet axiome pour le modèle des langages réguliers dans la suite.

En s'assurant que les axiomes ACT2 et ACT3 sont bien équivalents aux quatre axiomes S1, S2, S3 et S4 alors on a le théorème suivant :

Théorème I.3 (Pratt [Pra90], Theorem 5)**.** Pour toutes expressions E et F définies avec les opérateurs $+$, $\cdot$ et * et sur les générateurs Σ on a :

$$\vdash_{\text{Kozen}} E = F \qquad \Leftrightarrow \qquad \vdash_{\text{Pratt}} E = F$$

La théorie équationnelle des algèbres d'actions est donc une *extension conservative* de la théorie équationnelle des algèbres de Kleene.

Pour toute théorie équationnelle, la classe des modèles la satisfaisant est appelée *variété*. Pour toute théorie qui n'est pas axiomatisable de manière purement équationnelle mais avec des clauses de Horn, la classe des modèles la satisfaisant est appelée *quasi-variété*. On désigne par ACT la classe des modèles qui sont des algèbres d'actions. ACT est définie de manière quasi-équationnelle donc *a priori* ACT est une quasi-variété, cependant Pratt a prouvé qu'on pouvait axiomatiser ACT avec un ensemble fini d'axiomes purement équationnels, ceux de la figure 2. On a donc le théorème suivant :

Théorème I.4 (Pratt [Pra90], Theorem 7)**.** ACT est une variété finiment présentée.

Démonstration. Les axiomes de la figure 2 sont équivalents à ceux des algèbres d'actions. Les équivalences sont toutes prouvées clairement dans [Pra90] sans plus de technique que celles que nous avons rappelées précédemment. □

$$a + (b + c) = (a + b) + c$$
$$a + b = b + a$$
$$a + 0 = a$$
$$a + a = a$$
$$a(bc) = (ab)c$$
$$1a = a$$
$$a1 = a$$
$$ab \leq (a + a')(b + b') \qquad \text{(ACT4)}$$
$$0a = 0$$
$$a0 = 0$$

$$a \to b \leq a \to (b + b') \qquad \text{(ACT5)}$$
$$b \leq a \to ab \qquad \text{(ACT6)}$$
$$a(a \to b) \leq b \qquad \text{(ACT7)}$$
$$b \leftarrow a \leq (b + b') \leftarrow a \qquad \text{(ACT8)}$$
$$b \leq ba \leftarrow a \qquad \text{(ACT9)}$$
$$(b \leftarrow a)a \leq b \qquad \text{(ACT10)}$$
$$1 + a^*a^* + a \leq a^* \qquad \text{(ACT11)}$$
$$a^* \leq (a + b)^* \qquad \text{(ACT12)}$$
$$(a \to a)^* \leq a \to a \qquad \text{(ACT13)}$$

FIG. 2 – Axiomatisation équationnelle des algèbres d'actions de Pratt

Validité du modèle standard

Afin d'avoir une meilleure compréhension de ces axiomes, nous discutons de la validité du modèle standard, c'est-à-dire de l'interprétation sur les langages réguliers ; on se limitera aux axiomes ACT5, ACT6, ACT7 et ACT13. Des arguments symétriques permettent de valider les autres axiomes. Les variables a, b, c, *etc.* désignent maintenant des langages. Rappelons que l'opération de résiduation droite sur les langages est définie par $b \to a = \{v \mid \forall u, u \in b \Rightarrow uv \in a\}$.

- ACT5 :

  $$a \to b \leq a \to (b + b')$$

  Soit $w \in a \to b$, on a $\forall u \in a,\ uw \in b$, on en déduit que $\forall u \in a,\ uw \in b + b'$ et finalement $w \in a \to (b + b')$.
- ACT6 :

  $$b \leq a \to ab$$

  Soit $w' \in b$, par définition $a \to ab = \{w \mid \forall u, u \in a \Rightarrow\ uw \in ab\}$ donc $w' \in a \to ab$. L'exemple suivant montre que l'inégalité est bien nécessaire, c'est-à-dire que l'égalité seule ne suffit pas. On donne l'interprétation suivante aux symboles a et b de l'équation ACT6 : soient x et y deux symboles d'un alphabet Σ, on considère l'interprétation suivante $a \mapsto x^*$ et $b \mapsto y$. Alors on a :

  $$x^* \to x^*y \ = \ x^*y.$$

  Or $y \ \leq \ x^*y$, il existe donc des mots du langage de droite, par exemple $xxxxxy$, qui ne sont pas dans le langage de gauche.
- ACT7 :

  $$a(a \to b) \leq b$$

  Soit $w \in a(a \to b)$ alors $w = w_1w_2$ avec $w_1 \in a$ et $w_2 \in a \to b$. Or pour tout $w' \in a$, par définition du résidu à droite on a $w'w_2 \in b$, et donc en particulier on a $w_1w_2 \in b$.
  L'exemple suivant montre que l'inégalité est bien nécessaire, c'est-à-dire que l'égalité seule ne suffit pas. On donne l'interprétation suivante aux symboles a et b de l'équation ACT7 : soient x, y et z trois symboles d'un alphabet Σ, on considère l'interprétation suivante

$a \mapsto x$ et $b \mapsto xy + z$. Le langage $x \to (xy + z)$ est égal à y donc on a

$$x(x \to (xy + z)) \; = \; xy$$

Donc z est dans le langage de droite mais pas dans celui de gauche, il est donc nécessaire que l'inégalité ne soit pas une égalité.

– ACT13 :

$$(a \to a)^* \leq a \to a$$

Soit $w \in (a \to a)^*$, alors par définition de l'étoile sur les langages w est une concaténation de sous-mots de $a \to a$, on a donc :

$$w = v_1 \cdots v_n, \; \forall i \leq n, \; v_i \in a \to a$$

Puisque $v_i \in a \to a$ on a la propriété

$$P_i : \forall u \in a, uv_i \in a.$$

Soit $u \in a$, en utilisant successivement les propriétés P_1 jusqu'à P_n et l'associativité de la concaténation à chaque étape de raisonnement on obtient la suite de propriétés suivante :

$$\left| \begin{array}{ccccc} P_1(u) & \text{implique} & uv_1 \in a & & \\ P_2(uv_1) & \text{implique} & (uv_1)v_2 \in a & \text{donc} & u(v_1v_2) \in a \\ \vdots & \vdots & & \vdots & \\ P_n(uv_1 \cdots v_{n-1}) & \text{implique} & (uv_1 \cdots v_{n-1})v_n \in a & \text{donc} & u(v_1 \cdots v_{n-1}v_n) \in a \end{array} \right|$$

On a donc prouvé que pour tout $u \in a, \; u(v_1 \cdots v_n) = uw \in a$. Donc $w \in a \to a$.

2.3 Exemples

En donnant des exemples d'algèbres de Kleene et d'algèbres d'actions on discutera de l'indépendance des axiomes des théories associées. Des exemples aussi bien finis qu'infinis nous permettront de récapituler les résultats essentiels.

Exemples finis

Nous allons énumérer les exemples finis de dimension plus petite que quatre ; on entend par dimension le nombre d'éléments de l'ensemble qu'on considère. Ensuite nous exhiberons un exemple dû à Conway, appelé *saut de Conway*, qui contient seulement quatre éléments satisfaisant l'équation de plus petit point fixe de l'étoile S1 mais invalidant l'équation conditionnelle S3. Les exemples suivants sont dûs à Conway [Con71]. Il les donna sous forme de diagramme de Hasse en remarquant que la somme avec $\leq$ a la structure d'un demi-treillis. On remarque que l'étoile d'un élément a est le plus petit élément étoilé supérieur à a (c'est-à-dire $a \leq b^* \Rightarrow a^* \leq b^*$ par monotonicité du produit). On dit qu'un élément a est *étoilé* s'il existe un élément b tel que $a = b^*$, la remarque précédente indique qu'on peut marquer d'une étoile les éléments étoilés. Il reste juste à définir une table de multiplication pour le produit. Les éléments des exemples seront représentés par des entiers **0**, **1**, **2**, **3**, *etc.* qui sont typographiés en caractères gras pour les différencier des éléments neutres 0 et 1.

One $K = \{\mathbf{1}\}$
$0 = \mathbf{1}^* = \mathbf{1}$, c'est le modèle trivial.

Two $K = \{\mathbf{0}, \mathbf{1}\}$
$\mathbf{0}^* = \mathbf{1}$

Three** $K = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$
$\mathbf{0} < \mathbf{1}^* < \mathbf{2}^*$ La seule table de multiplication satisfaisant les axiomes est $\mathbf{2} \cdot \mathbf{2} = \mathbf{2}$.

Three* $K = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$
$\mathbf{0} < \mathbf{2} < \mathbf{1}^*$. On définit l'algèbre **Three*.a** en choisissant $\mathbf{2} \cdot \mathbf{2} = \mathbf{2}$, et on définit **Three*.b** en choisissant $\mathbf{2} \cdot \mathbf{2} = \mathbf{0}$. On ne peut choisir $\mathbf{2} \cdot \mathbf{2} = \mathbf{1}$ car puisque $\mathbf{2} \leq \mathbf{1}$ alors on aurait la contradiction $\mathbf{2} \cdot \mathbf{2} \leq \mathbf{1} \leq \mathbf{2} = \mathbf{2}$.
Remarquons que l'algèbre **Three*.b** est la première pour laquelle le produit n'est pas idempotent. Jusqu'à présent les exemples ont tous un produit commutatif. Voici un exemple d'algèbre à quatre éléments pour laquelle le produit n'est ni commutatif ni idempotent :

Four.a $K = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$
$\mathbf{0} < \mathbf{2} < \mathbf{3} < \mathbf{1}^*$ avec $\mathbf{2} \cdot \mathbf{2} = \mathbf{3} \cdot \mathbf{2} = \mathbf{0}$, $\mathbf{2} \cdot \mathbf{3} = \mathbf{2}$ et $\mathbf{3} \cdot \mathbf{3} = \mathbf{3}$.
Jusqu'à maintenant l'ordre partiel est un ordre total linéaire. Voici une algèbre de Kleene à 4 éléments pour laquelle l'ordre n'est pas total.

Four.b $K = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$
$\mathbf{0} < \mathbf{2}$, $\mathbf{0} < \mathbf{3}$, $\mathbf{2} < \mathbf{1}^*$, $\mathbf{3} < \mathbf{1}^*$ avec $\mathbf{2} \cdot \mathbf{2} = \mathbf{2}$, $\mathbf{2} \cdot \mathbf{3} = \mathbf{3} \cdot \mathbf{2} = \mathbf{0}$, $\mathbf{3} \cdot \mathbf{3} = \mathbf{3}$.

On pourrait continuer à lister les algèbres finies mais nous nous arrêtons là en montrant une algèbre finie supplémentaire qui contient seulement quatre éléments et qui satisfait tous les axiomes équationnels mais pas les conditions équationnelles ; on appelle cette algèbre le *saut de Conway* :

Saut de Conway $K = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$
$\mathbf{0} < \mathbf{1}^* < \mathbf{2} < \mathbf{3}^*$ avec $x \cdot y \stackrel{\text{def}}{=}$ si $x = \mathbf{0}$ ou $y = \mathbf{0}$ alors $\mathbf{0}$ sinon $max(x, y)$, c'est-à-dire que $\mathbf{2} \cdot \mathbf{2} = \mathbf{2}$ et que $\mathbf{2} \cdot \mathbf{3} = \mathbf{3} \cdot \mathbf{2} = \mathbf{3} \cdot \mathbf{3} = \mathbf{3}$. Cette algèbre vérifie les axiomes $1 \leq a^*$ et $aa^* \leq a^*$, mais ne vérifie pas l'axiome S3 $ab \leq b \Rightarrow a^*b \leq b$ (prendre $a = b = \mathbf{2}$). Donc le saut de Conway n'est pas une algèbre de Kleene.

Pratt a montré ([Pra90], Theorem 6) que toute algèbre de Kleene finie peut s'étendre en une algèbre d'actions ; il suffit de définir $a \to c = \sum_{ab \leq c} b$, c'est une somme finie. On a immédiatement que $ab \leq c \Rightarrow b \leq a \to c$. Pour l'autre direction en fixant a et c on a

$$\begin{aligned} a \cdot \sum_{ab \leq c} b &= \sum_{ab \leq c} a \cdot b \\ &\leq c \end{aligned}$$

et si $b \leq a \to c = \sum_{ab \leq c} b$ alors $ab \leq c$. On définirait l'opération $\leftarrow$ de manière analogue.

Exemples infinis

Notre premier exemple infini, et la principale motivation pour l'étude de ces algèbres, est l'ensemble des expressions régulières sur un alphabet Σ, quotienté par l'équivalence des langages engendrés associée. Pour tout Σ, ceci définit l'algèbre de Kleene libre engendrée par Σ. Un modèle isomorphe est l'ensemble des automates finis déterministes minimaux, munis des opérations idoines. Cette dernière présentation est canonique (l'égalité est l'identité). En fait il s'agit là de

familles de modèles, il y en a un par taille de Σ. Par exemple, pour Σ vide, on retrouve l'exemple **Two** des Booléens, le modèle libre n'est infini que pour Σ non vide.

Parmi les exemples moins directs, nous reprenons dans un premier temps un exemple dû à Pratt [Pra90] qui permet de discuter de l'indépendance des axiomes S3, S4, ACT3 d'une part, et de discuter du fait que les algèbres de Kleene sont une quasi-variété et non une variété (en exploitant le saut de Conway). Dans un second temps nous donnerons un exemple dû à Kozen qui montre qu'une algèbre de Kleene n'est pas nécessairement $*$-continue (propriété de continuité vraie dans le modèle standard mais fausse dans le cas général).

On considère l'algèbre d'actions $(\mathbb{N}^+, max, \bot, +, \mathbf{0}, \dot{-}, \dot{-}, {}^*)$ avec $\mathbb{N}^+ = \{\bot, \mathbf{0}, \mathbf{1}, \mathbf{2}, ..., \top\}$ l'ensemble des entiers naturels augmenté de deux éléments $\bot$ et $\top$. L'opération $\dot{-}$ est la soustraction tronquée ($\mathbf{7} \dot{-} \mathbf{3} = \mathbf{4}$, $\mathbf{3} \dot{-} \mathbf{7} = \mathbf{0}$). L'addition est commutative (c'est pour cette raison que les deux résiduations sont les même opérations) et satisfait $\bot i = \bot$ et pour $i \neq \bot$ on prend $i\top = \top$. L'opération * est définie par $\bot^* = \mathbf{0}^* = \mathbf{0}$ et $i^* = \top$ pour tout $i \geq 1$ ou $i = \top$.

Nous avons vu que toute algèbre d'actions est une algèbre de Kleene. L'exemple précédent permet de montrer de manière élégante que les algèbres de Kleene ne peuvent pas être axiomatisées par une liste finie d'axiomes équationnels. En effet la classe des modèles d'algèbres finiment axiomatisées équationnellement (variété finiment présentée) est fermée par homomorphisme, c'est une propriété importante de ces familles de modèles. Cependant il existe un homomorphisme de l'algèbre $(\mathbb{N}^+, max, \bot, +, \mathbf{0}, {}^*)$ vers le saut de Conway (à quatre éléments $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \}$), qui n'est pas une algèbre de Kleene. L'homomorphisme h a considérer est le suivant : $h(\bot) = \mathbf{0}$, $h(\mathbf{0}) = \mathbf{1}$, $h(\top) = \mathbf{3}$ et $h(i) = \mathbf{2}$ pour tout entier $i \geq \mathbf{1}$. On vérifiera que h est bien un homomorphisme, c'est-à-dire $h(a+b) = h(a) + h(b)$, $h(a \cdot b) = h(a) \cdot h(b)$ et $h(a^*) = (h(a))^*$.

Continuons avec cet exemple sur $\mathbb{N}^+$. On se donne un élément supplémentaire ∞ tel que $i \leq \infty \leq \top$ pour tout i fini ; on déduit à partir de de cet ordre l'extension du max. Par monotonicité on a que $\infty^* = \top$; donc l'étoile associe $\mathbf{0}$ à $\bot$ et à $\mathbf{0}$, et associe $\top$ à tous les autres éléments.

Si on impose $i\infty = j\infty$ et $\infty i = \infty j$ pour tous entiers i et j, alors on déduit que la table du produit dépend uniquement du choix de $\mathbf{1}\infty$, $\infty\infty$ et $\infty\mathbf{1}$ qui peuvent être tous égaux à ∞ ou $\top$; il y aurait donc huit tables possibles pour le produit. Cependant si $\infty\infty = \infty$ alors par monotonicité les deux autres valent aussi ∞. Il y a donc seulement cinq choix possibles qui correspondent à cinq algèbres différentes. On remarque que tous les axiomes des algèbres de Kleene, mis à part S3 et S4, sont valides pour ces cinq algèbres.

L'axiome ACT3 n'est valide que si $\infty\infty = \top$, l'axiome S3 n'est valide que si $\mathbf{1}\infty = \top$ et l'axiome S4 n'est valide que si $\infty\mathbf{1} = \top$. Cela permet de montrer l'indépendance des axiomes S3 et S4 lorsqu'on choisit une algèbre telle que $\infty\mathbf{1} \neq \mathbf{1}\infty$. Donc la seule algèbre qui soit une algèbre de Kleene sur les cinq possibles est celle pour laquelle $\infty\mathbf{1} = \mathbf{1}\infty = \top$.

Cependant cette algèbre de Kleene n'est pas une algèbre d'actions. En effet lorsque $\mathbf{1}\infty = \top$ alors $\mathbf{1} \rightarrow \infty$ est non-défini ; et symétriquement lorsque $\infty\mathbf{1} = \top$ alors $\infty \leftarrow \mathbf{1}$ est non-défini. Donc la seule algèbre de Kleene sur les cinq ne peut pas s'étendre en une algèbre d'actions.

Une algèbre de Kleene est dite $*$-*continue* lorsqu'elle vérifie l'identité suivante :

$$ab^*c = \sum_n ab^n c$$

avec $\sum$ défini comme le supremum induit par $\leq$. Nous allons donner un exemple dû à Kozen [Koz90] d'une algèbre de Kleene qui n'est pas $*$-continue. Soit ω^2 l'ensemble de paires d'entiers auquel on ajoute deux nouveaux éléments $\bot$ et $\top$. On ordonne ces éléments de la manière suivante : $\bot$ est le plus petit élément, $\top$ est le plus grand élément, et ω^2 est ordonné lexicographiquement. On

$$a + a = a$$
$$a + b = b + a$$
$$(a + b) + c = a + (b + c)$$
$$(ab)c = a(bc)$$
$$a(b + c) = ab + ac$$
$$(a + b)c = ac + bc$$
$$ab + b = ab \quad (8)$$
$$ab + a = ab \quad (9)$$
$$a^{**} = a^* \quad (10)$$
$$a^*a^* = a^* \quad (11)$$
$$(a + b)^* + a = (a + b)^* \quad (12)$$
$$(a + b)^* = (a^* + b)^* \quad (13)$$
$$(a + b)^* = (ab)^* \quad (14)$$
$$(ab)^*a = (ab)^* \quad (15)$$
$$a(ab)^* = (ab)^* \quad (16)$$

FIG. 3 – Axiomatisation de Yanov

définit l'opération $+$ comme le supremum pour cet ordre. On définit le produit $\cdot$ de la manière suivante :

$$\begin{aligned} a \cdot \bot = \bot \cdot a &= \bot \\ a \cdot \top = \top \cdot a &= \top, \ x \neq \bot \\ (a, b) \cdot (c, d) &= (a + c, b + d). \end{aligned}$$

L'élément $\bot$ est l'identité pour la somme et $(0, 0)$ est l'identité pour le produit. On définit aussi

$$a^* = \begin{cases} (0, 0) & , \text{ si } a = \bot \text{ ou } a = (0, 0) \\ \top & , \text{ sinon.} \end{cases}$$

On peut facilement vérifier que cette algèbre est une algèbre de Kleene. En revanche ce n'est pas une algèbre $*$-continue car $(0, 1)^* = \top$ mais

$$\sum_n (0, 1)^n = \sum_n (0, n) = (1, 0).$$

2.4 Compléments

Dans cette partie de compléments nous présentons une axiomatisation équationnelle finie pour les langages de Yanov, puis nous discuterons d'une variante des algèbres de Kleene, à savoir les $+$-algèbres dont l'étoile de Kleene (fermeture réflexive transitive du produit) est remplacée par l'opérateur $+$ (fermeture transitive du produit) qui est plus primitif.

Axiomatisation de Yanov

Nous appelons algèbre de Yanov une structure $(A, +, \cdot, ^*)$ telle que $+$ est associatif, commutatif et idempotent, et $\cdot$ est associatif, et vérifiant les axiomes de la figure 3. L'axiomatisation est finie et équationnelle. Cette axiomatisation est complète pour un modèle particulier de langages (inclus dans les langages réguliers), à savoir les langages dont les générateurs sont les langages de la forme $\{\epsilon, a\}$ pour tout symbole $a \in \Sigma$ et fermés par union, concaténation et étoile de Kleene. On les appelle les *langages de Yanov*. Par exemple l'expression $((a + b)cd)^*$ est interprétée par $((a^\epsilon + b^\epsilon)c^\epsilon d^\epsilon)^*$. On peut facilement retrouver les axiomes de la figure 3 à partir de ceux de Kozen en supposant que $1 \leq a$, $1 \leq b$ et $1 \leq c$. La propriété fondamentale des langages de Yanov est

$$a + (b + c) = (a + b) + c \qquad a \leq a^+ \quad \text{(P1)}$$
$$a + b = b + a \qquad aa^+ \leq a^+ \quad \text{(P2)}$$
$$a + 0 = a \qquad a^+a \leq a^+ \quad \text{(P3)}$$
$$a + a = a \qquad ab \leq b \Rightarrow a^+b \leq b \quad \text{(P4)}$$
$$a(bc) = (ab)c \qquad ba \leq b \Rightarrow ba^+ \leq b \quad \text{(P5)}$$
$$1a = a$$
$$a1 = a$$
$$a(b + c) = ab + ac$$
$$(a + b)c = ac + bc$$
$$0a = 0$$
$$a0 = 0$$

FIG. 4 – Axiomatisation des +-algèbres régulières

la suivante : soit L un langage de Yanov, alors pour tout mot $w = a_1 \cdots a_n \in L$, tout sous-mot $w' = a_{i_1} \cdots a_{i_m}$ avec $m < n$ et $i_1 < \cdots < i_m$ est tel que w' appartient aussi à L.

Cette axiomatisation a été proposée originellement par Yanov [Yan62] en 1962, Conway y fait référence dans son livre [Con71]. Plus récemment, Dolinka [Dol03] a étudié le fragment de ces algèbres n'utilisant pas l'opération de somme + dans le but d'obtenir des formes canoniques.

Axiomatisation des +-algèbres

L'étoile de l'algèbre de Kleene est utilisée pour définir une notion d'itération. Cette itération est définie comme la fermeture réflexive et transitive du produit. Il n'y a pas de raison fondamentale à considérer ces deux fermetures simultanément, on peut définir une autre opération qui définit une itération comme seulement la fermeture transitive du produit et retrouver l'étoile de Kleene à partir de cette nouvelle opération.

On note cette opération par la mise en exposant $^+$ comme a^+. Nous proposons une axiomatisation à la Kozen fournie en figure 4 pour cette algèbre légèrement différente de la traditionnelle algèbre de Kleene. Dans ce cas nous parlons non plus d'une algèbre de Kleene mais d'une *+-algèbre* $(A, +, \cdot, ^+, 0, 1)$.

Intuitivement, dans cette algèbre on peut définir l'étoile de Kleene à partir de l'opérateur $^+$ par $a^* \stackrel{\text{def}}{=} 1 + a^+$. Et réciproquement dans l'algèbre de Kleene on peut définir $a^+ \stackrel{\text{def}}{=} a \cdot a^*$. On constate donc que la +-algèbre est un peu plus expressive que l'algèbre de Kleene : en effet la définition de a^+ dans l'algèbre de Kleene nécessite de *dupliquer* l'expression a dans $a \cdot a^*$.

On montre un théorème d'équivalence des théories équationnelles d'une algèbre de Kleene axiomatisée à la Kozen et de sa +-algèbre correspondante.

Théorème I.5. L'axiomatisation de la +-algèbre et l'axiomatisation de Kozen de l'algèbre de Kleene sont équivalentes.

Démonstration. On vérifie l'équivalence des axiomes suivants : P1, P2, P3, P4 et P5 sont respectivement équivalents à S1, S2, S3 et S4.

- $a \leq a^+$: On veut montrer $a \leq aa^*$. On a $1 \leq a^*$ donc en multipliant à gauche par a on obtient $a1 \leq aa^*$ et donc $a \leq aa^*$.

- $aa^+ \leq a^+$: On veut montrer $a(aa^*) \leq aa^*$. On a $aa^* \leq 1 + aa^* \leq a^*$ en multipliant à gauche par a on obtient $a(aa^*) \leq aa^*$.
- $ab \leq b \Rightarrow a^+b \leq b$: On suppose $ab \leq b$ et on veut montrer $(aa^*)b \leq b$. Par S3 on a $a^*b \leq b$ et donc $aa^*b \leq (1 + aa^*)b \leq a^*b \leq b$.

Et dans l'autre sens :

- $1 + aa^* \leq a^*$: On veut montrer que $1 + a(1 + a^+) \leq 1 + a^+$. On a $1 \leq 1$ mais aussi $a \leq a^+$ par l'axiome P1 et $aa^+ \leq a^+$ par P2.
- $ab \leq b \Rightarrow a^*b \leq b$: On suppose $ab \leq b$ donc $a^+b \leq b$ par l'axiome P4 et donc en ajoutant b on obtient $b + a^+b \leq b + b$, en factorisant le terme de gauche et en utilisant l'idempotence à droite on obtient $(1 + a^+)b \leq b$.

□

Les +-algèbres apparaissent dans des travaux de Kozen [Koz98], elles ont de meilleures propriétés lorsque les objets de l'algèbre sont typés, comme c'est le cas pour les matrices rectangulaires (les types sont les dimensions des matrices). Le livre d'Eilenberg [Eil74] traite des expressions régulières mais surtout des expressions rationnelles sur un semi-anneau. Ces expressions rationnelles rendent compte des multiplicités (typiquement l'axiome Ax4 d'idempotence de l'addition est proscrit). Pour ces algèbres Eilenberg préfère une axiomatisation utilisant l'opération d'itération $^+$ plutôt que l'étoile de Kleene parce qu'elle permet un traitement plus fin de la présence de l'élément 1 unitaire.

3 Conclusion

Nous avons présenté dans une première partie les notions de base de la théorie des automates : mots, langages, langages réguliers, expressions régulières, automates finis, langages reconnaissables. Parmi les théorèmes importants du domaine nous avons rappelé le théorème de Kleene. Il établit l'égalité de la classe des langages reconnaissables (définis par les automates finis) et des langages réguliers (définis par des expressions régulières). En particulier cela permet de comprendre les expressions régulières comme une *syntaxe* pour décrire des automates finis qui seraient quant à eux plus un modèle pour le *calcul*.

Il existe un certain nombre d'équations intuitives qui rendent compte de l'égalité entre langages réguliers, il est donc naturel de se poser la question de savoir s'il existe un ensemble bien compris de tels axiomes qui permettent de prouver l'égalité de deux langages. Dans ce cadre nous avons étudié les axiomatisations d'algèbres relatives aux expressions régulières. Tout d'abord les algèbres de Kleene axiomatisées par Kozen sont présentées par une dizaine d'axiomes qui permettent de prouver toutes les égalités entre langages réguliers (décrits par des expressions régulières). Cette axiomatisation est décrite principalement par des axiomes purement équationnels avec deux autres axiomes quasi-équationnels (de la forme $E = E' \Rightarrow F = F'$). Il n'existe pas d'axiomatisation purement équationnelle pour les algèbres de Kleene qui ont la signature stricte $(A, +, \cdot, ^*, 0, 1)$. Cependant Pratt a proposé une extension avec ses algèbres d'actions, munies de deux opérateurs supplémentaires et de signature $(A, +, 0, \cdot, 1, \rightarrow, \leftarrow, ^*)$. L'avantage des algèbres d'actions est d'avoir une axiomatisation purement équationnelle. Les algèbres d'actions empruntent des idées tant aux algèbres de Kleene qu'aux algèbres de relations qui sont importantes dans de nombreux domaines de l'informatique et au cœur des machines d'Eilenberg.

Les études axiomatiques sont fondamentales pour bien comprendre les propriétés algébriques des objets mathématiques. Les langages réguliers dénotés par des expressions régulières forment une algèbre de Kleene aussi bien qu'une algèbre d'actions et on pourra désormais raisonner axiomatiquement lorsqu'il s'agira d'égalité. Cela vaut aussi pour n'importe quelle autre algèbre vérifiant les axiomes. Dans de récents travaux, Braibant et Pous [BP09] ont présenté une for-

malisation complète dans l'assistant de preuves Coq du théorème de complétude des algèbres de Kleene par Kozen. Ce développement fournit une procédure de décision effective pour prouver de manière automatique que deux éléments d'une algèbre de Kleene sont équivalents selon les axiomes. Il est aussi intéressant d'étudier ces axiomatisations dans un contexte de réécriture comme l'ont fait Antimirov et Mosses [AM95] en 1995, et plus récemment encore Almeida, Moreira et Reis [AMR08, AMR09].

Nous avons désormais tout l'arsenal nécessaire pour étudier le modèle de calcul des machines d'Eilenberg ; ce sera l'objet du chapitre suivant. Une machine d'Eilenberg est définie par un automate fini étiqueté par des relations binaires sur un ensemble D. En utilisant le théorème de Kleene on peut donc utiliser les expressions régulières comme une syntaxe pour décrire l'automate fini décrivant le contrôle d'une machine d'Eilenberg. Pour cette raison nous allons étudier dans le chapitre III les algorithmes de synthèse d'automates à partir d'expressions régulières.

Chapitre II

Simulation des Machines d'Eilenberg

1 Machines relationnelles

Une relation ρ d'un ensemble D vers un ensemble D' notée $\rho : D \to D'$ est un sous-ensemble du produit cartésien $D \times D'$. On appelle D le *domaine* et D' le *codomaine*. La notation du type par une fonction est justifiée par l'isomorphisme entre $\wp(D \times D')$ et $D \to \wp(D')$. Lorsqu'un élément $d \in D$ et un élément $d' \in D'$ sont en relation par ρ on le note $d\rho d'$. L'*inverse* d'une relation $\rho : D \to D'$ est notée $\tilde{\rho} : D' \to D$. On notera l'opération de *composition* de relations par $\circ$: considérons deux relations $\rho_1 : D \to D'$ et $\rho_2 : D' \to \mathcal{D}''$ alors la relation composition de ρ_1 et ρ_2 est

$$\rho_1 \circ \rho_2 = \{(d, d'') \mid \exists d', d\rho_1 d' \wedge d'\rho_2 d''\}.$$

Une opération usuelle sur les relations est l'*union*, notée $\cup$, et définie par l'union des ensembles de paires de chaque relation :

$$\rho_1 \cup \rho_2 = \{(d, d') \mid d\rho_1 d' \vee d\rho_2 d'\}.$$

Nous utiliserons la notation $\rho(d)$ pour l'ensemble suivant $\{\ d' \mid d' \in D',\ (d, d') \in \rho\}$.

Lorsque le domaine et le codomaine d'une relation sont égaux on dit que la relation est *homogène* et on définit la relation *identité* par $id_D = \{(d, d) \mid d \in D\}$. La relation identité est un élément neutre vis-à-vis de l'opération de composition, les relations homogènes forment donc une structure de monoïde. On notera R_D ce monoïde qu'on appelle *monoïde relationnel.*

Nous allons définir une notion de machine abstraite inspirée des travaux d'Eilenberg (X-machines [Eil74]). Nos machines sont non-déterministes par nature et exécutent des *actions.* Elle contiennent deux composantes, respectivement une *partie de contrôle* et une *partie de données*, qui interagissent par le biais de relations qui décrivent le comportement de telles machines. La *partie de contrôle* est similaire à un système de transitions d'états. Les transitions sont étiquetées par des générateurs d'actions. Par exemple on utilisera des expressions régulières pour synthétiser un système de transitions fini tel qu'un automate fini non-déterministe ; cependant on pourrait s'intéresser à des systèmes de transitions plus généraux, en particulier ceux comprenant un nombre d'états infini. Un programme se compilerait en de multiples parties de contrôles suivant différentes traductions adéquates. Nos machines sont aussi définies avec une *partie de données*, où les actions sont interprétées par une sémantique relationnelle. C'est-à-dire qu'on interprète les générateurs d'actions comme des relations binaires sur un certain domaine, le domaine des données. Ces relations seront elles-mêmes représentées comme des fonctions associant à une donnée d'entrée un flux de données de sortie. Cet appareillage applicatif remplace par des notions mathématiques précises les composantes impératives qu'on peut trouver dans la

théorie des automates (bandes, tête de lecture, compteurs, piles, *etc*).

Nous allons maintenant formaliser ces notions d'une manière telle qu'on fait apparaître une symétrie entre la partie de contrôle et la partie de données. Tout d'abord, on se donne un ensemble fini Σ de symboles qui sont les noms des actions primitives de la machine ; on les appelle les *générateurs*.

Pour la partie de contrôle, on se donne un ensemble Q, modélisant les *états* du contrôle de la machine. On modélise les successions possibles d'états à l'aide d'une *sémantique de contrôle* interprétant chaque générateur comme une relation binaire sur Q. Cette sémantique de contrôle est souvent présentée de manière currifiée sous forme de *fonction de transitions* δ de Q vers un ensemble de paires de (a, q) avec a un générateur et q un état (comme pour les automates). On peut énumérer cet ensemble à l'aide d'un flux ou d'une liste lorsque l'ensemble est fini. Pour finir, on choisit dans Q une sous-partie d'états initiaux I et une sous-partie d'états acceptants F.

Pour la partie de données, on se donne un ensemble D de données. On modélise les calculs possibles sur ces données à l'aide d'une *sémantique de données* interprétant chaque générateur d'actions $a \in \Sigma$ comme une relation binaire sur D. Une telle relation binaire sera présentée de manière currifiée sous forme d'une fonction de type $D \to \wp(D)$.

La version complètement symétrique de machines relationnelles serait définie par les paramètres suivants :

$$\begin{array}{rcl} & & \Sigma, D, Q \\ \delta & : & \Sigma \to (Q \to \wp(Q)) \\ \rho & : & \Sigma \to (D \to \wp(D)) \end{array}$$

Cependant, nous avons casser la symétrie de ces définitions afin que le contrôle guide le calcul ; en effet, c'est la partie contrôle qui fournit des générateurs, et la partie données les interprète par des relations calculant sur les données. Cela évite d'explorer tous les générateurs possibles pour avancer dans le calcul ; on adoptera donc plutôt la formalisation suivante :

$$\begin{array}{rcl} & & \Sigma, D, Q \\ \delta & : & Q \to (\wp(\Sigma \times Q)) \\ \rho & : & \Sigma \to (D \to \wp(D)) \end{array}$$

Pour avoir une caractérisation constructive de ces machines nous imposons que les ensembles utilisés soient récursivement énumérables et que les fonctions soient partielles récursives. Dans la partie suivante nous expliquerons nos choix de modélisation à base de *flux*. Lorsque l'ensemble Q des états est un ensemble fini, la relation de transition correspond en fait au graphe d'un automate fini non-déterministe, et donc ce modèle de machines relationnelles est le modèle des X-machines d'Eilenberg. Nous nous concentrerons par la suite sur ce modèle en rappelant les définitions et propriétés importantes qui lui sont reconnues. Nous discuterons ensuite de la manière dont nous proposons de simuler les machines d'Eilenberg en utilisant un moteur réactif. Nous prouverons en particulier, pour le cas des machines d'Eilenberg finies, qu'on peut formaliser complètement ce moteur dans un assistant de preuves formelles. Les machines d'Eilenberg finies sont un cas particulier des machines d'Eilenberg pour lesquelles les calculs sont finis en longueur mais aussi en largeur (non-déterminisme fini). Ensuite nous montrerons comment paramétrer ce moteur réactif avec des stratégies pour simuler des machines d'Eilenberg effectives (non nécessairement finies).

2 Programmation fonctionnelle en ML

Les algorithmes présentés seront formulés dans une variante du langage de programmation fonctionnelle ML ; il s'agit du langage de programmation OCaml [LDGV09]. Les lecteurs familiers des langages de programmation à la ML peuvent passer cette partie qui donne les rudiments permettant de comprendre les algorithmes présentés. La présentation que nous faisons du langage est issue de la section 1 de l'article [Hue05].

Le langage possède des types, des valeurs et des exceptions. Par exemple *1* est une valeur prédéfinie de type *int* alors que *"FP"* est une chaîne de caractères de types *string*. Les paires de valeurs sont du type du produit cartésien correspondant. La valeur (*1,"FP"*) est du type (*int* ∗ *string*). Les types définis de manière récursive créent de nouveaux types dont les valeurs sont construites récursivement à l'aide de constructeurs de types. Les constructeurs de types commencent par une lettre majuscule. Par exemple le type des booléens est défini comme la somme de deux constructeurs : **type** *bool* = [*True* | *False*]. Les types définis à l'aide de paramètres permettent d'obtenir des types de données *polymorphes*. Si x est une valeur de type t et l une liste de type *list* t alors on construit la liste ajoutant x à l par $[x :: l]$. La liste vide est notée [ ] et elle est du type *list 'a*; le paramètre *'a* indique que les éléments de la liste peuvent être d'un certain type non spécifié. Il est rarement nécessaire de devoir écrire le type des valeurs parce que les types principaux peuvent être inférés automatiquement.

Le langage est dit fonctionnel parce que les fonctions sont des objets de *première classe*. Par exemple la fonction calculant le double d'un entier s'écrit **fun** $x \rightarrow x{+}x$, et a le type *int* → *int*. On peut associer un nom à cette fonction :

value *double* = **fun** $x \rightarrow x + x$;

ou de manière équivalente écrire

value *double* $x = x + x$;

L'application de cette fonction à l'entier n s'écrit '*double n*' lorsqu'il n'y a pas d'ambiguïté. L'opération d'application est associative à gauche donc '$f\ x\ y$' est interprété par $((f\ x)\ y)$. Les fonctions définies récursivement sont déclarées à l'aide du mot-clé **rec**. On définit la fonction factorielle comme suit :

value rec *fact* n = **if** $n = 0$ **then** *1* **else** $n * (fact\ (n-1))$;

Les fonctions peuvent aussi être définies par *filtrage* (*pattern-matching*). Par exemple la première projection d'une paire s'écrit :

value *fst* = **fun** [$(x,y) \rightarrow x$] ;

Le filtrage peut aussi être exprimé à l'aide d'expressions **match**. Par exemple on peut tester si une liste est vide comme suit :

```
value empty l = match l with
  [ [] → True
  | _  → False
  ];
```

Le caractère underscore '_' signifie n'importe quelle valeur. L'ordre des cas dans les **match** expressions est pris en compte : les cas sont testés dans l'ordre où ils sont écrits.

L'évaluation de ML est stricte, ce qui veut dire qu'une expression e est évaluée avant f dans l'expression $(f\ e)$. Pour faire du partage de calcul on utilise les expressions **let**. Par exemple dans l'expression '**let** x = *fact 10* **in** $x{+}x$' la sous-expression (*fact 10*) est calculée en premier et ce calcul ne sera fait qu'une seule fois bien qu'utilisé deux fois dans la sous-expression '$x{+}x$'.

Le langage possède aussi des constructions impératives telles que les références, les tableaux, les séquences, les entrées-sorties et les exceptions. Les séquences d'instructions sont évaluées de gauche à droite dans des expressions **do** telles que **do**(*e1* ; ... *en*).

Parmi les types de données pervasifs que nous allons utiliser, nous utiliserons le type *unit*, qui ne contient qu'une seule valeur conventionnellement notée (). Nous utiliserons aussi le type de données polymorphe *option 'a* qui permet de représenter la présence ou l'absence d'une valeur *a*, respectivement par la valeur *None* ou par la valeur *Some a*.

Il est possible de grouper des définitions de types et de valeurs dans une unité qu'on appelle *module*. Cela permet de gérer l'espace de nommage des objets plus facilement mais aussi de structurer plus clairement les programmes. Par exemple, considérons un module implémentant les piles ; ce module comprend des définitions de types et de valeurs qu'on inclut dans une *structure* à l'aide de la construction **struct** ... **end** :

```
module Stack = struct
  type data = int;
  type stack = list int;
  value empty = [];
  value pop l = match l with
                [ [] → None
                | [d :: rest] → Some (d, rest)
                ];
  value push d l = [ d :: l ];
end;
```

Le nom du module est *Stack*. On peut accéder aux valeurs et aux types d'une structure de module en précédant leur nom par le nom du module suivi d'un point ; dans notre exemple, *Stack.push* fait référence à la fonction *push* du module *Stack*. Un module a aussi un type qu'on appelle *signature* et qui correspond à la liste des types définis et des types des valeurs de sa structure. Le module *Stack* a la signature suivante :

```
module type StackType =sig
  type data = int;
  type stack = list int;
  value empty : stack;
  value pop : stack → option (data * stack);
  value push : data → stack → stack;
end;
```

Lorsqu'on définit le contenu d'un module, on peut avoir besoin d'objets auxiliaires qu'on ne souhaite pas rendre visibles à l'utilisateur du module. Il existe un mécanisme permettant d'indiquer uniquement les définitions qu'on souhaite rendre visible et masquer les autres. Pour cela, on utilise la signature et on indique uniquement les types des définitions qu'on souhaite rendre visibles. Par exemple, le module précédent pourrait avoir une signature cachant la fonction *push*, le type du module correspondant serait le suivant :

```
module type IncompleteStackType =sig
  type data = int;
  type stack = list int;
  value empty : stack;
  value pop : stack → option (data * stack);
end;
```

Maintenant on peut définir, à partir du module *Stack*, un module *IncompleteStack* dans lequel il manque la fonction *push* :

```
module IncompleteStack : IncompleteStackType = Stack;
```

De cette manière, on a défini le module *IncompleteStack* à partir du module *Stack* et à partir de la signature *IncompeletStackType* qui agit comme un filtre. On aurait pu choisir aussi de cacher le fait que *data* était égal à *int*, ou que le type *stack* était égal à *list data*, en notant simplement dans la signature '**type** *data*;' et respectivement '**type** *stack*;'.

On peut définir des modules paramétrés par d'autres modules ; on les appelle *foncteurs*. Un foncteur est paramétré par une signature de module et retourne une structure. Par exemple, à partir d'un module de type *IncompleteStackType* on peut définir à nouveau un module de type *StackType* de la manière suivante :

```
module StackGen (M : IncompleteStackType) : StackType =struct
  type data = M.data;
  type stack = M.stack;
  open M;
  value empty = empty;  (* from M *)
  value pop = pop;      (* from M *)
  value push d l = [ d :: l ];
end;
```

Le résultat de l'appel à **open** *M* est de mettre à disposition toutes les définitions contenues dans le module *M* ; en pratique, cela permet aussi de ne pas utiliser '*M.*' pour nommer une composante. Le foncteur *Stack2* attend en argument un module *M* de type *IncompleteStackType* et retourne un module de type *StackType*. On utilise ce foncteur en l'appliquant au module *IncompleteStack* pour obtenir un nouveau module de pile :

```
module Stack2 = StackGen(IncompleteStack);
```

La syntaxe avec laquelle nous présentons le langage est une variante de la syntaxe officielle d'OCaml, qu'on appelle la *syntaxe révisée*. La syntaxe révisée a été définie par Daniel de Rauglaudre et se trouve implémentée sous la forme d'un pre-processeur dans le logiciel Camlp4. Le lecteur souhaitant connaître précisément les différences entre la syntaxe d'OCaml et la syntaxe révisée pourra se référer au manuel de référence de Camlp4 [dR03]. Néanmoins, Camlp4 fait partie de la distribution d'OCaml et les programmes présentés dans la syntaxe révisée sont donc directement exploitables par le compilateur OCaml [LDGV09] en indiquant l'option adéquate.

3 Relations calculables et flux

Les relations sont au cœur du modèle de machine qui nous intéresse ici, et nous allons montrer comment les modéliser en utilisant la notion de flux.

Dans un premier temps nous rappelons des notions de calculabilité et d'énumérabilité des ensembles pour nous guider dans la modélisation des flux. Dans la théorie de la récursivité [Rog67], on définit un *ensemble récursivement énumérable* de la manière suivante.

Définition 1 ([Rog67] page 58)**.** Un ensemble A est *récursivement énumérable* ssi il est vide ou bien s'il est l'image d'une fonction récursive partielle de $\mathbb{N} \rightarrow \mathbb{N}$.

Grâce au théorème qui suit, on peut modéliser un ensemble récursivement énumérable d'une autre manière :

Théorème II.1 ([Rog67] page 60)**.** Un ensemble A est récursivement énumérable ssi il est le domaine d'une fonction récursive partielle.

Les détails de la preuve du théorème sont intéressants parce qu'ils illustrent le fait que, même au niveau théorique de la récursivité, il est parfois nécessaire d'observer les fonctions récursives partielles en tenant compte de leur exécution dans le *temps* ; c'est-à-dire que pour toute fonction récursive partielle f et pour tout entier i, on s'autorise à parler de l'état du calcul de $f(i)$ sur ses n premiers *pas d'exécution*.

Soit f la fonction récursive partielle dont le domaine est A. Pour construire une fonction récursive partielle g dont l'image est A, on procède par étapes successives. À l'étape n, on cherche à définir le comportement de $g(n)$ de la manière suivante : observons le comportement de f sur l'ensemble des valeurs $[0, n]$, s'il existe un plus petit entier $i \in [0, n]$ tel que l'appel de $f(i)$ termine en moins de n pas d'exécution, et si i n'a pas été défini dans l'image de g pour les $n - 1$ étapes précédentes alors on définit $g(n) = i$; sinon, dans le cas où il n'existe pas de tel i, alors on définit $g(n)$ comme n'ayant pas de résultat. Rogers fait référence à cette construction en parlant d'*entrelacement* (*dovetailing* en anglais) et il insiste sur le fait que cet entrelacement est d'usage courant dans son livre, mais qu'il peut être dissimulé sous l'usage du théorème. Nous voyons donc que même à un niveau théorique, il est nécessaire de casser la fonctionalité d'un programme, en regardant précisément ses pas d'exécution. D'un point de vue plus pratique, on retrouve ce mécanisme à travers le mécanisme d'ordonnancement des tâches d'un système d'exploitation. En effet un système d'exploitation a besoin d'un mécanisme permettant d'interrompre une tâche à tout moment de son exécution, puis de sauver sont état et enfin de la redémarrer. Ce mécanisme est indispensable lorsqu'on souhaite exécuter plusieurs tâches en parallèle mais de manière séquentielle. Ce sera aussi le cas pour nos machines et afin de conserver un caractère fonctionnel et aussi de ne pas modéliser un ordonnanceur, nous utiliserons des flux légèrement différents de ceux habituellement rencontrés ; il s'agit de flux qui peuvent rendre la main alors qu'ils n'ont pas d'éléments à fournir.

On souhaite modéliser les relations calculables de manière effective. Une relation entre les domaines D_1 et D_2 est une fonction du type $D_1 \to \wp(D_2)$. Plusieurs choix s'offrent à nous pour modéliser les éléments de $\wp(D_2)$. Nous choisissons de procéder par énumération de l'ensemble. L'énumération est la solution la plus simple, on aurait pu utiliser des structures de données plus compliquées qui permettraient une représentation plus ou moins canonique d'un ensemble au vu de certaines propriétés. On pense par exemple à l'absence de doublons dans la représentation, ou encore un accès rapide pour tester l'appartenance.

Il est très courant d'utiliser des streams pour représenter les suites infinies de valeurs, qu'on appelle *flux* (*stream* en anglais) :

type *infstream 'a* = [*Elm* **of** *a* **and** (*unit* → *infstream 'a*)] ;

Ce type de données ne contient qu'un seul constructeur marquant la production d'un élément du flux, et le calcul de l'élément suivant est bloqué par une fonction qui attend un élément de type *unit*.

Nous proposons de modifier le type de données *infstream* afin de rendre compte d'énumérations pour nos besoins plus spécifiques. Premièrement, nous remarquons que ce type de flux est mal adapté à l'énumération d'ensembles finis puisqu'un flux est infini par définition ; nous proposons donc d'ajouter un constructeur pour marquer la fin de l'énumération (ce sera le constructeur *Done*). Deuxièmement, la production d'un nouvel élément pourrait être interminable et dans ce cas bloquer l'énumération. Ce blocage pose problème lorsqu'on calcule séquentiellement l'union de deux flux ; en effet, si le premier flux est bloquant alors cela fait obstacle à l'énumération du deuxième flux. Nous proposons donc d'ajouter un constructeur permettant de marquer une pause dans un processus d'énumération (ce sera le constructeur *Skip*).

type *flux 'data* =

```
  [ Done
  | Skip of delay 'data
  | Elm of 'data and delay 'data
  ]
and delay 'data = unit → flux 'data; (* frozen flux *)
```

Le constructeur *Elm* fournit un élément d'un flux. La suite de l'énumération est bloquée par une fonction de type *delay* qui attend un élément de type *unit* avant de renvoyer un nouvel élément du flux. Ce type de données est similaire à celui utilisé par Coutts, Leshchinskiy et Stewart [CLS07] avec des objectifs différents, liés à l'optimisation de compilation de programmes fonctionnels utilisant des opérations sur les listes infinies.

On peut définir le flux des entiers de la manière suivante :

```
value int_flux =
  let rec aux n =
    Elm n (fun () → aux (n+1)) in
  aux 0;
```

On peut définir un itérateur sur les flux qui prend en argument la fonction à itérer sur les éléments du flux :

```
(* iter_flux : ( 'a → unit) → flux 'a → unit *)
value iter_flux f e =
  let rec aux e =
    match e with
    [ Done → ()
    | Skip del → aux ( del ())
    | Elm d del → do{ f d ; aux ( del ()) }
    ] in
  aux e;
```

On se donne aussi une fonction qui tronque un flux à sa nième valeur :

```
(* cut : int → flux 'a → flux 'a *)
value rec cut n e =
  if n ≤ 0 then Done
  else
    match e with
    [ Done → Done
    | Skip del → Skip (fun () → cut n ( del ()))
    | Elm d del → Elm d (fun () → cut (n−1) (del ()))
    ];
```

On peut maintenant imprimer les dix premiers entiers du flux des entiers en invoquant

```
iter_flux print_int (cut 10 int_flux);
```

qui imprime sur la sortie standard *0123456789*. La fonction *print_int* est une fonction de la librairie standard d'OCaml qui permet d'imprimer des entiers.

Les flux sont définis à l'aide de fonctions générales pour lesquelles on n'a aucune garantie de terminaison. Or si les fonctions peuvent ne pas terminer alors les flux peuvent ne pas être productifs. Ces propriétés de productivité des flux et de terminaison des fonctions sont donc étroitement liées.

Définition 2. Pour tout type de données *t*, *t1* et *t2*

1. Une fonction *f : t1* → *t2* est *totale* si pour toute valeur *v : t1*, l'évaluation de *f v* termine (en produisant une valeur de type *t2*).
2. Un flux *e : flux t* est *productif* si
   - *e* = *Done*, ou bien
   - *e* = *Skip f*, avec *f* totale, et *f* () est un flux *productif*, ou bien
   - *e* = *Elm d f*, avec *f* totale, et *f* () est un flux *productif*.

On définit les fonctions *hd* et *tl* qui ont pour résultat respectivement la tête et la queue d'un flux productif. La valeur de retour sera donc optionnelle.

```
value hd e =
  match e with
  [ Done → None
  | Skip f → None
  | Elm d f → Some d
  ];

value tl e =
  match e with
  [ Done → None
  | Skip f → Some (f ())
  | Elm d f → Some (f ())
  ];
```

Pour tout flux productif *e*, les fonctions *hd* et *tl* terminent. On peut composer successivement la fonction *tl* afin d'obtenir le reste d'un flux à une profondeur donnée. Soit n un entier, la fonction tl^n itère n fois la fonction *tl* lorsque celle-ci ne renvoie pas une valeur *None* :

```
value rec nth_tail e n =
  if n ≤ 0 then Some e
  else match e with
       [ Done → None
       | Skip f → nth_tail (f ()) (n−1)
       | Elm d f → nth_tail (f ()) (n−1)
       ];
```

On définit la notion d'appartenance d'un élément d à un flux e par le prédicat $InFlux(d, e)$ de la manière suivante :

Définition 3. Pour toute donnée d et flux e, $InFlux(d, e)$ est vrai si et seulement s'il existe un entier n tel que *nth_tail e n* = *Some f* et *hd f* = *Some d*.

Jusqu'à présent les flux auxquels nous nous intéressons sont très généraux, et une classe importante de flux sont ceux qui sont finis. Un flux fini est un flux productif de taille finie.

Définition 4. Un flux e est *fini* si et seulement si

1. e est un flux productif,
2. il existe un entier n tel que *nth_tail e n* = *Some Done*.

On peut traduire un flux fini en un flux pour lequel les éléments de type *Skip* sont éliminés. Pour bien différencier ces deux types d'énumération on introduit un nouveau type de données *stream* avec seulement deux constructeurs *Void* et *Stream*[1] qui ont pour analogues *Done* et *Elm* du type *flux* :

[1] Cette notation est celle employée dans les articles [Raz08a, Raz08b].

```
type stream 'data =
  [ Void
  | Stream of 'data and delay 'data
  ]
and delay 'data = unit → stream 'data; (* frozen stream *)
```

On retrouve ici le type de données des listes paresseuses, c'est-à-dire que les éléments de la liste ne sont évalués qu'à la demande d'un consommateur. De même que pour les flux, l'évaluation est bloquée grâce aux valeurs de type *delay*[2]. Les notions de productivité pour les flux s'adaptent aussi aux streams. On peut maintenant traduire un flux fini en un stream :

```
(* erase_skip : flux 'a → stream 'a *)
value rec erase_skip e =
  match e with
  [ Done → Void
  | Skip f → erase_skip (f ())
  | Elm d f → Stream d (fun () → erase_skip (f ()) )
  ];
```

On pourrait facilement traduire un stream en un flux qui ne possède pas d'éléments *Skip*.

La notion de taille de flux pour les flux finis a un sens et on la définit de la manière suivante :

```
value rec length str =
   match str with
   [ Void → 0
   | Stream d del → 1 + length (del ())
   ];
```

On définit un prédicat d'appartenance d'un élément d dans un flux fini str par le prédicat $InStream(d, str)$. Ce prédicat est défini de manière analogue à $InFlux$.

Dans la suite on utilisera les flux et les streams pour représenter les relations effectives. Pour tous ensembles D_1 et D_2, une relation calculable ρ de D_1 vers D_2 est représentée par une fonction qui termine et ayant pour valeur de retour un flux productif :

$$\rho : D_1 \rightarrow \mathit{flux}\ D_2$$

et si la relation est finie alors nous représenterons ρ par une fonction qui termine ayant pour valeur de retour un stream productif :

$$\rho : D_1 \rightarrow \mathit{stream}\ D_2.$$

Pour des relations binaires, nous utiliserons le type polymorphe *relation* suivant :

```
type relation 'data = 'data → flux 'data;
```

Nous utiliserons le même nom de type pour les relations définies avec des streams :

```
type relation 'data = 'data → stream 'data;
```

Nous utiliserons les relations finies pour modéliser les machines d'Eilenberg finies. Les relations calculables seront utilisées pour le cas plus général des machines d'Eilenberg effectives.

[2]Pour être moins ambigu, nous aurions pu nommer différemment le type *delay* puisqu'il apparaît déjà dans la définition du type *flux*.

4 Machines d'Eilenberg

Dans cette partie nous rappelons la notion de *machine* introduite par Samuel Eilenberg [Eil74]. Cette notion de machine est étroitement liée à celle d'automate fini. Une machine d'Eilenberg est un automate fini non-deterministe dont l'alphabet utilisé pour étiqueter la machine est une classe de relations homogènes sur un ensemble D arbitraire. Les relations remplacent et généralisent les traditionnelles opérations de lecture et d'écriture des machines telles que les automates finis, automates à pile, transducteurs et machines de Turing. Eilenberg a défini des X-machines abstraites en se basant sur des notions ensemblistes, dans la présente section nous resterons avec ces notions ensemblistes mais dans les sections qui suivront (Sections 5 et 6) nous préciserons le modèle avec des machines d'Eilenberg *effectives*, qui sont douées de calculabilité.

4.1 Définition du noyau

On considère un ensemble Σ de générateurs, Q un ensemble fini d'états, et D un ensemble arbitraire qu'on appelle ensemble des *données*[3]. Une *D-machine* $\mathcal{M}$ est constituée d'un automate fini non-déterministe $(Q, \Sigma, \delta, I, F)$ et d'une sémantique de données $\rho : \Sigma \to (D \to \wp(D))$[4] qu'on appellera la *sémantique* de la machine puisqu'il n'y a plus d'ambiguïté possible avec la sémantique de contrôle qui est l'automate fini.

En utilisant la notion usuelle d'automate fini non-déterministe, on a que l'étiquette d'un chemin $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$ est le mot $w = a_1 a_2 \cdots a_n$ qu'on va interpréter par la composition successive des relations issues de la sémantique $\rho(a_1) \circ \cdots \circ \rho(a_n)$. Le comportement de $\mathcal{M}$ en tant que simple automate définit un langage qui lorsqu'on l'interprète par ρ définit le comportement de la machine noté $|\mathcal{M}|$, c'est donc l'ensemble des relations associées à toutes les étiquettes de chemins commençant par un état initial et terminant par un état acceptant :

$$|\mathcal{M}| = \{\rho(a_1) \circ \cdots \circ \rho(a_n) \mid \exists q_1 \in I, \exists q_{n+1} \in F, q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} q_{n+1}\} \cup \{id_D \mid \exists q, q \in I \wedge q \in F\}.$$

La distinction fondamentale entre un automate et une machine vient de l'utilisation de l'opération d'union sur les relations. La machine $\mathcal{M}$ définit une relation particulière, notée $||\mathcal{M}||$, définie comme la relation union de toutes les relations de $|\mathcal{M}|$:

$$||\mathcal{M}|| = \bigcup_{\rho \in |\mathcal{M}|} \rho \ .$$

Nous appelons cette relation $||\mathcal{M}||$ la *relation caractéristique* de la machine $\mathcal{M}$.

Jusqu'à présent nous avons seulement exploité les notions sur les automates afin d'expliquer comment une machine d'Eilenberg définit un calcul par sa relation caractéristique. Nous allons maintenant prendre en compte plus spécifiquement les données et montrer leur interaction avec les états de la machine pour décrire plus précisément les étapes de calcul.

Nous appelons *cellule* une paire $(d, q) \in D \times Q$ formée d'une donnée et d'un état. Une *transition* est un triplet $((d, q), a, (d', q'))$ noté $(d, q) \xrightarrow{a} (d', q')$ satisfaisant les deux conditions

- $(a, q') \in \delta(q)$,
- $d' \in \rho(a)(d)$.

Une *trace* est une séquence de transitions consécutives $t = c_0 \xrightarrow{a_1} c_1 \cdots \xrightarrow{a_n} c_n$. L'entier n est la taille de la trace. La cellule c_0 est dite *initiale* et la cellule c_n est dite *finale*. Pour toute donnée d et état q, la cellule (d, q) définit une trace de longueur nulle qui est initiale comme finale. On dit que la cellule (d, q) est *acceptante* lorsque q est acceptant. Une trace t est dite *acceptante* quand sa cellule finale est acceptante.

[3]Dans le livre d'Eilenberg [Eil74] l'ensemble D est noté X, d'où le nom de *X-Machines* dans la littérature.

[4]Notre présentation diffère de la présentation originale d'Eilenberg qui n'utilise pas l'ensemble Σ parce que les relations $\rho(a)$ sont directement des étiquettes de l'automate.

4.2 Modularité

Une machine d'Eilenberg $\mathcal{M}$ sur un domaine D définissant une relation caractéristique, on peut composer de telles machines de la manière suivante. Soit $\mathcal{A}$ un NFA sur Σ, et pour tout générateur a de Σ soit $\mathcal{N}_a$ une D-machine définie sur un ensemble de générateurs Σ_a. On transforme $\mathcal{A}$ en une D-machine sur Σ en considérant la sémantique qui associe à tout a la relation caractéristique $||\mathcal{N}_a||$.

De cette manière on peut construire des machines *"plus grandes"* à partir de machines *"plus petites"*. Cette propriété est essentielle parce que si on postule l'existence d'un simulateur (ce postulat deviendra réel) pour machines d'Eilenberg alors on pourra faire autant d'instances du simulateur qu'il y a de niveaux de machines qui compose le calcul. Cela permet d'éviter d'aplatir les différents niveaux de machines en un gros automate fourre-tout compliqué.

4.3 Interfaces

Jusqu'à présent nous avons donné la définition de ce que nous appelons le *noyau* d'une machine d'Eilenberg, celle qui fait référence au calcul mélangeant contrôle et données. La définition complète d'une machine d'Eilenberg nécessite une *interface* pour présenter les données en entrée de la machine et interpréter les données en sortie. On se donne deux ensembles D_- et D_+ appelés respectivement les ensembles d'*entrées* et de *sorties*, une relation d'entrée $\phi_- : D_- \to D$ et une relation de sortie $\phi_+ : D \to D_+$. Intuitivement, la relation ϕ_- fournit des données au noyau de la machine à partir d'entrées tandis que ϕ_+ interprète les résultats du noyau comme des sorties. Un noyau de machine muni d'une interface définit une relation $\rho : D_- \to D_+$ par $\rho = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$. Il est nécessaire de bien séparer le noyau de l'interface pour effectuer des calculs aussi complexes que nécessaire sans se limiter à un ensemble de données fixé.

4.4 Généralité du modèle

Nous allons montrer qu'on peut encoder de manière directe par des machines d'Eilenberg différents modèles utilisés dans la théorie des langages : les automates finis, les transducteurs, les automates à piles et les machines de Turing. Nous présentons certains des codages d'Eilenberg qui indiquent que le modèle présente le bon degré de généralité. Cependant le lecteur curieux trouvera des encodages pour d'autres formalismes tels que les automates bidirectionnels, les transducteurs temps-réel (*"accelerated transducer"* en anglais), et les transducteurs positifs (chapitre X section 5 de [Eil74]).

Le *problème du mot* est un problème important de la théorie des langages. Ce problème s'énonce de la manière suivante : étant donné un langage L et un mot w, on souhaite construire une machine $\mathcal{M}$ associée à L, qui lorsqu'on lui donne w en entrée décide si w appartient à L.

Les automates finis sont les machines qui résolvent ce problème pour les langages réguliers. Les automates à piles sont les machines qui résolvent ce problème pour les langages *hors-contexte*. Les machines de Turing sont les machines théoriques les plus puissantes permettant de résoudre le problème du mot pour les langages récursivement énumérables (arbitrairement calculables).

Modélisons les opérations basiques sur les mots d'un alphabet Σ. Ces opérations sont la concaténation et la segmentation de chaque côté du mot. Pour tout symbole $\sigma \in \Sigma$ on définit les quatre opérations suivantes :

$$L_\sigma = \{ \ (w, \sigma w) \mid w \in \Sigma^* \ \}$$

$$R_\sigma = \{ \ (w, w\sigma) \mid w \in \Sigma^* \ \}$$

$$L_\sigma^{-1} = \{ \ (\sigma w, w) \mid w \in \Sigma^* \ \}$$

$$R_\sigma^{-1} = \{\ (w\sigma, w) \mid w \in \Sigma^*\ \}$$

L_σ et R_σ concatènent à un mot un symbole σ. L_σ^{-1} et R_σ^{-1} retranchent d'un mot un symbole σ. La lettre L ou R indique que l'opération concerne la partie gauche (left) ou droite (right) sur le mot d'entrée. Ces quatre opérations sont définies en terme de relations mais ce sont en fait des fonctions partielles. Les deux opérations L_σ^{-1} et R_σ^{-1} sont les relations symétriques de L_σ et R_σ. On notera la relation identité sur Σ^* par id_{Σ^*}, c'est l'opération qui laisse le mot intact.

À l'aide de ces quatre opérations on peut utiliser les mots comme des bandes de lecture $(L_\sigma^{-1}, R_\sigma^{-1})$ ou d'écriture (L_σ, R_σ), mais aussi comme des piles $(L_\sigma, L_\sigma^{-1})$ ou bien encore comme des bandes non bornées avec des paires de mots $(L_\sigma, R_\sigma, L_\sigma^{-1}, R_\sigma^{-1})$.

Automates finis non-déterministes (NFA)

Nous considérons un automate fini non-déterministe $\mathcal{A} = (Q, \Sigma, \delta, I, F)$. Nous allons construire une machine d'Eilenberg $\mathcal{M}$ qui résout le problème du mot pour le langage régulier $|\mathcal{A}|$ reconnu par l'automate. La machine $\mathcal{M}$ a Σ comme ensemble de générateurs et on choisit $\mathcal{A}$ pour sa partie de contrôle. Pour la partie de données on choisit $D = \Sigma^*$ et comme sémantique la fonction ρ définie par $\rho(\sigma) = L_\sigma^{-1} \stackrel{\text{def}}{=} \{\ (\sigma w, w) \mid w \in \Sigma^*\ \}$, comme expliqué précédemment. De cette manière le noyau de la machine calcule la dérivée, $||\mathcal{M}||(w) = \{w' \mid \exists u \in |\mathcal{A}|, uw' = w\}$. On complète la définition de la machine avec une interface d'entrée qui est l'identité, c'est-à-dire $D_- = \Sigma^*$ et $\phi_- = id_{\Sigma^*}$ et une interface de sortie qui donne une valeur booléenne de $\mathbb{B} = \{\bot, \top\}$ pour indiquer si le mot de sortie du noyau est vide, c'est-à-dire que $D_+ = \mathbb{B}$ et

$$\phi_+(w) = \begin{cases} \top & , \quad \text{si } w = \epsilon \\ \bot & , \quad \text{sinon} \end{cases}$$

Dans ce cas la machine $\mathcal{M}$ munie de son interface définit bien une relation qui résout le problème du mot :

$$(\phi_- \circ ||\mathcal{M}|| \circ \phi_+)(w) = \begin{cases} \top & , \quad \text{si } w \in |\mathcal{A}| \\ \bot & , \quad \text{sinon} \end{cases}$$

On remarque qu'on peut étendre le traitement précédent au cas des automates non-déterministes étiquetés par des mots $\mathcal{A} = (Q, \Sigma^*, \delta, I, F)$, et dans ce cas il y a une coïncidence sur le fait que l'ensemble des générateurs d'actions de la machine et l'ensemble des données sont tous les deux égaux à Σ^*.

Transducteurs

On se donne maintenant deux alphabets Σ et Γ. On notera ϵ tant pour le mot vide de Σ^* que le mot vide de Γ^*. Un transducteur $\mathcal{A} : \Sigma \Rightarrow \Gamma$ est similaire à un automate non-déterministe dont les transitions sont étiquetées par des paires de mots de $D = \Sigma^* \times \Gamma^*$. Soit Ω l'ensemble fini de paires de mots apparaissant comme des étiquettes de $\mathcal{A}$. Le graphe de transitions de $\mathcal{A}$ peut donc être considéré comme un automate non-déterministe ordinaire sur l'alphabet Ω, et constitue la partie contrôle de machines qui résolvent différents problèmes de transduction.

Nous rappelons qu'un transducteur *"lit son entrée"* sur une bande d'entrée représentée par un mot de Σ^* et *"écrit en sortie"* sur une bande de sortie représentée par un mot de Γ^*. On considère une cellule $((x, y), q)$ avec $x \in \Sigma^*$, $y \in \Gamma^*$ et $q \in Q$. S'il existe une transition d'automate $(q, (u, v), q') \in \delta$ alors on produit la cellule $((x', y'), q')$ par une transition dans la machine

$$((x, y), q) \xrightarrow{(u,v)} ((x', y'), q')$$

si $x' = L_u^{-1}(x)$ et $y' = yv$. Lorsqu'au cours d'une succession de transitions commençant à un état initial, le mot d'entrée i produit le mot de sortie o en arrivant à un état acceptant avec une bande d'entrée vide, alors on dit que (i, o) appartient à la relation rationnelle définie par le transducteur $\mathcal{A}$, et on note $|\mathcal{A}|$ cette relation. On peut maintenant résoudre différents problèmes de décision sur $|\mathcal{A}|$ avec des machines utilisant $\mathcal{A}$ comme partie de contrôle, D comme partie de données et différentes sémantiques :

1. *Reconnaissance.* Soit (i, o) un couple de mots de $\Sigma^* \times \Gamma^*$, décider si (i, o) appartient à $|\mathcal{A}|$.
2. *Synthèse.* Soit i un mot de Σ^*, calculer l'ensemble $|\mathcal{A}|(i)$ de mots de Γ^*.
3. *Analyse.* Soit o un mot de Γ^*, calculer l'ensemble $|\mathcal{A}|^{-1}(o)$ de mots de Σ^*.

À chaque problème correspond un choix de sémantique.

1. **Reconnaissance.** La sémantique est définie par $\rho(u, v) = L_u^{-1} \times L_v^{-1}$. On complète la définition de la machine avec une interface d'entrée qui est l'identité, c'est-à-dire $D_- = \Sigma^* \times \Gamma^*$ et $\phi_- = id_{\Sigma^* \times \Gamma^*}$ et une interface de sortie qui calcule une valeur booléenne avec $D_+ = \mathbb{B}$ et $\phi_+(w, w') = \top$ si $w = w' = \epsilon$.

2. **Synthèse.** La sémantique est définie par $\rho(u, v) = L_u^{-1} \times R_v$, c'est-à-dire qu'à chaque transition l'entrée est lue de u et on concatène v à la fin de la sortie courante. On complète la définition de la machine par l'interface d'entrée $D_- = \Sigma^*$ et $\phi_-(w) = (w, \epsilon)$ et l'interface de sortie $D_+ = \Gamma^*$ et $\phi_+(w, w') = w'$ si $w = \epsilon$.

3. **Analyse.** On remarque que ce cas est symétrique de la synthèse en remplaçant L_u^{-1} par R_u et en remplaçant R_v par L_v^{-1}.

Automates à pile

Un automate à pile est un automate muni d'une mémoire représentée par une pile. Les opérations sur la bande d'entrée sont toujours les mêmes que celles des automates finis, à savoir qu'on lit en effaçant la bande d'entrée. Et on peut insérer des opérations de pile entre ces calculs d'automates. On définit donc un automate à pile comme une machine d'Eilenberg $\mathcal{M}$ donc la partie contrôle est définie par un automate avec trois types d'opération : lecture sur la bande, push sur la pile et pop sur la pile. La bande est un mot de Σ^* et la pile sera représentée par un mot de Γ^*. On utilisera donc un alphabet union de trois alphabets $\Sigma \cup \Gamma_{\text{push}} \cup \Gamma_{\text{pop}}$ qui contient deux copies de Γ. On notera a pour un élément de Σ, a_{push} pour un élément de Γ_{push} et a_{pop} pour un élément de Γ_{pop}. On choisit pour la partie données

$$D = \Sigma^* \times \Gamma^*$$

et la sémantique ρ

$$\begin{array}{rcl} \rho(\sigma) & = & L_\sigma^{-1} \times id_{\Gamma^*} \\ \rho(\gamma_{\text{push}}) & = & id_{\Sigma^*} \times L_\gamma \\ \rho(\gamma_{\text{pop}}) & = & id_{\Sigma^*} \times L_\gamma^{-1} \end{array}$$

Machines de Turing

On utilise les mots sur les booléens $\mathbb{B}$. Une paire de mots de $\mathbb{B}^* \times \mathbb{B}^*$ peut être vue comme un mot de $\mathbb{B}^*$ muni d'un pointeur. C'est ainsi qu'on modélise une bande sur laquelle on peut lire, écrire et se déplacer. Le pointeur représente la tête de lecture. La partie contrôle est définie par un automate dont les symboles sont des opérations push ou pop, à gauche (L) ou à droite (R) du pointeur

$$\mathbb{B} \times \{\text{push}, \text{pop}\} \times \{L, R\}.$$

On choisit pour la partie données $D = \mathbb{B}^* \times \mathbb{B}^*$ et la sémantique ρ :

$$\begin{array}{rcl}\rho(b, \text{push}, L) & = & R_b \times id_{\mathbb{B}^*} \\ \rho(b, \text{pop}, L) & = & R_b^{-1} \times id_{\mathbb{B}^*} \\ \rho(b, \text{push}, R) & = & id_{\mathbb{B}^*} \times L_b \\ \rho(b, \text{pop}, R) & = & id_{\mathbb{B}^*} \times L_b^{-1}\end{array}$$

5 Machines d'Eilenberg finies

5.1 Définitions

Une machine d'Eilenberg est non-déterministe de deux manières, premièrement par la partie de contrôle (définie par un automate fini non-déterministe) et deuxièmement par la partie de données (non-déterminisme dû aux relations qui constituent les actions de la machine). On remarque que le non-déterminisme de la partie de contrôle est de nature fini (le nombre de transitions partant d'un état est fini), alors que le non-déterminisme de la partie de données est potentiellement infini (à cause des flux). Dans le cas général, il faut donc tenir compte de cet infini et proposer une simulation de la machine à l'aide d'un parcours en largeur d'abord. Cependant cette simulation ne sera pas toujours la plus efficace, en particulier pour le cas où un parcours en profondeur d'abord est possible. En effet, il existe de nombreux cas pour lesquels on sait que, d'une part le non-déterminisme de la partie de données est fini, et d'autre part que les traces sont de profondeur finies ; dans ce cas on peut utiliser une recherche de solution à l'aide d'un parcours en profondeur d'abord. Pour cette raison, nous allons restreindre le modèle et définir la notion de *machine d'Eilenberg finie*. Une condition de finitude est déjà imposée à la partie de contrôle puisque le graphe de transition est celui d'un automate fini. Il reste donc à contraindre la partie de donnée et les traces qui peuvent être engendrées. Dans la suite, on se donne une machine $\mathcal{M}$ définie à partir d'un automate $(Q, \Sigma, \delta, I, F)$ et d'une sémantique ρ.

Tout d'abord, soient D_1 et D_2 deux ensembles, une relation $\rho : D_1 \to D_2$ est *localement finie* si et seulement si pour toute donnée d de D_1, l'ensemble $\rho(d)$ est fini. On dit que $\mathcal{M}$ est *localement finie* si et seulement si pour tout générateur $a \in \Sigma$ toute relation $\rho(a)$ est localement finie. On dit que la machine $\mathcal{M}$ est *globalement finie* si et seulement si sa relation caractéristique $||\mathcal{M}||$ est localement finie. On dit que la machine $\mathcal{M}$ est *nœthérienne* si et seulement s'il n'existe pas de trace infinie

$$c_0 \xrightarrow{a_1} c_1 \ \cdots \ \xrightarrow{a_n} c_n \ \cdots .$$

Définition 5. Une machine $\mathcal{M}$ est *finie* si et seulement si elle est localement finie et nœthérienne.

Ces deux restrictions qu'on impose à une machine d'Eilenberg correspondent à une condition de finitude en largeur (finitude des relations) et une condition de finitude en profondeur (pas de trace infinie).

Remarque 1. Une machine localement finie peut être ou ne pas être globalement finie. Et inversement une machine globalement finie peut être ou ne pas être localement finie.

Proposition II.1. Toute machine finie $\mathcal{M}$ est globalement finie.

Démonstration. Soit d une donnée d'entrée. On peut représenter l'ensemble des traces de $\mathcal{M}$ commençant par d avec une structure d'arbre de la manière suivante : les nœuds sont les cellules et on crée une arête entre deux nœuds s'il existe une transition entre les deux cellules correspondant aux nœuds. Le lemme de König nous assure que si l'arbre est à branchement fini et que la profondeur des branches est finie alors l'arbre est fini. Dans le cas des machines finies on peut utiliser le lemme de König ; en effet, la condition d'être localement finie correspond au branchement fini sur l'arbre et la condition nœthérienne correspond à la profondeur finie. On en déduit que l'arbre de trace est fini pour toute donnée d'entrée d, par conséquent la relation caractéristique $||\mathcal{M}||$ est localement finie et donc $\mathcal{M}$ est globalement finie. □

Corollaire II.1. Soient ϕ_- et ϕ_+ deux fonctions partielles. Soit $\mathcal{M}$ une machine finie munie de l'interface ϕ_- et ϕ_+ alors la relation $\phi_- \circ ||\mathcal{M}|| \circ \phi_+$ est localement finie.

La condition nœthérienne peut être complexe à vérifier parce qu'elle mélange le contrôle et les données de manière arbitrairement complexe. Il y a deux cas remarquables pour lesquels une machine est nœthérienne. Premièrement lorsque le graphe de l'automate d'une machine ne contient pas de cycle alors la machine est clairement nœthérienne, il n'y a pas de possibilité de créer de traces infinies. Deuxièment, du côté des données, on peut énoncer une condition suffisante pour qu'une machine soit nœthérienne :

Définition 6. Une machine $\mathcal{M}$ est de *sémantique nœthérienne* si et seulement si la relation suivante est nœthérienne :

$$\bigcup_{a \in \Sigma} \rho(a).$$

Proposition II.2. Toute machine $\mathcal{M}$ de sémantique nœthérienne est nœthérienne.

5.2 Automates finis et transducteurs

En réutilisant les encodages de la section 4.4, nous allons discuter des conditions pour lesquelles les automates finis et transducteurs sont des machines d'Eilenberg finies.

Automates finis non-déterministes (ϵ-NFA)

Nous considérons une machine d'Eilenberg $\mathcal{M}$ définie par un automate fini non-déterministe à transitions spontanées $(Q, \Sigma \cup \{\epsilon\}, \delta, I, F)$ muni de la sémantique ρ telle que $\rho(a) = L_a^{-1}$ et $\rho(\epsilon) = id_{\Sigma^*}$. Discutons les cas pour lesquels $\mathcal{M}$ est une machine d'Eilenberg finie. Tout d'abord $\mathcal{M}$ est localement finie parce que les relations qui composent la machine sont des fonctions partielles (segmentation gauche et identité).

La machine $\mathcal{M}$ est de sémantique nœthérienne si elle n'a aucune ϵ-transition, autrement dit, si c'est un NFA. Cette propriété est vraie parce que pour chaque transition $(u, q) \xrightarrow{w} (v, q')$ on a que $|w| > 1$ et alors $|v| < |u|$, ce qui montre que la taille de la trace est bornée par la taille du mot d'entrée et donc il ne peut y avoir de trace infinie.

La machine $\mathcal{M}$ est nœthérienne lorsqu'il n'y a pas d'ϵ-cycle dans le graphe. On entend par ϵ-cycle un cycle qui est étiqueté à chaque transition par ϵ (interprété par la relation identité). Puisque l'automate a un nombre fini d'états n, s'il n'y a pas d'ϵ-cycle alors la taille du mot décroît nécessairement d'une unité toute les n transitions. Et inversement s'il y a un ϵ-cycle alors il existe une trace infinie décrite par ce cycle.

Transducteurs

On rappelle qu'un transducteur des mots sur Σ vers les mots sur Γ est défini par une machine d'Eilenberg $\mathcal{M}$ dont l'automate est $(Q, \Omega, \delta, I, F)$ avec $\Omega \subset \Sigma^* \times \Gamma^*$ et que l'on munit de sémantiques différentes selon qu'on veut qu'il résolve le problème de reconnaissance, de synthèse ou d'analyse. Pour le cas synthèse, la sémantique ρ est définie par $\rho(w, w') = L_w^{-1} \times R_{w'}$. Discutons maintenant le cas où le transducteur de synthèse $\mathcal{M}$ est une machine d'Eilenberg finie. $\mathcal{M}$ est localement finie parce que les relations qui composent la machine sont des fonctions partielles. $\mathcal{M}$ est nœthérienne si et seulement s'il n'y a pas de cycle étiqueté par des symboles du type (ϵ, w'). Cette propriété est vraie pour des raisons similaires à celles concernant les automates finis dont nous avons parlé précédemment. En effet on a la preuve que la bande de lecture, c'est-à-dire la partie gauche de la paire de mots, décroît nécessairement au bout d'un nombre fini de transitions puisque le nombre d'états du transducteur est fini.

Pour que la machine $\mathcal{M}$ d'un transducteur en reconnaissance soit finie il faut et il suffit qu'il n'y ait pas de cycle de la forme (ϵ, ϵ).

5.3 Un moteur réactif de simulation

Nous allons présenter un algorithme pour simuler les machines d'Eilenberg finies. La simulation consiste à implémenter la relation caractéristique de la machine. L'algorithme est inspiré du *moteur réactif* de Gérard Huet [Hue05] qui a été conçu dans le but de résoudre des problèmes de transductions régulières. Ce moteur réactif est implanté dans la boîte à outils ZEN [Hue02] de linguistique computationnelle ; il a été mis en œuvre dans la gestion des lexiques, et l'analyse syntaxique superficielle (analyse morpho-phonétique, segmentation, étiquetage, *etc.*).

Dans la suite nous considérons une machine d'Eilenberg $\mathcal{M}$ avec une partie de contrôle $(Q, \Sigma, \delta, I, F)$ et munie d'une sémantique ρ sur un ensemble de données D. On modélise une telle machine en utilisant le système de modules d'OCaml. La machine $\mathcal{M}$ sera un module du type :

```
module type Kernel = sig
  type generator;
  type data;
  type state;
  value transition : state → list (generator * state);
  value initial : list state;
  value accept : state → bool;
  value semantics : generator → relation data;
end;
```

Nous nous concentrons sur la simulation du noyau des machines d'Eilenberg, c'est pour cette raison que nous appelons *Kernel* la signature d'une machine. La première composante *generator* correspond aux générateurs de Σ, ensuite *data* correspond au type des données du domaine D. Le type *state* modélise les états Q de $\mathcal{M}$, la fonction *transition* modélise la fonction de transition δ, la liste *initial* représente les états initiaux I et la fonction *accept* est la relation caractéristique de l'acceptance F. La sémantique ρ est modélisée par la fonction *semantics* qui associe à un générateur d'action une relation du type *relation data*, c'est-à-dire une fonction du type *data* → *stream data*.

Nous allons fournir un algorithme qui implémente la relation caractéristique de $\mathcal{M}$. Cet algorithme s'applique à toute machine d'Eilenberg finie du type *Kernel*; on utilisera donc un foncteur paramétré par une machine d'Eilenberg de ce type. Appellons *Engine* ce foncteur :

```
module Engine (M : Kernel) = struct
```

```
  open M;
   ... (* body *) ...
end;
```

Le corps du foncteur contient un ensemble de définitions de fonctions et de types. Tout d'abord nous avons besoin des types suivants :

```
type choice = list (generator * state);
type backtrack =
  [ Advance of (data * state)
  | Choose of (data * state * choice * (delay data) * state)
  ];
type resumption = list backtrack;
```

Le type de données *choice* représente les listes de couples générateur-état tels qu'ils sont engendrés par la fonction de transitions de l'automate δ. Nous avons pu voir que le modèle des machines d'Eilenberg est un modèle de calcul avec un fort degré de non-déterminisme. La source du non-déterminisme est double : premièrement l'automate fini est possiblement non-déterministe et deuxièmement les relations de calcul sur les données sont par nature non-déterministes. À l'aide d'un mécanisme de retour-arrière (*backtrack*) le simulateur va pouvoir gérer ce non-déterminisme pour énumérer toutes les solutions possibles. On entend par backtrack ce procédé similaire au *fil d'Ariane* qui permet d'explorer tout un espace de recherche, en gardant trace des points de choix qu'on a pu faire, afin de revenir en arrière pour explorer les autres pistes. Le type de données *backtrack* est la structure dont nous aurons besoin, elle représentera les points de choix éventuels lors de la simulation. On collectera des points de choix qu'on accumulera dans une liste de type *resumption*. Le cœur de l'algorithme de simulation est composé de trois fonctions mutuellement récursives : *react*, *choose* et *continue*. On l'appelle le *moteur réactif*. Il effectue la recherche de toutes les solutions à l'aide d'un parcours en profondeur d'abord, en accumulant les points de choix dans des valeurs de type *backtrack* :

```
(* react : data → state → resumption → stream data *)
value rec react d q res =
  let ch = transition q in
  if accept q
    then Stream d (fun () → choose d q ch res) (* Solution found *)
    else choose d q ch res

(* choose : data → state → choice → resumption → stream data *)
and choose d q ch res =
  match ch with
  [ [] → continue res
  | [ (a, q') :: rest ] →
    match (semantics a d) with
    [ Void → choose d q rest res
    | Stream d' del' → react d' q' [ Choose (d,q,rest,del',q') :: res ]
    ]
  ]

(* continue : resumption → stream data *)
and continue res =
  match res with
  [ [] → Void
```

```
  | [ Advance (d,q) :: rest ] → react d q rest
  | [ Choose (d,q,ch,del,q') :: rest ] →
    match (del ()) with
    [ Void → choose d q ch rest
    | Stream d' del' → react d' q' [ Choose (d,q,ch,del',q') :: rest ]
    ]
  ];
```

La fonction *react* teste l'acceptance de l'état courant et dans ce cas calcule un flux dont la tête est la donnée courante et dont la queue est définie par la suite de la simulation. La fonction *choose* effectue le choix non-déterministe de la transition de l'automate. Dans cette version simplifiée, ce choix est induit par la structure de liste. Lorsqu'une transition est choisie, les transitions inexplorées sont stockées dans un point de choix qui est ajouté à la résomption. La fonction *continue* est appelée lorsqu'une piste de recherche est abandonnée et qu'il faut explorer la résomption contenant tous les points de choix précédemment ajoutés. C'est donc *continue* qui gére le mécanisme de backtrack. La politique de backtrack est de toujours choisir le dernier point de choix ajouté à la résomption. Cette politique est naturelle par la structure de liste qui est utilisée.

Il faut remarquer, premièrement, que le moteur réactif n'effectue aucun effet de bord, et deuxièmenent que celui-ci est programmé de manière complètement *récursive-terminale* (en utilisant la résomption comme une continuation). Cette dernière propriété, dans un langage de programmation fonctionnelle, peut être exploitée par le compilateur pour créer des sauts directs, sans sauver le contexte lors d'appel de fonction. Ainsi on évite les débordements de pile liés aux appels des fonctions.

Finalement, le moteur réactif nous permet de décrire la fonction qui implémente la relation caractéristique de la machine :

```
(* simulation : relation data *)
value simulation d =
  let rec init_res l acc =
    match l with
    [ [] → acc
    | (q :: rest) → init_res rest (Advance(d,q) :: acc)
    ]
  in continue ( init_res initial [] );
```

Soit d une donnée qu'on veut fournir en entrée de $\mathcal{M}$. La fonction *simulation* initialise d'abord une résomption contenant tous les points de choix du genre *Advance*(*d,q*) pour les états initiaux de l'ensemble I représentés par la liste *initial* , puis elle fait appel à la fonction *continue* sur la résomption.

Nous résumerons le principe de simulation complet ainsi : la machine $\mathcal{M}$ implémentée par un module M de type *Kernel* a une relation caractéristique $||\mathcal{M}||$ simulée par la relation *Engine*(*M*)*.simulation*, c'est-à-dire la fonction simulation qui est obtenue par instanciation du foncteur *Engine* sur le module *M*.

5.4 Formalisation

Nous proposons ici de formaliser la correction du moteur réactif pour la simulation des machines d'Eilenberg finies. Une formalisation complète dans l'assistant de preuves Coq a été vérifiée [Raz08b]. Nous rappellerons ici les principales définitions et notions utiles à la formalisation. Pour effectuer cette formalisation nous nous sommes inspirés des travaux sur le moteur réactif originel [Hue05]. L'*ordre multi-ensemble* de Dershowitz et Manna [DM79] sera utile pour

nous aider à prouver la terminaison du moteur réactif et cette technique nous fournira un principe de récurrence adéquat.

Énoncé du théorème de correction

Tout d'abord nous devons préciser ce qu'on entend par correction de la simulation. Pour deux données d et d', on dit que d' est une solution de d par la relation caractéristique de $\mathcal{M}$ si et seulement s'il existe un état initial q et une trace t commençant par (d, q) et terminant par une cellule acceptante (d', q'). On notera ce prédicat $Solution(d, d')$.

Définition 7. Pour toutes données d et d',

$$Solution(d, d') \quad \stackrel{\text{def}}{=} \quad \exists q\ q' \in Q, (d, q) \stackrel{||\mathcal{M}||}{\longrightarrow} (d', q')$$

On dit que le moteur réactif est correct lorsque, pour toute donnée d'entrée d, les deux propriétés suivantes sont vraies :
– toute donnée d' apparaissant dans le flux de simulation est une solution (cohérence),
– toute solution d' apparaît dans le flux de la simulation (complétude).

Théorème II.2 (Correction)**.**

$$\forall d\ d', InStream(d', simulation\ d) \Leftrightarrow Solution(d, d').$$

Dans ce théorème, nous dirons que le sens direct de l'équivalence correspond à la *cohérence* de la simulation et que le sens opposé de l'équivalence correspond à la *complétude.*

Invariant de bonne formation

Le moteur réactif a un invariant de bonne formation sur les points de choix et résomptions engendrés. Pour un point de choix b l'invariant est défini par le prédicat $WellFormedBack(b)$ de la manière suivante :

1. $WellFormedBack(Advance(d, q))$ est vrai pour toute donnée d et état q.
2. $WellFormedBack(Choose(d, q, ch, del, q'))$ est vrai si et seulement s'il existe un générateur a qui satisfasse les 3 conditions suivantes :
   – $(a, q') \in\ transition\ \ q$,
   – $\forall d', InStream(d', del\ ()) \Rightarrow InStream(d',\ semantics\ a\ d)$,
   – $\forall e, e \in ch \Rightarrow e \in\ transition\ \ q$.

La première condition correspond au respect de la condition de graphe de la machine, le générateur a est étiquette de transition entre q et q' ; la deuxième condition correspond au respect du flux restant dans l'énumération de la relation rel ; la troisième condition indique que les transitions restantes ch sont bien incluses dans les transitions possibles pour l'état q.

On dit que la résomption res est bien formée si tous les points de choix de la résomption sont eux-même bien formés, on note $WellFormedRes(res)$ cette propriété :

$$WellFormedRes(res) \stackrel{\text{def}}{=} (\forall b, b \in res \Rightarrow WellFormedBack(b)).$$

Terminaison

Nous allons prouver la terminaison du moteur réactif. Nous travaillons ici dans le cadre des machines d'Eilenberg finies, donc $\mathcal{M}$ satisfait la condition nœthérienne ; c'est-à-dire que les cellules sont partiellement ordonnées par la relation de trace et que cet ordre partiel est

nœthérien. Nous prouverons la terminaison sur des triplets formés d'une cellule c et de deux entiers n_1 et n_2.

$$\langle c, n_1, n_2 \rangle$$

L'ordre lexicographique sur ces triplets définit une relation nœthérienne. On peut donc utiliser l'ordre multi-ensemble sur de tels triplets.

Nous allons maintenant définir une fonction χ qui associe un multi-ensemble de triplets à chaque appel de fonction *react*, *choose* et *continue*. En prouvant que le multi-ensemble décroît à chaque sous-appel dans la récursion on démontre la terminaison des trois fonctions.

Définition 8. Pour toute résomption *res* on définit la fonction $\chi(res)$ comme le multi-ensemble de triplets $\chi(back)$ associés à chaque point de choix *back* dans *res*, tel que

$$\chi(Choose(d, q, ch, del, q')) = \langle (d, q), |ch|, |del()| + 1 \rangle$$

$$\chi(Advance(d, q)) = \langle (d, q), \kappa(q), 1 \rangle$$

avec $\kappa(q) = |\ transition\ \ q| + 1$. Nous associons maintenant un multi-ensemble pour chaque appel de fonction *react*, *choose* et *continue* :

$$\chi(\ react\ \ d\ q\ res) = \{\langle (d, q), \kappa(q), 0 \rangle\} \oplus \chi(res)$$

$$\chi(choose\ d\ q\ ch\ res) = \{\langle (d, q), |ch|, 0 \rangle\} \oplus \chi(res)$$

$$\chi(continue\ res) = \chi(res)$$

avec $\oplus$ désignant l'union disjointe de deux multi-ensembles.

La fonction χ sert de mesure pour prouver la terminaison du moteur réactif ; les appels récursifs du moteur réactif font décroître la mesure associée à chaque sous-appel :

Théorème II.3 (Terminaison)**.** Le moteur réactif est un algorithme qui termine.

Démonstration. Pour chaque appel de *react*, *choose* et *continue* on associe une mesure calculée par χ. Il faut s'assurer que tout sous-appel fait décroître la mesure. Cela a été vérifié formellement dans l'assistant de preuve Coq [Raz08b]. □

Un corollaire de ce théorème est que le flux produit par le moteur réactif est un flux fini.

Corollaire II.2. Pour toute donnée d, l'exécution de la fonction *simulation* appliquée à d termine en calculant un flux fini.

Cohérence et complétude

Nous allons maintenant prouver la cohérence et la complétude du moteur réactif. Pour cela nous aurons besoin de prédicats pour modéliser le fait qu'une cellule, un point de choix ou encore une résomption, peuvent mener à une solution d'. Dans un premier temps, pour prouver la cohérence, nous définissons les prédicats $PartSol(c, d')$, $PartSolBack(back, d')$ et $PartSolRes(res, d')$.

Le prédicat $PartSol(c, d')$ modélise le fait qu'une cellule mène à une solution d' par une trace de la machine :

Définition 9. Pour toute cellule c et donnée d', $PartSol(c, d')$ est vrai si et seulement s'il existe une trace acceptante t commençant par c et finissant par la donnée d'.

On définit un prédicat analogue pour les points de choix de type *backtrack*, on notera ce prédicat $PartSolBack(back, d')$:

1. $PartSolBack(Advance(d,q),\ d')$ si et seulement si $PartSol((d,q),d')$
2. $PartSolBack(Choose(d,q,ch,del,q'),\ d')$ si et seulement si $PartSol((d,q),d')$

On étend ce prédicat pour une résomption *res* :

$$PartSolRes(res,d') \text{ ssi } \exists back \in res, PartSolBack(back,d').$$

Nous pouvons donner le lemme de cohérence du moteur réactif pour les fonctions *react*, *choose* et *continue* :

Lemme 6. Les trois propriétés suivantes sont vraies :

1. $\forall\ d\ q\ res\ d',\ WellFormedRes(res) \Rightarrow$
   $InStream(d', react\ d\ q\ res) \Rightarrow PartSol((d,q),\ d')\ \vee\ PartSolRes(res,d')$
2. $\forall\ d\ q\ ch\ res\ d',\ WellFormedRes(res) \Rightarrow ch \subseteq (\ transition\ \ q) \Rightarrow$
   $InStream(d', choose\ d\ q\ ch\ res) \Rightarrow PartSol((d,q),\ d')\ \vee\ PartSolRes(res,d')$
3. $\forall\ res\ d',\ WellFormedRes(res) \Rightarrow$
   $InStream(d', continue\ res) \Rightarrow PartSolRes(res,d')$

Démonstration. La preuve se fait par analyse de cas en utilisant un principe de récurrence nœthérienne sur la mesure associée par χ. □

On obtient donc le théorème de cohérence suivant :

Théorème II.4 (Cohérence)**.**

$$\forall d\ d', InStream(d', simulation\ d) \Rightarrow Solution(d,d').$$

Démonstration. Soit d une donnée, l'exécution de la fonction *simulation* appliquée à d conduit à l'exécution de la fonction *continue* appliquée à la résomption $res = (\ init_res\quad initial\quad [\,])$. On peut prouver que la résomption *res* est uniquement composée de points de choix de la forme $Advance(d,q)$ pour lequel l'état q est un état initial. De plus, la résomption *res* est bien-formée, $WellFormedRes(res)$ est vrai. Et en appliquant le troisième cas du lemme 6 on a que

$$InStream(d', continue\ res) \Rightarrow PartSolRes(res,d').$$

Mais encore, commme tous les états présents dans la résomption sont initiaux alors pour toute donnée d' on a la propriété suivante :

$$PartSolRes(res,d') \Leftrightarrow Solution(d,d').$$

On conclut donc par transitivité de l'équivalence. □

Maintenant nous allons prouver la complétude et pour cela nous avons besoin de trois prédicats $PartSolChoice$, $PartSolBack2$ et $PartSolRes2$ (plus précis que $PartSol$, $PartSolBack$ et $PartSolRes$). On définit le prédicat $PartSolChoice(d,ch,d')$ comme vrai si et seulement s'il existe un générateur a, un état q_1 et une donnée d_1 tels que $(a,q_1) \in ch$ et $InStream(d_1,\rho(a))$ et $PartSol((d_1,q_1),d')$ soient vrais.

On définit le prédicat $PartSolBack2(back,d')$ par cas sur la structure du point de choix :

1. $PartSolBack2(Advance(d,q),\ d')$ si et seulement si $PartSol((d,q),d')$,
2. $PartSolBack2(Choose(d,q,ch,del,q_1),\ d')$ si et seulement si $PartSolChoice(d,ch,d')$ ou il existe d_1 tel que $InStream(d_1,\ del\ \ ())$ et $PartSol((d_1,q_1),d')$.

On étend ce prédicat sur les résomptions : pour toute résomption res, $PartSolRes2(res, d')$ est vrai si et seulement s'il existe un point de choix $back$ dans res tel que $PartSolBack2(back, d')$ soit vrai. Nous pouvons maintenant donner le lemme de complétude du moteur réactif pour les fonctions *react*, *choose* et *continue* :

Lemme 7. Les trois propriétés suivantes sont vraies :

1. $\forall\ d\ q\ res\ d',\ WellFormedRes(res) \Rightarrow$
   $(PartSol((d, q),\ d')\ \vee\ PartSolRes2(res, d')) \Rightarrow InStream(d', \mathit{react}\ d\ q\ res)$
2. $\forall\ d\ q\ ch\ res\ d',\ WellFormedRes(res) \Rightarrow ch \subseteq (\ \mathit{transition}\ \ q) \Rightarrow$
   $(PartSolChoice(d,\ ch,\ d')\ \vee\ PartSolRes2(res, d')) \Rightarrow InStream(d', \mathit{choose}\ d\ q\ ch\ res)$
3. $\forall\ res\ d',\ WellFormedRes(res) \Rightarrow$
   $PartSolRes2(res, d') \Rightarrow InStream(d', \mathit{continue}\ res)$

Démonstration. La preuve se fait par analyse de cas en utilisant un principe de récurrence nœthérienne sur la mesure associée par χ. □

On obtient donc le théorème de complétude suivant :

Théorème II.5 (Complétude)**.**

$$\forall d\ d', Solution(d, d') \Rightarrow InStream(d', \mathit{simulation}\ d).$$

Démonstration. Soit d une donnée, l'exécution de la fonction *simulation* appliquée à d conduit à l'exécution de la fonction *continue* appliquée à la résomption $res = (\ \mathit{init_res}\ \ \ \mathit{initial}\ \ [\,])$. On suppose que $Solution(d, d')$ est vrai alors il existe nécessairement un point de choix de la forme $Advance(d, q)$ dans res tel que $PartSolBack2(Advance(d, q), d')$. Dans ce cas le prédicat $PartSolRes2(res, d')$ est vrai et en appliquant le troisième cas du lemme 7 on montre que

$$PartSolRes2(res, d') \Rightarrow InStream(d', \mathit{continue}\ res).$$

Et donc on a bien $InStream(d', \mathit{simulation}\ d)$. □

La preuve du théorème II.2 se déduit des théorèmes de cohérence et de complétude. En utilisant la terminaison, la cohérence et la complétude on a donc prouvé que le moteur réactif implémente une simulation en termes de relation finie du type *relation data* des machines d'Eilenberg finies. Cela correspond à une preuve constructive de la proposition II.1.

5.5 Formalisation du moteur réactif en Coq

Le lecteur pourra aussi trouver une preuve complètement vérifiée à l'aide de l'assistant de preuves Coq dans l'article [Raz08b].

La principale difficulté rencontrée lors du travail de formalisation du moteur réactif en Coq est qu'il est impératif de prouver sa terminaison en même temps qu'on le définit. Il n'est donc pas possible de faire comme précédemment, c'est-à-dire dans un premier temps définir le moteur puis dans un second temps prouver sa terminaison. Pour effectuer les deux tâches en même temps, nous avons eu besoin de définir les fonctions du moteur réactif en ajoutant des paramètres qui correspondent à des invariants vérifiés par le moteur. Le moteur réactif est donc toujours un ensemble de trois fonctions mutuellement récursives *react*, *choose* et *continue*, manipulant des paramètres de la machine et la résomption. Aux arguments habituels, il faut ajouter les invariants qui sont des propositions qui dépendent des autres arguments :

```
Fixpoint react (d : data) (q : state) (res : resumption)
  (h1: WellFormedRes res)
  (h : Acc Rext ((Chi (d, q) (S (length (transition q))) O) :: (chi_res res)))
  {struct h} : stream data :=
  ...

with choose (d : data) (q : state) (ch : choice) (res : resumption)
  (h1: WellFormedRes res)
  (h2 : incl ch (transition q))
  (h : Acc Rext ((Chi (d, q) (length ch) O) :: (chi_res res)))
  {struct h} : stream data :=
  ...

with continue (res : resumption)
  (h1: WellFormedRes res)
  (h : Acc Rext (chi_res res))
  {struct h} : stream data :=
  ...
.
```

Le paramètre nommé *h1* est l'argument indiquant que la résomption est bien formée, c'est-à-dire que *h1* est du type *WellFormedRes res*. Cet argument est présent pour les trois fonctions. L'argument nommé *h2* n'est utile que pour la fonction *choose* et il indique que les choix de transitions *ch* appartiennent effectivement à un ensemble de transition issu de l'état q ; cette propriété correspond à *incl ch (transition q)*. Le dernier argument supplémentaire est *h*, celui-ci est utile pour prouver la terminaison des trois fonctions *react*, *choose* et *continue*. Dans chacun des cas, le type de *h* est de la forme *Acc Rext x*, où *x* correspond à la complexité calculée par χ, où le paramètre *Rext* est une relation sur les éléments de complexité. La propriété *Acc Rext x* indique que tous les éléments plus petit que *x* par la relation *Rext* sont accessibles. La relation *Rext* correspond à la relation d'ordre multi-ensemble dont nous parlions précédemment dans la preuve mathématiques de la section 5.4. Cependant, nous n'avons pas eu besoin de cette généralité offerte par l'ordre multi-ensemble ; un ordre bien-fondé *ad hoc* sur les listes a suffit aux besoin de la preuve formelle.

Nous rappelons la définition du prédicat d'accessibilité pour une relation binaire R sur des éléments de type A :

```
Inductive Acc (R : A → A → Prop) (x : A) : Prop :=
  | Acc_intro : (∀ y:A, R y x → Acc R y) → Acc R x.
```

Une relation R est bien-fondée si tous les éléments de A sont accessibles par R :

```
Definition well_founded (R : A → A → Prop) := ∀ a:A, Acc R a.
```

Pour finir, pour toute relation bien-fondée on a un principe récurrence qui lui est associé :

```
Theorem well_founded_ind : ∀ (R : A → A → Prop),
  well_founded R →
    ∀ P : A → Prop,
      (∀ x:A, (∀ y:A, R y x → P y) → P x) → ∀ a:A, P a.
```

Le lecteur qui souhaiterait plus de détails sur la modélisation en Coq des relations bien-fondées et de leur application à la preuve de terminaison de programmes, peut se référer au livre de Bertot et Castéran [BC04].

Le déroulement de la preuve de correction du moteur réactif en Coq suit scrupuleusement le schéma que nous avons proposé dans la section 5.4 en effectuant des preuves par récurrence à l'aide du principe de récurrence de la relation bien-fondée *Rext*.

Nous souhaitons insister sur le fait que le programme OCaml de la section 5.4 est obtenu par extraction automatique depuis la formalisation Coq. En particulier tous les arguments propositionnels sont effacés automatiquement afin qu'il ne reste plus que le contenu calculatoire de l'algorithme.

5.6 Moteur réactif optimisé

Pour les besoins de la formalisation de la preuve de correction du moteur réactif on a introduit un argument supplémentaire à la fonction *choose* ainsi qu'au constructeur *Choose*. Il s'agit de l'état q qui sert uniquement de témoin pour indiquer la cellule d'origine d'une transition, et pour s'assurer que les choix *ch* sont bien inclus dans *transition q*.

Nous fournissons maintenant un moteur réactif débarassé de cet argument superflu :

```
module EngineOpt (Machine: Kernel) =struct
open Machine;

type choice = list (generator * state);

type backtrack =
  [ Advance of (data * state)
  | Choose of (data * choice * (delay data) * state)
  ];

type resumption = list backtrack;

(* react : data → state → resumption → stream data *)
value rec react d q res =
  let ch = transition q in
  if accept q
    then Stream d (fun () → choose d ch res) (* Solution found *)
    else choose d ch res

(* choose : data → choice → resumption → stream data *)
and choose d ch res =
  match ch with
  [ [] → continue res
  | [ (a, q') :: rest ] →
    match (semantics a d) with
    [ Void → choose d rest res
    | Stream d' del' → react d' q' [ Choose (d,rest,del',q') :: res ]
    ]
  ]

(* continue : resumption → stream data *)
and continue res =
  match res with
```

```
  [ [] → Void
  | [ Advance (d,q) :: rest ] → react d q rest
  | [ Choose (d,ch,del,q') :: rest ] →
     match (del ()) with
     [ Void → choose d ch rest
     | Stream d' del' → react d' q' [ Choose (d,ch,del',q') :: rest ]
     ]
  ]
;

(* simulation : relation data *)
value simulation d =
  let rec init_res l acc =
   match l with
    [ [] → acc
    | [ q :: rest ] → init_res rest [ Advance (d,q) :: acc ]
    ] in
  continue ( init_res initial [] );

end; (* module EngineOpt *)
```

Ce programme ne diffère de celui de la section 5.4 que par l'effacement de l'argument q dans le constructeur *Choose* du type *backtrack*, et dans la définition de la fonction *choose* du moteur réctif. On pourra facilement se convaincre que cet argument n'avait aucun effet sur le calcul en analysant méticuleusement le code du moteur réactif.

Nous rappelons que la preuve de correction du moteur réactif est inspirée de celle du moteur réactif de Zen [Hue05]. Cependant on remarque que ce moteur n'avait pas besoin d'un argument de décroissance mélangeant donnée et état. En effet, seul un argument de décroissance sur la taille de la bande suffisait parce qu'on avait la garantie qu'à chaque transition, la taille de la bande décroissait strictement. On a donc la preuve que le modèle des machines d'Eilenberg finies, muni de son moteur réactif, permet de simuler des machines plus complexes pour lesquelles l'argument de décroissance repose sur la notion de cellule (donnée-état), et non pas seulement sur la donnée.

5.7 Exemple : un automate à pile pour la grammaire du λ-calcul

Nous allons discuter dans cette section l'efficacité du moteur réactif décrit précédemment. Nous allons définir une machine d'Eilenberg bien particulière : un automate à pile qui reconnaît le langage décrit par une grammaire pour le λ-calcul. Ensuite nous discuterons de l'efficacité de la simulation (moteur réactif) de l'automate à pile en tant que machine d'Eilenberg finie.

On appelle λ-terme un terme du λ-calcul. Ces termes sont décrits par la grammaire suivante :

T	:=	x	(variable)
	|	λx$\cdot T$	(abstraction)
	|	T@T	(application)
	|	(T)	

Les *terminaux* de cette grammaire sont x, λ, ·, @, (,) et @. Le symbole T est *non-terminal*. Cette grammaire est ambiguë. Par exemple le λ-terme `"λx.x@λx.x"` peut être reconnu en tant que `"λx.(x@λx.x)"` ou bien en tant que `"(λx.x)@(λx.x)"`. Dans ce cas un analyseur syntaxique complet correspondant devrait renvoyer deux analyses possibles, ou deux solutions, pour un tel mot d'entrée. On rappelle que cette grammaire est de type *hors-contexte* et que les langages

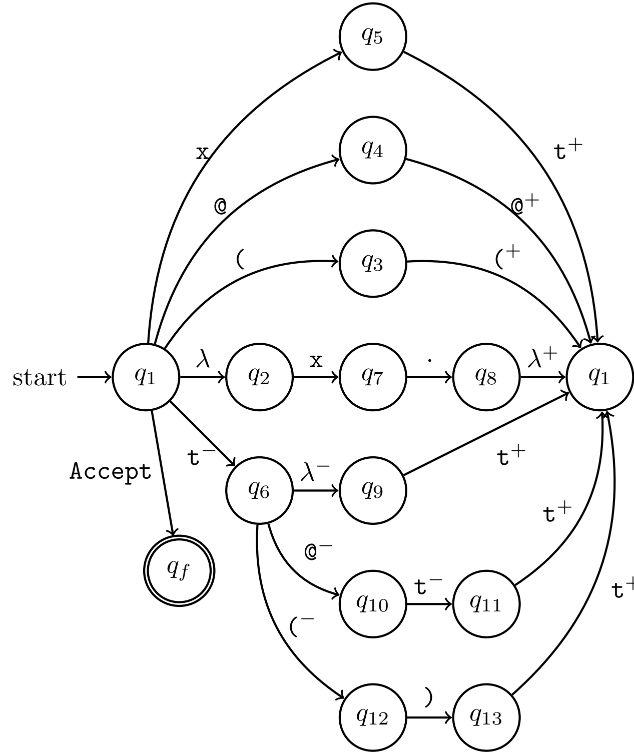


FIG. 1 – Automate à pile ambigu reconnaissant les λ-termes

reconnus par les grammaires hors-contextes sont exactement ceux reconnus par les automates à pile (PA, pour *pushdown automaton*). On rappelle aussi qu'un automate à pile manipule une *bande* de lecture et une *pile*. Le mot qu'on cherche à reconnaître est écrit sur la bande et la pile est utilisée comme une mémoire affaiblie (seules les opérations *push* et *pop* sont autorisées).

L'automate à pile de la figure 1 reconnaît les λ-termes de la grammaire. On remarquera que l'état initial q_1 est dupliqué dans le dessin de l'automate pour des raisons de clarté de la figure. Les symboles de pile utiles sont les suivants : `t`, λ, `@` et `(`. Les étiquettes notées avec un + en exposant (respectivement −) sont interprétées par des push (respectivement pop) sur la pile. Les étiquettes qui n'ont pas d'exposant sont interprétées par des opérations de segmentation (troncation) de la bande. Une exécution de cet automate est supposée être initialisée avec le mot à reconnaître écrit sur la bande et une pile vide. La condition d'acceptance de l'automate est que la bande soit vide et que la pile contienne seulement le symbole `t`. La transition de q_1 à q_f est étiquetée par `Accept` qui sera interprété par cette condition d'acceptance. En pratique on modélise la bande et la pile par deux listes sur deux alphabets différents. Les opérations sont des relations de type *relation* (*tape* $*$ *stack*), dans ce cas la première condition des machines d'Eilenberg finies est satisfaite. Mais aussi il existe une mesure fonction de l'état, de la bande et de la pile qui assure que la condition nœthérienne est satisfaite. Cette preuve a été formalisée dans l'assistant de preuve Coq.

Le mécanisme d'extraction de preuve de Coq fourni un automate à pile sous forme de module OCaml M de type *Kernel*. On utilise le moteur réactif pour simuler cet automate à pile. On obtient donc un reconnaisseur de λ-termes. Pour tout terme, le moteur réactif énumère toutes les solutions possibles, c'est-à-dire autant de solutions que de traces y conduisant. L'énumération est faite *à la demande* à l'aide de flux. Par exemple, l'exécution du moteur réactif pour l'entrée `"λx.x@(λx.λx.x@x)@x@x@λx.x@x"` produit un flux de longueur 522, c'est-à-dire qu'il y a 522

(a) (b)

Fig. 2 – Temps de calcul de la simulation pour : 2(a) la première solution pour des termes engendrés aléatoirement et 2(b) la solution unique pour des termes non-ambigus.

manières différentes de reconnaître ce terme ; le terme `"x@x@x@x@x@x@x@x@x@x@x@x@x"` produit un flux de longueur 208012.

L'énumération des solutions se fait par un parcours en profondeur d'abord. C'est la stratégie du moteur réactif pour les machines d'Eilenberg finies. Cette stratégie est adaptée pour cette simulation et elle est beaucoup plus efficace que ne le serait une stratégie en largeur d'abord. Nous illustrons les performances de la simulation avec deux bancs d'essais. La figure 2(a) montre les temps de calcul pour obtenir une première solution pour des termes engendrés aléatoirement, l'axe des abscisses représente la taille des termes et l'axe des ordonnées les temps de calcul. La figure 2(b) montre les temps de calcul pour obtenir l'unique solution pour un ensemble de terme aléatoires mais non-ambigus (complètement parenthésés). Ces deux bancs d'essais indiquent que la reconnaissance est en moyenne linéaire en la taille du terme. Le moteur réactif est donc efficace comme attendu.

Nous avons montré cet exemple pour illustrer le fait que les machines d'Eilenberg finies, bien qu'apparemment reposant sur les automates finis, permettent de résoudre des problèmes plus compliqués tels que la reconnaissance de langage hors-contexte (*context-free language* en anglais).

5.8 Applications

Les automates, transducteurs et plus généralement les machines d'états finis sont des techniques couramment utilisées pour résoudre des problèmes de linguistique computationnelle [KK94, Kar00, Moh97, RS95, RS97, Spr92]. Nous pensons que les machines d'Eilenberg sont très prometteuses dans ce domaine, en particulier grâce à la propriété de modularité (voir section 4.2). En fait, le modèle des *machines d'Eilenberg finies* est déjà le formalisme sous-jacent à des travaux de modélisation de la morphologie et de la phonologie [Hue05, HR06] dans le cadre du sanskrit (langue savante, phonétique et codifiée, de l'Inde antique). En sanskrit, l'analyse du sandhi est un problème d'*analyse* de transducteurs. Les règles de sandhi décrivent les liaisons qui déforment les mots aux frontières lorsqu'ils sont juxtaposés. Pour résoudre le sandhi il s'agit de défaire les liaisons dans une phrase. Le problème est particulièrement compliqué en sanskrit parce que la

langue écrite reflète précisément le flot oral, il n'y donc pas de séparation entre les mots. Pour faire l'analyse d'une phrase sanskrite il faut donc simultanément trouver les mots (comme appartenant à un lexique) et défaire les liaisons (décrite par le sandhi). Ces liaisons sont décrites par des règles de transduction. Le problème de sandhi se résout par une analyse de transducteurs définis par des lexiques annotés de règles de sandhi (transductions). On peut en faire plus et essayer de prendre en compte la morphologie des mots, pour cela il faut ajouter un automate, qu'on appelle automate de phases [HR06], qui décrit les séquences possibles de recherche dans les lexiques. Il y a donc deux niveaux d'automates qui permettent de décrire la morphologie d'une phrase sanskrite. Nous avons décrit un moteur réactif qui mélange ces deux niveaux d'automates dans [HR06]. Il se trouve que ces deux niveaux d'automates sont des instances de machines d'Eilenberg finies. La preuve de correction du moteur réactif de [HR06] était complexe parce que justement elle mélangeait ces deux niveaux. En utilisant les machines d'Eilenberg finies et leur moteur réactif, on peut identifier chaque niveau plus disctinctement et obtenir la preuve de correction de la simulation à partir de la preuve de correction du moteur réactif générique.

6 Machines d'Eilenberg effectives dans un cadre général de calculabilité

6.1 Moteur réactif avec stratégies

On suppose maintenant que les machines d'Eilenberg sont générales, c'est-à-dire que la sémantique associe à chaque générateur de Σ des relations calculables productives (utilisant des flux à trois constructeurs). En s'inspirant du moteur réactif pour les machines d'Eilenberg finies on peut définir un autre moteur réactif paramétré par une *stratégie* pour la recherche de solutions. Dans le cas des machines finies ci-dessus la stratégie correspondante est le parcours en profondeur d'abord. Dans le cas général, on fournit une stratégie *Complete* qui se trouve être complète parce qu'elle est équitable. Pour que la stratégie soit complète il faut que les relations soient productives sinon une relation peut bloquer la simulation. Donc si on veut appliquer ce moteur réactif modulairement à tous les niveaux d'un calcul décrit par un assemblage de machines d'Eilenberg on ne peut que souhaiter la productivité du flux de simulation calculé par le moteur réactif. Dans la présente version du moteur réactif cette propriété sera garantie statiquement ; c'est-à-dire qu'on pourra vérifier, en analysant le listing du moteur, que tout chemin d'exécution du moteur aboutit nécessairement à la production d'un élément de flux (*Done*, *Skip* ou *Elm*). Rappellons le type des flux et le type des relations :

```
type flux 'data =
  [ Done
  | Skip of delay 'data
  | Elm of 'data and delay 'data
  ]
and delay 'data = unit → flux 'data; (* frozen flux *)

type relation 'data = 'data → flux 'data;
```

Le noyau d'une machine d'Eilenberg effective est similaire à celui des machines finies *Kernel* ; la seule différence est le type des flux utilisé dans la sémantique de données *semantics* :

```
module type EMK =sig
  type generator;
  type data;
```

```
  type state;
  value transition : state → list (generator * state);
  value initial : list state;
  value accept : state → bool;
  value semantics : generator → relation data;
end;
```

Une machine d'Eilenberg peut être mise en paramètre d'un foncteur qui donnera *in fine* trois modules implémentant trois stratégies différentes de recherche de solutions :

```
module Engine (Machine: EMK) :
  sig
    type backtrack;
    module Strategy (Resumption : sig
        type resumption;
        value empty : resumption;
        value pop : resumption → option (backtrack * resumption);
        value push : backtrack → resumption → resumption;
      end) : (sig value simulation : relation Machine.data; end);

    module FEM : (sig value simulation : relation Machine.data; end);
    module Deterministic_Engine : (sig value simulation : relation Machine.data; end);
    module Complete_Engine : (sig value simulation : relation Machine.data; end);
  end = struct
open Machine;
```

Le module *Engine* est un foncteur qui produira une structure contenant un type de données *backtrack* pour les points de choix, un foncteur *Strategy* retournant une fonction de simulation. Le foncteur *Strategy* est paramétré par un module *Resumption* qui modélise une file de priorité sur les résomptions. Ce foncteur sera utilisé sur trois files de priorité différentes, afin de produire trois modules de simulation : *FEM*, *Deterministic_Engine* et *Complete_Engine*.

On utilise le type *choice* pour représenter les listes de paires générateur-état :

```
type choice = list (generator * state);
```

Le type des points de choix est le suivant :

```
type backtrack =
  [ React of data and state
  | Choose of data and choice
  | Relate of flux data and state
  ];
```

Un point de choix est de trois natures différentes. Le premier, *React*, indique qu'on veut avancer le calcul de la trace à partir d'une cellule (il est similaire au constructeur *Advance* des machines d'Eilenberg finies). Les deux autres constructeurs, *Choose* et *Relate*, servent à gérer les deux types de non-déterminisme, à savoir celui du contrôle et celui des données (cela correspond à séparer en deux le constructeur *Choose* des machines d'Eilenberg finies).

Désormais, la résomption est un paramètre abstrait, fourni dans un module *Resumption* qui modélise la politique de gestion des points de choix sous forme de file de priorité. Le module *Resumption* est passé en argument à un foncteur *Strategy* qui retournera une fonction de simulation.

```
module Strategy (* resumption management *)
  (Resumption : sig
    type resumption;
    value empty : resumption;
    value pop : resumption → option ( backtrack * resumption);
    value push : backtrack → resumption → resumption;
  end) : (sig value simulation : relation Machine.data; end) = struct
open Resumption;
```

Le moteur réactif se retrouve maintenant paramétré par la gestion de la résomption. La résomption est du type *resumption*, le paramètre *empty* correspond à la résomption vide, la fonction *pop* prend en argument une résomption et a pour valeur optionnelle de retour une paire composée d'un point de choix et de la résomption qui reste, ou de la valeur optionnelle *None* si la résomption est vide. La fonction *push* ajoute un point de choix dans une résomption et a pour valeur de retour la nouvelle résomption.

Le moteur réactif est maintenant constitué de quatre fonctions *react*, *choose*, *relate* et *resume*. On remarquera que les trois premières fonctions portent les noms des constructeurs de points de choix de type *backtrack*.

```
(* react : data → state → resumption → flux data *)
value rec react d q res =
  let ch = transition q in
  let res' = push (Choose d ch) res in
  if accept q
    then Elm d (fun () → resume res') (* Solution d found *)
    else Skip (fun () → resume res')

(* choose : data → choice → resumption → flux data *)
and choose d ch res =
  match ch with
  [ [] → Skip (fun () → resume res)
  | [ (g, q') :: rest ] →
    let res1 = push (Choose d rest) res in
    let res2 = push (Relate (semantics g d) q') res1 in
    Skip (fun () → resume res2)
  ]

(* relate : flux data → state → resumption → flux data *)
and relate str q res =
  match str with
  [ Done → Skip (fun () → resume res)
  | Skip del →
    let str = del () in
    Skip (fun () → resume (push (Relate str q) res))
  | Elm d del →
    let str = del () in
    let res1 = push (Relate str q) res in
    let res2 = push (React d q) res1 in
    Skip (fun () → resume res2)
  ]
```

```
(* resume : resumption → flux data *)
and resume res =
  match pop res with
  [ None → Done
  | Some c →
    let (b, res) = c in
    match b with
    [ React d q → react d q res
    | Choose d ch → choose d ch res
    | Relate str q → relate str q res
    ]
  ];
```

Les quatre fonctions sont paramétrées par la résomption courante notée *res*. La fonction *resume* cherche un point de choix dans la résomption avec la fonction *pop* et ensuite branche le point de choix obtenu sur l'une des trois autres fonctions. La fonction *react* est paramétrée par une cellule pour laquelle on calcule le point de choix de contrôle puis on l'ajoute à la résomption courante ; ensuite, si la cellule est dans un état acceptant alors la donnée de la cellule est retournée dans un élément de flux, sinon on retourne un flux constitué d'une pause suivi d'une queue calculée par un appel retardé à *resume*. La fonction *choose* considère une donnée d et un ensemble de transitions *ch*. S'il n'y a plus de choix de transitions alors on retourne un flux constitué d'une pause suivi d'une queue calculée par un appel retardé à *resume*, sinon on décompose la liste de transitions par sa tête et son reste, et on ajoute à la résomption les points de choix adéquats puis on retourne un flux constitué d'une pause suivi d'une queue calculée par un appel retardé à *resume*. La fonction *relate* considère un flux de données *str* pouvant aboutir à un état q. Si *str* est vide alors on retourne un flux défini d'une pause suivi d'une queue calculée par un appel retardé à *resume*. Si *str* est constitué d'une pause en tête alors on ajoute sa queue dans un point de choix et on retourne un flux constitué d'une pause suivie et d'une queue calculée par un appel retardé à *resume*. Si *str* est constitué d'un élément en tête alors on ajoute le point de choix avec la queue du flux, et on ajoute le point de choix pour la cellule (constitué de l'élément et de q), puis on retourne un flux constitué d'une pause et d'une queue calculée par un appel retardé à *resume*.

On remarque que tous les appels à *react*, *choose* et *relate* retournent un flux constitué d'un élément en tête ou d'une pause. On peut donc assurer que si les flux de la machine sont productifs alors le moteur réactif construit un flux productif.

La simulation complète de la machine est effectuée par la fonction suivante qui initialise une résomption pour une donnée d'entrée sur tous les états initiaux de la machine.

```
(* simulation : relation data *)
value simulation d =
  let rec init_res l acc =
    match l with
    [ [] → acc
    | [ q :: rest ] → init_res rest (push (React d q) acc)
    ] in
  resume ( init_res initial empty)
;

end; (* module Strategy *)
```

Cela termine la définition du moteur réactif paramétré par une stratégie. Donnons maintenant quelques stratégies typiques. On commence par retrouver la stratégie en profondeur d'abord des machines d'Eilenberg finies.

```
module DepthFirst = struct
type resumption = list backtrack;
value empty = [];
value push b res = [ b :: res ];
value pop res =
  match res with
  [ [] → None
  | [ b :: rest ] → Some (b,rest)
  ];
end; (* module DepthFirst *)
```

Cette stratégie correspond à une file de priorité réalisée par une pile ; c'est-à-dire que le point de choix le plus prioritaire est le dernier entré dans la pile. Cette stratégie se retrouvait implémentée implicitement dans le moteur réactif des machines d'Eilenberg finies.

Nous allons maintenant donner une stratégie pour des machines déterministes :

Définition 10. On dit qu'une machine est *déterministe* si pour toute donnée d et tout état q, il y a au plus un générateur a dans les transitions de q tel que $|\rho(a)(d)| > 0$ et pour ce a on a $|\rho(a)(d)| \leq 1$.

Cette notion de déterministe a son analogue pour les automates finis déterministes, à savoir qu'il n'y a qu'une seule transition qui peut faire avancer le mot de la bande. Pour les machines déterministes, on définit la stratégie suivante qui exploite la propriété de déterminisme en se permettant de jeter des points de choix produits par le moteur mais dont on sait qu'ils ne sont pas utiles :

```
module Det = struct
type resumption = list backtrack;
value empty = [];
value push b res =
  match b with
  [ React _ _ → [ b :: res ]
  | Choose _ _ → [ b ] (* cut : the list contains only one element *)
  | Relate _ _ → res (* no other delay *)
  ];
value pop res =
  match res with
  [ [] → None
  | [ b :: rest ] → Some (b,rest)
  ];
end; (* module Det *)
```

Pour finir, on donne une stratégie qu'on qualifiera d'*équitable* (*fair* en anglais) parce qu'elle explore l'espace de recherche du haut vers le bas (*top-down* en anglais) en mode boustrophédon. Il se trouve que cette stratégie est une parmi d'autres qui sont complètes pour toute machine d'Eilenberg effective :

```
module Complete = struct
```

```
type resumption = ( list backtrack * list backtrack);
value empty = ([],[]);
value push b res =
  let ( left , right) = res in
  ( left , [b :: right]);
value pop res =
  let ( left , right) = res in
  match left with
  [ [] → match right with
          [ [] → None
          | [ r :: rrest ] → Some (r, ( rrest , []))
          ]
  | [ l :: lrest ] → Some (l, ( lrest , right))
  ];
end; (* module Complete *)
```

Cette stratégie correspond à un parcours en largeur d'abord.

On instancie ces trois stratégies pour obtenir trois simulateurs différents :

```
module FEM =Strategy DepthFirst; (* for finite Eilenberg machines *)
module Deterministic_Engine = Strategy Det; (* The deterministic engine *)
module Complete_Engine =Strategy Complete; (* The fair engine *)

end; (* module Engine *)
```

Pour la stratégie *Complete* on pourrait s'assurer d'avoir une propriété de cohérence et complétude qu'on formulerait de manière similaire à celle des machines d'Eilenberg finies :

$$\forall d\ d', InFlux(d', simulation\ d) \Leftrightarrow Solution(d, d').$$

Nous laissons cette question à l'état de problème ouvert, mais nous indiquons que pour prouver la complétude il serait nécessaire de s'assurer que la stratégie respecte les deux conditions suivantes :

1. la stratégie n'oublie aucun point de choix de la résomption,
2. la stratégie ne retarde pas indéfiniment l'examen d'un point de choix (équité).

Nous pourrions envisager de faire une formalisaton en Coq de cette machine réactive, comme nous l'avons fait pour le moteur réactif des machines d'Eilenberg finies. Pour cela, il faut commencer par modéliser les flux possiblement infinis à la manière de Coutts, Leshchinskiy et Stewart [CLS07]. Leur type de flux permet de les définir en Coq sans utiliser les coinductifs, et cela après avoir explicité le type de la *"mémoire"* interne :

```
Inductive step (data s : Set) : Set :=
  | Done : step data s
  | Skip : s → step data s
  | Yield : data → s → step data s.

Inductive flux (data:Set) : Set :=
  MkFlux : ∀ (s : Set), (s → step data s) → s → flux data.
```

Un flux de données de type *data* est défini par la paire d'un état d'un certain type s (la mémoire) et d'une fonction qui associe à tout élément de s un constructeur indiquant la fin du flux (*Done*), ou une pause (*Skip*), ou encore un élément de flux (*Yield*). On remarque qu'il n'est pas nécessaire d'utiliser de type coinductif pour représenter ces flux. Cette représentation des flux devrait aussi

permettre de modéliser le moteur réactif comme quatre fonctions indépendantes qui ne sont plus mutuellement récursives (*react*, *choose*, *relate* et *resume*).

On dit du moteur qu'il est *réactif* parce qu'on peut vérifier statiquement que le flux construit est productif. En effet, pour peu que le module de résomption n'admette que des fonctions qui terminent, alors chaque appel aux fonctions *react*, *choose* et *relate* a pour valeur de retour un constructeur d'élément de flux ou une pause. Cependant, cela présente un inconvénient par rapport au moteur réactif des machines d'Eilenberg finies par exemple. Ce dernier ne produisait aucun élément de type *Skip*, alors que le présent moteur en produit autant que d'appels nécessaires aux fonctions composant le moteur. Pour cette raison nous allons présenter un moteur semi-réactif dans la section suivante qui solutionne ce problème.

6.2 Moteur semi-réactif avec stratégies

Nous allons définir un *moteur semi-réactif* dont la garantie de productivité des flux de la relation caractéristique est à la charge de la stratégie. De cette manière on peut définir une stratégie de recherche en profondeur d'abord qui se comporte comme pour le cas des machines finies (sans créer de pauses inutiles dans le flux). On peut définir aussi l'analogue de la stratégie complète qui engendre autant de pauses que la stratégie complète du moteur réactif.

Le programme du moteur semi-réactif est très semblable au moteur réactif de la section précédente, nous ne commenterons que les différences notables :

```
module Engine (Machine: EMK) =struct
open Machine;

type choice = list (generator * state);

type backtrack =
  [ React of data and state
  | Choose of data and choice
  | Relate of flux data and state
  ];
```

On introduit un nouveau type pour indiquer lorsque l'appel d'une résomption devra être précédé par une pause à l'aide *Skip* :

```
type waitgo = [ Wait | Go ];
```

Le module qui contient la stratégie est légèrement modifié dans sa spécification pour la fonction *pop* :

```
module Strategy (* resumption management *)
(Resumption : sig
  type resumption;
  value empty : resumption;
  value pop : resumption → option (waitgo * (backtrack * resumption));
  value push : backtrack → resumption → resumption;
end) = struct
open Resumption;
```

La fonction *pop* a pour valeur de retour un point de choix pour lequel il est indiqué s'il faut émettre une pause avant de continuer la recherche.

Le moteur semi-réactif a la même structure que précédemment :

```
(* react : data → state → resumption → flux data *)
value rec react d q res =
  let ch = transition q in
  let res' = push (Choose d ch) res in
  if accept q
  then Elm d (fun () → resume res') (* Solution d found *)
  else resume res'

(* choose : data → choice → resumption → flux data *)
and choose d ch res =
  match ch with
  [ [] → resume res
  | [ (g, q') :: rest ] →
    let res' = push (Choose d rest) res in
    let res'' = push (Relate (semantics g d) q') res' in
    resume res''
  ]

(* relate : flux data → state → resumption → flux data *)
and relate str q res =
  match str with
  [ Done → resume res
  | Skip del → let str = del () in
                resume (push (Relate str q) res)
  | Elm d del → let str = del () in
                 resume (push (React d q) (push (Relate str q) res))
  ]

(* resume: resumption → flux data *)
and resume res =
  match pop res with
  [ None → Done
  | Some (wg,(b,rest)) →
    let dispatch b rest =
      match b with
      [ React d q → react d q rest
      | Choose d ch → choose d ch rest
      | Relate str q → relate str q rest
      ] in
    match wg with
    [ Wait → Skip (fun () → dispatch b rest)
    | Go → dispatch b rest
    ]
  ];

(* simulation : relation data *)
value simulation d =
  let rec init_res l acc =
    match l with
```

```
    [ [] → acc
    | [ q :: rest ] → init_res rest (push (React d q) acc)
    ] in
  resume ( init_res initial empty)
;

end; (* module Strategy *)
```

Les différences sont les suivantes. Les fonctions *react*, *choose* et *relate* ne produisent plus de pauses directement. Les pauses sont uniquement émises par la fonction *relate*, lorsqu'elle reçoit cet ordre par l'intermédiaire d'une valeur *Wait* en provenance du module de résomption.

On retrouve le moteur réactif pour les machines d'Eilenberg finies avec la stratégie suivante :

```
module DepthFirst = struct
type resumption = list backtrack;
value empty = [];
value push b res = [ b :: res ];
value pop res =
  match res with
  [ [] → None
  | [ b :: rest ] → Some (Go, (b,rest))
  ];
end; (* module DepthFirst *)
```

Si la machine n'est pas une machine finie alors cette stratégie n'est ni productive ni encore moins complète.

On retrouve la stratégie complète du moteur réactif précédent avec la stratégie qui suit. Elle introduit prudemment des pauses à chaque étape de recherche :

```
module Complete =struct
type resumption = ( list backtrack * list backtrack);
value empty = ([],[]);
value push b res =
  let ( left , right) = res in
  ( left , [b :: right]);
value pop res =
  let ( left , right) = res in
  match left with
  [ [] → match right with
         [ [] → None
         | [ r :: rrest ] → Some (Wait, (r, ( rrest , [])))
         ]
  | [ l :: lrest ] → Some (Wait, (l, ( lrest , right)))
  ];
end; (* module Complete *)
```

On remarque que cette stratégie produit uniquement des résomptions avec un ordre de pause *Wait* (fonction *pop*). Elle a donc exactement le même comportement que la stratégie du moteur réactif.

Une formalisation, dans l'assistant de preuves Coq, de ce moteur semi-réactif est impossible parce qu'on n'a pas de propriétés pour garantir la terminaison des fonctions qui sont définies successivement ; c'est seulement avec une analyse globale de toutes les définitions qu'on pourrait envisager de prouver la terminaison.

7 Conclusion

Nous avons commencé ce chapitre en présentant un modèle de calcul à base de relations, les machines relationnelles. Elles ont l'avantage de présenter dans un formalisme symétrique un modèle de calcul non-déterministe mettant en scène deux composantes qui interagissent fortement, le contrôle et les données. Ce modèle de machines relationnelles, nous l'avons étudié sous une forme plus particulière qui sont les machines d'Eilenberg. À l'origine, le mathématicien Samuel Eilenberg y voyait un modèle unificateur pour traiter les différentes couches de la théorie des langages. Effectivement, un avantage de ce modèle de calcul est qu'il travaille sur un espace de données abstrait. Un autre avantage du modèle est sa modularité : on peut définir un calcul complexe par un assemblage de plusieurs machines d'Eilenberg. Pour cette raison les machines d'Eilenberg ne sont pas qu'un jouet théorique mais peuvent avoir des applications concrètes, en particulier en linguistique.

Nous avons proposé des méthodes pour simuler ce modèle de calcul de manière purement applicative. Les relations de calcul qui sont à la base du modèle se trouvent orientées sous forme de fonctions ayant pour valeur de retour des flux d'éléments. Dans un premier temps nous avons étudié la restriction des machines d'Eilenberg finies pour lesquelles les relations sont finies et pour lesquelles les calculs sont aussi finis. Nous avons dégagé un simulateur, le moteur réactif, qui de manière purement applicative effectue une recherche complète de toutes les solutions. Le développement est accompagné d'une preuve mathématique de la correction du moteur réactif, comprenant la terminaison de l'algorithme, mais aussi la cohérence et la complétude.

Cette étape a été essentielle pour généraliser, à partir du moteur réactif des machines d'Eilenberg finies, un moteur réactif destiné à des machines d'Eilenberg effectives. Ce moteur est paramétré par un module de stratégie. Il s'agit de la stratégie de recherche de solutions. En particulier on peut encoder la stratégie du parcours en profondeur d'abord qui est celle implicitement utilisée pour les machines d'Eilenberg finies. Nous avons aussi fourni une stratégie complète qui entrelace tous les choix possibles lors de la recherche de solutions, avec la garantie d'être équitable pour chaque choix dans la recherche. Une propriété importante de ce moteur réactif est qu'on sait statiquement que le flux de solutions de sortie est productif. C'est indispensable pour obtenir la modularité des machines ; en effet la stratégie complète peut être utilisée pour simuler un calcul défini comme un assemblage de plusieurs machines d'Eilenberg, avec une garantie de complétude d'énumération de toutes les solutions. Cette garantie a un coût, qui est d'insérer des pauses dans le flux, alors que parfois une propriété nous permettrait d'être moins prudent (comme pour les machines d'Eilenberg finies). Pour cette raison nous avons proposé une dernier simulateur, le moteur semi-réactif. Il est qualifié de semi-réactif parce qu'on n'a aucune garantie que les flux calculés soient productifs. Grâce à ce moteur semi-réactif on peut retrouver la simulation exacte des machines d'Eilenberg finies par une stratégie adéquate, et on peut retrouver aussi la stratégie complète du moteur réactif pour machines d'Eilenberg effectives.

Nous avons donc proposé des techniques pour simuler effectivement un modèle général d'automates. La modularité d'une part et la généralité du modèle des machines d'Eilenberg d'autre part sont des avantages par rapport aux autres travaux proposés jusque maintenant [Tho00, ZS01, Sar02] qui ont pour but de présenter des simulateurs de machines d'états finis.

Chapitre III

Algorithmes Fonctionnels de Synthèse d'Automates

Nous présenterons les principaux algorithmes de synthèse d'automates finis à partir d'expressions régulières. La technique des dérivées pour les expressions régulières est l'outil qui permet de concevoir des algorithmes efficaces de synthèse d'automates aussi bien déterministes que non-déterministes. Nous présentons les différents algorithmes de synthèse à base de cette technique et faisons un état de l'art de ceux qui produisent en particulier des automates non-déterministes. Les algorithmes seront complètement explicités en OCaml.

1 Structures primitives et algorithme de Thompson

1.1 Expressions régulières

Nous allons décrire un type de données pour les *expressions régulières* de signature $(\Sigma, +, \cdot, ^*, 0, 1)$. Nous utilisons un type algébrique pour décrire de telles valeurs :

```
type regexp 'a =
  [ Eps
  | Symb of 'a
  | Union of regexp 'a and regexp 'a
  | Conc of regexp 'a and regexp 'a
  | Star of regexp 'a
  ];
```

Il y a un constructeur par opérateur des expressions régulières plus ceux pour l'élément neutre et les symboles. On a choisi de nommer les constructeurs avec les noms des opérations associées sur le modèle des langages, c'est-à-dire que la somme est appelée *Union* pour l'union de langages, et que le produit est appelé *Conc* pour la concaténation de langages. On ne considérera pas l'élément 0 de l'algèbre de Kleene parce qu'il ne joue pas de rôle significatif pour définir des langages. Les symboles apparaissant dans une telle expression n'ont pas de type prédéfini ; cela se retrouve dans le type des expressions régulières qui est paramétré par un type abstrait *'a*. Par exemple on définit aussi bien des expressions dont les symboles sont des caractères que des chaînes de caractères, ou bien même des entiers :

```
value a = Star (Symb 'a');
value b = Star (Symb "b");
value c = Star (Symb 3);
```

Nous illustrerons les algorithmes présentés dans la suite sur l'expression régulière suivante :

Exemple 3. Soit E l'expression régulière suivante :

$$E = a(b(a^*c + d)^* + e) + d(a^*c + d)^*$$

On définit E en OCaml de la manière suivante :

```
value exm =
  let a = Symb 'a' in
  let b = Symb 'b' in
  let c = Symb 'c' in
  let d = Symb 'd' in
  let e = Symb 'e' in
  let exp = Star (Union (Conc (Star a) c) d) in
  let e1 = Conc a (Union (Conc b exp) e) in
  let e2 = Conc d exp in
  Union e1 e2;
```

En ajoutant une fonction d'impression qui traduit une telle expression en une formule mathématique on obtient :

$$(a{\cdot}(b{\cdot}(a^*{\cdot}c + d)^* + e) + d{\cdot}(a^*{\cdot}c + d)^*) \ .$$

Beaucoup d'exemples présentés dans ce chapitre seront en fait des résultats issus d'algorithmes qu'on insère dans le document.

1.2 Automates finis non-déterministes

Les états de l'automate seront le plus souvent représentés par des entiers mais on se laisse la liberté qu'ils soient représentés par d'autres types de données, ce paramètre de type sera *'state*. Les étiquettes des transitions des automates non-déterministes peuvent être des symboles mais aussi des étiquettes de transitions spontanées (ϵ), on paramétrise donc le type des étiquettes par *'label*. À chaque état on associe une liste de transitions possibles, représentées par des paires étiquette-état. Finalement un automate est un triplet formé d'un état initial, d'une table de transitions et d'une liste d'états acceptants.

```
type fanout 'state 'label = ('state * list ('label * 'state))
and transitions 'state 'label = list (fanout 'state 'label);

type automaton 'state 'label =
  ( 'state * ( transitions 'state 'label) * list 'state );
```

Ce choix de type de données pour représenter les automates a l'avantage d'avoir une taille de représentation linéaire par rapport aux automates tels qu'on les dessine ; c'est-à-dire linéaire par rapport au nombre effectif de transitions. Une représentation matricielle aurait l'inconvénient d'être quadratique en le nombre d'états de l'automate, alors que la matrice est souvent *creuse* pour de nombreux automates.

Par exemple, le langage défini par l'expression $a^* \cdot b \cdot c^*$ est décrit par l'automate suivant :

```
value aut1 = (0, [ (0, [('a',0) ; ('b',1)] ) ; (1, [('c',1)]) ], [1]);
```

Cette valeur a le type *automaton int char*, les états sont donc du type des entiers et les étiquettes sont du type des caractères.

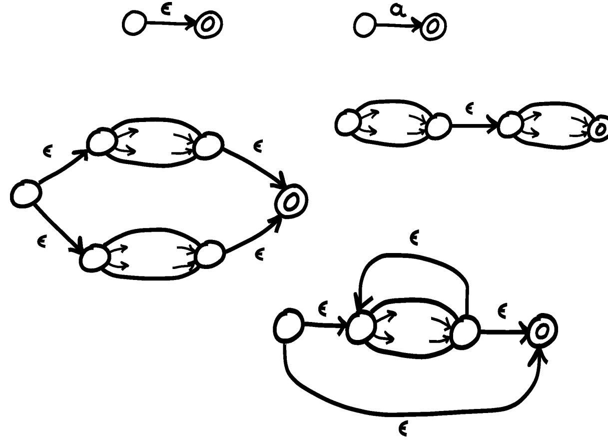


FIG. 1 – Algorithme de Thompson

1.3 Algorithme de Thompson

L'algorithme de Thompson construit un automate non-déterministe à transitions spontanées (ϵ-NFA). À chaque sous-expression de l'entrée est associé un tel automate, la construction se fait récursivement sur la structure de l'expression régulière et peut être représentée de manière schématique (voir figure 1). L'implémentation que nous proposons suit cette définition schématique :

```
value thompson : regexp 'a → automaton int (option 'a);

value thompson e =
  let rec thompson_rec e t n =
    match e with
    [ Eps → let n1=n+1 and n2=n+2 in
      (n1, [ (n1, [ (None, n2) ]) :: t ], n2)
    | Symb s → let n1=n+1 and n2=n+2 in
      (n1, [ (n1, [ (Some s, n2) ]) :: t ], n2)
    | Union e1 e2 →
      let (i1,t1,f1) = thompson_rec e1 t n in
      let (i2,t2,f2) = thompson_rec e2 t1 f1 in
      let n1=f2+1 and n2=f2+2 in
      (n1, [ (n1, [ (None, i1); (None, i2) ]) ::
                  [ (f1, [ (None, n2) ]) ::
                  [ (f2, [ (None, n2) ]) :: t2 ] ] ], n2)
    | Conc e1 e2 →
      let (i1,t1,f1) = thompson_rec e1 t n in
      let (i2,t2,f2) = thompson_rec e2 t1 f1 in
      (i1, [ (f1, [ (None, i2) ]) :: t2 ], f2)
    | Star e1 →
      let (i1,t1,f1) = thompson_rec e1 t n in
      let n1=f1+1 and n2=f1+2 in
      let t1' = [ (f1, [ (None, i1); (None, n2) ]) :: t1 ] in
      (n1, [ (n1, [ (None, i1); (None, n2) ]) :: t1' ], n2)
    ] in
```

```
let (i,t,f) = thompson_rec e [] 0 in
(i, [(f,[]) :: t], [f]);
```

Dans cet algorithme les additions servent à allouer de nouveaux états et la taille de l'ensemble des états déjà alloué au cours de la récursion est le paramètre n. Deux modifications sont à apporter à la structure intermédiaire d'automate produit par la fonction *thompson_rec*. Premièrement, puisque l'algorithme de Thompson a comme invariant que l'automate produit a un unique état acceptant, il faut transformer cet état en une liste d'états pour coller à notre spécification d'automate non-déterministe. Deuxièmement, il manque la transition pour l'état acceptant. La dernière ligne dans la définition de *thompson* effectue ces deux modifications.

Théorème III.1 (Correction de l'algorithme de Thompson)**.** Pour toute expression régulière e, le langage associé à e est égal au langage reconnu par *thompson e*.

Démonstration. Il faut s'assurer que les états de toutes les sous-expressions disjointes sont différents, autrement dit que l'allocation d'états est correcte. Pour ce faire il faut s'assurer que le paramètre correspondant à l'entier représentant l'état initial de l'automate produit est bien "frais" par rapport au contexte des autres automates. Avec ces précautions on peut faire une preuve par récurrence sur la structure de l'expression régulière. □

Bien que l'algorithme de Thompson appartienne au folklore de la discipline, l'implémentation que nous en proposons présente l'originalité d'être fonctionnelle et aussi directe que sa formulation schématique. Naïvement, l'automate est construit en manipulant récursivement des sous-automates, en fusionnant des états et en créant des transitions entre états. Notre approche est plus directe parce que l'espace des états de l'automate est créé lors du parcours de l'expression et on s'assure que les espaces d'états des sous-automates ne se superposent pas ; cela conduit à un algorithme aussi direct que sa représentation schématique (figure 1), c'est-à-dire linéaire en la taille de l'expression d'entrée.

L'automate obtenu par l'algorithme de Thompson est un ϵ-NFA, et il y aurait de nombreuses manières de modifier l'algorithme pour obtenir un automate plus petit. Nous n'avons pas cherché à faire l'état de l'art des algorithmes compilant ce type d'automates, le lecteur curieux pourra se référer au livre de Sippu et Soisalon-Soininen [SS88]. Récemment, Ilie et Yu [IY03] ont proposé un algorithme plus efficace encore. On peut aussi mentionner les travaux de Raymond [Ray96] qui propose un algorithme avec des applications pour le domaine des langages synchrones.

Dans la suite nous nous intéresserons à synthétiser des NFA sans transition spontanée. Cependant on peut déjà améliorer l'automate synthétisé par l'algorithme de Thompson en réduisant au préalable l'expression régulière d'entrée.

2 Réduire les expressions

Les expressions régulières peuvent être simplifiées en utilisant différents axiomes de l'algèbre de Kleene utilisés comme réductions de termes. Nous proposons dans cette partie différentes réductions qui jouent un rôle essentiel dans les algorithmes de synthèse présentés dans la suite.

2.1 Expressions régulières décorées

Les algorithmes sur les expressions sont souvent décrits par une succession de passes associant des informations à chaque nœud qui sont exploitées au cours du calcul. Il est donc utile de fournir un type de données pour les expressions décorées, c'est-à-dire dont les nœuds sont annotés par des valeurs arbitraires. Voici le type de structure de données adéquat que nous utiliserons par la suite :

```
type deco_regexp 'a 'b =
  [ DEps
  | DSymb of 'a
  | DUnion of dregexp 'a 'b and dregexp 'a 'b
  | DConc of dregexp 'a 'b and dregexp 'a 'b
  | DStar of dregexp 'a 'b
  ]
and dregexp 'a 'b = ( (deco_regexp 'a 'b) * 'b);
```

On appelle *deco_regexp* cette structure de données d'expressions régulières parce qu'on dit de ses nœuds qu'ils sont *annotés* ou encore *décorés*. Le type *deco_regexp 'a 'b* est très similaire au type *regexp 'a*. Pour une expression donnée les *annotations* ou *décorations* ont toutes le même type représenté par un type abstrait *'b*. Une paire formée d'une *deco_regexp 'a 'b* et d'une annotation *'b* est une expression de type *dregexp 'a 'b*.

Pour illustrer l'usage de telles décorations, nous proposons d'annoter une expression régulière par des booléens à chaque nœud indiquant si le langage dénoté par la sous-expression associée contient le mot vide. L'expression régulière résultante sera du type *dregexp 'a bool*. L'annotation de toute une expression se fait en parcourant l'expression avec la fonction *nullify* et on teste si une expression contient le mot vide en utilisant la fonction *null*.

```
value null : dregexp 'a 'b → 'b;
value nullify : regexp 'a → dregexp 'a bool;
value null e = snd e;
value rec nullify e =
  match e with
  [ Eps → (DEps, True)
  | Symb s → (DSymb s, False)
  | Union e1 e2 →
     let e1' = nullify e1
     and e2' = nullify e2 in
     (DUnion e1' e2', (null e1' || null e2'))
  | Conc e1 e2 →
     let e1' = nullify e1
     and e2' = nullify e2 in
     (DConc e1' e2', (null e1' && null e2'))
  | Star e → (DStar (nullify e), True)
  ];
```

La fonction *null* est simplement la fonction qui renvoie la deuxième composante d'une paire, c'est-à-dire l'annotation dans le cas d'une expression décorée. La fonction *nullify* exploite les annotations des sous-expressions pour combiner à l'aide des opérateurs booléens *et* et *ou* (respectivement && et ||) les booléens pour les opérations union et concaténation. Lorsque l'expression est une étoile alors le langage associé contient nécessairement le mot vide ϵ.

Par exemple, pour l'expression régulière

$$(a^*{\cdot}b + 1){\cdot}(1 + b)$$

le calcul des annotations indiquant la présence du mot vide ϵ est :

$$(a^*{\cdot}_f b{+}_t 1){\cdot}_t(1{+}_t b) \ .$$

On indique la présence du mot vide ϵ par un petit t en indice et l'absence du mot vide par un petit f en indice. Sur l'expression de l'exemple 3, le calcul de la fonction produit les annotations suivantes (on a omis d'indiquer le booléen pour le cas de l'étoile puisqu'il est toujours vrai) :

$$(a\cdot_f(b\cdot_f(a^*\cdot_f c+_f d)^*+_f e)+_f d\cdot_f(a^*\cdot_f c+_f d)^*)\ .$$

Un autre exemple d'annotation qui nous sera utile par la suite est la *linéarisation*. Une expression régulière est *linéaire* lorsque les symboles qui la composent sont uniques dans cette expression. On peut *linéariser* une expression régulière en associant à chaque symbole un entier unique :

```
value linearize : regexp 'a → regexp ('a * int);
value linearize e =
  let rec aux e i =
    match e with
    [ Eps → (Eps, i)
    | Symb s → (Symb (s,i), i+1)
    | Union e1 e2 →
      let (e1',i1) = aux e1 i in
      let (e2',i2) = aux e2 i1 in
      (Union e1' e2', i2)
    | Conc e1 e2 →
      let (e1',i1) = aux e1 i in
      let (e2',i2) = aux e2 i1 in
      (Conc e1' e2', i2)
    | Star e →
      let (e', i') = aux e i in (Star e', i')
    ] in
  let (e',_) = aux e 1 in
  e';
```

Cet algorithme a une complexité linéaire en la taille de l'expression. La linéarisation s'effectue avec un parcours en profondeur d'abord de gauche à droite de l'expression régulière. Les symboles d'une expression régulière linéaire font référence à des positions uniques dans l'expression. On notera donc $Pos(E)$ l'ensemble des positions d'une expression régulière. Sur l'expression de l'exemple 3 la linéarisation produit l'expression décorée suivante :

$$(a_1(b_2(a_3{}^*c_4+d_5)^*+e_6)+d_7(a_8{}^*c_9+d_{10})^*)\ .$$

On appelle *délinéarisation* l'opération inverse de la linéarisation qui efface les entiers de l'expression linéarisée. On notera cette fonction h dans la suite.

2.2 Associativité de la concaténation

Choisissons l'axiome d'associativité de la concaténation. On peut appliquer récursivement cette égalité sur toutes les sous-expressions d'une expression régulière pour obtenir une expression en forme normale dont toutes les concaténations successives sont associatives à gauche :

```
value la_conc_nf : regexp 'a → regexp 'a;
value la_conc_nf e =
  let rec aux e =
    match e with
    [ Eps → e
    | Symb s → e
```

```
    | Union e1 e2 →
      let le1 = aux e1 in
      let le2 = aux e2 in
      Union le1 le2
    | Conc e1 e2 →
      let le1 = aux e1 in
      let rec f left e2 = match e2 with
        [ Conc e3 e4 → let le3 = f left e3 in f le3 e4
        | _ → Conc left (aux e2)
        ] in
      f le1 e2
    | Star e → Star (aux e)
    ] in
  aux e;
```

La fonction récursive *aux* parcourt en profondeur d'abord l'expression pour appliquer l'associativité de la concaténation. Pour que la complexité de l'algorithme soit linéaire en la taille de l'expression il faut introduire la fonction récursive locale *f* pour le cas de la concaténation. La fonction *f* attend deux expressions *left* et *e2* en paramètres, et a pour résultat leur concaténation sous forme normale utilisant l'associativité. L'expression *left* doit être préalablement mise en forme normale d'associativité de la concaténation. Par illustrer l'effet de la fonction *la_conc_nf*, on considère l'expression d'entrée

$$((((a \cdot a) \cdot (1 \cdot 1)) \cdot (b \cdot (b \cdot (b \cdot b)))) + ((a \cdot (a \cdot (a \cdot a^*))))^*)$$

qui est mise en forme normale pour l'associativité à gauche de la concaténation en

$$(((((((((a \cdot a) \cdot 1) \cdot 1) \cdot b) \cdot b) \cdot b) \cdot b) + ((((a \cdot a) \cdot a) \cdot a^*))^*)\ .$$

On remarque que cet algorithme peut être reproduit et adapté pour l'associativité à droite de la concaténation mais aussi pour l'associativité de l'union à gauche ou à droite.

2.3 ϵ-réduction

On appelle *ϵ-réduction* l'opération qui remplace par *Eps* tous les nœuds qui dénotent uniquement le mot vide ϵ et simplifie l'expression selon la neutralité par rapport à la concaténation [1].

Pour effectuer cette réduction remarquons les faits suivants. Tout d'abord on sait que *Eps* est un élément neutre pour la concaténation, on peut donc réduire des concaténations de langages dénotant le mot vide par *Eps*. Ensuite, pour une expression de la forme *Union e Eps*, si *e* contient le mot vide alors elle est équivalente à *e*. Et enfin, l'expression *Star Eps* peut être remplacée par *Eps*. On obtient la forme normale correspondante en utilisant la fonction suivante :

```
value reduce_eps : dregexp 'a bool → dregexp 'a bool;
value rec reduce_eps e =
  let (e', d) = e in
  match e' with
  [ DEps → e
  | DSymb _ → e
  | DUnion e1 e2 →
```

[1] Il ne faut pas confondre l'ϵ-réduction avec une opération plus puissante qui éliminerait le maximum de nœuds ϵ d'une expression régulière.

```
    let e1r = reduce_eps e1 in
    let e2r = reduce_eps e2 in
    if is_eps e2r then
      if null e1r then e1r else (DUnion e1r e2r, True)
    else (DUnion e1r e2r, d)
  | DConc e1 e2 →
    let e1r = reduce_eps e1 in
    let e2r = reduce_eps e2 in
    if is_eps e1r then e2r
    else if is_eps e2r then e1r
         else (DConc e1r e2r, d)
  | DStar e1 →
    let e1r = reduce_eps e1 in
    if is_eps e1r then e1r else (DStar e1r, True)
  ]

and is_eps (e,_)= match e with [ DEps → True | _ → False ];
```

La fonction prend en entrée une expression de type *dregexp 'a bool* dont la décoration par un booléen d'annotation est ici celui désignant la présence du mot vide ϵ (calculé par *nullify*). La complexité de l'algorithme est linéaire en la taille de l'expression d'entrée. Pour l'expression régulière

$$(b + (1{\cdot}1 + 1)^*){\cdot}(1 + a^*)^*$$

l'ϵ-réduction produit l'expression

$$(b{+}1){\cdot}(1{+}a^*)^* \ .$$

2.4 Forme normale d'étoile

La présence d'étoiles dans une expression est la cause de l'apparition de cycles dans l'automate associé. Il est donc souhaitable d'éliminer le plus possible d'étoiles dans une expression. Par exemple, une réduction d'étoile simple serait $a^{**} \rightsquigarrow a^*$. Il existe d'autres cas plus complexes pour lesquels on peut éliminer des étoiles comme nous l'avons vu dans l'exemple 2 du chapitre I avec $(a^*b^\epsilon)^* \rightsquigarrow (a + b^\epsilon)^*$. Une partie des étoiles superflues se trouvent éliminées lorsqu'on met une expression sous une forme normale particulière qu'on appelle *forme normale d'étoile*. La forme normale d'étoile (*star-normal form* en anglais) a été introduite par Brüggemann-Klein en 1993 [BK93] et peut être calculée en temps linéaire en la taille de l'expression. Nous allons rappeller la propriété fondamentale que doit vérifier la forme normale d'étoile et présenter un algorithme pour la calculer.

Soit E une expression linéaire et L son langage associé, on définit $First(E)$ comme l'ensemble des premiers symboles des mots de L, on définit $Last(E)$ comme l'ensemble des derniers symboles des mots de L. Pour tout symbole a, on définit $Follow(E, a)$ comme l'ensemble des symboles qui suivent directement a dans un mot de L.

Définition 11 (Berry-Sethi 1986, [BS86] Définition 4.1)**.**

$$First(E) = \{\ a \mid \exists v,\ av \in L(E)\ \}$$

$$Last(E) = \{\ a \mid \exists v,\ va \in L(E)\ \}$$

$$Follow(E, a) = \{\ b \mid \exists u\ v,\ uabv \in L(E)\ \}$$

À l'aide de ces trois ensembles on peut définir la propriété fondamentale que doit vérifier une expression qui est en forme normale d'étoile :

Définition 12. Soit E une expression linéaire, E est en *forme normale d'étoile* si et seulement si elle satisfait la proposition suivante : toute sous-expression F^* de E est telle que pour tout $a \in Last(F)$, les ensembles $First(F)$ et $Follow(F,a)$ sont disjoints :

$$\forall a \in Last(F), First(F) \cap Follow(F,a) = \emptyset.$$

On dit qu'une expression régulière E (pas nécessairement linéaire) est en *forme normale d'étoile* si et seulement si l'expression linéaire associée à E est en forme normale d'étoile.

Par exemple l'expression $(a^*b^\epsilon)^*$ n'est pas en forme normale d'étoile parce que

$$First(a^*b^\epsilon) \cap Follow(a^*b^\epsilon, a) = \{a,b\} \cap \{a,b\} = \{a,b\} \neq \emptyset.$$

La forme normale d'étoile peut être obtenue par l'algorithme suivant :

```
value snf : dregexp 'a bool → dregexp 'a bool;
value snf e =
  let rec outside (e,_) =
    match e with
    [ DEps → Eps
    | DSymb s → Symb s
    | DUnion e1 e2 → Union (outside e1) (outside e2)
    | DConc e1 e2 → Conc (outside e1) (outside e2)
    | DStar e1 → Star (inside e1)
    ]
  and inside (e,_) =
    match e with
    [ DEps → Eps
    | DSymb s → Symb s
    | DUnion e1 e2 → Union (inside e1) (inside e2)
    | DConc e1 e2 →
      match (null e1, null e2) with
      [ (True, True) → Union (inside e1) (inside e2) (* Conc becomes Union *)
      | (True, False) → Conc (outside e1) (inside e2)
      | (False, True) → Conc (inside e1) (outside e2)
      | (False, False) → Conc (outside e1) (outside e2)
      ]
    | DStar e1 → inside e1 (* erases a star *)
    ] in
  nullify (outside e);
```

L'algorithme est composé de deux fonctions mutuellement récursives qui ont pour but d'effacer les étoiles emboîtées ($E^{***} \rightsquigarrow E^*$) en remplaçant les concaténations par des unions et en utilisant la règle suivante $(E+F^*)^* \rightsquigarrow (E+F)^*$. La fonction *outside* parcourt récursivement le terme et lorsqu'elle rencontre l'étoile d'une expression alors elle passe la main à la seconde fonction *inside* qui a pour but d'effacer si possible les étoiles redondantes parce que les sous-expressions sont déjà à l'intérieur d'une étoile.

Par exemple, l'expression régulière $(a^*{\cdot}(b^*{\cdot}c+1){\cdot}b^*)^*$ se réduit en forme normale d'étoile en $(a+((b^*{\cdot}c+1)+b))^*$.

L'implémentation que nous proposons de l'algorithme est fidèle aux définitions récursives de l'algorithme original [BK93]. Cependant la version originale réduirait une expression $(a+1)^*$ en a^* alors que notre algorithme la laisse intacte. Cette réduction n'est pas indispensable pour obtenir la propriété attendue de la forme normale d'étoile. On aurait donc pu simplifier le programme de la fonction *inside* concernant la concaténation, mais cette simplification serait erronée dans le cas où on manipulerait une +-algèbre (voir chapitre I section 2.4) plutôt qu'une algèbre de Kleene.

2.5 Expression régulière normalisée

On dira qu'une expression régulière est *normalisée* si elle est associative à gauche par rapport à la concaténation, ϵ-réduite et aussi en forme normale d'étoile. Pour effectuer cette normalisation on applique successivement les simplifications étudiées précédemment parce qu'elles préservent les propriétés déjà vérifiées. La partie difficile de la preuve serait d'établir que la mise en forme normale d'étoile préserve l'ϵ-réduction ; pour cela il faudrait faire une analyse par cas sur la structure de l'expression régulière.

```
value normalize : regexp 'a → dregexp 'a bool;
value normalize e =
  let e1 = la_conc_nf e in
  let e2 = nullify e1 in
  let e3 = reduce_eps e2 in
  let e4 = snf e3 in
  e4;
```

3 Dériver les expressions régulières

La dérivation d'expressions régulières est une technique pour synthétiser des automates. Intuitivement, une dérivée d'une expression E par rapport à un mot w est une expression E' telle que la concaténation de w et E' dénote un sous-langage de l'expression de départ E. Si le nombre de dérivées par rapport à l'ensemble des mots est fini alors ces dérivées définissent les états d'un automate qui reconnaît le langage associé à l'expression régulière.

Les dérivées se calculent de manière formelle par récurrence sur l'expression régulière. Il existe plusieurs variantes de dérivées qui permettent d'obtenir des automates différents. Brzozowski fut le premier en 1964 [Brz64] à introduire la notion de dérivée d'expression régulière pour synthétiser un automate déterministe. Cette idée originale fut le point de départ de nombreuses extensions qui donnent des algorithmes synthétisant des automates non-déterministes plus ou moins compacts.

Dans un premier temps nous rappellerons la définition des dérivées de Brzozowski afin de présenter l'idée générale des algorithmes de synthèse d'automates. Puis nous présenterons les dérivées linéaires qui permettent de définir l'automate de positions, ensuite nous présenterons les dérivées partielles qui permettent de définir l'automate d'équations et pour finir nous présenterons les dérivées canoniques qui donnent le bon cadre unificateur pour retrouver les automates précédents.

3.1 Dérivées de Brzozowski et méthode générale

Nous avons vu dans le chapitre I, la définition de la dérivée d'un langage par rapport à un autre langage (qu'on note $L_1 \setminus L_2$). Brzozowski a étudié le calcul effectif de la restriction des dérivées par rapport aux mots sur langages réguliers, afin d'en obtenir un algorithme pour

synthétiser des automates finis déterministes. Nous rappelons dans cette partie les résultats principaux de Brzozowski [Brz64].

On définit la fonction λ qui associe à toute expression E l'expression 1 si $L(E)$ contient le mot vide ϵ et 0 sinon :

$$\lambda(1) = 1$$
$$\lambda(0) = 0$$
$$\lambda(a) = 0$$
$$\lambda(E + F) = \lambda(E) + \lambda(F)$$
$$\lambda(E \cdot F) = \lambda(E) \cdot \lambda(F)$$
$$\lambda(E^*) = 1$$

Pour toute expression E, en utilisant l'ϵ-réduction ainsi qu'une opération analogue pour réduire les 0, on simplifie $\lambda(E)$ vers 1 ou 0. De manière analogue aux fonctions *nullify* et *null*, la fonction λ calcule l'appartenance du mot vide ϵ.

On rappelle la définition de dérivée de langage par rapport à un mot :

Définition 13 ([Brz64], Definition 3.1)**.** Soit L un ensemble de mot et soit w un mot, la *dérivée* de L par rapport à w est $w^{-1}L = \{w' \mid ww' \in L\} = w^{-1}L(E)$.

Par extension, on peut définir la dérivée sur les expressions régulières en faisant référence aux langages associés ; c'est-à-dire qu'on notera $L(w^{-1}E) = \{w' \mid ww' \in L(E)\}$.

Théorème III.2 ([Brz64], Theorem 3.1)**.** Pour toute expression régulière E, pour tout mot a de longueur 1, la dérivée de E en a se calcule récursivement de la manière suivante :

$$a^{-1}a = 1$$
$$a^{-1}b = 0, \text{avec } b \neq a$$
$$a^{-1}(E + F) = a^{-1}E + a^{-1}F$$
$$a^{-1}(E \cdot F) = (a^{-1}E) \cdot F + \lambda(E) \cdot (a^{-1}F)$$
$$a^{-1}(E^*) = (a^{-1}E) \cdot E^*$$

Théorème III.3 ([Brz64], Theorem 3.2)**.** La dérivée de E par rapport à un mot $w = a_1 \cdots a_n$ se calcule par récurrence sur la structure du mot w, en utilisant la dérivée par rapport à un symbole :

$$(a_1 \cdots a_n)^{-1}E = (a_2 \cdots a_n)^{-1}(a_1^{-1}E),$$
$$\epsilon^{-1}E = E.$$

Les théorèmes de Brzozowski permettent de calculer effectivement la dérivée d'un langage par rapport à un mot, mais pas la dérivée par rapport à un langage, ce qui reviendrait à calculer $F^{-1}E = \{w^{-1}E \mid w \in F\}$ et dans ce cas $F^{-1}E = F \setminus E$. Néanmoins, les résultats précédents sont suffisants pour en déduire un algorithme de synthèse d'automate en considérant tout particulièrement le théorème qui suit :

Théorème III.4 (Brzozowski 1964, Théorème 4.3 [Brz64])**.** Pour toute expression régulière E, l'ensemble des dérivées $w^{-1}E$, pour tout $w \in \Sigma^*$, est fini (modulo associativité commutativité et idempotence de l'opérateur $+$).

Une expression régulière E peut se décomposer en une somme d'expressions en utilisant les dérivées ([Brz64], Theorem 4.4) : $E = \lambda(E) + \sum_{a \in \Sigma} a(a^{-1}E)$. Puisque l'ensemble des dérivées est fini, on en déduit que cette décomposition s'applique aussi à chacune des dérivées de E ([Brz64], Theorem 4.5). On en déduit une construction d'automate fini de la manière suivante : on associe à chaque dérivée un état de l'automate, et les transitions de l'automate sont issues des décompositions.

Définition 14 (automate de Brzozowski)**.** Pour toute expression régulière E, l'automate de Brzozowski $(Q, \Sigma, \delta, i, F)$ est défini par :
- $Q = \{\ w^{-1}E \mid w \in \Sigma^*\ \}$
- $i = E$
- $F = \{\ q \mid \lambda(q) = 1\ \}$
- $\delta(q, a) = a^{-1}q,\ \forall q \in Q$ et $\forall a \in \Sigma$

Dans cette définition, puisque l'ensemble des états est l'ensemble des dérivées, on se permet de les noter q, comme par exemple dans la définition des états acceptants F : "$\lambda(q) = 1$".

Méthode générale. Cette construction qu'on appelle algorithme de Brzozowski est la méthode générale à appliquer pour synthétiser des automates finis à partir d'expressions régulières. Il s'agit d'utiliser une notion de dérivée, puis de prouver que l'ensemble des dérivées est fini (théorème III.4), et on finit par en déduire une construction d'automate (définition 14). Nous appliquerons cette méthode pour trois notions de dérivées : les dérivées linéaires, les dérivées partielles et les dérivées canoniques.

L'algèbre des expressions régulières présentée ici ne comprend que l'opérateur booléen +, les autres opérateurs booléens tels que l'intersection et le complément sont omis alors qu'ils sont aussi bien présents dans les travaux de Brzozowski. Les expressions contenant les opérateurs de l'algèbre booléenne sont celles qu'il faut pour des logiciels tels que les analyseurs lexicaux (lex). L'article de Owens *et al.* [ORT09] décrit les techniques d'implémentation en ML de l'algorithme de Brzozowski. Les auteurs utilisent avec succès leur implémentation pour compiler de nombreux analyseurs lexicaux définis dans des vraies applications. Ce travail fait revivre cette idée de 1964 qui a été jusque-là peu promue dans les applications pratiques.

3.2 Dérivées d'expressions régulières linéaires

On considère ici des expressions régulières *linéaires*. Les expressions régulières linéaires ont des propriétés intéressantes vis-à-vis de la dérivée. En effet Berry et Sethi [BS86] ont montré que l'ensemble des dérivées est fini (modulo associativité, commutativité et idempotence de +), et le nombre de dérivées est inférieur ou égal à la largeur de l'expression régulière (nombre d'occurrences de symboles). On en déduit un algorithme qui construit un automate avec un nombre d'états égal au nombre de symboles apparaissant dans l'expression régulière, plus un état pour l'état initial.

Nous rappelons qu'une expression linéarisée est une expression dans laquelle chaque symbole n'apparaît qu'une fois. Nous avons présenté un algorithme pour linéariser une expression, la fonction *linearize* en section 2.1.

Théorème III.5 (Berry-Sethi, Theorem 3.4 [BS86])**.** Pour toute expression régulière linéaire E, pour tout symbole a de E, les dérivées $(wa)^{-1}E$ sont nulles ou uniques modulo associativité, commutativité et idempotence de l'opérateur +.

Démonstration. La preuve se fait par récurrence sur la structure de l'expression régulière. Soit E une expression régulière linéaire. Si E est 0 ou 1 alors toutes les dérivées valent 0. Sinon si E est un symbole a alors toutes les dérivées valent 0 ou 1. Pour les cas restants on fait la preuve par récurrence sur la structure de E.

1. $E = E_1 + E_2$. Par définition des dérivées on a

$$(wa)^{-1}(E_1 + E_2) = (wa)^{-1}E_1 + (wa)^{-1}E_2.$$

   Puisque tous les symboles de E sont différents alors a est soit dans E_1 soit dans E_2. Si a est dans E_1 alors $(wa)^{-1}E_2 = 0$; sinon si a est dans E_2 alors $(wa)^{-1}E_1 = 0$. On conclut en utilisant l'hypothèse de récurrence sur E_1 ou E_2 suivant le cas.

2. $E = E_1 \cdot E_2$. Une preuve par récurrence sur la taille de w établit l'égalité suivante :

$$(wa)^{-1}(E_1 \cdot E_2) = ((wa)^{-1}E_1) \cdot E_2 + \sum_{wa=uva} \lambda(u^{-1}E_1) \cdot ((va)^{-1}E_2).$$

   Si a est dans E_1 alors dans la partie droite de l'équation, la grande somme est équivalente à 0 et seulement le premier terme peut éventuellement être non nul. Dans ce cas on utilise l'hypothèse de récurrence sur $(wa)^{-1}(E_1) \cdot E_2$ et donc $(wa)^{-1}(E_1 \cdot E_2) = F \cdot E_2$ pour une expression F fixée. Sinon, a est dans E_2 et dans ce cas le premier terme est nul. Rappellons que $\lambda(u^{-1}E_1)$ vaut 0 ou 1, pour tout u. En utilisant l'hypothèse de récurrence on a que tous les sous-termes de la forme $(va)^{-1}E_2$ valent 0 ou une expression F fixée, donc leur somme vaut soit 0 soit F.

3. $E = E^*$. On utilise le lemme auxiliaire suivant (lemme 3.3 de [BS86]) : Soit a un symbole, pour tout mot w, $(wa^{-1})(E^*)$ est équivalent à une somme de sous-termes de l'ensemble $\{(va^{-1}E \cdot E^*) \mid wa = uva\}$. La preuve se fait par récurrence sur la longueur de w.
   En utilisant ce lemme, on a que $(wa^{-1})E^*$ est équivalent à une somme d'expressions de $\{(va^{-1}E \cdot E^*) \mid wa = uva\}$. En utilisant l'hypothèse de récurrence on a que toutes les expressions de la forme $(va)^{-1}E$ valent 0 ou une expression F fixée, donc leur somme vaut soit 0 soit F. Par conséquent $(wa^{-1})(E^*)$ est équivalent à $F \cdot E^*$ si F ne vaut pas 0.

□

Ce théorème conduit à définir la notion de *continuation* d'une expression linéaire E par rapport à un symbole a, notée $C_a(E)$:

Définition 15 (Berry-Sethi, Definition 3.5 [BS86])**.** Pour toute expression régulière linéaire E, pour tout symbole a dans E, une *continuation en a* de E est définie par une expression $(wa)^{-1}E$ non nulle. Par récursion structurelle sur E, une telle expression doit exister et par le théorème précédent, de telles expressions sont toutes équivalentes. On peut donc parler de *la* continuation en a de E quelle que soit l'expression de cette classe d'équivalence, on la note C_a.

On rappelle qu'on note $Pos(E)$ l'ensemble des positions d'une expression régulière linéarisée E. Le théorème de Berry et Sethi permet de faire correspondre les continuations et les positions. On obtient un automate non-déterministe en utilisant les positions. On l'appelle l'*automate de positions* parce qu'on associe un état de l'automate à chaque position de symbole de l'expression linéaire de départ.

Définition 16 (automate de positions)**.** Pour toute expression régulière linéaire E, l'*automate de positions* $(Q, \Sigma, \delta, i, F)$ reconnaissant le langage associé à E est défini de la manière suivante :
- $Q = Pos(E) \cup \{0\}$
- $i = 0$

- $F = \{ x \mid \lambda(C_x) = 1 \}$
- $\delta(x, a) = \{ y \mid h(y) = a \wedge y^{-1}(C_x) = C_y \}$

On rappelle que la fonction h est la fonction qui délinéarise une expression, on se permet de l'utiliser sur les symboles d'une expression linéaire pour définir la fonction de transitions δ.

Nous venons de voir les notions qui permettent de définir l'automate de positions. Dans la suite, nous donnerons une implémentation de l'algorithme qui ne calcule jamais les continuations et dérivées pour construire l'automate, mais utilise plutôt le calcul des ensembles *First* et *Follow*.

3.3 Dérivées partielles

Antimirov [Ant96] a introduit la notion de *dérivées partielles* en 1996 afin de proposer une nouvelle technique de synthèse d'automates non-déterministes. Les dérivées partielles définissent maintenant des **ensembles** de dérivées. Historiquement, Mirkin proposa en 1966 des techniques similaires [Mir66] mais ce travail reste peu référencé. Cependant Brzozowski [Brz71] émit un commentaire très favorable dans un rapport sur l'article de Mirkin :

> *[...] The notion of a non-deterministic automaton is a very natural one when one is dealing with sets of equations for defining languages, and that the algorithm for constructing automata from expressions is considerably more efficient in the non-deterministic case. J. A. Brzozowski.*

Pour une expression régulière, les dérivées partielles sont en nombre fini et surtout plus petit que les dérivées d'une expression linéaire. On en déduit donc un automate plus compact encore que celui obtenu par les dérivées linéaires. Les dérivées partielles peuvent être vues comme des équations entre langages [Mir66] ; pour cette raison on appelle l'automate produit à partir des dérivées partielles l'*automate d'équations* [CZ01, CZ02].

Définition 17 (Dérivées partielles d'une expression régulière)**.**

$$\partial_a a = \{1\}$$

$$\partial_a b = \emptyset$$

$$\partial_a(E + F) = \partial_a E \cup \partial_a F$$

$$\partial_a(E \cdot F) = (\partial_a E) \odot F \cup \lambda(E) \odot (\partial_a F)$$

$$\partial_a(E^*) = (\partial_a E) \odot \{E^*\}$$

avec l'opération $\odot$ qui est définie comme l'extension naturelle de la concaténation sur des ensembles d'expressions régulières.

Définition 18. On définit la notion de dérivée partielle par rapport à un mot $w = a_1 \cdots a_n$ en utilisant la dérivée partielle par rapport à un symbole de la manière suivante :

$$\partial_{a_1 \cdots a_n} E = \partial_{a_2 \cdots a_n}(\partial_{a_1} E),$$

$$\partial_\epsilon E = E.$$

Le théorème principal concernant la technique des dérivées est aussi valable pour le cas des dérivées partielles :

Théorème III.6 (Antimirov 1996)**.** Pour toute expression régulière E, l'ensemble des dérivées partielles de E est fini modulo associativité, commutativité et idempotence de l'opérateur $+$, et la taille de cet ensemble est inférieure ou égale à $||E|| + 1$.

On en déduit un algorithme, suivant la méthode générale, produisant un automate plus compact que celui de Berry-Sethi exploitant cette fois la notion de dérivées partielles. On notera $\mathcal{PD}$ l'ensemble des dérivées partielles de E. On obtient un automate non-déterministe qu'on appelle *automate d'équations*.

Définition 19 (automate d'équations)**.** Pour toute expression régulière E, l'*automate d'équations* $(Q, \Sigma, \delta, i, F)$ reconnaissant le langage associé à E est défini de la manière suivante :
- $Q = \bigcup_{a \in \Sigma} \partial_a E \quad \cup \{E\}$
- $i = E$
- $F = \{q \mid \lambda(q) = 1\}$
- $\delta(q, a) = \partial_a(q), \ \forall a \in \Sigma$ et $\forall q \in Q$

L'automate d'équations est considéré comme l'automate le plus compact qu'on sache construire en utilisant les dérivées. Nous allons voir une dernière notion de dérivée, qui permet de retrouver les définitions de l'automate de positions et de l'automate d'équations, et qui permet aussi de les comparer les uns aux autres. Cette notion de dérivée jouera aussi un rôle important dans l'implémentation des algorithmes.

3.4 Dérivées canoniques

Les notions de *dérivées canoniques* et de *continuations canoniques* (c-continuations) ont été introduites par Champarnaud et Ziadi [CZ01, CZ02]. Cela fournit un cadre commun pour décrire les synthèses d'automates non-déterministes tels que l'automate de positions, l'automate de continuations (qui correspond à l'automate follow introduit récemment par Ilie et Yu [IY03]) et l'automate d'équations. On parle de dérivées canoniques parce qu'elles fournissent un représentant canonique des dérivées linéaires.

On peut construire directement les continuations d'une expression, notées $c_x(E)$ pour une expression régulière *linéaire* E, de la manière suivante :

Définition 20 (continuations canoniques)**.** La continuation canonique d'une expression régulière linéaire E par rapport à un symbole a de E est définie par :

$$c_a(a) = 1$$

$$c_a(b) = 0, \text{avec } b \neq a$$

$$c_a(E + F) = \text{si } c_a(E) \neq 0 \text{ alors } c_a(E) \text{ sinon } c_a(F)$$

$$c_a(E \cdot F) = \text{si } c_a(E) \neq 0 \text{ alors } c_a(E) \cdot F \text{ sinon } c_a(F)$$

$$c_a(E^*) = \text{si } c_a(E) \neq 0 \text{ alors } c_a(E) \cdot E^* \text{ sinon } 0$$

Dans cette définition, la linéarité de l'expression est nécessaire pour qu'il y ait adéquation avec la définition de continuation de la définition 15.

Il existe principalement trois constructions permettant de synthétiser des NFA. Les automates produits ne sont pas égaux, mais on peut les comparer car ce sont des automates quotients les uns des autres. Toujours en utilisant la méthode générale, à partir des continuations canoniques, qu'on appelle aussi *c-continuations*, on peut retrouver les trois automates issus des algorithmes suivants :

1. automate de positions, algorithme de Berry-Sethi
2. automate de continuations, algorithme d'Ilie-Yu
3. automate d'équations, algorithme d'Antimirov

On considère pour les trois définitions qui vont suivre que l'expression régulière E est linéarisée et on rappelle que la fonction h effectue la délinéarisation (efface les entiers introduits par la linéarisation). L'automate de positions peut être défini à l'aide des c-continuations.

Définition 21 (automate de positions)**.** L'automate de positions $(Q, \Sigma, \delta, i, F)$ d'une expression régulière E est défini par :
- $Q = \{(x, c_x(E)) \mid x \in Pos(E)\} \cup \{(0, E)\}$
- $i = (0, E)$
- $F = \{(x, c_x(E)) \mid \lambda(c_x(E)) = 1\}$
- $\delta((x, c_x(E)), a) = \{(y, c_y(E)) \mid h(y) = a \wedge y^{-1}(c_x(E)) \equiv c_y(E)\}$

On définit l'*automate de continuations* en utilisant les c-continuations.

Définition 22 (automate de continuations)**.** L'automate de continuations $(Q, \Sigma, \delta, i, F)$ d'une expression régulière E est défini par :
- $Q = \{c_x(E) \mid x \in Pos(E)\} \cup \{E\}$
- $i = E$
- $F = \{q \in Q \mid \lambda(q) = 1\}$
- $\delta(q, a) = \{c_y(E) \mid h(y) = a \wedge y^{-1}(q) \equiv c_y(E)\}$

On peut retrouver l'automate d'équations en délinéarisant les c-continuations à l'aide de la fonction h.

Définition 23 (automate d'équations)**.** L'automate d'équations $(Q, \Sigma, \delta, i, F)$ d'une expression régulière E est défini par :
- $Q = \{h(c_x(E)) \mid x \in Pos(E)\} \cup \{E\}$
- $i = E$
- $F = \{q \in Q \mid \lambda(q) = 1\}$
- $\delta(q, a) = \{h(c_y(E)) \mid h(y) = a \wedge y^{-1}(q) \equiv h(c_y(E))\}$

À renommage des états près, les c-continuations permettent de redéfinir l'automate de positions et l'automate d'équations.

Sur l'expression $(a{\cdot}(b{\cdot}(a^*{\cdot}c + d)^* + e) + d{\cdot}(a^*{\cdot}c + d)^*)$ de l'exemple 3, la linéarisation produit l'expression $(a_1(b_2(a_3{}^*c_4 + d_5)^* + e_6) + d_7(a_8{}^*c_9 + d_{10})^*)$. Nous allons donner toutes les continuations canoniques pour cette expression. On notera c_2 au lieu de c_{b_2} pour plus de lisibilité (il faudra donc faire attention à c_4 qui fait référence à c_{c_4} dans la partie gauche des égalités mais pas dans la partie droite) :

$$\begin{array}{rcl}
c_1 & = & b_2(a_3{}^*c_4 + d_5)^* + d_5 \\
c_2 & = & (a_3{}^*c_4 + d_5)^* \\
c_3 & = & a_3{}^*c_4(a_3{}^*c_4 + d_5)^* \\
c_4 & = & (a_3{}^*c_4 + d_5)^* \\
c_5 & = & (a_3{}^*c_4 + d_5)^* \\
c_6 & = & 1 \\
c_7 & = & (a_8{}^*c_9 + d_{10})^* \\
c_8 & = & a_8{}^*c_9(a_8{}^*c_9 + d_{10})^* \\
c_9 & = & (a_8{}^*c_9 + d_{10})^* \\
c_{10} & = & (a_8{}^*c_9 + d_{10})^*
\end{array}$$

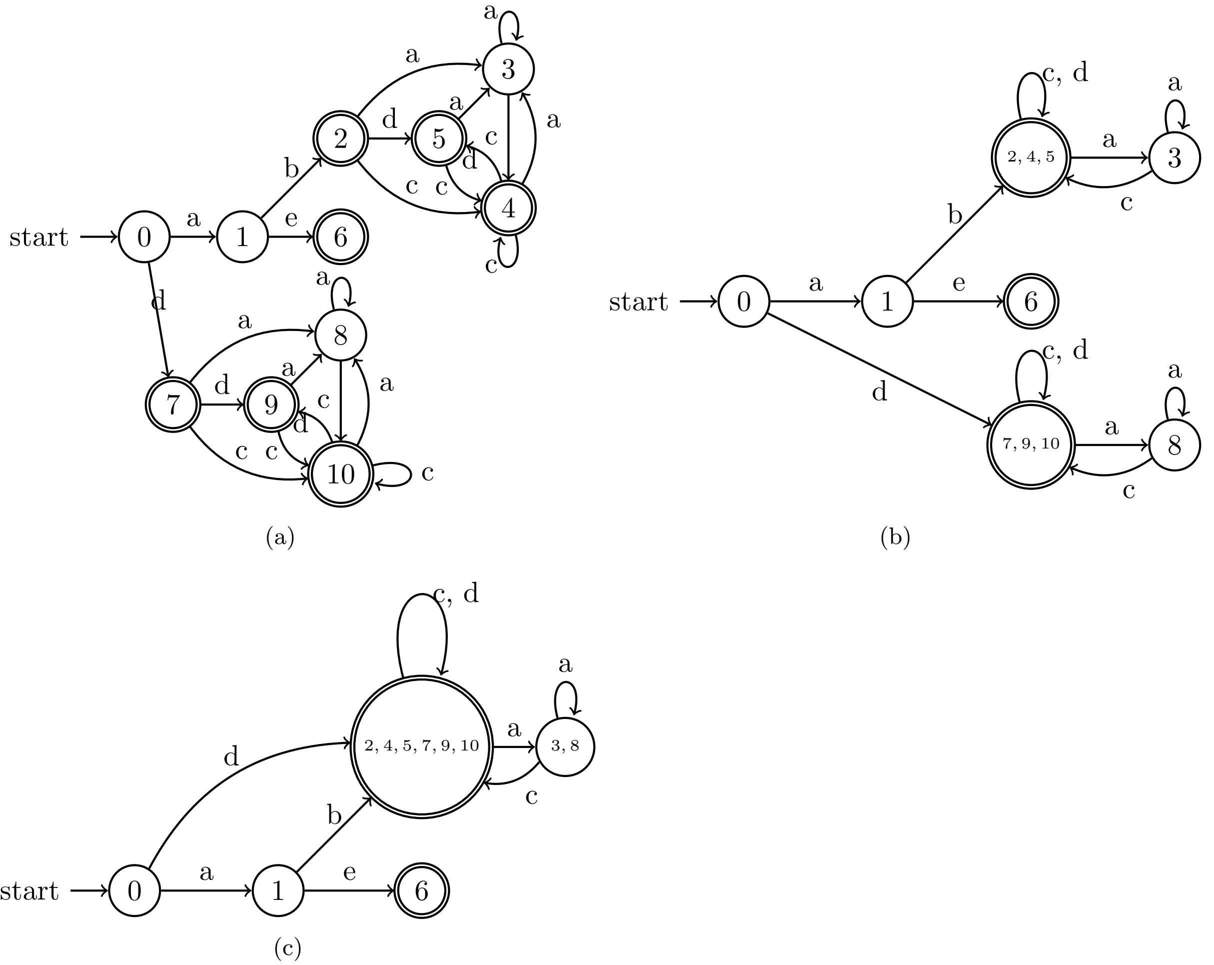


FIG. 2 – Automate de positions 2(a), automate de continuations 2(b) et automate d'équations 2(c) pour l'expression $a(b(a^*c+d)^*+e)+d(a^*c+d)^*$.

Pour la synthèse de l'automate de positions toutes ces continuations correspondent à des états différents. Pour la synthèse de l'automate de continuations, deux continuations identiques correspondent à un seul et même état, c'est-à-dire :

$$c_2 = c_4 = c_5,$$

$$c_7 = c_9 = c_{10}.$$

La synthèse de l'automate d'équations identifient les continuations qui sont identiques après délinéarisation, c'est-à-dire qu'on identifie les états suivants :

$$h(c_2) = h(c_4) = h(c_5) = h(c_7) = h(c_9) = h(c_{10}) = (a^*c+d)^*$$

et

$$h(c_3) = h(c_8) = a^*c(a^*c+d)^*.$$

Pour récapituler, l'automate de positions est donné en figure 2(a), l'automate de continuations est donné en figure 2(b) et l'automate d'équations est donné en figure 2(c).

Théorème III.7. Pour toute expression régulière, l'automate de continuations est un quotient de l'automate de positions, et l'automate d'équations est un quotient de l'automate de continuations. On en déduit donc la comparaison suivante sur la taille des automates en fonction du nombre d'états et de transitions :

$$\text{automate de positions} \geq \text{automate de continuations} \geq \text{automate d'équations}$$

Démonstration. Les quotients sont facilement explicités par les définitions des ensembles d'états, en utilisant la technique décrite dans la proposition 13 de [CZ02]. On définit une première relation $\sim_1$ par

$$(x, c_x) \sim_1 (y, c_y) \Leftrightarrow c_x = c_y$$

et on vérifie que la relation $\sim_1$ est invariante à droite, c'est-à-dire que pour tout $a \in \Sigma$ on a $(x, c_x) \sim_1 (y, c_y) \Rightarrow \delta((x, c_x), a) \sim_1 \delta((y, c_y), a)$[2]. Cela montre que l'automate de continuations est un quotient de l'automate de positions. Ensuite on introduit la relation $\sim_2$ par

$$c_x \sim_2 c_y \Leftrightarrow h(c_x) = h(c_y)$$

et on vérifie que la relation $\sim_2$ est invariante à droite, c'est-à-dire que pour tout $a \in \Sigma$ on a $c_x \sim_2 c_y \Rightarrow \delta(c_x, a) \sim_2 \delta(c_y, a)$. Cela montre que l'automate d'équations est un quotient de l'automate de continuations. □

Ilie et Yu [IY03] ont introduit l'*automate follow*. L'idée est de trouver des états équivalents dans l'automate de positions : Lorsque deux états ont les mêmes transitions et qu'ils sont tous les deux acceptants ou non, alors ils peuvent être considérés comme équivalents. L'automate obtenu à partir de l'automate de positions après cette réduction d'états est appelé *automate follow*.

Définition 24. L'automate follow est l'automate de positions dont les états sont fusionnés suivant l'équivalence $\equiv_f$ suivante : pour toutes positions x et y

$$x \equiv_f y \quad \text{si et seulement si} \quad \begin{array}{l} \mathit{Follow}(E, x) = \mathit{Follow}(E, y) \\ \text{et } x \in \mathit{Last}(E) \Leftrightarrow y \in \mathit{Last}(E) \end{array}$$

Ilie et Yu ont essayé de comparer en taille l'automate follow avec l'automate de positions et l'automate d'équations. L'automate follow est défini comme un quotient de l'automate de positions, il est donc nécessairement plus petit mais la comparaison avec l'automate d'équations est resté un problème ouvert. Le problème a été résolu par Champarnaud, Ouardi et Ziadi [COZ07] qui ont prouvé que pour toute expression normalisée l'automate d'équations était plus petit que l'automate follow. Ce résultat est assez technique parce que l'automate d'équations et l'automate follow ne sont pas de même nature : l'un est obtenu comme quotient de l'automate de positions et l'autre à partir de l'expression régulière (par les dérivées). Cependant le théorème 1 de Champarnaud, Nicart et Ziadi [CNZ06] permet d'identifier l'automate follow et l'automate de continuations dans le cas où l'expression est normalisée. Les définitions, présentées ici à base de c-continuations, permettent donc de comparer proprement l'automate de positions, l'automate de continuations et l'automate d'équations dans le théorème III.7. Nous allons voir que cette approche est très effective puisque nous donnons une implémentation en OCaml pour ces trois constructions.

[2] On rappelle que si $\sim$ est une relation d'équivalence sur un ensemble X, alors on définit une relation d'équivalence sur les sous-parties de X de la manière suivante : pour tout $Y, Z \subseteq X$, on a $Y \sim Z \Leftrightarrow Y_{/\sim} = Z_{/\sim}$.

4 Synthèse de l'automate de positions

L'algorithme de Berry-Sethi produit l'automate de positions. Historiquement les premières constructions ont été trouvées indépendamment par Glushkov [Glu61] d'une part, et par McNaughton et Yamada [MY60] d'autre part. D'après la définition de l'automate de positions, l'ensemble des états correspond à l'ensemble des symboles d'une expression régulière linéarisée plus un état initial. Le calcul de l'ensemble des états se fait en parcourant l'expression régulière et en collectant les indices des symboles linéarisés. Les fonctions *First* et *Last* sont essentielles pour la construction de la table de transitions ; on les implémente par de simples fonctions récursives sur l'expression régulière :

```
value first : dregexp 'a bool → list 'a → list 'a;
value rec first e l =
  let (e', d) = e in
  match e' with
  [ DEps → l
  | DSymb s → [s :: l ]
  | DUnion e1 e2 → first e1 ( first e2 l)
  | DConc e1 e2 → if null e1 then first e1 ( first e2 l) else first e1 l
  | DStar e1 → first e1 l
  ];

value last : dregexp 'a bool → list 'a → list 'a;
value rec last e l =
  let (e', d) = e in
  match e' with
  [ DEps → l
  | DSymb s → [s :: l ]
  | DUnion e1 e2 → last e1 ( last e2 l)
  | DConc e1 e2 → if null e2 then last e1 ( last e2 l) else last e2 l
  | DStar e1 → last e1 l
  ];
```

La liste l apparaît en argument supplémentaire des fonctions *first* et *last*. Ce paramètre est utilisé par les fonctions comme un accumulateur des symboles déjà collectés. Cette technique permet d'avoir une complexité linéaire en la taille de l'expression régulière, tout en utilisant de simples listes.

On peut construire efficacement l'automate de positions grâce à la proposition suivante qui relie le calcul des premiers éléments (*First*) d'une continuation au calcul des successeurs (*Follow*).

Proposition III.1 (Berry-Sethi, Proposition 4.2 [BS86])**.** Soit E une expression régulière linéaire, pour tout symbole a, si on considère $C_a(E)$ (la continuation en a de E), alors on a

$$First(C_a(E)) = Follow(E, a).$$

Le calcul de la table de transitions revient donc à construire l'ensemble des $Follow(E, a)$, pour tout symbole a. Le calcul de l'ensemble des paires $(a, Follow(E, a)))$ se fait récursivement en utilisant la proposition suivante :

Proposition III.2 (Berry-Sethi, Proposition 4.3 [BS86])**.** Soit E une expression régulière linéaire. La fonction $\mathbf{F}$, définie par les règles suivantes, est telle que $\mathbf{F}(E, \{\})$ calcule l'ensemble des paires

$(a, \mathit{Follow}(E, a)))$. Les règles sont :

$$
\begin{array}{rcl}
\mathbf{F}(a, S) & = & \{(a, S)\}, \\
\mathbf{F}(1, S) & = & \emptyset, \\
\mathbf{F}(E_1 + E_2, S) & = & \mathbf{F}(E_1, S) \cup \mathbf{F}(E_2, S), \\
\mathbf{F}(E_1 \cdot E_2, S) & = & \left\{ \begin{array}{ll} \mathbf{F}(E_1, \mathit{First}(E_2) \cup S) \cup \mathbf{F}(E_2, S) & \text{si } \lambda(E_2) = 1, \\ \mathbf{F}(E_1, \mathit{First}(E_2)) \cup \mathbf{F}(E_2, S) & \text{sinon.} \end{array} \right. \\
\mathbf{F}({E_1}^*, S) & = & \mathbf{F}(E_1, \mathit{First}(E_1) \cup S).
\end{array}
$$

La fonction suivante implémente ce calcul et produit une liste qui correspond à la liste de paires qui correspond à la table de transitions complète de l'automate.

```
value follow_bs : dregexp ('a * 'b) bool → list ('b * list ('a * 'b));

value follow_bs e =
  let rec aux e l fol =
    let (e', _) = e in
    match e' with
    [ DEps → fol
    | DSymb (s,i) → [ (i, l) :: fol ]
    | DUnion e1 e2 → aux e1 l (aux e2 l fol)
    | DConc e1 e2 →
        let fol2 = aux e2 l fol in
        let l1 = if null e2 then first e2 l else first e2 [] in
        aux e1 l1 fol2
    | DStar e1 → let l_res = first e1 l in
        aux e1 l_res fol
    ] in
  aux e [] [];
```

On remarquera cependant que les listes construites par la fonction *follow_bs* peuvent contenir des doublons. On surpasse cette difficulté en donnant en entrée de la fonction une expression régulière qui a été préalablement mise en *forme normale d'étoile* (star-normal form), et de cette manière on garantit qu'il ne peut y avoir de doublons.

L'algorithme de Berry-Sethi est le résultat d'une succession de passes de compilation : une expression régulière est d'abord linéarisée, puis décorée avec des booléens indiquant la présence du mot vide ϵ, puis mise en forme normale d'étoile et enfin on fait appel aux fonctions, *follow_bs*, *first* et *last* pour synthétiser la table de transitions avec l'état initial et la liste des états acceptants.

```
value berry_sethi : regexp 'a → automaton int 'a;

value berry_sethi e =
  let e1 = linearize e in
  let e2 = nullify e1 in
  let e3 = snf e2 in
  let initial = 0 (* the initial state *) in
  let acceptings = let l = List.map snd (last e3 []) in
    if null e3 then [ initial :: l ] else l in
  let transitions = [ (initial , first e3 []) :: follow_bs e3 ] in
  ( initial , transitions , acceptings);
```

Sur l'expression régulière de l'exemple 3 l'algorithme synthétise l'automate de la figure 2(a).

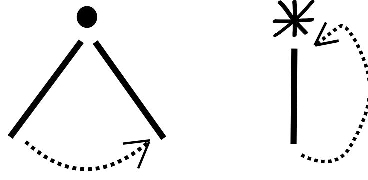


FIG. 3 – Définition des liens follow pour la concaténation et l'étoile.

5 Synthèse de l'automate de continuations

Nous allons synthétiser l'automate de continuations selon un algorithme similaire à l'algorithme de Berry-Sethi. La principale différence est qu'il faut préalablement identifier les positions qui sont équivalentes parce qu'elles correspondent à des c-continuations identiques. Pour calculer efficacement ces classes d'équivalence on utilise le théorème 2 de Champarnaud, Nicart et Ziadi [CNZ06] qui démontre que les c-continuations associées à deux positions sont identiques si et seulement si les *liens follow* les plus bas le sont aussi.

Rappelons brièvement ce qu'est un lien follow. Pour une expression régulière donnée, un lien follow lie deux sous-expressions. Le lien est orienté avec une origine et une destination. On crée des liens follow dans les situations suivantes :

- Lorsque l'expression est une concaténation alors on crée un lien de la sous-expression gauche vers la sous-expression droite.
- Lorsque l'expression est une étoile alors on crée un lien de la sous-expression vers l'expression étoilée de départ.

La figure 3 récapitule les liens avec des flèches en ligne pointillée. Une flèche pointe vers la destination.

Nous allons voir comment on peut calculer les liens follow pour toute une expression régulière. L'originalité de notre démarche est de considérer des liens de manière symbolique. Pour cela il faut des identifiants uniques associés à chaque nœud de l'expression. Lorsqu'il y a un lien follow de x vers y alors on annote le nœud x par l'identifiant de y. On utilisera la structure d'expression décorée pour effectuer ce calcul. Les identifiants sont des entiers et nous devons d'abord numéroter de manière unique tous les nœuds d'une expression.

```
value stamp : dregexp 'a bool → dregexp 'a (bool * int);
value stamp e =
  let rec aux e i =
    let (e',b) = e in
    let cons e = (e, (b,i)) in
    match e' with
    [ DEps → (cons DEps, i+1)
    | DSymb s → (cons (DSymb s), i+1)
    | DUnion e1 e2 →
      let (e1',i1) = aux e1 (i+1) in
      let (e2',i2) = aux e2 i1 in
      (cons (DUnion e1' e2'), i2)
    | DConc e1 e2 →
```

```
      let (e1', i1) = aux e1 (i+1) in
      let (e2', i2) = aux e2 i1 in
      (cons (DConc e1' e2'), i2)
    | DStar e1 → let (e1', i') = aux e1 (i+1) in (cons (DStar e1'), i')
    ] in
  let (e', _) = aux e 2 in
  e';
```

La numérotation commence à partir de *2* parce qu'on réserve l'état *0* pour l'état initial et l'état *1* pour un état acceptant. On peut récupérer l'entier annoté au nœud de tête en temps constant :

```
value pos : dregexp 'a (bool * int) → int;
value pos (e,(_, i)) = i;
```

Sur l'exemple 3 la fonction *stamp* produit l'expression annotée suivante :

$$(a_4 \cdot_3 (b_7 \cdot_6 (a_{12}{}^{*11} \cdot_{10} c_{13} +_9 d_{14})^{*8} +_5 e_{15}) +_2 d_{17} \cdot_{16} (a_{22}{}^{*21} \cdot_{20} c_{23} +_{19} d_{24})^{*18}) \ .$$

On peut calculer les liens follow les plus bas de toute une expression régulière en annotant celle-ci de manière récursive en temps linéaire et en une seule passe :

```
value follow_link : dregexp 'a (bool * int) → dregexp 'a ((bool * int) * int);
value follow_link e =
  let rec link_rec e k =
    let (e', annot) = e in
    let cons e = (e, (annot,k)) in
    match e' with
    [ DEps → cons DEps
    | DSymb s → cons (DSymb s)
    | DUnion e1 e2 → cons (DUnion (link_rec e1 k) ( link_rec e2 k))
    | DConc e1 e2 → let p2 = pos e2 in
      cons (DConc (link_rec e1 p2) ( link_rec e2 k))
    | DStar e1 → let (b,p) = annot in cons (DStar ( link_rec e1 p))
    ] in
  link_rec e 1 (* 1 is the default accepting state *);
```

Lorsqu'un nœud n'a pas de tel successeur on l'annote avec la valeur *1* réservée préalablement. Les liens follow sont propagés à travers les nœuds *Union*. On récupère donc à chaque nœud symbole de l'expression le lien follow le plus bas. Le calcul des liens follow sur l'expression de l'exemple 3 produit le résultat suivant :

$$(a_{4\to 5} \cdot (b_{7\to 8} \cdot (a_{12\to 11}{}^{*} \cdot c_{13\to 8} + d_{14\to 8})^{*} + e_{15\to 1}) + d_{17\to 18} \cdot (a_{22\to 21}{}^{*} \cdot c_{23\to 18} + d_{24\to 18})^{*}) \ .$$

Nous ne montrons ces liens que pour les nœuds symboles pour ne pas surcharger l'expression d'annotations.

Certains symboles peuvent avoir le même lien follow, il faut donc choisir un symbole unique comme représentant de ce lien. On choisit celui le plus à gauche en parcourant l'expression régulière. On place un booléen supplémentaire aux nœuds symboles s'ils sont choisis pour être représentants d'une classe d'équivalence. Ces booléens peuvent être calculés en temps linéaire pour toute l'expression et on effectue ce marquage en deux passes. Tout d'abord on doit indiquer si l'expression a accès à un symbole directement à droite sans traverser d'étoile puis on parcourt l'expression en profondeur d'abord et de gauche à droite pour choisir le symbole. Chaque symbole marqué sera le représentant unique de la classe d'équivalence des états de l'automate. La fonction suivante effectue cette première passe :

```
value is_direct : dregexp 'a ('b * bool) → bool;
value direct_symb : dregexp 'a 'b → dregexp 'a ('b * bool);
value is_direct e = let (e', (_,b)) = e in b;
value direct_symb e =
  let rec aux e =
    let (e', annot) = e in
    let cons e b = (e, (annot, b)) in
    match e' with
    [ DEps → cons DEps False
    | DSymb s → cons (DSymb s) True
    | DUnion e1 e2 →
      let e1' = aux e1 in
      let e2' = aux e2 in
      let b1 = is_direct e1' in
      let b2 = is_direct e2' in
      cons (DUnion e1' e2') (b1 || b2)
    | DConc e1 e2 →
      let e1' = aux e1 in
      let e2' = aux e2 in
      cons (DConc e1' e2') ( is_direct e2')
    | DStar e1 →
      let e1' = aux e1 in
      cons (DStar e1') False
    ] in
  aux e;
```

On effectue le marquage des symboles avec un booléen qui indique si le symbole est choisi comme représentant de la classe d'équivalence. La fonction *choose_states* effectue cette tâche et a comme invariant qu'il ne peut y avoir plus d'un symbole choisi comme représentant d'une même c-continuation :

```
value choose_states :
  dregexp 'a ((('b * 'c) * int) * bool) →
  dregexp ('a * (bool * int)) ((('b * 'c) * int) * bool);
value choose_states e =
  let rec aux e mark =
    let (e', info) = e in
    let (((_,_), k),_) = info in
    let cons e = (e, info) in
    match e' with
    [ DEps → cons DEps
    | DSymb s → cons (DSymb (s,(mark,k)))
    | DUnion e1 e2 →
      if is_direct e1
      then cons (DUnion (aux e1 mark) (aux e2 False))
      else cons (DUnion (aux e1 False) (aux e2 mark))
    | DConc e1 e2 →
      let e1' = aux e1 True in
      match e2 with
      [ (DStar e3, info3) →
        if is_direct e1
```

```
        then cons (DConc e1' (DStar (aux e3 False), info3))
        else cons (DConc e1' (DStar (aux e3 True), info3))
      | _ → cons (DConc e1' (aux e2 mark))
      ]
    | DStar e1 → cons (DStar (aux e1 True))
    ] in
  aux e True;
```

Sur l'exemple 3, le calcul du choix des états est le suivant :

$$(a_{\underline{5}}\cdot(b_{\underline{8}}\cdot(a_{\underline{11}}{}^{*}\cdot c_{8}+d_{8})^{*}+e_{\underline{1}})+d_{\underline{18}}\cdot(a_{\underline{21}}{}^{*}\cdot c_{18}+d_{18})^{*})\ .$$

On redéfinit les fonctions usuelles null, first, last et follow qui s'appliquent à des expressions régulières décorées adéquates :

```
value null_f e = let (_, (((b,_),_),_)) = e in b;

value rec first_f (e,_) l =
  match e with
  [ DEps → l
  | DSymb (s,(_,k)) → [ (s,k) :: l ]
  | DUnion e1 e2 → first_f e1 ( first_f e2 l)
  | DConc e1 e2 → if null_f e1 then first_f e1 ( first_f e2 l) else first_f e1 l
  | DStar e1 → first_f e1 l
  ];

value rec last_f (e,_) l =
  match e with
  [ DEps → l
  | DSymb (_,(b,k)) → if b then [ k :: l ] else l
  | DUnion e1 e2 → last_f e1 ( last_f e2 l)
  | DConc e1 e2 → if null_f e2 then last_f e1 ( last_f e2 l) else last_f e2 l
  | DStar e1 → last_f e1 l
  ];

value follow_f e =
  let rec aux (e,_) l fol =
    match e with
    [ DEps → fol
    | DSymb (s,(b,k)) → if b then [ (k,l) :: fol ] else fol
    | DUnion e1 e2 → aux e1 l (aux e2 l fol)
    | DConc e1 e2 →
      let fol2 = aux e2 l fol in
      let l1 = if null_f e2 then first_f e2 l else first_f e2 [] in
      aux e1 l1 fol2
    | DStar e1 → let l_res = first_f e1 l in
      aux e1 l_res fol
    ] in
  aux e [] [];
```

Maintenant on peut définir l'algorithme qui construit l'automate de continuations à partir d'une expression régulière. Nous donnons le nom de *ilie_yu* à l'algorithme en référence aux auteurs

Ilie et Yu parce que ce sont les premiers qui ont étudié l'automate follow ; cependant l'algorithme efficace qui est présenté ici suit de très près la démarche de Champarnaud, Nicart et Ziadi [CNZ06].

```
value ilie_yu : regexp 'a → automaton int 'a;
value ilie_yu e =
  let e1 = normalize e in
  let e2 = stamp e1 in
  let e3 = follow_link e2 in
  let e4 = direct_symb e3 in
  let e5 = choose_states e4 in
  let initial = 0 (* initial state *) in
  let acceptings = let l = last_f e5 [] in
    if null_f e5 then [ initial :: l ] else l in
  let transitions = [ ( initial , first_f e5 []) :: follow_f e5 ] in
  ( initial , transitions , acceptings);
```

Sur l'exemple 3 l'algorithme synthétise l'automate de la figure 2(b). Cet algorithme normalise l'expression préalablement. Il a été montré [CNZ06] que lorsque l'expression est normalisée alors l'algorithme produit l'automate follow d'Ilie et Yu [IY03] qui est égal à l'automate de continuations. Remarquons que si l'expression n'est pas normalisée alors l'automate construit est l'automate de continuations[3] tel que nous l'avons défini (définition 22).

Automate follow - automate de continuations. Nous rappelons le théorème qui permet d'identifier l'automate follow et l'automate de continuations dans le cas où l'expression est normalisée :

Théorème III.8 (Theorem 1 and Theorem 2 [CNZ06])**.** Pour toute expression normalisée E et toutes positions x et y de E, les conditions suivantes sont équivalentes :
- $x \equiv_f y$
- $\Delta_E(x) = \Delta_E(y)$
- $D_E(x) = D_E(y)$

Dans ce théorème, $\Delta_E(x)$ désigne l'ensemble des liens follow (ce qui revient à la c-continuation c_x en x), et $D_E(x)$ est le lien follow le plus bas en x. L'ensemble des $D_E(x)$ se calcule en temps linéaire sur toute l'expression par la fonction *follow_link* , et le choix des états se calcule aussi en temps linéaire par les fonctions *direct_symb* et *choose_states* , on en déduit donc que le calcul de l'automate de continuations est quadratique comme le calcul de l'automate de positions.

6 Synthèse de l'automate d'équations

Un premier algorithme pour produire l'automate d'équations est décrit par Antimirov [Ant96]. Cet algorithme a une complexité $O(||E||^3|E|^2)$ dans le pire cas, et elle a été grandement améliorée par Champarnaud et Ziadi [CZ01, CZ02] qui ont présenté un algorithme de complexité quadratique. Dans l'implémentation que nous présentons, nous utilisons principalement les travaux de Champarnaud-Ziadi en nous aidant d'une implémentation de la technique de hash-consing dûe à Conchon et Filliâtre [FC06]. En particulier pour implémenter cet algorithme il faut

[3] à quelques détails près : il faut quand même que l'expression soit ϵ-réduite. Un contre-exemple est le résultat sur l'expression $a1 + b1$; dans ce cas, la transition étiquetée par a et celle étiquetée par b n'arrivent pas sur le même état. Cependant, nous avons remarqué que si l'expression n'est pas ϵ-réduite l'implémentation produit tout de même un automate correct...

tester l'appartenance d'un élément dans une liste efficacement. L'algorithme de Champarnaud-Ziadi [CZ01, CZ02] utilise à l'origine l'algorithme de tri lexicographique de Paige-Tarjan [PT87] qui est adapté lorsqu'on considère les expressions comme des chaînes de caractères. Ici nous utilisons une représentation de données plus algébrique, pour laquelle la méthode de hash-consing [FC06] est très adaptée et décrite de manière générique. Nous utiliserons à deux reprises la méthode de hash-consing.

Les tables de hachage que nous utilisons ont une taille fixée. On choisit couramment une table qui a la taille d'un nombre premier pour avoir une meilleure répartition des clefs de hachage. Nous choisissons ici le sixième nombre de Mersenne premier. Pour l'usage que nous ferons des tables de hachage cela conviendra, toutefois on aurait pu choisir un autre nombre premier :

```
value size = 131071;
```

On commence par définir une fonction qui partage une expression régulière de manière maximale et annote les nœuds avec des entiers uniques, cela permet de tester l'égalité de deux expressions en temps constant. Pour ce faire on utilise une table de hachage.

```
value share_expression : dregexp 'a bool → dregexp 'a (bool * int);
value share_expression e =
  let memo =Array.create size [] in
  let i = ref 0 in
  let search e = List.find (fun x → eq_exp x e) in
  let share element info key =
    let bucket = memo.(key) in
    let fake_deco = (info,0) in
    try (search (element,fake_deco) bucket, key) with
    [ Not_found → let res = (element, (info, i.val)) in
      do{ i.val := i.val + 1; memo.(key) := [ res :: bucket ]; (res,key) }
    ] in

  let hash0 = 1 in
  let hash1 = 42 in
  let hash_union k1 k2 = (k1 + (19 * k2)) mod size in
  let hash_conc k1 k2 = (k2 + (47 * k1)) mod size in
  let hash_star k = (k + 39) mod size in
  let rec aux (e,b) =
    match e with
    [ DEps → share DEps b hash0
    | DSymb s → share (DSymb s) b hash1
    | DUnion e1 e2 →
      let (e1', k1) = aux e1 in
      let (e2', k2) = aux e2 in
      let key = hash_union k1 k2 in
      share (DUnion e1' e2') b key
    | DConc e1 e2 →
      let (e1', k1) = aux e1 in
      let (e2', k2) = aux e2 in
      let key = hash_conc k1 k2 in
      share (DConc e1' e2') b key
    | DStar e1 →
```

```
    let (e1', k1) = aux e1 in
    let key = hash_star k1 in
    share (DStar e1') b key
  ] in
fst (aux e);
```

Un table de hachage est créée dans la valeur *memo*. Les fonctions *search* et *share* servent chercher un élément dans la table et ajouter un élément dans la table lorsqu'il n'y est pas déjà. Les constantes et fonctions *hash0*, *hash1*, *hash_union*, *hash_conc* et *hash_star* servent à calculer les clés de hachage. La fonction auxiliaire *aux* parcourt l'expression en consultant la table de hachage à chaque nœud pour recopier ou créer la sous-expression correspondante.

Sur l'exemple 3 la fonction de partage produit l'expression annotée

$$(a_0 \cdot_{11} (b_1 \cdot_8 (a_0{}^{*2} \cdot_4 c_3 +_6 d_5)^{*7} +_{10} e_9) +_{13} d_5 \cdot_{12} (a_0{}^{*2} \cdot_4 c_3 +_6 d_5)^{*7}) \ .$$

Ensuite on définit une fonction qui construit les c-continuations récursivement et les clés de hachage associées. Une c-continuation est encodée comme une liste d'expressions pour laquelle la clé de hachage est fonction de la liste des identifiants des expressions de la liste.

```
value all_c_continuations :
  dregexp 'a (bool * int) →
  dregexp ('a * (list int * int)) (bool * int);
value all_c_continuations e =
  let hash0 = 1 in
  let hash key p = ( (53 * key) + p ) mod size in
  let rec aux e cc = (* cc is a c−continuation *)
    let (e', annot) = e in
    let cons e = (e, annot) in
    let addcc e =
      let (cc', k) = cc in
      let p = pos e in
      ([ p :: cc' ], hash k p) in
    match e' with
    [ DEps →          cons DEps
    | DSymb s →       cons (DSymb (s,cc))
    | DUnion e1 e2 → cons (DUnion (aux e1 cc) (aux e2 cc))
    | DConc e1 e2 → cons (DConc (aux e1 (addcc e2)) (aux e2 cc))
    | DStar e1 →      cons (DStar (aux e1 (addcc e)))
    ] in
  aux e ([], hash0);
```

Le calcul des c-continuations sur l'exemple 3 produit l'expression annotée

$$(a_{[10]} \cdot (b_{[7]} \cdot (a_{[2,3,7]}{}^{*} \cdot c_{[7]} + d_{[7]})^{*} + e_{[]}) + d_{[7]} \cdot (a_{[2,3,7]}{}^{*} \cdot c_{[7]} + d_{[7]})^{*}) \ .$$

Jusqu'à présent on a seulement calculé les c-continuations et les clés de hachage associées. Il faut maintenant identifier celles qui sont égales et choisir un représentant pour chacune des classes d'équivalence. Chaque symbole est donc annoté par l'identifiant de la classe d'équivalence de c-continuations, avec un booléen supplémentaire indiquant s'il est choisi comme le représentant.

```
value compute_states :
  dregexp ('a * (list int * int)) (bool * int) →
  dregexp ('a * (bool * int))      (bool * int);
```

```
value compute_states e =
  let memo =Array.create  size  [] in
  let i = ref 1 in (* states begin at 1 *)
  let search l = List.find (fun (l1, j) →  l1 = l) in
  let share l key =
    let bucket = memo.(key) in
    try ((search l bucket), False) with
    [ Not_found → let res = (l, i.val) in
      do{ i.val := i.val + 1; memo.(key) := [ res :: bucket ]; (res,True)}
    ] in

  let rec aux e =
    let (e', annot) = e in
    let cons e  = (e, annot) in
    match e' with
    [ DEps → cons DEps
    | DSymb (s,(cc,k)) → match share cc k with
      [ ((l,j), True) → cons (DSymb (s,(True, j)))
      | ((l,j), False)→  cons (DSymb (s,(False, j)))
      ]
    | DUnion e1 e2 →
      let e1' = aux e1 in
      let e2' = aux e2 in
      cons (DUnion e1' e2')
    | DConc e1 e2 →
      let e1' = aux e1 in
      let e2' = aux e2 in
      cons (DConc e1' e2')
    | DStar e1 →
      let e1' = aux e1 in
      cons (DStar e1')
    ] in
  aux e;
```

Sur l'exemple 3 le calcul des états produit l'expression suivante, dans laquelle on indice chaque symbole par un entier identifiant la c-continuation, et on souligne lorsque c'est le représentant de la classe de c-continuations :

$$(a_{\underline{1}}\cdot(b_{\underline{2}}\cdot(a_{\underline{3}}{}^{*}\cdot c_2+d_2)^{*}+e_{\underline{4}})+d_2\cdot(a_3{}^{*}\cdot c_2+d_2)^{*}) \ .$$

Nous ne redéfinirons par les fonctions null, first, last et follow spécifiques à la synthèse de l'automate d'équations, elles sont très similaires à celles pour synthétiser l'automate de continuations. Maintenant on peut définir l'algorithme qui construit l'automate d'équations. Nous donnons le nom de *antimirov* à l'algorithme en référence à l'inventeur des dérivées partielles [Ant96] ; cependant l'algorithme efficace présenté ici suit d'assez près les travaux de Champarnaud et Ziadi [CZ01, CZ02]. On remarquera que les c-continuations sont représentées sous forme de liste d'expressions au lieu d'expressions. Il est important que l'expression soit préalablement mise sous forme de concaténation à gauche pour que l'identification des c-continuations soit correcte. En effet, sur l'expression $a(bc) + (ab)c$ le calcul des états n'identifie pas comme égales les deux c-continuations en a. On normalise l'expression pour cette raison, mais aussi pour être dans les mêmes conditions que pour la synthèse de l'automate follow.

```
value antimirov : regexp 'a → automaton int 'a;
value antimirov e =
  let e1 = normalize e in
  let e2 = share_expression e1 in
  let e3 = all_c_continuations e2 in
  let e4 = compute_states e3 in
  let initial = 0 in
  let acceptings = let l = last_e e4 [] in
    if null_e e4 then [ initial :: l ] else l in
  let transitions = [ ( initial , first_e e4 []) :: (follow_e e4) ] in
  ( initial , transitions , acceptings);
```

Sur l'expression régulière de l'exemple 3 l'algorithme synthétise l'automate de la figure 2(c).

7 Performances des algorithmes

Nous proposons ici de comparer le temps de calcul et la taille de l'automate des quatre algorithmes de synthèse que nous venons de présenter : automate de Thompson, automate de positions, automate de continuations et automate d'équations. Notre banc d'essais prend en entrée des expressions régulières aléatoires de tailles variées. On considère les expressions régulières dont la taille est comprise entre 2 et 40000 ; pour chaque taille on engendre un certain nombre d'expressions régulières pour lesquelles on exécute les algorithmes, puis on rapporte les performances en divisant par le nombre d'expressions testées.

7.1 Temps de calcul des algorithmes

La figure 4 rend compte des temps d'exécution des différents algorithmes pour construire les automates de Thompson, de positions, de continuations et d'équations, en fonction de la taille de l'expression régulière (nombre de nœuds). On remarque qu'il y a un ordre dans ces temps d'exécution pour les différents algorithmes. L'algorithme de Thompson est le plus rapide, suivi de l'algorithme de Berry-Sethi puis de l'algorithme d'Ilie-Yu et pour finir celui d'Antimirov. Les temps de calcul rendent compte du fait que les algorithmes travaillent plus ou moins fort à identifier des états qui sont équivalents.

On peut mettre en balance la performance des algorithmes avec la difficulté à les implémenter. L'algorithme de Thompson s'implémente assez directement. L'algorithme de Berry-Sethi nécessite la maîtrise de techniques différentes dans l'implémentation mais aussi dans la preuve de correction. Ces difficultés vont croissantes pour l'algorithme d'Ilie-Yu et celui d'Antimirov.

7.2 Nombre de transitions-états de l'automate

La figure 5 rend compte de la taille des automates de Thompson, de positions, de continuations et d'équations en fonction de la taille de l'expression régulière. Ici la taille de l'automate est la somme du nombre d'états et du nombre de transitions. Cette mesure rend compte de la taille de l'automate telle que nous l'avons spécifiée. Il faut remarquer que le nombre de transitions-états pour l'automate de Thompson est si petit comparé aux autres algorithmes que la courbe se retrouve collée à l'axe des abscisses. Cependant il ne faut pas tirer de conclusions trop hâtives de supériorité, l'automate de Thompson n'est pas vraiment comparable aux autres puisqu'il construit un ϵ-NFA.

Comparons maintenant les algorithmes produisant des NFA. On dégage de ces mesures que l'automate de continuations et l'automate d'équations sont comparables, avec un léger avantage

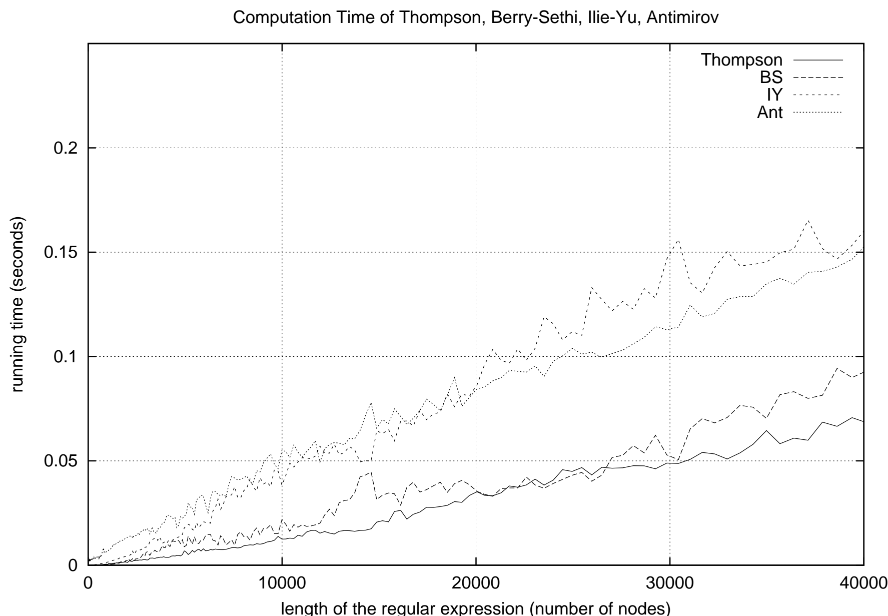


FIG. 4 – Temps d'exécution des algorithmes de synthèse d'automates en fonction de la taille de l'expression d'entrée

pour l'automate d'équations. En revanche ils sont tous les deux beaucoup plus compacts que l'automate de positions.

Lorsque l'automate est représenté sous forme matricielle il est plus approprié de considérer la taille de l'automate comme le nombre d'états seulement et non plus comme fonction du nombre de transitions.

7.3 Nombre d'états de l'automate

La figure 6 rend compte de la taille des automates de Thompson, de positions, de continuations et d'équations en fonction de la taille de l'expression régulière. Ici la taille de l'automate est le nombre d'états. Cette mesure pour la taille de l'automate est adaptée dans le cas où la représentation est matricielle. On vérifie bien que l'automate le plus petit est l'automate d'équations puis l'automate de continuations puis l'automate de positions et pour finir l'automate de Thompson.

Contrairement à la mesure précédente, le nombre d'états de l'automate de Thompson est beaucoup plus grand. Mais rappelons qu'il est inapproprié de comparer cet automate aux autres puisque c'est un ϵ-NFA alors que les autres sont des NFA.

On peut en déduire les mêmes conclusions que celle de la figure 5, à savoir que les automates de continuations et d'équations sont comparables avec un léger avantage pour l'automate d'équations. L'automate de positions est vraiment moins compact que les deux précédents.

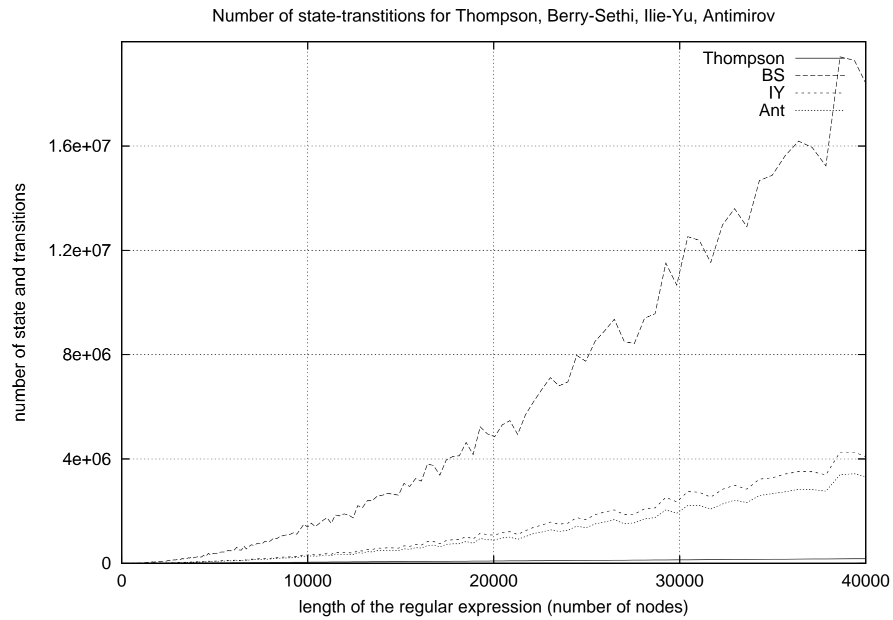


FIG. 5 – Nombre de transitions-états de l'automate produit en fonction de la taille de l'expression d'entrée

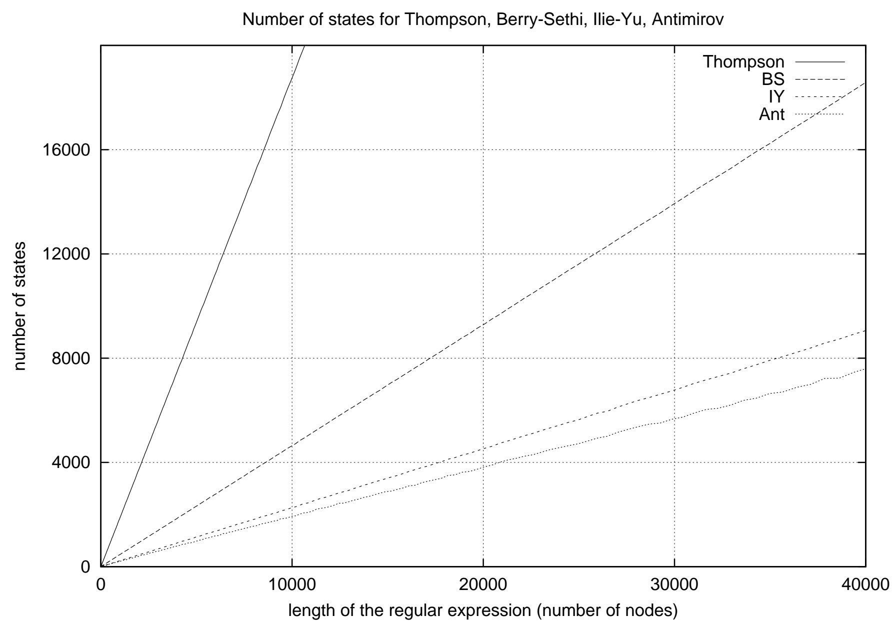


FIG. 6 – Nombre d'états de l'automate produit en fonction de la taille de l'expression d'entrée

7.4 Valider l'implémentation

Nous avons validé les algorithmes présentés en utilisant le *théorème d'égalité* d'Eilenberg (Theorem 8.1 du chapitre *IV* de [Eil74]). Ce théorème indique que deux NFA $\mathcal{A}_1$ et $\mathcal{A}_2$ de taille n_1 et n_2 sont équivalents si et seulement si pour tout mot w de longueur plus petit que $n_1 + n_2$ alors w est reconnu par $L(\mathcal{A}_1)$ si et seulement si w est reconnu par $L(\mathcal{A}_2)$. Pour tester l'équivalence de deux automates il suffit donc de tester leur reconnaissance sur l'ensemble fini de mots plus petits que la somme des tailles des automates. On rappelle qu'on peut utiliser le moteur réactif des machines d'Eilenberg (voir chapitre II) pour résoudre le problème de reconnaissance d'un automate. Les NFA synthétisés sont des machines d'Eilenberg finies, on utilise donc le moteur réactif pour ce type de machines, moteur dont la correction a été prouvée formellement [Raz08b]. On valide les algorithmes de synthèse (pour des automates pas trop gros) de la manière suivante : on choisit par exemple $\mathcal{A}_1$ comme étant l'automate de positions, pour lequel on est relativement sûr de l'implémentation, et on choisit pour $\mathcal{A}_2$ l'automate de continuations ou l'automate d'équations. On vérifie que les simulations des machines d'Eilenberg associées à $\mathcal{A}_1$ et $\mathcal{A}_2$ sont bien identiques pour tous les mots de longueurs plus petites que $n_1 + n_2$ et on en déduit l'équivalence des automates synthétisés par les deux algorithmes grâce au théorème d'égalité d'Eilenberg.

De plus, le théorème d'Eilenberg sous sa forme originale rend compte des multiplicités. C'est-à-dire qu'un mot est reconnu k fois s'il y a k chemins dans l'automate pour le reconnaître. Rien n'indique que les algorithmes présentés rendent compte des multiplicités mais puisque l'implémentation utilise une table de transitions représentée avec des listes, ces dernières peuvent contenir des doublons et donc présenter des multiplicités dans les transitions. On a remarqué, grâce au moteur réactif qui énumére toutes les solutions, que les algorithmes implémentés préservent les multiplicités de l'expression régulière. On notera que des extensions des algorithmes de synthèse pour rendre compte des multiplicités ont déjà fait l'objet de recherches [CLOZ04, LS05, AM06].

8 Remarques

La validation des algorithmes avec le théorème d'égalité d'Eilenberg est déjà satisfaisante pour vérifier que l'implémentation est correcte. Pour être complètement sûr de la correction des algorithmes il est courant de faire de la preuve formelle à l'aide d'un assistant de preuves (par exemple Coq). Pour cela, dans un premier temps il est nécessaire d'avoir une implémentation fonctionnelle sur laquelle on effectue une spécification formelle ; dans un second temps on effectue des preuves formelles sur cette spécification. Puisque nous avons déjà présenté nos algorithmes dans un fragment purement fonctionnel de OCaml (sauf pour l'automate d'équations), c'est un donc un premier pas vers la formalisation complète de ces algorithmes efficaces. On note aussi que la technique d'extraction de Coq permettrait de retrouver les algorithmes en OCaml tels qu'ils sont présentés.

L'automate d'équations est plus compact que l'automate de continuations et lui-même est plus compact que l'automate de positions. Cependant, ce gain de taille de l'automate d'équations s'explique par le fait que l'algorithme prend en compte un partage implicite parce qu'il ne peut être explicité dans le langage des expressions régulières. En imaginant une algèbre explicitant le partage dans les termes, on pourrait concevoir une extension de l'algorithme produisant l'automate de continuations qui exploite le partage. Dans le cas d'une expression partagée au maximum, cet algorithme hypothétique produirait l'automate d'équations.

Extension aux Multiplicités. On a remarqué que les algorithmes présentés dans cette thèse s'adapteraient très rapidement à une algèbre d'expressions rationnelles (avec multiplicités). En effet lorsqu'on valide nos algorithmes en utilisant le théorème d'égalité on s'aperçoit qu'ils conservent les multiplicités des expressions régulières. On pourrait donc étendre ces algorithmes à une algèbre d'expressions avec multiplicités. Des travaux ont déjà été proposés dans ce domaine [CLOZ04, LS05, AM06] avec des extensions des différents algorithmes pour des algèbres d'expressions rationnelles.

Une approche alternative à celle présentée dans cette thèse est celle proposée par Allauzen et Mohri [AM06]. Leur technique consiste à partir de l'algorithme de Thompson et à effectuer des opérations sur le graphe pour obtenir aussi bien l'automate de positions que l'automate follow ou bien encore l'automate d'équations. Ils ont aussi montré que leur approche s'étendait aux multiplicités. Cette approche est différente de la nôtre ; nous exploitons bien plus la structure de l'expression en travaillant le plus possible sur celle-ci et en produisant l'automate seulement dans une ultime passe de compilation. L'algorithmique de graphe est plus difficilement implémentable de manière efficace dans un langage purement fonctionnel à cause de la nature cyclique des automates. En revanche la programmation fonctionnelle est très adaptée à la manipulation de structures de données algébriques, c'est pour cette raison qu'on a préféré exploiter l'approche algébrique des algorithmes de synthèse d'automates à partir d'expressions régulières plutôt qu'une approche davantage basée sur une algorithmique de graphes.

9 Conclusion

Dans ce chapitre nous avons proposé une implémentation fonctionnelle d'algorithmes compilant les expressions régulières en automates. Bien que le problème se pose depuis plus de 50 ans, les développements récents ont montré que la problématique est encore vivante, tant sur le plan théorique que pratique, avec la technique des dérivées. Cette technique permet de développer des algorithmes efficaces qui sont dans le pire cas quadratiques en fonction de la taille de l'expression régulière d'entrée.

Nous avons proposé l'implémentation de quatre algorithmes de compilation d'expressions régulières en automates. Le premier est l'algorithme de Thompson [Tho68], un algorithme bien connu et très enseigné, qui produit un ϵ-NFA. Les trois autres algorithmes sont issus de travaux plus récents qui synthétisent des NFA, respectivement l'automate de positions, l'automate de continuations et l'automate d'équations. L'implémentation dans un langage de programmation unique nous a permis de faire une étude comparative de ces quatre algorithmes. Les conclusions de cette étude sont multiples. On ne peut élire un algorithme meilleur parmi les quatre, cela dépend de l'automate qu'on veut en sortie ou de la vitesse d'exécution de l'algorithme. Si la présence de transitions spontanées n'est pas prohibée alors l'algorithme de Thompson est le plus concis et obtenu le plus rapidement comparé aux trois autres algorithmes qui synthétisent des NFA. Si on souhaite absolument produire un NFA plutôt qu'un ϵ-NFA alors il faut comparer l'automate de positions, l'automate de continuations et l'automate d'équations. L'automate de positions est obtenu à partir de l'algorithme le plus simple mais ce n'est pas l'automate le plus compact. L'automate d'équations est obtenu par l'algorithme le plus compliqué mais il produit l'automate le plus compact. L'automate de continuations semble un bon compromis car son implémentation est à peine plus compliquée que celle de l'automate de positions, et il est toujours plus compact que ce dernier mais un peu plus gros que l'automate d'équations. Si l'expression possède beaucoup de sous-termes partagés alors c'est l'automate d'équations qui sera significativement le plus compact.

On pourrait exploiter les implémentations proposées ici dans le but de prouver formellement

ces algorithmes de synthèse d'automates. En effet, ces impléméntations étant présentées dans un langage de programmation fonctionnelle, cela constitue un avantage en vue d'un développement dans un assistant de preuves formelles tel que Coq.

Conclusion

Nous avons voulu montrer que le modèle de calcul des machines d'Eilenberg était effectif en plus d'avoir des vertus théoriques. Nous rappelons que ce travail a débuté en visant des applications à la linguistique computationnelle, où certains problèmes d'analyse sont décrits par un algorithme non-déterministe composé de plusieurs niveaux d'automates [HR06]. Les techniques d'automates sont cruciales au traitement de toutes les couches basses de la langue naturelle : phonologie, calculs morphologiques, représentation efficace des lexiques, segmentation, étiquetage, syntaxe superficielle. Il existe de nombreuses bibliothèques d'algorithmes pour effectuer ces traitements ; parmi celles qui sont développées avec un langage de programmation fonctionnelle il y a Zen [Hue05] et Grammatical Framework [Ran04], développés respectivement en OCaml et en Haskell. Nous nous sommes intéressés à l'étude du modèle des machines d'Eilenberg effectives qui permet de décrire des calculs non-déterministes de manière modulaire (généralisant l'article [HR06]). Pour appuyer cette thèse nous avons décomposé notre étude en trois chapitres regroupant chacun des contributions ouvrant la voie à de nouvelles perspectives de recherches.

Le chapitre I rappelle les notions fondamentales de la théorie des automates qui sont à la base des machines d'Eilenberg. Mais aussi, dans ce chapitre nous avons voulu insister sur la question de l'axiomatisation de l'égalité des expressions régulières. Ces problèmes d'axiomatisation apportent de nombreux résultats théoriques importants qui méritent d'être soulignés davantage. Une telle axiomatisation décrit très précisément la logique avec laquelle on peut prouver l'égalité de deux expressions régulières. La première contribution est de présenter les principales axiomatisations qui ont été obtenues progressivement tout au long des cinquante dernières années.

Toutefois, les axiomatisations ne sont pas correctes si on prend en compte la multiplicité des mots, c'est-à-dire le nombre de fois qu'un mot est dénoté par une expression. Cette extension s'étudie algébriquement à l'aide des séries formelles et plus particulièrement ici avec les séries rationnelles. Cette extension des expressions régulières aux multiplicités est en fait très générale, elle permet en particulier de rendre compte avec plus de finesse du non-déterminisme mais surtout elle permet l'extension aux probabilités qui sont cruciales en linguistique computationnelle. Les axiomatisations que nous avons présentées ne rendent pas compte de ces multiplicités, il serait donc intéressant d'étudier dans quelle mesure on peut les modifier à cet effet.

La deuxième contribution de cette thèse est l'étude d'une version effective des machines d'Eilenberg qui constitue le chapitre II. À l'origine le modèle des X-machines d'Eilenberg a été conçu pour modéliser des calculs définis par une composante de contrôle avec un automate fini, et une composante de données à l'aide de relations binaires sur un ensemble arbitraire X. La généralité du modèle permet d'une part de rendre compte de tous les modèles utilisant des automates d'états finis pour le contrôle (automates sur les mots, transducteurs, automates à pile, machines de Turing, *etc*) et d'autre part de définir des calculs comme plusieurs niveaux de machines (modularité). La contribution est d'étudier une version effective du modèle des

machines d'Eilenberg, et pour cela on précise les notions à l'aide de programmes qui permettent de simuler ces machines d'Eilenberg effectives. Pour nous attaquer à ce problème il d'abord été utile de restreindre le modèle à ce que nous appelons les *machines d'Eilenberg finies* [Raz08a], c'est-à-dire les machines qui ont une propriété de finitude sur les relations et sur les calculs. La simulation se trouve implémentée par un moteur réactif qui est décrit de manière purement applicative et pour lequel nous avons exhibé une preuve de correction très rigoureuse en nous appuyant sur un développement formel similaire dans l'assistant de preuves Coq [Raz08b]. Nous avons montré que le simulateur permettait d'effectuer une analyse complète d'un langage hors-contexte ambigu, ce qui illustre le fait que le modèle n'est pas cantonné à traiter uniquement des problèmes réguliers. En nous inspirant du moteur réactif pour les machines d'Eilenberg finies nous avons défini deux moteurs paramétrés par des stratégies qui permettent de simuler des machines d'Eilenberg variées (pas seulement finies). En particulier nous présentons des stratégies pour simuler les machines finies mais aussi pour simuler des machines d'Eilenberg sans aucune restriction.

Parmi les pistes de recherche envisageables il serait utile de considérer le problème du partage dans la recherche de solutions de ces machines. En effet les simulateurs que nous avons présentés sont purement applicatifs et effectuent des calculs de manière redondante. Certaines solutions sont le résultat de calculs issus de sous-calculs qui pourraient être partagés. Le défi est ici de bien comprendre où placer l'interface de partage alors que la composante de données est abstraite. Le simulateur verrait son efficacité améliorée d'un facteur exponentiel dans certains cas. Nous posons aussi comme problème ouvert la formalisation mathématique des moteurs munis de stratégies. Ce travail devrait faire se dégager une notion d'équité dans la gestion des points de choix. La formalisation fournie pour les machines d'Eilenberg finies est sans aucun doute une étape incontournable pour résoudre ce problème.

D'une part, le modèle des machines d'Eilenberg effectives permet de définir des calculs non-déterministes complexes en exploitant la généralité et la modularité du modèle. D'autre part les expressions régulières sont universellement reconnues comme l'outil adéquat pour définir des automates, et donc la partie contrôle d'une machine d'Eilenberg. Pour cette raison et aussi parce que les automates sont omniprésents en informatique, nous nous sommes intéressés dans le chapitre III aux algorithmes les plus efficaces qui permettent de synthétiser des automates non-déterministes à partir d'expressions régulières. Notre contribution est de présenter les principaux algorithmes issus de la technique des dérivées de Brzozowski et surtout de fournir une implémentation pour chacun d'eux en nous appuyant sur des travaux récents. Les algorithmes auxquels nous nous sommes intéressés sont les plus efficaces dans leur catégorie et nous avons attaché beaucoup d'importance à les implémenter en respectant cette propriété. Nous avons fourni des bancs d'essais permettant de comparer les différents algorithmes et de valider l'efficacité des implémentations proposées.

Nous voyons deux pistes de recherche prolongeant directement ces travaux. La première piste est de poursuivre la spécification au point de prouver formellement la correction des algorithmes. D'une part, la certification d'algorithme présente des enjeux cruciaux et d'autre part, les algorithmes de Thompson et de Berry-Sethi (pour ne citer qu'eux) sont tellement utilisés qu'il est intéressant de fournir des versions formellement vérifiées. Puisque nos implémentations sont faites dans le langage de programmation fonctionnelle OCaml nous avons donc déjà effectué une étape principale vers la formalisation complète dans un assistant de preuves tel que Coq. La seconde piste de recherche concerne l'extension aux multiplicités. En effet, il semble que ces algorithmes s'étendent facilement aux expressions régulières avec multiplicités et par ailleurs nous avons remarqué que nos implémentations respectent aussi les multiplicités des expressions, il s'agit donc de s'assurer de cette propriété.

Plus généralement, cette étude des machines d'Eilenberg ouvre la voie vers un nouveau paradigme de programmation à base de relations qu'on appelle la *programmation relationnelle*. Les relations effectives seraient au centre, non seulement de la composante de données, mais aussi de la composante de contrôle (comme cela a été présenté dans le chapitre II section 1). Le problème essentiel à résoudre pour mettre en œuvre ce paradigme de programmation relationnelle est de concevoir un langage de haut niveau permettant d'exprimer des problèmes relationnels, aussi bien sur la composante de contrôle que sur la composante de données. On considèrerait les expressions régulières, bien sûr, mais avec une notion de modularité permettant d'une part de composer des machines avec des interfaces, mais aussi d'instancier une signature d'actions avec les relations caractéristiques de sous-machines. Il pourrait y avoir des annotations permettant d'exprimer des propriétés telles que la terminaison ou la séquentialité, ce qui justifierait la compilation des machines avec certaines stratégies. On pourrait aussi spécifier des présentations relationnelles, soit comme des règles de réécriture entre expressions relationnelles, soit comme des relations paramétriques à la Prolog, soit comme des attachements sémantiques à des producteurs de streams ou de flux définis en ML. Nous avons déjà pu constater que cette approche est adaptée à la résolution de problèmes de linguistique computationnelle mais devrait s'appliquer à bien d'autres domaines qu'il faudra explorer. Nous conclurons cette thèse en disant que l'étude des machines d'Eilenberg ouvre donc la voie à de nombreuses pistes de recherche à l'interface de domaines scientifiques variés.

Bibliographie

[AM95] Valentin M. Antimirov and Peter D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1) :51–72, 29 May 1995.

[AM06] Cyril Allauzen and Mehryar Mohri. A unified construction of the Glushkov, Follow, and Antimirov automata. *Springer-Verlag LNCS*, 4162 :110–121, 2006.

[AMR08] Marco Almeida, Nelma Moreira, and Rogério Reis. Antimirov and Mosses's rewrite system revisited. In O. Ibarra and B. Ravikumar, editors, *CIAA 2008 : Thirteenth International Conference on Implementation and Application of Automata*, number 5448 in LNCS, pages 46–56. Springer-Verlag, 2008.

[AMR09] Marco Almeida, Nelma Moreira, and Rogério Reis. Antimirov and Mosses's rewrite system revisited. *International Journal of Foundations of Computer Science*, 2009. To appear.

[Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2) :291–319, 1996.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art :the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[BK93] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2) :197–213, 1993.

[BP09] Thomas Braibant and Damien Pous. A tactic for deciding Kleene algebras. In *Proceedings of the first Coq Workshop, to appear in the Journal of Formalized Reasoning*, 2009.

[BR88] Jean Berstel and Christophe Reutenauer. *Rational series and their languages*. Eatcs Monographs On Theoretical Computer Science, 1988.

[Brz64] Januz A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4) :481–494, October 1964.

[Brz71] Januz A. Brzozowski. Review. *The Journal of Symbolic Logic*, 36 :694–694, 1971.

[BS86] Gérard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1) :117–126, 1986.

[CLOZ04] Jean-Marc Champarnaud, Eric Laugerotte, Faissal Ouardi, and Djelloul Ziadi. From regular weighted expressions to finite automata. *International Journal of Foundations of Computer Science*, 15(5) :687–700, 2004.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion : From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.

[CNZ06] Jean-Marc Champarnaud, Florent Nicart, and Djelloul Ziadi. From the ZPC structure of a regular expression to its follow automaton. *International Journal of Algebra and Computation (IJAC)*, 16(1) :17–34, 2006.

[Con71] John Horton Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.

[COZ07] Jean-Marc Champarnaud, Faissal Ouardi, and Djelloul Ziadi. Normalized expressions and finite automata. *International Journal of Algebra and Computation (IJAC)*, 17(1) :141–154, 2007.

[CS63] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, pages 118–161, 1963.

[CZ01] Jean-Marc Champarnaud and Djelloul Ziadi. From c-continuations to new quadratic algorithms for automaton synthesis. *International Journal of Algebra and Computation (IJAC)*, 11(6) :707–736, 2001.

[CZ02] Jean-Marc Champarnaud and Djelloul Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theoretical Computer Science*, 289(1) :137 – 163, 2002.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8) :465–476, 1979.

[Dol03] Igor Dolinka. The multiplicative fragment of the Yanov equational theory. *Theoretical Computer Science*, 301(1-3) :417–425, 2003.

[dR03] Daniel de Rauglaudre. Camlp4 - Reference manual. Available on the Web, `http://caml.inria.fr/`, 2003.

[Eil74] Samuel Eilenberg. *Automata, languages, and machines, Volume A.* Academic Press, 1974.

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the ACM SIGPLAN Workshop on ML 2006.* ACM, 2006.

[Glu61] V M Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16 :1–53, 1961.

[Gro72] Maurice Gross. Notes sur l'histoire de la traduction automatique. *Langages*, 7(28) :40–48, 1972.

[HR06] Gérard Huet and Benoît Razet. The reactive engine for modular transducers. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 355–374. Springer-Verlag LNCS vol. 4060, 2006. `http://yquem.inria.fr/~huet/PUBLIC/engine.pdf`

[Hue02] Gérard Huet. The Zen computational linguistics toolkit. *ESSLLI 2002 Lectures, Trento, Italy*, 2002.

[Hue05] Gérard Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional programming*, 15, 2005.

[IY03] Lucian Ilie and Sheng Yu. Follow automata. *Inf. Comput.*, 186(1) :140–162, 2003.

[Kar00] Lauri Karttunen. Applications of finite-state transducers in natural language processing. In *Proceedings, CIAA-2000*, 2000.

[KK94] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20,3 :331–378, 1994.

[Kle56] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. *Automata Studies, Annals of Mathematics Studies*, 36, 1956.

[Koz90] Dexter Kozen. On Kleene algebras and closed semirings. In Rovan, editor, *Proc. Math. Found. Comput. Sci.*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47, Banska-Bystrica, Slovakia, 1990. Springer-Verlag.

[Koz94] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2) :366–390, May 1994.

[Koz98] Dexter Kozen. Typed Kleene algebra. Technical Report TR98-1669, Computer Science Department, Cornell University, March 1998.

[LDGV09] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2009.

[LS05] Sylvain Lombardy and Jacques Sakarovitch. Derivatives of rational expressions with multiplicity. *Theoretical Computer Science*, 332 :141–177, 2005.

[Mir66] B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering cybernetics*, 5 :110–116, 1966.

[Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23,2 :269–311, 1997.

[MY60] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9 :39–47, 1960.

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2) :173–190, March 2009.

[Pra90] Vaughan Pratt. Action logic and pure induction. In *Logics in AI : European Workshop JELIA '90*, pages 97–120. LNCS 478, Springer Verlag, 1990.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6), december 1987.

[Ran04] Aarne Ranta. Grammatical framework : A type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2) :145–189, 2004.

[Ray96] Pascal Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, Paderborn, Germany, July 1996. LNCS 1099, Springer Verlag.

[Raz08a] Benoît Razet. Finite Eilenberg machines. In *13th International Conference on Implementation and Application of Automata, (CIAA 2008)*, volume 5148, pages 242–251. Springer Verlag, LNCS, 2008.

[Raz08b] Benoît Razet. Simulating finite Eilenberg machines with a reactive engine. In *Mathematically Structured Functional Programming 2008, MSFP'08*. Electronic Notes in Theoretical Computer Science, ENTCS, 2008.

[Red64] V. N. Redko. On defining relations for the algebra of regular events. *Ukrain. Mat.*, 16 :120–126, 1964.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* MIT press, 1967.

[RS59] Michael Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2) :114–125, 1959.

[RS95] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21,2 :227–253, 1995.

[RS97] Emmanuel Roche and Yves Schabes. *Finite-state language processing.* MIT press, 1997.

[Sak03] Jacques Sakarovitch. Eléments de théorie des automates. *Vuibert*, 2003.

[Sal66] Arto Salomaa. Two complete axiom sytems for the algebra of regular events. *Journal of the Association for Computing Machinery*, 13(1) :158–169, 1966.

[Sar02] João Saraiva. HaLeX : A Haskell library to model, manipulate and animate regular languages. In *Proceedings of the ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02)*, october 2002.

[Sch61] Marcel-Paul Schützenberger. On the definition of a family of automata. *Information and control*, 4 :245–270, 1961.

[Spr92] Richard Sproat. *Morphology and Computation.* MIT Press, 1992.

[SS88] Seppo Sippu and Eljas Soisalon-Soininen. Parsing theory. vol. 1 : languages and parsing. *EATCS Monographs on Theoretical Computer Science*, 15, 1988.

[Tea09] The Coq Development Team. The Coq proof assistant. Software and documentation available on the Web, 1995–2009.

[Tho68] Ken Thompson. Programming techniques : Regular expression search algorithm. *Commun. ACM*, 11(6) :419–422, 1968.

[Tho00] Simon Thompson. Regular expressions and automata using Haskell. Technical report, University of Kent, january 2000.

[Yan62] Ju.I. Yanov. On identical transformations of regular expressions. *Dokl. Akad. Nauk SSSR*, 147 :327–330, 1962.

[ZS01] Guo-Qiang Zhang and Leon Smith. A collection of functional libraries for theory of computation. In *Proceedings of the 39th ACM-SE Conference*, pages 83–90. ACM Press, May 2001.

Index

algèbre de Kleene, 19
algèbre d'actions, 24
+-algèbre, 31
algorithme d'Antimirov, 87
  implémentation, 97
algorithme d'Ilie-Yu, 90
  implémentation, 93
algorithme de Berry-Sethi, 84
  implémentation, 91
algorithme de Brzozowski, 82
algorithme de Thompson, 75
alphabet, 15
automate, 17
  à transitions spontanées ϵ-NFA, 17
  déterministe DFA, 17
  non-déterministe NFA, 17
automate d'équations, 87
automate de continuations, 88
automate de positions, 85
automate follow, 90
axiomatisation, 19
  équationnelle, 19
  complète, 19
  finie, 19
  quasi-équationnelle, 19

cellule, 44
continuation, 85
c-continuation, 87
continuation canonique, 87

dérivée canonique, 87
dérivée de Brzozowski, 82
dérivée partielle, 86

ensemble récursivement énumérable, 39
ϵ-réduction, 79
expression régulière, 16
  annotée, 76
  décorée, 76
  forme normale d'étoile, 80
  largeur, 16
  linéaire, 78
  normalisée, 82
  taille, 16

flux, 40
  fini, 42
  productif, 41
forme normale d'étoile, 80

langage, 16
  dérivée, 17
  de Yanov, 30
  opérations, 16
  régulier, 16
  résiduation, 17
  reconnaissable, 17
lien follow, 93

machine d'Eilenberg, 44
  globalement finie, 48
  interface, 45
  localement finie, 48
  noyau, 45
  nœthérienne, 48
  relation caractéristique, 44
machine d'Eilenberg finie, 48
machine relationnelle, 35
monoïde, 15
  libre, 15
mot, 15
moteur réactif, 50
  machine d'Eilenberg effective, 62
  machine d'Eilenberg finie, 51
moteur semi-réactif, 68
  machine d'Eilenberg effective, 68

OCaml, 37
  foncteur, 39
  module, 38

programmation fonctionnelle, 37

relation, 35

calculable, 39
localement finie, 48

saut de Conway, 28
semigroupe, 15
symbole, 15

théorème d'égalité, 104
théorème de Kleene, 18
trace, 44

variété, 25
quasi-variété, 25