Université Lille 1 — N^o **d'ordre : 41446**

Département de formation doctorale en informatique — **École doctorale SPI Lille**
UFR IEEA

Normalisation et Apprentissage de Transductions d'Arbres en Mots

THÈSE

présentée et soutenue publiquement le 4 Juin 2014

pour l'obtention du

Doctorat de l'Université Lille 1 - Sciences et Technologies
(spécialité informatique)

par

Grégoire Laurence

Composition du jury

Président :	Olivier Carton (Olivier.Carton@liafa.univ-paris-diderot.fr)
Rapporteurs :	Olivier Carton (Olivier.Carton@liafa.univ-paris-diderot.fr) Marie-Pierre Béal (beal@univ-mlv.fr)
Directeur de thèse :	Joachim Niehren (joachim.niehren@lifl.fr)
Co-Encadreur de thèse :	Aurélien Lemay (aurelien.lemay@univ-lille3.fr)

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Résumé

Le stockage et la gestion de données sont des questions centrales en informatique. La structuration sous forme d'arbres est devenue la norme (XML, JSON). Pour en assurer la pérennité et l'échange efficace des données, il est nécessaire d'identifier de nouveaux mécanismes de transformations automatisables.

Nous nous concentrons sur l'étude de transformations d'arbres en mots représentées par des machines à états finies. Nous définissons les transducteurs séquentiels d'arbres en mots ne pouvant utiliser qu'une et unique fois chaque nœud de l'arbre d'entrée pour décider de la production.

En réduisant le problème d'équivalence des transducteurs séquentiels à celui des morphismes appliqués à des grammaires algébriques (Plandowski, 95), nous prouvons qu'il est décidable en temps polynomial.

Cette thèse introduit la notion de transducteur travailleur, forme normalisée de transducteurs séquentiels, cherchant à produire la sortie le «plus tôt possible» dans la transduction. A l'aide d'un algorithme de normalisation et de minimisation, nous prouvons qu'il existe un représentant canonique, unique transducteur travailleur minimal, pour chaque transduction de notre classe.

La décision de l'existence d'un transducteur séquentiel représentant un échantillon, i.e. paires d'entrées et sorties d'une transformation, est prouvée NP-difficile. Nous proposons un algorithme d'apprentissage produisant à partir d'un échantillon le transducteur canonique le représentant, ou échouant, le tout en restant polynomial. Cet algorithme se base sur des techniques d'inférence grammaticales et sur l'adaptation du théorème de Myhill-Nerode.

Titre : Normalisation et Apprentissage de Transductions d'Arbres en Mots

Abstract

Storage, management and sharing of data are central issues in computer science. Structuring data in trees has become a standard (XML, JSON). To ensure preservation and quick exchange of data, one must identify new mechanisms to automatize such transformations.

We focus on the study of tree to words transformations represented by finite state machines. We define sequential tree to words transducers, that use each node of the input tree exactly once to produce an output.

Using reduction to the equivalence problem of morphisms applied to context-free grammars (Plandowski, 95), we prove that equivalence of sequential transducers is decidable in polynomial time.

We introduce the concept of earliest transducer, sequential transducers normal form, which aim to produce output "as soon as possible" during the transduction. Using normalization and minimization algorithms, we prove the existence of a canonical transducer, unique, minimal and earliest, for each transduction of our class.

Deciding the existence of a transducer representing a sample, i.e. pairs of input and output of a transformation, is proved NP-hard. Thus, we propose a learning algorithm that generate a canonical transducer from a sample, or fail, while remaining polynomial. This algorithm is based on grammatical inference techniques and the adaptation of a Myhill-Nerode theorem.

Title : Normalization and Learning of Tree to Words Transductions

Remerciements

Cet espace me permettant de remercier toutes les personnes m'ayant aider à effectuer, rédiger, soutenir et fêter cette thèse, je tiens tout d'abord à saluer le travail effectué par Marie-Pierre Béal et Olivier Carton, qui ont du rapporter et être jury de cette thèse. Je les remercie d'avoir eu le courage de relire l'intégralité de ce manuscrit, des retours qu'ils m'en ont fait, et de l'intérêt qu'ils ont portés à mon travail.

Cette thèse n'aurait jamais eu lieu sans la présence de mes encadrants, tout particulièrement Joachim Niehren qui m'a permis d'intégrer cette équipe dès mon stage de recherche, et lancé les travaux qui mènent à ce que vous tenez maintenant entre vos mains. Merci à Joachim, Aurélien Lemay, Slawek Staworko et Marc Tommasi de m'avoir suivi pendant cette (longue) épreuve, de m'avoir guidé, soutenu, et aider à mener cette thèse pour arriver à ce résultat qui je l'espère vous fait autant plaisir qu'a moi. Tout n'a pas toujours été facile, je n'ai pas toujours été aussi investi qu'il aurait fallut, mais vous avez toujours réussi à me remettre sur la route, et permis d'arriver à cette étape.

Je ne peut remercier mes encadrants sans penser à l'intégralité des membres de l'équipe Mostrare (links, magnet, et même avant) qui m'ont permis de passer ces quelques années dans un parfait environnement de travail (mais pas que). Ayant plus ou moins intégré cette équipe depuis mon stage de maitrise (grâce à Isabelle Tellier et Marc), il me serait difficile d'énumérer ici l'intégralité des membres que j'y ai croisé, et ce que chacun m'a apporté. Je m'adonne quand même à ce petit exercice pour l'ensemble des doctorants qui ont partagé cette position de thésard dans l'équipe : Jérôme, Olivier, Edouard, Benoît, Antoine, Tom, Jean, Adrien, Radu, et Guillaume (même si il n'était pas doctorant).

Ces années ont également été occupées par mes différents postes dans l'enseignements à Lille 3 ou encore Lille 1 et je remercie l'ensemble des enseignants qui m'ont aidés et accompagné durant cette tâche, tout particulièrement Alain Taquet avec qui j'ai partagé mes premiers enseignements.

Même si il n'ont pas à proprement contribué à ce que contient cette thèse (enfin ça dépend lesquels), malgré eux ils y sont pour beaucoup, je parle bien entendu de mes amis et ma famille. Je n'en ferait pas non plus la liste ici, n'en déplaise à certains, mais je remercie tout particulièrement mes parents, ma soeur pour m'avoir supporter et permis d'en arriver là, mon parrain et ma marraine pour leur présence et l'intérêt tout particulier qu'ils ont portés à ma thèse. Merci à toute ma famille et mes amis d'avoir été présents avant, pendant, et je l'espère encore longtemps après cette thèse, d'avoir réussi à me la

faire oubliée parfois. Une pensée toute particulière à Sébatien Lemaguer qui a eu la lourde tâche de relire l'intégralité de cette thèse (sauf ces remerciements) à la recherche de fautes (trop nombreuses).

Je ne pouvait pas finir ces remerciement sans parler de mon amie, amour, Manon, a qui cette thèse appartient au moins autant qu'a moi, support inconditionnel dans l'ombre sans qui je n'aurai surement pas réussi à terminer cette thèse. Elle a vécue au moins autant que moi toute cette épreuve, jusqu'à son dernier moment, et m'a permis de malgré tout mes périodes de doutes d'aboutir à ce résultat. Merci à toi,

et merci à vous tous.

Table des matières

Introduction **1**

1 Automates **11**
1.1 Mots . . . 11
1.1.1 Mots et langages . . . 11
1.1.2 Grammaires . . . 14
1.1.3 Automates . . . 15
1.2 Arbres d'arité bornée . . . 18
1.2.1 Définition . . . 19
1.2.2 Automates d'arbres . . . 21
1.3 Arbres d'arité non-bornée . . . 25
1.3.1 Definition . . . 25
1.3.2 Comment définir les automates ? . . . 25
1.3.3 Codage binaire curryfié (ascendant) . . . 26
1.3.4 Codage binaire frère-fils (descendant) . . . 29
1.4 Mots imbriqués . . . 31
1.4.1 Définition . . . 31
1.4.2 Linéarisation d'arbres d'arité non-bornée . . . 32
1.4.3 Automates de mots imbriqués . . . 33
1.4.4 Automate descendant . . . 34

2 Transducteurs **37**
2.1 Transducteurs de mots . . . 39
2.1.1 Transducteurs rationnels . . . 39
2.1.2 Transducteurs déterministes . . . 50
2.1.3 Transducteurs déterministes avec anticipation . . . 76
2.2 Transducteur d'arbres d'arité bornée . . . 80
2.2.1 Transducteur descendants . . . 81
2.2.2 Transducteur ascendant . . . 88
2.2.3 Transducteur avec anticipation . . . 90
2.2.4 Macro-transducteurs descendants . . . 93
2.3 Transducteurs d'arbres en mots . . . 95
2.3.1 Transducteurs descendants . . . 95
2.3.2 Transducteurs ascendants . . . 98
2.4 Transducteurs d'arbres d'arité non bornée . . . 100
2.4.1 Transducteurs de mots imbriqués en mots . . . 101
2.4.2 Transducteurs de mots imbriqués en mots imbriqués . . . 104

3 Transformations XML **105**
3.1 XSLT . . . 105
3.1.1 XSLT en Pratique . . . 105
3.2 Transducteurs et XSLT . . . 109
3.2.1 *d*ST2W . . . 109
3.2.2 Macro-Transducteurs et XPATH . . . 110

4 Équivalence de transducteurs séquentiels d'arbres en mots **115**
4.1 Relation avec l'équivalence de morphismes sur CFGs . . . 115
4.1.1 Exécution d'un dNW2W . . . 116
4.1.2 Arbre syntaxique étendu . . . 116
4.2 Relation entre *d*B2W et *d*T2W . . . 121

5 Normalisation et minimisation des transducteurs descendants d'arbres en mots **125**
5.1 *d*ST2W travailleur . . . 125
5.2 Caractérisation sémantique . . . 127
5.2.1 Approche naïve . . . 127
5.2.2 Décompositions et résiduels . . . 129
5.2.3 Transducteur canonique . . . 133
5.3 Minimisation . . . 134
5.3.1 Minimisation des *ed*ST2W*s* . . . 135
5.3.2 Minimisation de *d*ST2W*s* arbitraires . . . 138
5.4 Normalisation . . . 142
5.4.1 Réduction de langages . . . 143
5.4.2 Faire traverser un mot dans un langage . . . 143
5.4.3 Déplacement de gauche à droite . . . 148
5.4.4 Algorithme de normalisation . . . 151
5.4.5 Bornes exponentielles . . . 158

6 Apprentissage de transducteurs descendants d'arbres en mots **161**
6.1 Théorème de Myhill-Nerode . . . 161
6.2 Consistence des *d*ST2W*s* . . . 162
6.3 Modèle d'apprentissage . . . 165
6.4 Algorithme d'apprentissage . . . 166
6.5 Décompositions, résiduels et équivalence . . . 166
6.6 Echantillon caractéristique . . . 170

7 Conclusion **177**
7.1 Perspectives . . . 178

Bibliographie **183**

Introduction

De tout temps, le stockage, la gestion et l'échange de données sont des questions centrales en informatique, plus encore de nos jours, la quantité de données devenant de plus en plus importante et partagée entre différents services. La structuration des données participe énormément à ces défis. Elle permet par exemple d'intégrer une certaine sémantique absolument nécessaire pour l'échange, d'effectuer plus efficacement des traitements comme des requêtes ou des transformations. Les arbres de données sont une telle représentation structurée des informations. Mon travail de thèse est une contribution à l'étude de ces arbres de données et particulièrement leurs transformations. Une question centrale que j'aborde consiste en la définition d'algorithmes d'apprentissage de ces programmes de transformations, que je représente sous la forme de machines à états finis : les transducteurs.

Arbres de données

Les arbres de données permettent d'apporter une structure à des données textuelles. Ces arbres sont composés de nœuds étiquetés par des symboles issus d'un alphabet fini. Les valeurs textuelles se retrouvent au niveau du feuillage de l'arbre ou dans ses nœuds internes sous forme d'attributs. Elles sont composés à partir d'un alphabet fini, comme par exemple celui de l'ASCII ou de l'unicode, mais ne sont pas bornées en taille.

Certains modèles existants, tels que le XML (recommandation du W3C) ou JSON (JavaScript Object Notation), cherchent à homogénéiser le format des données des documents structurés sous forme d'arbre en proposant une syntaxe permettant de les représenter. Que ce soit sous forme d'arbres d'arité non bornée, ordonnés, associés à des champs textuels et des enregistrements pour le XML, ou d'arbres d'arité bornée et d'ensembles d'enregistrements non bornés ni ordonnés pour ce qui est de JSON, le but reste de spécifier une syntaxe en limitant le moins possible la sémantique que l'utilisateur souhaite associer à la structure.

Certains formats de données qui en découlent limitent le choix des étiquettes de nœuds utilisables dans la structure. Ces étiquettes (les balises) doivent appartenir à un ensemble fini et fixé et portent chacune une sémantique forte. C'est le cas de fichiers HTML ou des méta-données contenues dans certains document issus de logiciels de traitement de texte. Les arbres de données peuvent se trouver également dans les bases de données, ou encore

dans le «Cloud computing», où même si parfois ils partagent des syntaxes proches, peuvent être utilisés dans des domaines totalement différents avec des sémantiques qui leurs sont propres. Que ce soit pour l'échange, la mise en relation ou la sélection de données, il devient de plus en plus nécessaire, au delà des modifications automatiques de texte, comme pouvait le permettre des programmes tels que Perl, d'identifier de nouveaux mécanismes de transformation basés sur la structure. Il est surtout intéressant de voir jusqu'où les méta-données contenues dans la structure même jouent un rôle dans le choix des transformations à effectuer.

Transformations

Pour cela, il faut dans un premier temps se demander quel type de transformation nous intéresse. Il existe déjà plusieurs moyens de transformer des arbres de données, que ce soit pour une tâche spécifique ou comme le permet le langage XSLT, une représentation des transformations dans le sens le plus large du terme. Le langage de programmation XSLT (Clark, 1999), pour «eXtensible Stylesheet Language Transformations», est un langage défini au sein de la recommandation XSL du consortium W3C. Il permet de transformer des arbres de données, représentés par un ou plusieurs XML, en données de différents types, XML, textuels ou même binaires. Il est lui même représenté sous la forme d'un arbre de données dont certaines étiquettes de noeuds spécifient les fonctions de transformation à appliquer. Cette application passe par la sélection de noeuds à transformer à l'aide de requêtes XPATH (Clark et DeRose, 1999). Ces requêtes permettent la sélection de noeuds dans un arbre de données à l'aide d'opérations primitives de déplacement dans la filiation de l'arbre. Une grande expressivité se faisant souvent au détriment de la complexité, XSLT est un programme Turing-complet (Kepser, 2004; Onder et Bayram, 2006). Pour pouvoir étudier théoriquement et se diriger vers un possible apprentissage, Il est nécessaire d'en introduire des sous classes, moins expressives.

L'intérêt d'un arbre de données étant de structurer des informations pour en simplifier la recherche et l'utilisation, il n'est pas surprenant qu'une des fonctions les plus utilisées dans le langage XSLT soit la fonction «$xsl : value - of$» retournant l'ensemble des valeurs textuelles contenues dans les feuilles des sous-arbres traités. Par exemple, comme l'illustre la figure 1, cherchons à récupérer le nom et prénom du premier contact contenu dans un carnet d'adresse sous format XML, pour un futur affichage dans un document HTML . Cela donne en XSLT, l'appel de la balise suivante :

```
<xsl:value-of select="/contacts/contact[1]/identite" />
```

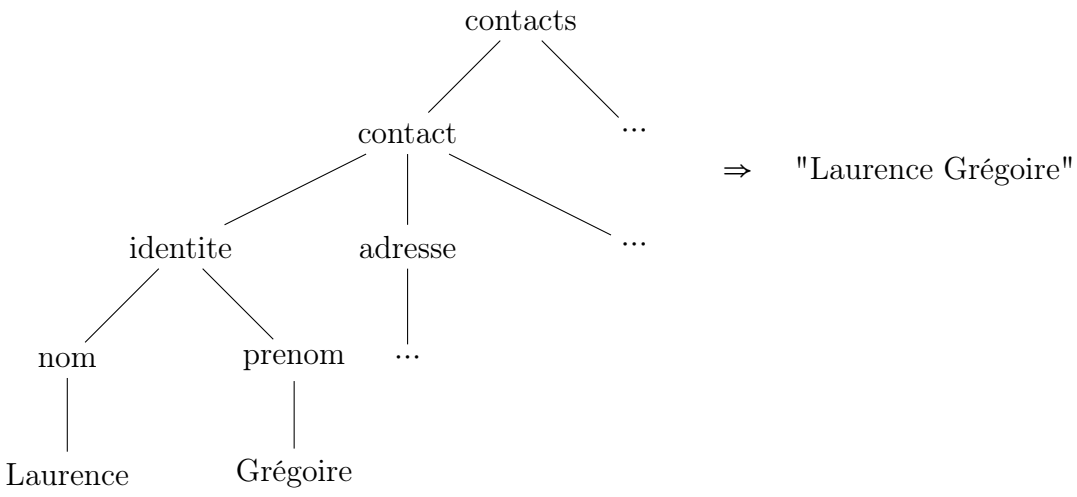


FIGURE 1 – Exemple de transformation XSLT basique

Seule la structure de l'arbre importe ici pour décider du traitement à effectuer sur les feuilles. De ce fait, les valeurs textuelles pourraient être abstraites sous la forme de balises dédiées ne servant qu'à les identifier, et permettre leur réutilisation dans la sortie.

Toutefois, la structure interne n'est pas toujours suffisante pour décider du traitement à effectuer. Les valeurs textuelles sont parfois des éléments centraux d'une transformation. Les opérations telles que les jointures utilisent ces valeurs textuelles comme repères pour regrouper certaines informations. Dès lors elles font partie intégrante de la sélection des noeuds à manipuler et ne peuvent plus être ignorés.

Qu'en est il de la sortie ? Il n'est pas toujours intéressant de garder la structure de l'entrée. Comme l'illustre la transformation représentée dans la figure 1, il est parfois nécessaire de se concentrer sur la concaténation de champs textuels sélectionnés. Même si une structure est parfois nécessaire, il reste tout à fait possible de la représenter sous format XML, une chaîne composée de balises ouvrantes et fermantes représentant un arbre de données. Cette liberté dans la représentation de la sortie se fait au détriment du contrôle de la structure de sortie, qui n'est plus reconnue en tant qu'arbre de données, empêchant ainsi son utilisation directe en entrée d'une autre transformation. Il n'est plus possible dès lors de composer plusieurs transformations sans prétraitements sur les données intermédiaires.

Nous décidons dans cette thèse de nous concentrer sur les transformations d'arbres en mots qui permettent la concaténation dans la sortie. Cela représente une simplification des transformations d'arbres de données à arbres de données. Il est dès lors plus possible d'exprimer les opérations de jointures.

Nous perdons également les propriétés de compositions, la structure d'entrée n'étant plus présente dans la production. Cela nous permet de nous concentrer sur la possibilité de manipuler les chaînes de sortie, ce qui nous intéresse ici.

Motivation générale

Notre but reste l'automatisation de la tâche de transformation. Cela passe par un apprentissage de la transformation que l'on souhaite effectuer. L'apprentissage considéré ici revient à chercher à identifier une cible, appartenant à une classe de langages connue, à partir d'exemples (et de possibles contre-exemples). Il nous reste donc à identifier formellement une classe de transformations d'arbres en mots, de choisir le modèle d'apprentissage que nous souhaitons appliquer, et les exemples à partir desquels apprendre.

L'apprentissage que nous souhaitons utiliser se base sur l'inférence grammaticale (Gold, 1967) en cherchant à identifier à la limite un langage cohérent avec les exemples d'entrée. Le plus souvent cette technique se divise en deux étapes, une représentation des exemples dans un formalisme les regroupant, et la généralisation de ce modèle pour en déduire un langage reconnaissant souvent un langage plus large que celui représenté par les exemples, mais tout en restant cohérent avec cet échantillon.

Pour ce qui est des exemples, la transformation d'arbres en mots peut être vue comme un ensemble de paires composées d'un arbre et de la chaîne de mots résultant de la transformation de cet arbre. Il est connu que, pour apprendre un langage de mots régulier, ne considérer que des exemples positifs n'est pas suffisant en inférence grammaticale (Gold, 1967). Il est nécessaire d'avoir des contre-exemples ou autres éléments permettant d'éviter une sur-généralisation. Qu'en est-il de nos transformations ?

Une des propriétés permettant de contrôler cette possible sur-généralisation est le fait qu'une transformation doit rester fonctionnelle. Nous devons pour cela nous assurer qu'une entrée ne puisse être transformée qu'en au plus un résultat. Le deuxième contrôle peut être fait à l'aide du domaine d'entrée, le langage d'arbres sur lequel la transformation peut s'effectuer. Il existe pour cela de nombreuses possibilités, que ce soit à l'aide d'exemples négatifs ou encore par la connaissance directe du domaine, donné en entrée de l'apprentissage. L'apprentissage de langage d'arbres étant un problème connu et ayant déjà été résolu par inférence grammaticale (Oncina et Garcia, 1992), nous opterons pour la deuxième solution, en supposant que le domaine nous est déjà donné.

Une fois le type d'exemples et d'apprentissage choisi, il nous reste à choisir quel modèle utiliser pour représenter notre transformation, vérifier que ce mo-

dèle dispose des propriétés nécessaires pour un apprentissage, et enfin définir l'algorithme d'apprentissage à proprement parler. Nous choisissons d'utiliser les transducteurs, machines à états finis permettant d'évaluer une entrée en produisant la sortie associée.

Nous souhaitons donc que toute transformation de la classe qui nous intéresse soit définissable par la classe de transducteurs choisie, ce qu'il reste à prouver. Nous voulons également que la cible d'un apprentissage soit unique, qu'a chaque transformation lui soit associé un transducteur *canonique* la représentant. Cela repose sur la normalisation du transducteur, pour en homogénéiser le contenu, et de sa minimisation pour assurer l'unicité. Il est donc nécessaire d'introduire une classe de transducteurs permettant de représenter les transformations d'arbres en mots, disposant d'une forme normale. La cible de l'apprentissage d'une transformation sera le transducteur canonique, unique minimal, de cette classe la représentant. Il est donc nécessaire de définir un théorème de type Myhill-Nerode (Nerode, 1958) pour ce modèle, assurant, entre autre, l'existence d'un transducteur canonique pour chaque transformation de notre classe. L'algorithme d'apprentissage en lui même se résume à la décomposition de la transformation représentée par les exemples, pour en déduire les états du transducteur cible. Cela repose sur la possibilité de tester efficacement l'équivalence de fragments de la transformation pour identifier les fragments communs. Il reste à spécifier le modèle de transducteurs à l'aide duquel nous souhaitons modéliser notre transformation.

Transducteurs

Que ce soit sur les mots ou les arbres, de nombreux modèles de transducteurs ont déjà été proposés et étudiés dans le domaine. Nous nous intéressons particulièrement aux modèles déterministes, ne permettant qu'une exécution possible pour une donnée d'entrée fixée, qui en plus d'assurer la fonctionnalité de la transformation représentée, simplifie la décidabilité de problèmes importants. Avant de nous concentrer sur un modèle permettant de transformer des arbres en mots, nous pouvons évoquer deux classes de transducteurs permettant respectivement de gérer les mots et les arbres.

Les transducteurs sous-séquentiels, transducteurs déterministes de mots introduits par Schützenberger (1975), transforment, lettre par lettre un mot d'entrée dans sa sortie. Cette production est obtenue par concaténation de toutes les chaînes produites par le transducteur. Pour ce qui est des arbres, les transducteurs déterministes d'arbres d'arité bornée, basés sur les travaux de Rounds (1968) et Thatcher (1970), permettent, en parcourant un arbre de sa racine à ses feuilles, de produire un arbre. Chaque règle de cet arbre associe

à un noeud de l'arbre d'entrée et ses fils, un contexte, arbre à trous. Chacun de ces sous-arbres manquants s'obtient par l'application d'une transduction à un fils de ce noeud. Aucune contrainte n'est faite sur l'ordre d'utilisation de ces fils ou encore du nombre de fois qu'ils sont utilisés. Cela revient à autoriser la copie et le réordonnancement des fils d'un noeud dans la sortie. Il n'est cependant pas possible de fusionner le contenu de plusieurs noeuds ou de supprimer des noeuds internes.

Ces classes de transducteurs disposent de résultats de décidabilité sur les problèmes théorique importants. Ainsi, le problème d'équivalence est montré décidable, par Engelfriet *et al.* (2009) pour les transducteurs d'arbres, et par Choffrut (1979) pour les transducteurs de mots. Une forme normale, assurant à l'aide de contraintes une représentation normalisé d'un transducteur, a été introduite pour chacun d'entre eux, que ce soit par Choffrut (1979) pour les mots ou par Lemay *et al.* (2010) pour les arbres, afin d'assurer l'existence d'un transducteur canonique, normalisé minimal unique, pour chaque transformation exprimable par un transducteur de la classe. Des algorithmes d'apprentissages ont été proposés pour chacune de ces classes, que ce soit l'algorithme OSTIA (Oncina *et al.*, 1993) pour les transducteurs sous-séquentiels, ou plus récemment sur les transducteurs d'arbres en arbres (Lemay *et al.*, 2010).

Pour ce qui est des transformations d'arbres en mots que nous cherchons à représenter, nous nous dirigeons sur une machine proche des transducteurs d'arbres, du moins sur le traitement de l'entrée, mais produisant cette fois-ci des mots. Les règles produisent maintenant, pour un noeud et ses fils, la concaténation de mots et du résultat de transduction des fils. Les problèmes théoriques, tel que l'équivalence et l'existence de représentants canoniques, sont ouverts pour cette classe de transductions.

Il est également possible de pousser vers d'autres modèles de transducteurs plus expressifs tels que les macro transducteurs d'arbres (Engelfriet, 1980), plus proche des transformations de type XSLT que les modèles évoqués précédemment. Mais ce modèle autorisant également des opérations telles que la concaténation, il n'est pas intéressant d'attaquer cette classe alors que des classes strictement moins expressives, et utilisant le même type d'opérateurs, n'ont toujours pas de résultats de décidabilité sur les problèmes théoriques.

Maintenant que nous avons décidé du type de transformation qui nous intéresse, d'arbres en mots, ainsi que la manière de les représenter, transducteurs déterministes d'arbres en mots, nous pouvons nous concentrer sur l'apprentissage en lui-même.

Contributions

Dans cette thèse, nous définissons et étudions les transducteurs séquentiels déterministes descendants d'arbres d'arité bornée en mots. Être séquentiel signifie que chaque règle de ces transducteurs associe à un noeud une production obtenue par concaténation de mots et de l'application d'une transduction sur chacun de ces fils. Chaque fils doit être utilisé une et une seule fois, et ce dans leur ordre d'apparition dans l'entrée. Les règles d'un transducteur séquentiel sont de la forme :

$$\langle q, f(x_1, \ldots, x_k)\rangle \to u_0 \cdot \langle q_1, x_1\rangle \cdot \ldots \langle q_k, x_k\rangle u_k$$

qui spécifie que la transformation d'un arbre $f(t_1, \ldots, t_k)$ à partir d'un état q sera le résultat de la concaténation des chaînes u_i et des résultats de transduction des sous arbres t_i à partir des états q_i respectifs. Le fait d'interdire la réutilisation et le réordonnancement des fils dans la production de la sortie est une simplification du modèle assez conséquente. Mais comme nous le verrons par la suite, cette restriction est nécessaire pour décider efficacement de problèmes théoriques cruciaux dans l'élaboration d'un apprentissage. Le problème d'équivalence de ce modèle est par exemple connu décidable pour la famille plus générale de transduction. Toute fois, il ne permet pas de disposer d'une solution efficace.

Équivalence efficace

En réduisant le problème d'équivalence à celui de l'équivalence de morphismes appliqués à des grammaires algébriques, prouvé décidable en temps polynomial dans la taille des morphismes et de la grammaire (Plandowski, 1995), nous prouvons que l'équivalence des transducteurs séquentiels est décidable également en temps polynomial dans la taille du transducteur considéré. Cette réduction se base sur la définition d'une grammaire représentant l'exécution parallèle de transducteurs séquentiels sur lesquels sont définis deux morphismes recomposant la sortie de transducteurs séquentiels respectifs. Toute la sortie est contenue dans les règles représentées sur cette exécution.

Malgré des expressivités foncièrement différentes, les transducteurs d'arbres en mots ascendants, descendants, et les transducteurs de mots imbriqués en mots partagent des représentations d'exécutions similaires, l'ordre de l'entrée y étant toujours gardé. Cette similarité permet de mettre en relations ces principales classes de transductions d'arbres en mots en étendant le résultat de décidabilité de l'équivalence à chacune d'entre elles (Staworko *et al.*, 2009).

Normalisation

Par la suite (Laurence *et al.*, 2011), nous introduisons une forme normale pour les transducteurs séquentiels d'arbres en mots, cherchant à produire la sortie le plus tôt possible dans le transducteur. Cette notion de «plus tôt» pouvant être perçue de plusieurs manières nous allons l'illustrer à l'aide d'un court exemple.

Exemple 1. *Prenons un transducteur séquentiel M composé de deux états q_0 et q_1, de l'axiome initial permettant de produire de la sortie avant même de commencer la lecture d'un arbre entrée q_0, n'ajoutant ici aucune sortie, et des règles suivantes.*

$$\langle q_0, f(x_1, x_2)\rangle \to \langle q_1, x_1\rangle \cdot ac \cdot \langle q_1, x_2\rangle, \quad \langle q_1, g(x_1)\rangle \to \langle q_1, x_1\rangle \cdot abc, \quad \langle q_1, a\rangle \to \varepsilon.$$

Nous pouvons représenter son exécution par l'annotation de l'arbre transformé, la sortie s'obtenant par la concaténation des mots produits en suivant le parcours en profondeur, comme l'illustre le premier arbre de la figure 2.

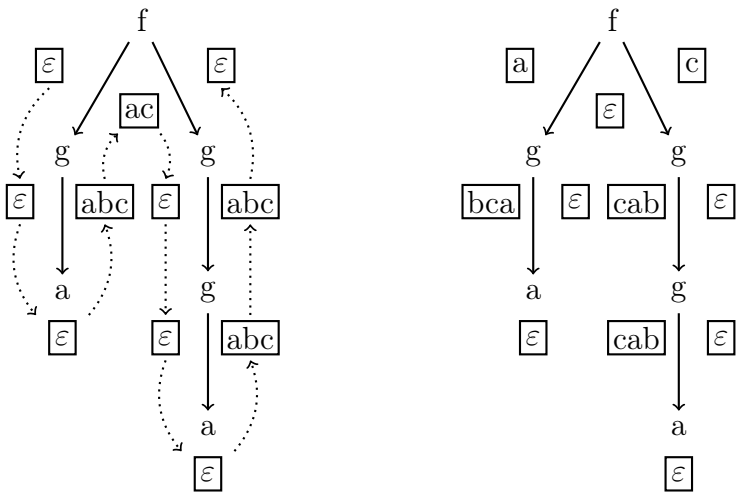


FIGURE 2 – Exemple d'exécution d'un transducteur séquentiel et sa version normalisée

Nous choisissons comme normalisation de débuter par la sortie la production le plus haut possible dans l'arbre. Par exemple, si l'arbre à pour racine f, nous savons directement que sa sortie débutera par un "a" et se terminera par un "c". Il nous faut donc remonter ces deux lettres, respectivement à gauche et à droite, en les faisant passer à travers la transduction des sous arbres. Une fois que la sortie se trouve le plus haut possible, pour assurer un seul résultat pour la normalisation d'un arbre, nous avons décider de forcer la sortie à être

produite le plus à gauche d'une règle, comme l'illustre le passage de droite à gauche des productions sur les noeuds étiquetés par un g.

Cela donne le transducteur suivant. L'état q_1 a été divisé en deux états q_g et q_d, le traitement du sous arbre gauche et droit étant maintenant différents. Les règles du transducteur obtenu sont les suivantes :

$$\langle q_0, f(x_1, x_2)\rangle \to \langle q_g, x_1\rangle \cdot ac \cdot \langle q_d, x_2\rangle, \qquad \langle q_g, g(x_1)\rangle \to bca \cdot \langle q_g, x_1\rangle,$$
$$\langle q_g, a\rangle \to \varepsilon, \qquad \langle q_d, g(x_1)\rangle \to cab \cdot \langle q_d, x_1\rangle, \qquad \langle q_d, a\rangle \to \varepsilon.$$

L'exécution de ce transducteur est illustré sur le deuxième arbre de la figure 2.

Nous définissons un algorithme de normalisation permettant à partir de tout transducteur séquentiel de le renvoyer sous forme normalisée. La difficulté de cette normalisation est la manipulation de la sortie, les mots devant à la fois être tirés à gauche et droite du transducteur et parfois poussés à travers la transduction de sous arbres, comme l'illustre l'exemple. Si on se limite à la sortie, cette opération revient à manipuler des mots à travers des grammaires algébriques de mots, projection du transducteur sur la sortie. La force de notre normalisation est, qu'a partir de contraintes globales devant s'appliquer les unes après les autres, nous avons réussi à en déduire des contraintes locales. En arrivant à représenter ces modifications à l'aide d'un langage d'opérations sur les mots, pouvant être directement appliquées aux états d'un transducteur, nous avons mis en place une normalisation efficace et locale d'un transducteur séquentiel d'arbres en mots.

Nous identifions également les bornes sur la taille d'un transducteur séquentiel normalisé en fonction de la taille du transducteur de base, pouvant parfois atteindre une taille double exponentielle dans celle de la source.

Apprentissage

Nous aboutissons, en nous basant sur les précédents résultats, sur l'apprentissage des transducteurs séquentiels d'arbres en mots (Laurence *et al.*, 2014). Pour cela, il faut tout d'abord identifier les transformations apprenables (i.e. représentables par des transducteurs séquentiels). Pour cela, on cherche à retrouver directement la structure des transducteurs dans une transformation. Cette restructuration passe par la décomposition de la sortie par rapport à l'entrée, l'introduction de sous transformations associées aux chemins de l'arbre d'entrée, ainsi que l'instauration de classes d'équivalence regroupant ces résiduels. À partir de cela, nous pouvons introduire des propriétés sur les transformations assurant une représentation possible sous la forme d'un transducteur séquentiel, puis prouver ces résultats par l'adaptation du théorème de Myhill-Nerode à notre classe de transduction.

Il reste à adapter cette même approche pour l'apprentissage, non plus à partir d'une transformation complète en notre possession, mais à partir d'un échantillon d'exemples et d'un automate représentant son domaine. Nous limitant au modèle séquentiel, toute transformation et tout échantillon d'exemples n'est pas représentable par un transducteur séquentiel. Notre algorithme doit pouvoir échouer si les exemples ne peuvent être représentés par un transducteur séquentiel. En adaptant chaque étape, de la décomposition à l'identification des classes d'équivalences, l'algorithme s'assure en temps polynomial de l'existence d'un transducteur séquentiel cohérent avec l'entrée, et produit le transducteur canonique correspondant, ou le cas échéant échoue. À l'aide de la notion d'échantillon caractéristique, nous prouvons cependant que cet algorithme n'échoue pas dans les cas où l'entrée est suffisante pour aboutir à un transducteur cible représenté par cet échantillon.

Plan de thèse

Après avoir rappelé les différentes notions nécessaires à la compréhension de cette thèse, nous rappellerons également les différents modèles d'automates permettant de reconnaître des langages d'arbres, ces modèles étant la base des transducteurs qui nous intéressent. Nous y présenterons également les différents résultats tels que la décidabilité de l'équivalence ou encore l'apprentissage, ce qui nous permettra à travers des modèles plus simples d'illustrer les approches que nous appliquerons par la suite sur les transducteurs.

Le deuxième chapitre nous permettra, à travers un survol des différentes classes de transductions existantes, de montrer les différents résultats existant dans le domaine, et où se placent les modèles que nous étudieront dans cet ensemble. Nous profiterons du troisième chapitre pour montrer une des facettes pratiques de la transformation à travers le langage XSLT, qui permet de représenter et effectuer la transformation de fichiers XML sous différents formats.

Les trois derniers chapitres présenteront les contributions de cette thèse. Le premier chapitre se concentre sur le problème d'équivalence, en améliorant la décidabilité pour les transducteurs d'arbres en mots. La réduction qui y est introduite permettra également d'apporter un autre regard sur les différentes classes de transductions d'arbres en mots évoquées. Les deux derniers chapitres présenteront les résultats centraux de cette thèse, à savoir la normalisation, minimisation et l'apprentissage des transducteurs séquentiels.

Chapitre 1
Automates

Avant de pouvoir parler de transducteurs, il est indispensable de poser les bases sur lesquelles ils se construisent. Après avoir défini formellement les différents modèles de données manipulés dans cette thèse, nous nous attarderons sur les automates, structure de base à partir desquelles les transducteurs sont définis. Pour cela, nous partirons des automates de mots, pour ensuite survoler les différents types d'automates permettant la manipulations d'arbres.

1.1 Mots

1.1.1 Mots et langages

Un *alphabet* est un ensemble fini et non vide de symboles. La taille d'un alphabet Σ est le nombre de symboles qu'il contient, noté $|\Sigma|$.

Un *mot* est une séquence possiblement vide de symboles appartenant à un alphabet Σ. On représente par ε le mot vide. L'étoile de Kleene est la clôture réflexive et transitive d'un langage, noté Σ^* pour l'alphabet Σ. Σ^* est l'ensemble de tout les mots définissable à partir de l'alphabet Σ.

On note $u_1 \cdot u_2$ la concaténation de deux mots u_1 et u_2. La taille d'un mot u est le nombre de symboles qui le compose, noté $|u|$. Les mots sont liés entre eux par une relation d'ordre totale $<_{ord}$, telle que $u <_{ord} v$ ssi $|u| < |v|$ ou $u = v$ et u précède v dans l'ordre lexicographique. L'ensemble des positions $pos(u)$ de ce mot est composé d'entiers i, compris entre 1 et $|u|$, tel que $u(i)$ correspond à la i-ième lettre de ce mot. Les mots ou symboles pourront parfois être présentés entre apostrophes, comme "a", afin de les distinguer clairement de la phrase dans laquelle ils apparaissent.

Les *préfixes* d'un mot u sont l'ensemble des quotients à gauche de u par l'opération de concaténation. Les *suffixes* quand à eux sont l'ensemble des quotients à droite d'un mot. En d'autres termes, nous pouvons définir deux fonctions $prefixe(u)$ et $suffixe(u)$ associant à un mot de Σ^* un ensemble de mots des Σ^* respectivement préfixes et suffixes de u par :

$$prefixe(u) = \{v \mid v' \in \Sigma^*, u = v \cdot v'\} \quad suffixe(u) = \{v' \mid v \in \Sigma^*, u = v \cdot v'\}$$

Un *langage de mots* sur Σ est un ensemble de mots sur ce même alphabet. Il peut être *vide*, dénoté $\varnothing$, *universel* sur Σ, dénoté Σ^*, i.e. contenant l'ensemble de tous les mots sur Σ. L'union de deux langages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$ est l'ensemble $\mathcal{L}_1 \cup \mathcal{L}_2 = \{u \mid u \in \mathcal{L}_1 \text{ ou } u \in \mathcal{L}_2\}$. L'intersection de $\mathcal{L}_1$ et $\mathcal{L}_2$ est l'ensemble $\mathcal{L}_1 \cap \mathcal{L}_2 = \{u \mid u \in \mathcal{L}_1, u \in \mathcal{L}_2\}$.

L'opérateur de concaténation est étendu aux langages en concaténant les mots des langages respectifs, $\mathcal{L}_1 \cdot \mathcal{L}_2 = \{u \cdot v \mid u \in \mathcal{L}_1, v \in \mathcal{L}_2\}$. On définit de même la concaténation d'un mot à un langage en concaténant ce mot à chaque mot du langage. Cette concaténation peut être préfixe, $u \cdot \mathcal{L} = \{u \cdot v \mid v \in \mathcal{L}\}$, ou suffixe, $\mathcal{L} \cdot u = \{v \cdot u \mid v \in \mathcal{L}\}$. Par $\mathcal{L}^i$ on représente la concaténation de $\mathcal{L}$ avec lui même i fois, $\mathcal{L} \cdot \mathcal{L} \cdot \ldots \cdot \mathcal{L}$. A partir de cela, l'étoile (de Kleene) d'un langage $\mathcal{L}$ est l'ensemble $\mathcal{L}^* = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} \mathcal{L}^i$.

Nous pouvons étendre aisément la notion de préfixes (et suffixes) aux langages par l'union des préfixes (resp. suffixes) des mots qui le compose :

$$\mathit{prefixe}(\mathcal{L}) = \bigcup_{u \in \mathcal{L}} \mathit{prefixe}(u) \quad \mathit{suffixe}(\mathcal{L}) = \bigcup_{u \in \mathcal{L}} \mathit{suffixe}(u)$$

Il est aussi nécessaire de pouvoir identifier le plus grand représentant des préfixes et suffixes communs au mots d'un langage. Par $\mathit{lcp}(\mathcal{L})$ et $\mathit{lcs}(\mathcal{L})$ (venant de « largest common prefix » et « largest common suffix ») nous identifions le plus grand préfixe et suffixe commun à tous les mots d'un langage $\mathcal{L}$:

$$\mathit{lcp}(\mathcal{L}) = \mathit{max}_{<_{ord}}(\bigcap_{u \in \mathcal{L}} \mathit{prefixe}(u)) \quad \mathit{suffixe}(\mathcal{L}) = \mathit{max}_{<_{ord}}(\bigcap_{u \in \mathcal{L}} \mathit{suffixe}(u))$$

On appelle *singleton* tout ensemble contenant un unique élément. Le *résiduel* (gauche) d'un langage $\mathcal{L}$ par un mot u, noté $u^{-1} \cdot \mathcal{L}$, est l'ensemble des continuations possibles du mot u dans $\mathcal{L}$, i.e. $v \in u^{-1} \cdot \mathcal{L}$ ssi $u \cdot v \in \mathcal{L}$. Le fait qu'un mot u n'appartienne pas à l'ensemble des préfixes de $\mathcal{L}$ implique que le résiduel associé $u^{-1} \cdot \mathcal{L} = \varnothing$. De façon analogue nous pouvons définir le résiduel droit $\mathcal{L} \cdot v^{-1} = \{u \mid u \cdot v \in \mathcal{L}\}$.

Dès lors une notion d'équivalence entre deux préfixes d'un langages peut naître, deux mots u et v étant équivalents pour un langage $\mathcal{L}$ si leurs résiduels respectifs sont égaux, i.e. $u^{-1} \cdot \mathcal{L} = v^{-1} \cdot \mathcal{L}$. L'ensemble des préfixes partageant un même résiduel forme une classe d'équivalence représentée par son plus petit représentant et une relation d'équivalence $\equiv_{\mathcal{L}}$.

Un langage est *clos* pour une opération si le résultat de l'application de cette opération sur tout élément du langage génère un résultat inclus dans ce même langage. Nous pouvons aussi dire que ce langage est *stable* pour cette opération. La classe des *langages réguliers de mots* est le plus petit ensemble de langages sur Σ, clos par union, concaténation, et étoile, contenant $\varnothing$,$\{\varepsilon\}$

et $\{a\}$ pour tout $a \in \Sigma$. De même, un langage est donc *clos* par préfixe si tout préfixe d'un mot du langage appartient au langage.

Les expressions rationnelles sont un système de formules décrivant un langage à partir de singletons et d'applications d'unions, concaténations et étoiles. Une *expression rationnelle* (ou régulière) e sur Σ, et le langage $\mathcal{L}(e)$ qu'elle représente, sont définis inductivement par :

- $e = \varnothing$ est une expression rationnelle correspondant au langage $\mathcal{L}(e) = \varnothing$,
- $e = \varepsilon$ est une expression rationnelle correspondant au langage singleton $\mathcal{L}(e) = \{\varepsilon\}$,
- $\forall a \in \Sigma$, a est une expression rationnelle correspondant au langage singleton $\mathcal{L}(e) = \{a\}$,
- si e_1 et e_2 sont des expressions rationnelles sur Σ alors $e_1 \cdot e_2$, $e_1 + e_2$, e_1^* et e_2^* le sont aussi et correspondent respectivement aux langages $\mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$, $\mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, $\mathcal{L}(e_1)^*$ et $\mathcal{L}(e_2)^*$.

La taille d'une expression e, noté $|e|$, est définie par le nombre d'opérateurs et de symboles qui la composent. Pour plus de lisibilité, l'opérateur de concaténation sera parfois omis mais sera toujours compté dans la taille de cette expression.

Exemple 2. *Prenons l'exemple d'un langage $\mathcal{L}$ sur $\Sigma = \{a, b\}$ composé de tous les mots contenant un nombre pair, possiblement nul, de "a". C'est un langage régulier puisqu'il peut être exprimé par l'expression rationnelle $b^* \cdot (a \cdot b^* \cdot a \cdot b^*)$. Chaque mot de ce langage est composé d'une répétition de "b" à laquelle nous pouvons concaténer une répétition possiblement vide, de mots contenant deux "a".*

Comme tout ensemble, il est stable par union, ainsi que par intersection. Il est cependant stable par concaténation ce qui n'est pas le cas de tous les langages rationnels. En effet, nous pouvons vérifier que $\mathcal{L} \cdot \mathcal{L} = \mathcal{L}$. Pour vérifier cela il faut que tout mot de $\mathcal{L}$ puisse être exprimé par une concaténation de mots du même langage, ce qui est le cas puisque $\varepsilon \in \mathcal{L}$ et que $\forall u \in \mathcal{L}$ nous avons bien $\varepsilon \cdot u \in \mathcal{L}$. Il faut aussi que toute concaténation possible de deux mots u et v appartenant à $\mathcal{L}$ reste dans ce langage. Si u et v possèdent chacun un nombre pair de "a", leur concaténation aussi, est donc comprise dans $\mathcal{L}$.

Pour illustrer le fait que tout langage n'est pas stable par concaténation nous n'avons qu'a considérer le langage $\mathcal{L}'$ des mots contenant un nombre impair de "a". Si nous prenons le mot $u = a$ contenu dans ce langage, la concaténation $u \cdot u = aa \notin \mathcal{L}'$.

Un langage régulier peut être aussi nommé *langage rationnel* puisqu'un tel langage peut être identifié à l'aide d'expressions rationnelles.

1.1.2 Grammaires

Définition 1. *Une* grammaire algébrique *(« Context-Free Grammar » en anglais), noté* CFG*, sur un alphabet* Σ *est un tuple* $G = (\Sigma, Q, init, rul)$ *où* Σ *est l'alphabet des symboles terminaux,* Q *l'ensemble des symboles non-terminaux, ou états,* $init \subseteq Q$ *l'ensemble des états initiaux, ou axiomes, et* $rul \subseteq Q \times (\Sigma \cup Q)^*$ *l'ensemble de ces règles.*

On représente une règle r par $q \to u$ pour laquelle $q \in Q$ est la partie gauche, $gch(r)$, et $u \in Q^* \cup \Sigma$ la partie droite, $dt(r)$.

Par G_q on représente la grammaire obtenue à partir d'une grammaire $G = (\Sigma, Q, q_0, rul)$ en débutant à l'état q ce qui nous donne $G_q = (\Sigma, Q, q, rul)$.

La taille d'une règle r, $|r|$, est la taille de sa partie droite. La taille d'une grammaire G est la somme du nombre d'états, de symboles de l'alphabet et de la taille de chacune de ces règles i.e. $|G| = |\Sigma| + |Q| + \sum_{r \in rul} |r|$.

Définition 2. *Un langage* algébrique *est un langage de mots exprimable par une CFG.*

Exemple 3. *Les langages algébriques contiennent strictement les langages réguliers. L'exemple classique est le langage* $a^n \cdot b^n$ *sur* $\Sigma = \{a, b\}$ *pour* n *un entier quelconque. Il n'est pas possible de l'exprimer par une expression régulière mais la grammaire* G_3*, ayant pour unique état et état initial* q*, et pour règles* $q \to a \cdot q \cdot b$ *et* $q \to \varepsilon$*, le représente.*

Pour représenter l'exécution d'une grammaire G pour un mot $w \in \mathcal{L}_G$, nous pouvons utiliser une structure en forme d'arbre, structure sur laquelle nous reviendrons en détails par la suite, dit *arbre syntaxique*, gardant une trace de chaque non terminal utilisé.

Définition 3. *Un* arbre syntaxique *(ou de dérivation) est un arbre permettant de représenter la syntaxe d'un mot reconnu par une CFG. Chaque noeud interne est un symbole non terminal de cette grammaire, et chaque feuille un symbole terminal.*

Une grammaire est dite *non ambiguë* si elle ne permet pas deux arbres syntaxiques différents pour un même mot. Par exemple, la grammaire G_3 n'est pas ambiguë puisqu'il n'existe qu'une règle permettant la production, chaque mot ne peut être qu'une application répétée de cette règle.

Comme l'illustre l'exemple 3, les *CFG* contiennent strictement les langages réguliers, Il est cependant possible de restreindre les grammaires pour reconnaître uniquement les langages réguliers.

Définition 4. *Une* grammaire régulière *(dite aussi linéaire) est une grammaire dont toutes les règles sont sous l'une des forme suivantes :*

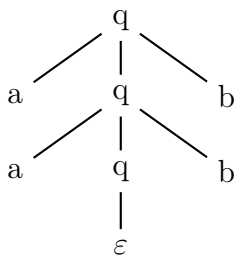


FIGURE 1.1 – Arbre syntaxique de "*aabb*" par G_3

- $q \to a \cdot p$
- $q \to \varepsilon$
- $q \to a$

Nous pouvons remarquer que les ε-productions, règles aboutissant à ε, peuvent être supprimées d'une grammaire G, sauf si $\varepsilon \in \mathcal{L}_G$, dans quel cas il est nécessaire de garder une unique règle $q_0 \to \varepsilon$, où q_0 étant un état initial non accessible par d'autres règles.

1.1.3 Automates

1.1.3.1 Définitions

Les automates de mots, machines à états finis, sont une autre manière de reconnaitre des ensembles (langages) de mots.

Définition 5. *Un* automate de mots A *sur* Σ *est défini par un tuple* $A = (Q, init, fin, rul)$*, où* Q *est un ensemble fini d'*états *avec* $init \subseteq Q$ *et* $fin \subseteq Q$ *respectivement les ensembles finis d'états* initiaux *et* finaux*, et* $rul \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ *un ensemble fini de règles (ou transitions).*

Les règles d'un automate sont représentées par $q \xrightarrow{a} p$ et $q \xrightarrow{\varepsilon} p$ respectivement pour $(q, a, p) \in rul$ et $(q, \varepsilon, p) \in rul$. Les règles de la forme $q \xrightarrow{\varepsilon} p$ sont appelées ε-règles (ou ε-transitions). La notation $\xrightarrow{\varepsilon}{}^*$ correspond à la clôture transitive de la relation $\xrightarrow{\varepsilon}$.

On dit d'une règle qu'elle *entre* dans un état si l'état est celui atteint à la fin de l'application de cette règle, et qu'elle en *sort* si l'état est celui à partir duquel il est appliqué. Par $lect(r)$ nous dénotons le symbole lu par la règle r, par $gch(r)$ l'état à gauche par lequel débute cette règle et par $dt(r)$ l'état atteint par cette règle.

Nous pouvons également représenter l'intégralité d'un automate sous forme d'un graphe orienté dont chaque noeud est étiqueté par un état, et chaque arête par un symbole de l'alphabet, ou ε.

$$Q = \{q_0, q_1\}$$
$$init = fin = \{q_0\}$$
$$q_0 \xrightarrow{a} q_a \quad q_0 \xrightarrow{b} q_0$$
$$q_a \xrightarrow{b} q_a \quad q_a \xrightarrow{a} q_0$$

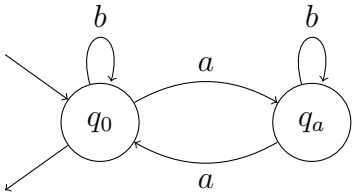


FIGURE 1.2 – Automate de mots sur $\{a, b\}$ et sa représentation. Le fait que q_0 est initial et final est représenté respectivement par la flèche entrante et sortante. L'automate reconnait l'ensemble des mots contenant un nombre pair de "a".

L'état d'un automate A est *accessible* s'il est atteignable depuis un état initial, en suivant un chemin sur la représentation de l'automate. De même, il est dit *co-accessbile* si un chemin part de cet état pour atteindre un état final. Seuls les états à la fois accessibles et co-accessibles sont utiles dans un automate. On appelle automate *émondé* un automate dont chaque état est à la fois accessible et co-accessible.

On dénote par A_q l'automate partageant la même signature que A dont l'ensemble des états initiaux est substitué par le singleton $\{q\}$.

Chaque automate A définit une fonction d'évaluation $eval_A : \Sigma^* \to (Q_A)^*$. Nous pouvons la définir à partir de fonctions $eval_{A_q} : \Sigma^* \to (Q_A)^*$ évaluant des mots à partir de chaque état $q \in Q_A$. Les résultats de ces fonctions se définissent pour tout état q et tout mot $u \in \Sigma$, par les plus petits ensembles tels que :

$$q \in eval_{A_q}(\varepsilon)$$
$$p \in eval_{A_q}(u) \text{ si } \begin{cases} p' \in eval_{A_q}(u) \text{ et } p' \xrightarrow{\varepsilon}{}^* p \in rul \\ u = u' \cdot a, p' \in eval_{A_q}(u'), p' \xrightarrow{a} p \in rul \end{cases}$$

Chaque évaluation de mot dans un automate commençant par un état initial,

$$eval_A(u) = \{q \mid q \in eval_{A_p}, p \in init_A\}$$

.

Un mot $u \in \Sigma^*$ est *reconnu* par un automate A si une de ces évaluations atteint un état final, i.e. $evalA(u) \cap fin_A \neq \varnothing$. Ainsi le langage reconnu par l'automate A, dénoté $\mathcal{L}(A)$ est l'ensemble des mots respectant cette propriété :

$$\mathcal{L}(A) = \{u \mid u \in \Sigma^*, eval_A(u) \cap fin_A \neq \varnothing\}$$

.

Pour les automates, $\mathcal{L}(A)$ peut aussi être écrit $[\![A]\!]$ représentant la *sémantique* de l'automate. Pour tout état q, A_q étant un automate à part entière, nous pouvons définir son langage de la même manière, composé des mots reconnus à partir de l'état q.

Un automate est dit *déterministe* s'il autorise au plus une évaluation pour chaque donnée d'entrée. Cette restriction étant liée directement à l'exécution de l'automate, elle peut s'exprimer de manière syntaxique sur la définition de l'automate.

Définition 6. *Un automate de mots A est* déterministe *s'il n'autorise aucune ε-règle et s'il existe au plus une règle $q \xrightarrow{a} p \in rul$ pour tout état $q \in Q$ et tout symbole $a \in \Sigma$.*

Comme le montre le théorème de Rabin et Scott (1959), tout langage reconnu par un automate de mots non déterministe l'est aussi par un automate déterministe. En revanche, cette déterminisation se fait en temps exponentiel dans le nombre d'états de l'automate.

L'évaluation d'un mot étant unique sur un automate déterministe, il est possible de la représenter pour un mot $u \in \Sigma^*$ par un couple $u * \beta$, β étant une fonction partielle de $pos(u)$ vers rul, associant à chaque position dans le mot u une règle de l'automate. Une exécution $u * \beta$ est dite *valide* si la règle à la position $i \in pos(u)$ lit le bon symbole, $lect(\beta(i)) = u(i)$, et si elle est la continuation de la règle qui la précède, i.e. pour $i - 1 \in pos(u)$ nous vérifions $dt(\beta(i-1)) = gch(\beta(i))$. L'exécution est *complète* si elle est valide, si elle débute à un état initial et si elle atteint un état final, i.e. $gch(\beta(1)) \in init$ et $dt(k) \in fin$.

Le fait que sous une forme déterministe, il existe au plus un état atteint par l'évaluation d'un mot $u \in prefixe(\mathcal{L}(A))$, cela implique que $u^{-1} \cdot \mathcal{L}(A) = \mathcal{L}(A_q)$ pour $q = eval_A(u)$.

On appelle automate *complet* un automate évaluant l'intégralité des mots de Σ^*. Syntaxiquement cette propriété se traduit, pour tout symbole $a \in \Sigma$ et tout état $q \in Q$, par l'existence d'une règle $q \xrightarrow{a} p \in rul$. Cela peut s'obtenir par l'ajout d'une règle $q \xrightarrow{a} p_\perp$ s'il n'existe pas de règle à partir de q évaluant a, $q_\perp$ étant un état puits non co-accessible. Les seules règles sortant de $q_\perp$ sont de la forme $q_\perp \xrightarrow{a} q_\perp$ pour tout $a \in \Sigma$ et $q_\perp \notin fin$.

Les résultats de Kleene (1956) montrent que la classe des langages réguliers correspond exactement à celle des langages reconnus par des automates finis. La classe des langages reconnus par des automates de mots partagent donc les mêmes propriétés de clôture par union, intersection et complémentaire.

Le complément d'un automate A, dénoté A^{C}, est l'automate reconnaissant $\Sigma^* \setminus \mathcal{L}(A)$. Il est obtenu à partir d'un automate complet sans ε-transition, en

remplaçant l'ensemble des états finaux *fin* par $Q \setminus fin$. Cette opération garde le déterminisme et toute autre propriété de l'automate.

1.1.3.2 Equivalence et minimisation

Deux automates A et A' sont dit *équivalents* s'ils reconnaissent un même langage, $\mathcal{L}(A) = \mathcal{L}(A')$. Cette notion d'équivalence peut être facilement étendue aux états q et p d'un même automate A si les langages reconnus à partir de ces états sont égaux, $\mathcal{L}(A_q) = \mathcal{L}(A_p)$.

S'il existe la possibilité de définir pour chaque langage régulier un unique automate le représentant, tester l'équivalence de deux automates revient à vérifier que leurs représentants partagent une même définition modulo le changement de noms des états. Une manière d'identifier un représentant unique est de passer un automate dans sa version minimale déterministe. La minimisation d'un automate se fait par la minimisation du nombres d'états qui le compose. Pour tout automate déterministe, nous pouvons nous ramener à son représentant minimal en suivant l'algorithme d'Hopcroft (1971) en $O(n\ log(n))$ connu aussi sous le nom d'algorithme des parties que nous ne détaillerons pas ici. Nous pouvons remarquer que cet algorithme ne s'applique pas aux automates non déterministes, déterminisme pouvant être obtenu en temps exponentiel dans la taille de l'automate.

Le principal point qui nous intéresse ici est de montrer qu'en minimisant le nombre d'états, l'automate minimal obtenu est unique et peut être utilisé comme représentant canonique pour décider de l'équivalence mais aussi comme base nécessaire à un apprentissage.

1.2 Arbres d'arité bornée

Les arbres sont des structures de données récursives permettant d'ajouter aux mots une structure plus riche. Nous nous limiterons aux arbres finis, ordonnés et étiquetés. Nous pouvons distinguer deux principales familles d'arbres, les arbres d'arité bornée, où le nombre d'enfants de chaque noeud de l'arbre est fixé, et les arbres d'arité non bornée, ne limitant pas ce nombre.

Malgré le fait que les arbres d'arité non bornée soient plus fréquents dans la pratique, la littérature porte le plus souvent sur le modèle d'arité bornée permettant de simplifier la manipulation des arbres et leur représentation à travers des automates comme nous allons le présenter maintenant.

1.2.1 Définition

Un *alphabet d'arité bornée* peut être défini par un couple (Σ, ar) où Σ est un ensemble fini (non vide) de symboles et $ar : \Sigma \to \mathbb{N}$, une fonction associant à chaque symbole son arité. Une autre notation possible, que l'on rencontrera par la suite, associe directement à chaque symbole de Σ son arité sous forme d'indice. Ainsi l'alphabet $(\{a, b\}, ar)$ tel que $ar(a) = 2$ et $ar(b) = 0$ sera dénoté par l'ensemble $\{a^{(2)}, b^{(0)}\}$. Nous représenterons aussi par $\Sigma^{(i)}$ un ensemble de symboles d'arité i.

L'ensemble des *arbres d'arité bornée* $\mathcal{T}_\Sigma$ est le plus petit ensemble contenant tout les arbres $f(t_1, \ldots, t_k)$ tel que $f^{(k)} \in \Sigma$ et $t_i \in \mathcal{T}_\Sigma$. Formellement, un *arbre fini, ordonné et étiqueté* $t \in \mathcal{T}_\Sigma$ peut être vu comme un ensemble de positions, ou noeuds $noeuds(t)$ auxquels nous associons des étiquettes. Chaque noeud peut être identifié de manière unique par le chemin qui le rattache à la racine. L'ensemble $noeuds(t) \subseteq \mathbb{N}^*$ est l'ensemble fini non vide :

$$noeuds(f(t_1, \ldots, t_k)) = \{\varepsilon\} \cup \{i \cdot \pi \mid \pi \in noeuds(t_i)\}$$

Nous associons à chacun de ces noeuds son *étiquette* à l'aide d'une fonction $labels_t : noeuds(t) \to \Sigma$. La racine d'un arbre est le noeud identifié par ε. À part la racine, chaque noeud peut se noter de la forme $\pi \cdot i$, a un unique *père* π, et est le *fils* de π. Un noeud n'ayant pas de fils est appelé une *feuille*. Deux noeuds ayant un même père sont appelés *frères* et sont ordonnés entre eux. Pour tout $\pi \in noeuds(t)$, nous identifions par $\pi^{-1}t$ le sous arbre de t ayant pour racine le noeud à la position π.

Un *langage* d'arbres d'arité bornée $\mathcal{L}$ sur Σ est un sous ensemble de $\mathcal{T}_\Sigma$. Nous introduisons la notation à $\mathcal{T}_\Sigma(S)$ correspondant à $\mathcal{T}_{\Sigma \cup S}$ où S est un ensemble de symbole d'arité 0 permettant l'ajout de nouveaux symboles de feuilles dans $\mathcal{T}_\Sigma$.

Un *contexte* C sur Σ est un arbre défini sur $\mathcal{T}_\Sigma(x^{(0)})$ autorisant une unique feuille x non comprise dans Σ représentant un *trou* dans cet arbre. Par $C[t]$ nous identifions l'arbre obtenu en substituant la feuille x du contexte par l'arbre t correspondant. Dans les prochains chapitres nous pourrons avoir besoin d'une version étendue de ces contextes autorisant plusieurs trous, chacun présent une et unique fois. Ils seront définis sur $\mathcal{T}_\Sigma(\{x_1, \ldots, x_k\})$ et $C[t_1, \ldots, t_k]$ correspondra à l'arbre obtenu en substituant chacun des trous x_i du contexte par l'arbre t_i correspondant.

Le résiduel d'un langage d'arbres par un contexte C est l'ensemble des arbres $\{t \mid C[t] \in \mathcal{T}\}$. Cette notion de résiduel est trop faible, la sur-spécification en se limitant à ce contexte et non à l'ensemble des contextes possibles pour ce chemin restreint le nombre de résultats à un sous ensemble très restreint.

Pour généraliser cette définition, il faut se concentrer non plus sur le

contexte mais uniquement sur le chemin menant à ce noeud. Il faut pour cela introduire la notion de *chemin étiqueté*, ou *chemin*, identifiant un noeud comme précédemment en y ajoutant les étiquettes de ses ancêtres. Un chemin se définit sur $\bigcup_{k>0} \Sigma^{(k)} \times \{1, \ldots, k\}$. Le chemin ε mène à la racine. Si un chemin p mène à un noeud étiqueté par un f, alors $u \cdot (f, i)$ représente son i-ième fils. Par $chemins(t)$ nous dénotons l'ensemble des chemins étiquetés de t et par $chemins(\mathcal{T})$ l'ensemble des chemins des arbres qui composent $\mathcal{L}$.

Un langage d'arbres $\mathcal{L}$ est *clos par chemins* si $\mathcal{L} = \{t \in \mathcal{T}_\Sigma \mid chemins(t) \subseteq chemins(\mathcal{T})\}$. La classe des langages d'arbres clos par chemins peut être aussi définie par une *propriété d'échange* pour un chemin u. En effet, si nous prenons deux contextes C_1 et C_2 ayant un unique trou x en p, la définition de clôture par chemins impose que pour tous sous arbres t_1 et t_2 tels que $C_1[t_1], C_2[t_2] \in \mathcal{L}$ alors $C_1[t_2]$ et $C_2[t_1]$ appartiennent également à $\mathcal{L}$. Un langage d'arbres $\mathcal{L}$ est donc *clos par chemins* si la propriété d'échange est respecté pour tout chemin $u \in chemins(\mathcal{L})$.

Être *clos par chemins* implique que pour deux contextes C_1 et C_2 ayant tous deux un trou en p alors $C_1^{-1}\mathcal{L} = C_2^{-1}\mathcal{L}$. Dès lors, nous pouvons définir le résiduel d'un langage d'arbres $\mathcal{L}$ par un chemin p comme le résiduel de n'importe quel contexte C présent dans $\mathcal{L}$ dont le trou est en p.

Exemple 4. *Prenons un langage d'arbres simple :*

$$\mathcal{L} = \{f(a,a), f(a,b), f(b,a), f(b,b)\}$$

Ce langage est clos par chemin puisque toutes les possibilités de sous arbres sont représentées pour chaque chemin. Par exemple pour le chemin $f \cdot 1$ les contextes possibles sont $C_1 = f(x,a)$ et $C_2 = f(x,b)$ pour lesquels x peut être remplacé par a ou b, i.e. $C_1^{-1}\mathcal{L} = C_2^{-1}\mathcal{L} = \{a,b\}$. L'arbre $f(a,a)$, qui est aussi identifié par $C_1[a]$ a un ensemble de positions de noeuds : $noeuds(f(a,a)) = \{\varepsilon, 1, 2\}$. Ces noeuds sont respectivement accessibles par les chemins étiquetés ε, $f \cdot 1$ et $f \cdot 2$. Ce langage étant clos par chemin, le résiduel par le chemin $f \cdot 1^{-1}\mathcal{L}$ correspond au résiduel par n'importe quel contexte ayant un trou à cet emplacement, soit C_1 ou C_2.

La notion de taille pour un arbre est souvent rattachée au nombre de noeuds qui le composent, avant de s'intéresser à l'étiquetage de ces noeuds. De ce fait, la taille d'un arbre t, dénoté $|t|$ revient au nombre de noeuds qui le composent, i.e. la cardinalité de l'ensemble $noeuds(t)$. La notion d'ordre entre les arbres n'est quant à elle pas si triviale ni naturelle. En effet, même si la taille est un premier élément permettant d'établir cet ordre, elle ne permet pas de distinguer de nombreux arbres partageant un même nombre de noeuds. Même si il n'a pas de sémantique propre, l'existence d'un ordre global entre

les arbres peut s'avérer nécessaire. Dès lors, il faut choisir des propriétés arbitraires, n'ayant pas forcément une sémantique forte, permettant d'ordonner tout arbre. Si nous nous intéressons au *poids* d'un arbre, i.e. à la somme des tailles des étiquettes de chacun de ses noeuds, nous pouvons rapidement nous apercevoir que ce n'est pas encore suffisant. Prenons par exemple les arbres $f(a, b(a, a))$ et $f(b(a, a), a)$. Ces deux arbres ont un même nombre de noeuds et partagent les mêmes étiquettes, seul l'ordre change. Une solution possible est d'utiliser la linéarisation des arbres comme des mots.

Définissons pour un arbre t, la linéarisation, notée $lin_{Mot}(t)$, sous forme d'une concaténation des étiquettes qui la composent, séparés par des # . Plus formellement, $lin_{Mot}(t) = lin^{\#}_{Mot}(\text{LINEAR}(t))$ tel que :

$$lin^{\#}_{Mot}(\varepsilon) = \varepsilon \qquad lin^{\#}_{Mot}((,u) \cdot w) = \# \cdot u \cdot lin^{\#}_{Mot}(w)$$

Il suffit maintenant d'étendre l'ordre $<_{ord}$ à ces mots en ajoutant le symbole # à l'alphabet usuel, que nous considérerons être le plus petit symbole, ordre dénoté $<^{\#}_{ord}$.

Ainsi nous pouvons définir un ordre total $<_{\mathcal{T}}$ entre les arbres, tel que $t_1 <_{\mathcal{T}} t_2$ ssi $|t_1| < |t_2|$, ou $|t_1| = |t_2|$ et $lin_{Mot}(t_1) < lin_{Mot}(t_2)$. Par $\min_{Arbre}(T)$ nous dénotons l'arbre minimal selon $<_{\mathcal{T}}$ (T) d'un ensemble non vide d'arbres T. L'ordre $<_{\mathcal{T}}$ peut être appliqué aux contextes, puisque les contextes sont des arbres avec un noeud particulier x représentant un trou. On défini également $\min_{Ctx}(Cs)$ comme le plus petit contexte de l'ensemble non vide de contextes Cs suivant l'ordre $<_{\mathcal{T}}$. L'ordre entre les chemins, dénoté $<_{chemins}$, correspond à l'ordre total des mots $<_{ord}$ étendu aux paires (étiquette, indice), une paire (a, i) étant inférieure à (b, j) si $a <_{ord} b$, ou si $a = b$ et $i < j$.

1.2.2 Automates d'arbres

Avant de nous intéresser aux transducteurs, centraux dans cette thèse, il est important de prendre le temps de poser de bonnes bases sur la notion d'automates. Possédant une structure souvent proche de celles des transducteurs, mais ne devant pas gérer de productions, les automates peuvent être vus comme une simplification des transducteurs. Comprendre les problèmes rencontrés par ces machines et les complexités associées sont donc indispensables avant d'ajouter les opérations liées à la production telles que les copies, les réordonnancement, que l'on rencontre dans certaines classes de transduction (Lemay *et al.*, 2010; Alur et D'Antoni, 2012).

Il existe deux types d'automates d'arbres d'arité bornée qui se distinguent par le sens dans lequel ils parcourent l'arbre. Les automates dit *descendants* parcourent l'arbre de la racine jusqu'à ses feuilles alors que les automates *ascendants* remontent des feuilles jusqu'à la racine d'un arbre. Nous verrons que

même s'ils partagent une syntaxe très proche, en imposant le déterminisme, les modèles n'ont pas la même expressivité.

1.2.2.1 Automates ascendants

Définition 7. *Un* automate d'arbres ascendant *de $\mathcal{T}_\Sigma$ est un triplet $A = (Q, fin, rul)$ composé d'un ensemble fini d'états Q, d'un ensemble d'états finaux $fin \subseteq Q$, et d'un ensemble fini de règles $rul \subseteq (\bigcup_{k\leq 0}\{f^{(k)}\} \times Q^{k+1}) \cup (Q^2)$. Ces règles sont représentées par $f(q_1, \ldots, q_k) \to q$ pour $(f^{(k)}, q, q_1, \ldots, q_k)$, et $q \xrightarrow{\varepsilon} q'$ pour $(q, q') \in rul$.*

Les ε-transitions restent identiques à celle des automates de mots, partageant une même clôture transitive représentée par $q \xrightarrow{\varepsilon}{}^{*} p$.

A l'instar des automates de mots, une *fonction d'évaluation* $eval_A$ peut être définie pour ces automates. Un automate ascendant peut être vu comme une machine associant un ensemble d'états aux feuilles d'un arbre, puis progressivement aux noeuds internes en tenant compte des états de ses fils, jusqu'à atteindre la racine. L'intégralité d'un sous arbre est parcourue par l'automate avant d'associer un état à sa racine. Ainsi pour un automate ascendant $A = (Q, fin, rul)$, $eval_A : \mathcal{T}_\Sigma \to (Q)^*$ tel que :

$$eval_A(f(t_1, \ldots, t_k)) = \{q \mid q_i \in \quad eval_A(t_i) \text{ pour } 1 \leq i \leq k, \\ f(p_1, \ldots, p_k) \to q' \text{ et } q' \xrightarrow{\varepsilon}{}^{*} q \in rul\}$$

Un arbre t est *accepté* par un automate ascendant A si $eval_A(t) \cap fin \neq \varnothing$. Le langage $\mathcal{L}(A)$ des arbres reconnus par un automate A est composé de l'ensemble des arbres acceptés par cet automate.

Définition 8. *Un automate d'arbres est* déterministe ascendant*, noté d*TA$^{\uparrow}$*, s'il n'accepte aucune ε-règle et s'il existe au plus une règle pour une partie gauche donnée.*

L'évaluation d'un sous arbre est maintenant au plus un singleton.

Il est *complet* si chaque partie gauche possible se retrouve dans une de ces règles. L'expressivité reste la même puisque tout automate d'arbres non déterministe peut être rendu déterministe ascendant en temps exponentiel (Thatcher, 1973).

L'ensemble des langages d'arbres reconnaissables par un automate défini la classe des *langages réguliers d'arbres*, résultat que nous pouvons retrouver, parmis les autres résultats principaux sur les automates, dans le livre « Tree Automata Techniques and Applications » (Comon *et al.*, 2007b).

Exemple 5. *Prenons l'automate A_5 reconnaissant le langage $\mathcal{L}(A_5) = \{f(a, a), f(b, b)\}$ composé de deux arbres dont le sous arbre gauche est identique au sous arbre*

droit, pour une même racine binaire f. Il est composé de trois états $Q_{A_5} = \{q_a, q_b, q_f\}$ dont uniquement q_f est final. En regardant ces règles, présentées dans la figure 1.3, nous pouvons voir qu'il est déterministe (aucune ne partageant une même partie droite) mais pas complet. A_5 étant déterministe, son exécution sur un arbre peut être représentée, comme l'illustre la figure 1.3, par une annotation de ses noeuds, chaque état correspondant à l'évaluation du sous arbre dont il annote la racine.

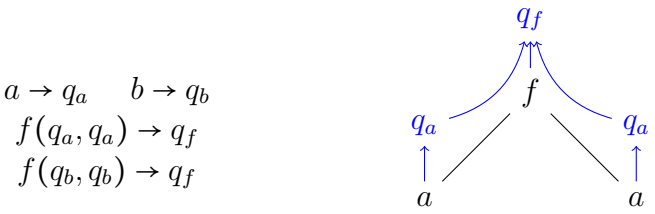


FIGURE 1.3 – Automate A_5 ascendant déterministe et son exécution représentée en bleu.

L'exécution d'un automate ascendant déterministe peut être représentée, d'une manière similaire aux mots, par une paire $t * \beta$, β associant à chaque noeud au plus un état. Nous pouvons nous baser sur la définition globale d'évaluation pour définir cette fonction, de sorte que pour tout noeud $\pi \in noeuds(t)$, $\beta(\pi) = q$ ssi $eval_A(\pi^{-1}t) = \{q\}$.

1.2.2.2 Automates descendants

Les automates descendants partagent une signature identique à celles des automates ascendants, et diffèrent dans leur sémantique et leur représentation. L'ensemble *fin* des états sur lesquels nous terminions dans l'approche ascendante est remplacé par un ensemble *init* d'états initiaux.

Définition 9. *Un* automate d'arbres descendant *de $\mathcal{T}_\Sigma$ est un triplet $A = (Q, init, rul)$ composé d'un ensemble fini d'états Q, d'un semble d'états initiaux $init \subseteq Q$, et d'un ensemble fini de règles $rul \subseteq (\bigcup_{k\leq 0}\{f^{(k)}\} \times Q^{k+1}) \cup (Q^2)$. Ses règles sont représentés par $q \xrightarrow{f} (q_1, \ldots, q_k)$ pour $(f^{(k)}, q, q_1, \ldots, q_k)$,et $q \xrightarrow{\varepsilon} q'$ pour $(q, q') \in rul$. Pour les constantes, symboles d'arité 0, on représente la règle par $q \xrightarrow{a}$ pour $(a, q) \in rul$.*

Les automates ascendants et descendant partagent une même expressivité et de mêmes propriétés, le non déterminisme permettant de passer de l'un à l'autre. La différence sémantique se fait ressentir lors du passage à un modèle déterministe.

Définition 10. *Un automate d'arbres descendant est* déterministe*, noté d*TA$^{\downarrow}$*, s'il n'accepte aucune ε-règle, s'il n'a qu'un état initial, et s'il existe au plus une règle pour une partie gauche donnée.*

Comme vu précédemment, les automates ascendants tiennent compte, pour choisir l'état associé à un noeud, de l'intégralité du sous arbre dont il est la racine. Les automates descendants déterministes, quant à eux, ne tiennent compte que des ancêtres de ce noeud. L'état ne dépend que du chemin qui relie ce noeud à la racine de l'arbre. Cette restriction d'informations aux chemins rend les automates descendants strictement moins expressifs que les automates ascendants (déterministes ou non).

Pour illustrer cette perte d'expressivité il suffit de regarder l'exemple 5 qui ne peut pas être exprimé par un automate déterministe descendant. En effet, l'automate ne pourrait pas tenir compte du sous arbre droit pour décider de l'annotation du sous arbre gauche, et inversement.

Pour les dTA$^{\downarrow}s$, l'ensemble des règles peut être vu comme une fonction partielle $\hat{rul} : Q \times \Sigma \to Q^*$ associant à un état et un symbole un ensemble d'états. Cette fonction peut être étendue aux chemins étiquetés. Connaître le chemin reliant tout noeud de l'arbre à la racine suffit à décider de l'état à lui associer. Cela peut se définir de manière récursive par :

$$\hat{rul}(q, \varepsilon) = q, \qquad \hat{rul}(q, (f, i) \cdot p) = q_i \text{ pour } \hat{rul}(q, f) = q_1 \cdot \ldots \cdot q_i \cdot \ldots \cdot q_k.$$

La classe des langages reconnus par les automates d'arbres descendants déterministes correspond à la classe des langages clos par chemins (Virágh, 1980; Comon *et al.*, 2007b).

Une solution pour décider de l'équivalence est le passage par le représentant minimal unique reposant une fois de plus sur les classes d'équivalences, ou classes de Myhill-Nerode, liant les états, ou résiduels de chemins pour les langages reconnus. Par la suite on nommera *indice de Myhill-Nerode* le nombre de classes d'équivalences distinctes.

La relation d'équivalence de Myhill-Nerode $\equiv_{\mathcal{L}}$ se définit sur les chemins annotés, de sorte que $p_1 \equiv_{\mathcal{L}} p_2$ ssi $p_1^{-1}\mathcal{L} = p_2^{-1}\mathcal{L}$. La classe d'équivalence d'un chemin p est l'ensemble des chemins $[p]_{\mathcal{L}} = \{p' \in \mathit{chemins}(\mathcal{L}) \mid p' \equiv_{\mathcal{L}} p\}$ et l'indice de Myhill-Nerode de $\mathcal{L}$ est le nombre de classes d'équivalences différentes (possiblement infinies).

Théorème 1. *(Myhill-Nerode pour les dTA$^{\downarrow}s$) Un ensemble d'arbres $\mathcal{L}$ est clos par chemins et un indice de Myhill-Nerode fini si et seulement s'il est reconnu par un automate d'arbres déterministe descendant.*

Cette notion de résiduels et de classes d'équivalences sur les chemins nous permet de définir un dTA$^{\downarrow}$ *minimal canonique* reconnaissant un langage $\mathcal{L}$ clos par chemins.

Définition 11. *Le* dTA$^{\downarrow}$ *minimal canonique* $A_{\mathcal{L}}$ *représentant un langage* $\mathcal{L}$*, clos par chemins, et d'indice de Myhill-Nerode fini, est défini par un quadruplet* $A_{\mathcal{L}} = (\Sigma, Q, [\varepsilon]_{\mathcal{L}}, rul)$*, pour lequel l'ensemble des états* Q *correspond à l'ensemble des classes d'équivalences* $\{[p]_{\mathcal{L}} \mid p \in chemins(\mathcal{L})\}$ *de* $\mathcal{L}$ *et dont chaque transition est de la forme :*

$$[p]_L \xrightarrow{f} ([p \cdot (f,1)]_{\mathcal{L}}, \ldots, [p \cdot (f,1)]_{\mathcal{L}})$$

pour tout chemin $p \in chemins(\mathcal{L})$*,* $\mathcal{L}$ *ayant un arbre* t *avec le symbole* f *d'arité* k *en* p*.*

Naturellement $[\![A_{\mathcal{L}}]\!] = \mathcal{L}$, et $A_{\mathcal{L}}$ est minimal, le nombre d'états ne pouvant pas être réduit. Réduire sa taille reviendrait à fusionner deux classes d'équivalences différentes.

1.3 Arbres d'arité non-bornée

Intéressons nous maintenant aux arbres dont l'arité d'un symbole n'est pas bornée, comme nous pouvons le trouver dans la plupart des formats de données utilisées. La liberté offerte par l'absence de borne sur le nombre de fils que peut avoir le noeud d'un arbre pose un réel problème sur la représentation et manipulation du modèle théorique. Dans cette section nous présenterons les diverses représentations et encodages permettant de les manipuler efficacement.

1.3.1 Definition

Soit Σ un alphabet fini possiblement composé de symboles sans arité associée. L'ensemble $\mathcal{T}_{\Sigma}^{u}$ des arbres d'arité non bornée est le plus petit ensemble qui contient Σ et tout tuple $a(t_1, \ldots, t_n)$ où $n \geq 0$ tel que $t_1, \ldots, t_n \in \mathcal{T}_{\Sigma}^{u}$ soient des arbres d'arité non-bornée.

1.3.2 Comment définir les automates ?

La recherche de bonnes notions d'automates pour les arbres d'arité non-bornée peut se révéler complexe (comme le chapitre 8 du livre de Comon *et al.* (2007b) peut l'illustrer). Tout particulièrement, avoir de bonnes intuitions sur la notion de déterminisme de ces machines peut rapidement poser problème (Martens et Niehren, 2007). Pour manipuler ces arbres d'arité non bornée, deux types de solutions s'offrent à nous. La première, que nous verrons dans cette section, consiste à passer par l'encodage des arbres d'arité non bornée

dans des arbres binaires. Les automates standards peuvent alors être appliqués. La deuxième solution, qui sera développée dans la section 1.4, passe par la linéarisation des arbres en mots imbriqués, séquences de balises ouvrantes et fermantes représentant les noeuds d'un arbre et leurs contenus.

L'intérêt de passer par des codages en arbres binaires est de pouvoir appliquer les résultats déjà existants dans la littérature, pour les automates (et transducteurs) ascendants et descendants déterministes, sur les arbres d'arité bornée. Les codages choisis dépendent principalement des automates et transducteurs que nous souhaitons appliquer par la suite.

Pour identifier les approches possibles, il est intéressant de ce concentrer sur la manière par laquelle il nous est possible de parcourir un arbre. Comme nous avons pu le voir précédemment, deux parcours s'offrent à nous, le parcours ascendant et le parcours descendant. Les informations associées à un noeud lors de l'exécution d'automates étant différentes selon l'ordre de parcours, il est nécessaire d'avoir à dispositions des encodages tenant compte de ces spécificités. Nous présentons ici le codage curryfié, adapté à un parcours ascendant, et le codage frère-fils, qui lui est destiné à un parcours descendant.

1.3.3 Codage binaire curryfié (ascendant)

L'idée du codage *curryfié* lui même nous vient du lambda-calcul (Curry et Feys, 1958), mais n'avait pas été adapté aux arbres avant les travaux de Carme *et al.* (2004a). Dans le domaine des langages fonctionnels, la curryfication correspond à la décomposition d'une fonction à plusieurs paramètres en une succession d'applications de fonctions à un seul paramètre. Une fonction $f(a,b,c)$ sera remplacée par sa version curryfiée f' tel que $f(a,b,c) = ((f'(a))b)c$. A la place d'une fonction possédant un nombre non fixée de paramètres, nous avons une succession de fonctions à un seul argument.

La curryfication est étendue aux arbres, considérant chaque noeud comme une fonction, l'ensemble de ses fils étant ses paramètres, eux même des fonctions. Ainsi, à l'aide d'un opérateur @, nommé *opérateur d'extension*, $f(a,b,c)$ donne l'arbre binaire $f@a@b@c$ dans sa version infixée. L'opérateur @ est prioritaire à gauche, l'arbre représenté par l'exemple précédent est donc l'arbre $@(@(@(f,a),b),c)$ dans sa version préfixée. Plus formellement, la curryfication est la bijection $curry : \mathcal{T}^u_\Sigma = \mathcal{T}_{\Sigma^{(0)} \cup \{@^{(2)}\}}$ définie par :

$$curry(f(t_1, \dots t_k, t)) = @(curry(f(t_1, \dots, t_k), curry(t)))$$
$$curry(a) = a$$

Exemple 6. *La figure 1.4 illustre un exemple d'encodage curryfié d'un arbre avec plusieurs niveaux. Nous considérons la formule de logique propositionnelle* $t = or(and(...), and(...), false)$ *comme un arbre d'arité non-bornée avec*

la signature $\{and, or, true, false\}$*. Comme tel, nous pouvons exprimer des conjonctions et disjonctions avec un nombre non-borné d'arguments. Le codage curryfié* $curry(t) = or@(and@...)@(and@...)@false$*, par contre, applique les conjonctions et disjonctions une à une de gauche à droite. Dans cette figure la correspondance des noeuds entre l'arbre original et son codage curryfié est indiqué par des lignes rouges. L'exécution de l'automate présenté dans l'exemple suivant, sur le codage et sa pré-image correspondante sur l'arbre de base, sont indiqués par les annotations bleues des noeuds par les états de l'automate.*

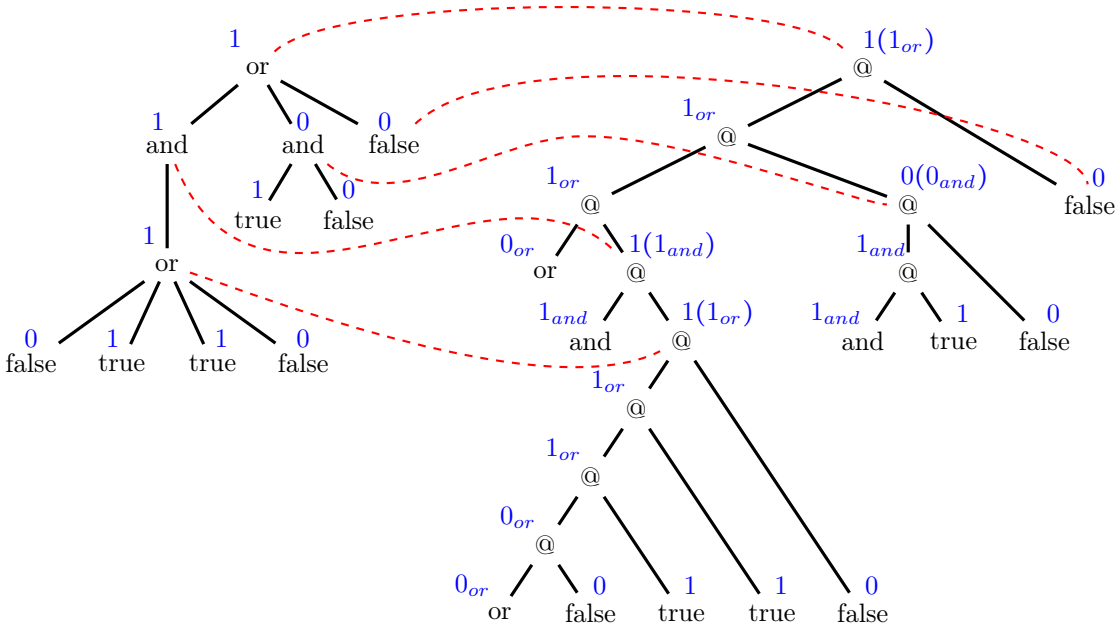


FIGURE 1.4 – L'encodage frère-fils.

Nous pouvons maintenant identifier des arbres d'arité non-bornée à l'aide d'automates d'arbres standards reconnaissant le codage curryfié de ces arbres. Les automates d'arbres travaillant sur cet encodage sont appelés *stepwise*.

Définition 12. *Un* automate d'arbres stepwise *(Carme* et al.*, 2004b) sur* Σ *est un automate d'arbres ascendant sur* $\Sigma_@ = \Sigma \cup \{@\}$ *ou chaque symbole de* Σ *est vu comme un symbole d'arité* 0*. On note* $a \cdot q$ *les règles constantes pour les constantes de* Σ*, et* $q_1@q_2 \to q$*, au lieu de* $@(q_1, q_2) \to q$*, les règles binaires.*

Les règles d'un automate stepwise peuvent être représentés graphiquement d'une autre manière. Une règle binaire $q_1@q_2 \to q$ y est représentée par une arête allant de l'état annotant le premier fils, q_1, à l'état annotant le noeud, q, étiquetée par l'état obtenu sur le deuxième fils, q_2. Les règles constantes,

quant à elles, sont des arêtes entrant dans l'automate par l'état correspondant au symbole annotant cette arête.

Exemple 7. *Par exemple, considérons l'évaluation des formules logiques de la figure 1.4, après codage curryfié. Il nous faut un automate d'arbre ascendant sur la signature* $\{@^{(2)}, and^{(0)}, or^{(0)}, false^{(0)}, true^{(0)}\}$. *Nous avons besoin des 6 états* 1_{and}, 0_{and}, 1_{or}, 0_{or}, 1, *et* 0.

Les états 0 *et* 1 *représentent la valeur booléenne du sous arbre parcourus. Atteindre un état* 1_{and} *indique qu'une conjonction (and) non-bornée est en cours d'évaluation, dont la valeur actuelle est vraie* (1). *Les* ε*-transitions «* $1_{and} \xrightarrow{\varepsilon} 1$ *» et «* $0_{and} \xrightarrow{\varepsilon} 0$ *» permettent de terminer l'évaluation d'une conjonction en associant maintenant au sous arbre sa valeur finale. la transition «* $and \to 1_{and}$ *» permet d'amorcer une disjonction par la valeur «vrai». La sémantique des autres états est analogue. Les autres règles de l'automate correspondent aux lignes des tables de vérité des opérateurs logiques.*

$$
\begin{array}{llll}
0_{and} \xrightarrow{\varepsilon} 0, & 1_{and} \xrightarrow{\varepsilon} 1, & 0_{and}\ @\ 0 \to 0_{and}, & 0_{or}\ @\ 0 \to 0_{or} \\
0_{or} \xrightarrow{\varepsilon} 0, & 1_{or} \xrightarrow{\varepsilon} 1, & 0_{and}\ @\ 1 \to 0_{and}, & 0_{or}\ @\ 1 \to 1_{or}, \\
and \to 1_{and} & or \to 0_{or} & 1_{and}\ @\ 0 \to 0_{and}, & 1_{or}\ @\ 0 \to 1_{or}, \\
true \to 1, & false \to 0, & 1_{and}\ @\ 1 \to 1_{and}, & 1_{or}\ @\ 1 \to 1_{or}.
\end{array}
$$

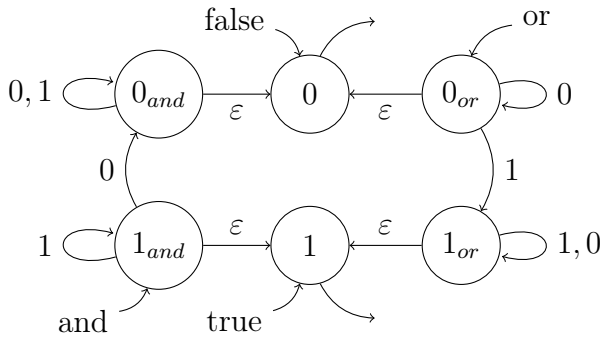


FIGURE 1.5 – Automate stepwise pour l'exemple de la figure 1.4

L'opération de curryfication étant une bijection, (illustré dans la figure 1.4 par les liens en pointillés), l'exécution d'un automate sur le codage d'un arbre peut correspondre à une annotation de l'arbre d'arité non-bornée correspondant. Il est également possible de traduire tout automate ascendant dans un automate stepwise sur son encodage. En revanche, Cette réécriture de l'automate préserve le déterminisme ascendant. En revanche, le déterminisme descendant ne sera pas préservé.

Une définition du langage régulier d'arbres d'arité non-bornée peut se faire à travers le langage $\mathcal{L}(A)$ d'un automate d'arbre A reconnaissant l'ensemble des codages curryfiés de ces arbres. Tout langage régulier d'arbres peut ainsi être reconnu par un automate d'arbre ascendant déterministe.

1.3.4 Codage binaire frère-fils (descendant)

L'autre codage binaire des arbres d'arité non-bornée, que nous appelons ici le codage *frère-fils fcns* (de l'anglais « first-child next-sibling »), est le plus répandu dans la littérature des bases de données XML.

Le codage binaire $fcns(t)$ d'un arbre $t \in \mathcal{T}^u_\Sigma$ non bornée le représente sous forme binaire où chaque fils d'un noeud est respectivement le premier fils à gauche et le premier frère à droite. Chaque noeud ne possédant pas forcément un voisin ou/et un fils, on introduit le symbole # d'arité 0 remplaçant les membres manquants. L'alphabet de l'arbre obtenu est donc composé de l'alphabet $\Sigma^{(2)} = \{f^{(2)} \mid f \in \Sigma\}$ d'arité 2 auquel on ajoute le symbole $\#^{(0)}$.

Formellement, l'encodage frère-fils repose sur une bijection d'un ensemble d'arbres vers des arbres, $fcns : (\mathcal{T}^u_\Sigma)^* \to \mathcal{T}_{\Sigma^{(2)} \cup \{\#^{(0)}\}}$, définie par :

$$\begin{aligned} fcns(\varepsilon) &= \# \\ fcns(f(t_1, \ldots, t_k)) &= f(fcns(t_1, \ldots, t_k), \#) \\ fcns(f(t_1, \ldots, t_k), t'_1, \ldots, t'_m) &= f(fcns(t_1, \ldots, t_k), fcns(t'_1, \ldots, t'_m)) \end{aligned}$$

Exemple 8. *Dans la figure 1.6, nous considérons un arbre t d'arité non-bornée qui représente une bibliographie, et présentons son codage $fcns(t)$. Nous pouvons observer la présence de liens rouges sombres représentant la relation entre un noeud π et son premier fils π'. Dans sa forme frère-fils le noeud correspondant à π' est le premier fils du noeud correspondant à π. De même, les liens rouges plus clairs représentent la relation entre un noeud et son prochain frère, deuxième fils dans le codage $fcns(t)$.*

La grammaire suivante défini, à travers ces arbres syntaxiques, l'ensemble des bibliographies valides, l'état initial étant *bib*. Le symbole spécial PCDATA (comme en XML) représente le mot vide (pas de fils mais possiblement une valeur textuelle que nous décidons d'ignorer).

$$\begin{aligned} bib &\to book^* & \qquad auth &\to \text{PCDATA} \\ book &\to auth^+ tit & \qquad tit &\to \text{PCDATA} \end{aligned}$$

Pour reconnaître l'ensemble des codages *fcns* de bibliographies valides, nous pouvons nous servir de l'automate d'arbre descendant suivant avec pour

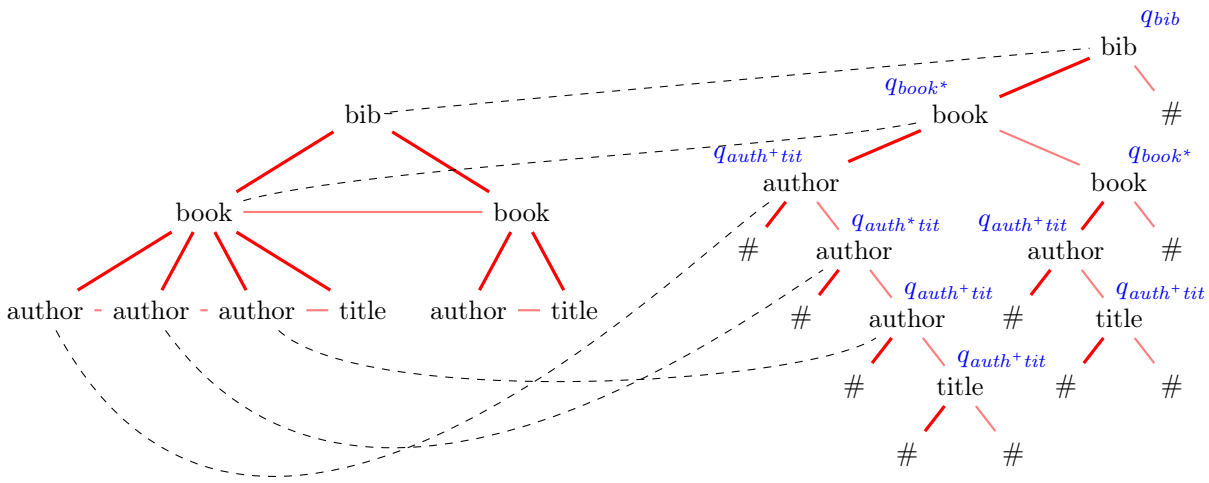


FIGURE 1.6 – L'encodage frère-fils.

seul état initial q_{bib}. Les règles de cet automate correspondent aux règles d'un schéma décrivant cet arbre et de la sémantique de ces expressions régulières :

$$
\begin{array}{ll}
q_{bib} \xrightarrow{bib} (q_{book^*}, q_{\#}), & q_{book^*} \xrightarrow{book} (q_{auth^+tit}, q_{book*}), \\
q_{book^*} \xrightarrow{\#} (), & q_{auth^+tit} \xrightarrow{auth} (q_{\text{PCDATA}}, q_{auth^*tit}), \\
q_{auth^*tit} \xrightarrow{tit} (q_{\text{PCDATA}}, q_{\#}), & q_{auth^*tit} \xrightarrow{auth} (q_{\text{PCDATA}}, q_{auth^*tit}), \\
q_{\text{PCDATA}} \xrightarrow{\#} (), & q_{\#} \xrightarrow{\#} ().
\end{array}
$$

Cet exemple illustre le fait que les automates descendants sont adaptés pour exprimer les schémas que sont les DTD (Brüggemann-Klein, 1993) ou XML SCHEMA, qui peuvent être vus comme des grammaires permettant de représenter des arbres de données XML. Le fait d'être dans un codage frère-fils permet dans un parcours descendant de connaître les frères qui précédent un noeud ainsi que ses ancêtres, les seules choses nécessaires pour décider si l'étiquette de ce noeud respecte le schéma, ce qui peut donc se faire de manière déterministe si les schémas sont non ambigus.

Le fait de connaître les étiquettes des frères qui le précède ajoute cependant un biais dans le parcours descendant. En effet, pour un arbre d'arité bornée, un automate descendant déterministe se limite à la connaissance des ancêtres de ce noeud.

Exemple 9. *Prenons le langage* $\mathcal{L} = \{f(b,a), f(a,b)\}$. *Si nous considérons maintenant le langage* $\mathcal{L}'$ *composé des encodages frère-fils de ces arbres, cela nous donne* $\mathcal{L}' = \{f(b(\#,a(\#,\#)),\#), f(a(\#,b(\#,\#)),\#)\}$. *Il existe un automate* A *déterministe reconnaissant ce langage, nécessitant* 5 *états* q_0, q, q_a, q_b

et $q_\#$ dont q_0 état initial, et les règles suivantes :

$$\begin{array}{ll} q_0 \xrightarrow{f} (q, q_\#), & q_\# \xrightarrow{\#} (), \\ q \xrightarrow{a} (q_\#, q_b), & q \xrightarrow{b} (q_\#, q_a), \\ q_a \xrightarrow{a} (q_\#, q_\#), & q_b \xrightarrow{b} (q_\#, q_\#). \end{array}$$

L'existence d'un tel automate déterministe vient du fait qu'en arrivant au deuxième fils de f nous avons déjà pris connaissance du premier fils. Le langage $\mathcal{L} = \{f(b,a), f(a,b)\}$ d'arité bornée n'est cependant par reconnaissable par un automate déterministe descendant.

1.4 Mots imbriqués

Une autre manière de représenter les données arborées non bornées est de remplacer l'arbre par une suite d'ouvertures et de fermetures de noeuds en parcourant l'arbre en profondeur. On appelle une telle suite d'opérations *un mot imbriqué*, comme introduit par Alur et Madhusudan (2009), modèle que nous présenterons par la suite avec quelques restrictions mineures.

1.4.1 Définition

Soit Σ un alphabet. Une parenthèse ouvrante sur Σ est un élément de $\{\texttt{op}\} \times \Sigma$ et une parenthèse fermante un élément de $\{\texttt{cl}\} \times \Sigma$. Cette notion de parenthèses ou opérations d'ouverture et de fermeture se retrouve dans les structures de fichiers XML. Nous décidons ici de rester dans la notion d'arbres introduite précédemment, sans alourdir l'alphabet par des noeuds textuels ou autres noeuds n'appartenant pas à la structure propre d'un arbre.

Des mots imbriqués sur Σ sont des mots définis sur l'alphabet $\hat{\Sigma} = \{\texttt{op}, \texttt{cl}\} \times \Sigma$. Nous supposons que chaque parenthèse ouvrante $(\texttt{op}, a)$ est bien fermée par une parenthèse fermante $(\texttt{cl}, a)$ postérieure, et que chaque parenthèse fermante $(\texttt{cl}, a)$ soit bien ouverte par une parenthèse $(\texttt{op}, a)$ antérieure. Un tel mot est appelé *bien imbriqué*, si la relation entre les parenthèses ouvrantes et fermantes peut se faire sans chevauchement, comme l'illustre la figure 1.7. En d'autres mots nous interdisons les mots du type :

$$(\texttt{op}, a) \dots (\texttt{op}, b) \dots (\texttt{cl}, a) \dots (\texttt{cl}, b)$$

où les parenthèses ayant pour label a correspondent avec les parenthèses ayant pour label b et non entre elles.

L'ensemble de tous les mots imbriqués sur Σ est dénoté par $\mathcal{NW}_\Sigma$. Par définition, $\mathcal{NW}_\Sigma \subseteq \hat{\Sigma}^*$. La relation de correspondance entre parenthèses ouvrantes

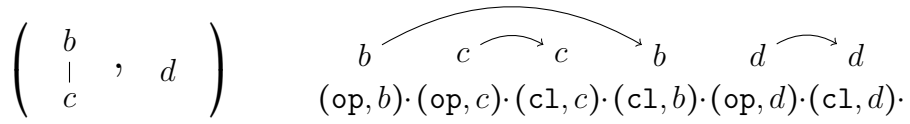
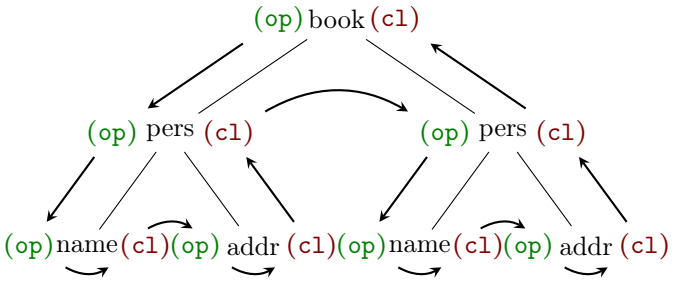


FIGURE 1.7 – Exemple de mot imbriqué et de la sequence d'arbres d'arité non-bornée correspondante



$\text{LINEAR}(t) =$
$(\texttt{op}, book)\cdot$
$\quad(\texttt{op}, pers)\cdot$
$\qquad(\texttt{op}, name)\cdot(\texttt{cl}, name)\cdot$
$\qquad(\texttt{op}, addr)\cdot(\texttt{cl}, addr)\cdot$
$\quad(\texttt{cl}, pers)\cdot(\texttt{op}, pers)\cdot$
$\qquad(\texttt{op}, name)\cdot(\texttt{cl}, name)\cdot$
$\qquad(\texttt{op}, addr)\cdot(\texttt{cl}, addr)\cdot$
$\quad(\texttt{cl}, pers)\cdot$
$(\texttt{cl}, book)$

FIGURE 1.8 – Parcours en profondeur d'un arbre t et sa linéarisation

et fermantes fait partie du mot imbriqué, ce que l'on suppose manipuler par la suite et qui ne nécessitera pas de contrôle. Cette correspondance peut cependant être déduite aisément à l'aide d'une analyse syntaxique classique, comme chaque parseur XML le fait, sous l'hypothèse que le mot est bien imbriqué.

1.4.2 Linéarisation d'arbres d'arité non-bornée

Les documents XML stockés dans des fichiers, ou échangés sur flux, sont clairement des mots imbriqués, pouvant être obtenus par la linéarisation d'un arbre contenant des valeurs textuelles. Un exemple de mot imbriqué et de séquences d'arbres d'arité non-bornée qu'il représente sont données en figure 1.7. Toute séquence d'arbres d'arité non-bornée sur $(\mathcal{T}^u_\Sigma)^*$ peut être linéarisée dans un mot imbriqué sur l'alphabet $\hat{\Sigma}$. Les lettres internes (non associées à une ouverture ou fermeture) ne sont pas nécessaires ici, les valeurs de données textuelles étant ignorées. Un exemple est donné en FIGURE 1.8.

La linéarisation s'obtient en parcourant l'arbre en profondeur. Cela peut se faire récursivement, en concaténant la balise ouvrante de la racine à la linéarisation successive des fils, et à la balise de fermeture de la racine. Elle peut être obtenue par la fonction $\text{LINEAR} : (\mathcal{T}^u_\Sigma)^* \to \mathcal{NW}_\Sigma$ définie par :

$$\begin{aligned} \text{LINEAR}(t_1, \ldots, t_k) &= \text{LINEAR}(t_1) \cdot \ldots \cdot \text{LINEAR}(t_k) \\ \text{LINEAR}(a(t_1, \ldots, t_k)) &= (\texttt{op}, a) \cdot \text{LINEAR}(t_1, \ldots, t_k) \cdot (\texttt{cl}, a) \end{aligned}$$

Il n'est pas difficile de voir que tout mot imbriqué est l'image d'une, et d'une

seule, séquence d'arbres d'arité non-bornée. En effet, nous supposons que chaque parenthèse ouvrante $(\texttt{op}, a)$ est fermée par une parenthèse fermante correspondante $(\texttt{cl}, a)$ et inversement [1]. La fonction LINEAR est donc une bijection.

1.4.3 Automates de mots imbriqués

Définition 13. *Un* automate de mots imbriqués *ou* NWA *(de l'anglais* Nested Words Automaton*) A est représenté par un tuple $A = (\Sigma, Q, \Gamma, \mathit{init}, \mathit{fin}, \mathit{rul})$. Ce tuple est composé d'un ensemble fini d'états Q, comprenant les états initiaux $\mathit{init} \in Q$ et états finaux $\mathit{fin} \in Q$, d'un ensemble de symboles de piles Γ utilisés pour définir un ensemble de règles $\mathit{rul} \subseteq Q \times \hat{\Sigma} \times \Gamma \times Q \subseteq \mathit{rul}$. Les règles peuvent être représentées par $q \xrightarrow{\alpha\, a\, :\, \gamma} q'$ pour $(q, (\alpha, a), \gamma, q') \in \mathit{rul}$.*

L'exécution d'un NWA peut être décrite par une succession de configurations de l'automate pendant la lecture d'un mot. Une *configuration* est définie par un tuple sur $\hat{\Sigma} \times Q \times \Gamma^*$ telle que la configuration $C = (u, q, S, w)$ symbolise le fait qu'un mot u reste à lire, que l'état q a été atteint, et que la pile est représentée par la concaténation de symboles S.

Deux types de règles se présentent à nous :

Les règles ouvrantes $q \xrightarrow{\texttt{op}\, a'\, :\, \gamma} q'$ qui s'appliquent à un état q et qui en lisant $(\texttt{op}, a)$, ajoute γ sur la pile (la concatène à S) avant d'atteindre l'état q'. Cette étape se traduit dans une configuration par :

$$\frac{q \xrightarrow{\texttt{op}\, a\, :\, \gamma} q' \in \mathit{rul}}{((\texttt{op}, a) \cdot u, q, S, v) \to (u, q', \gamma \cdot S)}$$

Les règles fermantes $q \xrightarrow{\texttt{cl}\, a\, :\, \gamma} q'$, quant à elle, doivent, en partant de l'état q, retirer le symbole γ du sommet de la pile (le supprimer de S) et lire $(\texttt{cl}, a)$ pour pouvoir atteindre l'état q' ce qui nous donne :

$$\frac{q \xrightarrow{\texttt{cl}\, a\, :\, \gamma} q' \in \mathit{rul}}{((\texttt{cl}, a) \cdot u, q, \gamma \cdot S, v) \to (u, q', S)}$$

Un mot u appartient au langage $\mathcal{L}_A$ si et seulement s'il existe un état initial $q_0 \in \mathit{init}$ et une séquence d'instructions qui à partir de $C = (u, q_0, \varepsilon)$ nous amène à $C' = (\varepsilon, q_f, \varepsilon)$, tel que $q_f \in \mathit{fin}$. L'exécution peut donc échouer si l'on essaye de dépiler un symbole de pile ne correspondant pas au symbole requis par la règle fermante, ou si aucun état final n'est atteint.

1. Cette hypothèse n'est pas supposée dans le cas général de Alur et Madhusudan (2009), mais s'obtient gratuitement dans le cas du XML

Définition 14. *Un* NWA *est dit* déterministe *et noté d*NWA *si :*

1. *pour tout* $q \in Q$ *et* $a \in \Sigma$ *il existe au plus un symbole de pile* $\gamma \in \Gamma$ *et un état* $q' \in Q$ *tel que* $q \xrightarrow{\texttt{op}\, a\, :\, \gamma} q' \in rul$.
2. *pour tout* $q \in Q$, $a \in \Sigma$ *et* $\gamma \in \Gamma$ *il existe au plus un état* $q' \in Q$ *tel que* $q \xrightarrow{\texttt{cl}\, a\, :\, \gamma} q' \in rul$.

Exemple 10. *Pour illustrer l'exécution, prenons comme exemple le d*NWA N_{10} *reconnaissant une version simplifiée de répertoire d'adresses. Ces répertoires sont représentés par des arbres ayant pour racine un noeud « book ». Ils contiennent un nombre non borné de fils correspondant aux différents contacts qui y sont stockés, à savoir des sous arbres ayant pour racine un noeud « pers » ayant pour fils le nom « name », et l'adresse « addr » du dit contact.*

L'exécution peut se voir comme une annotation non plus des noeuds mais des opérations d'ouverture (état à gauche) et fermeture (état à droite) de chacun de ces noeuds, l'état y étant annoté correspondant à l'état atteint après cette opération. Au dessus de chaque noeud se trouve le symbole de pile devant être posé à l'ouverture de ce noeud et consommé à sa fermeture.

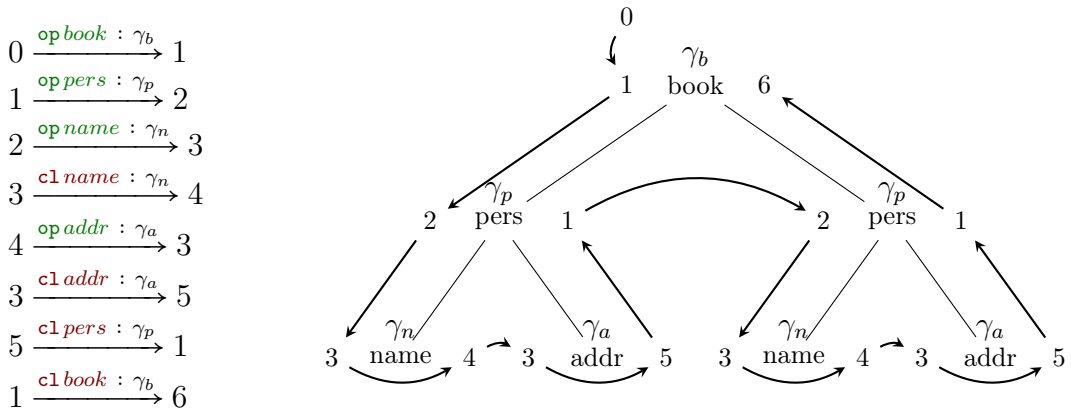


FIGURE 1.9 – Règles et exemple d'exécution de N_{10}

Comme l'illustre l'exécution de N_{10}, un des intérêts des NWAs est de combiner à la fois un parcours descendant et ascendant, et cela sur la linéarisation d'arbres non bornés.

1.4.4 Automate descendant

Grâce aux symboles de piles, les NWAs peuvent être restreints à un modèle s'approchant des $d\text{TA}^{\downarrow}s$. Pour mieux comprendre ce parallèle, identifions

l'information connue pour décider de l'état annotant un noeud. Pour les $d\text{TA}^{\downarrow}s$, seule l'information sur les ancêtres d'un noeud permet de décider de l'état utilisé. Si on utilise cette approche sur l'encodage frère-fils, les arbres identifiés par un NWA pouvant être d'arité non bornée, l'état d'un noeud dépend de ses ancêtres et de ses frères gauches. On peut se restreindre à cette connaissance dans les NWAs en remplaçant les symboles de piles par des états. En restreignant les symboles de pile aux états, et en assurant que l'état atteint lors d'une règle fermante soit l'état retiré de la pile, on assure que les futures actions de l'automate ne dépendent que de ce qui précède l'ouverture de ce noeud, et ne tiennent donc pas compte du traitement du sous arbre.

Définition 15. *Un* dNWA *est dit* descendant*, dénoté* $d\text{NWA}^{\downarrow}$*, si les symboles de piles sont restreints aux états, i.e.* $\Gamma \subseteq Q$*, et que les règles fermantes soient de la forme* $q \xrightarrow{\texttt{cl}\,a\,:\,q'} q'$*. L'état atteint correspond alors au symbole de pile.*

Cette restriction étend la notion de descendant aux arbres d'arité non bornée. Mais pour pouvoir affirmer cela, il faut d'abord vérifier quel lien réel associe ces deux classes.

Proposition 1. *Pour tout* $d\text{TA}^{\downarrow}A$ *il existe un* $d\text{NWA}^{\downarrow}$ N *tel que* $\mathcal{L}_A = \mathcal{L}_N$*.*

Preuve. (esquisse) Soit $A = (\Sigma, states_A, init_A, rul_A)$. Nous associons à chaque règle r son arité, dénoté $ar(r)$ correspondant à celle du symbole lu.

L'automate correspondant est défini par $N = (\Sigma, Q_N, \Gamma_N, init_N, rul_N, fin_N)$, tel que $Q_N = \Gamma_N = \{\langle r, j\rangle \mid r \in rul_A, 0 \leq j \leq ar(r)\} \cup \{\texttt{o}, \texttt{f}\}$, $init_N = \{\texttt{o}\}$, $fin_N = \{\texttt{f}\}$, et les règles rul_A suivantes :

$$\frac{\begin{array}{c} q_0 \in init_A \\ r = q_0(a(x_1, \ldots, x_k)) \to (\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \in rul_A \end{array}}{\texttt{o} \xrightarrow{\texttt{op}\,a\,:\,\texttt{f}} \langle r, 0\rangle \qquad \langle r, k\rangle \xrightarrow{\texttt{cl}\,a\,:\,\texttt{f}} \texttt{f}}$$

$$\frac{\begin{array}{c} r = q_0(a(x_1, \ldots, x_k)) \to (\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \in rul_A \\ r' = q_j(b(x_1, \ldots, x_m)) \to (\langle p_1, x_1\rangle, \ldots, \langle p_m, x_m\rangle) \in rul_A \text{ pour } 1 \leq j \leq k \end{array}}{\langle r, j-1\rangle \xrightarrow{\texttt{op}\,b\,:\,\langle r,j\rangle} \langle r', 0\rangle \qquad \langle r', m\rangle \xrightarrow{\texttt{cl}\,b\,:\,\langle r,j\rangle} \langle r, j\rangle}$$

Intuitivement, quand l'automate est dans l'état $\langle r, j\rangle$, il s'apprête à traiter l'état q_{j+1} (ou à fermer une parenthèse et remonter à la règle précédente si $j = k$). L'automate simule l'exécution de la règle de A en gardant, à travers le symbole de pile, la règle et l'état atteint. Dès lors, l'égalité $\mathcal{L}_A = \mathcal{L}_N$ peut être prouvée par une preuve inductive. □

Proposition 2. *Pour tout* $d\text{NWA}^{\downarrow}$ N *nous pouvons construire en temps* $O(|\Sigma|^2 * |Q_N|^2)$ *un* $d\text{TA}^{\downarrow}A$ *tel que* $\mathcal{L}_A = \{fcns(t) \mid t \in \mathcal{L}_N\}$*.*

Preuve. (esquisse) Let $N = (\Sigma, Q_N, \Gamma_N, init_N, rul_N, fin_N)$. Nous pouvons remarquer que $\Gamma_N = Q_N$ à cause de la propriété descendante de N.

On défini le tuple $A = (\Sigma \cup \{\#\}, Q_A, init_A, rul_A)$ tel que $Q_A = init_N \cup \{q_\#\} \cup Q_N \times \Sigma \times \Gamma_N$, $init_A = init_N$, et les règles rul_A sont définies par :

$$\frac{q_0 \in init_N \qquad q_2 \in fin_N \qquad q_0 \xrightarrow{\texttt{op}\, a\, :\, q_1} q_1 \in rul_N}{q_0(a(x_1, x_2)) \to (\langle\langle q_1, a, q_2\rangle, x_1\rangle, \langle q_\#, x_2\rangle)} \qquad \frac{q \xrightarrow{\texttt{cl}\, a\, :\, q'} q' \in rul_N}{\langle q, a, q'\rangle(\#) \to ()}$$

$$\frac{b \in \Sigma \qquad p \in Q_N \qquad q \xrightarrow{\texttt{op}\, a\, :\, q_2} q_1 \in rul_N}{\langle q, b, p\rangle(a(x_1, x_2)) \to (\langle\langle q_1, a, q_2\rangle, x_1\rangle, \langle\langle q_2, b, p\rangle, x_2\rangle)} \qquad \frac{true}{q_\#(\#) \to ()}$$

Un état $\langle q, a, \gamma\rangle$ correspond à une étape de l'exécution de N étant dans l'état q, avec γ en sommet de pile et a l'étiquette du parent du noeud courant. L'état $q_\#$, quant à lui, permet d'assurer que A n'accepte que des arbres résultant d'un codage frère-fils. Une preuve par induction peut montrer que $\mathcal{L}_A = \{fcns(t) \mid t \in \mathcal{L}_N\}$. □

Nous pouvons remarquer que même s'il est possible de passer d'un modèle à l'autre à travers des encodages, les manières dont les noeuds sont traités ne sont pas identiques. En effet, une information sur les frères antécédents peut être transmise à travers les symboles de pile et les états dans les dNWAs$^\downarrow$, ce qui n'est pas le cas pour les dTA$^\downarrow$$s$, décidant directement de l'état à appliquer sur chaque sous arbre d'un noeud. Cette possibilité de réduire un modèle dans l'autre permet cependant de partager, modulo la réduction, la complexité de décision de certains problèmes tels que l'équivalence.

Corollaire 1. *Le problème d'équivalence des d*NWAs$^\downarrow$ *peut être réduit en temps polynomial à celui des d*TA$^\downarrow$$s$*, et vice versa.*

Chapitre 2
Transducteurs

Notre thème de recherche porte sur les transformations de données structurées et plus particulièrement la définition, l'étude et l'apprentissage de transducteurs les représentant. Mais intéressons nous d'abord aux autres modèles ayant déjà été étudiés dans le domaine.

Dans un premier temps, on abordera les transducteurs de mots. Ils sont introduits par Mealy (1955) et Moore (1956) sous la forme d'automates «lettre à lettre», où les lettres de sorties sont respectivement associées aux transitions ou aux états. Ces familles de transducteurs se voient rapidement déclinées sous différentes formes. Cela se fera à travers les travaux de Elgot et Mezei (1965), ou encore sur ceux de Choffrut (1978) concernant la classe des fonctions rationnelles et sous-séquentielles. Le livre de Berstel (1979) permet aussi d'approfondir les connaissances, parfois en adoptant une vue plus algébrique, sur les différentes classes de transductions de mots. Les modèles de transduction de mots, plus simples et parfois plus épurés, permettent une meilleure compréhension des mécanismes inhérents à toute transduction sans s'encombrer de la complexité accrue liée aux formats plus complexes que sont les données arborées. Largement étudiés, les transducteurs et transformations de mots bénéficient de nombreux résultats sur la décidabilité de problèmes usuels tels que l'équivalence, le vide, l'universalité, ou encore l'apprentissage de ces machines et les complexités associées.

Grâce à la richesse des données structurées et aux façons possibles de les parcourir, il existe de nombreux formalismes permettant d'en définir la transformation. Dans la deuxième partie de ce chapitre nous parlerons des différentes classes de transducteurs d'arbres en arbres ainsi que des principaux résultats de décidabilité et complexité sur les divers problèmes évoqués précédemment.

Contents

1.1 Mots 11
1.1.1 Mots et langages 11
1.1.2 Grammaires 14
1.1.3 Automates 15
1.1.3.1 Définitions 15
1.1.3.2 Equivalence et minimisation 18
1.2 Arbres d'arité bornée 18
1.2.1 Définition 19
1.2.2 Automates d'arbres 21
1.2.2.1 Automates ascendants 22
1.2.2.2 Automates descendants 23
1.3 Arbres d'arité non-bornée 25
1.3.1 Definition 25
1.3.2 Comment définir les automates ? 25
1.3.3 Codage binaire curryfié (ascendant) 26
1.3.4 Codage binaire frère-fils (descendant) 29
1.4 Mots imbriqués 31
1.4.1 Définition 31
1.4.2 Linéarisation d'arbres d'arité non-bornée 32
1.4.3 Automates de mots imbriqués 33
1.4.4 Automate descendant 34

2.1 Transducteurs de mots

Dans cette section, nous discutons des classes de transducteurs mots en mots. Les transducteurs les plus généraux, appelés *transducteurs rationnels*, définissent des relations entre les mots. Pour que les problèmes informatiques (équivalence, normalisation, apprentissage) puissent être traitables, il est nécessaire de se restreindre aux transducteurs définissant des fonctions partielles, associant à chaque entrée une sortie unique. Pour cela, nous considérerons la classe des transducteurs déterministes avec ou sans anticipation.

2.1.1 Transducteurs rationnels

Soient Σ et Δ deux alphabets finis que nous appellerons respectivement les alphabets d'entrée et de sortie. Notre objectif est de définir des relations entre des mots fournis au transducteur, les mots d'entrée, et les mots qu'il produit, les mots de sortie.

2.1.1.1 Syntaxe

Les transducteurs de mots, ou transducteurs rationnels, sont des automates de mots associant à chaque étape de la lecture une production de sortie. Le modèle introduit par la suite permet, à la différence des automates vus précédemment, de lire plus qu'un symbole à chaque étape de son exécution.

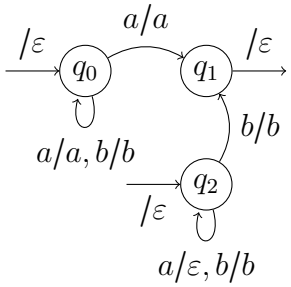
Définition 16. *Un transducteur rationnel, noté* TM*, est un couple* $M = (\Sigma, \Delta, Q, \mathit{init}, \mathit{fin}, \mathit{rul})$ *où* Q *est un ensemble fini d'*états*,* $\mathit{init} \subseteq Q \times \Delta^*$ *et* $\mathit{fin} \subseteq Q \times \Delta^*$ *sont respectivement l'ensemble des états initiaux et finaux associés à des mots produits, et* $\mathit{rul} \subseteq (Q \times \Sigma^*) \times (\Delta^* \times Q)$ *est l'ensemble fini des* règles*.*

Une règle (q_1, u, w, q_2) pourra être représentée par $q_1 \xrightarrow{u/w} q_2$. Une production initiale $(q, w) \in \mathit{init}$, notée $\xrightarrow{/w} q$, et une production finale $(q, w) \in \mathit{fin}$ par $q \xrightarrow{/w}$. Pour une règle r, égale à $q_1 \xrightarrow{u/w} q_2$, on apelle $\mathit{gch}(r) = q_1$ sa partie gauche et $\mathit{dt}(r) = q_2$ sa partie droite. Son entrée est le mot lu $\mathit{lect}(r) = u$ et sa sortie, le mot produit, $\mathit{prod}(r) = w$. On remarque que les règles peuvent être de la forme $q_1 \xrightarrow{\varepsilon/w} q_2$ permettant une production sans avoir « consommé » l'entrée. Elles sont appelées des *epsilon-productions*(ε-productions).

On représente par $q \xrightarrow{u/w}_* q'$ la clôture transitive des règles d'un TM correspondant à l'enchaînement d'une ou plusieurs règles de transition. Nous avons $q \xrightarrow{u_0 \cdot u/w' \cdot w}_* q' \in \mathit{rul}^*$ avec $u_0,\ u \in \{\varepsilon\} \cup \Sigma^*$ ssi $q \xrightarrow{u_0/w'} q''$ et $q'' \xrightarrow{u/w}_* q'$, ou $q \xrightarrow{u_0/w'} q'$ pour $u = \varepsilon$.

$$\Sigma = \Delta = \{a, b\},$$
$$Q = \{q_0, q_1, q_2\},$$
$$init = \{(q_0, \varepsilon), (q_2, \varepsilon)\},$$
$$fin = \{(q_1, \varepsilon)\},$$
$$rul = \left\{ \begin{array}{ll} q_0 \xrightarrow{a/a} q_0, & q_0 \xrightarrow{b/b} q_0, \\ q_0 \xrightarrow{a/a} q_1, & q_2 \xrightarrow{a/\varepsilon} q_2, \\ q_2 \xrightarrow{b/b} q_2, & q_2 \xrightarrow{b/b} q_1 \end{array} \right\}$$

(a)



(b)

FIGURE 2.1 – Transducteur de mots $M_{2.1}$: copie de chaque mot se terminant par un a et effacement toutes les occurrences de "a" dans un mot finissant par un "b".

Exemple 11. *Pour illustrer cette definition, un premier transducteur $M_{2.1}$ est défini en figure 2.1. Son alphabet d'entrée est identique à celui de sortie. Ils contiennent uniquement les lettres a et b, soit $\Sigma = \{a, b\} = \Delta$. Le transducteur $M_{2.1}$ possède deux états initiaux, q_0 et q_2, par lesquels il peut débuter la réécriture d'un mot de manière non déterministe. Dans q_0, seuls les mots d'entrée se terminant par un "a" peuvent aboutir à un résultat, sa copie. L'état q_2 par contre n'accepte en entrée que des mots se terminant par un "b". Dans ce cas, le mot de sortie obtenu en supprimant toutes les occurrences de "a" dans le mot fourni au transducteur.*

Supposons que le transducteur $M_{2.1}$ lise un mot "ab". Dans ce cas, il a le choix d'aller dans l'état q_0 ou dans l'état q_2.

S'il choisit q_0, il consomme le "a" de l'entrée, le produit dans la sortie, et se met soit dans l'état q_1, soit il reste en q_0. À partir de q_1, le transducteur sera bloqué, n'ayant aucune règle permettant de consommer une lettre "b" dans l'entrée. À partir de q_0 le seul moyen de lire un "b" est de le recopier en restant dans le même état n'autorisant pas de transition finale. Les sorties partielles possibles deviennent alors invalides et sont abandonnées.

Par contre, s'il choisit de débuter dans l'état initial q_2, il peut consommer la lettre "a" de l'entrée sans rien produire et rester dans l'état q_2. Puis, il peut consommer la lettre "b" de l'entrée, la sortir, en passant dans l'état q_1. Cet état est final et ne produit rien de supplémentaire, ce qui nous donne pour sortie finale le mot "b".

Le transducteur peut choisir l'état par lequel il commence dans l'ensemble des états initiaux. Les langages d'entrées acceptés à partir de chacun de ses états initiaux (ensemble des mots atteignants une transition finale) étant dis-

joints, le transducteur associe à tout mot au plus un résultat. Pour cette raison, la relation définie par ce transducteur est bien une fonction. Cette fonction n'est pas totale. Par exemple, le mot vide n'est pas accepté en entrée.

2.1.1.2 Sémantique

Un transducteur rationnel défini une relation entre des mots d'entrée et mots de sortie. Cette relation sera formellement définie dans ce paragraphe.

Plus particulièrement, on peut voir que chaque état q d'un transducteur M défini lui même une relation entre des mots d'entrée et des mots de sortie ; relation que nous représentons par $[\![M]\!]_q$. Ces relations sont les plus petites relations $[\![M]\!]_q \subseteq \Sigma^* \times \Delta^*$ telles que pour tout état q :

$$\begin{array}{rcl} [\![M]\!]_q & = & \{(u \cdot u', w \cdot w') \mid q \xrightarrow{u/w} q' \in rul,\ (u', w') \in [\![M]\!]_{q'}\} \\ & \cup & \{(\varepsilon, w) \mid (q, w) \in fin\} \end{array}$$

À partir de l'état q, on peut donc lire un préfixe u de l'entrée et le transformer en w par une règle, qui de q passe dans q', uniquement si le reste de l'entrée u' peut ensuite être transformée à partir de q' en un mot w'. Dans ce cas, l'entrée $u \cdot u'$ peut être transformée en $w \cdot w'$ à partir de l'état q. Il y a aussi la possibilité que q soit final, dans ce cas le mot vide ε est transformé en w à partir de q.

Une fois les transformations associées à chacun des états identifiés, la transformation finale $[\![M]\!]$ s'obtient en ajoutant, à chaque sortie d'une transformation $[\![M]\!]_{q_0}$ associée à un état initial q_0, la sortie initiale qui la précède. Ceci nous donne :

$$[\![M]\!] = \bigcup_{(q,w) \in init} w \cdot [\![M]\!]_q$$

Exemple 12. *Reconsidérons le transducteur $M_{2.1}$ de la figure 2.1. La relation $[\![M_{2.1}]\!]_{q_0}$ est égale à $\{(u, u) \mid u \in (a+b)^* a\}$, puisque l'état q_0 accepte en entrée uniquement les mots se terminant par a en les recopiant dans la sortie. La relation $[\![M_{2.1}]\!]_{q_2}$, quant à elle, est égale à $\{(a_1 \cdot \ldots \cdot a_n, u_1 \cdot \ldots \cdot u_n) \mid a_n = b,\ a_i = a \Rightarrow u_i = \varepsilon,\ \text{et } a_i = b \Rightarrow u_i = b\}$ puisque q_2 accepte tous les mots d'entrée se terminant par b, n'en recopiant que les b dans la sortie. La sémantique du transducteur est la fonction partielle $[\![M_{2.1}]\!] = [\![M_{2.1}]\!]_{q_0} \cup [\![M_{2.1}]\!]_{q_2}$ puisqu'aucune sortie n'est produite dans les productions initiales .*

Définition 17. *Nous disons de deux transducteurs M et M' qu'ils sont* équivalents *s'ils ont la même sémantique $[\![M]\!] = [\![M']\!]$.*

2.1.1.3 Simplification des règles

En manipulant le formalisme de transduction choisi, nous montrons dans cette partie les différentes propriétés qui lui sont associées, et le rapport entres les différents modèles de transducteurs de mots existants.

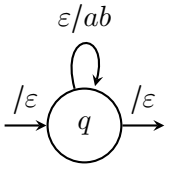
Les TM*s* permettent la production d'une partie de la sortie lors de l'application des règles initiales et finales. Ce choix syntaxique découle directement de la sémantique de ces machines : le fait qu'une partie de la sortie peut être commune à toute sortie, et donc ne nécessite pas de lire le mot de l'entrée motive la présence d'une production initiale. Cependant rien ne nous empêche de nous ramener à un transducteur sans initialisation ou *init-libre* en utilisant comme état initial $q_i \notin Q$ remplaçant chaque paire $(q, w) \in init$ par une ε-transition $q_i \xrightarrow{\varepsilon/w} q$. L'ensemble des initialisations est maintenant un singleton $init = \{(q_i, \varepsilon)\}$.

Le fait de terminer la lecture d'un mot est aussi une information pouvant permettre la production d'une sortie supplémentaire. De la même manière, nous pouvons créer un modèle sans finalisation ou *fin-libre* en remplaçant l'ensemble des finalisations par un singleton $fin = \{(q_f, \varepsilon)\}$ tel que $q_f \notin Q$. Cet état est atteignable uniquement par l'ensemble des ε-transitions $\{q \xrightarrow{\varepsilon/w} q_f \mid (q, w) \in fin\}$.

Les ε-transitions permettent, comme vu précédemment, de produire de la sortie sans avoir besoin de lire l'entrée. Sans ces règles, la taille de la sortie serait bornée par une expression dépendant de la taille de l'entrée et de la taille des règles. La présence de boucles productives composées d' ε-transitions, i.e. $q \xrightarrow{\varepsilon/w}{}^{*} q \in rul$ avec $w \neq \varepsilon$ permet la génération de mots possiblement infinis. Cette génération ne peut être représentée sans l'aide d'ε-transitions.

Un transducteur est dit *ε-libre* si il n'existe pas d' ε-transitions. La déterminisation d'un transducteur nécessite de pouvoir passer dans un modèle ε-libre, ce qui est rendu impossible par l'existence de telles boucles.

Exemple 13.
Prenons le transducteur composé uniquement d'une ε-transition partant et arrivant dans un même état. Le transducteur transforme le mot vide en une répétition non bornée de "ab", i.e. ab^*. La règle ne peut ni être remplacée ni supprimée. Elle est l'unique moyen d'obtenir une répétition possiblement infinie. De plus, aucune partie des mots produits ne peut être projetée dans la phase d'initiation ou de finalisation car le transducteur peut produire ε.



Supposons maintenant qu'il n'y ait plus de boucles composées d'ε-transitions. Deux possibilités s'offrent à nous pour supprimer chaque ε-transition : soit fu-

sionner l'ε-transition à chaque règle la précédant, soit la fusionner avec chaque règle la suivant. Pour la fusion à droite, on supprime donc chaque règle $q \xrightarrow{\varepsilon/w} q'$ en créant pour toute règle $q' \xrightarrow{u/w'} q''$ la règle $q \xrightarrow{u/w\cdot w'} q''$. Le symbole u pouvant être lui même ε, seul le fait qu'il n'existe pas de boucle composée uniquement d'ε-transitions nous assure que ce procédé se termine. De plus, si $(q', w'') \in \mathit{fin}$, on ajoute $(q, w \cdot w'')$ à l'ensemble *fin*.

Il est cependant impossible, dans ce formalisme, de pouvoir se ramener dans tous les cas à un transducteur à la fois *init*-libre et *fin*-libre. Le mot ε pouvant être transformé par le transducteur de départ, l'absence de ε-transition, d'initialisation et finalisation ne permettrait pas de représenter la transduction de ce mot.

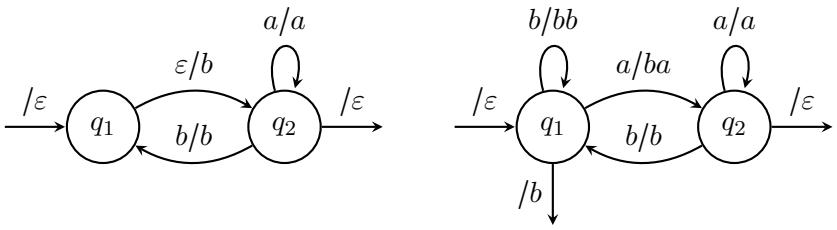


FIGURE 2.2 – Suppression des ε-transitions

Exemple 14. *Le transducteur $M_{2.2(a)}$ est un exemple de* TM *ayant les propriétés nécessaires pour passer sous forme ε-libre. La production de la règle $q_1 \xrightarrow{\varepsilon/b} q_2$ préfixe la production de toutes les règles sortant de q_2. Par exemple la règle $q_2 \xrightarrow{a/a} q_2$ implique que nous pouvons lire "a" à partir de q_1 en produisant "ba" et en arrivant sur q_2. Le fait que $(q_2, \varepsilon) \in \mathit{fin}$ nous donne que $M_{2.2(a)}$ transforme "ε" en "b" donc $(q_1, b) \in \varepsilon$. Cette finalisation ne peut être supprimée en gardant une forme ε-libre.*

Dans la littérature, les transducteurs rationnels sont parfois représentés avec des règles définies sur $Q \times \Sigma \cup \{\varepsilon\} \times \Delta \cup \{\varepsilon\} \times Q$, permettant de lire et produire au plus un symbole. Les formalismes *à lecture et production lente*, que nous introduisons par la suite, représentent respectivement la limitation à un symbole, et non au mot, dans la lecture et la production d'une règle. La proposition 3 montre que tout TM est définissable sous cette forme. Ce résultat repose sur la réécriture des règles du TM en passant dans un premier temps à une lecture lente, et ensuite à une production lente.

Nous disons d'un transducteur rationnel qu'il est *à lecture lente* si il consomme au plus un symbole de l'entrée à chaque transition. Ces règles sont de la forme

$q_1 \xrightarrow{a/w} q_2$ ou $q_1 \xrightarrow{\varepsilon/w} q_2$.

Lemme 1. *Tout* TM *peut être exprimé en* lecture lente*, i.e. pour tout* TM M *il existe un* TM *en lecture lente* M' *qui est équivalent.*

Preuve. (construction) Pour obtenir un transducteur M' à lecture lente à partir de M, il faut remplacer chaque transition $q \xrightarrow{u/w} q$, telle que $u = a_1 \cdot a_2 \cdot \ldots \cdot a_k$ avec $k > 1$, par :

$$(\varepsilon, q) \xrightarrow{a_1/\varepsilon} (a_1, q) \xrightarrow{a_2/\varepsilon} (a_1 \cdot a_2, q) \ldots \xrightarrow{a_{k-1}/\varepsilon} (u \cdot a_k^{-1}, q) \xrightarrow{a_k/w} (\varepsilon, q')$$

Ou plus formellement, pour $M = (\Sigma, \Delta, Q_M, \mathit{init}_M, \mathit{fin}_M, \mathit{rul}_M)$ nous construisons le transducteur $M' = (\Sigma, \Delta, Q_{M'}, \mathit{init}_{M'}, \mathit{fin}_{M'}, \mathit{rul}_{M'})$ dans lequel l'ensemble des états devient un ensemble de paires composé d'un état et d'un mot $Q_{M'} = \{(v, q) \mid q \xrightarrow{u/w} q' \in \mathit{rul}_M,\ v \in \mathit{prefixe}(u) \setminus \{u\}\}$. Les productions initiales et finales restent les même s'adaptant au nouveau type d'état, i.e. $\mathit{init}_{M'} = \{((\varepsilon, q), w) \mid (q, w) \in \mathit{init}_M\}$, $\mathit{fin}_{M'} = \{((\varepsilon, q), w) \mid (q, w) \in \mathit{fin}_M\}$. Les règles, quant à elles, sont définies de la manière suivante :

$$\begin{aligned}
\mathit{rul}_{M'} \ = \{&(\varepsilon, q) \xrightarrow{u/w} (\varepsilon, q') \mid q \xrightarrow{u/w} q' \in \mathit{rul}_M,\ u \in \Sigma \cup \{\varepsilon\}\} \\
\cup \{&(v, q) \xrightarrow{a/\varepsilon} (v \cdot a, q) \mid q \xrightarrow{v \cdot a \cdot v'/w} q' \in \mathit{rul}_M,\ a \in \Sigma,\ v \in \Sigma^*, v' \in \Sigma^+\} \\
\cup \{&(v, q) \xrightarrow{a/w} (\varepsilon, q') \mid q \xrightarrow{v \cdot a/w} q' \in \mathit{rul}_M,\ a \in \Sigma,\ v \in \Sigma^+\}
\end{aligned}$$

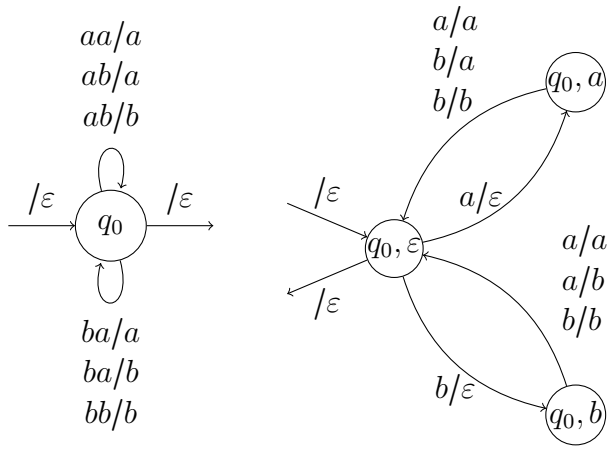
□

Exemple 15. *En passant le transducteur* $M_{2.3}$ *en figure 2.3(a) en lecture lente, on obtient le transducteur en figure 2.3(b). Ce dernier mémorise si la première lettre lue est un "a" ou un "b", puis agit en conséquence. On peut remarquer que la transformation reconnue par le transducteur* $M_{2.3}$ *n'est pas fonctionnelle, comme l'illustre les couples* (ba, a) *et* (ba, b) *contenus dans* $[\![M_{2.3}]\!]$.

En suivant la même intuition, la sortie peut être aussi normalisée de sorte qu'au plus un caractère soit produit à la fois. Un tel transducteur est appelé un *transducteur rationnel à production lente*. Formellement il est *init*-libre et *fin*-libre, et ces règles de transition sont de la forme $q_1 \xrightarrow{u/a} q_2$ ou $q_1 \xrightarrow{u/\varepsilon} q_2$.

Lemme 2. *Tout* TM *peut être exprimé en* production lente*, i.e. pour tout* TM M *il existe un* TM *en production lente* M' *qui est équivalent.*

Preuve. Pour la suite on considère M *init*-libre et *fin*-libre ce qui peut être obtenu facilement. On note w le mot $a_1 \cdot a_2 \cdot \ldots \cdot a_k$ avec $k > 1$. Cette fois



(a) $M_{2.3}$ (b) un transducteur équivalent en lecture lente

FIGURE 2.3 – TM $M_{2.3}$ et sa version en lecture lente

ci l'éclatement de la règle se fait de la manière suivante. Chaque transition $q \xrightarrow{u,w} q'$ de M est remplacée par :

$$(\varepsilon, q, \varepsilon) \xrightarrow{u/a_1} (u, q, a_1) \xrightarrow{\varepsilon/a_2} (u, q, a_1 \cdot a_2) \dots \xrightarrow{a_{k-1}/\varepsilon} (u, q, w \cdot a_k^{-1}) \xrightarrow{\varepsilon/a_k} (\varepsilon, q', \varepsilon)$$

Le transducteur garde en mémoire dans ses états (u, q, w) l'entrée lue u, et la sortie produite w depuis son passage par q. □

Proposition 3. *Tout transducteur rationnel M peut être défini par un transducteur M' à lecture et production lente.*

Preuve. Le transducteur M' est obtenu par l'application successive des constructions définies pour le lemme 1 et le lemme 2 sur M. Le passage d'un TM en TM à production lente ne fait qu'ajouter des ε-transitions et conserve donc la propriété de lecture lente. □

2.1.1.4 Propriétés de clôture

Le fait de pouvoir passer dans un format à lecture et production lente nous permet de prouver plus facilement que les TM*s* sont clos par composition.

Lemme 3. *La composition de relation définies par deux transducteurs rationnels $[\![M_2]\!] \circ [\![M_1]\!]$ est définissable par un transducteur rationnel.*

Preuve. On suppose que M_1 et M_2 sont à la fois *init*-libres et *fin*-libres. Nous passons M_1 dans sa version en production lente M_1' et M_2 dans sa version en

lecture lente M'_2. Le but est de définir une composition séquentielle de M'_1 et M'_2 telle que le deuxième prenne comme entrée la production du premier. Le transducteur résultant sera appelé M_3.

Les états de M_3 sont des couples d'états issus de M'_1 et M'_2, identifiant la progression de l'exécution parallèlement dans M'_1 et M'_2. Il se définit par les règles données en figure 2.4. La règle du bas est présente dans le transducteur produit si les propriétés du haut sont vérifiées.

Une exécution de M_3 débute sur les paires d'états initiaux de M'_1 et M'_2 comme le montre la règle 2.4(a). Aucune sortie n'y est produite les transducteurs de base étant *init*-libres.

Ensuite, l'exécution peut continuer pour l'état (q, p) si une règle poursuit respectivement les états q et p dans M'_1 et M'_2.

Si la règle de M'_1 produit un symbole, il est nécessaire qu'une règle de M'_2 le consomme pour pouvoir suivre cette transition. La règle correspondante est générée par la construction 2.4(c).

L'exécution peut aussi continuer si il existe une transition sans production dans M'_1 à partir du premier état de la paire, comme l'illustre la règle 2.4(d). Dans ce cas, l'exécution se poursuit uniquement dans M'_1 sans rien produire et en ne modifiant pas l'état associé à M'_2.

L'inverse est aussi possible, comme le montre la règle 2.4(e), à savoir une ε-transition dans M'_2 ne dépendant pas de M'_1. La règle ne consomme rien en entrée et produit la sortie de la règle de M'_2, modifiant ainsi le deuxième état de la paire sans changer l'état représentant l'avancement dans M'_1.

Une transformation sera reconnue uniquement si elle atteint un état final composé de deux états finaux de M'_1 et M'_2 comme le montre la construction 2.4(b) . □

$$\frac{(q_0,\varepsilon) \in init_{M'_1} \qquad (p_0,\varepsilon) \in init_{M'_2}}{((q_0,p_0),\varepsilon) \in init_{M_3}} \text{(a)} \qquad \frac{(q_f,\varepsilon) \in fin_{M_1} \qquad (p_f,\varepsilon) \in fin_{M'_2}}{((q_f,p_f),\varepsilon) \in fin_{M_3}} \text{(b)}$$

$$\frac{q_1 \xrightarrow{u/a} q_2 \in rul_{M_1} \qquad p_1 \xrightarrow{a/w} p_2 \in rul_{M'_2} \qquad a \neq \varepsilon}{(q_1,p_1) \xrightarrow{u/w} (q_2,p_2) \in rul_{M_3}} \text{(c)}$$

$$\frac{q_1 \xrightarrow{u/\varepsilon} q_2 \in rul_{M_1} \qquad (q_1,p) \in Q_{M_3}}{(q_1,p) \xrightarrow{u/\varepsilon} (q_2,p) \in rul_{M_3}} \text{(d)} \qquad \frac{(q,p_1) \in Q_{M_3} \qquad p_1 \xrightarrow{\varepsilon/w} p_2 \in rul_{M_2}}{(q,p_1) \xrightarrow{\varepsilon/w} (q,p_2) \in rul_{M_3}} \text{(e)}$$

FIGURE 2.4 – règles de compositions

Lemme 4. *L'union de relations définie par deux transducteurs rationnels M_1 et M_2, $[\![M_2]\!] \cup [\![M_1]\!]$, est définissable par un transducteur rationnel.*

Preuve. Le transducteur obtenu correspond à l'union des états, règles, productions initiales et finales de M_1 et M_2 qui est bien un transducteur rationnel. □

Nous considérons des paires de mots d'entrée et de sortie. L'opérateur de concaténation "$\cdot$ " sur ces paires est défini par $(u, w) \cdot (v, w') = (uv, ww')$, et peut être étendu aux relations entre mots d'entrée et de sortie de la manière suivante, $X \cdot Y = \{x \cdot y \mid x \in X,\ y \in Y\}$.

Lemme 5. *La concaténation de relations définies par deux transducteurs rationnels $[\![M_2]\!] \cdot [\![M_1]\!]$ est définissable par un transducteur rationnel.*

Preuve. (construction) Supposons que les ensembles d'états de M_1 et M_2 sont disjoints, ce qui peut s'obtenir facilement en leur ajoutant une information d'appartenance à un transducteur. Le transducteur M_3 reconnaissant la concaténation est défini par le tuple :

$$(\Sigma_{M_1} \cup \Sigma_{M_2}, \Delta_{M_1} \cup \Delta_{M_2}, Q_{M_1} \cup Q_{M_2}, init_{M_1}, fin_{M_2}, rul_{M_1} \cup rul_{M_2} \cup trans).$$

où *trans* correspond à l'ensemble des transitions venant remplacer la production finale de M_1 et la production initale de M_2 afin d'assurer la jointure entre les deux transducteurs. Plus formellement *trans* est défini par :

$$\frac{(q, w) \in fin_{M_1} \qquad (q', w') \in init_{M_2}}{q \xrightarrow{\varepsilon/w \cdot w'} q' \in rul_{M_3}}$$

□

Définition 18. *Soit M le monoïde :*

$$M = (\{(a, \varepsilon) \mid a \in \Sigma\} \cup \{(\varepsilon, b) \mid b \in \Delta\} \cup \{(\varepsilon, \varepsilon)\}, \cdot, (\varepsilon/\varepsilon))$$

La classe $\mathcal{R}$ des transductions rationnelles *est le plus petit sous-ensemble de M satisfaisant les conditions suivantes :*

1. *$\{(a, \varepsilon) \mid a \in \Sigma\}$, $\{(\varepsilon, b) \mid b \in \Delta\}$, $\{(\varepsilon, \varepsilon)\}$ appartiennent à $\mathcal{R}$*
2. *Si $X, Y \in \mathcal{R}$, alors $X \cup Y$, $X \cdot Y \in R$,*
3. *Si $X \in \mathcal{R}$, alors $X^* \in \mathcal{R}$ où $X^* = \{(\varepsilon/\varepsilon)\} \cup X \cdot X^*$*

Proposition 4. *La classe des transducteurs rationnels correspond exactement à celle des transductions rationnelles.*

2.1.1.5 Domaines, images et typage

Nous montrons dans cette partie que le domaine, $dom(\llbracket M \rrbracket)$, et l'image, $im(\llbracket M \rrbracket)$, d'un transducteur rationnel M sont réguliers, et par conséquence, que le problème du typage est décidable pour cette classe de transducteurs.

Lemme 6. *Le domaine $dom(\llbracket M \rrbracket)$ est régulier.*

Preuve. Le domaine de M est reconnu par l'automate de mots A obtenu par projection sur l'entrée de M. Formellement, nous définissons l'automate $A = (\Sigma, Q_M, init_A, fin_A, rul_A)$, tel que $init_A = \{q \mid (q, w) \in init_M\}$, $fin_A = \{q \mid (q, w) \in fin_M\}$ et $rul_A = \{q \xrightarrow{u} q' \mid (q \xrightarrow{u/w} q') \in rul_M\}$.

□

Le lemme suivant montre que des restrictions de domaine, représentées par des schémas réguliers, peuvent directement s'intégrer dans la définition d'un transducteur rationnel.

Lemme 7. *Pour tout automate fini A et transducteur rationnel M il existe un transducteur rationnel M' tel que $\llbracket M' \rrbracket = \{(u, w) \in \llbracket M \rrbracket \mid u \in \mathcal{L}(A)\}$.*

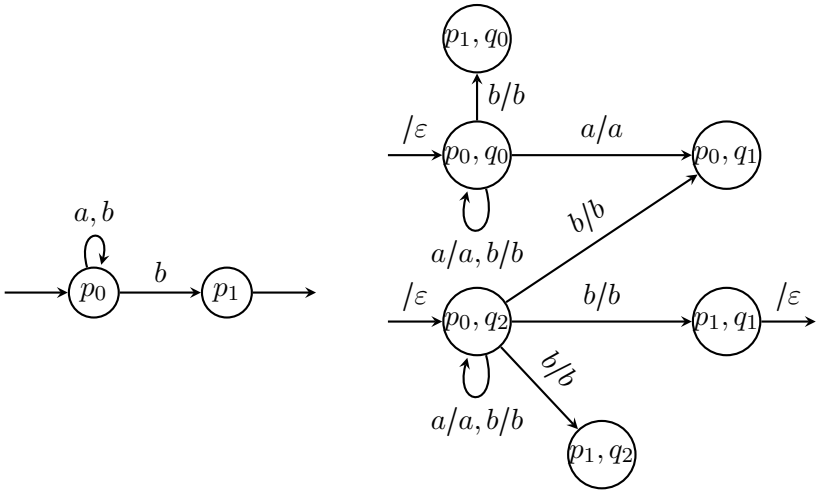
Preuve. La construction du transducteur M restreint au domaine d'un automate A repose sur la projection des états de A dans ceux de M. Pour correspondre plus facilement à la forme de l'automate, on suppose que M est en lecture lente.

Pour $A = (\Sigma, Q_A, init_A, fin_A, rul_A)$ représentant le domaine, nous définissons $M' = (\Sigma, \Delta, Q_M \times Q_A, init_{M'}, fin_{M'}, rul_{M'})$ tel que :

$$\begin{aligned}
init_{M'} &= \{((p,q), w) \mid p \in init_A,\ (q, w) \in init_M\},\\
fin_{M'} &= \{((p,q), w) \mid p \in fin_A,\ (q, w) \in fin_M\},\\
rul_{M'} &= \{(p,q) \xrightarrow{a/w} (p', q') \mid p \xrightarrow{a} p' \in rul_A,\ q \xrightarrow{a/w} q' \in rul_M\}\\
&\cup \{(p,q) \xrightarrow{\varepsilon/w} (p, q') \mid p \in Q_A,\ q \xrightarrow{\varepsilon/w} q' \in rul_M\}.
\end{aligned}$$

□

Exemple 16. *Reconsidérons le transducteur $M_{2.1}$ de la figure 2.1 avec l'objectif de le restreindre aux mots se terminant par un "b". Un automate A reconnaissant ce domaine ($((a+b)^*b)$) – que nous appelons le schéma – est donné en figure 2.5(a). Le transducteur obtenu en intégrant le schéma est visible dans la figure 2.5(b). On note que ce dernier n'est pas émondé. En supprimant les états n'aboutissant pas à un état final (on garde uniquement les états (p_0, q_2) et (p_1, q_1)), on s'aperçoit que le transducteur revient à une annotation de A où l'on recopie chaque caractère lu en entrée.*



(a) Schema A (b) Restriction M_1' de M_1 au schema A

FIGURE 2.5 – $M_{2.3}$ réduit au domaine $\mathcal{L}(A)$

Lemme 8. *L'image d'un transducteur rationnel est regulier.*

Preuve. Soit M un transducteur rationnel en production lente. L'image $im(\llbracket M \rrbracket)$ est reconnue par un automate de mots A' correspondant à la projection sur la sortie de M. Nous définissons cet automate A' par le tuple $(\Sigma, Q_M \cup \{I, F\}, \{I\}, \{F\}, rul_A)$ tel que I et F sont deux nouveaux états et :

$$\begin{aligned} rules_A = \quad & \{(q, w, q') \mid (q, u, w, q') \in rul_M\} \cup \{(I, w, q) \mid (q, w) \in init_M\} \\ & \cup \{(q, w, F) \mid (q, w) \in fin_M\} \end{aligned}$$

□

Le problème du typage pour la classe des TM est le suivant. Etant donnés deux automates de mots A et B et un TM M, est-ce que tout mot reconnu par A est transformé par M en un mot reconnu par B ?

Proposition 5. *Le problème du typage pour la classe des transducteurs rationnels est décidable.* [1]

Preuve. Il faut d'abord tester, si $\mathcal{L}(A) \subseteq dom(\llbracket M \rrbracket)$. Pour ce faire, il suffit de calculer l'automate qui reconnait le domaine de M, en suivant la construction de la preuve du Lemme 6. Ensuite, il faut tester si $im(\llbracket M \rrbracket(\mathcal{L}(A))) \subseteq im(\llbracket M \rrbracket(\mathcal{L}(B)))$. Ceci peut être fait en réduisant M par le schéma A suivant le Lemme 7, en calculant un automate B' qui reconnait l'image du transducteur réduit (Lemme 8), et en testant l'inclusion $\mathcal{L}(B') \subseteq \mathcal{L}(B)$. □

1. En appliquant l'algorithme de la preuve, le problème du typage peut même être calculé en temps polynomial, si on impose que A, B, et M sont déterministes.

2.1.1.6 Equivalence

L'absence de contraintes liées au déterminisme permettent une plus grande expressivité et parfois une simplification du modèle. Mais une telle liberté d'expression se fait souvent au détriment de la décidabilité de problèmes classiques des modèles théoriques tel que l'équivalence.

Théorème 2 ((Griffiths, 1968))**.** *L'équivalence est indécidable pour la classe des transducteurs rationnels.*

2.1.2 Transducteurs déterministes

La puissance des transducteurs rationnels remet aussi en cause l'existence de formes normales sur lesquelles sont basées les approches d'apprentissage par inférence grammaticale. Deux restrictions possibles semblent naturelles. Sémantiquement, on peut se restreindre à des transducteurs qui définissent des fonctions et non des relations plus générales. On peut également imposer syntaxiquement le déterminisme, ce qui est plus contraignant.

Nous commençons avec la classe de transducteurs de mots déterministes, souvent appelés *transducteurs sous-séquentiels*. Ce deuxième nom, introduit par Schützenberger (1975), est dû au fait qu'ils peuvent définir la classe des fonctions sous-séquentielles.

2.1.2.1 Determinisme

L'idée principale d'une machine déterministe est de fixer le comportement de la machine afin que pendant son exécution, une seule possibilité de continuation lui soit offerte.

Définition 19. *Un* transducteur déterministe de mots *ou un* transducteur sous-séquentiel*, noté* dTM*, est un transducteur rationnel* $M = (Q, init, fin, rul)$ *à lecture lente et sans* ε*-transition. Les initialisations* $init : Q \times \Sigma$ *et finalisations* $fin : Q \times \Sigma^*$ *sont maintenant des fonctions partielles dépendant de l'état initial ou final, le domaine de* $init$ *contenant au plus un élément. Les règles sont définies par* $rul : (Q \times \Sigma) \times (\Delta^* \times Q)$*, une fonction partielle dépendant de l'état et du symbole lu.*

Pour la suite, on supposera que chaque dTM considéré est émondé, forme qui peut facilement être obtenue par la suppression de tout état non accessible (à partir des état initiaux) et non co-accessible (à partir duquel aucun état final ne peut être atteint).

Il est évident que chaque transducteur déterministe M définit une transformation $[\![M]\!]$ pouvant être vue comme une fonction partielle, fonction pouvant

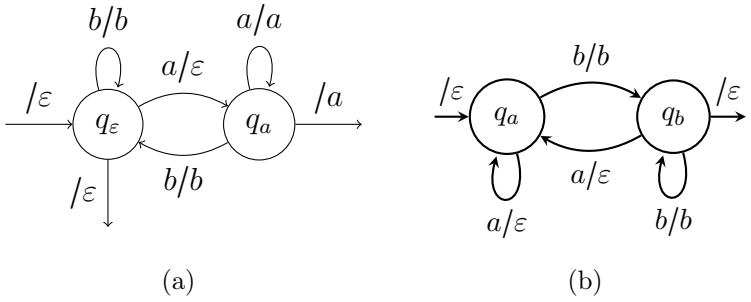


FIGURE 2.6 – Exemples de dTMs

également être définie pour chaque état q du transducteur, notée $[\![M]\!]_q$. Ceci nous permet d'associer à un mot $u \in dom[\![M]\!]$ son image unique $[\![M]\!](u)$, et similairement $[\![M]\!]_q(u)$ pour l'image unique d'un mot $u \in dom[\![M]\!]_q$, à partir de l'état q.

2.1.2.2 Fonctions sous-séquentielles

Nous illustrons dans cette section la richesse et les limitations de la classe de fonctions définies par les transducteurs déterministes de mots.

Définition 20. *Une fonction est appelée* sous-séquentielle *si et seulement si elle est égale à $[\![M]\!]$ pour un transducteur déterministe de mots M.*

L'exemple suivant illustre le fait que, contrairement aux transducteurs rationnels, l'étape de finalisation peut être indispensable pour les transducteurs déterministes.

Exemple 17. *Le dTM $M_{2.6(a)}$ en figure 2.6(a) transforme tout mot composés de lettres "a" et "b" en supprimant chaque "a" suivi d'un b. Comme les dTM$s parcourent les mots d'entrée de gauche à droite, $M_{2.6(a)}$ ne peut pas anticiper le fait qu'un b se trouve derrière un a. Après avoir lu un a, il doit attendre de lire le prochain caractère avant de pouvoir décider la production appropriée. Une anticipation bornée demande d'intégrer les symboles lus dans les états et mène rapidement à une multiplication importante du nombre d'états.*

Après avoir lu un "a", le fait de se trouver dans un état final apporte une information supplémentaire permettant de produire le a en attente. Sans étape de finalisation, la fonction ne pourrait pas être définie par un dTM.

Les transducteurs déterministes ne recouvrent pas l'intégralité des fonctions exprimables par des transducteurs rationnels. Prenons comme contre exemple la fonction $[\![M_{2.1}]\!]$ avec le transducteur $M_{2.1}$ donné en figure 2.1. Le

problème est que ce transducteur doit produire une sortie de taille non-bornée dépendant du fait que la dernière lettre soit un "a" ou un "b", ce qui ne se décidera que dans le futur. La seule solution serait d'attendre de lire ce dernier caractère pour décider de la sortie et donc de mémoriser dans l'état chaque mot lu jusque là, ce qui nous donnerait un nombre infini d'états.

Exemple 18. *Par contre, si on restreint le domaine de $M_{2.1}$ aux mots qui se terminent par "a", on obtient une fonction sous-séquentielle. La sortie peut être produite de manière spéculative, et validée uniquement quand la dernière lettre du mot a été lue. Cette fonction partielle est définie par le dTM $M_{2.6(b)}$ en figure 2.6(b). En effet, le transducteur ne peut se terminer qu'après avoir lu un "b" ce qui correspond à l'état q_b final. Si il a lu un "a", il retourne ou reste en q_a.*

Comme précédemment évoqué dans la simplification des TM*s*, l'absence d'ε-transitions dans les dTM*s* nous assure qu'il n'est pas possible de se ramener à une forme *fin*-libre. Le transducteur $M_{2.6(a)}$ illustre un cas où la suppression de la production finale est impossible. Par contre, rien ne nous empêche de pousser la production initiale.

Lemme 9. *Tout dTM peut être ramené à un dTM init-libre.*

Preuve. Considérons un dTM M. Le dTM *init*-libre M' garde le même alphabet d'entrée et de sortie, les mêmes finalisations, l'ensemble des états correspond à celui de M auquel on ajoute un nouvel état initial $rul_{M'} = rulM \cup \{q_i\}$. L'initialisation est remplacée par $init_{M'} = \{(q_i, \varepsilon)\}$. Les règles, quant à elles, sont enrichies de nouvelles règles venant de l'état initial. Plus formellement, pour $(q_0, w) \in init_M$:

$$rul_{M'} = rul_M \cup \{q_i \xrightarrow{a/w\cdot w'} q \mid q_0 \xrightarrow{a/w'} q \in rul_M\}.$$

□

2.1.2.3 Équivalence

Comme évoqué précédemment, la décidabilité de l'équivalence est un problème classique permettant d'estimer la complexité du modèle et les possibilités d'utilisations qui lui sont offertes. Il existe plusieurs manières de prouver que l'équivalence de deux dTM*s* est décidable. Ici, nous allons nous intéresser à deux manières de résoudre ce problème : soit en passant par l'utilisation de morphismes ; soit, de manière plus classique, en passant par la représentation unique d'un transducteur.

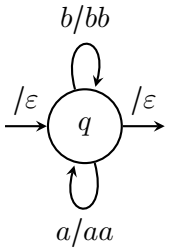
La méthode proposée par Plandowski (1995), basée sur les morphismes, permet d'obtenir un algorithme de décision efficace de la décidabilité de l'équivalence. Étant utilisée pour d'autres classes de transduction (présentées ultérieurement dans cette thèse) nous allons détailler ici les mécanismes principaux de cette preuve.

Définition 21. *Un* morphisme de mots *est une fonction totale* $\mu : \Sigma^* \to \Delta^*$ *tel que* $\mu(u \cdot v) = \mu(u) \cdot \mu(v)$.

Un morphisme est donc défini par les valeurs possibles que peuvent prendre chaque lettre de Σ et représenté par l'ensemble $\{(a, \mu(a)) \mid a \in \Sigma\}$. Il peut être aussi représenté par un transducteur déterministe M, complet, à un seul état q initial et final, ne produisant rien lors de la phase d'initialisation et finalisation. Il faut également que pour toute lettre $a \in \Sigma$, $q \xrightarrow{a/\mu(a)} q \in Q_M$. L'exemple suivant illustre cette représentation d'un morphisme par un transducteur.

Exemple 19.

*Considérons le morphisme μ répétant deux fois chaque lettre d'un mot défini sur $\Sigma = \{a, b\} = \Delta$. μ est représenté par l'ensemble {(a,aa),(b,bb)}. Le résultat de l'application du morphisme μ au mot "aab" est $\mu(aab) = \mu(a) \cdot \mu(a) \cdot \mu(b) =$ "aaaabb". Elle correspond à la fonction reconnue par le d*TM *ci-joint .*



La taille d'un morphisme μ est la taille de ce transducteur, soit $\sum_{a \in \Sigma} |\mu(a)|$.

Pour réduire le problème d'équivalence, nous allons nous intéresser aux exécutions possibles d'un transducteur déterministe M. Une exécution de M sur un mot u du domaine de $[\![M]\!]$ correspond à la succession de règles appliquées par M sur u pour le transformer. Chaque étape de cette exécution peut être identifiée de manière unique par le mot u et une position dans ce mot. Nous représenterons une exécution pour un mot u par la paire $u * \beta$ où $\beta \subseteq pos(u) \to rul_M$ est une fonction associant à chaque position de u une règle du transducteur. La notion de validité et de complétude d'une telle exécution ne dépendant que de l'entrée, elles gardent les mêmes définitions que pour les automates déterministes de mots.

La production associée au mot u est :

$$[\![M]\!](u) = init(gch(\beta(1))) \cdot prod(\beta(1)) \cdot \ldots \cdot prod(\beta(k)) \cdot fin(dt(\beta(k))).$$

Exemple 20. *Prenons $M_{2.6(a)}$. L'exécution complète pour le mot "aba" est $u * \beta$ et peut être présentée comme la suite de règles suivante :*

$$q_\varepsilon \xrightarrow{a/\varepsilon} q_a \xrightarrow{b/b} q_\varepsilon \xrightarrow{a/\varepsilon} q_a$$

où $\beta(i)$ correspond à la i-ième règle de cette succession. $M_{2.6(a)}$ étant déterministe, u peut être directement retrouvé sur l'entrée des règles. De plus comme l'initialisation ne produit rien, $(q_\varepsilon, \varepsilon) \in init$, nous obtenons donc :

$$\begin{aligned}[[M_{2.6(a)}]](aba) &= prod(\beta(1)) \cdot prod(\beta(2)) \cdot prod(\beta(3)) \cdot fin(q_a) \\ &= \varepsilon \cdot b \cdot \varepsilon \cdot a\end{aligned}$$

Une *exécution parallèle complète* pour deux transducteurs M_1 et M_2 appartient à l'ensemble $\{u * \beta_1 * \beta_2 \mid u \in dom(M_1) \cap dom(M_2)\}$ où β_i est une exécution complète de M_i.

Lemme 10. *Les deux problèmes de décision suivants sont équivalents modulo réduction en temps polynomial :*

1. *L'équivalence de deux morphismes sur un langage régulier donné par un* DFA
2. *L'équivalence de deux transducteurs de mots déterministes*

Preuve. "1 ⇒ 2". Soit A un DFA avec signature Σ, soient μ_1 et μ_2 des morphismes avec ce même alphabet pour l'entrée. Ces morphismes sont des transducteurs déterministes. Nous pouvons donc intégrer A dans μ_i suivant le Lemme 7 pour tout $i \in \{1, 2\}$. Les résultats sont deux autres transducteurs M_i, dont le domaine est $\mathcal{L}(A)$, et qui produisent la même sortie que μ_i sur tous les mots du domaine. Ces transducteurs sont déterministes puisque A et μ_i le sont. Pour tester l'équivalence de μ_1 et μ_2 sur $\mathcal{L}(A)$, il suffit donc de tester l'équivalence de M_1 et M_2.

"2 ⇒ 1". On va réduire le problème d'équivalence de transducteurs déterministes au problème d'équivalence de morphismes sur un langage algébrique.

Considérons deux dTM*s* M_1 et M_2 de Σ^* vers Δ^*. Dans un premier temps, il faut vérifier qu'ils sont définis sur le même domaine d'entrée. Comme le montre le Lemme 6, ce test revient à tester l'équivalence de deux automates de mots déterministes, correspondant à la suppression de la sortie de M_1 et M_2. Ce test est réalisable en temps polynomial.

Ces transducteurs étant définis sur le même domaine, on représente par le DFA A le langage des exécutions parallèles de M_1 et M_2, deux dTM*s* *init*-libres, auquel on ajoute en fin d'exécution la paire des états finaux. A est donc défini sur l'alphabet $\Sigma_A = rul_{M_1} \times rul_{M_2} \cup \{(q_1, q_2) \mid q_1 \in dom(fin_{M_1}),\ q_2 \in dom(fin_{M_2})\}$. L'ensemble des états $Q_A \subseteq (Q_{M_1} \times Q_{M_2}) \cup q_f$ correspond aux états atteints par les transducteurs pendant l'exécution parallèle et un nouvel état final unique q_f. Les règles, quant à elles, sont créées de la manière suivante pour chaque exécution $u * \beta_1 * \beta_2$:

$$\forall i \in pos(u),\ (gch(\beta_1(i)), gch(\beta_2(i))) \xrightarrow{(\beta_1(i), \beta_2(i))} (dt(\beta_1(i)), dt(\beta_2(i))) \in rul_A$$

$$\text{pour } |u| = k,\ (dt(\beta_1(k)), dt(\beta_2(k))) \xrightarrow{(dt(\beta_1(k)),dt(\beta_2(k)))} q_f$$

On introduit deux morphismes μ_1 et μ_2 de $\mathcal{R}$ à Δ^* tel que $\mu_i((r_1, r_2)) = prod(r_i)$ pour $r_i \in rul_{M_i}$ et $\mu_i(q_1, q_2) = w_i$ pour $(q_i, w_i) \in fin_i$.

Nous obtenons ainsi que, pour M_1 et M_2, deux dTM*s* *init*-libres, et pour A l'automate représentant les exécutions parallèles de ces transducteurs, $\forall u \in A.\ (u, \mu_i(u)) = [\![M_i]\!]$. De plus, d'après le lemme 9, tout dTM pouvant être redéfini sous forme *init*-libre, nous pouvons réduire l'équivalence de dTM*s* à celle de morphismes sur un DFA. □

Théorème 3 (Plandowski (1995))**.** *Le problème d'équivalence de morphismes sur un langage algébrique peut être décidé un temps polynomial (dans la taille des morphismes et la grammaire algébrique définissant le language regulier).*

Ce théorème dépend de notions trop conséquentes pour être abordées dans cette thèse. Par contre, il permet d'obtenir des résultats de décidabilités tel que le corollaire suivant, correspondant au Théorème 37 de la thèse de Plandowski (1995).

Corollaire 2. *L'équivalence de deux dTM*s* peut être décidée en temps polynomial.*

Ce corollaire découle directement du lemme 10 et du théorème 3. Il peut également être prouvé par la définition d'une forme normale minimale représentant tout dTM, afin d'en calculer l'égalité directe.

2.1.2.4 Normalisation de sortie

L'existence d'une forme normale unique est également fondamental à l'apprentissage de transducteurs déterministes. Nous définissons dans cette section une normalisation des transducteurs sous séquentiels.

Dans une première étape, nous considérons la normalisation de la production de sortie d'un transducteur. Si nous voyons les dTM*s* comme des machines qui, grâce à leur déterminisme, peuvent produire la sortie pendant l'évaluation d'un mot, alors la manière la plus naturelle de produire cette sortie est de le faire le plus tôt possible. Cela revient à pousser la sortie vers le début du transducteur afin qu'elle puisse être produite dès que la partie du mot lue nous permet de prendre cette décision.

Exemple 21. *Considérons par exemple le dTM $M_{2.7}$. On peut remarquer que le langage de l'image est de la forme $((ba)|b)^*b$ ce qui implique que chaque mot de sortie est préfixé par "b" . Ce préfixe pourrait être directement sorti pendant la phase d'initialisation. Cette remarque est confortée par le fait que le plus grand préfixe des mots produits à partir de l'état inital q_0 est $lcp([\![q_0]\!]) = b$.*

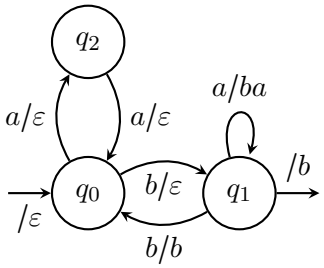


FIGURE 2.7 – Transducteur non normalisé

2.1.2.5 Transducteurs travailleurs

La notion de transducteur *travailleur*, résultant de cette idée de sortie normalisée, associe à chaque règle le plus de sorties possibles connues à ce point. Plus formellement :

Définition 22. *Un d*TM M *est* travailleur *(noté ed*TM*) si et seulement si pour tout état $q \in Q$, le plus grand préfixe commun à chacune des productions à partir de cet état, noté $\Lambda(q)$ est vide :*

$$\Lambda(q) = lcp(im([\![M]\!]_q)) = \varepsilon$$

Proposition 6. *Tout transducteur déterministe peut être défini par un transducteur travailleur équivalent.*

Cette existence est prouvée une première fois par Choffrut (1979). Dans une étude plus approfondie, Choffrut (2003) décrit la construction d'un transducteur travailleur à partir d'un transducteur déterministe.

L'idée est que, dans un transducteur déterministe de mot M quelconque, il peut exister des états q tel que $\Lambda(q) \neq \varepsilon$. Dès lors, il faut chercher à produire la sortie le plus tôt possible dans l'exécution d'un transducteur, ce qui nous amène à *remonter* un mot dans un transducteur, le déplacer dans une règle précédant la règle dans laquelle il apparaissait initialement. Si cette opération est faite localement, on parlera de remonter un mot *à travers un état* q. Cela se fait en retirant un préfixe des productions des règles sortant de q pour l'ajouter en fin des productions des règles entrant en q.

Considérons maintenant une règle sortant de cet état $q \xrightarrow{a/w} p$. Dans le transducteur travailleur M' équivalent, une partie de cette sortie devrait déjà être produite précédemment, ce qui revient à retirer une partie de la production de cette règle. Cependant, une portion de ce préfixe peut être produit après l'état p. Cette étape de suppression doit donc suivre l'ajout de $\Lambda(p)$ à la fin de w. Grâce à cette méthode, on peut réévaluer les productions du

transducteur de manière globale, séparément sur chaque règle afin de créer un nouvel ensemble de règles de la manière suivante :

$$rul_{M'} = \{q \xrightarrow{a/\Lambda(q)^{-1}\cdot w\cdot\Lambda(p)} p \mid q \xrightarrow{a/w} p \in rul_M\}$$

sans oublier les phases d'initialisation et finalisation auxquelles sont respectivement ajoutées et retirées une partie possiblement vide.

$$init_{M'} = \{(q, w\cdot\Lambda(q)) \mid (q, w) \in init_M\}$$

$$fin_{M'} = \{(q, \Lambda(q)^{-1}\cdot w) \mid (q, w) \in init_M\}$$

Par définition, le fait d'être travailleur dépend uniquement de l'automate obtenu par projection sur la sortie, suivant la construction détaillée dans la preuve du lemme 8. L'automate $A = (Q, \{I\}, \{F\}, rul)$ ainsi obtenu permet de calculer les $\Lambda(q)$ et est le modèle utilisé par la suite.

Beal et Carton (2001) proposent une manière itérative pour identifier une à une les lettres des préfixes communs $\Lambda(q)$ et les tirer vers le début de l'automate. Une étape consistant à identifier la première lettre de ce mot, que l'on note $\lambda(q)$, et de la déplacer sur l'automate.

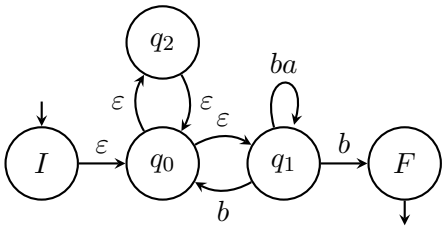


FIGURE 2.8 – Exemple de projection de la sortie

Prenons, pour illustrer les différentes étapes nécessaires au calcul de ce préfixe, la projection sur la sortie $A_{2.8}$ du transducteur $M_{2.7}$. Dans ses travaux, Mohri (1994) avait proposé une méthode locale permettant de faire remonter la production vers le début du transducteur en calculant pour chaque état, le préfixe des productions des règles qui en sortaient. Cette méthode repose sur le fait que si une ε-règle sort d'un état q, tel que $\Lambda(q) \neq \varepsilon$, cette sortie commune serait remontée sur l'ε-règle à partir d'un certain nombre d'itérations. Une difficulté vient de l'existence de boucles d'ε-transitions empêchant parfois de remonter une partie du préfixe commun. Par exemple, dans l'automate $A_{2.8}$, "b" pourrait être remonté à travers q_1, chaque règle sortant de cet état étant préfixée par un "b". Une fois cela fait, aucune autre normalisation

serait possible alors que chaque mot commence par un "b", lettre qui devrait atterrir dans la première règle de l'automate.

En regardant de plus près l'automate A_ε, correspondant au sous automate de A ne gardant que les ε-règles, nous pouvons observer l'apparition de *groupes fortement connexes* (i.e. groupe où chaque état peut atteindre tout membre du groupe et est atteint à partir de ces états). Les états composant un tel groupe partagent un préfixe commun sur la production qui les suit. Ce préfixe, possiblement non vide, est impossible à propager localement par la méthode de Mohri (1994).

Par un simple parcours en profondeur, nous pouvons définir la fonction $\mathit{connect} : Q \to Q$ qui associe à chaque état, un représentant unique du groupe fortement connexe auquel il appartient.

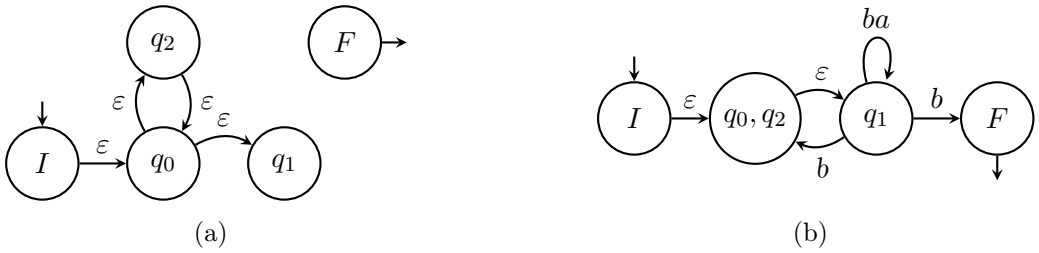


FIGURE 2.9 – Détection et représentation des groupes connexes

Exemple 22. *Si nous projetons l'automate $A_{2.8}$ sur les ε-règles, nous obtenons l'automate $A_{2.9(a)}$ qui permet d'identifier le groupe fortement connexe formé par q_0 et q_2. Les autres états quant à eux ne forment pas de groupes fortement connexes et seront donc traités distinctement.*

Calculer les première lettres associées à chaque état revient à calculer ces mêmes premières lettres sur l'automate $A_{2.9(b)}$, le préfixe de $\mathcal{L}(A_{q_0})$ étant celui de $\mathcal{L}(A_{q_2})$.

Beal et Carton (2001) cherchent, après avoir identifié ces groupes, à déterminer la première lettre, si elle existe, des préfixes communs des langages définis à partir de chaque état d'un ensemble fortement connexe : $X_q = \{q' \mid \mathit{connect}(q') = q\}$. Cette lettre ne correspond pas toujours à $\lambda(q)$ mais dans tout les cas contribue à son calcul. On identifiera cette lettre candidate par $\mathit{candidat}(q)$.

Calculons le candidat d'un ensemble X_q donné. Ce candidat dépend principalement des labels des règles sortant des états de l'ensemble X_q : $\mathrm{LAB}_q = \{w \mid q' \xrightarrow{\varepsilon} q'', q' \in X_q\}$. Si deux mots non vides de cet ensemble diffèrent dès

la première lettre, alors il ne peut exister de préfixe commun, ce qui implique que $\lambda(q) = candidat(q) = \varepsilon$.

Si toutes les règles sortant de cet ensemble connexe, i.e. toute règle $q' \xrightarrow{w} q'' \in rul$ telle que $q' \in X_q$ et $q'' \notin X_q$, sont productives alors la lettre préfixe $\lambda(q)$ est la lettre préfixe de l'ensemble (LAB_q). A fortiori, cela fixe également le candidat.

Pour estimer la valeur de $candidat(q)$ il faut s'intéresser à l'ensemble des labels présents sur les règles sortant des états de X_q. Si deux de ces labels, différents de ε, diffèrent dès la première lettre il ne pourra avoir en aucun cas un préfixe commun, et donc $\Lambda(q') = candidat(q) = \varepsilon$. Si aucune ε-transition ne sort de cet ensemble, i.e. pour tout $q' \in X_q$ il n'existe pas d'état $q' \notin X_q$ tel que $p \xrightarrow{\varepsilon} q'$, alors

Dans les autres cas, nous pouvons juste supposer que le λ_q corresponde au symbole commun des labels autres que ε, ce que l'on pourra confirmer ou infirmer par la suite. Dans le cas où aucune des règles sortant des états de X_q n'est productive, aucune information directe n'est donnée sur cette lettre commune. Le candidat prend donc pour valeur $\perp$. $\perp$ est un symbole défini de sorte que $lcp(\perp, w) = w$ pour tout $w \in \Delta^*$.

Le candidat d'un état q' correspond donc au préfixe commun des candidats des états appartenant à X_q avec $q = connect(q)$. Cela nous donne formellement, pour $q' = connect(q)$:

$$candidat(q) = \begin{cases} lcp\left(\begin{array}{c} \{a \mid p \xrightarrow{a \cdot w} p' \in rul.\ p \in X_{q'},\ a \in \Sigma\} \\ \cup \{\varepsilon \mid F \cap X_{q'} \neq \varnothing\} \end{array} \right) & \text{si défini} \\ \perp & \text{sinon} \end{cases}$$

Le calcul de $\lambda(q)$ pour un représentant q revient maintenant à vérifier que le candidat de cet état est compatible avec les candidats de chaque état atteignable par des ε-transitions, i.e. :

$$\lambda(q) = lcp(\{candidat(q)\} \cup \{candidat(connect(p)) \mid q' \in X_q,\ q' \xrightarrow{\varepsilon}_* p \in rul\})$$

Cette équation est interprétable par un unique parcours en profondeur de l'automate A_ε.

Exemple 23. *Comme nous pouvons le voir dans la figure 2.9, seuls q_0 et q_2 sont fortement connexes et forment un ensemble, les autres états composant chacun un singleton. Nous devons définir pour chaque état, sauf q_2, une première lettre candidate. Si l'on regarde l'automate $A_{2.9(b)}$, on s'aperçoit que les états des ensembles X_{q_0} et X_I ont uniquement des ε-règles qui les suivent, et donc aucune information sur le candidat. F étant final, il accepte ε dans son langage, ce qui implique qu'il ne peut y avoir de première lettre commune. Les*

labels de chaque règle sortant de q_1 ont des labels non vides préfixés par un "b", ce qui nous donne que le candidat "b" est la première lettre commune $\lambda(q_1)$.

$$\begin{aligned} candidat(q_0) &= \bot \\ candidat(I) &= \bot \\ candidat(q_1) &= b = \lambda(q_1) \\ candidat(F) &= \varepsilon = \lambda(F) \end{aligned}$$

En explorant l'automate en suivant les ε-règles on obtient que

$$\lambda(q_2) = \lambda(q_0) = lcp(candidat(q_0), candidat(q_1)) = lcp(\bot, b) = b$$
$$\lambda(I) = lcp(candidat(I), candidat(q_0), candidat(q_1)) = lcp(\bot, \bot, b) = b$$

b peut être remonté à travers q_0, q_1 et q_2, I étant l'état initial.

Un dernier parcours de cet automate est nécessaire afin de répercuter le déplacement de chacune de ces lettres par l'application de l'équation suivante à chacune des règles.

$$\begin{aligned} rul_{A'} = \quad & \{q \xrightarrow{\lambda(q)^{-1}\cdot w\cdot\lambda(p)} p \mid q \xrightarrow{w} p \in rul_A.\ q \notin init\} \\ & \cup \{q_i \xrightarrow{w\cdot\lambda(q)} q \mid q_i \xrightarrow{w} q \in rul.\ q_i \in init\} \end{aligned} \tag{2.1}$$

La proposition 3.1 de Beal et Carton (2001) prouve que cet algorithme calcule $\lambda(q)$ pour chaque état q de l'automate A. Le coût de l'opération se résume à trois fois le parcours en profondeur de l'automate.

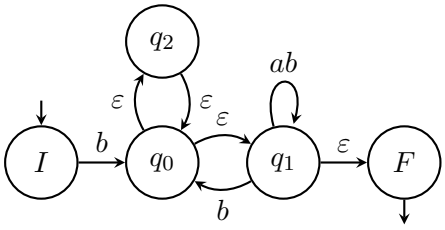


FIGURE 2.10 – Automate normalisé

Exemple 24. *b est la lettre préfixe pour les états I, q_0, q_1 et q_2 ce qui donne, après l'avoir déplacée, l'automate $A_{2.10}$. Certaines règles ont uniquement une lettre supprimée au début de leur label, comme $q_1 \xrightarrow{b} F$, où le label est remplacé par le mot $\lambda(q_1)^{-1}\cdot b\cdot\lambda(F) = b^{-1}\cdot b\cdot\varepsilon = \varepsilon$. D'autres récupèrent uniquement une lettre à la fin de leurs labels comme $I \xrightarrow{b} q_0$ qui, partant de l'état initial, ne peut pas remonter le "b" plus tôt et obtient comme label le mot $\varepsilon\cdot\lambda(q_0) = b$. Les*

autres règles modifiées voient leurs labels à la fois tronquer à gauche et enrichi à droite. Cette opération peut être invisible, comme l'illustre l'ε-règle $q_2 \xrightarrow{\varepsilon} q_0$. En effet, elle reste inchangée puisque $\lambda(connect(q_2))^{-1} \cdot \varepsilon \cdot \lambda(connect(q_0)) = b^{-1} \cdot b = \varepsilon$. D'autres règles voient une lettre de leur label se faire transposer, comme le montre $q_1 \xrightarrow{ba} q_1$ dont le label se décale d'un rang pour devenir $b^{-1} \cdot ba \cdot b = ab$.

Soit P_A la taille maximale du plus grand préfixe commun d'un état de A, sauf l'état initial, les deux propositions suivantes découlent directement de la proposition 3.3 de Beal et Carton (2001) et assurent que l'on arrive à l'automate normalisé reconnaissant le même langage dans un nombre d'étapes bornées par P_A.

Proposition 7. *L'automate, obtenu après déplacement des lettres, définit le même langage que l'automate de départ.*

Preuve. (esquisse) Considérons un automate de mots A et l'automate A' obtenu après le déplacement des lettres. Nous pouvons prouver par induction sur la taille des mots que $\mathcal{L}(A')_q = \lambda_A(q)^{-1} \cdot \mathcal{L}(A)$ pour tout état $q \in Q \setminus init$. Une fois cette hypothèse prouvée, il suffit de voir qu'un mot w est dans $\mathcal{L}(A)$ ssi il existe un mot $w'' \in \mathcal{L}(A)_q$ et une règle venant de l'état initial $I \in init$ vers q, $I \xrightarrow{w''} q \in rul_A$ tel que $w = w' \cdot w''$. Par construction, $I \xrightarrow{w'' \cdot \lambda(q)} q \in rul_{A'}$ donc par l'hypothèse $w' \cdot \lambda(q) \cdot \lambda(q)^{-1} \cdot w'' = w$ est dans $\mathcal{L}(A')$. □

Proposition 8. *Soit A' l'automate obtenu par une étape de normalisation de A alors $P_{A'} = P_A - 1$.*

Preuve. Si $\mathcal{L}(A')_q = \lambda_A(q)^{-1} \cdot \mathcal{L}(A)$ pour tout état $q \in Q \setminus init$ les préfixes communs sont automatiquement réduits d'au moins une lettre, s'ils ne sont pas déjà vides, ce qui est donc le cas pour le plus grand. □

À partir de cela, il suffit de montrer que tout préfixe commun, et particulièrement P_A, est borné dans la taille de A. Cela permet de fixer une borne sur le calcul de $\Lambda(q)$ pour chaque état, et prouve que la normalisation d'un automate de mots peut être effectuée en temps polynomial.

Proposition 9. *Soit A un automate. Pour tout état $q \in Q$, $|\Lambda(q)|$ est borné par $|A|$.*

Preuve. On se ramène à l'automate émondé en supprimant les états qui ne sont pas co-accessibles. Pour montrer que le plus grand préfixe commun des mots reconnus à partir de q est de taille bornée, il suffit de montrer qu'il existe un mot de taille bornée, le *lcp* étant de taille inférieure ou égale à ce mot. Pour

cela, il suffit de montrer qu'il existe une exécution de q vers un état final dont le nombre de règles est borné.

Considérons un état q de A, qui est co-accessible, impliquant qu'il existe une exécution $u * \beta$ tel que $gch(\beta(1)) = q$ et $dt(\beta(|u|)) \in dom(fin)$. Supposons maintenant que le nombre de règles la composant est supérieur au nombre d'états, alors il existe une portion de l'exécution démarrant et terminant dans un même état. On peut donc se ramener à une règle de taille inférieure. Cet argument de pompage permet de borner la taille d'une exécution au nombre d'états de l'automate. □

2.1.2.6 Minimisation

Le fait d'être travailleur n'est pas suffisant pour assurer l'unicité du représentant d'une fonction sous-séquentielle. Par exemple $M_{2.11(a)}$ reconnaissant la copie de mots sur a^* pourrait être exprimé par l'*ed*TM $M_{2.11(b)}$, composé d'un seul état en fusionnant q_0 et q_1, qui reconnaît une même transformation $[\![M_{2.11(a)}]\!]_{q_1} = [\![M_{2.11(a)}]\!]_{q_0}$.

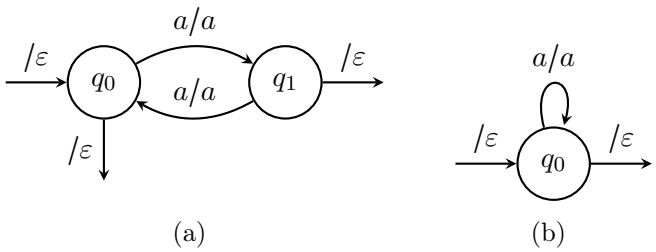


FIGURE 2.11 – DFA et sa version minimale

Un transducteur travailleur M se minimalise de la même manière qu'un DFA, en tenant compte que deux états finaux ne peuvent pas être équivalents si les mots produits lors de la finalisation sont différents. L'autre propriété est que deux états équivalents ont, pour chaque règle sortante lisant une même entrée, une même sortie produite. Pour sa minimisation, on adapte un algorithme de minimisation classique, tel que celui de Hopcroft, marchant par raffinement d'ensembles d'états équivalents. Dans notre cas, on ne débute plus avec d'une part les états finaux et d'autre part avec le reste des états, mais cette fois ci avec l'ensemble de partitions de Q suivant :

$$\left(\bigcup_{w \in im(fin)} \{\{q \mid (q,w) \in fin\}\} \right) \cup \{Q \setminus \{q \mid (q,w) \in fin\}\}$$

Pour résoudre le problème d'équivalence de transducteurs, il reste à montrer l'unicité du transducteur travailleur minimal représentant une transduc-

tion, modulo un renommage des états. Pour cela, Choffrut (2003) montre comment construire ce représentant directement à partir de la transformation. Il montre également l'existence d'un unique morphisme, passant de tout transducteur qui la reconnait vers ce représentant, ce qu'il prouve dans la proposition 2.

Une autre façon de le voir est de considérer directement les transducteurs travailleurs et les propriétés que partagent deux *ed*TM*s* représentant une même transduction. Commençons par regarder les sorties associées à un mot préfixe.

Définition 23. *Pour $u \in prefixe(M)$ nous introduisons la sortie préfixe correspondant à la sortie associée à u dans le transducteur M :*

$$sortie_M(u) = w \cdot prod(\beta(1)) \cdot \ldots \cdot (\beta(|u|))$$

où $u * \beta$ est une exécution valide sur M telle qu'il existe $(gch(\beta(1)), w) \in init$.

Lemme 11. *Soient M_1 et M_2 reconnaissant une même transduction,*

$$\forall u \in dom(M_1),\ sortie_{M_1}(u) = sortie_{M_2}(u)$$

Preuve. Ce lemme peut se prouver par l'absurde. Supposons qu'il n'est pas vérifé. M_1 et M_2 reconnaissant la même transduction, on peut supposer sans perte de généralité qu'il existe un mot $u \in \Sigma$ et un mot $w \in \Delta^*$ non vide, tels que $sortie_{M_1}(u) = sortie_{M_2}(u) \cdot w$. Ceci implique que $sortie_{M_2}(u) \cdot w$ est préfixe de toute sortie produite en transformant un mot préfixé par u. Autrement dit, l'état q, atteint après avoir lu le mot u, a un préfixe commun $\Lambda(q)$ différent de ε, ce qui contredit le fait que le transducteur est travailleur. □

À partir de cela, si nous voyons les transformations reconnues à partir des états comme les résiduels de la transduction pour chacun des mots les atteignants, nous pouvons traduire cette propriété de la manière suivante.

Lemme 12. *Soit M_1 et M_2 deux ed*TM*s équivalents, $q \in Q_{M_1}$ ssi il existe un état $p \in Q_{M_2}$ tel que $[\![M_1]\!]_q = [\![M_2]\!]_p$.*

Preuve. Considérons un mot $u \in prefixe(dom[\![M_1]\!])$. M_1 et M_2 étant équivalents, ils sont définis sur le même domaine et représentent une même transduction. Pour toute continuation v du mot u dans M_1, $u \cdot v \in dom[\![M_1]\!]$ implique donc que $u \cdot v \in dom[\![M_2]\!]$ et $[\![M_1]\!](u \cdot v) = [\![M_2]\!](u \cdot v)$.

Il existe donc deux exécutions complètes $(u \cdot v) * \beta_1$ et $(u \cdot v) * \beta_2$ sur M_1 et M_2 dont les sorties suivantes sont égales :

$$\overbrace{init(gch(\beta_1(1))) \cdot prod(\beta_1(1)) \cdot \ldots \cdot prod(\beta_1(|u|))}^{sortie_{M_1}(u)} \cdot \overbrace{\ldots \cdot prod(\beta_1(k)) \cdot fin(dt(\beta_1(k)))}^{[\![M_1]\!]_q(v)}$$

$$\underbrace{init(gch(\beta_2(1))) \cdot prod(\beta_2(1)) \cdot \ldots \cdot prod(\beta_2(|u|))}_{sortie_{M_2}(u)} \cdot \underbrace{\ldots \cdot prod(\beta_2(k)) \cdot fin(dt(\beta_2(k)))}_{[\![M_2]\!]_p(v)}$$

Les états atteints après la transduction de u par M_1 et M_2 sont respectivement $q = dt(\beta_1(|u|))$ et $p = dt(\beta_2(|u|))$.

$[\![M_2]\!]_p$ et $[\![M_1]\!]_q$ sont donc définis sur le même domaine. L'hypothèse de départ et le lemme 11 nous donnent comme résultat que pour tout $v \in u^{-1}dom([\![M_1]\!])$:

$$[\![M_1]\!]_q(v) = sortie_{M_1}(u)^{-1} \cdot [\![M_1]\!](u \cdot v) = sortie_{M_2}(u)^{-1} \cdot [\![M_2]\!](u \cdot v) = [\![M_2]\!]_p(v)$$

□

Proposition 10. *Pour tout transducteur déterministe de mots M il existe un unique ed*TM *minimal $can(M)$ équivalent, modulo renommage des états.*

Preuve. L'existence de l'*ed*TM minimal découle de la propriété 6 assurant l'existence d'un *ed*TM et de l'application de l'algorithme de minimisation. Le fait qu'il soit unique provient de la propriété observée par le lemme 12, assurant que tout *ed*TM équivalent est composé d'états qui définissent les mêmes sous-transductions, et contient, après minimisation, le même nombre d'états que $can(M)$. Par conséquent, tout *ed*TM équivalent est identique à $can(M)$ modulo renommage de ces états. □

Cette proposition nous permet d'élaborer une nouvelle manière de prouver le corollaire 2. Pour décider de l'équivalence de deux transducteurs déterministes de mots, il faut tester l'égalité de leurs formes normales minimales respectives, ce qui peut être fait en temps polynomial.

De cette manière, nous avons un algorithme polynomial pour décider de l'équivalence de morphismes sur des ensemble réguliers.

2.1.2.7 Théorème de Myhill-Nérode

Nous cherchons maintenant à définir le *représentant canonique* non plus d'une transduction mais directement de la fonction sous-séquentielle τ. Le représentant $can(\tau)$ repose sur les notions de résiduels de transformation associés aux mots préfixes et les classes d'équivalence qui les lient. Ces classes correspondront directement aux états du *ed*TM minimal recherché. Dans cette partie, nous proposons une adaptation du théorème de Myhill-Nérode (Nerode (1958)) aux fonctions sous-séquentielles qui nous assure de l'existence d'un représentant unique, $can(\tau)$, à nombre d'états fini pour toute fonction sous-séquentielle. $can(\tau)$ correspond directement au transducteur minimal $can(M)$ pour tout *d*TM M reconnaissant cette transformation, et se base sur les définitions précédentes adaptées aux fonctions sous-séquentielles.

La notion de résiduel pour les transformations sous-séquentielles (notée $u^{-1}\tau$) se base sur la définition de résiduels pour un langage de mots (son domaine) auquel il faut associer une sortie. Pour les fonctions sous-séquentielles, la répartition de cette sortie se base sur les définitions de transducteurs travailleurs. Cela nous donne comme sortie associée à tout mot préfixe appartenant à $\mathit{prefixe}(\mathit{dom}(\tau))$:

$$\mathit{sortie}_\tau(u) = \mathit{lcp}(\{\tau(u \cdot v) \mid u \cdot v \in \mathit{dom}(\tau))$$

Si $u \notin \mathit{prefixe}(\mathit{dom}(\tau))$ la sortie associée à ce préfixe est indéfini.

Définition 24. *Le* résiduel *d'une fonction sous-séquentielle τ pour un mot $u \in \mathit{prefixe}(\mathit{dom}(\tau))$ est une fonction (partielle) de Σ^* à Δ^* correspondant aux continuations possibles du mot u et leurs productions respectives. Plus formellement :*

$$u^{-1}\tau = \{(v, \mathit{sortie}_s(u)^{-1}\tau(u \cdot v)) \mid u \cdot v \in \mathit{dom}(\tau)\}$$

Ces résiduels correspondent à la sémantique associée aux états d'un transducteur travailleur reconnaissant cette transformation. Pour obtenir le transducteur travailleur minimal, il faut définir des relations d'équivalence entre ces résiduels et les préfixes les représentant.

Définition 25. *Pour une fonction sous-séquentielle τ, une relation d'équivalence $\equiv_\tau$ lie deux mots u et v appartenant à $\mathit{prefixe}(\mathit{dom}(\tau))$ ssi les résiduels respectifs sont égaux. En d'autres termes,*

$$u \equiv_\tau v \text{ si et seulement si } u^{-1}\tau = v^{-1}\tau$$

On nomme index de Myhill-Nerode *la cardinalité de l'ensemble des classes d'équivalences de τ.*

On représentera chacune de ces classes d'équivalences par le plus petit mot qu'elle contient. À chaque préfixe $u \in \mathit{prefixe}(\mathit{dom}(\tau))$ on associe le représentant de la classe d'équivalence à laquelle il appartient, $[u]_\tau = \mathit{min}_{<_{ord}}(\{v \mid v \equiv_\tau u\})$ pour tout $u \in \mathit{prefixe}(\tau)$. L'ensemble de ces mots compose les plus courts représentants des classes d'équivalence, et des états de $\mathit{can}(\tau)$:

$$\mathit{minPref}(\tau) = \{[u]_\tau \mid u \in \mathit{dom}(\tau)\}$$

Une fois les états du transducteur minimal identifiés, il ne reste qu'a identifier les sorties à produire sur chacune des règles.

Définition 26. *On appelle* facteur, *noté fact_M d'un mot $u \cdot a$ pour une transformation τ la sortie ne pouvant être produite qu'après lecture de sa dernière lettre. Pour $\mathit{fact}_\tau(u \cdot a) = \mathit{sortie}_\tau(u)^{-1}\mathit{sortie}_\tau(u \cdot a)$*

En nous basant sur ces notions, nous pouvons à partir d'une fonction sous-séquentielle τ définir le transducteur travailleur minimal.

Définition 27. *Le transducteur travailleur minimal $can(\tau)$ est défini par le n-uplet $(\Sigma, \Delta, Q, init, fin, rul)$ tel que :*

- $Q = minPref(\tau)$
- $init = \{(\varepsilon, sortie_\tau(\varepsilon)\}$
- $fin = \{u, sortie_\tau(u)^{-1}\tau(u)) \mid u \in minPref(\tau) \cap dom(\tau)\}$
- $rul = \{u \xrightarrow{a/fact_\tau(u\cdot a)} [u\cdot a]_\tau \mid u \in minPref(\tau),\ u\cdot a \in prefixe(\tau)\}$

Théorème 4. *(Myhill-Nerode) Pour toute transformation τ les trois propositions suivantes sont équivalentes :*

1. *τ est une fonction sous-séquentielle à index de Myhill-Nérode fini*
2. *$can(\tau)$ est l'unique ed*TM *ayant le minimum d'états reconnaissant τ*
3. *τ est reconnaissable par un ed*TM

Preuve.

- (2) ⇒ (3) se vérifie puisque $can(\tau)$ est un *ed*TM particulier reconnaissant τ.
- (3) ⇒ (1) s'obtient par définition des fonctions sous-séquentielles.
- Pour que (1) ⇒ (2) il faut prouver que le transducteur $can(\tau)$ (noté M dans cette preuve) est un *ed*TM reconnaissant la transformation τ ; En d'autres termes, que $[\![M]\!]_{[u]_\tau} = u^{-1}\tau$. Plus précisément, il faut vérifier qu'ils partagent un même domaine, $dom([\![M]\!]_{[u]_\tau}) = \{v \mid u\cdot v \in dom(\tau)\}$, et une même sortie produite le plus tôt possible sur M, $[\![M]\!]_{[u]_\tau} = sortie_\tau(u)^{-1}\tau(u\cdot v)$ pour tout $v \in dom([\![M]\!]_{[u]_\tau})$. La preuve se fait par induction sur la taille des suffixes v.
  - Si $v = \varepsilon$:
    Par construction, on peut remarquer que $\varepsilon \in dom([\![M]\!]_{[u]_\tau})$ ssi $u \in dom(fin)$ et donc $u \in dom(\tau)$. De plus, la sortie produite $[\![M]\!]_{[u]_\tau}(\varepsilon) = fin([u]_\tau) = sortie_\tau(u)^{-1}\tau(u)$.
  - Sinon, $v = a\cdot v'$ tel que $a \in \Sigma$:
    Pour ce qui est du domaine, on peut remarquer que $a\cdot v' \in (dom([\![M]\!]_{[u]_\tau}))$ ssi $v' \in dom([\![M]\!]_{[u\cdot a]_\tau})$ ce qui par l'hypothèse d'induction équivaut à $u\cdot a\cdot v' \in dom(\tau)$. La production $[\![M]\!]_{[u]_\tau}(a\cdot v')$ est donc égale à :

$$
\begin{aligned}
& w\cdot [\![M]\!]_{[u\cdot a]_\tau}(v') \text{ pour } [u]_\tau \xrightarrow{a/w} [u\cdot a]_\tau \\
&= sortie_\tau(u)^{-1}\cdot sortie_\tau(u\cdot a)\cdot [\![M]\!]_{[u\cdot a]_\tau}(v') && \text{def. } rul \\
&= sortie_\tau(u)^{-1}\cdot sortie_\tau(u\cdot a)\cdot sortie_\tau(u\cdot a)^{-1}\cdot \tau(u\cdot a\cdot v') && \text{induction} \\
&= sortie_\tau(u)^{-1}\cdot \tau(u\cdot a\cdot v')
\end{aligned}
$$

Pour vérifier que $[\![M]\!] = \tau$, il suffit d'observer que $dom([\![M]\!]_{[\varepsilon]_\tau}) = \{u \mid \varepsilon \cdot u \in dom(\tau)\} = dom(\tau)$ et que pour tout $u \in dom(\tau)$ nous avons :

$$\begin{aligned} [\![M]\!](u) &= sortie_\tau(\varepsilon) \cdot [\![M]\!]_{[\varepsilon]_\tau}(u) \\ &= sortie_\tau(\varepsilon) \cdot sortie_\tau(\varepsilon)^{-1} \cdot \tau(u) \\ &= \tau(u) \end{aligned}$$

M est un *ed*TM puisque pour tout état $[u]_\tau$:

$$\begin{aligned} lcp([\![M]\!]_{[u]_\tau}) &= lcp(\{sortie_\tau(u)^{-1} \cdot \tau(u \cdot v) \mid u \cdot v \in dom(\tau)\}) \\ &= sortie_\tau(u)^{-1} \cdot lcp(\{\tau(u \cdot v) \mid u \cdot v \in dom(\tau)\}) \\ &= \varepsilon \end{aligned}$$

M est un *ed*TM reconnaissant τ mais il reste à vérifier qu'il est l'unique minimal. Le lemme 12 nous dit que tout autre *ed*TM reconnaissant τ est composé d'états définissant les mêmes sous-transductions et donc même résiduels. Puisque $Q = minPref(\tau)$, on sait que deux états n'acceptent pas une même sous-transduction et que leur nombre ne peut être baissé. Par conséquent $can(\tau)$ est l'unique *ed*TM minimal représentant τ.

□

2.1.2.8 Apprentissage

Maintenant, nous allons nous intéresser au problème d'apprentissage des fonctions sous-séquentielles.

La technique utilisée ici repose sur les techniques d'inférence grammaticale introduites par Gold (1967) permettant d'apprendre un langage à partir d'exemples de mots de ce langage. Adapté par Oncina *et al.* (1993), l'algorithme OSTIA repose sur la création d'un transducteur représentant un ensemble de couples d'entrées et de sorties, suivi de sa minimisation par fusion d'états. La limite de ces fusions vient de l'impossibilité de synchroniser la sortie, générant une non fonctionnalité. Dans cet algorithme, aucune information n'est donnée sur le domaine de transformation.

Divers moyens permettent d'intégrer de l'information sur ce domaine, comme des paires d'exemples négatifs, des mots ou préfixes du domaine d'entrée non acceptés. Ce domaine pouvant être appris à part, en utilisant les algorithmes usuels d'apprentissage de langage, nous supposerons par la suite que le domaine nous est donné sous forme d'un DFA.

L'algorithme que nous exposons ensuite est basé sur l'algorithme RPNI proposé par Oncina et Garcia (1992). On étend cet algorithme permettant d'apprendre des langages réguliers de mots aux transducteurs. Le processus se

divise en deux étapes principales, l'identification des différentes classes d'équivalence entre les préfixes, les parcourant dans l'ordre $<_{ord}$, puis la création du transducteur travailleur minimal correspondant. Nous conclurons sur un théorème d'apprentissage, assurant que toute fonction sous-séquentielle peut être apprise en temps et taille polynomial dans la taille de l'échantillon.

Le résultat de l'évaluation de cette fonction d'apprentissage dépend de l'échantillon considéré.

Définition 28. *Un* échantillon S *est composé d'un ensemble de couples associant à un mot de l'entrée une sortie possible, i.e.* $S \subseteq \Sigma^* \times \Delta^*$. *Il existe au plus un couple ayant un même mot d'entrée.*

S peut être considéré comme une fonction partielle et sera parfois utilisé en temps que tel par la suite. Le domaine d'un échantillon S correspond à l'ensemble des parties gauches des couples qui le compose. Réciproquement, son image correspond à sa partie droite.

$$dom(S) = \{u \mid (u, v) \in S\}$$
$$im(S) = \{v \mid (u, v) \in S\}$$

Définition 29. *Un échantillon* S *est dit* compatible *avec une fonction sous-séquentielle* τ *si* $S \subseteq \tau$.

Dans un apprentissage, prenant en compte un ensemble d'exemples et non pas une transformation, il faut se montrer plus souple sur la notion d'équivalence entre les mots et leur résiduels. Il faut uniquement veiller à ce que la fonctionnalité ne soit pas remise en cause. Afin d'éviter une sur-généralisation, on utilisera les informations connues sur le domaine, données par un DFA.

Pour cela on introduit la notion de compatibilité entre deux mots par rapport à un échantillon.

Définition 30. *Pour un échantillon* S *et un* DFA A *représentant le domaine d'apprentissage, et deux mots* u_1, u_2 *de prefixe*$(dom(S))$, u_1 *est dit* compatible *avec* u_2 *(noté* $u_1 \approx_{(S,A)} u_2$*) si et seulement si :*

1. $\forall v \in u_1^{-1} dom(S) \cap u_2^{-1} dom(S),\ (u_1^{-1}S)(v) = (u_2^{-1}S)(v)$
2. $u_2^{-1} dom(S) \subseteq u_1^{-1} dom(\mathcal{L}(A))$ *et* $u_1^{-1} dom(S) \subseteq u_2^{-1} dom(\mathcal{L}(A))$

Pour assurer l'apprentissage d'une transformation τ, il est nécessaire d'introduire la notion d'échantillon caractéristique. Le but étant, en présence d'un échantillon caractéristique de τ ou tout ensemble le contenant et compatible avec τ, que le résultat soit le transducteur canonique $can(\tau)$ reconnaissant cette transformation.

Cet échantillon caractéristique est surtout défini par des contraintes permettant d'assurer la reconstruction de $can(\tau)$ durant l'apprentissage. Tous les états doivent pouvoir être identifiés et dissociés. L'algorithme parcourant les préfixes dans l'ordre $<_{ord}$, chaque classe doit être représentée par le plus petit mot qu'elle contient, appartenant à $minPref(\tau)$. Chaque règle liant deux états doit aussi y être représentée. L'ensemble des préfixes devant être présents dans l'échantillon, dénommé $noyau(\tau)$, comprend donc les représentants de chaque état ainsi que ceux des règles :

$$noyau(\tau) = \{u \cdot a \mid a \in \Sigma \cup \{\varepsilon\},\ u \in minPref(\tau), u \cdot a \in prefixe(\tau)\}$$

Il faut assurer le fait de pouvoir différencier deux classes d'équivalences, ce qui peut se détecter par un conflit dans le domaine où par la perte de fonctionnalité. Il ne reste plus qu'à assurer la bonne distribution de la sortie sur le transducteur, ce qui nécessite un préfixe commun vide pour chaque production d'un même résiduel d'un mot de $noyau(\tau)$. Les représentants d'un état doivent aussi permettre de savoir s'il est final ou non, et être présents dans le domaine de S si c'est le cas.

Définition 31. *L'échantillon S est caractéristique pour la transformation τ pour le domaine exprimé par un* DFA *A si il est :*

(C) compatible *:*
$S \subseteq \tau$ *et* $[\![A]\!] = dom(\tau)$

(S) structurellement complet *:*
$\forall u \in noyau(\tau),\ \exists u \cdot v \in dom(\tau).\ (u \cdot v, \tau(u \cdot v)) \in S.$

(E) conservateur d'équivalences *:*
$\forall u \in minPref(\tau),\ \forall v \in noyau(\tau).\ u \not\equiv_\tau v$ *implique qu'il existe des exemples tel que* $u \not\equiv_S v$.

(P) complet sur la sortie, *i.e.* $\forall u \in minPref(\tau)$ *:*
$\exists (u \cdot v_1, \tau(u \cdot v_1)), (u \cdot v_2, \tau(u \cdot v_2)) \in S$ *tel que* $lcp(\tau(u \cdot v_1), \tau(u \cdot v_2)) = \varepsilon$.

(F) finalisation. *Que chaque transition finale soit représentée i.e.* $\forall u \in minPref(\tau)$ *tel que* $(u, w) \in \tau$, *il existe* $(u, w) \in S$.

Proposition 11. *Tout échantillon compatible avec une transformation τ et contenant un échantillon caractéristique est caractéristique.*

Preuve. Les contraintes (S), (E) et (P), (F) sont monotones. La compatibilité (C) est une condition de la proposition. □

Lemme 13. *Pour toute fonction sous-séquentielle τ et son transducteur canonique $can(\tau)$, un échantillon caractéristique peut être généré en temps et taille polynomial dans la taille de $can(\tau)$.*

Preuve. Construisons les exemples de S à partir du transducteur canonique $can(\tau)$ (noté M).

Pour chaque état $q \in Q$, nous identifions par v_q le plus petit mot du domaine de $[\![M]\!]_q$. Nous identifions par la même occasion le représentant d'un état u_q, le plus petit mot permettant d'y accéder. Formellement cela nous donne :

$$u_q = min_{<_{ord}}\{u \mid q_0 \xrightarrow{u/w}_* q,\ (q_0, w_0) \in init\}$$
$$v_q = min_{<_{ord}}\{v \mid v \in dom([\![M]\!]_q)\}$$

Nous pouvons remarquer que chaque état correspondant à une classe d'équivalence de $[\![M]\!]$, M étant minimal, l'ensemble des plus petits préfixes *minPref* correspond à l'ensemble des u_q pour chaque état $q \in Q$. Cela nous donne pour M :

$$minPref([\![M]\!]) = \{u_q \mid q \in Q\}$$
$$noyau([\![M]\!]) = \{u_q \cdot a \mid q \in Q,\ a \in \{a \mid q \xrightarrow{a/w} q' \in rul\} \cup \{\varepsilon\}\}$$

ensemble de taille linéaire dans celle de M.

Pour assurer la propriété (S), il faut ajouter à S les exemples correspondants. Pour toute règle $r \in rul$,

$$S = S \cup \{(u, [\![M]\!](u)) \mid r = q \xrightarrow{a/w} q' \in rul,\ u = u_q \cdot a \cdot v_{q'}\}$$

soit $|rul|$ exemples.

(F) nécessite l'ajout d'un exemple pour chaque transition finale. Pour tout état $q \in Q$ tel que $fin(q) = w$,

$$S = S \cup \{(u_q, [\![M]\!](u_q))\}$$

La propriété (P) repose sur le fait que $lcp(im([\![M]\!]_q)) = \varepsilon$ revenant à l'existence :

Soit d'un mot u tel que $(u, \varepsilon) \in [\![M]\!]_q$ ce qui donne :

$$S = S \cup \{(u_q \cdot u, [\![M]\!](u_q \cdot u))\}$$

Soit de deux paires $(u, w),\ (v, w') \in [\![M]\!]_q$ telles que $lcp(w, w') = \varepsilon$, dans quel cas :

$$S = S \cup \{(u', [\![M]\!](u')), (v', [\![M]\!](v'))\} \text{ où } u' = u_q \cdot u \text{ et } v' = u_q \cdot v$$

Au plus $|Q| \times 2$ exemples seront donc nécessaires.

Pour préserver les classes d'équivalences, et donc les états, il est nécessaire d'avoir au plus deux exemples permettant de dissocier ces classes. Cela revient, pour chaque état q, à s'assurer que $u_q \not\equiv_{[\![M]\!]} u'$, pour tout mot $u' \in noyau([\![M]\!])$,

est vérifié si u' conduit à un état différent de q. Soit (u_q, u') une de ces paires, u atteignant l'état q', deux possibilités de montrer cette inégalité s'offre à nous :

Soit il existe un mot $v \in dom([\![M]\!]_q) \cup dom([\![M]\!]_{q'})$ tel que $(v, w) \in [\![M]\!]_q$, $(v, w') \in [\![M]\!]_{q'}$ et $w \neq w'$. Il suffit d'ajouter les exemples correspondant à S :

$$S = S \cup \{(v', [\![M]\!](v')), (v'', [\![M]\!](v''))\} \text{ où } v' = u_q \cdot v \text{ et } v'' = u' \cdot v$$

Soit il existe un mot $v \in \Sigma^*$ tel que $v \in dom([\![M]\!]_q)$ et $v \notin dom([\![M]\!]_{q'})$, ou l'inverse. Cela demande respectivement l'ajout de $(u_q \cdot v, [\![M]\!](u_q \cdot v))$ ou $(u' \cdot v, [\![M]\!](u' \cdot v))$ dans S.

On peut remarquer que la compatibilité (C) est assurée tout au long de la construction car chaque exemple positif appartient à $[\![M]\!] = \tau$. Les autres propriétés sont monotones, i.e. conservées pendant l'ensemble du processus. Nous pouvons donc, pour tout transducteur canonique M d'une transformation τ, définir un échantillon caractéristique dans une taille polynomiale dans celle de M.

□

Exemple 25. *Construisons les exemples S de l'échantillon caractéristique de $[\![M_{2.6(b)}]\!]$ qui transforme un mot de $(a+b)^*b$ dans sa copie en supprimant chaque "a". On remarque que $[\![M_{2.6(b)}]\!]$ est un dTM non travailleur. En tirant le premier b produit, on obtient cependant le représentant travailleur unique $M_{2.12}$ de cette transduction que nous allons apprendre.*

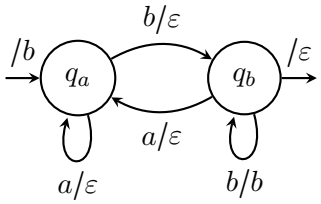


FIGURE 2.12 – $M_{2.6(b)}$ en forme normale minimale

Pour le moment $S = \varnothing$. Les plus petits mots transformables à partir de q_a et q_b sont respectivement $v_{q_a} = b$ et $v_{q_b} = \varepsilon$.

Les mots représentants les états, c'est à dire les plus petits mots permettant d'y accéder, sont $u_{q_a} = \varepsilon$ et $u_{q_b} = b$.

Pour assurer la propriété S, il faut ajouter à S des exemples de chaque transition. Pour q_a, il existe deux règles $q_a \xrightarrow{a/\varepsilon} q_a$ et $q_a \xrightarrow{b/\varepsilon} q_b$. On ajoute donc à S les exemples pour les mots $u_a = u_{q_a} \cdot a \cdot v_{q_a} = ab$ et $u_b = u_{q_a} \cdot b \cdot v_{q_b} = b$ à savoir (ab, b) et (b, b). De la même façon, on ajoute pour l'état q_b les couples (bb, bb) et (bab, bb) à S.

Pour l'instant, l'échantillon $S = \{(b,b),(ab,b),(bb,bb),(bab,bb)\}$. *Pour ce qui est de la propriété* (P), *puisque* $(b,\varepsilon) \in [\![M_{2.12}]\!]_{q_a}$, $(\varepsilon,\varepsilon) \in [\![M_{2.12}]\!]_{q_b}$ *et que* $u_{q_a} \cdot b = u_{q_b} \cdot \varepsilon = b$, *il suffirait d'ajouter* (b,b) *à* S, *ce qui est déjà fait.*

La distinction entre les états q_a *et* q_b *nécessite que* $u_{q_a} = \varepsilon \not\equiv_M b = u_{q_b}$. *Pour cela on peut générer un conflit sur la fonctionnalité en se basant sur le fait que que* $(ab,\varepsilon) \in [\![M_{2.12}]\!]_{q_a}$ *et* $(ab,b) \in [\![M_{2.12}]\!]_{q_b}$, *et en ajoutant les exemples* (ab,b) *et* (bab,bb) *correspondants. L'autre manière, utilisée ici, est de trouver un mot d'entrée dans le domaine d'un état et non présent dans celui de l'autre, pour notre cas* $\varepsilon \notin dom([\![M_{2.12}]\!]_{q_a})$ *et* $\varepsilon \in dom([\![M_{2.12}]\!]_{q_b})$. *La présence de* (b,b) *suffit donc, à l'aide de l'automate* A *représentant le domaine, à apporter ce conflit.*

Pour assurer que $b \not\approx_S ba$, *on choisit aussi* ε *comme continuation possible de* b *mais qui ne permet pas une transformation pour* ba. *De même pour montrer que* $\varepsilon \not\approx_S bb$.

L'unique finalisation possible à partir de l'état q_b *est déjà représentée par* (b,b).

L'échantillon $S = \{(b,b),(ab,b),(bb,bb),(bab,bb)\}$ *obtenu est bien caractéristique pour* $M_{2.12}$ *et produit bien ce même transducteur par l'algorithme d'apprentissage comme vu précédemment.*

Algorithme d'apprentissage

Si nous observons la construction de $can(\tau)$ à partir de τ dans la partie précédente, nous pouvons identifier plusieurs étapes de conception. Il faut d'abord définir l'ensemble des plus petits mots représentants les différentes classes d'équivalences, $minPref(\tau)$, qui composeront l'ensemble des états. Dans notre cas, il faut identifier l'ensemble de ces mots, qu'on note Q_{ok}, et les classes qu'ils représentent. Q_{ok} est composé dans un premier temps par ε, plus petit préfixe et état initial. Par la suite, cet ensemble est enrichi par des mots issus des Q_{border}, préfixes représentant toutes les transitions du transducteur cible pas encore considérés. Ils sont les prolongations des mots de Q_{ok}. Les mots de Q_{border} sont considérés l'un après l'autre suivant l'ordre $<_{ord}$. Si on connait τ, un mot $v \in Q_{border}$ appartient à une classe d'équivalence représentée par un mot $u \in Q_{ok}$, et est *fusionné* à l'état correspondant, uniquement s'il est équivalent à celui ci $(v \equiv_\tau u)$.

Dans notre cas, la classe d'équivalence se construit au fur et à mesure. On définit progressivement la fonction $[x]_S$ qui associe aux préfixes x le représentant de la classe d'équivalence à laquelle ils appartiennent, de manière à ce que :

$$[v]_S = u \text{ ssi } \forall u' \in prefixe(dom(S)) \text{ tel que } u' <_{ord} v.\ [u']_S = u \Leftrightarrow u' \not\approx_{(S,A)} v$$

La fusion d'états n'est donc possible entre un représentant $u \in Q_{ok}$ et un

candidat v que si le résiduel est compatible avec chaque élément de la classe d'équivalence de u.

Définition 32. *Pour un échantillon S et un* DFA *A, on définit la fonction $fusionnable(v,u)$ qui nous dit pour deux préfixes u et v si v peut être intégré par la classe d'équivalence représentée par u.*

$$fusionnable(u,v) = \begin{cases} vrai & Si\ \forall u' \in prefixe(dom(S))\ tel\ que\ u' <_{lex} u : \\ & \qquad [u']_S = v \Rightarrow u' \approx_{(S,A)} u \\ faux & sinon \end{cases}$$

Une fois ces classes d'équivalences identifiées, les états de $can(S)$ le sont aussi. Il reste à construire les règles de $can(S)$ en se basant sur l'échantillon S, et l'automate A représentant le domaine. La manière de faire est identique à celle de $can(\tau)$.

Plus formellement, l'algorithme d'apprentissage est le suivant :

RPNI$_d$TM

entrée : Un echantillon S, un DFA A
$Q_{border} := \{\varepsilon\}$
$Q_{ok} := \varnothing$
tant que $Q_{border} \neq \varnothing$ **do**
  $u := min_{<_{ord}}(Q_{border})$
  $Q_{border} := Q_{border} \setminus \{u\}$
  $OK = \{v \in Q_{ok} \mid fusionnable(u,v)\}$
  si $OK \neq \varnothing$
   alors $[u]_S := min_{<_{ord}}(OK)$
   sinon $[u]_S := u$
     $Q_{ok} := Q_{ok} \cup \{u\}$
  $Q_{border} := (Q_{border} \cup \{u \cdot a \mid u \cdot a \in prefixe(dom(S))\}) \setminus \{u\}$
pour tout $u \cdot a \in prefixe(dom(S))$ **en suivant l'ordre** $<_{ord}$
  si $\nexists [u]_S \xrightarrow{a/w} q \in rul$ **alors**
    $rul \leftarrow \{[u]_S \xrightarrow{a/fact_S(u\cdot a)} [u\cdot a]_S \mid u \cdot a \in prefixe(dom(S))\}$
$init = \{(\varepsilon, sortie_S(\varepsilon))\}$
$fin = \{(u,w) \mid \exists v \in prefixe(dom(S)), [v]_S = u, (\varepsilon, w) \in v^{-1}S\}$

Exemple 26. *Prenons l'ensemble d'exemples $S = \{(b,b),(ab,b),(bb,bb),(bab,bb)\}$ et le* DFA *A représentant le domaine de $M_{2.12}$.*

Au départ, $Q_{border} = \{\varepsilon\}$ et $Q_{ok} = \varnothing$. On débute avec le premier préfixe $u = \varepsilon$. Q_{ok} étant vide, l'ensemble OK correspondant aux états déjà identifiés

compatibles avec le mot courant est vide aussi. ε *est donc un nouvel état. Maintenant,* $Q_{ok} = \{\varepsilon\}$ *et* $[\varepsilon]_S = \varepsilon$*. Les prochains mots à considérer sont toutes les continuations possibles, à savoir* $Q_{border} = \{a, b\}$*.*

Passons maintenant à $u = a$*. Le préfixe* a *est fusionnable avec* ε *puisque* $\varepsilon \approx_{(S,A)} a$*. En effet, l'union des résiduels* $a^{-1}S$ *et* $\varepsilon^{-1}S$ *produit un ensemble fonctionnel* $\{(b, \varepsilon), (ab, \varepsilon), (bb, b), (bab, b)\}$*. Au niveau des domaines, le résiduel* $a^{-1}S = \{(b, \varepsilon)\}$ *appartient bien à* $\varepsilon^{-1}\mathcal{L}(A)$ *et il en va de même pour* $\varepsilon^{-1}S$ *et* $a^{-1}\mathcal{L}(A)$*.* Q_{ok} *n'est pas modifié et la fonction des représentants est enrichie par* $[a]_S = \varepsilon$*.* Q_{border} *est uniquement privé de* a *puisque aucun mot préfixé par* a *peut être identifié comme représentant d'une classe d'équivalence.*

Le mot suivant dans Q_{border} *est* $u = b$*. Le fait que* $(ab, \varepsilon) \in \varepsilon^{-1}S$ *et* $(ab, b) \in b^{-1}S$ *et donc que* $\varepsilon \not\approx_S b$*, nous donne que* $[b]_S = b$ *est l'identifiant d'un nouvel état* $b \in Q_{ok}$*. La non compatibilité peut aussi être obtenue du fait que* $\varepsilon \in b^{-1}dom(S)$ *alors que* $\varepsilon \notin a^{-1}\mathcal{L}(A)$*. Maintenant,* $Q_{border} = \{ba, bb\}$*.*

De la même manière on vérifie que $ba \approx_{(S,A)} \varepsilon$*, et* $bb \approx_{(S,A)} b$ *alors que* $bb \not\approx_S \varepsilon$*. Ce qui nous donne respectivement que* $[ba]_S = \varepsilon$*,* $[bb]_S = b$*. Le dernier préfixe à considerer,* $Q_{border} = \{bab\}$ *vérifie que* $bab \approx_{(S,A)} b$ *alors que* $bab \not\approx_S \varepsilon$ *donc* $[bab]_S = b$*.*

Pour ce qui est de la règle initiale, par construction nous obtenons :

$$init = \{(\varepsilon, sortie_S(\varepsilon))\} = \{(\varepsilon, b)\}$$

b *est l'unique état final. En utilisant le mot* b *on obtient que :* $fin = \{(b, \varepsilon)\}$
Les règles obtenues par la dernière partie de l'algorithme sont les suivantes :

$$rul = \{\varepsilon \xrightarrow{a/\varepsilon} \varepsilon,\ \varepsilon \xrightarrow{b/\varepsilon} b,\ b \xrightarrow{a/b} \varepsilon,\ b \xrightarrow{b/b} b\}$$

Le transducteur ainsi obtenu est bien le transducteur $M_{2.6(b)}$ *souhaité modulo le renommage des états* ε *en* q_a *et* b *en* q_b*.*

Il faut maintenant vérifier, pour tout échantillon caractéristique, que l'algorithme produit bien le *ed*TM minimal représentant la transformation cible, de taille polynomiale dans celle de l'échantillon, et en temps polynomial dans cette même taille.

Lemme 14. *Si* S *est l'échantillon caractéristique de* τ *sur le domaine exprimé par* A*, représenté par le transducteur minimal travailleur* $can(\tau)$*, alors le résultat de l'apprentissage* $\text{RPNI}_d\text{TM}(S, A)$ *est un transducteur travailleur isomorphe à* $can(\tau)$*.*

Preuve. On s'intéresse à l'apprentissage de $\min(\tau)$ (noté M) à partir de l'échantillon S caractéristique pour τ et A. On identifie une étape de l'apprentissage sur S par le mot u_0 considéré à cet instant. Dans cette preuve,

nous allons définir les propriétés invariantes à chaque étape de l'algorithme d'apprentissage permettant de vérifier que le transducteur attendu est bien isomorphe à M.

1. $Q_{border} = \{u \cdot a \mid [u]_\tau \xrightarrow{a/w} [u \cdot a]_\tau \in rul_M, u <_{ord} u_0, u \cdot a > u_0\}$.
   Cette propriété s'obtient par construction dans l'algorithme d'apprentissage et du fait que S est compatible à τ, donc tout préfixe de S est représenté par une exécution partielle de M.
2. $Q_{ok} = \{u <_{ord} u_0 \mid u \in Q_M\}$.
   (C) nous apporte que S est compatible avec τ et ne peut donc pas identifier plus d'états que ceux de M. De plus, grâce à (E) nous savons que u et v seront fusionnables seulement si ils sont compatibles. Associé au fait que les préfixes sont parcourus dans l'ordre $<_{ord}$ (1), le premier identifiant d'une classe d'équivalence est bien son plus petit représentant.
3. $\forall u <_{ord} u_0$, $u \xrightarrow{a/w} [u \cdot a]_\tau \in rul_M \Rightarrow u \in Q_{ok}$ et il existe $u' \in \Sigma^*$ et $w' \in \Delta^*$ tel que $(u \cdot a \cdot u', w') \in S$ et $w = sortie_S(u)^{-1} sortie_S(u \cdot a)$.
   L'existence du couple dans l'échantillon découle de (S). Le fait que $w = sortie_S(u)^{-1} sortie_S(u \cdot a)$ nous vient de (C) assurant la compatibilité et (P) assurant un bon partitionnement de la sortie.
4. $\forall u <_{ord} u_0$, $u \xrightarrow{a/w} [u \cdot a]_S \in rul_M \Leftrightarrow$ Pour tout $u \in Q_{ok}$, $u' \in \Sigma^*$ et $w' \in \Delta^*$ tel que $(u \cdot a \cdot u', w') \in S$ et $w = sortie_S(u)^{-1} sortie_S(u \cdot a)$.
   ($\Leftarrow$) S'obtient par construction dans l'algorithme d'apprentissage. En effet, les premières règles créées pour une classe d'équivalence identifiée par u se font pour $u \cdot a \in prefixe(dom(S))$ avec $a \in \Sigma$.
   ($\Rightarrow$) S respectant la condition (S), toutes les règles y sont représentées. Ainsi, les règles pour un échantillon caractéristique sont générées uniquement pour les $u \cdot a$ avec $u \in minPref(S)$. Aucune autre règle ne sera donc créée pour cet état.
5. $fin_M : \{u \in Q_{ok} \mid (u, w) \in S\} \to \Delta^*$ tel que $fin(u) = sortie_S(u)^{-1} w = fin(M)$.
   (F) et (P) nous assurent qu'un représentant de chaque transition finale existe et qu'elle correspond à celles de M.

Nous avons donc la première phase de l'algorithme qui identifie correctement les états de M avec (2). Ensuite (3) et (4) nous assurent que le transducteur créé M aura les mêmes règles que M reliant les états des même classes d'équivalences. Les finalisations sont assurées par (5) et l'initialisation par la construction dans l'échantillon caractéristique. (P) assure que trop de sorties ne seront pas produites durant ces phases. □

On peut remarquer que tout apprentissage produit un *ed*TM M, valide au regard de son échantillon S et de son domaine représenté par A, i.e. $S \subseteq [\![M]\!]$ et $dom(M) \subseteq A$. Dans le pire des cas, M sera un arbre préfixe annoté des sorties communes aux mots du même préfixe. Les états finaux représentant un unique mot, la finalisation assure que chaque sortie pourra être produite.

Nous pouvons conclure sur le théorème d'apprentissage suivant :

Théorème 5 ((Oncina *et al.*, 1993))**.** *Les fonctions sous-séquentielles sont apprenables en temps et taille polynomiale dans le ed*TM *cible.*

Cette classe de transducteurs a donc de nombreuses propriétés intéressantes mais perd beaucoup d'expressivité par rapport aux transducteurs rationnels. Maintenant nous allons nous intéresser aux transducteurs avec anticipations qui assurent une fonctionnalité tout en s'approchant le plus possible de l'expressivité des TM.

2.1.3 Transducteurs déterministes avec anticipation

La perte d'expressivité des transducteurs déterministes vient de sa dépendance à l'ordre du parcours du mot d'entrée. Un transducteur rationnel comme $M_{2.1}$ ne peut pas être exprimé de manière déterministe car il faut attendre la fin du mot pour décider quoi faire de tout ce qui a été lu précédemment. Pour palier ce problème nous allons introduire ici un pré-traitement sous la forme d'un automate fini qui parcourt les mots de droite à gauche et les annote par des états qui permettront par la suite d'exécuter un *d*TM sur ces mots annotés, de gauche à droite. Ces machines, appelées *transducteurs déterministes avec anticipation*, ont été introduites par Elgot et Mezei (1965). Ils ont la même expressivité que les transducteurs de mots fonctionnels, i.e. une relation $[\![M]\!] \subseteq \Sigma^* \times \Delta^*$, exprimable par un TM M, autorisant, au plus, une sortie pour un mot de l'entrée.

2.1.3.1 Syntaxe

L'anticipation est représentée par un automate déterministe de mots A, appelé *automate d'anticipation*, sur l'alphabet Σ et l'ensemble d'états P parcourant les mots de droite à gauche. Elle sort, pour chaque lettre d'un mot, cette même lettre associée à l'état atteint par l'automate. Une règle de l'automate $q \xrightarrow{a} q'$ peut être considérée comme une règle de transduction $q \xleftarrow{a/(a,q')} q'$. De plus, A doit être complet.

Un transducteur déterministe avec anticipation *dTl* est donc une paire $N = (A, M)$ où A est un $d\text{TA}^{\uparrow}$ ayant pour ensemble d'états P, et M un *d*TM défini sur l'alphabet d'entrée $\Sigma \times P$.

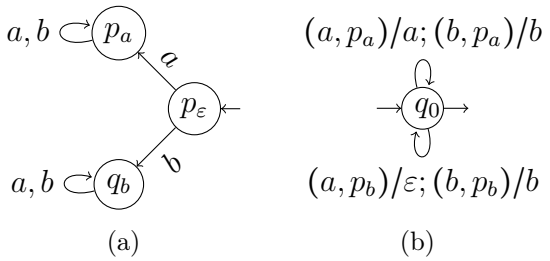


FIGURE 2.13 – transducteur avec anticipation N_1 pour $M_{2.1}$

2.1.3.2 Sémantique

La sémantique de l'automate d'anticipation A est donc une fonction rationnelle $[\![A]\!]$ qui produit pour un mot $u \in \Sigma^*$ le même mot annoté des états de A, $[\![A]\!](u) \in (\Sigma \times P)^*$. La fonction $[\![A]\!]$ est totale puisque A doit couvrir l'ensemble des mots de Σ^*.

Un *dTl* $N = (A, M)$ défini donc une fonction rationnelle $[\![N]\!] = [\![M]\!] \circ [\![A]\!]$. Un mot u de l'entrée est d'abord annoté par les états de droite à gauche puis transformé par M de gauche à droite.

Exemple 27. *Reconsidérons $M_{2.1}$ de la figure 2.1. Il peut être exprimé par le dTl N_1 de la figure 2.13. En effet l'automate (a) permet d'annoter chaque lettre d'un mot $u \in \Sigma^*$ par un état p_a ou p_b indiquant la dernière lettre lue. Le transducteur (b) n'a plus qu'a tenir compte de cet état pour décider si la transduction à appliquer correspond à un mot se terminant par "a" ou "b".*

Pour illustrer le procédé, utilisons le mot "aabab". L'automate d'anticipation commence en p_ε. La dernière lettre étant un "b" il l'annote par p_b. Puis chaque autre lettre lue reste dans le même état ce qui nous donne :

$$(a, p_b)(a, p_b)(b, p_b)(a, p_b)(b, p_b)$$

En appliquant ensuite le transducteur, chaque symbole (a, p_b) est supprimé et chaque symbole (b, p_b) est remplacé par "b" ce qui nous donne "bb".

2.1.3.3 Expressivité

Comme dit précédemment, l'intérêt principal de ce formalisme est qu'il définit un procédé déterministe permettant de définir l'ensemble des fonctions rationnelles. Nous devons ce résultat à Elgot et Mezei (1965) à travers leur théorème de décomposition suivant.

Théorème 6. *(Elgot et Mezei) Une fonction partielle est rationnelle si et seulement si elle est définie par un dTl.*

Le modèle, malgré son déterminisme, reste cependant difficile à appréhender. Par exemple, la définition de domaine ou encore d'image dépend à la fois des deux machines et n'est pas obtenu aussi facilement que pour un dTM. Cela remet en cause les manières de répondre aux principaux problèmes qui nous intéressent, comme le test de l'équivalence ou encore l'apprentissage.

2.1.3.4 Équivalence

Comme nous avons pu le voir précédemment, l'exécution d'un dTl N se compose de deux étapes : l'annotation du mot de droite à gauche par un automate d'anticipation A, puis la transduction appliquée par un transducteur déterministe M.

Pour obtenir les deux exécutions sur une même représentation nous introduisons l'ensemble des règles suivantes :

$$rul^{ant} = \{(q,p) \xrightarrow{a/w} (q',p') \mid q \xrightarrow{(a,p)/w} q' \in rul_M,\ p' \xrightarrow{a} p \in rul_A\}$$

Á l'instar des dTM, nous définissons, pour un mot u, une exécution $u * \beta$ où β associe à chaque position de u une règle de rul^{ant}. Une exécution $u * \beta$ est dite *valide* si la règle à la position $i \in pos(u)$ consomme la bonne entrée $u(i) = lect(\beta(i))$, et si elle est la continuation de la règle qui la précède (i.e. pour $i-1 \in pos(u)$, $gch(\beta(i-1)) = dt(\beta(i))$). Suivant la même logique, une exécution $u * \beta$ est dite *complète* si elle est valide, que l'état à l'extrémité gauche soit composé d'un état initial pour M et final pour A, et que celui à l'extrémité droite soit composé d'un état final pour M et initial pour A. Plus formellement, $gch(\beta(1)) = (q,p)$ implique que $q \in dom(init_M)$, et $dt(\beta(|u|)) = (q',p')$ implique que $q' \in dom(fin_M)$ ainsi que $p' \in init(A)$.

Les deux étapes étant déterministes, il n'existe qu'une exécution de A sur un mot u et une exécution de M sur le mot obtenu par l'anticipation. Il n'existe donc qu'une unique exécution complète pour u sur N. La notion d'exécution *parallèle complète* sur deux dTl, N_1 et N_2, correspond à la définition pour les dTM, à savoir un triplet $u * \beta_1 * \beta_2$ pour $u \in dom(N_1) \cap dom(N_2)$ où β_i est une exécution complète de N_i.

Lemme 15. *Les deux problèmes de décisions suivants sont équivalents modulo des réductions en temps polynomial :*

1. *L'équivalence de deux morphismes sur un langage régulier donné par un* DFA *peut être décidé en temps polynomial.*
2. *L'équivalence de deux transducteurs avec anticipation peut être décidé en temps polynomial.*

Preuve. « 1 ⇒ 2 ». Une transduction exprimée par un transducteur déterministe de mots M peut être exprimée par un transducteur avec anticipation $N = (M, A)$, où A est un automate à un état reconnaissant Σ^* et n'apportant aucune information lors de l'annotation du transducteur. Le problème d'équivalence de transducteurs déterministes se réduit dans celui de transducteurs avec anticipation. Nous obtenons le résultat souhaité en utilisant le lemme 10.

« 2 ⇒ 1 ». Prenons deux dTl $N = (M, A)$ et $N' = (M', A')$. Avant de suivre la même approche que pour le lemme 10, en définissant un DFA représentant les exécutions parallèles des transducteurs et les morphismes correspondants, nous devons tester que les deux transducteurs sont définis sur un même domaine. Pour cela, il faut exprimer les deux transducteurs sur un même automate d'anticipation obtenu par le produit $A'' = A \times A'$. Le langage obtenu étant *local*, i.e. le langage reconnu peut être représenté par un ensemble des parties constituant un mot. Son inverse l'est aussi et est donc reconnaissable par un automate déterministe ayant des états définis dans $(Q_A \times Q'_A) \cup \{p_i\}$, où p_i est l'état initial. Une fois cet automate obtenu, il faut calculer le produit de A'' et des automates obtenus par projection sur l'entrée de M et M', qui reconnaitront tous deux des mots sur $\Sigma \times Q_A \times Q'_A$ sous la forme d'automates déterministes. Il suffit maintenant de tester l'équivalence de ces deux automates.

Si les deux transducteurs sont définis sur le même domaine, on procède comme dans la preuve du lemme 10 en représentant toutes les exécutions parallèles par un DFA A''. Chaque règle de l'exécution parallèle est obtenue par l'utilisation des rul^{ant} associés à N et N'. Etant annotés directement par les règles des transducteurs, nous pouvons, à partir de ces règles, définir deux morphismes récupérant les productions respectives des transducteurs. Si ces deux morphismes sont équivalents sur A'' alors les deux transducteurs avec anticipation le sont aussi.

□

2.1.3.5 Apprentissage

La définition d'une forme normale, représentant une transformation exprimable par un dTl, n'est pas aussi simple que pour une fonction sous-séquentielle. La notion de minimalisation n'a pas de sens direct sur les dTl qui sont composés de deux machines. Une forme normale proposée par Reutenauer et Schützenberger (1991) pour les bi-machines, modèle pouvant s'adapter aux transducteurs avec anticipation, revient à minimaliser l'automate d'anticipation, seule partie pouvant être minimalisée indépendamment. Une fois l'automate minimalisé, il reste à normaliser le transducteur correspondant. Le problème est qu'une telle minimalisation peut entrainer une explosion de la

taille du transducteur associé.

Exemple 28. *Pour illustrer cette explosion, prenons la transformation définie sur* Σ^* *pour* $\Sigma = \{a, b\}$ *qui sort* 1 *si la nième lettre à partir de la fin du mot est un "a", sinon* 0*. Un automate d'anticipation permettant de remonter l'information nécessaire à une transduction ultérieure d'obtenir le même résultat, n'a besoin que de* $n + 2$ *états permettant de compter le nombre de caractères rencontrés et rester en* p_{n+1} *pour toute lettre supplémentaire. Le transducteur associé se limite à 2 états finaux* q_0 *et* q_1 *sortant respectivement* 0 *et* 1 *lors de la finalisation . Le passage de* q_0 *à* q_1 *est possible uniquement en lisant* (a, p_n) *signifiant que la nième lettre est bien un "a".*

L'automate d'anticipation minimal est complet. Il annote tout mot de Σ^**, et est composé d'un unique état laissant le travail au transducteur. Dans ce cas, le transducteur doit mémoriser les n dernières lettres qu'il a lues et donc avoir autant d'états que de mots possibles de taille 1 à n.*

Une autre solution serait de minimaliser le transducteur mais la notion de transducteur minimal n'est pas claire : sa définition dépend fortement de l'automate d'anticipation. De plus, le même problème d'explosion de taille de l'automate, en réduisant celle du transducteur, peut être facilement obtenu comme le montre l'exemple précédent.

La forme normale proposée par Boiret *et al.* (2012), appelée forme normale diagonale, se définit directement par son algorithme d'apprentissage. Cet algorithme se base sur la minimalisation de deux bornes l et m, l bornant le nombre d'états de l'automate d'anticipation et m étant en lien avec la taille du transducteur associé. Le but est de minimiser à la fois la taille de l'automate d'anticipation et celle du transducteur en jouant sur ces bornes. Le transducteur obtenu est forcément de taille inférieure à celui de la forme normale de Reutenauer et Schützenberger (1991), l'automate d'anticipation quant à lui ne pouvant être que de taille supérieure ou égale.

Dans le pire des cas, cette recherche se termine et nous ramène à la forme normale de Reutenauer et Schützenberger (1991).

Théorème 7 ((Boiret *et al.*, 2012))**.** *Les fonctions rationnelles sont apprenables en temps et taille polynomiaux dans la forme normale diagonale ciblée.*

2.2 Transducteur d'arbres d'arité bornée

Nous allons maintenant nous intéresser à la transformation d'arbres. Comme nous avons pu le voir précédemment il existe deux types d'arbres, d'arité bornée ou non. Dans un premier temps nous allons nous concentrer sur les

arbres d'arités bornés facilitant la manipulation et représentation par des automates ou transducteurs. Dans les premiers transducteurs présentés, la structure d'arbre est gardée. Le fait de garder cette même structure permettra des libertés de restructuration telles que l'inversion de noeuds ou encore leur duplication, ce que nous identifierons comme difficile dans des modèles de transductions d'arbres en mots. Par contre la concaténation naturelle dans les mots ne trouve pas d'équivalent efficace dans les arbres. Nous verrons ici trois principaux modèles identifiés par l'ordre de parcours de l'arbre d'entrée, de manière descendante, ascendante et descendante permettant une pré-annotation de l'arbre de manière ascendante. Nous terminerons sur la famille des macro-transducteurs descendants, plus expressive et permettant de lier le domaine des transducteurs aux modèles logiques de transformation.

2.2.1 Transducteur descendants

2.2.1.1 Définitions

Nous fixons une séquence infinie de variables d'entrée distinctes $(x_i)_{i\in\mathbb{N}}$ et, pour tout $k \geq 0$, nous définissons l'ensemble $X_k = \{x_1, \ldots, x_k\}$ et plus particulièrement $X_0 = \varnothing$.

Un transducteur d'arbres descendant représente la transformation d'arbres de $\mathcal{T}_\Sigma$ en arbres de $\mathcal{T}_\Delta$. Pour ce faire, il parcourt l'arbre d'entrée de sa racine à ses feuilles sans devoir pour autant considérer chaque sous arbre qu'il rencontre. À chaque étape, il associe un contexte représentant la sortie à produire ainsi que la manière de traiter les sous arbres considérés.

Définition 33. *Un transducteur descendant d'arbres* TOP *est défini par un n-uplet* $M = (\Sigma, \Delta, Q, init, rul)$. *Il est composé des alphabets bornés* Σ *et* Δ, *des symboles des arbres d'entrée et de sortie, d'un ensemble fini d'états* Q, *d'un ensemble d'axiomes initiaux* $init \subseteq \mathcal{T}_\Delta(Q \times \{x_0\})$. *Les règles sont représentés par une relation :*

$$rul \subseteq \bigcup_{k\geq 0} (Q \times \Sigma^{(k)}) \times \mathcal{T}_\Delta(Q \times X_k)$$

associant à un état et à un symbole f, *d'arité* k, *un arbre dont certaines feuilles sont annotées par un état et une variable d'entrée nous indiquant quel(s) sous-arbres sont autorisé ici.*

Chaque état d'un transducteur M défini une relation entre arbres d'entrée et sortie qui correspond à la plus petite relation $[\![M]\!]_q \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Delta$ telle que pour tout état q :

$$\begin{aligned}[\![M]\!]_q \quad = \quad & \{(t, s[\langle q', x_i\rangle \leftarrow s' \mid 1 \leq i \leq n,\ q' \in Q,\ (t_i, s') \in [\![M]\!]_{q'}]) \mid \\ & t = f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma,\ ((q, f), s) \in rul\}\end{aligned}$$

Similairement, la relation $[\![M]\!]$ peut être définie de la manière suivante en démarrant avec un des axiomes :

$$[\![M]\!] \quad = \quad \{(t, s[\langle q, x_0\rangle \leftarrow s' \mid q \in Q,\ (t, s') \in [\![M]\!]_q]) \mid s \in \mathit{init}\}$$

Un TOP est *total* si *init* n'est pas vide et s'il existe une règle *rul* pour chaque paire d'états de Q et de symboles de Σ.

Nous définissons le déterminisme pour les TOP*s* comme le changement de configuration de cette machine et la production de sortie qui peuvent toujours se faire sans aucun choix.

Définition 34. *Un transducteur descendant d'arbres $M = (\Sigma, \Delta, Q, \mathit{init}, \mathit{rul})$ est appelé* déterministe ou un *d*TOP*, si rul est une fonction partielle associant à chaque paire d'états $q \in Q$ et de symbole $f \in \Sigma^{(k)}$ au plus un arbre dans $\mathcal{T}_\Delta(Q \times X_k)$ et si init n'autorise au plus qu'un axiome initial.*

On peut vérifier que chaque dTOP déterministe définit une fonction partielle $[\![M]\!]$, et que pour chacun de ses états q, $[\![M]\!]_q$ est une fonction partielle.

Exemple 29. *Prenons le transducteur M_{29} représentant la transformation de tout arbre défini sur $\Sigma = \{f^{(2)}, a^{(0)}\}$ vers des arbres sur $\Delta = \{g^{(2)}, b^{(1)}, c^{(1)}, d^{(0)}\}$. Il débute avec l'axiome $\langle q, x\rangle$ et utilise les règles suivantes :*

$$\begin{array}{llll} q(f(x_1,x_2)) & \to & g(\langle q_b, x_1\rangle, \langle q_c, x_1\rangle) & q(a) \to d \\ q_b(f(x_1,x_2)) & \to & b(\langle q_b, x_1\rangle) & q_b(a) \to d \\ q_c(f(x_1,x_2)) & \to & c(\langle q_c, x_1\rangle) & q_c(a) \to d \end{array}$$

Si l'arbre d'entrée n'est composé que du symbole "a" il le renomme en "d". Sinon, il doit être de la forme $f(t_1, t_2)$ et le transducteur le transforme en un arbre ayant pour racine le symbole "g" ayant deux fils obtenus respectivement par les transformations $[\![M_{29}]\!]_{q_b}$ et $[\![M_{29}]\!]_{q_c}$ appliquées au même sous arbre d'entrée t_1. Le sous arbre de droite ne se retrouve pas dans la partie droite et est donc supprimé. Les sous arbres obtenus par l'application de $[\![M_{29}]\!]_{q_b}$ et $[\![M_{29}]\!]_{q_c}$ sont des arbres linéaires représentant le chemin le plus à gauche de t_1 sur lequel chaque label interne a été remplacé respectivement par "b" et "c" et la feuille par un "d".

Le modèle permet donc la copie et la réutilisation de sous arbres pour produire plusieurs sorties différentes comme l'illustre M_{29} , mais aussi le réordonnancement et la suppression des noeuds à chaque étape de la transduction. La suppression ne permet cependant pas de supprimer un noeud ou une partie interne de l'arbre. En effet, chaque règle doit avoir pour partie droite un contexte défini sur $\mathcal{T}_\Delta(Q \times X_k)$ et ne permet pas de sortir une forêt qui pourrait venir se greffer dans le précédent contexte. Le fait de produire uniquement

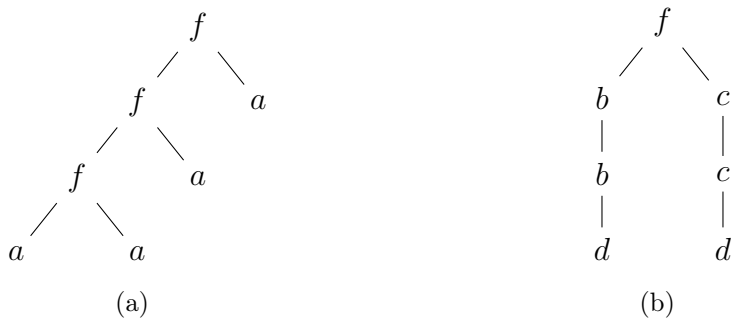


FIGURE 2.14 – arbre et le résultat de la transduction M_{29}

une paire sur $Q \times X_k$ permettrait de supprimer l'équivalent du noeud considéré dans la production mais ne pourrait traiter qu'un des fils de ce noeud.

Un dTOP M est dit *productif* si chacun de ces états q intervient dans une transduction valide, i.e. il existe un arbre $t \in \mathcal{T}_\Sigma$ tel que q se trouve dans le contexte droit d'une règle atteignable par un chemin u tel que $u \dashv t$ et que $[\![M]\!](t) \neq \varnothing$.

2.2.1.2 Régularité de l'image et du domaine

Les transducteurs d'arbres descendants ne partagent pas toutes les bonnes propriétés concernant les images et domaines, que nous avions énumérées pour les transducteurs de mots.

En contraste avec les transducteurs de mots par exemple, les images des dTOP ne sont pas toujours régulières, parce que ces transducteurs permettent de copier leurs entrées.

Lemme 16. *Il existe un dTOP dont l'image n'est pas régulière.*

Preuve. Si nous reprenons l'exemple 29 les arbres de sorties sont de la forme d ou $g(b^n(d), c^n(d))$ pour tout $n > 0$. Cette irrégularité vient de la copie permettant à plusieurs sous arbres de partager de l'information ce qui ne peut être défini par un dTA $^\downarrow$. □

Le domaine d'un dTOP par contre est toujours régulier, et même reconnu par un automate d'arbres ascendent déterministe.

Lemme 17 (Proposition 2 de Engelfriet *et al.* (2009))**.** *Le domaine d'un dTOP peut être reconnu par un automate d'arbres descendant.*

Par contre, en contraste avec les transducteurs de mots (Lemme 6), la taille de cet automate d'arbres peut être exponentielle dans la taille du transducteur. Cela est dû au fait que l'entrée d'un TOP peut être copiée et évaluée

parallèlement dans plusieurs états. Un tel saut exponentiel est inévitable dans le pire des cas, comme le montre la construction suivante.

Soit une séquence de dTA $^{\downarrow}A_1, \ldots, A_n$ ayant la même signature, on peut définir un dTOP M de taille linéaire, qui copie l'entrée n fois, et teste que l'arbre d'entrée appartient à l'intersection des langages reconnus par tout A_i.

$$\frac{q_1 \in init_{A_1} \ldots q_n \in init_{A_n}}{init_T = \{copy(q_1(x), \ldots, q_n(x))\}}$$

$$\frac{1 \leq i \leq n \qquad q \xrightarrow{f} (q_1, \ldots, q_n) \in rul_{A_i}}{q(f(x_1, \ldots, x_k)) \to f(q_1(x_1), \ldots, q_n(x_n)) \in rul(M)}$$

2.2.1.3 Équivalence

La construction de l'exemple précédent implique une différence de complexité pour le problème d'équivalence entre les dTOPs et les transducteurs de mots.

Proposition 12. *Le problème d'équivalence de d*TOP*s est* PSPACE*-dur.*

Preuve. Par réduction au problème du vide d'une intersection d'automates d'arbres déterministes. Ce problème peut être réduit au problème de vide d'une intersection de DFA qui est PSPACE-complet. Pour décider si $L(A_1) \cap \ldots \cap L(A_n) = \varnothing$ nous pouvons calculer le dTOP M comme ci dessus, et décider si $[\![M]\!] = \varnothing$. □

Cette borne inférieure de complexité n'a pas encore été donnée dans la littérature d'après nos connaissances. Elle semble remettre en question le titre "Deciding equivalence of top-down XML transformations in polynomial time" de Engelfriet *et al.* (2009). Ces résultats de polynomialité ne sont valides que pour des dTOPs travailleurs et uniformes. Ces deux propriétés, qui seront discutées en sections 2.2.1.5 et 2.2.1.6, peuvent toujours être obtenues par des algorithmes exponentiels, qui s'appliquent à la classe des dTOPs enrichie par de l'inspection.

2.2.1.4 Inspection du domaine

Le traitement du domaine nécessite donc beaucoup d'attention, notamment à cause du possible effacement d'arbres d'entrée. Les transducteurs déterministes de mots permettent en plus de représenter une transformation de contrôler le domaine ; chaque mot devant être parcouru entièrement avant de pouvoir produire un résultat valide. Ce n'est plus le cas des transducteurs descendants. En effet, les dTOP ne sont pas clos par restriction de leur domaine

à celui d'un dTA $^{\downarrow}$: étant donné un dTOP M et un dTA$^{\downarrow}$ A il n'existe pas toujours de dTOP M' reconnaissant la restriction de $[\![M]\!]$ au langage reconnu par A. Le problème est lié au fait qu'un sous arbre de l'entrée supprimé par un dTOP ne peut pas être parcouru.

Exemple 30. *Définissons un transducteur M_{30} permettant de transformer tout arbre d'entrée de $\mathcal{T}_\Sigma$ en une constante $b \in \mathcal{T}_\Delta$. Le transducteur M_{30} ci dessous produit la sortie commune b directement dans l'axiome :*

$$M_{30} = \{\Delta, \Delta, \varnothing, \{b\}, \varnothing\}$$

Il est composé uniquement d'un axiome init = $\{b\}$, sans règle ni état. Une autre manière pour définir cette transformation par un dTOP serait de sortir le "b" à la racine ou à une feuille dédiée. Dans tous les cas, le transducteur ne pourra pas parcourir entièrement les arbres d'entrée, puisque chaque sous arbre parcouru doit produire une sortie non vide. Pour cette raison, il n'est pas possible de définir par un dTOP la fonction constante "b" dont le domaine serait exactement $f(\{a, b\}, \{a, b\})$ par exemple.

Ceci montre que lors d'une suppression, un dTOP doit nécessairement ignorer totalement la forme du sous-arbre supprimé. Ceci peut empêcher le dTOP d'inspecter le domaine d'entrée comme pourrait le faire un automate d'arbre descendant déterministe.

Définition 35. *Un* transducteur descendant déterministe avec inspection*, ou dTOP^i, est une paire (M, A) composée d'un dTOP M et d'un dTA $^{\downarrow}A$ telle que le domaine de A est la signature d'entrée de M.*

Nous définissons la fonction partielle d'un dTOP^i par restriction du domaine du dTOP sous-jacent :

$$[\![(M, A)]\!] = \{(s, t) \in [\![M]\!] \mid s \in L(A)\}$$

On peut remarquer qu'un dTOP^i peut être obtenu à partir d'un dTOP en lui associant un dTA $^{\downarrow}$ayant pour langage $\mathcal{T}_\Sigma$, i.e. un dTA $^{\downarrow}$à un état total.

2.2.1.5 Transducteurs travailleurs

Notre prochain objectif est de normaliser la sortie des dTOPs avec inspection. L'idée d'une sortie normale est de produire des sorties communes le plus tôt possible, similairement aux transducteurs de mots travailleurs.

Pour formaliser cette idée, il nous faut calculer la plus grande partie commune d'un ensemble d'arbres de sortie. Pour ceci, Engelfriet *et al.* (2009) introduisent l'opérateur $\sqcup$, associatif et commutatif, produisant pour deux arbres

de $\mathcal{T}_\Delta(\{\bot\})$ un contexte sur $\mathcal{T}_\Delta(\{\bot\})$ qui correspond au plus grand contexte commun de ces deux arbres :

$$f(t_1,\dots,t_k) \sqcup f'(t'_1,\dots,t'_k) = \begin{cases} f(t_1 \sqcup t'_1,\dots t_k \sqcup t'_k) & \text{si } f = f' \\ \bot & \text{sinon.} \end{cases}$$

Comme cet opérateur est associatif et commutatif, nous pouvons l'appliquer à un ensemble de la manière suivante : $\bigsqcup\{t_1,\dots,t_k\} = t_1 \sqcup \dots \sqcup t_k$.

Considérons la fonction partielle $\tau \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Delta$. La *sortie maximale* de τ, notée $sortie_\tau$, est la plus grande partie commune des transformations des arbres de $dom(t)$, sous l'hypothèse que ce domaine soit non vide :

$$sortie_\tau = \bigsqcup\{\tau(t) \mid t \in dom(\tau)\}$$

Les transducteurs travailleurs sont introduits par Engelfriet *et al.* (2009) sur les transducteurs avec anticipation et reposent sur une production le plus tôt possible de la sortie. Cela se traduit par une sortie maximale égale à $\bot$ pour chaque transduction reconnue à partir d'un état.

Définition 36. *Un* dTOP M *productif est travailleur si* $sortie_{[\![M]\!]_q} = \bot$ *pour tout état* q *de* M*.*

Un $d\text{TOP}^i$ (M, A) est dit travailleur si M est travailleur. Le prochain exemple montre, que l'inspection peut être necéssaire pour rendre un dTOP travailleur.

Exemple 31. *Soit* $\Sigma = \Delta = \{f^{(1)}, b^{(0)}\}$ *et considérons la fonction partielle produisant toujours "*b*" à partir des arbres du domaine* $\{f(a), f(b)\}$*. Cette fonction peut être définie par un* dTOP*, qui sort le "*b*" en arrivant à la feuille, et ayant ainsi vérifié que l'arbre appartienne bien au domaine.*

En le rendant travailleur, "b*" doit être produit dans son axiome initial. Le domaine ne peut donc être inspecté que par un* $d\text{TA}^\downarrow$*. Tout parcours d'arbre devant être productif dans un* dTOP*, il est impossible de définir cette fonction par un travailleur* dTOP *sans reposer sur une inspection du domaine.*

Théorème 8 (Théorème 11 de Engelfriet *et al.* (2009))**.** *Tout* dTOP *ou* $d\text{TOP}^i$ *est équivalent à un* $d\text{TOP}^i$ *travailleur.*

La preuve de Engelfriet *et al.* (2009) se fait par une construction directe d'un transducteur travailleur avec inspection équivalent. Cela passe par le calcul, pour chaque état du préfixe commun, de chacune des sorties produites à partir de cet état. Une fois cette sortie commune identifiée, le procédé est similaire à celui vu précédemment sur les $d\text{TM}s$. On ajoute pour chaque noeud,

constitué d'une paire d'états et de variables, les préfixes communs correspondants avant de supprimer le nouveau préfixe commun de ce contexte de sortie. Cette opération peut amener la suppression de certains états, puisqu'ils ne sont plus appelés dans la partie droite. Cela peut amener une perte d'information sur le domaine du transducteur, certains sous arbres n'étant plus parcourus. L'inspection permet dans ce cas d'assurer que le domaine est préservé.

2.2.1.6 Minimisation

Une preuve alternative a été obtenue par Lemay *et al.* (2010) par un théorème du type Myhill-Nérode. Ce théorème, basé sur une congruence de transformations définie à partir d'une fonction partielle τ, montre comment on peut définir un $d\text{TOP}^i$ travailleur à partir de cette transformation. L'automate inspecteur du $d\text{TOP}^i$, obtenu pour τ, teste le domaine d'une manière «canonique», qui est appelé «compatible» par Lemay *et al.* (2010).

Théorème 9 (Théorème 28 de Lemay *et al.* (2010))**.** *Pour tout* $d\text{TOP}$ *il existe un* $d\text{TOP}^i$ *équivalent, compatible et travailleur. Un tel* $d\text{TOP}^i$ *avec un nombre minimal d'états est unique modulo renommage des états.*

Un résultat très similaire a été obtenu dans le théorème 16 de Engelfriet *et al.* (2009).

Ces résultats sont un moyen de décider de l'équivalence de deux dTOPs en les ramenant aux transducteurs uniques canonique les représentant.

2.2.1.7 Apprentissage

La minimisation unique du théorème 9 nous permet de normaliser des transducteurs de la classe $d\text{TOP}^i$. Pour chaque transformation τ définissable par un tel transducteur, nous notons par $can(\tau)$ l'unique $d\text{TOP}^i$ minimal travailleur et compatible définissant τ.

Lemay *et al.* (2010) ont montré ensuite l'apprentissage des transformations τ de cette classe à partir :

1. d'un échantillon de taille finie de paires d'entrée et de sortie $(t, \tau(t))$, et
2. d'un dTA $^{\downarrow}$reconnaissant le domaine de τ.

L'idée est d'utiliser un algorithme du type RPNI qui identifie le transducteur $\min(\tau)$ à partir de ces informations sur τ. Pour ce faire, ils se basent sur une caractérisation de $\min(\tau)$ obtenue avec leur théorème de type Myhill-Nerode pour les dTOPs (que nous ne détaillerons pas ici).

Plus formellement, nous fixons un dTA $^{\downarrow}A$ et considérons la classe $\mathcal{M}$ des dTOPs travailleurs compatibles avec A. Il existe un algorithme CHAR qui pour chaque transducteur $M \in \mathcal{M}$ retourne un échantillon de taille polynomiale

$[\![(M, A)]\!]$ appelé «caractéristique» , et un algorithme $\text{RPNI}_{d\text{TOP}}$ qui calcule à partir de $\text{CHAR}(M)$ le transducteur $can([\![(M, A)]\!])$ en temps polynomial.

Théorème 10 (Théorème 38 de Lemay *et al.* (2010))**.** *La classe de transformation τ définissable par des $d\text{TOP}^i s$ et représentée par $can(\tau)$ est apprenable en temps polynomial à partir d'un $d\text{TA}^{\downarrow}$ reconnaissant le domaine de τ et d'un échantillon pour τ de cardinalité polynomiale, qui est appelé caractéristique.*

2.2.2 Transducteur ascendant

Les transducteurs ascendants sont un autre formalisme permettant de représenter les transformations d'arbres de $\mathcal{T}_\Sigma$ en arbres de $\mathcal{T}_\Delta$ mais cette fois en parcourant l'arbre d'entrée de ses feuilles à sa racine.

Définition 37. *Un transducteur ascendant d'arbres* BOT *est défini par un n-uplet $M = (\Sigma, \Delta, Q, fin, rul)$ composé des alphabets bornés Σ et Δ des symboles des arbres d'entrée et de sortie, d'un ensemble fini d'états Q, une relation finale $fin \subseteq Q \times \mathcal{T}_\Delta(x_0)$ associant à un état les productions finales possibles et un ensemble de règles rul de la forme :*

$$(f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle), (q, t)) \text{ pour } f \in \Sigma^{(k)},\ q, q_1, \ldots q_k \in Q \text{ et } t \in \mathcal{T}_\Delta(X_k)$$

Chaque état q d'un transducteur M définit une relation, entre arbres d'entrée et de sortie, qui correspond à la plus petite relation $[\![M]\!]_q \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Delta$ telle que pour tout état q :

$$[\![M]\!]_q = \{(t, s[x_i \leftarrow s_i \mid 1 \le i \le k,\ q_i \in Q,\ (t_i, s_i) \in [\![M]\!]_{q_i}]) \mid t = f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma,\ (f(q_1, \ldots, q_k), (q, s)) \in rul\}$$

En se basant sur cette définition, la relation $[\![M]\!]$ est définie de la manière suivante :

$$[\![M]\!] = \{(t, s[x_0 \leftarrow s' \mid q \in Q,\ (t, s') \in [\![M]\!]_q]) \mid \langle q, s\rangle \in fin\}$$

La notion déterminisme pour les BOT*s* s'approche de celle des TOP*s*, partageant une même structure mais travaillant dans le sens inverse. Un BOT est *déterministe*, noté dBOT, s'il n'existe qu'une règle pour une même partie gauche.

Dans ce cas, l'ensemble des règles *rul* peut être vu comme une fonction partielle, chaque règle représentée de la forme $f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to (q, t)$. De même l'étape de finalisation *fin* est une fonction partielle associant à chaque état au plus un contexte de sortie.

Exemple 32. *Prenons une transduction permettant à partir d'un arbre représentant une opération logique de sortir un arbre contenant un unique noeud, résultat de cette opération. Dans cet exemple, nous nous limiterons au* dBOT M_{32} *travaillant avec l'opérateur «et» d'arité 2 ainsi que les valeurs booléennes* 0 *et* 1 *d'arité 0 qui constituent l'alphabet d'entrée.*

Il est composé d'états q_0 *et* q_1 *correspondant au calcul du sous arbre. L'ensemble des règles rul est composé de :*

$$\begin{array}{ll} 0 \to (q_0, 0) & 1 \to (q_1, 1) \\ et(\langle q_0, x_1\rangle, \langle q_0, x_2\rangle) \to (q_0, 0) & et(\langle q_0, x_1\rangle, \langle q_0, x_2\rangle) \to (q_0, 0) \\ et(\langle q_1, x_1\rangle, \langle q_0, x_2\rangle) \to (q_0, 0) & et(\langle q_1, x_1\rangle, \langle q_1, x_2\rangle) \to (q_0, 1) \end{array}$$

La finalisation correspond à sortir le résultat obtenu, soit $fin(q_0) = x_0$ *et* $fin(q_1) = x_0$.

Pour illustrer son exécution prenons l'arbre $et(1, et(1, 0))$. *Le résultat de la transduction sur l'arbre nous donne le couple* $(q_0)(0)$ *qui nous produit* 0 *par la fonction partielle fin.*

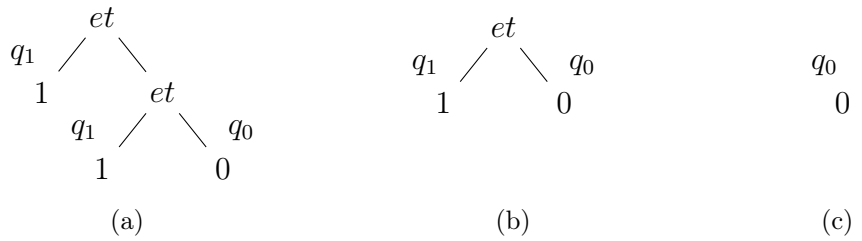


FIGURE 2.15 – transduction M_{32}

L'intégralité de l'arbre a été prise en compte avant de produire une sortie composée d'un unique symbole ce qui n'était pas possible dans un dTOP. *De plus la sortie dépend du contenu des deux sous arbres à chaque étape, point qui ne peut être pris en compte dans une transformation descendante. L'absence de variable dans le contexte de sortie est dûe au fait que ce transducteur prend en compte les sorties produites par un sous arbre sans les intégrer dans sa sortie finale, chose impossible pour un* dTOP *qui devrait ignorer de ce fait les sous arbres.*

Malgré la forte ressemblance entre les formalismes utilisés pour les dBOTs et dTOPs, les modèles ont une expressivité fortement différente. Dans les dTOPs la copie permet de réutiliser plusieurs fois un même sous arbre de l'entrée pour produire des sorties possiblement différentes ce qui n'est pas le cas des dBOTs copiant directement le résultat unique de la transduction de ces sous arbres. Une transformation comme l'illustre l'exemple 29 n'est donc pas définissable par ces machines.

La suppression d'un sous arbre pour les dBOTs nécessite un parcours correct de ce sous arbre et ne produit pas la sortie associée même si l'état atteint peut être pris en compte. Du coup, les transducteurs ascendants sont aptes à contrôler leur domaine intégralement sans inspection, en contraste avec des dTOPs qui ne peuvent pas inspecter les sous arbres qu'ils suppriment.

2.2.2.1 Forme normale

Une forme normale pour les dBOTs est proposée par Friese *et al.* (2010), afin de distinguer des représentants uniques minimaux. Ils sont sujets à de nombreuses propriétés techniques qui ne seront pas abordées ici (voir son théorème 16). En particulier, il est nécessaire de normaliser la sortie en définissant une notion de dBOT travailleur. Il n'existe cependant pas de manière unique de pousser la sortie vers les feuilles, en particulier pour les transformations finies. La notion de travailleur n'est donc pas aussi facilement définissable que pour les dTOPs, mais existe.

Le théorème 16 assure la décidabilité du problème d'équivalence entre deux dBOTs, revenant au test d'égalité entre leurs représentants respectifs.

2.2.2.2 Apprentissage

Dans sa thèse, Sylvia Friese (Friese, 2011) propose une caractérisation des transformations reconnues par les transducteurs déterministes ascendants en introduisant la notion de «partitions». Ces partitions associent, à chaque sous arbre des arbres d'entrée d'une transformation τ, une fonction résiduelle composée d'informations sur les possibles contextes de ce sous arbre.

Une congruence y est introduite pour relier ces fonctions résiduelles et arriver à la définition du théorème 6.1 dans le style de Myhill-Nérode. Ce théorème lie les transformations aux dBOTs et à l'unique représentant canonique de chacune de ces transductions.

Malgré l'existence d'un théorème de Myhill-Nérode, il n'existe pour le moment aucun résultat d'apprenabilité pour ce modèle, particulièrement l'existence d'un algorithme polynomial d'apprentissage reste un problème ouvert.

2.2.3 Transducteur avec anticipation

Le principal défaut des dTOPs est généralement de ne pas pouvoir évaluer des sous-arbres avant de les réécrire. La seule manière d'évaluer partiellement des sous-arbres revient à annoter certaines informations de l'entrée dans l'état du transducteur, avant de démarrer la réécriture.

Par exemple, il existe un dTOP qui transforme chaque sous-arbre, dont l'intégralité des enfants ont pour étiquette "b", en l'arbre B. Il est cependant

impossible pour un dTOP de substituer par B chaque sous-arbre de l'entrée dont l'ensemble des feuilles sont étiquetées par b. Ceci peut être fait par un dBOT, puisqu'ils évaluent un sous-arbre avant de le réécrire. La réécriture ne tient pas, dans la plupart des cas, en compte le contexte de celui ci.

Pour pallier ce manque, Engelfriet (1977) introduit un modèle autorisant au transducteur descendant d'évaluer les sous-arbres d'un noeud avant de le transformer. À l'instar des transducteurs de mots, cette *anticipation* peut être faite à travers un automate d'arbres ascendant annotant chaque arbre d'entrée avant sa transduction.

Définition 38. *On appelle* transducteur descendant déterministe avec anticipation*, une paire $N = (A, M)$ composée d'un automate d'arbres ascendant déterministe A défini par le n-uplet (Σ, P, fin, rul_A) et d'un dTOP M défini sur l'alphabet d'entrée $\Sigma \times P$ correspondant aux noeuds d'un arbre de $\mathcal{T}_\Sigma$ annoté par A.*

La sémantique reste classique. Ainsi une paire $(t, s) \in [\![N]\!]$ si et seulement si $t \in \mathcal{L}_A$ et qu'il existe une exécution de M sur l'arbre annoté t' tel que $(t', s) \in [\![M]\!]$.

Exemple 33. *L'exemple 2.15 qui n'était définissable que par un transducteur ascendant l'est par un dTOP$^\ell$ $N_{33} = (A_{33}, M_{33})$ où A_{33} à pour alphabet d'entrée $\Sigma = \{0^{(0)}, 1^{(0)}, et^{(2)}\}$, avec deux états acceptants p_0 et p_1 et les règles suivantes :*

$$\begin{array}{lll} 0 \to p_0 & et(p_0, p_0) \to p_0 & et(p_0, p_1) \to p_0 \\ 1 \to p_1 & et(p_1, p_0) \to p_0 & et(p_1, p_1) \to p_1 \end{array}$$

Le dTOP M_{33}, quant à lui, est défini sur $(\Sigma \times Q_{A_{33}}, \{0^{(0)}, 1^{(1)}\}, \{\langle q_0, x_0, \rangle\}, rul_{M_{33}})$, l'ensemble des règles ne contenant que quatre règles produisant directement l'intégralité de la sortie en fonction de l'état annoté :

$$\begin{array}{ll} q_0(0, p_0) \to 0 & q_0((et, p_0)(x_1, x_2)) \to 0 \\ q_0(1, p_1) \to 1 & q_0((et, p_1)(x_1, x_2)) \to 1 \end{array}$$

En reprenant pour exemple l'arbre $et(1, et(1, 0)$, la représentation de l'exécution correspond à l'arbre annoté par A de la figure 2.16. L'automate d'anticipation fini par associer à la racine l'état p_0. Cet état permet au transducteur M de produire l'arbre 0 dès la lecture de la racine.

Toute transduction définie par un dTOP^i l'est aussi par un dTOP$^\ell$ ayant comme anticipation un automate représentant le domaine de l'inspection, annotations qui ne seront pas pris en compte par le dTOP correspondant à celui du dTOP^i initial. L'automate ascendant peut être de taille exponentielle dans celle de l'automate d'inspection descendant.

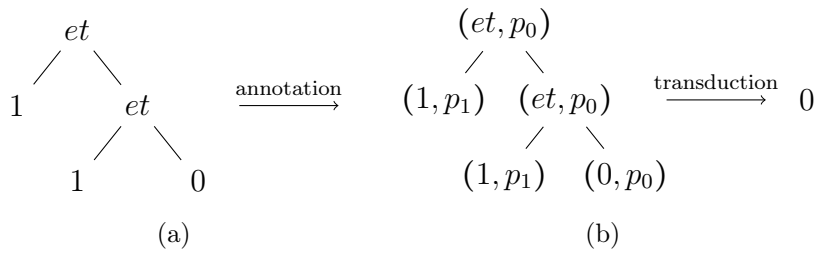


FIGURE 2.16 – Exemple d'annotation et transduction par N_{33}

Théorème 11 (Théorème 3.2 de Engelfriet (1977))**.** *L'ensemble des transductions définies par les d*BOT*s est strictement contenu dans l'ensemble des transductions définissables par les d*TOP$^\ell$*s.*

Ce résultat est illustré par l'exemple 33 mais la réduction n'est pas toujours aussi directe, car les états atteints ne sont pas toujours suffisants pour décider de la sortie à produire et de la continuation à suivre. La preuve de ce résultat peut se faire par construction du transducteur correspondant. Pour un dBOT $M = (\Sigma, \Delta, P, fin_M, rul_M)$, on créé dans un premier temps un automate d'anticipation annotant l'arbre par les règles de M représentant son exécution. A est défini par le tuple (Σ, rul_M, rul_A) ayant pour états les règles de M et pour règles :

$$\frac{r = f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \ \to q(t) \in rul_M \qquad \forall r_i \in rul_M,\ 1 \le i \le k, dt(r_i) = q_i(t_i)}{f(r_1, \ldots r_k) \to r \in rul_A}$$

$dt(r)$ étant la partie droite de la règle r. La taille de cet automate peut être exponentielle dans le nombre et la taille des règles de M.

Une fois l'arbre annoté, il ne reste plus qu'a produire la sortie correspondante aux règles annotées grâce au dTOP $M' = (\Sigma \times rul_M, \Delta, \{\diamond, \bullet\}, \{\langle \diamond, x_0\rangle\}, rul_{M'})$. Les règles à partir de l'état $\bullet$ sont de la forme :

$$\frac{r = f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \ \to q(t) \in rul_M}{\bullet(f, r)(x_1, \ldots, x_k) \to t[x_i \leftarrow \langle \bullet, x_i\rangle \mid 1 \le i \le k]}$$

Les règles sortant de l'état initial doivent également produire la sortie associée à la finalisation du transducteur ascendant :

$$\frac{r = f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \ \to q(t) \in rul_M \qquad (q, t_0) \in fin_M}{\diamond(f, r)(x_1, \ldots, x_k) \to t_0[x_0 \leftarrow t[x_i \leftarrow \langle \bullet, x_i\rangle \mid 1 \le i \le k]]}$$

L'arbre annoté ainsi obtenu stocke sur chaque noeud l'intégralité de l'information correspondant à une exécution de M sur cet arbre et plus précisément le contexte de sorties à sortir par le transducteur descendant généré en suivant

ces informations. Le transducteur reste dans une taille équivalente à celle du dBOT de départ.

Théorème 12 (Corollaire 7 de Engelfriet *et al.* (2009))**.** *Le problème d'équivalence pour la classe de dTOP$^\ell s$ est décidable.*

Ceci est un corollaire du théorème 18, qui montre que l'équivalence des dTOPs est décidable. La preuve repose sur la création d'un nouvel automate d'anticipation annotant chaque noeud par les états des deux précédents automates d'anticipation. Une fois obtenu, il suffit de tester l'équivalence des deux dTOPs adaptés sur ce même domaine. En suivant le précédent résultat, cela assure, une fois de plus, la décidabilité de l'équivalence pour la classe des dBOTs.

L'existence d'une bonne notion de dTOP$^\ell s$ travailleurs ou de tout autre forme normale assurant un représentant unique reste un problème ouvert. Ceci semble être un problème difficile vu que les dTOP$^\ell s$ sont plus expressifs que les dBOTs et qu'il ne semble pas possible de minimiser l'anticipation comme dans le cas des mots (Théorème 2 par Reutenauer et Schützenberger (1991)).

2.2.4 Macro-transducteurs descendants

Les *macro-transducteurs d'arbres*, dénotés MTT, ont été introduits par Engelfriet (1980) et combinent les transducteurs d'arbres descendants aux macro-grammaires introduites par Fisher (1968). Ils associent aux règles classiques d'un transducteur d'arbre descendants un ensemble de paramètres $Y_m = \{y_1, \ldots, y_m\}$ permettant de transmettre la sortie à travers l'appel d'un état. Chaque état n'autorise qu'un même nombre de paramètres dans la partie gauche de ces règles. L'arbre de sortie contient maintenant des noeuds internes composés de couples états/variables et de paramètres à certaines feuilles. Chacun des fils d'un état est un paramètre pour l'évaluation du sous arbre associé. Le nombre de paramètres peut donc être vu comme l'arité associée à cet état. Une règle de cette machine sera donc de la forme suivante :

$$q(f(x_1, \ldots, x_k)) : (y_1, \ldots, y_m) \to t$$

où k est l'arité de q, m celle de l'état q et t le contexte de sortie défini sur $T_{(\Delta \cup (Q \times X_k))}(Y_m)$.

Exemple 34. *Pour illustrer ces machines, prenons un exemple où l'on souhaite linéariser un arbre d'arité borné défini sur $\Sigma = \{f^{(2)}, b^{(1)}, a^{(0)}\}$ dans la suite de balises ouvrantes et fermantes le représentant sous forme XML, sur $\Delta = \{\langle f\rangle^{(1)}, \langle /f\rangle^{(1)}, \langle b\rangle^{(1)}, \langle /b\rangle^{(1)}, \langle a\rangle^{(1)}, \langle /a\rangle^{(1)}, \#^{(0)}\}$ dans le bon ordre.*

Le MTT *correspondant est défini par* $M_{34} = (\Sigma, \Delta, \{q_0, q\}, \{q_0\}, rul)$ *où* q_0 *est l'état initial ne prenant aucun paramètre et* q *un état prenant un paramètre. Les règles sont de la forme :*

$$\begin{array}{ll} q_0(x_0) & \to q(x_0) : (\#) \\ q(a) : (y_1) & \to \langle a\rangle(\langle /a\rangle(y_1)) \\ q(f(x_1, x_2)) : (y_1) & \to \langle f\rangle(q(x_1) : (q(x_2) : (\langle /f\rangle(y_1)))) \\ q(b(x_1)) : (y_1) & \to \langle b\rangle(q(x_1) : (\langle /b\rangle(y_1))) \end{array}$$

M_{34} permet à travers ces paramètres de produire de la sortie en tête et en queue de l'arbre produit en gardant en mémoire la sortie nécessaire comme l'illustre la figure 2.17 sur l'arbre $t = f(b(a), a)$.

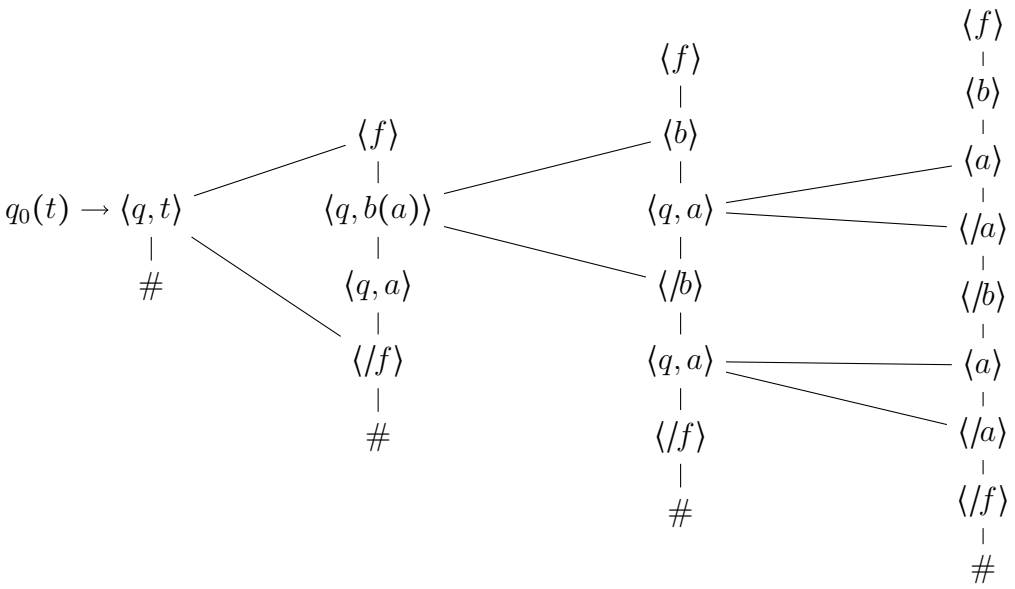


FIGURE 2.17 – Exemple d'exécution de M_{34} sur l'arbre t.

M_{34} est cependant un cas particulier ne supprimant ou dupliquant aucun des paramètres. Cela donne l'effet de développer les résultats de transductions à l'intérieur de l'arbre ce qui n'est pas réellement le cas, les sous arbres retenus étant à chaque étape un paramètre transmis à travers la règle.

La classe des macro-transducteurs d'arbres contient strictement la classe des transducteurs descendants d'arbres, chacun d'entre eux correspondant à un macro-transducteur où chaque état n'accepterait aucun paramètre. Plus expressive, comme l'illustre l'exemple 34, non exprimable par un des modèles précédemment considérés elle reste cependant transversale aux modèles ascendants.

En ajoutant une anticipation aux macro-transducteurs d'arbres, de la même manière que sur les transducteurs descendants, Engelfriet et Maneth (1999) donnent une caractérisation de ces machines et établissent un parallèle aux transformations définissables par les modèles logiques.

Les problèmes classiques, tels que celui de l'équivalence, restent cependant ouverts pour les macro-transducteurs avec ou sans anticipation.

2.3 Transducteurs d'arbres en mots

Dans les précédentes sections nous nous sommes intéressés aux transductions gardant le même type de structure de données dans la sortie que celle de l'entrée, ne pouvant changer que l'alphabet sur lesquels ils sont définis (mots en mots ou arbre d'artité bornée en arbre d'arité bornée).

Nous passons maintenant à des classes de transductions d'arbres en mots. Nous considérons deux classes distinguables par l'ordre de parcours de l'arbre d'entrée choisi, qui peut se faire de manière ascendante ou descendante. Ils produisent à chaque étape des facteurs du mot de sortie. Ces facteurs sont concaténés au fur et à mesure de l'exécution de la transduction. Cette concaténation ajoute une nouvelle expressivité, l'équivalent étant inexistant dans les classes de transducteurs d'arbres à arbres étudiés précédemment. Pour chacune de ces classes, nous identifierons une restriction limitant la sortie à n'utiliser qu'une fois chaque noeud de l'entrée et gardant l'ordre du parcours de l'entrée lors de la production de la sortie, nommée modèle séquentiel.

L'intérêt de ce genre de machines est leur capacité de représentation des transformations qui produisent des mots imbriqués en sortie et donc des documents XML ou séquences d'arbres contenant des données textuelles pour lesquels la question de l'arité, bornée ou non, ne se pose pas. Le fait de pouvoir sortir des séquences de documents et de pouvoir concaténer les facteurs de telles séquences est particulièrement important pour la sortie de documents XML, opération présente et indispensable lors transformation du type XSLT. La concaténation permet aussi de simuler la production de sortie de transducteurs d'arbres ascendants ou descendants. Cette simulation, que nous utiliserons ultérieurement comme moyen de décider de l'équivalence de deux modèles d'une même classe, ne dépend pas du fait que la production soit séquentielle ou non.

2.3.1 Transducteurs descendants

Définition 39. *Un* transducteur descendant d'arbres en mots déterministe *(d*T2W*) est un tuple* $M = (\Sigma, \Delta, Q, init, rul)$ *composé d'un alphabet* Σ *(d'arité bornée) de symboles étiquettant les arbres d'entrée, d'un alphabet* Δ *de mots*

de sortie, d'un ensemble fini d'états Q, d'une règle initiale $init \in (\Delta^* \mid (Q \times \{x_0\}))^*$ et de règles rul représentées par une fonction partielle de $Q \times \Sigma^{(k)}$ vers $(\Delta \cup (Q \times X_k))^*$.

De base, aucune restriction n'est faite sur l'ordre et le nombre d'utilisations des variables de X_k dans la production. Une règle d'un dT2W peut être notée de la manière suivante :

$$\langle q, f(x_1, \ldots, x_k)\rangle \to w_0 \cdot \langle q_1, x_{i_1}\rangle \cdot \ldots \cdot \langle q_m, x_{i_m}\rangle \cdot w_m$$

où l'ensemble $\{i_1, \ldots, i_m\}$ est une séquence d'indices compris entre $1 \leq i_j \leq k$ pour tout $1 \leq j \leq m$.

La règle initiale d'un dT2W M peut se réduire, sans perte d'expressivité, à un unique état. Prenons $init = \{w_0 \cdot \langle q_0, x_0\rangle \cdot \ldots \langle q_k, x_0\rangle \cdot w_k\} = \{w\}$. Pour remplacer la règle initiale par un état unique p_0, il faut que pour tout $f \in \Sigma$ d'arité k', s'il existe une règle $\langle q_j, f(x_1, \ldots, x_{k'})\rangle \to w_j \in rul$ pour chaque $1 \leq j \leq k$, alors on ajoute la règle $\langle p_0, f(x_1, \ldots, x_{k'})\rangle \to w[\langle q_j, x_0\rangle \leftarrow w_j]$ dans l'ensemble des règles.

Nous nous intéresserons principalement aux dT2Ws dits *séquentiels*, notés dST2Ws, ne permettant ni copie ni réordonnancement des noeuds de l'entrée pour la production de la sortie. Cela se traduit par une partie droite faisant apparaitre une et une seule fois chaque variable de X_k dans l'ordre de l'entrée, ainsi qu'une seule apparition de la variable x_0 dans la phase d'initialisation. Les règles d'un dST2Ws sont de la forme :

$$q(f(x_1, \ldots, x_k)) \to u_0 \cdot \langle q_1, x_1\rangle \cdot \ldots \cdot \langle q_k, x_k\rangle \cdot u_k$$

Pour une meilleure lisibilité, les variables X_k pourront être éludées sur ces règles, sachant que chacune d'elles n'est présente qu'une et unique fois dans l'ordre d'entrée. Cela donnera le formalisme, proche des automates d'arbres, suivant :

$$q \xrightarrow{f} u_0 \cdot q_1 \cdot \ldots \cdot q_k \cdot u_k$$

Dans ce cas, même un sous arbre supprimé, i.e. n'étant pas utilisé pour définir la sortie, nécessitera quand même un parcours complet sans production. Le domaine de ces machines correspond à l'automate d'arbres obtenu en supprimant toutes les sorties possibles. Dans sa forme déterministe, le domaine d'un dST2W est défini par un $d\text{TA}^{\downarrow}$. On peut donc facilement étendre la définition de la fonction $\hat{rul}$ aux transducteurs, en associant à un état et un chemin étiqueté l'état atteint s'il existe.

Pour les dST2Ws non séquentiels, l'obtention du domaine n'est pas si triviale. L'absence de certaines variables dans la partie droite implique que

certains sous arbres ne sont pas parcourus et peuvent donc prendre n'importe quelle valeur. D'autres variables répétées plusieurs fois correspondent à la copie d'un sous arbre qui doit être valide pour toutes les transductions correspondantes, soit un domaine dépendant de multiples automates.

La transformation $[\![M]\!]$ définie par un dT2W M passera par la définition de fonctions partielles $[\![M]\!]_q$ d'arbres en mots associant à un état les transformations qu'ils expriment. Cette fonction se définit par :

$$[\![M]\!]_q(f(t_1,\dots,t_k)) = \begin{cases} \delta[\langle q_i, x_j\rangle \leftarrow [\![M]\!]_{q_i}(t_j)] \\ \quad \text{si } q(f(x_1,\dots,x_k)) \to \delta \in rul_M \\ \text{indéfini sinon.} \end{cases}$$

Une fois ces fonctions partielles définies, la transformation $[\![M]\!]$ correspond à la fonction partielle d'arbres en mots suivante :

$$[\![M]\!](t) = \delta[\langle q, x_0\rangle \leftarrow [\![M]\!]_q(t)] \text{ pour } \delta \in init_M$$

La notion d'exécution d'un dST2W sur un arbre t peut être définie par une paire $t * \beta$ où $\beta \subseteq noeuds(t) \to rul$, associant à chaque noeud de t une règle de transducteur. De la même manière que les transducteurs de mots, cette exécution est dite *valide* si toute règle la composant est la continuation de celle qui la précède et si elle consomme le bon label. Formellement, pour chaque noeud $\pi \in noeuds(t)$, $\beta(\pi) = \langle q, f(x_1,\dots,x_k)\rangle \to w_0 \cdot \langle q_1, x_1\rangle \cdot \dots \langle q_k, x_k\rangle \cdot w_k$ implique que $labels_t(\pi) = f$ d'une même arité $k = ar(f)$, et que chaque fils $\pi\cdot j$, pour $1 \le j \le k$, est étiqueté par une règle de la forme $\beta(\pi \cdot j) = \langle q_j, \_\rangle \to \_$. Elle est dite *complète* si elle est valide et si $\beta(\varepsilon) = \langle q_0, \_\rangle \to \_$ alors $init \in (\Delta^* \times \{\langle q_0, x_0\rangle\} \times \Delta^*)$.

Si un dST2W n'autorise qu'un unique état comme initialisation, ce qui peut être obtenu pour tout dST2W, alors toute la production est directement contenue dans son exécution. Nous pouvons donc définir une fonction $sortie : \mathcal{T}_{rul} \to \Delta^*$ se définissant pour une règle $r = \langle q, f(x_1,\dots x_k)\rangle = w_0 \cdot \langle q_1, x_1\rangle \cdot \dots \cdot \langle q_k, x_k\rangle \cdot w_k$, de la manière suivante :

$$sortie(r(t_1,\dots,t_k)) = w_0 \cdot sortie(t_1) \cdot \dots \cdot sortie(t_k) \cdot w_k$$

La définition d'une exécution d'un dT2W, non séquentiel, ne peut pas se faire par une simple annotation de t puisque l'intégralité de sa structure peut être modifiée durant l'exécution du transducteur. Il est cependant possible de la représenter, non plus par une annotation de t mais, par un autre arbre. Cet arbre, proche de la notion d'arbre syntaxique étendu pour une *CFG*, a pour noeud une règle r. Chacun de ces noeuds a pour fils l'ensemble des règles correspondant à la continuation de l'exécution pour les paires $\langle q_i, x_i\rangle$ contenues dans la règle l'annotant. Nous ne rentrerons pas plus en détail sur cette définition.

Exemple 35. *Pour illustrer ces machines, intéressons nous à un* dT2W M_{35} *définissant la transformation des arbres de* $f(g^m(a), g^n(a))$ *aux mots* $(abc)^m ac(abc)^n$ *correspondants.*

M_{35} *a pour règle initiale* $\langle q_0, x_0\rangle \in init$ *ne produisant aucune sortie et les règles suivantes :*

$$q_0(f(x_1, x_2)) \to \langle q_1, x_1\rangle \cdot ac \cdot \langle q_1, x_2\rangle$$

$$q_1(g(x_1)) \to \langle q_1, x_1\rangle \cdot abc \qquad q_1(a) \to \varepsilon$$

Les macro-transducteurs d'arbres descendants englobent les transducteurs descendants d'arbres en mots en représentant le résultat de la transduction sous la forme d'une linéarisation de la sortie, avant la concaténation, finalisée par le symbole #.

Proposition 13. *Pour tout* dT2W M*, nous pouvons construire un* MTT M' *en temps et taille linéaire dans la taille de* M*.*

Preuve. (esquisse) Prenons un dT2W $M = (\Sigma, \Delta, Q, init, rul)$, nous pouvons construire un MTT M' défini sur le même alphabet d'entrée Σ et l'alphabet de sortie $\Delta' = \{u^{(1)} \mid u \in \Delta\} \cup \{\#\}$, d'un même ensemble d'états auquel on ajoute q_0 l'état initial, dont les règles sont construites de manière suivante :

$$\frac{q(f(x_1, \ldots, x_k)) \to u_0 \cdot \langle q_1, x_{i_1}\rangle \cdot \ldots \langle q_m, x_{i_m}\rangle \cdot u_m \in rul}{q(f(x_1, \ldots, x_k)) : (y_1) \to u_0(\langle q_1, x_{i_1}\rangle(u_1(\langle q_2, x_{i_2}\rangle \ldots \langle q_m, x_{i_m}\rangle(u_m(\#)) \ldots)))}$$

et de la même manière, d'une initialisation :

$$\frac{u_0 \cdot \langle q_1, x_0\rangle \cdot \ldots \cdot \langle q_m, x_0\rangle \cdot u_m}{q_0(x_0) \to u_0(\langle q_0, x_0\rangle(u_1(\ldots(\langle q_m, x_0\rangle(u_m))))) \in init}$$

□

2.3.2 Transducteurs ascendants

Définition 40. *Un* transducteur ascendant d'arbre en mots déterministe *(*dB2W*) est un tuple* $M = (\Sigma, \Delta, Q, fin, rul)$ *composé d'un alphabet d'arité bornée* Σ *contenant les labels des arbres d'entrée, d'un alphabet* Δ *de mots de sortie, d'une fonction partielle finale fin de* $Q \times \{x_0\}$ *vers* $(\Delta^* \mid \{x_0\})^*$ *et de règles rul représentées par une fonction partielle de* $\Sigma \times (Q \times X_k)^*$ *vers* $Q \times (\Delta \cup X_k)^*$*.*

Les règles se représentent sous la forme suivante :

$$f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q, w_0 \cdot x_{i_1} \cdot \ldots x_{i_m} \cdot w_m\rangle$$

où l'ensemble $\{i_1, \ldots, i_m\}$ est une séquence d'indices compris entre $1 \leq i_j \leq k$ pour tout $1 \leq j \leq m$.

Comme dans le modèle descendant, les dB2Ws peuvent être restreints au modèle séquentiel, devant utiliser dans la partie droite chaque variable de la partie gauche une et unique fois et dans le même ordre. Les règles sont donc de la forme :

$$f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q, u_0 \cdot x_1 \cdot \ldots x_k \cdot u_k\rangle$$

Malgré une forte similitude entre la syntaxe des dT2Ws et dB2Ws séquentiels, ils se distinguent dans leurs sémantiques. L'état d'un dT2W dépend du chemin qui le lie à la racine, chemin étant directement responsable de la manière dont la sortie associé au sous arbre correspondant est produite puis greffé dans la sortie globale. Pour les dB2Ws, l'effet est inverse. L'état dépend de l'intégralité du sous arbre dont il annote la racine, sous arbre dont la production est déjà associée à cet état. L'état participe à la décision de la sortie qui l'englobe et ne la subit pas.

La similitude syntaxique permet une même représentation de l'exécution pour le cas séquentiel, à savoir une paire $t * \beta$. Même si l'exécution ne se fait pas dans le même sens, le résultat final est le même à savoir un arbre annoté de règles. La validité d'une telle exécution se vérifie d'une manière équivalente à celle des dT2Ws. L'exécution est dite *complète* si elle est valide et si $\mathit{fin}(\langle q_0, x_0\rangle)$ est défini pour une règle de la forme $\beta(\varepsilon) = \_ \to \langle q_0, \_\rangle$ à la racine.

À la différence des dT2Ws, le fait de n'avoir que des états dans la finalisation, à la place d'une production et d'un état, restreint l'expressivité. Le fait d'être dans la racine est le seul événement permettant de savoir si oui ou non cette production doit être effectuée. Cet événement reste indétectable par la partie gauche d'une règle ascendante. Il n'est donc pas possible de rester déterministe en poussant cette sortie.

Prenons par exemple la finalisation $\mathit{fin}(\langle q, x_0\rangle) = w_0 \cdot x_0 \cdot w_1$ et une règle $f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q, w\rangle$. Le seul moyen de produire la sortie avant la finalisation reviendrait à ajouter $f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q_f, w_0 \cdot w \cdot w_1\rangle$, q_f étant un état final, sans pour autant supprimer l'ancienne règle ce qui implique deux parties droites possibles pour une même partie gauche.

Cependant nous pouvons créer virtuellement de nouvelles règles sur les exécutions associées pour avoir toute la sortie contenue dans l'arbre d'exécution. Ces exécutions étant terminées, le fait d'être à la racine de sa représentation nous apporte l'information précédemment manquante. Ainsi on peut construire, à partir d'une exécution $t*\beta$ de M, une exécution $t*\beta'$ dite *virtuelle totale* en remplaçant la règle associée à la racine, $\beta(\varepsilon) = f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to$

$\langle q, w\rangle$, par $\beta'(\varepsilon) = f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q_f, w_0 \cdot w \cdot w_1\rangle$, le reste de la fonction restant identique.

Pour ce qui est de la transformation $[\![M]\!]$ définie par un dB2W, nous pouvons l'identifier à l'aide d'une fonction partielle $[\![M]\!]_q$, donné par le plus ensemble tel que :

$$\frac{a \to \langle q, w_0\rangle \in rul}{[\![M]\!]_q(a) = w_0}$$

$$\frac{f(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q, \delta\rangle \in rules \qquad [\![M]\!]_{q_i}(t_i) = w_i}{[\![M]\!]_q(f(t_1, \ldots, t_k)) = (\delta[x_i \leftarrow w_i])}$$

Une fois définie, pour tout arbre $t \in \mathcal{T}_\Sigma$ tel que $[\![M]\!]_{q_0} = w$ tel que $fin_M(\langle q_0, x_0\rangle) = \delta$, la transformation $[\![M]\!]$ est défini pour cet arbre est défini par $[\![M]\!](t) = \delta[x_0 \leftarrow w]$.

L'équivalence était connue décidable pour les deux modèles restreints aux séquentiels sans pour autant en connaître la complexité. Dans le cas général, sans restriction sur les copies et le réordonnancement, le problème d'équivalence reste indécidable. Il n'existe pas de résultat pour ce qui est de la normalisation et de l'apprentissage dans la littérature.

2.4 Transducteurs d'arbres d'arité non bornée

Jusqu'à présent, tous les arbres considérés étaient d'arité bornée, chaque noeud d'un même label ayant toujours un même nombre de fils. En se limitant à ce format, les transducteurs bénéficient d'une structure plus simple. Mais chacune de ces règles, étant de taille fixée par l'étiquette du noeud considéré, ces modèles se détachent des cas réels. En effet, la plupart des arbres de données se trouvant dans les cas pratiques sont d'arité non bornée. Deux principaux moyens s'offrent à nous pour traiter ce genre de données.

Nous pouvons nous ramener à des arbres d'arité bornée à l'aide d'encodages tels que le codage curryfié, plus adapté aux machines ascendantes, et le codage frère-fils pour les descendantes. Le principal problème est qu'en exécutant des transducteurs, sur ces encodages, l'expressivité s'exprime plus difficilement en fonction de l'arbre de base. Des codages comme celui guidé par les schémas permettent de rester plus proche de l'arbre d'entrée et n'a été que récemment introduit.

L'autre possibilité, que nous allons développer dans la suite, est l'utilisation ddes transducteurs de mots imbriqués, travaillant sur la représentation linéaire d'arbres. Nous reprendrons ici les automates de mots imbriqués auxquels nous associeront une sortie structurée ou non.

2.4.1 Transducteurs de mots imbriqués en mots

Les *transducteurs de mots imbriqués en mots*, introduits dans un premier temps par Raskin et Servais (2008) sous le nom de « visibly pushdown », permettent de passer de mots imbriqués en mots imbriqués. La définition, que nous donnons ici, et que nous utiliserons dans de futurs chapitres, n'effectue aucun contrôle sur la sortie et permet de produire tout type de mots sans vérifier une possible bonne imbrication.

Les *transducteurs de mots imbriqués en mots* correspondent à un NWA où les règles sont enrichies d'un chaîne de sortie qui sera produite au fur et à mesure de l'exécution du transducteur.

Exemple 36. *Si nous reprenons l'exemple 10 pour produire un fichier* HTML*, qui reprend uniquement les noms d'une liste, il suffit de modifier chaque règles en y ajoutant la sortie correspondante. Si, au lieu d'annoter chaque ouverture et fermeture de noeud par l'état atteint, nous y annotons la sortie produite, la production finale correspondra à la concaténation de chaque partie de cette sortie en parcourant l'arbre en profondeur.*

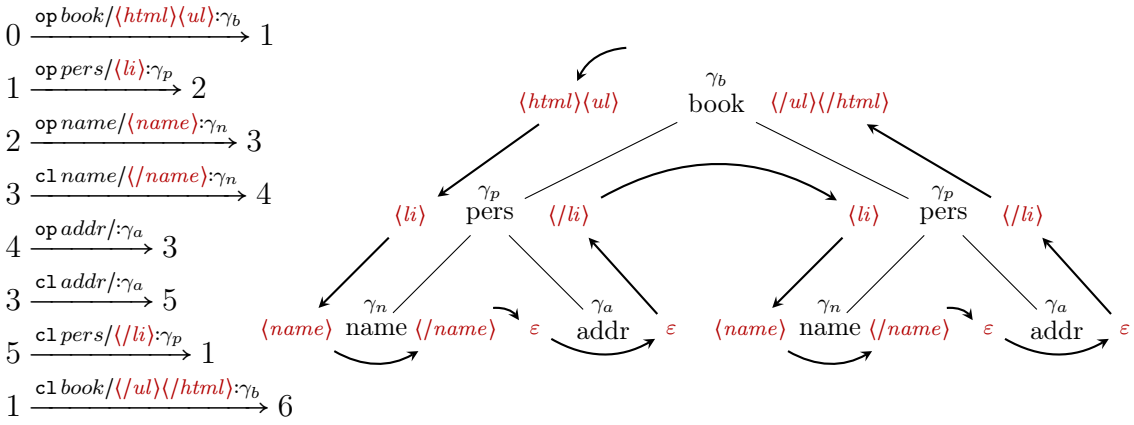


FIGURE 2.18 – Exemple de transduction de mots imbriqués en mots

Définition 41. *Un* transducteur de mots imbriqués en mots *ou* NW2W M *est un défini par un t-uplet d'ensembles finis* $M = (\Sigma, \Delta, Q, \Gamma, init, fin, rul)$ *qui défini une transduction d'un alphabet d'entrée* Σ*, à un alphabet de sortie* Δ*, utilisant des états* Q *comprenant un ensemble d'états initiaux init et d'états finaux dans fin, des symboles de piles* Γ *permettant de contrôler la bonne imbrication de l'entrée utilisés pour définir des règles* $Q \times \hat{\Sigma} \times \Gamma \times \Delta^* \times Q \in rul$*. Les règles sont notés* $q \xrightarrow{\alpha\, a/u:\gamma} q'$ *pour* $(q, (\alpha, a), \gamma, w, q') \in rul$*.*

Comme remarqué précédemment, notre définition n'autorisant pas les règles ayant ε pour entrée, un NW2W peut être vu comme un NWA où chaque sortie est égale à ε et peut servir à définir le domaine d'entrée d'un NW2W. Il partage avec les autres transducteurs la notion de partie gauche gch, d'une partie droite dt, et d'une production $prod$. Travaillant avec des mots imbriqués, l'entrée d'une règle r n'est plus un label mais une *action* $act(r)$, et elle contient en plus un symbole de pile $pile(r)$.

La sortie ne modifiant pas l'exécution du transducteur, la notion de déterminisme est identique à celle des NWAs. On note dNW2Ws les NW2Ws déterministes. L'exécution peut être décrite par un système de configurations identiques à celles des NWAs, auxquelles on ajoute la chaîne de sortie produite progressivement. Une configuration est définie sur $\hat{\Sigma} \times Q \times \Gamma^* \times \Delta^*$ de sorte qu'une étape de transduction se traduit par :

$$\frac{q \xrightarrow{-a/w':\gamma} q' \in rul}{(u, q, S, v) \to (u', q', S', w \cdot w')}$$

u' , q' et S' étant obtenus de la même manière que dans les NWAs.

Un mot $[\![M]\!]$ transforme un mot imbriqué u en un mot w s'il existe un état initial $q_0 \in init$ et une séquence d'instructions qui, à partir de $C = (u, q_0, \varepsilon, \varepsilon)$, nous amène à $C' = (\varepsilon, q_f, \varepsilon, w)$ tel que $q_f \in fin$. Dans ce cas, le couple (u, w) appartient à la relation $[\![M]\!]$. Nous pouvons considérer l'entrée comme la linéarisation d'un arbre non bornée et étendre la définition de $[\![M]\!]$ directement aux arbres que l'entrée représente. Pour tout arbre $t \in \mathcal{T}^u_\Sigma$ et mot $w \in \Delta^*$, $(t, w) = [\![M]\!]$ ssi $(\textsc{linear}(t), w) \in [\![M]\!]$. Si l'on se limite aux dNW2Ws, $[\![M]\!]$ peut être vu comme une fonction partielle.

À partir de cette définition, nous pouvons définir pour tout dNW2W M, et pour chacun de ses état q, une fonction partielle $[\![M]\!]_q : \mathcal{T}^*_\Sigma \to \Delta^*$ représentant la production possible et l'état atteint à partir d'un état du transducteur pour une forêt donnée en paramètre. Ainsi, $[\![M]\!]_q(t_1, \ldots, t_k) = (w, q')$ si et seulement si à partir d'une configuration $(\textsc{linear}(t_1, \ldots, t_k), q, \varepsilon, \varepsilon)$ on atteint la configuration $(\varepsilon, q', \varepsilon, w)$.

Nous pouvons aussi limiter les transducteurs à un parcours descendant, comme les dNWAs$^\downarrow$ pour les automates, en limitant les symboles de piles aux états. On appelle cette sous-classe les dNW2W$s^\downarrow$.

En se limitant aux dT2Ws sans copie et sans réordenancement on remarque que la relation qui les lient aux dNW2W$s^\downarrow$ est la même que la relation liant les dNWAs$^\downarrow$ et les dTA$^\downarrow s$. Plus formellement, les propositions 2 et 1 peuvent être réécrites de la manière suivante, les preuves marchant de la même manière que précédemment, en ajoutant de la production.

Proposition 14. *Tout* dT2W M *sans copie ni réordenancement peut être converti en un* dNW2W$^{\downarrow}$ N *tel que* $[\![N]\!] = [\![M]\!]$ *en temps* $O(|M| * n)$ *où* $n = \max(ar(x) \mid x \in \Sigma)$.

Preuve. (esquisse) Comme évoqué précédemment, cette preuve repose sur celle de la proposition 1, la possibilité de produire de M ne modifiant pas la structure des transducteurs mais les annotant d'une sortie pouvant être rattaché aux opérations d'ouvertures et fermetures de N. Hormis l'ajout d'un alphabet de sortie, et une modification des règles de la manière suivante, rien ne change de la construction de l'ancienne preuve.

$$\frac{q_0 \in init_M \qquad r = q_0(a) \to u_0 \cdot q_1 \cdot \ldots \cdot q_k \cdot u_k \in rul_M}{\mathtt{o} \xrightarrow{\mathtt{op}\, a/u_o:\mathtt{f}} \langle r, 0\rangle \qquad \langle r, k\rangle \xrightarrow{\mathtt{cl}\, a/\varepsilon:\mathtt{f}} \mathtt{f}}$$

$$\frac{r = q(a) \to u_0 \cdot q_1 \cdot \ldots \cdot q_j \cdot u_j \cdot \ldots \cdot q_k \cdot u_k \in rul_M \qquad r' = q_j(b) \to v_0 \cdot p_1 \ldots p_m \cdot v_m \quad 1 \le j \le k}{\langle r, j-1\rangle \xrightarrow{\mathtt{op}\, b/v_o:\langle r,j\rangle} \langle r', 0\rangle \qquad \langle r', m\rangle \xrightarrow{\mathtt{cl}\, b/u_j:\langle r,j\rangle} \langle r, j\rangle}$$

□

Proposition 15. *Pour tout* dNW2W$^{\downarrow}$ N *nous pouvons construire en temps* $O(|\Sigma|^2 * |Q_N|^2)$ *un* dT2Ws *sans copie et sans réordonnancement* M *tel que* $[\![M]\!] = \{(fcns(t), u) \mid (t, u) \in [\![N]\!]\}$.

Preuve. (esquisse) La preuve marche de la même manière que celle de la proposition 2, les règles du transducteur dT2W sont cette fois si générées de la manière suivante :

$$\frac{q_0 \in init_N \quad q_2 \in fin_N \quad q_0 \xrightarrow{\mathtt{op}\, a/w:q_2} q_1 \in rul_N}{q_0(a) \to w \cdot \langle q_1, a, q_2\rangle \cdot q_\#} \qquad \frac{q \xrightarrow{\mathtt{cl}\, a/w:q'} q' \in rul_N}{\langle q, a, q'\rangle(\#) \to w}$$

$$\frac{b \in \Sigma \quad p \in Q_N \quad q \xrightarrow{\mathtt{op}\, a/u:q_2} q_1 \in rul_N}{\langle q, b, p\rangle(a) \to u \cdot \langle q_1, a, q_2\rangle \cdot \langle q_2, b, p\rangle} \qquad \frac{true}{q_\#(\#) \to \varepsilon}$$

□

De ces deux propriétés nous obtenons directement le corollaire suivant.

Corollaire 3. *L'équivalence des* dNW2W$s^{\downarrow}$ *sans copie et sans réordonnancement peut être réduit en temps polynomial à l'équivalence des* dT2Ws *et vice versa.*

2.4.2 Transducteurs de mots imbriqués en mots imbriqués

Le modèle des transducteurs de flux d'arbres noté NW2NW, de mots imbriqués en mots imbriqués, a été proposé par Alur et D'Antoni (2012). Il permet de produire, à partir d'une linéarisation d'arbre, non plus un arbre mais une linéarisation d'arbres également. Dans cela, il s'approchent plus du modèle de transformation XSLT sur lequel on reviendra plus précisément. Il faut remarquer qu'ici, la définition de mots imbriqués diffère ce celle introduite dans cette thèse, le modèle y autorisant des symboles internes, correspondant aux possibles valeurs textuelles.

Ce modèle se limite à un unique passage sur la linéarisation, pouvant donc être vue comme un flux d'informations et calcul en temps linéaire la sortie, à l'aide d'un nombre fini d'états, d'une pile, en effectuant des opérations de concaténation et d'insertion dans les mots imbriqués dans la sortie.

Le principal intérêt de ce modèle vient de son expressivité fortement lié avec le modèle logique des transformations MSO-définissables comme nous le montre Alur et D'Antoni (2012). En effet, le théorème 11 de cet article montre que toute transformation de mots imbriqué définissable par un NW2NW est aussi MSO-définissable. Les observations faite à la suite des Propositions 12 et 13 montre que toute transformation MSO-définissable l'est aussi par un NW2NW si on considère les transformations MSO-définissables d'arbres binaires à arbres binaires.

Ce modèle a aussi de bonnes propriétés, telle que la clôture par composition, évoqué dans le théorème 7. Le théorème 18 montre que l'inclusion de l'image de sortie d'un NW2NW dans le langage, reconnu par un NWA, est décidable en temps polynomial.

L'équivalence de deux NW2NW est décidable, et comme le montre le Théorème 20 en temps NExpTime.

Chapitre 3

Transformations XML

Il est important de rappeler que la modélisation de transformations doit répondre avant tout à un besoin pratique. En effet, pouvoir formaliser la représentation, manipulation ou toute autre opération sur des données structurées, tel que le XML évoqué précédemment, est indispensable pour la gestion des informations transitants sur les différents outils informatiques. Nous allons présenter dans ce chapitre le langage XSLT, permettant de représenter et d'effectuer la transformation de données XML vers XML. Le choix d'XSLT repose sur le fait que ce langage de transformation utilise principalement la structure des données, ce qui nous intéresse ici. Il a de plus comme but de rester le plus général possible. Il ne se limite pas à une tâche précise, ou à un type de données particulier.

Après avoir brièvement présenté ce qu'est le langage XSLT, à travers sa syntaxe et quelques exemples pour en dévoiler les principaux mécanismes, nous montrerons le lien entre les transformations exprimables par XSLT et les transducteurs précédemment décrits. Pour cela, nous parlerons des divers fragments de XSLT étudiés par la communauté, le modèle global étant trop expressif pour pouvoir être comparé aux modèles théoriques classiques. Nous nous attarderons particulièrement sur les macro-transducteurs d'arbres et la manière dont on peut les exprimer dans le langage XSLT.

3.1 XSLT

Tout comme le XML, le langage XSLT, de l'anglais « eXtensible Stylesheet Language Transformations », est un langage défini au sein de la recommandation XSL du consortium W3C. Il a pour but premier de permettre la visualisation de fichiers XML sous format HTML en décrivant le passage de l'un à l'autre. Cependant, même si l'entrée doit être nécessairement du XML, il y a peu de contraintes sur la sortie, qui peut être au format XML, texte, etc...

3.1.1 XSLT en Pratique

XSLT est un langage de programmation permettant, à l'aide d'un analyseur adapté, de transformer un fichier XML. Il est lui même au format XML mais

avec certaines balises appartenant à l'espace de nommage de XSL préfixé par **xsl:** et représentant les diverses opérations à effectuer.

Prenons pour premier exemple le simple passage d'un fichier XML, contenant un répertoire, vers une page HTML, listant les noms des contacts qu'elle contient. Le fichier de base est le suivant :

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <repertoire>
3   <contact>
4     <nom>Laurence</nom>
5     <prenom>Gregoire</prenom>
6     <email>gregoire.laurence@inria.fr</email>
7     <tel>01.23.45.67.89</tel>
8   </contact>
9   <contact>
10    <nom>Doe</nom>
11    <prenom>John</prenom>
12    <email>John@doe.com</email>
13    <tel>02.46.80</tel>
14  </contact>
15 </repertoire>
```

Un fichier XSLT repose sur un système de *modèles*, contenus dans des noeuds **xsl:template** appliqués récursivement. La transformation s'effectue par une successions d'appels, de choix, et d'applications de modèles sur des noeuds choisis à l'aide de requêtes XPATH.

```
1 <xsl:template match="/">
2   <html>
3     <head>
4       <title>Liste de noms</title>
5     </head>
6     <body>
7       <ol>
8         <xsl:apply-templates select="repertoire/contact" />
9       </ol>
10    </body>
11  </html>
12 </xsl:template>
```

Le choix du modèle dépend de l'attribut **match** décrivant un noeud. Dans cet exemple, le noeud considéré est la racine de l'arbre ("/"). Le contenu d'un modèle est un mélange de XML classique et de balises XSL. La sortie est obtenue en remplaçant chaque balise XSL par le ou les résultats correspondants. Dans notre cas, le noeud $xsl : apply - templates$ va récupérer l'ensemble des noeuds correspondants à la requête XPATH contenue dans l'attribut *select*, pour appliquer récursivement à chaque noeud le modèle lui correspondant le mieux, dans l'ordre spécifié, par défaut l'ordre d'apparition des noeuds dans

l'arbre. Il est également possible d'ajouter un *mode* marchant comme les états d'un transducteur, permettant de diriger le choix du modèle à appliquer. Dans notre cas, le modèle suivant sera appliqué à chaque noeud *contact*, fils d'un noeud *repertoire*, lui même fils de l'élément courant : la racine. Le fichier XSLT étant avant tout un XML, toute portion de sortie contenant du XML doit être bien balancée, et chaque noeud ouvert dans un modèle doit se fermer dans celui ci.

```
1 <xsl:template match="child::contact">
2   <li>
3     <xsl:value-of select="child::nom/child::text()"/>
4     <xsl:value-of select="child::prenom/child::text()"/>
5   </li>
6 </xsl:template>
```

Pour chacun de ces contacts, nous avons décider de sortir une concaténation des champs textuels contenus dans le **nom** et **prenom**, et d'inclure le résultat dans un noeud *li* permettant son affichage en liste. La récupération de la valeur d'un noeud se fait à l'aide de l'opération **xsl: value-of**. Cette opération permet, non seulement, de récupérer les valeurs textuelles contenues dans un noeud, mais également d'effectuer d'autres opérations, allant du traitement de chaînes jusqu'à certaines opérations mathématiques ou logiques. Plus aucun autre appel de modèle n'étant fait, XSLT peut produire la sortie suivante au format HTML.

```
1 <html>
2   <head>
3     <title>Liste de noms</title>
4   </head>
5   <body>
6     <ol>
7       <li> Laurence  Gregoire </li>
8       <li> Doe John </li>
9     </ol>
10  </body>
11 </html>
```

Nous avons, pour ce premier exemple, choisi de garder une transformation séquentielle (l'ordre des noeuds n'étant pas modifié) et descendante (aucune requête XPATH ne remontant dans l'arbre). Nous avons également décidé de nous restreindre aux bases d'une transformation XSLT même si la concaténation de noeuds est déjà quelques chose d'inaccessibles pour la plupart des transducteurs vus précédemment.

Le langage XSLT est cependant beaucoup plus puissant. Il peut à tout moment accéder à toute partie de l'arbre d'entrée grâce aux requêtes XPATH utilisées pour choisir quel noeud copier ou, sur lesquels, appliquer un modèle.

Les noeuds peuvent donc être réordonnés et copier sans limite. Prenons, pour illustrer cela, le modèle suivant qui permet de générer, à partir d'un contact, un lien HTML vers son mail.

```
1 <xsl:template match="child::contact" mode = "lien">
2   <xsl:for-each select="child::email">
3     <a>
4       <xsl:attribute name="href">
5  mailto:
6  <xsl:copy-of select="parent::*/child::nom/text()"/>
7  <xsl:copy-of select="parent::*/child::prenom/text()"/>
8  &lt;<xsl:copy-of select="self::text()"/>&gt;
9       </xsl:attribute>
10      <xsl:copy-of select="self::text()"/>
11    </a>
12  </xsl:for-each>
13 </xsl:template>
```

Le lien obtenu est de la forme :

```
<a href="mailto:Laurence Gregoire <gregoire.laurence@inria.fr>">
        gregoire.laurence@inria.fr
</a>
```

Le noeud **xsl:for-each** permet d'appliquer le modèle qu'il contient à tous les noeuds obtenus par la requête sans avoir à passer par un autre modèle. Étant au niveau du noeud *email*, fils du noeud *contact*, rien ne nous empêche d'explorer ses frères, de concaténer leur contenu dans un nouvel attribut. Même en ajoutant cette liberté, rendant déjà les transformations impossibles à reproduire par la plupart des transducteurs que nous avons vus, nous sommes encore loin d'atteindre toute l'expressivité des XSLT, langage de programmation à part entière.

Le fait de pouvoir utiliser des variables et des paramètres enrichie fortement l'expressivité du modèle. L'utilisation de noeuds **xsl:variable** permet de mémoriser certains noeuds ou valeurs récupérées par une requête XPATH. Elle permet surtout de pouvoir définir une transformation XSLT qui pourra être par la suite utilisée comme une entrée de la transformation.

Pour illustrer ce point, reprenons l'exemple des liens dont nous allons stocker le résultat dans une variable globale, ce qui permet de récupérer ces mêmes liens plusieurs fois sans avoir besoin de les recalculer.

```
1 <xsl:variable name="liens">
2   <liens>
3     <xsl:for-each select="/descendant::contact">
4       <lien>
5  <adresse>
```

```
6     <xsl:value-of select="child::email/text()"/>
7           </adresse>
8           <xsl:for-each select="child::email">
9     <a>
10       ...
11             </a>
12    </xsl:for-each>
13        </lien>
14     </xsl:for-each>
15   </liens>
16 </xsl:variable>
```

Nous définissons donc une variable contenant un XML créé par une suite d'opération XSLT, associant à chaque email du document d'entrée, le lien correspondant. Le coeur de cette variable fonctionne exactement de la même manière que le traitement présenté précédemment. Il suffit de créer une requête XPATH venant chercher le lien existant dans cette variable, et il n'est plus nécessaire de le recréer :

```
1 <xsl:template select="child::email" mode="lien">
2   <xsl:variable name="email" select="text()"/>
3   <xsl:copy-of
4       select="$liens/descendant::a[parent::lien/adresse = $email]"
5       />
6 </xsl:template>
```

La principale propriété découlant de l'utilisation de variables est le fait que les transformations décrites par le langage XSLT sont closes par composition. Dans notre exemple, un fichier XML à part entière est créé et réutilisé pour une autre transformation au sein d'un même XSLT.

3.2 Transducteurs et XSLT

XSLT travaille avec des arbres d'arité non bornée, permet la concaténation de noeuds, la copie non bornée, le réordonnancement des noeuds, de se déplacer dans l'arbre d'entrée comme bon lui semble, et de composer plusieurs transformations XSLT au sein d'un même fichier de transformation. Ceci rend le modèle plus expressif que tout les modèles vus précédemment. Pour s'en persuader nous allons maintenant traduire différents modèles de transductions en XSLT.

3.2.1 *d*ST2W

Pour débuter intéressons nous aux *d*ST2W*s*, modèle de transduction qui nous intéresse le plus. Rappelons que c'est un transducteur d'arbres en mots

déterministe descendant, et séquentiel (i.e. sans copie ni réordenancement).

Pour obtenir un XSLT exprimant la transformation d'un dST2W M, il suffit de traduire chacune des règles $q(f(x_1,\ldots,x_k)) \to u_0 \cdot \langle q_1, x_1\rangle \cdot \ldots \cdot \langle q_k, x_k\rangle \cdot u_k$ de M par le modèle suivant :

```
<xsl:template mode="q" match="f">
  u_1
  <xsl:apply-templates
     mode="q_1" select="child::*[position()=1]" />
  u_2
  ...
  <xsl:apply-templates
     mode="q_k" select="child::*[position()=k]" />
  u_k
</xsl:template>
```

Le modèle à appliquer sur un noeud est bien celui correspondant à la règle ; le choix de ce modèle dépendant du mode égal à l'état et de la racine du sous arbre à traiter.

Le modèle initial, appelé à la racine de l'arbre, correspond exactement à l'axiome initial du transducteur. Ainsi, pour un axiome $init = u_0 \cdot q(x_0) \cdot u_1$ nous obtenons le modèle suivant :

```
<xsl:template match="/">
  u_0
  <xsl:apply-templates mode="q_0" select="child::*" />
  u_1
</xsl:template>
```

Pour ce qui est des autres modèles descendants classiques, les seuls changements résident dans la partie droite de la règle définissant la sortie, ce qui change uniquement le contenu des modèles du fichier XSLT.

3.2.2 Macro-Transducteurs et XPATH

Rappelons que les macro-transducteurs d'arbres enrichissent les transducteurs d'arbres descendants en ajoutant, aux règles classiques d'un transducteur d'arbres descendant, un ensemble de paramètres $Y_m = \{y_1,\ldots,y_m\}$ permettant de transmettre, à travers des variables, une sortie pré-construite pour une future utilisation. Nous obtenons donc des règles de la forme suivante :

$$q(f(x_1,\ldots,x_k)) : (y_1,\ldots,y_m) \to t$$

où k est l'arité de q, m celle de l'état q et t le contexte de sortie défini sur $T_{(\Delta\cup(Q\times X_k))}(Y_m)$.

Il est possible, à partir d'un macro-transducteur, de définir un XSLT décrivant la même transformation. Le mécanisme de paramètres étant très proche

des variables utilisées ici. En effet, un modèle (template) peut être vu comme un état q d'un macro-transducteur. Pour cela, il suffit d'autoriser uniquement, à part pour le traitement de la racine, des modèles ayant un *mode* appartenant à Q et une sélection (*match*) ne spécifiant que le noeud traité par un symbole dans Σ, sans autre information sur son voisinage. À partir de là, chaque modèle correspond au plus à une règle du macro-transducteur dont le contenu est obtenu par le membre droit de cette règle. Pour cela, il suffit de représenter, au format XML, l'arbre contenu dans la partie droite en remplaçant tout appel à état $\langle q, x_1\rangle(y_1, \ldots, y_k)$ par une application de modèle de la forme suivante :

```
<xsl:apply-templates mode="q" select="child::*[position()=i]">
  <xsl:with-param name="y1">
    t_1
  </xsl:with-param>
  ...
  <xsl:with-param name="yk">
    t_k
  </xsl:with-param>
  t
</xsl:apply-templates>
```

Exemple 37. *Pour mieux visualiser ce passage, reprenons l'exemple 34 permettant de transformer un arbre dans sa linéarisation et définissons le fichier* XSLT *correspondant (dans la figure 3.1).*

$M_{34} = (\Sigma, \Delta, \{q_0, q\}, \{q_0\}, rul)$ *avec pour règles :*

$$\begin{array}{ll} q_0(x_0) & \to \langle q, x_0\rangle(\#) \\ q(a) : (y_1) & \to \langle a\rangle(\langle /a\rangle(y_1)) \\ q(f(x_1, x_2)) : (y_1) & \to \langle f\rangle(\langle q, x_1\rangle(\langle q, x_2\rangle(\langle /f\rangle(y_1)))) \\ q(b(x_1)) : (y_1) & \to \langle b\rangle(\langle q, x_1\rangle(\langle /b\rangle(y_1))) \end{array}$$

Une fois de plus ce passage se fait en restreignant XSLT à un parcours descendant. Une transformation XSLT pouvant à tout moment accéder à l'intégralité de l'arbre à transformer.

Le langage XSLT peut donc facilement représenter toutes les familles de transducteurs évoquées précédemment, et bien plus encore. Le premier point sur lequel il se distingue est l'utilisation d'arbres de données en entrée, et non juste des arbres comme le font les différentes familles de transducteurs présentés. Les opérations de jointures rendues impossible en s'intéressant uniquement à la structure de la donnée redeviennent possibles.

De plus, le système de variables de XSLT ne se limite pas à l'utilisation qu'en fait les macro-transducteurs à savoir le stockage de fragments de sorties, chaque variable pouvant être utilisé comme une entrée classique. Cela

permet, entre autre, la gestion de multiples fichiers de manière asynchrone et la composition directe de transductions au sein d'un même XSLT.

Des modèles tels que les macro-transducteurs *avec anticipation régulière* essayent de pallier cette différence imposante en s'enrichissant de requêtes XPATH. Ce sont des macro-transducteurs classiques pour lesquels chaque choix de transduction (règles à utiliser) dépend non seulement de l'état ainsi que le noeud dans lequel on se trouve, mais aussi de l'intégralité du domaine, accessible par des requêtes XPATH. Cela permet uniquement de s'abstraire de l'ordre du parcours en permettant d'avoir un regard sur l'intégralité du domaine, mais une fois de plus ne permet pas la liberté donnée par le langage XSLT pouvant récupérer à tout moment tout sous-arbre de l'entrée pour le manipuler.

Comme nous pouvons le rappeler, XSLT est un programme Turing-complet Kepser (2004); Onder et Bayram (2006), et ne peut donc être modélisé par des modèles théoriques disposant de propriétés nécessaires sur la décidabilité de problèmes classiques. Il existe cependant des langages tels que X-FUN (Labath et Niehren, 2013), langage fonctionnel de transformations d'arbres de données XML en forêt d'arbres de données, qui permettent d'exprimer un fragment des transformations XSLT ayant de meilleures propriétés théoriques. Sans entrer dans les détails, X-FUN fourni un langage de transformations équivalents au macro-transducteurs avec anticipation régulière.

Il existe donc plusieurs solutions déjà implémentées pour permettre d'exprimer la transformation de langages, mais qui en cherchant à rester le plus expressifs tombent dans des complexités trop fortes,XSLT étant Turing-complet. D'autres langages de transformations peuvent être rattachés à des modélisations théoriques, comme les macro-transducteurs. Mais ces modèles ne disposent que de peu de résultats théoriques et nécessitent des travaux préliminaires sur des modèles plus simples tel que les transducteurs d'arbres en mots.

```
<xsl:stylesheet>
  <xsl:template match="/">
    <xsl:apply-templates mode="q" select="f">
      <xsl:with-param name="y1">
        <end/>
      </xsl:with-param>
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template mode="q" match="a">
    <xsl:param name="y1"/>
    <op-a>
      <cl-a>
<xsl:copy-of select="$y1"/>
      </cl-a>
    </op-a>
  </xsl:template>

  <xsl:template mode="q" match="f">
    <xsl:param name="y1"/>
    <op-f>
      <xsl:apply-templates mode="q" select="*[position()=1]">
        <xsl:with-param name="y1">
          <xsl:apply-templates mode="q" select="*[2]">
            <xsl:with-param name="y1">
        <cl-f>
  <xsl:copy-of select="$y1"/>
        </cl-f>
      </xsl:with-param>
    </xsl:apply-templates>
          </xsl:with-param>
      </xsl:apply-templates>
    </op-f>
  </xsl:template>

  <xsl:template mode="q" match="b">
    <xsl:param name="y1"/>
    <op-b>
      <xsl:apply-templates mode="q" select="child::*[1]">
<xsl:with-param name="y1">
  <cl-b>
    <xsl:copy-of select="$y1"/>
  </cl-b>
          </xsl:with-param>
      </xsl:apply-templates>
    </op-b>
  </xsl:template>
</xsl:stylesheet>
```

FIGURE 3.1 – Macro-transducteur de l'exemple 34 en XSLT

Chapitre 4

Équivalence de transducteurs séquentiels d'arbres en mots

Le précédent chapitre nous a permis d'identifier plusieurs familles de transduction, traitant des arbres, sous différentes formes (arbres d'arité bornée, encodage et mots imbriqués) pour produire des mots ou encore des arbres. Malgré cette diversité de données, les problèmes rencontrés témoignent d'une certaine similitude ; parfois directe comme l'illustre la Proposition 15 et la Proposition 14 réduisant un modèle dans un autre. Parfois cela ce fait de manière plus implicite, comme nous allons le voir maintenant.

Dans ce chapitre nous avons décider de nous intéresser au problème d'équivalence qui, au delà de son utilité, sera ici un outil permettant de faire apparaître les liens existants entres ces différentes classes. Nous en profiterons pour apporter des bornes de complexité polynomiale pour le problème d'équivalence uniquement connu décidable pour certaines de ces classes.

4.1 Relation avec l'équivalence de morphismes sur CFGs

Une étape de ce chapitre repose sur la réduction du problème d'équivalence des $d\text{NW2W}s^{\downarrow}$ à celui des morphismes (et inversement). il est nécessaire d'adopter des formalismes permettant de mettre en relation l'exécution de $d\text{NW2W}s^{\downarrow}$ Pour réduire le problème d'équivalence des $d\text{NW2W}s^{\downarrow}$ à celui des morphismes (et inversement) nous allons nous baser sur les exécutions parallèles, d'une manière proche de celle déjà utilisé précédemment sur les transducteurs de mots (cf. lemme 19). Nous avons vu comment représenter l'exécution d'un $d\text{NW2W}$ par une suite de configurations. Pour pouvoir représenter des exécutions parallèles, il est nécessaire de définir ces exécutions en se basant cette fois sur les événements d'ouvertures et de fermetures de noeuds d'un arbre.

4.1.1 Exécution d'un dNW2W

Un *événement* associé à un arbre est l'ouverture (ou la fermeture) d'un noeud de cet arbre. L'opération d'ouverture `op` (ou de fermeture `cl`) n'est plus associée à une étiquette de l'arbre, comme dans $\hat{\Sigma}$ pour les mots imbriqués, mais directement au noeud d'un arbre t, i.e. appartenant à $noeuds(t)$. On note $\widehat{noeuds}(t)$ l'ensemble des événements d'un arbre t . Cet ensemble est totalement ordonné, débutant par l'ouverture de la racine de l'arbre $(\texttt{op}, \varepsilon)$ et se terminant par la clôture de ce même noeud $(\texttt{cl}, \varepsilon)$. L'ordre $(\alpha, \pi) < (\alpha', \pi')$ est vérifié si (α, π) est un événement précédent à (α', π') dans le parcours en profondeur de l'arbre. Nous dénotons particulièrement l'événement « prédécesseur direct » d'un événement par $pred(\alpha, \pi) \in \widehat{noeuds}(t)$.

Dès lors, l'*exécution* d'un dNW2W sur un arbre $t \in \mathcal{T}^u_\Sigma$ est représentée, à l'instar de celle d'un transducteur de mots, par un couple $t * \beta$ où $\beta \subseteq \widehat{noeuds}(t) \to rul$.

Elle est dite *valide* si $gch(\beta((\texttt{op}, \varepsilon))) \in init$ et pour tout $(\alpha, \pi) \in \widehat{noeuds}(t)$, $pile(\beta(\texttt{op}, \pi)) = pile(\beta(\texttt{cl}, \pi))$, $dt(\beta(pred(\alpha, \pi))) = gch(\beta((\alpha, \pi)))$, et $act(\beta(\alpha, \pi)) = (\alpha, a)$ où "a" est l'étiquette du noeud π de t. Elle est *complète* si elle est valide et que $dt(\beta((\texttt{op}, \varepsilon))) \in fin$.

Lemme 18. *$[\![M]\!] = \{(t, prod(\beta(e_1)) \cdot \ldots \cdot prod(\beta(e_n))) \mid t \in \mathcal{T}^u_\Sigma$, $t * \beta$ est une exécution valide d'un dNW2W T sur t et les événements de t sont $e_1 < \ldots < e_n\}$.*

4.1.2 Arbre syntaxique étendu

Un *arbre syntaxique* est un arbre permettant de représenter la syntaxe d'un mot reconnu par une *CFG*. Chaque noeud interne est un symbole non terminal de cette grammaire, et chaque feuille un symbole terminal. Un *arbre syntaxique étendu* est un arbre dont les noeuds internes ne sont plus composés de symboles non terminaux mais directement des règles utilisées à ces endroits. Formellement, un arbre syntaxique étendu pour une *CFG* G est un arbre d'arité bornée $t \in \mathcal{T}_{rul \cup \Sigma}$ tel que tout noeud $\pi \in noeuds(t)$ respecte les trois contraintes suivantes :

1. chaque noeud interne est étiqueté par une règle et chaque feuille par un symbole terminal ;
2. Un noeud étiqueté par une règle $q \to q_1, \ldots, q_n$ est d'arité n et pour $1 \leq i \leq n$, le fils $\pi \cdot i$ est étiqueté par une règle ayant pour partie gauche q_i ;
3. Un noeud étiqueté par une règle $q \to a$ est d'arité 1 et a comme unique fils une feuille étiquetée par "a".

On appelle *feuillage* d'un arbre t, donné par la fonction $\mathit{feuilles}(t)$, la concaténation des labels de chaque feuille de l'arbre t prise de gauche à droite. Le feuillage d'un arbre syntaxique, étendu ou non, appartient au langage de la grammaire correspondante, i.e. $w \in \mathcal{L}(G)$, si et seulement s'il existe un arbre syntaxique (étendu) t pour la *CFG* G tel que $\mathit{feuilles}(t) = w$.

Lemme 19. *Pour tout morphisme $\mu : \Sigma^* \to \Delta^*$ et CFG G ayant pour alphabet Σ, nous pouvons construire, en temps $O(|\mu| + |G|^2)$, un* dNW2W$^{\downarrow}$ *M, sur les alphabets Σ et Δ, tel que :*

$$[\![M]\!] = \{(t, \mu(\mathit{feuilles}(t))) \mid t \text{ arbre syntaxique étendu de } G\}$$

Preuve. Pour un morphisme $\mu : \Sigma^* \to \Delta^*$ et une *CFG* $G = (\Sigma, Q_G, \mathit{init}_G, \mathit{rul}_G)$, on cherche à construire le dNW2W $^{\downarrow}M = (\Sigma', \Delta, Q_M, \Gamma_M, \mathit{init}_M, \mathit{rul}_M, \mathit{fin}_M)$ prenant en entrée les arbres syntaxiques étendus de G et en sortie le morphisme appliqué aux feuillages de ces arbres.

Les arbres syntaxiques étendus contiennent, grâce aux noeuds internes étiquetés par les règles de la grammaire, toute l'information nécessaire à un parcours déterministe descendant.

Le transducteur que nous construisons est défini sur l'alphabet d'entrée $\Sigma' = Q_G \cup \Sigma$. L'ensemble de ses états et symboles de piles sont composés de paires de règles et d'indices permettant de se situer dans l'évaluation de la grammaire $Q_M = \Gamma_M = \{\langle r, j\rangle \mid r \in \mathit{rul}_G,\ 0 \le j \le |r|\} \cup \{\mathtt{o}, \mathtt{d}, \mathtt{f}\}$, ainsi qu'un état initial $\mathit{init}_M = \{\mathtt{o}\}$, final $\mathit{fin}_M = \{\mathtt{f}\}$ et transitoire $\mathtt{d}$. Les règles, quant à elles, sont produites de la manière suivante :

$$\frac{r \in \mathit{rul}_G \quad \mathit{gch}(r) \in \mathit{init}_G}{\begin{array}{c}\mathtt{o} \xrightarrow{\mathtt{op}\, r/\varepsilon:\mathtt{f}} \langle r, 0\rangle \\ \langle r, |r|\rangle \xrightarrow{\mathtt{cl}\, r/\varepsilon:\mathtt{f}} \mathtt{f}\end{array}} \qquad \frac{r \in \mathit{rul}_G \quad \mathit{dt}(r) = a}{\begin{array}{c}\langle r, 0\rangle \xrightarrow{\mathtt{op}\, a/\mu(a):\langle r,1\rangle} \mathtt{d} \\ \mathtt{d} \xrightarrow{\mathtt{cl}\, a/\varepsilon:\langle r,1\rangle} \langle r, 1\rangle\end{array}}$$

$$\frac{r, r' \in \mathit{rul}_G \quad \mathit{dt}(r) = q_1 \cdots q_k \quad 1 \le j \le |r| \quad \mathit{gch}(r') = q_j}{\begin{array}{c}\langle r, j-1\rangle \xrightarrow{\mathtt{op}\, r'/\varepsilon:\langle r,j\rangle} \langle r', 0\rangle \\ \langle r', |r'|\rangle \xrightarrow{\mathtt{cl}\, r'/\varepsilon:\langle r,j\rangle} \langle r, j\rangle\end{array}}$$

Il nous reste à prouver que $\forall r \in \mathit{rul}_G$, $\forall t = r(t_1, \ldots, t_k) \in \mathcal{T}_{\mathit{rul}\cup\Sigma}$, et $\forall w \in \Delta^*$, nous avons $[\![M]\!]_{\langle r,0\rangle}(t_1, \ldots, t_k) = (w, \langle r, k\rangle)$ si et seulement s'il existe une règle $r = q \to q_1 \ldots q_k \in \mathit{rul}_G$, où t et t_i sont respectivement les arbres syntaxiques étendus de G_q etG_{q_i}, et $\mu(\mathit{feuilles}(t_1) \cdot \ldots \cdot \mathit{feuilles}(t_k)) = w$. On prouve cette proposition par induction sur la taille des forêts prises en paramètres.

Pour les règles concernant les feuilles, $r = q \to a \in \mathit{rul}_G$, nous avons par construction $[\![M]\!]_{\langle r,0\rangle}(a) = (\mu(a), \langle r, 1\rangle)$.

Considérons maintenant une forêt $h = (t_1, \dots, t_m)$ de n noeuds et une règle $r = q \to q_1 \dots q_j \dots q_m \in rul_G$ tels que t_j est un arbre syntaxique étendu de G_{q_j} pour tout $j \in [1, m]$. Supposons que la proposition précédente est vérifiée pour des forêts contenants au plus $n-1$ noeuds. Cela implique que $[\![M]\!]_{\langle,0\rangle}(t_j^1, \dots, t_j^k) = (\mu(feuilles(t_j)), \langle r', k\rangle)$ avec $t_j = r'(t_j^1, \dots, t_j^k)$. De plus, l'existence des règles $\langle r, j-1\rangle \xrightarrow{\mathtt{op}\, r'/\varepsilon:\langle r,j\rangle} \langle r', 0\rangle$ et $\langle r', k\rangle \xrightarrow{\mathtt{cl}\, r'/\varepsilon:\langle r,j\rangle} \langle r, j\rangle$ dans rul_M implique que $[\![M]\!]_{\langle r,j-1\rangle}(t_j) = \mu(feuilles(t_j))$ pour tout $j \in [1, m]$. Par conséquent, $[\![M]\!]_{\langle r,0\rangle}(t_1, \dots, t_m)$ est la concaténation de μ appliquée aux feuillage des sous arbres $\mu(feuilles(t_1)) \cdot \dots \cdot \mu(feuilles(t_m))$.

La propriété étant maintenant vérifiée, il suffit d'observer le fait que, pour chaque arbre syntaxique étendu de G $r(t_1, \dots, t_n)$, il existe une règle $r \in rul_G$ telle que $gch(r) \in init_G$. Par définition, pour chaque règle de ce type dans G, il existe $\mathtt{o} \xrightarrow{\mathtt{op}\, r/\varepsilon:\mathtt{f}} \langle r, 0\rangle$ et $\langle r, n\rangle \xrightarrow{\mathtt{cl}\, r/\varepsilon:\mathtt{f}} \mathtt{f}$ dans rul_M ce qui nous donne :

$$[\![M]\!] = \{(t, \mu(feuilles(t))) \mid t \text{ un arbre syntaxique étendu de } G\}$$

□

On en déduit directement la complexité de la réduction du problème d'équivalence des morphismes sur une *CFG* à celui des dNW2W$s^{\downarrow}$

Proposition 16. *Le problème d'équivalence de morphismes sur une CFG peut être réduit en temps quadratique à celui des* dNW2W$s^{\downarrow}$.

Preuve. Prenons deux morphismes $\mu_1, \mu_2 : \Sigma^* \to \Delta^*$, une *CFG* G, et M_1, M_2 deux dNW2W s$^{\downarrow}$obtenus par la construction définie pour le Lemme 19 respectivement pour μ_1 et μ_2 sur G. Cette construction se fait en temps et taille $O(|G|^2 + |\mu_1| + |\mu_2|)$. M_1 et M_2 partageant un même domaine, dépendant de G, il est clair que M_1 est équivalent à M_2 si et seulement μ_1 et μ_2 le sont sur G. □

Nous souhaitons maintenant réduire le problème d'équivalence des NW2Ws à celui des morphismes sur une *CFG*. Comme précédemment évoqué nous allons nous baser sur les constructions utilisées pour le Lemme 10.

Une *exécution parallèle complète* pour deux NW2Ws M_1 et M_2, partageant un même alphabet d'entrée Σ et de sortie Δ, appartient à l'ensemble $\{t * \beta_1 * \beta_2 \mid t \in dom(M_1) \cap dom(M_2)\}$ avec $t * \beta_i$: une exécution complète de M_1. Une exécution parallèle peut être représentée par un mot sur $rul_{M_1} \times rul_{M_2}$, les règles contenant toute l'information sur l'entrée nous permettant de retrouver t. Pour cela, il suffit d'appliquer parallèlement β_1 et β_2 sur la linéarisation de t.

Lemme 20. *Pour tout* NW2W*s* M_1 *et* M_2*, partageant un même alphabet, il existe une CFG* G *telle que* $\mathcal{L}(G)$ *représente l'ensemble de toutes les exécutions parallèlles complètes.*

Preuve. On construit la grammaire $G = (\mathcal{R}, Q_G, init_G, rul_G)$ représentant les exécutions parallèles complètes de la manière suivante. L'ensembles des non terminaux $Q_G = \{\mathtt{o}\} \cup Q_{M_1}^2 \times Q_{M_2}^2$. Un symbole non terminal $\langle (p_1, q_1), (p_2, q_2) \rangle$ correspond à une exécution parallèle de M_1 de p_1 à q_1 et de de M_2 de p_2 à q_2 (sur une même entrée). La grammaire n'accepte qu'un symbole initial $\mathtt{o}$ ($init_G = \{\mathtt{o}\}$) et les règles sont construites, à partir des deux NW2W*s* M_1 et M_2, par :

$$\frac{\begin{array}{cc} q_1 \in init_{M_1} & p_1 \in fin_{M_1} \\ q_2 \in init_{M_2} & p_2 \in fin_{M_2} \end{array}}{\mathtt{o} \to \langle (q_1, p_1), (q_2, p_2) \rangle} \qquad \frac{q_1 \in Q_{M_1},\ q_2 \in Q_{M_2}}{\langle (q_1, q_1), (q_2, q_2) \rangle \to \varepsilon}$$

$$\frac{\begin{array}{ccc} r_1, r_1' \in rul_{M_1} & r_1 = p_1 \xrightarrow{\mathtt{op}\, a/w_1:\gamma_1} q_1 & r_1' = p_1' \xrightarrow{\mathtt{cl}\, a/w_1':\gamma_1} q_1', \\ r_2, r_2' \in rul_{M_2} & r_2 = p_2 \xrightarrow{\mathtt{op}\, a/w_2:\gamma_2} q_2 & r_2' = p_2' \xrightarrow{\mathtt{cl}\, a/w_2':\gamma_2} q_2' \end{array}}{\langle (p_1, q_1'), (p_2, q_2') \rangle \to (r_1, r_2) \cdot \langle (q_1, p_1'), (q_2, p_2') \rangle \cdot (r_1', r_2')}$$

$$\frac{p_1, p_1', q_1 \in Q_{M_1} \quad p_2, p_2', q_2 \in Q_{M_2}}{\langle (p_1, q_1), (p_2, q_2) \rangle \to \langle (p_1, p_1'), (p_2, p_2') \rangle \cdot \langle (p_1', q_1), (p_2', q_2) \rangle}$$

Pour prouver que cette construction reconnait bien toutes les exécutions parallèles complètes de M_1 et M_2, il faut étendre cette construction aux exécutions parallèles sur des linéarisations de forêts et pas uniquement sur des arbres.

Dans un premier temps, on prouve par induction que $\forall r_1 \dots r_m \in rul_{M_1}$, $\forall r_1' \dots r_m' \in rul_{M_2}$, $s = (r_1, r_1') \cdot \dots \cdot (r_m, r_m') \in \mathcal{L}(G_{((q,p)(q',p'))})$ ssi s est la représentation des exécutions parallèles complètes sur la forêt h, avec $gch(r_1) = q$, $gch(r_1') = q'$, $dt(r_m) = p$ et $dt(r_m') = p'$

Si la forêt ne contient aucun arbre, la règle de construction nous dit que $((q, q)(q', q')) \to \varepsilon$ est dans l'ensemble des règles de G pour tout état $q \in Q_{M_1}$ et $q' \in Q_{M_2}$. On suppose que l'hypothèse d'induction est vérifiée pour toute forêt contenant au plus n noeuds. Prenons une forêt de taille $n + 1$:

1. La forêt est un arbre, $h = a(h')$: Si il existe une exécution parallèle valide $s = (r_1, r_1') \cdot (r_2, r_2') \cdot \dots \cdot (r_{k-1}, r_{k-1}') \cdot (r_k, r_k')$ pour cet arbre, elle implique l'existence des règles $r_1 = q_1 \xrightarrow{\mathtt{op}\, a/w_1:\gamma_1} q_2$, $r_k = q_k \xrightarrow{\mathtt{cl}\, a/w_k:\gamma_1} p$, $r_1' = q_1' \xrightarrow{\mathtt{op}\, a/u_1':\gamma_1'} q_2'$, $r_k' = q_k' \xrightarrow{\mathtt{cl}\, a/u_k':\gamma_1'} p'$ pour $q_2 = gch(r_2)$, $q_2' = gch(r_2')$, $q_k = dt(r_k)$, et $q_k' = dt(r_k')$. Par l'hypothèse d'induction, la trace d'exécution $(r_2, r_2') \cdot \dots \cdot (r_{k-1}, r_{k-1}')$ est dans $\mathcal{L}(G_{((q_2,q_k),(q_2',q_k'))})$. Nous pouvons en

déduire que $\langle (q_1,p),(q'_1,p')\rangle \to (r_1,r'_1)\cdot\langle (q_2,q_k),(q'_2,q'_k)\rangle\cdot(r_k,r'_k)$ et donc que $s \in \mathcal{L}(G_{\langle (q_1,p),(q'_1,p')\rangle})$.

2. La forêt contient plus d'un arbre,$h = (t_1,t_2,\ldots,t_n)$ where $n \geq 2$:
   Considérons une exécution parallèle complète $s = (r_1,r'_1)\cdot\ldots\cdot(r_i,r'_i)\cdot(r_{i+1},r'_{i+1})\cdot\ldots\cdot(r_k,r'_k)$ sur cette forêt. Elle peut être divisée en deux sous exécutions respectivement sur t_1 et $h' = (t_2,\ldots,t_n)$. Il existe donc un i tel que $s' = (r_1,r'_1)\cdot\ldots\cdot(r_i,r'_i)$ est défini sur t_1 et que $s'' = (r_{i+1},r'_{i+1})\cdot\ldots\cdot(r_k,r'_k)$ est défini sur le reste de la forêt. Si nous fixons $gch(r_1) = q$, $gch(r'_1) = q'$, $dt(r_i) = gch(r_{i+1}) = q_i$, $dt(r_i) = gch(r_{i+1}) = q'_i$, $dt(r_k) = p$, et $dt(r'_k) = p'$, l'hypothèse d'induction nous dit que que $s' \in \mathcal{L}(G_{\langle (q,q_i),(q',q'_i)\rangle})$ et $s'' \in \mathcal{L}(G_{\langle (q_i,p),(q'_i,p')\rangle})$. De plus, d'après les règles de construction de G, $\langle (q,p),(q',p')\rangle \to \langle (q,q_i),(q',q'_i)\rangle\cdot(q_i,p)(q'_i,p')$, ce qui implique que $s = s'\cdot s'' \in L(G_{\langle (q,p),(q',p')\rangle})$.

Une fois cette hypothèse vérifiée, il suffit d'observer que toute exécution parallèle complète $t*\beta_1*\beta_2$, sur M_1 et M_2, repose sur l'existence des exécutions $t*\beta_1$ et $t*\beta_2$ sur ces deux transducteurs. Ces deux exécutions commencent respectivement aux états initiaux $q \in init_{T_1}$, $q' \in init_{T_2}$ et aboutissent aux états finaux $p \in fin_{T_1}$ and $p' \in fin_{T_2}$. Par construction, G contient la règle $\circ \to \langle (q,p),(q',p')\rangle$ et par induction nous avons prouvé que l'exécution parallèle complète s sur t appartient à $\mathcal{L}(G_{\langle (q,p),(q',p')\rangle})$.

□

Pour retrouver la sortie produite par une exécution parallèle nous pouvons définir deux morphismes μ_1 et μ_2 de rul à Δ tels que :

$$\mu_i((r_1,r_2)) = u \quad \text{if } r_i = q \xrightarrow{\alpha\, a/u:\gamma} q'$$

Si $t*\beta_1*\beta_2$ est une exécution parallèle compléte de M_1 et M_2 alors $(t,\mu_i(u)) \in [\![M_i]\!]$.

Proposition 17. *L'équivalence de d*NW2W*s peut être réduite à celle de morphismes sur une CFG.*

Preuve. Prenons deux *d*NW2W*s* M_1 et M_2 définis sur de mêmes alphabets d'entrée Σ et de sortie Δ. Décider de leur équivalence passe tout d'abord par vérifier qu'ils partagent un même domaine. Cela revient à tester l'équivalence de deux *d*NWAs, obtenus par suppression des productions respectives de M_1 et M_2, ce qui ce fait en temps polynomial comme le montre le Corollaire 1.

Si M_1 et M_2 partagent un même domaine, pour prouver leur équivalence, il faut tester l'équivalence des morphismes μ_1 et μ_2, introduits précédemment, sur la *CFG* G obtenue en suivant la méthode décrite dans le Lemme 20. □

En passant par l'équivalence des morphismes sur une grammaire, i.e. en composant la Proposition 17 et le Lemme 19, nous pouvons réduire le problème d'équivalence des dNW2Ws à celui des dNW2W$s^{\downarrow}$. L'inverse est trivial.

Corollaire 4. *Le problème d'équivalence des dNW2Ws peut être réduit à celui des dNW2W$s^{\downarrow}$ en temps polynomial, et vice versa.*

4.2 Relation entre dB2W et dT2W

Comme nous avons pu le remarquer précédemment, la syntaxe des dT2Ws est très proche de celle des dB2Ws. En adaptant la notion d'exécution parallèle à ces modèles, nous verrons comment réduire le problème d'équivalence de dT2Ws à celui des dB2Ws.

L'*exécution parallèle complète* représentée par un triplet $t * \beta_1 * \beta_2$, est définie pour deux dT2Ws M_1 et M_2 si et seulement si $t * \beta_1$ et $t * \beta_2$ sont les exécutions complètes respectives de M_1 et M_2. Dans notre cas, on limitera M_1 et M_2 à des dT2Ws ayant pour initialisation un unique état. Cela assure que toute la sortie est présente dans les règles de la représentation de l'exécution définie sur $\mathcal{T}_{(rul_{M_1} \times rul_{M_2})}$.

Pour ce qui est des dB2Ws, on appelle *exécution parallèle complète* pour deux dB2Ws M_1 et M_2 sur t, le triplet $t * \beta_1 * \beta_2$ tel que $t * \beta_1$ et $t * \beta_2$ sont les exécutions virtuelles totales de M_1 et M_2.

Nous pouvons étendre la définition des fonction *entree* et *sortie*, préalablement introduites sur les exécutions, aux exécutions parallèle.

Lemme 21. *Soit M_1 et M_2 deux dT2Ws (resp. dB2Ws) sur les alphabets Σ et Δ. Pour tout arbre $t \in \mathcal{T}_\Sigma$ et tout mots $w_1, w_2 \in \Delta^*$, nous avons $(t, w_1) \in [\![M_1]\!]$ et $(t, w_2) \in [\![M_2]\!]$ ssi il existe une exécution parallèle complète s de M_1 et M_2 tel que $entree(s) = t$, $sortie_1(s) = w_1$ et $sortie_2(s) = w_2$.*

Proposition 18. *Le problème d'équivalence de deux dB2Ws séquentiels M_1 et M_2 peut être réduit en $O(|M_1| \times |M_2|)$ au problème d'équivalence de dT2Ws séquentiels ,et vice versa.*

Preuve. Prenons deux dB2Ws séquentiels N_1 et N_2. Leurs domaines respectifs sont définis par des automates déterministes descendants, dTA$^{\downarrow}s$, obtenus en temps linéaire par la simple suppression de la sortie. De ce fait, tester qu'ils sont tous deux définis sur un même domaine se fait en temps $O(|N_1| \times |N_2|)$ Champavère *et al.* (2008). Nous pouvons maintenant assumer que N_1 et N_2 partagent un même domaine, l'inégalité à ce niveau impliquerait la non équivalence des deux transducteurs.

Nous construisons deux dT2Ws M_1 et M_2 transformant les exécutions parallèles complètes de N_1 et N_2 dans leurs sorties respectives, i.e. $(s, w_i) \in [\![M_i]\!]$ ssi $(entree(s), w_i) \in [\![N_i]\!]$.

Le transducteur M_i a pour ensemble d'états $Q_{M_i} = Q_{N_1} \times Q_{N_2} \times \{\mathtt{o}\}$ dont la règle initiale est $init = \{\langle \mathtt{o}, x_0 \rangle\}$. L'ensemble des règles est construit de la manière suivante :

$$\frac{\begin{array}{c} r_1 = f(\langle q_1^1, x_1^1\rangle, \ldots, \langle q_1^k, x_1^k\rangle) \to \langle q_1^0, w_1^0 \cdot x_1^1 \cdot \ldots \cdot x_1^k \cdot w_1^k\rangle \in rul_{N_1} \\ r_2 = f(\langle q_2^1, x_2^1\rangle, \ldots, \langle q_2^k, x_2^k\rangle) \to \langle q_2^0, w_2^0 \cdot x_2^1 \cdot \ldots \cdot x_2^k \cdot w_2^k\rangle \in rul_{N_2} \end{array}}{[q_1^0, q_2^0]([r_1, r_2](x_1, \ldots, x_k)) \to w_i^0 \cdot \langle [q_1^1, q_2^1], x_1\rangle \cdot \ldots \cdot \langle [q_1^k, q_2^k], x_k\rangle \cdot w_i^k \in rul_{M_i}}$$

auquel on ajoute les règles initiales :

$$\frac{\begin{array}{c} r_1 = f(\langle q_1^1, x_1^1\rangle, \ldots, \langle q_1^k, x_1^k\rangle) \to \langle q_1^0, w_1^0 \cdot x_1^1 \cdot \ldots \cdot x_1^k \cdot w_1^k\rangle \in rul_{N_1} \\ r_2 = f(\langle q_2^1, x_2^1\rangle, \ldots, \langle q_2^k, x_2^k\rangle) \to \langle q_2^0, w_2^0 \cdot x_2^1 \cdot \ldots \cdot x_2^k \cdot w_2^k\rangle \in rul_{N_2} \\ init_{M_2}(q_1^0) = w_1^d \cdot x_0 \cdot w_1^f \qquad\qquad init_{M_2}(q_2^0) = w_2^d \cdot x_0 \cdot w_2^f \end{array}}{\begin{array}{c} r_i' = f(\langle q_i^1, x_i^1\rangle, \ldots, \langle q_i^k, x_i^k\rangle) \to \langle q_i^0, w_i^d \cdot w_i^0 \cdot x_i^1 \cdot \ldots \cdot x_i^k \cdot w_i^k \cdot w_i^f\rangle \text{ pour } 1 \le i \le 2 \\ \mathtt{o}([r_1', r_2'](x_1, \ldots, x_k)) \to w_i^0 \cdot \langle [q_1^1, q_2^1], x_1\rangle \cdot \ldots \cdot \langle [q_1^k, q_2^k], x_k\rangle \cdot w_i^k \in rul_{M_i} \end{array}}$$

M_1 et M_2 sont bien déterministes descendants, chaque partie gauche contenant les règles servant à déterminer de manière unique la partie droite. Il reste à prouver que $(s, w_i) \in [\![M_i]\!]_{[q_1, q_2]}$ si et seulement s'il existe une exécution parallèle s atteignant respectivement les états q_1 et q_2 de N_1 et N_2 et $sortie_i(s) = w_i$.

Pour ce qui est des feuilles, nous avons une exécution parallèle s'il existe les règles $r_i = a \to \langle q_i, w_i\rangle \in rul_{N_i}$. Par construction, $[q_1, q_2]([r_1, r_2]) = w_i \in rul_{M_i}$ et donc $([r_1, r_2], w_i) \in [\![M_i]\!]_{[q_1, q_2]}$. Supposons maintenant que notre hypothèse est vérifiée pour toute exécution parallèle contenant au plus n noeuds. Considérons une exécution parallèle $s = [r_1, r_2](s_1, \ldots, s_k)$ de taille $n+1$. Son existence implique que $r_i = f(\langle q_i^1, x_i^1\rangle, \ldots, \langle q_i^k, x_i^k\rangle) \to \langle q_i^0, w_i^0 \cdot x_i^1 \cdot \ldots \cdot x_i^k \cdot w_i^k\rangle \in rul_{N_i}$. La construction nous donne que $[q_1^0, q_2^0]([r_1, r_2](x_1, \ldots, x_k)) \to w_i^0 \cdot \langle [q_1^1, q_2^1], x_1\rangle \cdot \ldots \cdot \langle [q_1^k, q_2^k], x_k\rangle \cdot w_i^k \in rul_{M_i}$. Par hypothèse d'induction, $(s_j, x_i^j) \in [\![M_i]\!]_{[q_1^i, q_2^i]}$ pour $1 \le j \le k$, tel que $sortie_i(s_j) = x_i^j$. Par conséquent, $(s, w_i) \in [\![M_i]\!]_{[q_1^0, q_2^0]}$ avec $w_i = w_i^0 \cdot x_i^1 \cdot \ldots \cdot x_i^k \cdot w_i^k$ correspondant à $sortie_i(s)$. Cette propriété est vérifiée pour toute paire d'états atteints lors d'une exécution parallèle, particulièrement pour les états finaux dont la construction mène de la même manière à l'état inital de M_i, ce qui implique que $(s, w_i) \in [\![M_i]\!]$ ssi il existe une exécution parallèle complète s sur N_1 et N_2 avec $sortie_i(s) = w_i$.

L'équivalence de M_1 et M_2 revient à celle de N_1 et N_2 puisque pour chaque $(t, w_1) \in [\![N_1]\!]$ et $(t, w_2) \in [\![N_2]\!]$ il existe une exécution parallèle complète s sur S_1 et S_2 tel que $entree(s) = t$ et $sortie_i(s) = w_i$, et $[\![M_1]\!] = [\![M_2]\!]$ ssi $w_1 = w_2$.

De plus, la non équivalence de M_1 et M_2, partageant un même domaine, peut uniquement découler de l'existence d'au moins deux paires $(s, w_1) \in [\![TOP_1]\!]$ et $(s, w_2) \in [\![TOP_2]\!]$ tel que $w_1 \neq w_2$. M_1, M_2, N_1 et N_2 étant déterministes, ce sont les deux seuls couples partageant une même partie gauche s et donc les uniques w_1 et w_2 tel que $(entree(s), w_i) \in [\![N_i]\!]$. N_1 et N_2 sont donc eux aussi inéquivalents.

Le problème d'équivalence des dT2Ws séquentiels se réduit de la même manière à celui des dB2Ws séquentiels, sans avoir besoin de gérer la sortie initiale, tout dT2W pouvant se ramener à une forme n'ayant qu'un unique état en initialisation. Le seul vrai changement vient du test d'égalité du domaine, reposant maintenant sur l'équivalence de deux dTA$^{\downarrow}s$. □

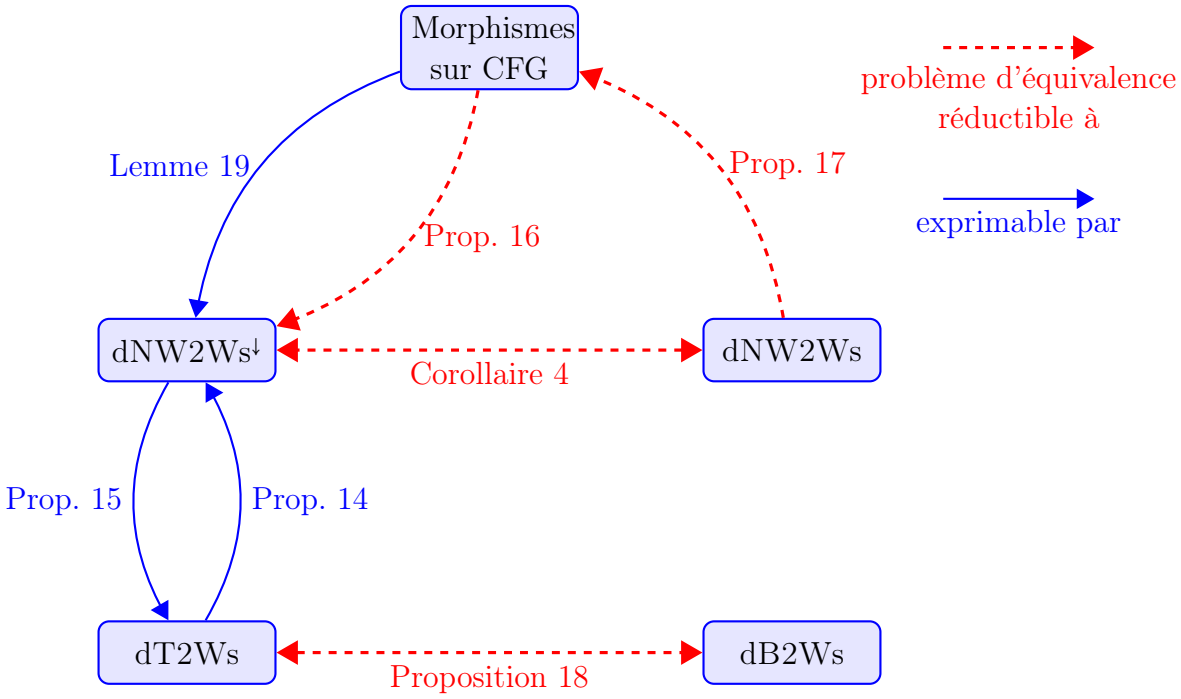


FIGURE 4.1 – Relations liants les différentes classes de transductions

Chapitre 5

Normalisation et minimisation des transducteurs descendants d'arbres en mots

Avant de nous intéresser à l'apprentissage d'une classe de transducteurs d'arbres en mots, les dST2Ws, il est nécessaire de caractériser quelle famille de transformations nous cherchons à apprendre. Suivant le modèle d'inférence grammaticale, il est également nécessaire de définir une forme normale pour les dST2Ws. Le but est d'assurer l'existence d'un transducteur canonique (minimal et unique) représentant chaque transformation définissable par un dST2W. Pour cela nous introduirons un algorithme de normalisation et étudierons le problème de minimisation des dST2Ws.

5.1 dST2W travailleur

La notion de dST2W *travailleur* partage la même dynamique que celle pour les mots, à savoir, à produire le plus de sortie le plus tôt possible. Cette notion de « plus tôt » est moins naturelle pour les arbres, et dépend principalement du choix de l'ordre dans lequel cet arbre est parcouru. Ici, nous choisissons de nous baser sur le parcours en profondeur d'un arbre, nous cherchons donc à produire la sortie le plus en haut et à gauche possible.

Définition 42. *Un* dST2W $M = (\Sigma, \Delta, Q, init, rul)$ *est dit* travailleur *(ed*ST2W*) si et seulement si il respecte les deux propriétés suivantes :*

(E1) $lcp(im(\llbracket M \rrbracket_q)) = lcs(im(\llbracket M \rrbracket_q)) = \varepsilon$ *pour tout* $q \in Q$,

(E2) $lcp(im(\llbracket M \rrbracket_{q_0}) \cdot u_1) = \varepsilon$ *pour l'axiome initial* $u_0 \cdot q_0 \cdot u_1 \in init$ *et pour toute règle* $q \xrightarrow{f} u_0 \cdot q_1 \cdot \ldots q_k \cdot u_k$ *et tout* $1 \leq i \leq k$ *nous avons* $lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i) = \varepsilon$.

Intuitivement, la condition **(E1)** assure qu'aucune production ne peut être tirée plus haut. L'absence de préfixe ou suffixe commun assure que toute la sortie commune, ne dépendant pas du sous arbre qu'il reste à lire, a déjà été produite. La condition **(E2)** vérifie quant à elle que rien ne peut être produit

plus tôt, à gauche, sur une règle. Le fait que la condition **(E2)** assure d'être « travailleur à gauche » repose sur le lemme suivant :

Lemme 22. *Pour toute règle $q \xrightarrow{f} u_0 \cdot q_1 \cdot \ldots q_k \cdot u_k$ d'un d*ST2W*, le fait que $lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i) = \varepsilon$ pour tout $1 \leq i \leq k$ implique que $lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i \cdot \ldots im(\llbracket M \rrbracket_{q_k}) \cdot u_k) = \varepsilon$.*

Preuve. Dénotons par $lcp_i^r = lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i \cdot \ldots im(\llbracket M \rrbracket_{q_k}) \cdot u_k)$. Ce lemme peut être prouvé par récurrence sur la taille de la partie de la règle concernée. Nous pouvons tout d'abord remarquer que $lcp_k^r = lcp(im(\llbracket M \rrbracket_{q_k}) \cdot u_k) = \varepsilon$ par définition.

Supposons maintenant que, pour un $i \neq k$ donné, $lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i) = \varepsilon$ et $lcp_i^r \neq \varepsilon$. La seule possibilité pour vérifier cela est que $\varepsilon \in im(\llbracket M \rrbracket_{q_i}) \cdot u_i$. En effet, la seule autre possibilité pour que $lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i) = \varepsilon$ est qu'il existe dans cet ensemble deux mots non vides partageant aucun préfixe commun. Cela impliquerai que $lcp_i^r = \varepsilon$, ce qui s'écarte de notre supposition initiale.

Si $\varepsilon \in im(\llbracket M \rrbracket_{q_i}) \cdot u_i$, alors le fait que $lcp_i^r \neq \varepsilon$ nous donne que $lcp_(i+1) \neq \varepsilon$.

Si $lcp(im(\llbracket M \rrbracket_{q_i}) \cdot u_i) = \varepsilon$ pour tout $1 \leq i \leq k$, l'hypothèse que $lcp_j^r \neq \varepsilon$ pour $1 \leq j \leq k$ nous ramène, par récurrence, au fait que $lcp_k^r = lcp(im(\llbracket M \rrbracket_{q_k}) \cdot u_k) \neq \varepsilon$ ce qui est impossible. □

Exemple 38. *Reprenons le d*ST2W *M_{35} (de l'exemple 35) défini sur l'alphabet d'entrée $\Sigma = \{f^{(2)}, g^{(1)}, a^{(0)}\}$ et un alphabet de sortie $\Delta = \{a, b, c\}$. On rappelle qu'il a pour états $Q = \{q_0, q_1\}$ et pour axiome initial $init = \{q_0\}$. L'ensemble de ses règles rul est le suivant :*

$$q_0 \xrightarrow{f} q_1 \cdot ac \cdot q_1, \qquad q_1 \xrightarrow{g} q_1 \cdot abc, \qquad q_1 \xrightarrow{a} \varepsilon.$$

*Nous remarquons que toute production de ce transducteur est de la forme $(abc)^* \cdot ac \cdot (abc)^*$, ou $a \cdot (bca)^* \cdot (cab)^* \cdot c$, la répétition de "bca" du nombre de "g" dans le sous arbre gauche, et "cab" du nombre de "g" dans le sous arbre droit. M_{35} n'est donc pas travailleur, le "a" et le "c" pouvant être produits dès l'axiome initial, ce qui donnerait naissance à deux états q_g et q_d remplaçant q_1 pour traiter respectivement le sous arbre gauche, et droit. Les règles du ed*ST2W *M_{38}, tel que $\llbracket M_{35} \rrbracket = \llbracket M_{38} \rrbracket$, sont les suivantes :*

$$q_0 \xrightarrow{f} q_g \cdot ac \cdot q_d, \quad q_g \xrightarrow{f} bca \cdot q_g, \quad q_g \xrightarrow{a} \varepsilon, \quad q_d \xrightarrow{f} cab \cdot q_d, \quad q_g \xrightarrow{a} \varepsilon.$$

avec comme règle initiale $a \cdot q_0 \cdot c$.

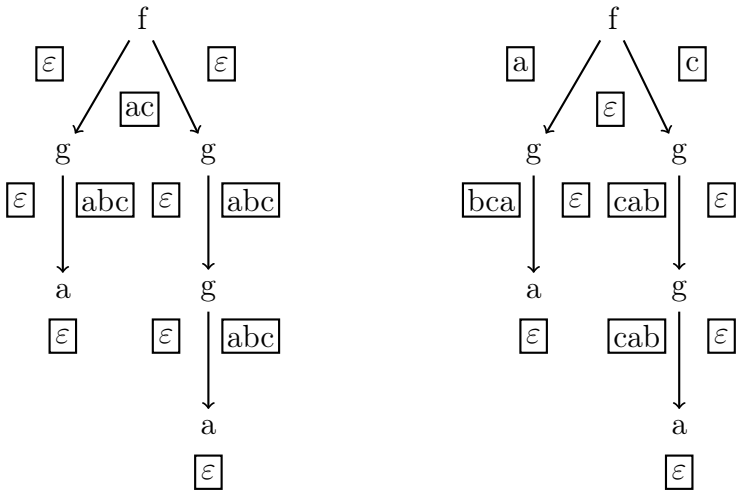


FIGURE 5.1 – Exemple d'exécution de M_{38} et sa version « travailleuse »

5.2 Caractérisation sémantique

Il est nécessaire maintenant d'identifier la classe des transformations définissables par les *ed*ST2W*s*. L'approche des *ed*ST2W*s* étant descendante, chaque transformation à partir d'un état dépendant du chemin qui le lie à la racine, il est nécessaire de pouvoir formaliser dans chaque transformation la notion de résiduel pour un chemin donné. Contrairement aux transducteurs de mots et divers automates, la notion de résiduel pour notre classe de transformations n'est pas si évidente que cela. Pour l'illustrer, nous verrons en quoi la production séquentielle de mots sur les arbres fait échouer la définition classique d'un résiduel, avant de proposer notre solution, se basant sur une application récursive d'un procédé de décomposition d'une transformation en une séquence de mots et de sous transformations, comme nous pouvons le trouver dans les règles d'un *ed*ST2W.

5.2.1 Approche naïve

Le domaine des *d*ST2W*s* correspondant à celui des $d\text{TA}^{\downarrow}s$, à savoir les langages réguliers d'arbres clos par chemins, Il serait naturel de vouloir adopter la définition des résiduels de ces dernier à notre modèle. Cette approche, définissant le résiduel d'un chemin à partir de n'importe quel contexte ayant un trou à cet endroit, est cependant voué à l'échec. En effet, en cherchant à diviser chaque production en deux parties, une dépendant du contexte, et une autre dépendant du sous arbre venant combler le trou, on s'aperçoit que ce découpage ne dépend pas uniquement des sous arbres considérés mais aussi du contexte.

Considérons une transformation τ et un contexte C appartenant à son domaine. Le résiduel $C^{-1}\cdot\tau$ devrait être une transformation ayant pour domaine $C^{-1}\cdot dom(\tau)$ et dont les productions satisferaient l'équation suivante $\tau(C[t]) = Gauche(C,\tau)\cdot C^{-1}\tau(t)\cdot Droite(C,\tau)$ où $Gauche(C,\tau)$ et $Droite(C,\tau)$ sont intuitivement les partie gauche et droite de la production dépendant du contexte C. Ces parties gauche et droite correspondraient à la plus grande partie commune à gauche et à droite de tous les arbres de $dom(\tau)$ partageant un même contexte C, i.e. :

$$Gauche(C,\tau) = lcp\{\tau(C[t]) \mid C[t] \in dom(\tau)\},$$
$$Droite(C,\tau) = lcs\{Gauche(C,\tau)^{-1}\cdot\tau(C[t]) \mid C[t] \in dom(\tau)\}.$$

On décide de mettre le plus de sortie à gauche, le calcul de la partie droite nécessite donc d'avoir en premier lieu retirer la partie gauche pour éviter de compter deux fois une partie commune.

On souhaiterait que, à l'instar des automates d'arbres, la propriété de clôture par chemins nous permette de dire que deux contexte C_1 et C_2 partageant un même trou x en p, ont également un même résiduel, $C_1^{-1}\cdot\tau = C_2^{-1}\cdot\tau$. Dans l'exemple qui suit, nous montrons que cette propriété n'est pas partagée par toutes les transformations définissables par les dST2Ws.

Exemple 39. *Considérons le* dST2W M_{39} *sur l'alphabet d'entrée* $\Sigma = \{f^{(2)}, a^{(0)}, b^{(0)}\}$*, et l'alphabet de sortie* $\Delta = \{a, b\}$*, composé d'un ensemble d'états* $Q = \{q_0, q_1, q_2\}$*, dont* q_0 *est l'état initial, et des règles suivantes :*

$$q_0 \xrightarrow{f} q_1\cdot q_2, \qquad q_1 \xrightarrow{a} \varepsilon, \qquad q_1 \xrightarrow{b} ab, \qquad q_2 \xrightarrow{a} \varepsilon, \qquad q_2 \xrightarrow{b} a.$$

Il défini la transformation $\tau_{39} = [\![M_{39}]\!]$ *:*

$$\tau_{39}(f(a,a)) = \varepsilon, \quad \tau_{39}(f(b,a)) = ab, \quad \tau_{39}(f(a,b)) = a, \quad \tau_{39}(f(b,b)) = aba.$$

On s'intéresse aux deux contextes appartenant au domaine $dom(\tau_{39})$*, ayant un trou* x *à* $p = (f,1)$*, à savoir* $C_1 = f(x,a)$ *et* $C_2 = f(x,b)$*, pour en calculer les parties gauches et droites :*

$$Gauche(C_1,\tau_{39}) = lcp\{\varepsilon, ab\} = \varepsilon, \qquad Gauche(C_2,\tau_{39}) = lcp\{a, aba\} = a,$$
$$Droite(C_1,\tau_{39}) = lcs\{\varepsilon, ab\} = \varepsilon, \qquad Droite(C_2,\tau_{39}) = lcs\{\varepsilon, ba\} = \varepsilon.$$

Ce qui nous donne les résiduels suivants :

$$c_1^{-1}\cdot\tau_{39}(a) = \varepsilon, \qquad c_2^{-1}\cdot\tau_{39}(a) = \varepsilon,$$
$$c_1^{-1}\cdot\tau_{39}(b) = ab, \qquad c_2^{-1}\cdot\tau_{39}(b) = ba.$$

Nous n'avons pas l'égalité supposée $c_1^{-1}\cdot\tau_{39} \neq c_2^{-1}\cdot\tau_{39}$ *ce qui rend impossible la définition d'un résiduel* $p^{-1}\cdot\tau_{39}$*.*

Le problème vient du fait que les résiduels sont calculés indépendamment pour chaque contexte ce qui entraine des problèmes de compatibilité. Pour palier à ce problème, nous allons introduire la notion de décomposition d'une transformation, se basant sur l'intégralité de la transformation, à partir de laquelle nous pourrons construire récursivement les résiduels de chemins.

5.2.2 Décompositions et résiduels

La solution qui s'offre à nous est d'essayer de retranscrire la sémantique des *ed*ST2W*s* dans les transformations qu'ils reconnaissent. Pour cela, nous allons chercher à *décomposer* une transformation en une séquence de sous-transformations et de mots de sorties, à l'image d'une règle d'un *d*ST2W*s*.

Il est nécessaire dans un premier temps de traduire le fait d'être travailleur pour une transformation. Une transformation τ est dite *réduite* si aucune sortie commune ne peut en être produite, i.e. $lcp(im(\tau)) = lcs(im(\tau)) = \varepsilon$. C'est particulièrement le cas pour les transformations $[\![M]\!]_q$ que nous cherchons à retrouver dans notre décomposition.

Le *noyau* d'une transformation τ, i.e. sa partie réduite, $Noyau(\tau)$, s'obtient de la manière suivante :

$$Gauche(\tau) = lcp(im(\tau)), \qquad Droite(\tau) = lcs(Gauche(\tau)^{-1} \cdot im(\tau)),$$
$$Noyau(\tau) = \{(t, Gauche(\tau)^{-1} \cdot \tau(t) \cdot Droite(\tau)^{-1}) \mid (t, w) \in \tau\}.$$

Exemple 40. *Considérons la transformation τ_{40} suivante :*

$$\tau_{40}(f(a,a)) = abc, \qquad \tau_{40}(f(a,b)) = abbc,$$
$$\tau_{40}(f(b,a)) = abbbc, \qquad \tau_{40}(f(b,b)) = abbbbc.$$

Elle n'est pas réduite, chacune de ses production ayant un préfixe commun $Gauche(\tau_{40}) = ab$, et un suffixe commun $Droite(\tau_{40}) = c$ n'étant pas vides. Son noyau, $\tau^o_{40} = Noyau(\tau_{40})$, est obtenu en supprimant ce préfixe et suffixe de chaque production, soit :

$$\tau^o_{40}(f(a,a)) = \varepsilon, \quad \tau^o_{40}(f(a,b)) = b, \quad \tau^o_{40}(f(b,a)) = bb, \quad \tau^o_{40}(f(b,b)) = bbb.$$

τ^o_{40} est clairement réduit, le plus grand préfixe et suffixe commun ayant été retiré. De plus, $\tau_{40}(t) = ab \cdot \tau^o_{40}(t) \cdot c$.

Nous pouvons directement observer les propriétés suivantes :

Proposition 19. *Pour toute transformation τ, son noyau $\tau^o = Noyau(\tau)$ est une transformation réduite, elle partage un même domaine $dom(\tau) = dom(\tau^o)$, et pour tout arbre $t \in dom(\tau)$ la sortie produite respecte $\tau(t) = Gauche(\tau) \cdot \tau^o(t) \cdot Droite(\tau)$.*

Par *n-chemins*(t), nous représentons l'ensemble des chemins de t auxquels nous ajoutons à la suite les labels possibles du noeud considéré.

Définition 43. *Soit τ une transformation réduite, et $f^{(k)} \in$ n-chemins$(dom(\tau))$ un symbole de la racine, la* décomposition *de τ pour f est un tuple $(u_0, \tau_1, u_1, \ldots, \tau_k, u_k)$, pour lequel chaque u_i est un mot de Δ et chaque τ_i est une transformation, respectant les contraintes suivantes :*

(D1) *$f(t_1, \ldots, t_k) \in dom(\tau)$ si et seulement si $t_i \in dom(\tau_i)$, pour $1 \leq i \leq k$;*

(D2) *$\tau(f(t_1, \ldots, t_k)) = u_0 \cdot \tau_1(t_1) \cdot \ldots \cdot \tau(t_k) \cdot u_k$ pour tout $f(t_1, \ldots, t_k) \in dom(\tau)$;*

(C1) *τ_i est réduit, pour $1 \leq i \leq k$;*

(C2) *$lcp(im(\tau_i) \cdot u_i \cdot \ldots \cdot im(\tau_k) \cdot u_k) = \varepsilon$ pour $1 \leq i \leq k$.*

Nous pouvons remarquer que la condition **(D1)** traduit que le domaine est clos par chemin. Nous notons également que chaque symbole d'une transformation n'a pas nécessairement de décomposition comme l'illustre l'exemple suivant.

Exemple 41. *Prenons la transformation τ_{41} définie par :*

$$\tau_{41}(f(a,a)) = abc, \quad \tau_{41}(f(a,b)) = abac, \quad \tau_{41}(f(b,a)) = aabc, \quad \tau_{41}(f(b,b)) = aabac,$$
$$\tau_{41}(g(a,a)) = \varepsilon, \quad \tau_{41}(g(b,a)) = b, \quad \tau_{41}(g(a,b)) = bc, \quad \tau_{41}(g(b,b)) = bbb.$$

La décomposition de τ_{41} pour f est le tuple $(a, \tau'_{41}, b, \tau'_{41}, c)$, dans lequel $\tau'_{41} = \{(a,\varepsilon), (b,a)\}$. Le symbole "g" n'a, quant à lui, n'a aucune décomposition possible.

Considérons à nouveau la transformation τ_{39} (exemple 39) définie par les équations :

$$\tau_{39}(f(a,a)) = \varepsilon, \quad \tau_{39}(f(b,a)) = ab, \quad \tau_{39}(f(a,b)) = a, \quad \tau_{39}(f(b,b)) = aba.$$

Sa décomposition pour le symbole f est $(\varepsilon, \tau'_{39}, \varepsilon, \tau''_{39}, \varepsilon)$, où $\tau'_{39} = \{(a,\varepsilon), (b,ab)\}$ et $\tau'_{39} = \{(a,\varepsilon), (b,a)\}$.

La transformation τ^{o}_{40} (exemple 40) a pour décomposition en f le tuple $(\varepsilon, \tau_{bb}, \varepsilon, \tau_b, \varepsilon)$ avec $\tau_{bb} = \{(a,\varepsilon), (b,bb)\}$ et $\tau_b = \{(a,\varepsilon), (b,b)\}$.

Nous pouvons cependant montrer que s'il existe une décomposition alors elle est unique. Cela repose, en analogie avec les conditions **(E1)** et **(E2)** d'un *ed*ST2W, sur les conditions **(C1)** et **(C2)** qui assurent la canonicité de la décomposition.

Lemme 23. *Pour toute transformation τ et tout symbole $f \in$ n-chemins$(dom(\tau))$, τ a au plus une décomposition f.*

Preuve. Pour le prouver supposons l'existence de deux décompositions $(u_0, \tau_1, u_1, \ldots, \tau_k, u_k)$ et $(u'_0, \tau'_1, u'_1, \ldots, \tau'_k, u'_k)$ de τ pour le symbole f. Nous pouvons dans un premier temps remarquer que pour tout $1 \leq i \leq n$, $dom(\tau_i) = dom(\tau'_i)$, ce que nous obtenons directement par la propriété **(D1)**. Par la suite, nous montrons par induction que ces deux décompositions sont égales élément par élément.

Prenons comme hypothèse d'induction les deux conditions suivantes pour un i choisi entre1 et k :

1. $u_j = u'_j$ pour $0 \leq j < i$,
2. $\tau_j = \tau'_j$ pour $0 < j < i$.

Montrons tout d'abord que $u_i = u'_i$. On peut remarquer par **(D2)** que, pour $f(t_1, \ldots, t_k) \in dom(\tau)$,

$$u_0 \cdot \tau_1(t_1) \cdot \ldots \cdot \tau_k(t_k) \cdot u_k = u'_0 \cdot \tau'_1(t_1) \cdot \ldots \cdot \tau'_k(t_k) \cdot u'_k,$$

et par hypothèse d'induction

$$u_i \cdot \tau_i(t_i) \cdot \ldots \cdot \tau_k(t_k) \cdot u_k = u'_i \cdot \tau'_i(t_i) \cdot \ldots \cdot \tau'_k(t_k) \cdot u'_k.$$

Le mot u_i est donc préfixe de u'_i ou vice versa. Sans perte de généralité, supposons que u_i est le préfixe de u'_i, i.e. $u'_i = u_i \cdot v$, donc, pour tout $f(t_1, \ldots, t_k) \in dom(\tau)$,

$$u'_i \cdot v \cdot \tau_i(t_i) \cdot \ldots \cdot \tau_k(t_k) \cdot u_k = u'_i \cdot \tau'_i(t_i) \cdot \ldots \cdot \tau'_k(t_k) \cdot u'_k.$$

En conséquence, v est le préfixe commun de $im(\tau_i) \cdot u_i \ldots im(\tau_k) \cdot u_k$ ce qui, d'après **(C2)**, peut uniquement être ε, ce qui implique que $u_i = u'_i$. Cela montre aussi que pour tout $f(t_1, \ldots, t_k) \in dom(\tau)$, nous avons

$$\tau_i(t_i) \cdot \ldots \cdot \tau_k(t_k) \cdot u_k = \tau'_i(t_i) \cdot \ldots \cdot \tau'_k(t_k) \cdot u'_k. \tag{5.1}$$

Fixons un arbre $f(s_1, \ldots, s_k) \in dom(\tau)$ et les mots w et w' suivants :

$$w = u_i \cdot \tau_{i+1}(s_{i+1}) \cdot \ldots \cdot \tau_k(s_k) \cdot u_k \qquad w' = u'_i \cdot \tau'_{i+1}(s_{i+1}) \cdot \ldots \cdot \tau'_k(s_k) \cdot u'_k.$$

D'après (5.1), w est un suffixe de w' ou l'inverse. De nouveau, on choisit que w et un suffixe de w', i.e. $w' = v \cdot w$. Tout $t_i \in dom(\tau_i) = dom(\tau'_i)$ vérifie donc que

$$\tau_i(t_i) \cdot w = \tau'_i(t'_i) \cdot w' \quad \text{ce qui implique que} \quad \tau_i(t_i) = \tau'_i(t_i) \cdot v.$$

En suivant la condition **(C1)**, on sait que τ_i est réduit, et particulièrement que $lcs(im(\tau_i)) = \varepsilon$. Le mot v étant le suffixe commun de toutes les images de τ_i, il doit être égal à ε, ce qui implique que $\tau_i = \tau'_i$. □

Il est aussi intéressant de montrer que les conditions (**C1**) et (**C2**) d'une transformation sont analogues aux conditions (**E1**) et (**E2**) d'un *ed*ST2W. Ce fait permet de lier les transformations définies à partir d'états d'un *ed*ST2W aux transformations que l'on retrouve dans la décomposition.

Proposition 20. *Soit M un ed*ST2W*, pour tout état q de M, $[\![M]\!]_q$ est réduit, de plus pour tout $f \in \Sigma$ tel que $q \xrightarrow{f} u_0 \cdot q_1 \cdot u_1 \cdot \ldots \cdot q_k \cdot u_k \in rul_M$ implique que $[\![M]\!]_q$ a comme décomposition, pour f, le tuple $(u_0, [\![M]\!]_{q_1}, u_1, \ldots, [\![M]\!]_{q_k}, u_k)$.*

Preuve. La transformation $[\![M]\!]_q$ est par définition, (**E1**), réduite. Les conditions (**D1**) et (**D2**) suivent directement la définition de $[\![M]\!]_q$. Les conditions (**C1**) et (**C2**) sont respectivement les adaptations des conditions (**E1**) et (**E2**) pour les transformations. □

Comme évoqué précédemment, la notion de décomposition est centrale pour la définition des résiduels.

Définition 44. *Le* résiduel *d'une transformation réduite τ par un chemin $p \in chemins(dom(\tau))$ est une transformation définie récursivement par :*

$$\varepsilon^{-1}\tau = \tau$$

$$[(f,i)\cdot p]^{-1}\tau = \begin{cases} p^{-1}\tau_i & \text{si } f^{-1}\tau \text{ et } d \text{ sont définis,} \\ \text{indéfini} & \text{sinon,} \end{cases}$$

$d = (u_0, \tau_1, u_1, \ldots, \tau_k, u_k)$ étant la décomposition de $f^{-1}\tau$.

Le résiduel $p^{-1}\tau$ d'une transformation τ, pas forcément réduite, et d'un chemin $p \in chemins(dom(\tau))$ est défini par $p^{-1}Noyau(\tau)$ (à condition qu'il soit défini).

Grâce à cela, nous pouvons maintenant montrer la relation entre les résiduels et les transformations définies par les états d'un *ed*ST2W.

Lemme 24. *Prenons M un ed*ST2W *ayant pour état initial q_0. Pour tout chemin $p \in chemins(dom([\![M]\!]))$, le résiduel $p^{-1}[\![M]\!]$ existe et est égal à $[\![M]\!]_{\hat{rul}(q_0,p)}$.*

Preuve. La preuve se fait par induction sur la taille du chemin p. Pour $p = \varepsilon$, en supposant que l'axiome initial de M est $u_0 \cdot q_0 \cdot u_1$, on observe grace à (**E1**) que $Gauche([\![M]\!]) = u_0$ et $Droite([\![M]\!]) = u_1$. Cela implique que $\varepsilon^{-1}[\![M]\!] = Noyau([\![M]\!]) = [\![M]\!]_{q_0}$.

Considérons maintenant le chemin $p \cdot (f,i)$, pour lequel nous supposons que $\hat{rul}(q_0,p) = q$ et $p^{-1}[\![M]\!] = [\![M]\!]_q$. D'après la proposition 20, $[\![M]\!]_q$ a pour décomposition le tuple $(u_0, [\![M]\!]_{q_1}, u_1, \ldots, [\![M]\!]_{q_k}, u_k)$ avec $q \xrightarrow{f} u_0 \cdot q_1 \cdot u_1 \cdot \ldots \cdot q_k \cdot u_k \in rul$. Par conséquent, $[p\cdot(f,i)]^{-1}[\![M]\!] = [\![M]\!]_{q_i}$, sachant que l'état $q_i = \hat{rul}(q_0, p\cdot(f,i)) = q_i$, ce qui montre que $[p\cdot(f,i)]^{-1}[\![M]\!] = [\![M]\!]_{\delta(q_0,p\cdot(f,i))}$. □

5.2.3 Transducteur canonique

Maintenant que la notion de résiduel est bien définie nous pouvons formaliser la classe des transformations reconnaissables par les dST2Ws.

Définition 45. *Une transformation d'arbres en mots τ est* séquentielle descendante *si $p^{-1}\tau$ existe pour tout chemin $p \in chemins(dom(\tau))$. On dénote cette classe de transformation par $\mathcal{STW}$ (de l'anglais « Sequential Trees to Words »).*

Nous introduisons l'équivalence de Myhill-Nerode $\equiv_\tau$ pour les chemins appartenants au domaine d'une transformation séquentielle descendante τ. Pour deux chemins $p_1, p_2 \in chemins(dom(\tau))$ nous avons :

$$p_1 \equiv_\tau p_2 \Leftrightarrow p_1^{-1}\tau = p_2^{-1}\tau.$$

L'index de Myhill-Nerode se définit, comme dans les chapitres précédents, par le nombre de classes d'équivalences $\equiv_\tau$ distinctes.

Nous pouvons dès à présent chercher à construire le *transducteur canonique* d'une transformation $\mathcal{STW}$,noté $Cano(\tau)$. Nous proposons ici une construction de ce transducteur, que nous prouverons par la suite comme étant l'unique edST2W minimal représentant la transformation τ. Ce transducteur est un edST2W définit par le tuple suivant $(\Sigma, \Delta, Q, init, rul)$. Les états correspondent aux classes d'équivalence de Myhill-Nerode $\equiv_\tau$:

$$Q = \{[p]_\tau \mid p \in chemins(dom(\tau))\},$$

où $[p]_\tau = \{p' \in chemins(dom(\tau)) \mid p \equiv_\tau p'\}$. L'axiome initial est construit de la manière suivante

$$init = Gauche(\tau) \cdot [\varepsilon]_\tau \cdot Droite(\tau).$$

et pour tout $p \cdot f \in n\text{-}chemins(dom(\tau))$, la règle correspondante est construite à partir de la décomposition $(u_0, \tau_1, u_1, \ldots, \tau_k, u_k)$ de $p^{-1}\tau$ pour f par :

$$[p]_\tau \xrightarrow{f} u_0 \cdot [p \cdot (f,1)]_\tau \cdot u_1 \cdot \ldots \cdot [p \cdot (f,k)]_\tau \cdot u_k.$$

Nous pouvons remarquer que cette règle ne dépend pas du représentant de la classe d'équivalence choisie ; $p_1 \equiv_\tau p_2$ impliquant que les résiduels $p_1^{-1}\tau$ et $p_2^{-1}\tau$ sont identiques, et à fortiori partagent une même décomposition.

Proposition 21. *Le transducteur $Cano(\tau)$ définit la transformation τ.*

Preuve. Il suffit de prouver que $[\![Cano(\tau)]\!]_{[p]_\tau} = p^{-1}\tau$ pour tout $p \in chemins(dom(\tau))$. Cela peut se faire par simple induction sur le poids des arbres pris en entrée.

Pour tout chemin $p \in chemins(dom(\tau))$, le fait qu'il existe un arbre $a^{(0)} \in dom(p^{-1}\tau)$ implique qu'il existe une règle $[p]_\tau \xrightarrow{a} u_0$, u_0 étant la décomposition de $p^{-1}\tau$ pour $a^{(0)}$. La propriété **(D2)** des décompositions permet d'en déduire que $u_0 = p^{-1}\tau(a) = [\![Cano(\tau)]\!]_{[p]_\tau}(a)$.

Prenons pour hypothèse d'induction le fait que pour un n donné, $[\![Cano(\tau)]\!]_{[p]_\tau}(t) = p^{-1}\tau(t)$ pour tout chemin $p \in dom(\tau)$ et tout arbre $t \in dom(p^{-1}\tau)$ ayant pour taille $|t| \leq n$.

Considérons un arbre $t = f(t_1, \ldots, t_k) \in dom(p^{-1}\tau)$ de taille $|t| = n + 1$ pour n'importe quel chemin $p \in chemins(dom(\tau))$. Il existe donc, d'après **(D1)**, une décomposition de $p^{-1}\tau$ pour f de la forme $(u_0, \tau_1, u_1, \ldots, \tau_k, u_k)$. Par construction, il existe une règle de $Cano(\tau)$ égale à $[p]_\tau \xrightarrow{f} u_0 \cdot [p\cdot(f,1)]_\tau \cdot u_1 \cdot \ldots \cdot [p\cdot(f,k)]_\tau \cdot u_k$. Par hypothèse d'induction et par définition des résiduels, nous avons que, pour tout $1 \leq i \leq k$,

$$\tau_i(t_i) = (p \cdot (f,i))^{-1}\tau(t_i) = [\![Cano(\tau)]\!]_{[p\cdot(f,i)]_\tau}(t_i),$$

ce qui implique, d'après $(D2)$, que $[\![Cano(\tau)]\!]_{[p]_\tau}(t) = p^{-1}\tau(t)$. L'hypothèse d'induction est donc vérifié pour tout arbre de taille $n + 1$.

□

Il reste maintenant à prouver que $Cano(\tau)$ est bien l'unique edST2W *minimal* représentant τ, et que toute transformation exprimable par un dST2W l'est aussi par un edST2W pour pouvoir définir un théorème de Myhill-Nerode pour notre classe et aboutir à un algorithme d'apprentissage.

5.3 Minimisation

Nous nous intéressons ici au problème de minimisation des dST2Ws. Le but étant de réduire la taille d'un transducteur, taille qui correspond à la somme du nombre d'états, à laquelle on ajoute la taille des règles. La taille d'une règle étant le nombre d'états contenus dans sa partie droite et la somme des tailles des mots qui la compose, i.e. la taille d'une règle $q \xrightarrow{f} u_0 \cdot q_1 \cdot \ldots \cdot q_k \cdot u_k$ est égale à $k + |f| + |u_1| + \ldots + |u_k|$.

Le principal problème de la minimisation ne vient pas de construire, pour un transducteur M appartenant à une classe $\mathcal{M}$, un transducteur de plus petite taille équivalent mais de montrer que l'on ne peut pas le réduire encore plus. Nous pouvons donc voir ce problème comme la définition et l'étude d'une fonction `MINIMISER` cherchant à répondre à la question suivante :

Problème : $\texttt{MINIMISER}_{\mathcal{M}}$
Entrée : $M \in \mathcal{M}$ et un entier naturel K.
Question : Existe t'il un $M' \in \mathcal{M}$ equivalent à M tel que $|M'| \leq K$?

5.3.1 Minimisation des edST2Ws

Avant de nous intéresser à la minimisation du modèle plus général, commençons par la minimisation des edST2Ws, plus simple. En effet, la construction d'un edST2W minimal à partir d'un edST2W revient à étudier les relations d'équivalences (de Myhill-Nerode) liant ces états afin de joindre les états équivalents et produire le transducteur canonique correspondant.

Mais pour se persuader que ce transducteur canonique est l'unique edST2W minimal, il est intéressant de pointer la propriété de *bi-simulation* des edST2Ws, utilisée dans le reste de cette section.

Lemme 25. *Considérons deux edST2Ws $M = (\Sigma, \Delta, Q, init, rul)$ et $M' = (\Sigma, \Delta, Q', init', rul')$ définissant une même transformation $\tau = [\![M]\!] = [\![M']\!]$ et ayant comme axiomes initiaux respectifs $init = u_0 \cdot q_0 \cdot u_1$ et $init' = u'_0 \cdot q'_0 \cdot u'_1$. Alors, $u_0 = u'_0$ et $u_1 = u'_1$, et pour tout $p \in chemins(dom(T))$, les deux états $q = \hat{rul}(q_0, p)$ et $q' = \hat{rul}'(q'_0, p)$ vérifient :*

1. $[\![M]\!]_q = [\![M]\!]_{q'}$,
2. $\hat{rul}(q, f)$ *est défini si et seulement si* $\hat{rul}'(q', f)$ *l'est, pour tout* $f \in \Sigma$,
3. *si* $q \xrightarrow{f} u_0 \cdot q_1 \cdot u_1 \cdot \ldots \cdot q_k \cdot u_k \in rul$ *et* $q' \xrightarrow{f} u'_0 \cdot q'_1 \cdot u'_1 \cdot \ldots \cdot q'_k \cdot u'_k$, *alors* $u_i = u'_i$ *for* $0 \leq i \leq k$.

Preuve. Tout d'abord, il faut remarquer que pour $t \in dom(\tau)$:

$$[\![M]\!](t) = u_0 \cdot [\![M]\!]_{q_0}(t) \cdot u_1 = u'_0 \cdot [\![M']\!]_{q'_0}(t) \cdot u'_1 = [\![M']\!](t).$$

Par conséquent, u_0 est le préfixe de u'_0 ou inversement. On suppose, sans perte de généralité, que u_0 est le préfixe de u'_0, i.e. $u'_0 = u_0 \cdot v$. Si v n'est pas le mot vide alors, par définition, il est contenu dans le préfixe commun de $im([\![M']\!]_{q'_0}) \cdot u'_1$ qui, d'après (**E2**), est ε, donc $v = \varepsilon$ et $u_0 = u'_0$.

Il en va de même pour u_1, devant être un suffixe de u'_1 ou inversement. Supposons que u_1 est le suffixe de u'_1, i.e. $u_1 = v \cdot u'_1$. Puisque nous savons, par définition, que $u_0 \cdot [\![M]\!]_{q_0}(t) \cdot u_1 = u_0 \cdot [\![M']\!]_{q'_0}(t) \cdot v \cdot u_1$, et que l'on a prouvé que $u_0 = u'_0$, v doit être un suffixe commun des productions $im([\![M]\!]_{q_0})$. Sachant que $lcs(im([\![M]\!]_{q_0})) = \varepsilon$ (**E1**), $v = \varepsilon$ et $u_1 = u'_1$.

Nous avons donc les égalités $u_0 = u'_0$, et $u_1 = u'_1$, impliquant que $[\![M]\!]_{q_0} = [\![M']\!]_{q'_0}$, les transducteurs décrivant une même transformation.

Maintenant, nous allons prouver pour un chemin $p \in chemins(dom(\tau))$, et deux états $q = \hat{rul}(q_0, p)$ et $q' = \hat{rul}(q'_0, p)$, le fait que $[\![M]\!]_q = [\![M']\!]_{q'}$ implique que :

1. $\hat{rul}(q, f)$ est défini si et seulement si $\hat{rul}'(q', f)$ l'est, et cela pour tout $f \in \Sigma$, et
2. si $\hat{rul}(q, f) = u_0 \cdot q_1 \cdot u_1 \cdot \ldots \cdot q_k \cdot u_k$ et $\hat{rul}'(q', f) = u'_0 \cdot q'_1 \cdot u'_1 \cdot \ldots \cdot q'_k \cdot u'_k$, alors $u_i = u'_i$ for $0 \leq i \leq k$ et $[\![M]\!]_{q_i} = [\![M']\!]_{q'_i}$, et cela pour tout $1 \leq i \leq k$.

La première condition découle directement de l'hypothèse d'induction, deux transformations égales partageant un même domaine, i.e. $dom([\![M]\!]_q) = dom([\![M']\!]_{q'})$. La deuxième condition se prouve par induction sur i en utilisant les mêmes mécanismes que pour les axiomes. Si nous supposons pour tout $1 \leq j < i \leq k$ que $u_j = u'_j$ et $[\![M]\!]_{q_j} = [\![M']\!]_{q'_j}$, nous pouvons montrer que $u_i = u'_i$ puis que $[\![M]\!]_{q_{i+1}} = [\![M]\!]_{q'_{i+1}}$ en nous basant sur les propriétés (**E1**) et (**E2**).

Puisque $[\![M]\!]_q = [\![M']\!]_{q'}$, alors pour tout arbre $t = f(t_1, \ldots, t_k) \in dom([\![M]\!]_q)$

$$\begin{aligned}[\![M]\!]_q(t) &= u_0 \cdot [\![M]\!]_{q_1}(t_1) \cdot u_1 \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \\ &= u'_0 \cdot [\![M']\!]_{q'_1}(t_1) \cdot u'_1 \cdot \ldots \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k = [\![M']\!]_{q'}(t),\end{aligned}$$

ce qui, par hypothèse d'induction, nous donne l'égalité suivante :

$$u_i \cdot [\![M]\!]_{q_{i+1}}(t_{i+1}) \cdot u_{i+1} \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k = u'_i \cdot [\![M']\!]_{q'_{i+1}}(t_{i+1}) \cdot u'_{i+1} \cdot \ldots \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k.$$

Cette égalité implique que u_i est le préfixe de u'_i ou inversement. Sans perte de généralité, nous pouvons supposer que $u'_i = u_i \cdot v$, v étant préfixe de $lcp(im([\![M]\!]_{q_{i+1}} \cdot u_{i+1} \cdot \ldots \cdot [\![M]\!]_{q_k} \cdot u_k))$. D'après(**E2**), ce préfixe est égal à ε, donc $v = \varepsilon$ et $u_i = u'_i$.

Soit $t = f(t_1, \ldots, t_k) \in dom([\![M]\!]_q)$, $w = u_{i+1} \cdot [\![M]\!]_{q_{i+2}}(t_{i+2}) \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k$ et $w' = u'_{i+1} \cdot [\![M']\!]_{q'_{i+2}}(t_{i+2}) \cdot \ldots \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k$, nous pouvons remarquer que w est le suffixe de w' ou l'inverse. Comme précédemment, choisissons w comme suffixe de w', i.e. $w' = v \cdot w$. Pour tout arbre $t'_{i+1} \in (f, i+1)^{-1} dom([\![M]\!]_q) = (f, i+1)^{-1} dom([\![M']\!]_{q'})$ nous pouvons remarquer que

$$[\![M]\!]_{q_{i+1}}(t'_{i+1}) \cdot w = [\![M']\!]_{q'_{i+1}}(t'_{i+1}) \cdot w'$$

et que par conséquent

$$[\![M]\!]_{q_{i+1}}(t'_{i+1}) = [\![M']\!]_{q'_{i+1}}(t'_{i+1}) \cdot v,$$

impliquant que v est un suffixe commun de $im([\![M]\!]_{q_{i+1}})$ (puisque $(f, i+1)^{-1} dom([\![M]\!]_q) = dom([\![M]\!]_{q_{i+1}})$). D'après (**E1**) tout suffixe est vide, particulièrement $v = \varepsilon$. Nous vérifions donc que $[\![M]\!]_{q_{i+1}} = [\![M']\!]_{q'_{i+1}}$. □

Pour la suite de cette sous section, nous représentons par M un *ed*ST2W défini par le tuple $(\Sigma, \Delta, Q, init, rul)$, ayant pour axiome initial $init = u_0 \cdot q_0 \cdot u_1$.

Nous définissons une relation d'équivalence entre les états de M : $q \equiv_M q'$ si et seulement si $[\![M]\!]_q = [\![M]\!]_{q'}$. Cette relation d'équivalence peut être pré-calculée en utilisant des tests d'équivalence en temps polynomial (comme le rappelle la figure 4.1, ces tests pouvant être, au pire, ramenés aux tests d'équivalence de morphismes sur une *CFG* décidable en temps polynomial).

Par $[q]_{\equiv_M} = \{q' \in Q \mid q \equiv_M q'\}$, nous identifions la classe d'équivalence de l'état q suivant la relation $\equiv_M$. Le résultat de la minimisation est le *transducteur quotient* $M/_{\equiv_M} = (\Sigma, \Delta, Q', init', rul')$, pour lequel $Q' = \{[q]_{\equiv_M} \mid q \in Q\}$, $init' = u_0 \cdot [q_0]_{\equiv_M} \cdot c_1$, et $[q]_{\equiv_M} \xrightarrow{f} u_0 \cdot [q_1]_{\equiv_M} \cdot u_1 \cdot \ldots \cdot [q_k]_{\equiv_M} \cdot u_k \in rul'$ pour toute règle $q \xrightarrow{f} u_0 \cdot q_1 \cdot u_1 \cdot \ldots \cdot q_k \cdot u_k \in rul$. Le précédent lemme 25 (avec $M' = M$) assure que la construction des règles de $M/_{\equiv_M}$ est indépendante des règles du M le représentant.

Lemme 26. *$M/_{\equiv_M}$ est l'unique ed*ST2W *minimal définissant $[\![M]\!]$.*

Preuve. Soit $M/_{\equiv_M} = (\Sigma, \Delta, Q', init', rul')$ ayant pour règle initiale $init' = u'_0 \cdot q'_0 \cdot u'_1$. Choisissons n'importe quel *ed*ST2W $M'' = (\Sigma, \Delta, Q'', init'', rul'')$ equivalent à M tel que $init'' = u''_0 \cdot q''_0 \cdot u''_1$.

Premièrement, montrons que M'' a au moins autant d'états que $M/_{\equiv_M}$. Pour tout état $q' \in Q'$, nous choisissons arbitrairement un chemin $p_{q'}$, tel que $\hat{rul}'(q'_0, p_{q'}) = q'$ (travaillant avec des transducteurs émondés, l'existence d'au moins un chemin de ce type est assurée), et nous identifions par $\lambda(q') = \hat{rul}''(q''_0, p_{q'})$, l'état correspondant dans M''. Il faut vérifier que λ est une fonction injective, i.e. $\lambda(q'_1) = \lambda(q'_2)$ implique que $q'_1 = q'_2$. Appelons q''_1 l'état obtenu par $\lambda(q'_1)$, et par q''_2 celui obtenu par $\lambda(q'_2)$. Il est à noter que $[\![M'']\!]_{q''_1} = [\![M'']\!]_{q''_2}$, ce qui d'après le lemme 25 implique que $[\![M']\!]_{q'_1} = [\![M']\!]_{q'_2}$. Rappelons que q'_1 et q'_2 sont des ensembles d'états de M et, en suivant la construction $M/_{\equiv_M}$, nous pouvons nous apercevoir que $[\![M]\!]_{q_1} = [\![M]\!]_{q_2}$ pour tout $q_1 \in q'_1$ et tout $q_2 \in q'_2$. Par conséquent, $q_1 \equiv_M q_2$ pour tout $q_1 \in q'_1$ et tout $q_2 \in q'_2$, ce qui implique que $q'_1 = q'_2$.

De manière similaire, nous utilisons le lemme 25 pour montrer que toute règle de $M/_{\equiv_M}$ a sa copie contenue dans M'' (modulo renommage des états λ). Donc, M'' contient plus d'états que $M/_{\equiv_M}$, ou M'' a le même nombre d'états, et le même nombre de règles, identiques à celle de $M/_{\equiv_M}$ (modulo le renommage des états effectué par λ). □

Théorème 13. *La minimisation d'un ed*ST2W *se fait en* PTIME*, i.e. pour tout ed*ST2W *il existe un unique ed*ST2W *minimal équivalent pouvant être construit en temps polynomial.*

Les développements présentés dans cette section ont un impact plus important que la minimisation en elle même. Cela vient du fait que la relation d'équivalence $\equiv_M$ est, par essence, une représentation de la relation d'équivalence de Myhill-Nerode $\equiv_{[\![M]\!]}$ et, par conséquent, le transducteur quotient est le transducteur canonique représentant cette transformation.

Lemme 27. *Pour tout ed*ST2W M, $Cano([\![M]\!]) = M/_{\equiv_M}$.

Preuve. Prenons $M/_{\equiv_M} = (\Sigma, \Delta, Q', init', rul')$ avec $init' = u'_0 \cdot q'_0 \cdot u'_1$. Le lemme 24 nous montre que pour tous chemins $p_1, p_2 \in chemins(dom([\![M]\!]))$, $p_1 \equiv_{[\![M]\!]} p_2$ si et seulement si $\hat{rul}(q'_0, p_1) = \hat{rul}(q'_0, p_2)$. Introduisons une application λ des états de $Cano([\![M]\!])$ à ceux de $M/_{\equiv_M}$, de sorte que $\lambda([p]_{[\![M]\!]}) = \hat{rul}(q'_0, p)$. λ est clairement défini pour chaque état, mais il est, de plus, facile de se persuader que λ est une bijection (simple renommage des états). En se reposant sur le lemme 25 nous obtenons que ces deux transducteurs sont identiques (modulo le renommage des états λ). □

Cela nous permet de prouver la minimalité du représentant canonique d'une transformation $\mathcal{STW}$ τ introduit précédemment.

Corollaire 5. *Le transducteur canonique* $Cano(\tau)$ *de n'importe quelle transformation séquentielle descendante* τ *est l'unique ed*ST2W *minimal reconnaissant* τ.

5.3.2 Minimisation de *d*ST2W*s* arbitraires

Dans la classe générale des *d*ST2W*s*, la sortie n'a aucune contrainte et peut être distribuée de manière arbitraire sur les règles, ce qui constitue le principal problème pour la minimisation des transducteurs de cette classe.

Il apparaît difficile de résoudre ce problème efficacement ; le résultat suivant montrant que, à part si PTIME = NP, il n'y a pas d'algorithme polynomial permettant de minimiser un *d*ST2W arbitraire.

Théorème 14. *La minimisation d'un d*ST2W *,i.e. pouvoir décider pour un d*ST2W *et un* $K \geq 0$ *donné s'il existe un d*ST2W *équivalent de taille bornée par* K, *est* NP*-complet.*

Preuve. La preuve de NP-complétude de $\texttt{MINIMISER}_{\text{dST2W}}$ se fait par réduction au problème de satisfiabilité suivant, connu NP-complet (Papadimitriou, 1994, Exercice 9.5.3.).

Problème : $\texttt{SAT}_{\texttt{ONE-IN-THREE}}$
Entrée : $\varphi \in \texttt{3CNF}$, i.e. une conjonction de clauses, où chaque clause est une disjonction de 3 littéraux (un littéral étant une variable ou sa négation).
Question : Existe t'il une valuation satisfaisante φ de sorte que pour chaque clause de φ exactement un littéral soit vrai ?

Prenons n'importe quelle formule 3CNF $\varphi = c_1 \wedge \ldots \wedge c_k$ sur l'ensemble des variables booléennes $x_1, \ldots, x_n$, avec $c_j = L_{j,1} \vee L_{j,2} \vee L_{j,3}$, une disjonction d'exactement trois variables pour $j \in \{1, \ldots, k\}$. On fixe que $k \geq 1$. On oblige également le fait qu'il n'existe pas deux clauses identiques (modulo réordonnancement). Le dST2W M_φ construit prend en entrée les clauses de la formule φ pour produire un unique symbole $\mathbf{t}$. En outre, le dST2W M_φ accepte (plusieurs copies) des clauses triviales $x_i \vee \neg x_i$ et produit également un seul caractère $\mathbf{t}$. Pour empêcher la rétraction du caractère unique dans la règle initiale, la transformation définie par le transducteur fait également correspondre à une simple constante d le mot vide.

Pour rendre la sémantique, du transducteur construit, plus claire, nous utiliserons des symboles d'entrée proches des notations introduites précédemment. En particulier, les symboles ternaires $c_1, \ldots, c_k$ sont utilisées pour représenter les clauses avec pour constantes $x_1, \neg x_1, \ldots, x_n, \neg x_n$ (les $\neg x_i$ étant des constantes simples). Chaque close triviale $x_i \wedge \neg x_i$ est représentée trois fois à l'aide de symboles binaires $c_{i,1}$, $c_{i,2}$, et $c_{i,3}$. Formellement nous utilisons l'alphabet d'entrée $\Sigma = \Sigma^{(3)} \cup \Sigma^{(2)} \cup \Sigma^{(0)}$, tel que

$$\Sigma^{(3)} = \{c_1, \ldots, c_k\}, \quad \Sigma^{(2)} = \{c_{1,1}, c_{1,2}, c_{1,3}, \ldots, c_{n,1}, c_{n,2}, c_{n,3}\},$$
$$\Sigma^{(0)} = \{x_1, \neg x_1, \ldots, x_n, \neg x_n\} \cup \{d\}.$$

L'alphabet de sortie, quant à lui, contient exactement un symbole $\Delta = \{\mathbf{t}\}$. Pour simplifier la compréhension, nous présentons dans un premier temps un exemple de transformation reconnue par transducteur construit. Pour $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee \neg x_3)$, la transformation $\tau_{\varphi_0} = [\![M_{\varphi_0}]\!]$ est

$$\tau_{\varphi_0}(c_1(x_1, \neg x_2, x_3)) = \mathbf{t}, \qquad \tau_{\varphi_0}(c_2(x_2, x_4, \neg x_3)) = \mathbf{t}, \qquad \tau_{\varphi_0}(d) = \varepsilon,$$
$$\tau_{\varphi_0}(c_{i,\ell}(x_i, \neg x_i)) = \mathbf{t}, \quad \text{pour } i \in \{1,2,3,4\} \text{ et } \ell \in \{1,2,3\}.$$

Le transducteur $M_\varphi = (\Sigma, \Delta, Q, \mathit{init}, \mathit{rul})$ est construit de la manière suivante. Les états de M_φ sont

$$Q = \{q_0\} \cup \{q_{x_1}, \ldots, q_{x_n}\} \cup \{q_{\neg x_1}, \ldots, q_{\neg x_n}\}.$$

Sa règle initiale est $\mathit{init} = q_0$ et les transitions sont définies comme suit :

1. $q_0 \xrightarrow{c_j} \mathbf{t} \cdot q_{L_{j,1}} \cdot q_{L_{j,2}} \cdot q_{L_{j,3}}$ pour tout $1 \leq j \leq k$ avec $c_j = L_{j,1} \vee L_{j,2} \vee L_{j,3}$;

2. $q_0 \xrightarrow{c_{i,\ell}} \mathbf{t} \cdot q_{p_i} \cdot q_{\neg p_i}$ pour $1 \leq i \leq n$ et $1 \leq \ell \leq 3$;
3. $q_{x_i} \xrightarrow{x_i} \varepsilon$ et $q_{\neg x_i} \xrightarrow{\neg x_i} \varepsilon$ pour $1 \leq i \leq n$;
4. $q_0 \xrightarrow{d} \varepsilon$.

Le transducteur M_φ définit la transformation τ_φ satisfaisant exactement l'équation suivante (et pas d'autres) :

$$\begin{aligned}
&\tau_\varphi(c_j(L_{j,1}, L_{j,2}, L_{j,3})) = \mathbf{t}, && \text{pour } j \in \{1, \dots, k\},\\
&\tau_\varphi(c_{i,\ell}(x_i, \neg x_i)) = \mathbf{t}, && \text{pour } i \in \{1, \dots, n\} \text{ and } \ell \in \{1, 2, 3\},\\
&\tau_\varphi(d) = \varepsilon.
\end{aligned}$$

Nous pouvons calculer que la taille de M_φ est de $11n + 4k + 1$. La réduction repose principalement sur l'utilisation des clauses triviales $x_i \vee \neg x_i$ avec trois symboles distincts $c_{i,1}$, $c_{i,2}$, and $c_{i,3}$ permettant une représentation plus compacte de la transformation τ_φ s'l existe une valuation satisfaisant l'équation. La principale proposition est que

$$\varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}} \Leftrightarrow (M_\varphi, 11n + 4k + 1) \in \mathtt{MINIMIZE}_{\mathcal{STW}}.$$

($\Rightarrow$) : nous prenons la valuation $V : \{x_1, \dots, x_n\} \to \{\mathit{true}, \mathit{false}\}$ observant que $\varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}}$ et nous construisons un dST2W M_V obtenu à partir de M_φ en *poussant vers le bas* les symboles $\mathbf{t}$ sur les sous arbres correspondant aux littéraux évalués à vrai pour V. Formellement, le transducteur M_V diffère de M_φ seulement pour les règles :

1. $q_0 \xrightarrow{c_j} q_{L_{j,1}} \cdot q_{L_{j,2}} \cdot q_{L_{j,3}}$ pour tout $j \in \{1, \dots, k\}$ tel que $c_j = L_{j,1} \vee L_{j,2} \vee L_{j,3}$;
2. $q_0 \xrightarrow{c_{i,\ell}} q_{x_i} \cdot q_{\neg x_i}$ pour tout $1 \leq i \leq n$ et $1 \leq \ell \leq 3$;
3. $q_{x_i} \xrightarrow{x_i} = \mathbf{t}$ et $q_{\neg x_i} \xrightarrow{\neg x_i} = \varepsilon$ pour tout $1 \leq i \leq n$ tel que $V(x_i) = \mathit{true}$;
4. $q_{x_i} \xrightarrow{x_i} \varepsilon$ et $q_{\neg x_i} \xrightarrow{\neg x_i} \mathbf{t}$ pour tout $1 \leq i \leq n$ tel que $V(x_i) = \mathit{false}$;
5. $q_0 \xrightarrow{d} \varepsilon$.

M_φ et M_V sont équivalents puisque V est le témoin que $\varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}}$. De plus, la taille de M_V est exactement de $11n + 4k + 1$.

($\Leftarrow$) : considérons le dST2W $M = (\Sigma, \Delta, Q_M, \mathit{init}_M, \mathit{rul}_M)$ équivalent à M_φ dont la taille est bornée par $11n + 4k + 1$. Pour $\mathit{init}_M = w_0 \cdot q_0 \cdot w_1$, nous pouvons observer que w_0 et w_1 sont égaux à ε puisque $[\![M_\varphi]\!](d) = \varepsilon$. Nous pouvons également observer que M nécessite au moins $1 + 2n$ états puisque ce chiffre correspond à l'index de Myhill-Nerode pour le domaine $\mathit{dom}([\![M_\varphi]\!])$. Par conséquent, M a besoin d'au moins

1. 1 règle initiale de taille 1,
2. k règles d'arité 3 pour les symboles c_j ($j \in \{1, \dots, k\}$),

3. $3n$ règles d'arité 2 pour les symboles $c_{i,\ell}$ ($i \in \{1,\dots,n\}$ and $\ell \in \{1,2,3\}$),
4. n règles d'arité 0 pour les symboles x_i ($i \in \{1,\dots,n\}$),
5. n règles d'arité 0 pour les symboles $\neg x_i$ ($i \in \{1,\dots,n\}$), et
6. 1 règle d'arité 0 pour le symbole d.

En ignorant la taille des sorties possibles sur ces règles, leur taille globale est d'au moins $13n+4k+3$. Cela implique qu'il y a, au plus, n *marques* (symboles) $\mathbf{t}$ devant être distribuées sur les règles . On peut affirmer que les marqueurs sont *poussés vers le bas* sur les états pouvant capter les variables de manière à ce qu'ils soient uniquement associés aux états permettant une valuation positive.

Pour le reste de la preuve, nous notons $rul(q,f)$ la règles ayant q et f comme partie gauche, i.e. de la forme $q \xrightarrow{f} \_$.

Nous montrons dans un premier temps qu'effectivement n marqueurs exactement doivent être distribués sur les règles. Pour $i \in \{1,\dots,n\}$ et $\ell \in \{1,2,3\}$ nous définissons $q_{x_i,\ell} = \hat{rul}_M(q_0, c_{i,\ell}\cdot 1)$ et $q_{\neg x_i,\ell} = \hat{rul}_M(q_0, c_{i,\ell}\cdot 2)$. Nous définissons également, pour tout $i \in \{1,\dots,n\}$, R_i comme le sous ensemble des règles $rul_M(q_0, c_{i,\ell})$, $rul_M(q_{x_i,\ell}, x_i)$, et $rul_M(q_{\neg x_i,\ell}, \neg x_i)$ pour $\ell \in \{1,2,3\}$. Nous pouvons remarquer que, pour tout $i \in \{1,\dots,n\}$, les règles R_i utilisent au moins un $\mathbf{t}$. De plus, pour tout $\ell_1,\ell_2 \in \{1,2,3\}$ et tout $i_1,i_2 \in \{1,\dots,n\}$ tel que $i_1 \neq i_2$ nous avons $q_{\ell_1,i_1} \neq q_{\ell_2,i_2}$ ou le domaine M serait différent du domaine de τ_φ. Ainsi, pour deux $i_1 \neq i_2$, les ensembles des règles R_{i_1} et R_{i_2} sont disjoints et, par conséquent, au moins n marqueurs $\mathbf{t}$ différents sont utilisés sur ces règles.

Puisqu'au moins n marqueurs sont utilisés, le transducteur M a exactement $2n+1$ états incluant l'état initial q_0, et particulièrement, $q_{x_i,\ell_1} = q_{x_i,\ell_2}$ et $q_{\neg x_i,\ell_1} = q_{\neg x_i,\ell_2}$ pour tout $i \in \{1,\dots,n\}$ et $\ell \in \{1,2,3\}$ (sinon, une fois de plus, les domaines de M et τ_φ seraient différents). Nous pouvons donc simplifier la notation en retirant ℓ pour se ramener simplement q_{x_i} et $q_{\neg x_i}$. De manière similaire, on montre que $\hat{rul}_M(q_0, c_j\cdot\ell) = L_{i,\ell}$, pour tout $j \in \{1,\dots,k\}$ et tout $\ell \in \{1,2,3\}$, tel que $c_j = L_{j,1} \vee L_{j,2} \vee L_{j,3}$.

Nous pouvons maintenant montrer que les marqueurs $\mathbf{t}$ sont attribués uniquement sur les règles $rul_M(q_{x_i}, x_i)$ et $rul_M(q_{\neg x_i}, \neg x_i)$. De plus, on peut affirmer que, pour tout $1 \leq i \leq n$, exactement un $\mathbf{t}$ est attribué sur chaque règle $rul_M(q_{x_i}, x_i)$ ou $rul_M(q_{\neg x_i}, \neg x_i)$. Pour prouver cette affirmation, nous identifions l'ensemble R des $i \in \{1,\dots,n\}$ tel que $\mathbf{t}$ ne soit utilisé ni dans $rul_M(q_{x_i}, x_i)$, ni dans $rul_M(q_{\neg x_i}, \neg x_i)$:

$$R = \{i \mid 1 \leq i \leq n,\ rul_M(q_{x_i}, x_i) = \varepsilon,\ rul_M(q_{\neg x_i}, \neg x_i) = \varepsilon\}.$$

Nous pouvons remarquer que pour n'importe quel $i \notin R$, le fait que $\tau_\varphi(c_{i,1}(x_i, \neg x_i)) = \mathbf{t}$ implique qu'exactement une des règles $rul_M(q_{x_i}, x_i)$ et

$rul_M(q_{\neg x_i}, \neg x_i)$ a une occurrence de $\mathbf{t}$, ce qui utilise $n-|R|$ marqueurs. D'autre part, pour chaque $i \in R$ et chaque $\ell \in \{1,2,3\}$,chaque règle $rul_M(q_0, c_{i,\ell})$ utilise un marqueur $\mathbf{t}$, puisque $\tau_\varphi(c_{i,\ell}(x_i, \neg x_i)) = \mathbf{t}$. Il y a donc au total $(n-|R|)+3|R| = n + 2|R|$ marqueurs utilisés. Cela implique que R doit être vide ou trop de marqueurs seraient utilisés. Nous pouvons donc définir la valuation suivante V_M :

$$V_M(x_i) = \begin{cases} true & \text{si } \delta(q_{x_i}, x_i) = \mathbf{t}, \\ false & \text{si } \delta(q_{\neg x_i}, \neg x_i) = \mathbf{t}. \end{cases}$$

V_M vérifie φ puisque le transducteur M sur lequel il se base est équivalent à M_φ.

L'appartenance de $\texttt{MINIMISER}_{d\text{ST2W}}$ à NP est dû au fait que tester l'équivalence des dST2Ws est connu décidable en PTIME (cf chapitre précédent). Il est donc nécessaire d'avoir une machine de Turing non détérministe pour évaluer si un dST2W M' est de taille inférieure à $\min\{|M|, K\}$, et pour tester l'équivalence de M et M'. □

5.4 Normalisation

Les edST2Ws étant la seule forme minimisable efficacement, et qui sera donc notre cible dans un futur apprentissage, il est nécessaire de s'assurer que toute transformation définissable par un dST2W l'est aussi par un edST2W. Dans cette section, nous répondons à cette attente en présentant une méthode de normalisation efficace convertissant un dST2W en un unique edST2W équivalent. La normalisation ce divise en deux étapes : dans un premier temps, la conversion du dST2W en un edST2W, et ensuite sa minimisation. La minimisation, comme le montre la section précédente, est relativement simple. Le principal challenge de cette section reste le passage en version *travailleur* d'un dST2W. C'est pour cela que durant cette section il y aura souvent confusion entre la normalisation et la conversion entre un dST2W et un edST2W équivalent.

La normalisation d'un dST2W implique de changer la répartition de la sortie dans les règles de ce transducteur afin de satisfaire les conditions **(E1)** et **(E2)**, qui nous le rappelons, consistent à :

(E1) $lcp(im(\llbracket M \rrbracket_q)) = lcs(im(\llbracket M \rrbracket_q)) = \varepsilon$ pour tout $q \in Q$,

(E2) $lcp(im(\llbracket M \rrbracket_{q_0}) \cdot u_1) = \varepsilon$ pour l'axiome initial $u_0 \cdot q_0 \cdot u_1 \in init$ et pour

Ce processus traite essentiellement la sortie sans modifier la structure propre du transducteur. Par conséquent, nous commençerons par l'introduction de plusieurs notions et constructions inspirées par les conditions **(E1)** et **(E2)**, mais en nous plaçant dans le cadre plus simple des langages de mots. Nous

considérons uniquement les langages non vides, les images des états étant non vide pour les dST2Ws émondés auxquels on s'intéresse. Lorsque nous évaluons la faisabilité des constructions, nous considérerons uniquement des langages exprimables par des *CFGs*.

5.4.1 Réduction de langages

L'adaptation de **(E1)** revient à construire ce qu'on appelle une version *réduite* d'un langage, dans lequel tout préfixe, ou suffixe, commun à chacun des mots qui le constitue est réduit au mot vide. Un langage non vide $\mathcal{L}$ est *réduit* si et seulement si $lcp(\mathcal{L}) = \varepsilon$ et $lcs(\mathcal{L}) = \varepsilon$. Remarquons que le fait de travailler uniquement avec des langages non vides est essentiel ici. Prenons maintenant un langage non vide $\mathcal{L}$, possiblement non réduit. On divise $\mathcal{L}$ en trois partie, son *noyau*, noté $Noyau(\mathcal{L})$, et deux mots $Gauche(\mathcal{L})$ et $Droite(\mathcal{L})$ tel que $Noyau(\mathcal{L})$ soit réduit et :

$$\mathcal{L} = Gauche(\mathcal{L}) \cdot Noyau(\mathcal{L}) \cdot Droite(\mathcal{L}). \tag{5.2}$$

Nous pouvons observer différentes divisions possibles. Par exemple, $\mathcal{L} = \{a, aba\}$ peut être représenté par $\mathcal{L} = a \cdot \{\varepsilon, ba\} \cdot \varepsilon$ et $\mathcal{L} = \varepsilon \cdot \{\varepsilon, ab\} \cdot a$. Nous supprimons l'ambiguïté en choisissant la première possibilité, étant la seule compatible avec la condition **(E2)** cherchant à *pousser vers la gauche*. Formellement, $Gauche(\mathcal{L}) = lcp(\mathcal{L})$ et $Droite(\mathcal{L}) = lcs(Gauche(\mathcal{L})^{-1} \cdot \mathcal{L}')$. Le noyau, $Noyau(\mathcal{L})$, quant à lui, est obtenu en suivant l'équation 5.2. Par exemple, la division en forme réduite de $\mathcal{L}_{q_0} = im([\![M]\!]_{q_0}) = (abc)^* \cdot ac \cdot (abc)^*$ de l' exemple 38 est $Gauche(\mathcal{L}_{q_0}) = a$, $Droite(\mathcal{L}_{q_0}) = c$, et $Noyau(\mathcal{L}_{q_0}) = (bca)^*(cba)^*$.

Nous pouvons trouver dans la littérature que si $\mathcal{L}$ est un langage exprimable par une *CFG*, alors $lcp(\mathcal{L})$ (et $lcs(\mathcal{L})$) peuvent être exponentielles dans la taille de la grammaire définissant $\mathcal{L}$ (cf. exemple 43). Par conséquent, la borne minimale pour calculer le noyau est exponentielle. Nous pouvons cependant remarquer que la décomposition produite peut être calculée en temps polynomial dans $|Gauche(\mathcal{L})| + |Droite(\mathcal{L})|$. De plus, il est connu que les mots $Gauche(\mathcal{L})$ et $Droite(\mathcal{L})$ sont représentables à l'aide de grammaire singleton de taille polynomiale dans la taille de la grammaire définissant $\mathcal{L}$ (cf. Karhumäki et Plandowski (1994)), Il faut cependant vérifier que la construction se fait également en temps polynomial.

5.4.2 Faire traverser un mot dans un langage

Dans cette sous-section nous travaillerons uniquement avec des langages *non vides* et *réduits*.

La condition **(E2)** introduit le problème de devoir faire passer des mots, la production associée à une règle, à travers des langages, les transformations associées aux états de cette même règle. Pour illustrer cela, prenons le langage $\mathcal{L} = \{\varepsilon, a, aa, aaab\}$ et un mot $w = aab$, qui ensemble donne le langage $\mathcal{L} \cdot w = \{aab, aaab, aaaab, aaabaab\}$. Le but est d'identifier le plus long préfixe v de w tel que $\mathcal{L} \cdot w = v \cdot \mathcal{L}' \cdot u$, avec $w = v \cdot u$ et $\mathcal{L}'$ un langage dérivé de $\mathcal{L}$. L'intuition est de chercher à pousser le mot w (en partie) vers l'avant, i.e. de droite à gauche, à travers le langage $\mathcal{L}$. Dans l'exemple précédent la solution est clairement $v = aa$, $\mathcal{L}' = \{\varepsilon, a, aa, abaa\}$, et $u = b$ ($\mathcal{L}'$ étant ici différent de $\mathcal{L}$). Dans cette sous-section , nous montrerons que ce procédé est toujours possible pour les *CFGs* et qu'il peut se faire de manière constructive.

Le résultat d'une telle opération pour un mot w à travers un langage $\mathcal{L}$ peut être identifié par trois mots :

- $pre(\mathcal{L}, w)$, le plus long préfixe de w pouvant être poussé à travers $\mathcal{L}$,
- $reste(\mathcal{L}, w)$, la partie de w ne pouvant pas être poussée à travers le langage,
- $report(\mathcal{L}, w)$, un mot permettant de connaître comment obtenir $\mathcal{L}'$ à partir de $\mathcal{L}$.

Nous pouvons identifier trois classes de langage réagissant de manière différentes à cette opération de transfert et qui déterminent la manière dont le mot pourra, ou non, être poussé.

La première classe contient le langage *trivial* $\mathcal{L} = \{\varepsilon\}$. Cela correspond aux états ne produisant rien. Ce langage permet à tout mot d'être poussé à travers et n'est jamais modifié par ce procédé.

La seconde classe regroupe les langages *périodiques* non triviaux, classe essentiellement composée des langages contenus dans la clôture de Kleene pour certains mots appelés *période*. Pour illustrer cette classe, reprenons l'exemple 38. Le langage $\mathcal{L}_{q_1} = im(\llbracket M_1 \rrbracket_{q_1}) = (abc)^* = \{\varepsilon, abc, abcabc, \ldots\}$ et a pour période abc. Les langages périodiques permettent de pousser des mots composés d'une répétition de la période suivie d'un préfixe de cette période. Par exemple, pour $w_1 = abcabcaba$, nous obtenons que $pre(\mathcal{L}_{q_1}, w_1) = abcabcab$ et $reste(\mathcal{L}_{q_1}, w_1) = a$. Le report, pour cette classe de langages, est le préfixe de cette période, situé après la répétition, soit $report(L_{q_1}, w_1) = ab$.

La troisième classe contient l'ensemble des langages restants, i.e. les langages non périodiques. Nous montrons que pour tout langage de cette classe, il existe un plus long mot pouvant être poussé à travers le langage. De plus, nous pouvons remarquer que tout mot pouvant être poussé à travers ce langage est préfixe de ce mot. Par exemple, pour le langage $\mathcal{L}_3 = \{\varepsilon, a, aab\}$, aa est le plus long mot pouvant être poussé à travers ce langage. En choisissant $w_2 = ab$, nous obtenons que $pre(\mathcal{L}_3, w_2) = a$ et $reste(L_3, w_2) = b$. Le langage n'étant pas périodique, $report(L_3, w_2) = pre(L_3, w_1) = a$, qui est bien préfixe de aa. Il

est à noter que la classe contient aussi les langages n'autorisant aucun mot à être poussé à travers eux, ces langages devant contenir au moins deux mots non vides différents dès la première lettre. Nous pouvons remarquer qu'aucun préfixe commun n'existe pour ces langages.

Nous pouvons maintenant définir formellement l'opération permettant de pousser un mot à travers un langage. Dans un premier temps nous définissons, pour tout langage $\mathcal{L} \subseteq \Delta^*$, l'ensemble des mots pouvant être complètement poussés à travers $\mathcal{L}$:

$$\mathit{trousseau}(L) = \{w \in \Delta^* \mid w \text{ est un préfixe commun de } L \cdot w\}.$$

Par exemple, $\mathit{trousseau}(\mathcal{L}_{q_1}) = \{\varepsilon, a, aa\}$ et $\mathit{trousseau}(\mathcal{L}_{q_0}) = (abc)^* \cdot \{\varepsilon, a, ab\}$. On peut remarquer que $\mathit{trousseau}(\{\varepsilon\}) = \Delta^*$, comme nous l'avions souligné pour le langage trivial, et que cet ensemble contient au moins le mot vide ε puisque les langages étudiés sont supposés non vides.

Il est à noter que si $a \cdot w \in \mathit{trousseau}(\mathcal{L})$, alors "$a$" est la première lettre de tout mot non vide de $\mathcal{L}$. De plus, $\mathcal{L}$ étant supposé réduit, $\mathcal{L}$ doit contenir le mot vide, sinon $\mathit{lcp}(\mathcal{L})$ serait différent de ε. Il est important de souligner la proposition suivante :

Proposition 22. *Si $\mathcal{L}$ est réduit et non trivial, alors $\mathit{trousseau}(\mathcal{L})$ est clos par préfixe et totalement ordonné par la relation de préfixe.*

Preuve. Montrer que $\mathit{trousseau}(\mathcal{L})$ est clos par préfixe découle directement de la définition. Considérons n'importe quel $w \in \mathit{trousseau}(\mathcal{L})$, et un préfixe w' de w ayant pour longueur $|w'| = k$. Choisissons maintenant un mot $v \in \mathcal{L}$, par définition w est un préfixe de $v \cdot w$. Puisque w' est un préfixe de w, alors $v \cdot w'$ est un préfixe de $v \cdot w$, d'ailleurs $v \cdot w'$ est de longueur au moins k. Cela implique que w' est un préfixe de $v \cdot w'$.

Nous montrons que $\mathit{trousseau}(L)$ est ordonné grâce à une simple induction sur la taille des mots de $\mathit{trousseau}(L)$. Prenons deux mots $w \cdot a, w \cdot b \in \mathit{trousseau}(L)$. L étant non trivial, il existe un mot non vide $u \in L$. Par définition, $w \cdot a$ est un préfixe de $u \cdot w \cdot a$, et $w \cdot b$ est un préfixe de $u \cdot w \cdot b$. La taille de u étant d'au moins 1 symbole, $w \cdot a$ et $w \cdot b$ sont tous deux préfixes de $u \cdot w$, ce qui implique que $a = b$. □

À présent, nous nous intéressons aux langages périodiques et à leur définition (cf. Lothaire (1997)).

Définition 46. *Un langage $\mathcal{L} \subseteq \Delta^*$ est* périodique *si et seulement s'il existe un mot non vide $v \in \Delta^*$, appelé la* période *de $\mathcal{L}$, tel que $\mathcal{L} \subseteq v^*$.*

Un mot w est *primitif* s'il ne peut être construit par la répétition d'un autre mot, i.e. il n'existe pas de mot $v \in \Delta^*$ ni d'entier $n \geq 0$ tels que $w = v^n$.

On rappelle, d'après Lothaire (1997), que tout langage non trivial périodique $\mathcal{L}$ a une unique période primitive, que l'on notera $periode(\mathcal{L})$. Par exemple, le langage $\mathcal{L} = \{\varepsilon, abab, abababab\}$ est périodique et a pour primitive $periode(\mathcal{L}) = ab$. Le mot $abab$ est aussi sa période, mais pas sa primitive.

Dans la proposition suivante nous présentons une caractérisation alternative des langages périodiques qui sera utile pour la suite.

Proposition 23. *Un langage $\mathcal{L}$ est périodique si et seulement si toute paire de mots commute, i.e. $w_1 \cdot w_2 = w_2 \cdot w_1$ pour tout $w_1, w_2 \in L$.*

Le résultat précédent est une conséquence directe de faits connus sur les langages périodiques (Lothaire, 1997, Proposition 1.3.2). Nous allons maintenant, par le biais de l'identification des mots pouvant être poussés à travers un langage, donner une nouvelle caractérisation de cette classe importante pour le problème que nous cherchons à traiter.

Proposition 24. *Soit $\mathcal{L}$ un langage non trivial réduit, $trousseau(\mathcal{L})$ est infini si et seulement si $\mathcal{L}$ est périodique. De plus, si $\mathcal{L}$ est périodique alors $trousseau(L) = periode(L)^* \cdot prefixe(periode(L))$.*

Preuve. ($\Rightarrow$), considérons un langage non trivial $\mathcal{L} \subseteq w^*$, nous avons donc $w^k \in trousseau(\mathcal{L})$, pour tout $k \geq 0$. De plus, d'après la proposition 22, nous savons que $trousseau(\mathcal{L}) = w^* \cdot prefixe(w)$.

($\Leftarrow$), rappelons que la proposition 23 montre que les langages périodiques non triviaux correspondent exactement à la famille de ceux permettant la commutation de leurs éléments, i.e. un langage non trivial $\mathcal{L}$ est périodique si et seulement si pour tout mot $w_1, w_2 \in \mathcal{L}$, $w_1 \cdot w_2 = w_2 \cdot w_1$.

Premièrement, nous pouvons observer que $\varepsilon \in \mathcal{L}$. Cela vient du fait que $trousseau(\mathcal{L})$ contient un mot non vide $a \cdot w \in trousseau(\mathcal{L})$, ce qui implique que a est la première lettre de tout mot non vide de $\mathcal{L}$. $\mathcal{L}$ étant réduit, $\mathcal{L}$ doit contenir le mot vide, sinon $lcp(\mathcal{L})$ ne pourrait pas être ε.

Ensuite, prenons n'importe quels mots $w_1, w_2 \in \mathcal{L}$. $trousseau(\mathcal{L})$ étant infini, il existe un mot $v \in trousseau(\mathcal{L})$ ayant une longueur supérieur à $|w_1|+|w_2|$. De plus, on remarque que v est un préfixe de v, $w_1 \cdot v$, et $w_2 \cdot v$. Cela nous permet d'inférer que $v = w_1 \cdot v'$ et $v = w_2 \cdot v''$, ce qui implique que $w_1 \cdot v = w_1 \cdot w_2 \cdot v''$ et $w_2 \cdot v = w_2 \cdot w_1 \cdot v'$. Puisque la longueur de v est supérieur à celle de $|w_1|+|w_2|$, les deux précédents mots correspondent aux premières $|w_1|+|w_2|$ lettres ce qui nous donne $w_1 \cdot w_2 = w_2 \cdot w_1$. □

Ce résultat et les différentes observations précédemment faites permettent d'identifier trois classes pertinentes de langages permettant de caractériser $trousseau(\mathcal{L})$ pour tout langage $\mathcal{L}$.

0^o $\mathcal{L} = \{\varepsilon\}$ (langage trivial) alors $trousseau(\mathcal{L}) = \Delta^*$,

1^o $\mathcal{L}$ est un langage périodique, non trivial ($\mathcal{L} \neq \{\varepsilon\}$), alors $\mathit{trousseau}(\mathcal{L}) = \mathit{periode}(\mathcal{L})^* \cdot \mathit{prefixe}(\mathit{periode}(\mathcal{L}))$.

2^o Pour $\mathcal{L}$ un langage non périodique, alors $\mathit{trousseau}(\mathcal{L}) = \mathit{prefixe}(v)$ quelque soit $v \in \mathit{trousseau}(\mathcal{L})$.

Supposons maintenant que nous souhaitons *pousser* un mot $w \in \Delta^*$ *à travers* un langage réduit $L \subseteq \Delta^*$. Identifions par s le plus long préfixe de w présent dans $\mathit{trousseau}(L)$, et r le reste, de sorte que $w = s \cdot r$. Remarquons que $\mathit{trousseau}(L)$ étant clos par préfixe, la définition de s est non-ambigüe. Définissons $\mathit{pre}(\mathcal{L}, w)$, $\mathit{reste}(\mathcal{L}, w)$, and $\mathit{report}(\mathcal{L}, w)$ selon la classe à laquelle appartient $\mathcal{L}$:

0^o $\mathcal{L} = \{\varepsilon\}$: $\mathit{pre}(\mathcal{L}, w) = w$, $\mathit{reste}(\mathcal{L}, w) = \varepsilon$, et $\mathit{report}(\mathcal{L}, w) = \varepsilon$.

1^o $\mathcal{L}$ est périodique et non trivial : $s = \mathit{periode}(\mathcal{L})^k \cdot o$ avec o un préfixe (maximal) de $\mathit{periode}(\mathcal{L})$, et on assigne $\mathit{pre}(\mathcal{L}, w) = s$, $\mathit{reste}(\mathcal{L}, w) = r$, et $\mathit{report}(\mathcal{L}, w) = o$.

2^o $\mathcal{L}$ est non périodique : $\mathit{pre}(\mathcal{L}, w) = s$, $\mathit{reste}(\mathcal{L}, w) = r$, et $\mathit{report}(\mathcal{L}, w) = s$.

Les reports jouent un rôle central pour la normalisation de la sortie, indiquant quelle partie du mot poussé doit se répercuter sur les règles. Cette normalisation est rendue possible grâce au résultat suivant.

Proposition 25. *L'ensemble* $\mathit{Reports}(\mathcal{L}) = \{\mathit{report}(\mathcal{L}, w) \mid w \in \Delta^*\}$ *est fini pour tout langage réduit* $\mathcal{L}$*. De plus, si* $\mathcal{L}$ *est défini par une CFG* G*, alors la taille de* $\mathit{Reports}(L)$*est au plus double-exponentielle dans la taille de* G *et* $\mathit{Reports}(L)$ *peut être construit en temps polynomial dans sa taille.*

Preuve. Le fait que l'ensemble $\mathit{Reports}(\mathcal{L})$ soit fini découle directement des définitions des trois classes de langages. Nous montrons juste que cet ensemble peut être construit en temps double-exponentiel dans la taille de la *CFG* G définissant $\mathcal{L}$. L'algorithme décrit si-dessous permet de également de classifier le langage $\mathcal{L}$ dans la classe à laquelle il appartient.

Pour le cas 0^o, on remarque que la condition $\mathcal{L} = \{\varepsilon\}$ peut être facilement contrôlée pour G en testant que tout non-terminal (atteignable) de G ne produit aucun mot non-vide. Cela peut se faire par un simple algorithme de clôture en temps polynomial sur la taille de G.

On traite les deux autres cas en même temps. On note $w_{\min}$ le plus petit mot non vide de $\mathcal{L}$. Tout d'abord, nous pouvons observer que $\mathcal{L}$ est périodique si et seulement si sa période primitive $\mathit{periode}(\mathcal{L})$ est aussi la période primitive de $w_{\min}$. Soit v le plus court préfixe de $w_{\min}$ tel que $w_{\min} = v^k$ pour $k > 0$ (possiblement $w_{\min}$ lui même). On peut facilement se persuader que $\mathit{periode}(\{w_{\min}\})$ est le plus court préfixe v de $w_{\min}$ tel que $w_{\min} = v^k$ avec $k > 0$. Par conséquent, $\mathcal{L}$ est périodique si et seulement si $\mathcal{L} \subseteq v^*$.

Le mot $w_{\min}$ peut être de taille exponentielle dans la taille de G (cf. exemple 43), ce qui peut également être la taille de v. Tester l'inclusion $\mathcal{L} \subseteq v^*$, quand il existe une *CFG* G définissant $\mathcal{L}$, peut être effectué en utilisant les techniques classiques des automates : On construit un automate descendant A_G définissant $\mathcal{L}$, un DFA A_v définissant v^*, et son complément $A_v^{\complement}$ définissant $\Delta^* \setminus v^*$. Il suffit maintenant de tester le vide du produit $P = A_G \times A_v^{\complement}$. Nous avons clairement que $\mathcal{L} \subseteq v^*$si et seulement si P définit un langage vide. Pour ce qui est de la compléxité, nous pouvons déjà remarquer que la taille de A_G est en $poly(|G|)$, les tailles de A_v et $A_v^{\complement}$ sont en $poly(|v|) = exp(|G|)$, et par conséquent l'automate P est de taille $exp(|G|)$.

Si $\mathcal{L}$ est périodique, alors $O = prefixe(periode(\mathcal{L})) = prefixe(v)$ et sa taille est en simple exponentielle dans la taille de G. Si $\mathcal{L}$ n'est pas périodique, alors le test décrit précédemment échoue, i.e. P est non vide et accepte un mot non vide w_0. P étant un automate descendant de taille $exp(|G|)$, le mot le plus court pouvant être reconnu par P est possiblement de taille double exponentielle dans la taille de G. On peut affirmer que si un mot peut être poussé à travers $\mathcal{L}$, alors il ne peut pas avoir une longueur supérieure à w_0. La preuve est combinatoire et sera omise ici. □

5.4.3 Déplacement de gauche à droite

Jusqu'à maintenant nous nous sommes uniquement intéressés au fait de pousser un mot à travers un langage de droite à gauche, la forme normale nécessitant que la sortie soit produite le plus tôt possible (le plus à gauche). Cependant, comme l'illustre l'exemple 38, il est parfois nécessaire de pousser les mots dans le sens inverse, de gauche à droite. En effet, si l'on considère l'état q_1 dans la règle $q_0 \xrightarrow{f} q_1 \cdot ac \cdot q_1$,nous voyons assez aisément que manipuler un mot (ici le "c") dans le sens opposé est nécessaire. Ces deux processus sont interdépendants mais, au lieu de chercher à les appliquer parallèlement, nous présentons une extension du monoïde libre Δ^* dans un pré-groupe (ou groupoïde) $\mathbb{G}_\Delta$. Cela permet de manipuler ces deux opérations d'une manière unifiée afin de simplifier l'algorithme de normalisation de la production.

Un *pré-groupe de mots sur* Δ est l'ensemble $\mathbb{G}_\Delta = \Delta^* \cup \{w^{-1} \mid w \in \Delta^+\}$, tel que w^{-1} est un terme représentant l'inverse d'un mot non vide w. Cet ensemble est associé à deux opérateurs, l'opérateur unaire d'inversion : $(w)^{-1} = w^{-1}$, $\varepsilon^{-1} = \varepsilon$, et $(w^{-1})^{-1} = w$ pout tout $w \in \Delta^*$, et l'opérateur partiel étendant l'opérateur de concaténation, satisfaisant les équations suivantes : $w^{-1} \cdot w = \varepsilon$ et $w \cdot w^{-1} = \varepsilon$ pour tout $w \in \Delta^*$, et $v^{-1} \cdot u^{-1} = (uv)^{-1}$ pout tout $u, v \in \Delta^*$. Dans la suite, nous utiliserons $w, u, v, \ldots$ pour représenter les mots sur Δ^* uniquement, et $z, z_1, \ldots$ pour représenter les éléments de $\mathbb{G}_\Delta$. Nous pouvons remarquer que certaines expressions nécessitent d'être évaluées avec précaution, comme

$ab \cdot (cb)^{-1} \cdot cd = ab \cdot b^{-1} \cdot c^{-1} \cdot cd = ad$, et que d'autres ne peuvent être évaluées, par exemple $ab \cdot a^{-1}$. De plus, certaines expressions peuvent être indéfinies sauf si nous appliquons proprement les différents axiomes. Cela est illustré par l'exemple précédent, ou encore par l'équation $ab \cdot (cb)^{-1} \cdot cd$ pouvant être évaluée comme $ab \cdot [(cb)^{-1} \cdot cd] = ab \cdot [b^{-1} \cdot d]$, ce qui n'est pas défini, mais peut aussi être évaluée par $ab \cdot b^{-1} \cdot c^{-1} \cdot cd = ad$.

Revenons dès à présent sur l'opération consistant à pousser un mot w de gauche à droite (dans le sens «inverse» à celui considéré jusqu'à présent) à travers un langage $\mathcal{L}$, ce qui peut se résumer à trouver un découpage $u \cdot v = w$ et un langage $\mathcal{L}'$ tels que $w \cdot L = u \cdot L' \cdot v$. Nous pouvons voir cette opération comme le fait de pousser l'inverse du mot w^{-1} à travers $\mathcal{L}$, i.e. on essaye de trouver $u \cdot v = w$ tel que $\mathcal{L} \cdot w^{-1} = v^{-1} \cdot \mathcal{L}' \cdot u^{-1}$, puisque dans ce cas la, $L \cdot v^{-1} = v^{-1} \cdot L'$, ce qui implique que $w \cdot L = (u \cdot v) \cdot (v^{-1} \cdot L' \cdot v) = u \cdot L' \cdot v$.

Pour bien visualiser le fait de pousser vers l'arrière un mot, nous utilisons une autre vue basée sur l'opération classique d'inversion d'un mot, par exemple $(abc)^{\mathrm{rev}} = cba$. Á proprement parler, pousser un mot w à travers $\mathcal{L}$ dans le sens inverse revient essentiellement à pousser w^{rev} à travers $\mathcal{L}^{\mathrm{rev}}$ puisque $(w \cdot \mathcal{L})^{\mathrm{rev}} = \mathcal{L}^{\mathrm{rev}} \cdot w^{\mathrm{rev}}$, et que si $\mathcal{L}^{\mathrm{rev}} \cdot w^{\mathrm{rev}} = v_0 \cdot \mathcal{L}_0 \cdot u_0$, alors $w \cdot \mathcal{L} = u_0^{\mathrm{rev}} \cdot \mathcal{L}_0^{\mathrm{rev}} \cdot v_0^{\mathrm{rev}}$. Nous définissons donc :

$$\begin{aligned} pre(\mathcal{L}, w^{-1}) &= (pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}})^{-1}, \\ reste(\mathcal{L}, w^{-1}) &= (reste(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}})^{-1}, \\ report(\mathcal{L}, w^{-1}) &= (report(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}})^{-1}. \end{aligned}$$

La principale équation permettant de pousser des mots à travers des langages (sans restriction sur le sens des opérations) est : pour tout langage non vide $\mathcal{L}$ et $z \in \mathbb{G}_\Delta$

$$\mathcal{L} \cdot z = pre(\mathcal{L}, z) \cdot (report(\mathcal{L}, z)^{-1} \cdot \mathcal{L} \cdot report(\mathcal{L}, z)) \cdot reste(\mathcal{L}, z).$$

Puisque la procédure de normalisation de sortie travaille sur les dST2Ws, et non sur des langages, pour prouver sa justesse nous avons besoin d'une déclaration plus forte permettant de traiter indépendamment chaque mot du langage.

Proposition 26. *Soit $\mathcal{L} \subseteq \Delta^*$ un langage réduit non vide et $z \in \mathbb{G}_\Delta$, pour tout mot $u \in \mathcal{L}$*

$$u \cdot z = pre(\mathcal{L}, z) \cdot (report(\mathcal{L}, z)^{-1} \cdot u \cdot report(\mathcal{L}, z)) \cdot reste(\mathcal{L}, z).$$

Preuve. Nous distinguons une fois de plus les trois différents cas, à savoir $\mathcal{L}$ est trivial, $\mathcal{L}$ est périodique et non trivial, ou $\mathcal{L}$ est non périodique.

Si $\mathcal{L}$ est trivial, $pre(\mathcal{L}, z) = reste(\mathcal{L}, z) = report(\mathcal{L}, z) = \varepsilon$ et la proposition est vérifiée.

Si $\mathcal{L}$ est périodique et non trivial. Considérons dans un premier temps le cas où $z = w \in \Delta^*$. Soit $v = periode(\mathcal{L})$, cela nous donne que :

$$\begin{aligned} trousseau(\mathcal{L}, w) &= v^* \cdot prefixe(p) \\ pre(\mathcal{L}, w) &= v^i \cdot o \text{ pour un } i > 0 \text{ et un } o \in prefixe(v) \\ report(\mathcal{L}, w) &= ((v^i \cdot o) \mod v) = o \end{aligned}$$

Rappelons nous que $pre(\mathcal{L}, w)$ est un préfixe de w et que tout préfixe u d'un mot w' vérifie que $u \cdot (u^{-1} \cdot w') = w'$. Tout mot u de $\mathcal{L}$ est de la forme v^k avec $k > 0$, et :

$$\begin{aligned} pre(\mathcal{L}, w) \cdot (report(\mathcal{L}, w)^{-1} \cdot u \cdot report(\mathcal{L}, w)) \cdot reste(\mathcal{L}, w) \\ = v^i \cdot o \cdot (o^{-1} \cdot v^k \cdot o) \cdot (v^i \cdot o)^{-1} \cdot w \\ = v^k \cdot v^i \cdot o \cdot (v^i \cdot o)^{-1} \cdot w \\ = v^k \cdot w = u \cdot w \end{aligned}$$

Considérons maintenant le cas où $w \in \Delta^*$. Soit $v^{\text{rev}} = periode(\mathcal{L}^{\text{rev}})$, alors :

$$\begin{aligned} trousseau(\mathcal{L}^{\text{rev}}, w^{-1}) &= (v^{\text{rev}})^* \cdot prefixe(v^{\text{rev}}) \\ pre(\mathcal{L}^{\text{rev}}, w^{-1}) &= (v^{\text{rev}})^i \cdot o \text{ pour } i > 0 \text{ et } o \in prefixe(v^{\text{rev}}) \\ report(\mathcal{L}^{\text{rev}}, w^{-1}) &= ((v^{\text{rev}})^i \cdot o) \mod (v^{\text{rev}}) = o. \end{aligned}$$

Soit $u = p^k \in \mathcal{L}$,

$$\begin{aligned} & pre(\mathcal{L}, w^{-1}) \cdot \left(report(\mathcal{L}, w^{-1})^{-1} \cdot u \cdot report(\mathcal{L}, w^{-1})\right) \cdot reste(\mathcal{L}, w^{-1}) \\ &= (pre(\mathcal{L}^{\text{rev}}, w^{\text{rev}})^{\text{rev}})^{-1} \cdot \left(report(\mathcal{L}^{\text{rev}}, w^{\text{rev}})^{\text{rev}} \cdot u \cdot (report(\mathcal{L}^{\text{rev}}, w^{\text{rev}})^{\text{rev}})^{-1}\right) \\ &\quad \cdot (reste(\mathcal{L}^{\text{rev}}, w^{\text{rev}})^{\text{rev}})^{-1} \\ &= (o^{\text{rev}} \cdot v^i)^{-1} \cdot \left(o^{\text{rev}} \cdot v^k \cdot (o^{\text{rev}})^{-1}\right) \cdot \left(\left(((v^{\text{rev}})^i \cdot o)^{-1} \cdot w^{\text{rev}})\right)^{\text{rev}}\right)^{-1} \end{aligned}$$

Rappelons pour tout u et w' de Δ^* que $(u^{-1} \cdot w')^{\text{rev}} = (w')^{\text{rev}} \cdot (u^{\text{rev}})^{-1}$ et $(u \cdot (w')^{-1})^{-1} = w' \cdot u^{-1}$. Par conséquent on obtient que :

$$\begin{aligned} \left(\left(((v^{\text{rev}})^i \cdot o)^{-1} \cdot w^{\text{rev}})\right)^{\text{rev}}\right)^{-1} &= \left((w^{\text{rev}})^{\text{rev}} \cdot \left(((v^{\text{rev}})^i \cdot o)^{\text{rev}}\right)^{-1}\right)^{-1} \\ &= \left((v^{\text{rev}})^i \cdot o\right)^{\text{rev}} \cdot w^{-1} \\ &= o^{\text{rev}} \cdot \left((v^{\text{rev}})^i\right)^{\text{rev}} \cdot w^{-1} \\ &= o^{\text{rev}} \cdot v^i \cdot w^{-1} \end{aligned}$$

Donc,

$$\begin{aligned}
&pre(L, w^{-1}) \cdot \big(report(L, w^{-1})^{-1} \cdot u \cdot report(L, w^{-1})\big) \cdot reste(L, w^{-1}) \\
&= (o^{\mathrm{rev}} \cdot v^i)^{-1} \cdot \big(o^{\mathrm{rev}} \cdot v^k \cdot (o^{\mathrm{rev}})^{-1}\big) \cdot o^{\mathrm{rev}} \cdot v^i \cdot w^{-1} \\
&= \big(v^{k-i} \cdot (o^{\mathrm{rev}})^{-1}\big) \cdot o^{\mathrm{rev}} \cdot v^i \cdot w^{-1}
\end{aligned}$$

Puisque o^{rev} est un suffixe de v, on obtient que $v^k \cdot w^{-1} = u \cdot w^{-1}$. Le lemme est donc vérifié.

Considérons maintenant le cas où $\mathcal{L}$ est non périodique et le cas où $z = w \in \Delta^*$. Alors, il existe un mot s tel que $trousseau(\mathcal{L}) = prefixe(s)$. Notons que s est aussi égal à $noyau(\mathcal{L})$. Dans ce cas, $pre(\mathcal{L}, w) = lcp(\{w, s\})$, ce qui implique que $pre(\mathcal{L}, w)$ est un préfixe de s. Nous obtenons que $report(\mathcal{L}, w) = pre(\mathcal{L}, w)$, et par conséquent que pour tout $u \in \mathcal{L}$ nous avons $pre(\mathcal{L}, w) \cdot (report(\mathcal{L}, w)^{-1} \cdot u \cdot report(\mathcal{L}, w)) \cdot reste(\mathcal{L}, w) = pre(\mathcal{L}, w) \cdot (pre(\mathcal{L}, w)^{-1} \cdot u \cdot pre(\mathcal{L}, w)) \cdot pre(\mathcal{L}, w)^{-1} \cdot w$. $pre(\mathcal{L}, w)$ est dans $trousseau(\mathcal{L})$, et d'après la définition de *trousseau* on peut remarquer que $pre(\mathcal{L}, w)$ est un préfixe de $u \cdot pre(\mathcal{L}, w)$. De plus, $pre(\mathcal{L}, w)$ est un préfixe de w. On obtient donc que $pre(\mathcal{L}, w) \cdot (pre(\mathcal{L}, w)^{-1} \cdot u \cdot pre(\mathcal{L}, w)) \cdot pre(\mathcal{L}, w)^{-1} \cdot w = u \cdot w$.

Le cas où $z = w^{-1}$ et $w \in \Delta^*$ est similaire. Il existe un mot $s = noyau(\mathcal{L}^{\mathrm{rev}})$ tel que :

$$trousseau(\mathcal{L}^{\mathrm{rev}}) = prefixe(s), \text{ et } pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}}) = lcp(\{w, s\}) = report(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}}).$$

Par conséquent, pour tout $u \in \mathcal{L}$ nous vérifions que :

$$\begin{aligned}
&pre(\mathcal{L}, w^{-1}) \cdot (report(\mathcal{L}, w^{-1})^{-1} \cdot u \cdot report(\mathcal{L}, w^{-1})) \cdot reste(\mathcal{L}, w^{-1}) \\
&= (pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}})^{-1} \cdot \big(pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}} \cdot u \cdot (pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}})^{-1}\big) \\
&\qquad \cdot \big((pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{-1} \cdot w^{\mathrm{rev}})^{\mathrm{rev}}\big)^{-1} \\
&= \big(u \cdot (pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}})^{-1}\big) \cdot pre(\mathcal{L}^{\mathrm{rev}}, w^{\mathrm{rev}})^{\mathrm{rev}} \cdot w^{-1} \\
&= u \cdot w^{-1}
\end{aligned}$$

□

5.4.4 Algorithme de normalisation

Nous cherchons maintenant à définir la normalisation d'un dST2W M. Pour cela on fixe $M = (\Sigma, \Delta, Q, init, rul)$ et on introduit les macros suivantes :

$$\mathcal{L}_q = im(\llbracket M \rrbracket_q), \quad \mathcal{L}_q^\circ = Noyau(\mathcal{L}_q), \quad Gauche(q) = Gauche(\mathcal{L}_q), \quad Droite(q) = Droite(\mathcal{L}_q),$$
$$pre(q, z) = pre(\mathcal{L}_q^\circ, z), \quad report(q, z) = report(\mathcal{L}_q^\circ, z), \quad reste(q, z) = reste(\mathcal{L}_q^\circ, z).$$

De même, notons $Reports(q) = \{report(q,z) \mid z \in \mathbb{G}_\Delta\}$, qui d'après la proposition 25, est connu fini. Le dST2W $M' = (\Sigma, \Delta, Q', init', rul')$ que nous sommes en train de construire a pour états :

$$Q' = \{\langle q, z\rangle \mid q \in Q,\ z \in Reports(q)\}.$$

Notre construction assure que $[\![M]\!] = [\![M']\!]$ et que pour tout état $q \in Q$, tout $z \in Reports(q)$, et tout $t \in dom([\![M]\!]_q)$:

$$[\![M']\!]_{\langle q,z\rangle}(t) = z^{-1} \cdot Gauche(q)^{-1} \cdot [\![M]\!]_q(t) \cdot Droite(q)^{-1} \cdot z$$

Chaque report à effectuer étant associé à l'état correspondant et connaissant la formule permettant de calculer la production restante, il reste à tirer la sortie le plus en haut et à gauche possible et de propager les modifications en partant de l'axiome initial jusqu'à stabilisation du nombre d'états. L'algorithme de normalisation complet d'un dST2W est présenté si dessous.

Algorithme 1: Normalisation d'un dST2W.

```
1  normaliser(M)
   Entrées : M = (Σ, Δ, Q, init, rul) ; init = u_0 · q_0 · u_1;
2  Q' := {⟨q, z⟩ | q ∈ Q, z ∈ Reports(q)};
3  v := Droite(q_0) · u_1;
4  q'_0 := ⟨q_0, report(q_0, v)⟩;
5  u'_0 := u_0 · Gauche(q_0) · pre(q_0, v);
6  u'_1 := reste(q_0, v);
7  init' := u'_0 · q'_0 · u'_1;
8  pour q ∈ Q, q -f-> u_0 · q_1 · u_1 · ... · u_{k-1} · q_k · u_k ∈ rul, et z ∈ Reports(q) faire
9      z_k := Droite(q_k) · u_k · Droite(q)^{-1} · z;
10     pour i := k, ..., 1 faire
11         u'_i := reste(q_i, z_i);
12         q'_i := ⟨q_i, report(q_i, z_i)⟩;
13         z_{i-1} := Droite(q_{i-1}) · u_{i-1} · Gauche(q_i) · pre(q_i, z_i);
14     u'_0 := z^{-1} · Gauche(q)^{-1} · z_0;
15     ⟨q, z⟩ -f-> u'_0 · q'_1 · u'_1 · ... · u'_{k-1} · q'_k · u'_k ∈ rul
16 M' := (Σ, Δ, Q', init', δ');
   Sorties : M'
```

Nous pouvons observer que les états de Q' ne nécessitent pas tous d'être atteignables à partir de l'état initial, les états atteignables pouvant être identifiés par la procédure de normalisation *à la volée*. Cette observation est la base de l'algorithme de conversion polynomial dans la taille de la sortie.

Exemple 42. *On normalise le* d2W M_1 *de l'exemple 38. La règle initiale* q_0 *devient* $a \cdot \langle q_0, \varepsilon\rangle \cdot c$ *puisque* $Gauche(q_0) = a$ *et* $Droite(q_0) = c$ *sont tirés de* q_0*, mais rien n'est poussé à travers* q_0*. La construction de l'état* $\langle q_0, \varepsilon\rangle$ *déclanche l'algorithme de normalisation pour la règle* $q_0 \xrightarrow{f} q_1 \cdot ac \cdot q_1 \in rul$ *avec* $Gauche(q_0) = a$ *et* $Droite(q_0) = c$ *devant être retirés respectivement de droite et gauche de cette règle (et aucun mot poussé à travers puisque* $z = \varepsilon$*). Pour créer cette règle, on concatène à gauche et à droite l'inverse des mots retirés, à savoir* $a^{-1} \cdot q_1 \cdot ac \cdot q_1 \cdot c^{-1}$*, avant de l'évaluer. On cherche tout d'abord à pousser le plus à gauche les mots de la règle ce qui nous donne à une première étape* $a^{-1} \cdot q_1 \cdot ac \cdot c^{-1} \cdot \langle q_1, c^{-1}\rangle$ *et ensuite* $a^{-1} \cdot a \cdot \langle q_1, a\rangle \cdot \langle q_1, c^{-1}\rangle$*. On obtient au final* $\langle q_0, \varepsilon\rangle \xrightarrow{f} \langle q_1, a\rangle \cdot \langle q_1, c^{-1}\rangle$*. Notons que* $Reports(q_1) = \{(bc)^{-1}, c^{-1}, \varepsilon, a, ab\}$*, et qu'il est nécessaire de construire seulement deux états.*

Ensuite, nous construisons les règles pour l'état $\langle q_1, a\rangle$ *avec* $z = a$ *et* $Gauche(q_1) = Droite(q_1) = \varepsilon$*. On commence avec la règle* $q_1 \xrightarrow{a} \varepsilon$ *à laquelle on concatène* a^{-1} *au début de sa production et "*a*" à sa fin :* $a^{-1} \cdot \varepsilon \cdot a = \varepsilon$*, ce qui produit la règle* $\langle q_1, a\rangle \xrightarrow{a} \varepsilon \in rul'$*. Pour ce qui est de la règle* $q_1 \xrightarrow{g} q_1 \cdot abc$ *on obtient l'expression* $a^{-1} \cdot q_1 \cdot abca$*. Rappelons que* $\mathcal{L}_{q_1} = (abc)^*$ *est un langage périodique, ce qui implique que* $pre(q_1, abca) = abca$*,* $reste(q_1, abca) = \varepsilon$*, et* $report(q_1, abca) = a$*. Par conséquent, on obtient la règle* $\langle q_1, a\rangle \xrightarrow{g} = bca \cdot \langle q_1, a\rangle \in rul'$*. Ici, il est important d'utiliser uniquement le report et non l'intégralité du mot poussé* $\langle q_1, abca\rangle$ *qui nous ferait entrer dans une boucle infinie de création d'états. De la même manière on obtient :* $\langle q_1, c^{-1}\rangle \xrightarrow{g} cab \cdot \langle q_1, c^{-1}\rangle$ *et* $\langle q_1, c^{-1}\rangle \xrightarrow{a} \varepsilon$ *dans* rul'*.*

Théorème 15. *Pour un* d2W M*, appelons* M' *le* d2W *obtenu par l'algorithme précedent appliqué sur* M*. Alors,* M' *est équivalent à* M *et satisfait les conditions* **(E1)** *et* **(E2)***. De plus,* M' *peut être construit en temps polynomial dans la taille de* M'*, qui est au plus de taille double-exponentielle dans la taille de* M*.*

On remarque que le lemme 25 montre essentiellement que le résultat de minimisation de deux transducteurs équivalents est le même transducteur (modulo renommage des états) ce qui est une piste de preuve du théorème 15. Plus formellement, ce théorème se prouve à l'aide de plusieurs résultats auxiliaires.

Proposition 27. *Tout langage réduit* $\mathcal{L}$ *vérifie que*

1. $\forall w \in prefixe(noyau(\mathcal{L}))$ *le langage* $w^{-1} \cdot \mathcal{L} \cdot w$ *est réduit,*
2. $\forall w \in suffixe(noyau(\mathcal{L}^{\text{rev}}))$ *le langage* $w \cdot \mathcal{L} \cdot w^{-1}$ *est aussi réduit.*

Preuve. Nous prouvons uniquement la première ligne, la deuxième en étant une conséquence directe. Si $\varepsilon \in \mathcal{L}$ alors $\varepsilon \in w^{-1} \cdot \mathcal{L} \cdot w$ et la proposition est triviale.

Autrement, il existe deux mots u et v variant sur la première lettre. Dans ce cas, $noyau(\mathcal{L})$ est réduit à ε et la proposition est toute aussi triviale. □

Lemme 28. *$report(\mathcal{L}, z)$ appartient à l'ensemble*

$$prefixe(noyau(\mathcal{L})) \cup suffixe(noyau(\mathcal{L}^{\text{rev}}))$$

Preuve. Nous prouvons ce lemme pour $z = w \in \Delta^*$, l'autre cas étant similaire. Nous procédons par une étude de cas dépendant de la classe de $\mathcal{L}$. Si $\mathcal{L}$ est trivial alors $report(\mathcal{L}, z) = \varepsilon$. Si $\mathcal{L}$ est périodique, alors $trousseau(\mathcal{L}) = noyau(\mathcal{L})^* \cdot prefixe(noyau(\mathcal{L}))$. Par conséquent, $pre(\mathcal{L}, w)$ est de la forme $noyau(\mathcal{L})^i \cdot o$ où $i \geq 0$ et o est un préfixe de $noyau(\mathcal{L})$. Par conséquent, $report(\mathcal{L}, w) = (noyau(\mathcal{L})^i \cdot o) \bmod noyau(\mathcal{L}) = o$. Sinon, $\mathcal{L}$ est non périodique et non trivial, $pre(\mathcal{L}, w) = lcp(\{w, noyau(\mathcal{L})\})$ et est par conséquent un préfixe de $noyau(\mathcal{L})$. Par définition de *report*, $report = pre(\mathcal{L}, w)$ et le lemme est vérifié. □

Lemme 29. *Pour tout état $q \in Q$, tout mot $z \in Reports(q)$, et tout arbre $t \in dom(\llbracket M \rrbracket_q)$*

$$\llbracket M' \rrbracket_{\langle q,z \rangle}(t) = z^{-1} \cdot Gauche(q)^{-1} \cdot \llbracket M \rrbracket_q(t) \cdot Droite(q)^{-1} \cdot z \tag{5.3}$$

Preuve. Les règles sont construites en utilisant l'algorithme suivant :

---

1 $z_k \coloneqq Droite(q_k) \cdot u_k \cdot Droite(q)^{-1} \cdot z$;
2 **pour** $i \coloneqq k, \ldots, 1$ **faire**
3 $\quad u'_i \coloneqq reste(q_i, z_i)$;
4 $\quad q'_i \coloneqq \langle q_i, report(q_i, z_i) \rangle$;
5 $\quad z_{i-1} \coloneqq Droite(q_{i-1}) \cdot u_{i-1} \cdot Gauche(q_i) \cdot pre(q_i, z_i)$
6 $u'_0 \coloneqq z^{-1} \cdot Gauche(q)^{-1} \cdot z_0$;

Nous prouvons ce lemme par induction sur la structure des termes.

Le cas de base concerne les constantes et, de ce fait, les règles de la forme $\langle q, z \rangle \xrightarrow{a} u' \in rul'$ et $\llbracket M \rrbracket_q(a) = u$. L'algorithme calcule u' de la manière suivante

$$u' = z^{-1} \cdot Gauche(q)^{-1} \cdot u \cdot Droite(q)^{-1} \cdot z$$

ce qui correspond à l'hypothèse du lemme, à savoir $\llbracket M' \rrbracket_{\langle q,z \rangle}(a) = z^{-1} \cdot Gauche(q)^{-1} \cdot u \cdot Droite(q)^{-1} \cdot z$.

Considérons à présent un terme $t = f(t_1, \ldots, t_k)$, un mot z et une règle $q \xrightarrow{f} u_0 \cdot q_1 \cdot \ldots \cdot q_k \cdot u_k$. On suppose que l'hypothèse d'induction, i.e. l'Equation (5.3), est vérifiée pour chaque sous terme $t_1, \ldots, t_k$. Il reste à prouver

que :

$$u'_0 \cdot [\![M']\!]_{q'_1}(t_1) \cdot u'_1 \cdot \ldots \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k = \\ z^{-1} \cdot Gauche(q)^{-1} \cdot u_0 \cdot [\![M]\!]_{q_1}(t_1) \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \cdot Droite(q)^{-1} \cdot z$$

Pour appliquer notre hypothèse d'induction il est nécessaire de vérifier que $report(q_i, z_i)$ appartient à l'ensemble $prefixe(noyau(L_{q_i})) \cup suffixe(noyau(L^{\text{rev}}_{q_i}))$, ce que l'on vérifie grâce au lemme 28.

Fait 1.

$$pre(q_i, z_i) \cdot [\![M']\!]_{q'_i}(t_i) \cdot reste(q_i, z_i) = \\ Gauche(q_i)^{-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot Droite(q_i)^{-1} \cdot z_i.$$

Preuve. La preuve se fait par hypothèse d'induction. On développe $[\![M]\!]_{q'_i}(t_i) = report(q_i, z_i)^{-1} \cdot Gauche(q_i)^{-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot Droite(q_i)^{-1} \cdot report(q_i, z_i)$. On remarque que $Gauche(q_i)^{-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot Droite(q_i)^{-1}$ est un mot de $L^{\circ}_{q_i}$ ce qui nous permet de conclure cette preuve en utilisant la proposition 26. □

Nous prouvons les invariants suivants de l'algorithme pour tout $1 \leq i \leq k$:

$$z_{i-1} \cdot [\![M']\!]_{q'_i}(t_i) \cdot u'_i \cdot \ldots \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k = \\ Droite(q_{i-1}) \cdot u_{i-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot u_i \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \cdot Droite(q)^{-1} \cdot z \quad (5.4)$$

Pour le démontrer nous procédons par induction sur i de k à 1. Le cas initial, où $i = k$, est le suivant :

$$\begin{aligned} & z_{k-1} \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k \\ & \quad = Droite(q_{k-1}) \cdot u_{k-1} \cdot Gauche(q_k) \cdot pre(q_k, z_k) \cdot [\![M']\!]_{q'_k}(t_k) \cdot reste(q_k, z_k) \\ & \quad = Droite(q_{k-1}) \cdot u_{k-1} \cdot Gauche(q_k) \cdot Gauche(q_k)^{-1} \cdot \\ & \qquad [\![M]\!]_{q_k}(t_k) \cdot Droite(q_k)^{-1} \cdot Droite(q_k) \cdot u_k \cdot Droite(q)^{-1} \cdot z \\ & \quad = Droite(q_{k-1}) \cdot u_{k-1} \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \cdot Droite(q)^{-1} \cdot z \end{aligned}$$

ce qui montre que l'invariant est bien respecté dans ce cas de base. Supposons

maintenant qu'il soit vérifié pour un certain $1 \leq i \leq k$. Pour $i-1$ nous avons :

$$\begin{aligned}
&z_{i-1} \cdot [\![M']\!]_{q'_i}(t_i) \cdot u'_i \cdot \ldots \cdot [\![M]\!]_{q'_k}(t_k) \cdot u'_k \\
&\quad = \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot \mathit{Gauche}(q_i) \cdot \mathit{pre}(q_i, z_i) \cdot [\![M']\!]_{q'_i}(t_i) \cdot \mathit{reste}(q_i, z_i) \\
&\qquad \cdot \ldots \cdot [\![M']\!]_{q'_k}(t_k) \cdot u'_k \\
&\quad = \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot \mathit{Gauche}(q_i) \cdot \mathit{Gauche}(q_i)^{-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot \mathit{Droite}(q_i)^{-1} \cdot z_i \\
&\qquad \cdot \ldots \cdot T_{q'_k}(t_k) \cdot u'_k \\
&\quad = \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot \mathit{Droite}(q_i)^{-1} \cdot \mathit{Droite}(q_i) \cdot u_i \\
&\qquad \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \cdot \mathit{Droite}(q)^{-1} \cdot z \\
&\quad = \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot [\![M]\!]_{q_i}(t_i) \cdot u_i \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \cdot \mathit{Droite}(q)^{-1} \cdot z
\end{aligned}$$

Ce qui prouve l'invariant.

Puisque $u'_0 = z^{-1} \cdot \mathit{Gauche}(q)^{-1} \cdot z_0$ nous obtenons que

$$\begin{aligned}
&u'_0 \cdot [\![M']\!]_{q'_1} \cdot u'_1 \cdot \ldots \cdot [\![M']\!]_{q'_k} \cdot u'_k = \\
&\qquad z^{-1} \cdot \mathit{Gauche}(q)^{-1} \cdot u_0 \cdot [\![M]\!]_{q_1}(t_1) \cdot \ldots \cdot [\![M]\!]_{q_k}(t_k) \cdot u_k \cdot \mathit{Droite}(q)^{-1} \cdot z
\end{aligned}$$

□

Lemme 30. (**E1**) *est satisfait par* M'.

Preuve. Pour prouver (**E1**), il est nécessaire de montrer que $\mathcal{L}_{\langle q,z\rangle}$ est réduit. D'après le lemme 29 précédent, chaque état $q \in Q$, pour tout $z \in \mathit{Reports}(q)$, et tout $t \in \mathit{dom}([\![M]\!]_q)$

$$\mathcal{L}_{\langle q,z\rangle} = z^{-1} \cdot \mathit{Gauche}(q)^{-1} \cdot \mathcal{L}_q \cdot \mathit{Droite}(q)^{-1} \cdot z$$

Puisque $z \in \mathit{Reports}(q)$, alors $z = w \in \mathit{trousseau}(\mathcal{L}_q^\circ)$ ou $z = w^{-1}$ avec $w \in \mathit{trousseau}((\mathcal{L}_q^\circ)^{\mathrm{rev}})$. Les langages $\mathit{Gauche}(q)^{-1} \cdot \mathcal{L}_q \cdot \mathit{Droite}(q)^{-1} = \mathcal{L}_q^\circ$ sont réduits. Par conséquent, en nous basant sur la proposition 27, on obtient que (**E1**) est vérifié. □

Lemme 31. *Soit* $\mathcal{L}$ *un langage réduit, pour tout* $w \in \mathit{prefixe}(\mathit{Gauche}(\mathcal{L} \cdot \mathcal{L}'))$ *nous avons* $w \in \mathit{prefixe}(\mathcal{L}')$, $\mathit{pre}(\mathcal{L}, w) = w$ *et* $\mathit{reste}(\mathcal{L}, w) = \varepsilon$.

Preuve. Si $w = \varepsilon$ alors le lemme est trivial. Sinon, nous savons que $\varepsilon \in \mathcal{L}$ (sinon $\mathcal{L}$ ne serait pas réduit). Dans ce cas, w est un préfixe commun de chaque mot non vide de $\mathcal{L}'$, ce qui implique que $\mathcal{L} \cdot w$ a également w comme préfixe, et par définition $w \in \mathit{trousseau}(\mathcal{L})$. On a donc $\mathit{pre}(\mathcal{L}, w) = w$ et $\mathit{reste}(\mathcal{L}, w) = \varepsilon$. □

Lemme 32. *Pour toute règle* $\langle q, z\rangle \xrightarrow{f} u'_0 \cdot q'_1 \cdot u'_1 \cdot \ldots \cdot u'_{k-1} \cdot q'_k \cdot u'_k$ *de* M', *et chaque* $0 \leq i \leq k$, $u'_i \in \Delta^*$.

Preuve. Considérons une règle $\langle q, z\rangle \xrightarrow{f} u'_0 \cdot q'_1 \cdot u'_1 \cdot \ldots \cdot u'_{k-1} \cdot q'_k \cdot u'_k$ de M' et $w \in \mathit{suffixe}(\mathit{noyau}((\mathcal{L}_q^\circ)^{\mathrm{rev}})) \cup \mathit{prefixe}(\mathit{noyau}(\mathcal{L}_q^\circ))$. Nous avons principalement à démontrer que, pour chaque z_i calculé par l'algorithme de normalisation, si $z_i = w_i^{-1}$ tel que $w_i \in \Delta^*$, alors $u_i = \mathit{reste}(\hat{\mathcal{L}}_{q_i}^{\mathrm{rev}}, w_i^{\mathrm{rev}}) = \varepsilon$.

Nous définissons $\mathcal{L}_i$ par :

$$\mathcal{L}_i = z^{-1}\cdot \mathit{Gauche}(q)^{-1}\cdot u_0 \cdot \mathit{Gauche}(\mathcal{L}_{q_1})\cdot \mathcal{L}_{q_1}^\circ \cdot \mathit{Droite}(\mathcal{L}_{q_1})\cdot u_1 \cdot \ldots \cdot u_{i-1}\cdot \mathit{Gauche}(\mathcal{L}_{q_i})$$

Nous pouvons remarquer que w_i est un suffixe de $\mathit{Droite}(\mathcal{L}_i \cdot \mathcal{L}_{q_i}^\circ)$ alors w_i^{rev} est un préfixe de $\mathit{Gauche}((\mathcal{L}_{q_i}^\circ)^{\mathrm{rev}} \cdot \mathcal{L}_i^{\mathrm{rev}})$. Le langage $(\mathcal{L}_{q_i}^\circ)^{\mathrm{rev}}$ est réduit et par le lemme 31, $\mathit{reste}(\mathcal{L}_{q_i}^{\mathrm{rev}}, w_i^{\mathrm{rev}}) = \varepsilon$ et $\mathit{pre}(\mathcal{L}_{q_i}^{\mathrm{rev}}, w_i^{\mathrm{rev}}) = w_i^{\mathrm{rev}}$. Il suffit donc de prouver que pour tout i, w_i est un suffixe de $\mathit{Droite}(\mathcal{L}_i \cdot \hat{\mathcal{L}}_{q_i})$.

Cela peut se faire par induction sur i de k à 1. Pour $i = k$, $z_k = w_k^{-1} = \mathit{Droite}(q_k) \cdot u_k \cdot \mathit{Droite}(q)^{-1} \cdot z$. On obtient que $\mathcal{L}_{\langle q,z\rangle} = \mathcal{L}_k \cdot \mathcal{L}_{q_k}^\circ \cdot w_k^{-1}$ et en suivant le lemme 30, $\mathcal{L}_{\langle q,z\rangle}$ est réduit. Par conséquent $w_k = \mathit{Droite}(\mathcal{L}_k \cdot \mathcal{L}_{q_k}^\circ)$.

Supposons maintenant, par hypothèse d'induction, que w_i est un suffixe de $\mathit{Droite}(\mathcal{L}_i \cdot \mathcal{L}_{q_i}^\circ)$, et d'après le lemme 31, que w_i est également un suffixe de $\mathit{Droite}(\mathcal{L}_i)$. De plus, $\mathit{pre}(\mathcal{L}_{q_i}^{\mathrm{rev}}, w_i^{\mathrm{rev}}) = w_i^{\mathrm{rev}}$. En suivant l'algorithme nous pouvons observer que $z_{i-1} = \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot \mathit{Gauche}(q_i) \cdot \mathit{pre}(q_i, z_i)$. Par conséquent, $z_{i-1} = \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot \mathit{Gauche}(q_i) \cdot w_i^{-1}$. Si $z_{i-1} \in \Delta^*$ l'induction est vérifiée ; autrement, $z_{i-1} = w_{i-1}^{-1}$ avec $w_{i-1} \in \Delta^*$. Puisque w_i est un suffixe de $\mathit{Droite}(\mathcal{L}_i) = \mathit{Droite}(\mathcal{L}_{i-1} \cdot \mathit{Droite}(q_{i-1}) \cdot u_{i-1} \cdot \mathit{Gauche}(q_i))$, alors z_{i-1} est un suffixe de $\mathit{Droite}(\mathcal{L}_{i-1} \cdot \mathcal{L}_{q_{i-1}}^\circ)$.

□

Lemme 33. *Soit $\mathcal{L} \subseteq \Delta^*$ un langage réduit et $z \in \mathbb{G}_\Delta$:*

$$\mathit{lcp}((\mathit{report}(\mathcal{L}, z)^{-1} \cdot \mathcal{L} \cdot \mathit{report}(\mathcal{L}, z)) \cdot \mathit{reste}(\mathcal{L}, z)) = \varepsilon$$

Preuve. On choisit de traiter le cas où $z = w \in \Delta^*$, l'autre cas étant symétrique. En se basant sur le lemme 27, on peut remarquer que $\mathcal{L}' = \mathit{report}(\mathcal{L}, z)^{-1} \cdot \mathcal{L} \cdot \mathit{report}(\mathcal{L}, z)$ est réduit. Si $\mathcal{L}'$ ne contient pas ε alors le lemme est trivial. Considérons le cas où $\varepsilon \in \mathcal{L}'$.

La preuve de ce lemme se fait par contradiction. Prenons $u \neq \varepsilon$ tel que $u = \mathit{lcp}((\mathit{report}(\mathcal{L}, w)^{-1} \cdot \mathcal{L} \cdot \mathit{report}(\mathcal{L}, w)) \cdot \mathit{reste}(\mathcal{L}, w))$. On note v le mot $\mathit{pre}(\mathcal{L}, w) \cdot u$. Nous prouvons que v est un préfixe de w. En effet, puisque $\varepsilon \in \mathcal{L}'$, u est un préfixe de $\mathit{reste}(\mathcal{L}, w) = \mathit{pre}(\mathcal{L}, w)^{-1} \cdot w$. En utilisant la proposition 26, on obtient que v est un préfixe commun de $\mathcal{L} \cdot w$. Par conséquent, v appartient à $\mathit{trousseau}(\mathcal{L})$. Cela contredit le fait qu'il est le préfixe maximal de w appartenant à $\mathit{trousseau}(\mathcal{L})$.

□

Lemme 34. **(E2)** *est satisfait par M'.*

Preuve. D'après le lemme 30, tout langage $\mathcal{L}_{\langle q,z\rangle}$ est réduit. Pour la règle initiale, (**E2**) est une conséquence directe du lemme 33.

Considérons maintenant la règle $f \xrightarrow{q'} u'_0 \cdot q_1 \cdot \ldots \cdot q_k \cdot u'_k$. Le lemme 32 nous montre que u'_i est soit ε, soit un mot de Δ^*. De plus, d'après la ligne 3 de l'algorithme, pour tout $i > 0$, $u'_i = reste(q_i, z_i)$. Par conséquent, en nous reposant sur le lemme 33, on obtient que $lcp(\mathcal{L}_{q'_i} \cdot u'_i) = \varepsilon$. Il est à noter que si deux langage $\mathcal{L}$ et $\mathcal{L}'$ ont un plus grand préfixe commun $lcp(\mathcal{L}) = lcp(\mathcal{L}') = \varepsilon$ alors $lcp(\mathcal{L}\cdot\mathcal{L}') = \varepsilon$. Donc, $lcp(\mathcal{L}_{q'_i}\cdot u'_i\cdot\ldots\cdot\mathcal{L}_{q'_k}\cdot u'_k) = \varepsilon$ et (**E2**) est par conséquent vérifiée pour chacune des règles.

□

Lemme 35. *M est équivalent à M'*

Preuve. Nous prouvons tout d'abord que pour tout arbre t, $u_0 \cdot [\![M]\!]_{q_0}(t) \cdot u_1 = u'_0 \cdot [\![M']\!]_{q'_0}(t) \cdot u'_1$. Dans un premier temps on s'attarde sur la règle initiale. En utilisant le lemme 29, on obtient, pour $v = Droite(q_0) \cdot u_1$:

$$\begin{aligned} u'_0 \cdot [\![M']\!]_{q'_0}(t) \cdot u'_1 &= u_0 \cdot Gauche(q_0) \cdot pre(q_0, v) \cdot report(q_0, v)^{-1} \cdot Gauche(q_0)^{-1} \\ &\quad \cdot [\![M]\!]_{q_0}(t) \cdot Droite(q_0)^{-1} \cdot report(q_0, v) \cdot reste(q_0, v) \\ (\text{Prop. 26}) &= u_0 \cdot Gauche(q_0) \cdot Gauche(q_0)^{-1} \cdot [\![M]\!]_{q_0}(t) \cdot Droite(q_0)^{-1} \cdot v \\ &= u_0 \cdot [\![M]\!]_{q_0}(t) \cdot u_1 \end{aligned}$$

□

5.4.5 Bornes exponentielles

Nous cherchons à montrer ici que la taille d'une règle peut exploser exponentiellement durant la normalisation.

Exemple 43. *Pour $n \geq 0$ on définit un* dST2W *M_n sur l'alphabet $\Sigma = \{f^{(2)}, a^{(0)}\}$ ayant comme état initial q_0, et les règles suivantes (avec $0 \leq i < n$) :*

$$q_i \xrightarrow{f} q_{i+1} \cdot q_{i+1}, \qquad\qquad q_n \xrightarrow{a} a.$$

La transformation définie par M_n transforme un arbre binaire parfait de taille n dans la concaténation de ces feuilles a^{2^n}. M_n n'est pas travailleur, toute la sortie pouvant être produite dès le départ puisqu'un seul arbre peut être transformé ce qui amène à une seule production possible. Pour le rendre travailleur, il suffit donc de remplacer la règle initiale par $a^{2^n} \cdot q_0(x_0)$ et la dernière règle par $q_n \xrightarrow{a} \varepsilon$.

L'exemple suivant montre quant à lui comment le nombre d'états peut lui aussi devenir exponentiel.

Exemple 44. *Pour $n \geq 0$ nous définissons le* d$\mathrm{ST2W}$ N_n *sur* $\Sigma = \{g_1^{(1)}, g_0^{(1)}, a_1^{(0)}, a_0^{(0)}\}$, *ayant pour règle initiale q_0. L'ensemble rul est constitué, pour $0 \leq i < n$, des règles suivantes :*

$$q_i \xrightarrow{g_0} q_{i+1}, \qquad q_n \xrightarrow{a_0} \varepsilon,$$
$$q_i \xrightarrow{g_1} q_{i+1} \cdot a^{2^i}, \qquad q_n \xrightarrow{a_1} a^{2^n} \cdot \#.$$

Même si la taille du transducteur est exponentielle dans n, les facteurs exponentiels a^{2^i} peuvent être aisément compressés pour obtenir un d$\mathrm{ST2W}$ *linéaire dans la taille de n (cf. exemple 43). M_n vérifie la condition* **(E1)** *mais pas la* **(E2)**. *Elle définit la transformation suivante :*

$$[\![M_n]\!] = \{(g_{b_0}(g_{b_1}(\dots g_{b_{n-1}}(a_0)\dots)), a^{\mathbf{b}}) \mid \mathbf{b} = (b_{n-1}, \dots, b_0)_2\} \cup$$
$$\{(g_{b_0}(g_{b_1}(\dots g_{b_{n-1}}(a_1)\dots)), a^{2^n} \cdot \# \cdot a^{\mathbf{b}}) \mid \mathbf{b} = (b_{n-1}, \dots, b_0)_2\},$$

pour laquelle $(b_{n-1}, \dots, b_0)_2 = \sum_i b_i * 2^i$.

La version normalisée M'_n a pour règle initiale $\langle q_0, \varepsilon\rangle$ et les règles suivantes :

$$\langle q_i, a^j\rangle \xrightarrow{g_0} \langle q_{i+1}, a^j\rangle, \qquad \langle q_n, a^k\rangle \xrightarrow{a_0} \varepsilon,$$
$$\langle q_i, a^j\rangle \xrightarrow{g_1} a^{2^i} \cdot \langle q_{i+1}, a^{j+2^i}\rangle, \qquad \langle q_n, a^k\rangle \xrightarrow{a_1} a^{2^n-k}\#a^k,$$

où $0 \leq i < n$, $0 \leq j < 2^i$, et $0 \leq k < 2^n$. On remarque également que M'_n est le ed$\mathrm{ST2W}$ *minimal reconnaissant $[\![M_n]\!]$.*

Chapitre 6

Apprentissage de transducteurs descendants d'arbres en mots

Nous pouvons maintenant nous intéresser au problème de l'apprentissage à proprement parler. Nous nous basons sur les méthodes d'apprentissage vues précédemment pour les transducteurs de mots, ou encore les $d\text{TA}^{\downarrow}s$. La mise en place de l'algorithme d'apprentissage, tel qu'on le conçoit, passe par une adaptation du théorème de Myhill-Nerode pour les $\mathcal{STW}s$, assurant entre autre que pour toute transformation il existe un dST2W la représentant.

Ce théorème sera la base nécessaire à la définition d'un modèle d'apprentissage, afin d'élaborer un algorithme d'apprentissage permettant à partir d'un échantillon $S \subseteq T_\Sigma \times \Delta^*$ de construire un edST2W consistant avec S. Cet algorithme se basera sur la caractérisation sémantique des transformations cibles, ainsi que les propriétés propres aux dST2Ws normalisés. Il devra respecter des contraintes de polynomialité pour le temps d'exécutions et la taille des résultats produits.

6.1 Théorème de Myhill-Nerode

Théorème 16. *Pour toute transformation τ, les conditions suivantes sont équivalentes :*

1. *τ est définissable par un edST2W ;*
2. *τ appartient à la classe des $\mathcal{STW}s$ et a un index de Myhill-Nerode fini ;*
3. *$Cano(\tau)$ est l'edST2W minimal définissant τ.*

Preuve. $((1) \Rightarrow (2))$, Prenons un dST2W M. Comme le montre le théorème 15, il est possible de construire un edST2W M' équivalent, ayant un nombre fini d'états. Le lemme 24, quant à lui, nous assure qu'il existe une transformation $\tau = [\![M']\!]$ appartenant à la classe des $\mathcal{STW}s$ dont l'index de Myhill-Nerode est égal au nombre d'états de M', donc fini.

$((2) \Rightarrow (3))$, Il faut que $Cano(\tau)$ soit l'unique edST2W minimal décrivant τ, ce qui est prouvé par le corrolaire 5.

$((3) \Rightarrow (1))$ est trivial. □

6.2 Consistence des dST2Ws

On souhaiterait définir un algorithme pouvant, pour tout échantillon S, nous produire un dST2W M étant cohérent avec S, i.e. $S \subseteq [\![M]\!]$, et cela polynomialement en temps et taille dans la taille de l'échantillon. Malheureusement, à part si $P = NP$, les résultats suivants excluent l'existence d'un tel algorithme.

Problème : $\texttt{CONS}_d$ST2W
Entrée : $S \subseteq T_\Sigma \times \Delta^*$ (un ensemble fini)
Question : Existe t'il un dST2W M tel que $S \subseteq [\![M]\!]$?

Théorème 17. *Le problème de consistance pour les* dST2Ws *est* NP*-complet.*

Ce théorème se prouve en deux temps. Premièrement, nous montrons que le problème de consistance est NP, puis qu'il est NP-dur.

Lemme 36. $\texttt{CONS}_d$ST2W *est dans* NP*.*

Preuve. Pour prouver l'appartenance de $\texttt{CONS}_d$ST2W à NP, nous introduisons la notion d'arbres d'alignement. Un *arbre d'alignement* est un arbre d'arité bornée $\Gamma = \bigcup_{i=0}^{N} \Gamma^{(i)}$, pour lequel $\Gamma^{(i)} = \Sigma^{(i)} \times (\Delta^*)^{i+1}$, N étant l'arité maximale d'un symbole de Σ. Un arbre d'alignement α représente l'exemple correspondant $Ex(\alpha) = (In(\alpha), Out(\alpha))$ avec In et Out défini par :

$$In(\langle f, u_0, \ldots, u_1\rangle(\alpha_1, \ldots, \alpha_k)) = f(In(\alpha_1), \ldots, In(\alpha_k))$$
$$Out(\langle f, u_0, \ldots, u_1\rangle(\alpha_1, \ldots, \alpha_k)) = u_0 \cdot Out(\alpha_1) \cdot u_1 \cdot Out(\alpha_2) \cdot u_2 \cdot \ldots \cdot u_{k-1} \cdot Out(\alpha_k) \cdot u_k$$

On peut remarquer qu'il peut exister un nombre important d'arbres d'alignement qui représente une même paire (t, w). La taille de ces arbres d'alignement est proportionnel à la somme des taille de t et w.

Intuitivement, un arbre d'alignement peut être vu comme la la représentation d'une exécution d'un dST2W sur l'entrée. Comme nous avions pu le constater auparavant, un dST2W peut être vu comme un $d\text{TA}^{\downarrow}$ sur Γ. Une condition suffisante pour traduire avec succès un $d\text{TA}^{\downarrow}$ arbitraire $A = (\Gamma, Q, q_0, \delta)$ sur Γ en un dST2W repose sur la condition suivante : $(\star)$ Pour tout $f \in \Sigma^{(k)}$ et tout $q \in Q$, δ est défini sur $(q, \langle f, u_0, \ldots, u_k\rangle)$ pour au plus un tuple $(u_0, \ldots, u_k)$. Cette condition permet le passage presque direct entre les deux formalismes.

Il est trivial de montrer qu'un échantillon S est cohérent si et seulement s'il existe un $d\text{TA}^{\downarrow}$ A respectant $(\star)$ tel que pour chaque paire $(t, w) \in S$ de l'automate A reconaissent au moins un arbre d'alignement de (t, w). De plus, il est connu que pour un ensemble d'arbres T il existe un $d\text{TA}^{\downarrow}$ B capturant T

i.e., $T \subseteq [\![B]\!]$, si et seulement si il existe un $d\text{TA}^{\downarrow}$ B' capturant T dont la taille est polynomiale dans la somme des tailles des arbres de T.

Nous construisons donc une machine de Turing non déterministe pour tester $\texttt{CONS}_{d\text{ST2W}}$ marchant de la manière suivante :

1. elle devine un arbre d'alignement pour tout exemple de S,
2. elle devine un $d\text{TA}^{\downarrow}$ sur Γ,
3. elle vérifie que ce $d\text{TA}^{\downarrow}$ accepte tous les alignements construits dans l'étape 1.

□

Lemme 37. $\texttt{CONS}_{d\text{ST2W}\ s}$ *est* NP*-dure.*

Preuve. Pour montrer la NP-dureté du problème, nous réduisons une variante de $\texttt{SAT}_{\texttt{ONE-IN-THREE}}$ de satisfiabilité connue pour être NP-complet (Papadimitriou, 1994, Exercice 9.5.3.). Nous rappelons que, d'après la preuve du théorème 14, les formules 3CNF φ appartiennent à $\texttt{SAT}_{\texttt{ONE-IN-THREE}}$ s'il existe des valeurs satisfaisant exactement un litéral de chaque clause de φ.

Fixons maintenant une formule 3CNF φ sur les variables Booléennes $x_1, \ldots, x_n$, de sorte à ce que chaque close $c_j = L_{j,1} \vee L_{j,2} \vee L_{j,3}$est une disjonction d'exactement trois literaux ($j \in \{1, \ldots, k\}$). L'échantillon construit correspond aux exemples d'un transformation prenant en entrée des clause de φ et retournant un unique caractère $\mathbf{t}$.

Pour que la représentation des clauses reste claire, il vaut mieux utiliser des notations similaires à celles utilisées précédemment : on utilise un symbole n-aire c pour représenter une clause, les symboles binaires $x_1, \ldots, x_n$ pour les variables, et deux constantes $\bullet$ et $\circ$. De plus, $x_i(\bullet, \circ)$ correspond au littéral positif x_i, et $x_i(\circ, \bullet)$ au littéral négatif $\neg x_i$, alors que $x_i(\circ, \circ)$ signifie que la variable x_i n'est pas utilisée. Enfin $x_i(\bullet, \bullet)$ correspond au fait que ni le littéral positif, ni le négatif ne sont utilisés. Par exemple, à partir de la formule $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee \neg x_3)$, nous construisons l'échantillon S_{φ_0} contenant les exemples suivants :

$$
\begin{aligned}
&(c(x_1(\bullet,\circ), x_2(\circ,\bullet), x_3(\bullet,\circ), x_4(\circ,\circ)), \mathbf{t}),\\
&(c(x_1(\circ,\circ), x_2(\bullet,\circ), x_3(\circ,\bullet), x_4(\bullet,\circ)), \mathbf{t}),\\
&(c(x_1(\bullet,\bullet), x_2(\circ,\circ), x_3(\circ,\circ), x_4(\circ,\circ)), \mathbf{t}),\\
&(c(x_1(\circ,\circ), x_2(\bullet,\bullet), x_3(\circ,\circ), x_4(\circ,\circ)), \mathbf{t}),\\
&(c(x_1(\circ,\circ), x_2(\circ,\circ), x_3(\bullet,\bullet), x_4(\circ,\circ)), \mathbf{t}),\\
&(c(x_1(\circ,\circ), x_2(\circ,\circ), x_3(\circ,\circ), x_4(\bullet,\bullet)), \mathbf{t}),\\
&(c(x_1(\circ,\circ), x_2(\circ,\circ), x_3(\circ,\circ), x_4(\circ,\circ)), \varepsilon).
\end{aligned}
$$

En résumé, les deux premiers exemples encodent les deux clauses de φ_0. Les quatre exemples suivants encodent les clause triviales $x_i \vee \neg x_i$, assurant que chaque variable encode une évaluation propre. Le dernier exemple encode la clause vide, où aucune variable n'est utilisée, assurant que la sortie $\mathbf{t}$ ne peut être générée que si les variables sont utilisés.

Formellement, nous avons pour alphabet d'entrée $\Sigma = \{c^{(3)}, x_1^{(2)}, \dots, x_n^{(2)}, \bullet^{(0)}, \circ^{(0)}\}$ et pour alphabet de sortie $\Delta = \{\mathbf{t}\}$. L'échantillon S_φ contient les exemples suivants :

(1) $(c(x_1(\circ,\circ), \dots, x_n(\circ,\circ)), \varepsilon)$

(2) $(c(x_1(\circ,\circ), \dots x_{i-1}(\circ,\circ), x_i(\bullet,\bullet), x_{i+1}(\circ,\circ), \dots, x_n(\circ,\circ)), \mathbf{t})$ for $i \in \{1, \dots, n\}$,

(3) $(c(x_1(\ell^+_{j,1}, \ell^-_{j,1}), \dots, x_n(\ell^+_{j,n}, \ell^-_{j,n})), \mathbf{t})$ pour $j \in \{1, \dots, m\}$,
$\ell^+_{j,i}$ est égal à $\bullet$ si la clause c_j utilise x_i , $\circ$ sinon, et de la même manière,
$\ell^-_{j,i}$ est égal à $\bullet$ si la clause c_j utilise $\neg x_i$,$\circ$ sinon (avec $i \in \{1, \dots, n\}$).

L'assertion principale est

$$S_\varphi \in \mathtt{CONS}_{d\mathrm{ST2W}} \Leftrightarrow \varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}}.$$

($\Leftarrow$), Prenons la fonction d'évaluation $V : \{x_1, \dots, x_n\} \to \{\mathbf{true}, \mathbf{false}\}$ fixant les variables telles que $\varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}}$. On défini un dST2W $M_V = (\Sigma, \Delta, Q, \mathit{init}, \mathit{rul})$ ayant pour états $Q = \{q_i, q_{\langle i,+\rangle}, q_{\langle i,-\rangle} \mid i \in \{1, \dots, n\}\} \cup \{q_0\}$, pour règle initiale $\mathit{init} = q_0$ et les règles suivantes (pour $i \in \{1, \dots, n\}$) :

$$q_0 \xrightarrow{c} q_1 \cdot \dots \cdot q_n, \qquad q_i \xrightarrow{x_i} q_{\langle i,+\rangle} \cdot q_{\langle i,-\rangle},$$

$$q_{\langle i,+\rangle} \xrightarrow{\circ} \varepsilon, \qquad q_{\langle i,-\rangle} \xrightarrow{\circ} \varepsilon,$$

$$q_{\langle i,+\rangle} \xrightarrow{\bullet} \begin{cases} \mathbf{t} & \text{if } V(x_i) = \mathit{true}, \\ \varepsilon & \text{if } V(x_i) = \mathit{false}, \end{cases} \qquad \delta(q_{\langle i,-\rangle}, \bullet) = \begin{cases} \varepsilon & \text{if } V(x_i) = \mathit{true}, \\ \mathbf{t} & \text{if } V(x_i) = \mathit{false}. \end{cases}$$

Il est simple de vérifier que M_V est cohérent avec S_φ : Le respect des exemples (1) et (2) nous vient directement de la définition de M_V et la cohérence avec les exemples (3) nous vient du fait que V associe à chaque variable une valeur telle que $\varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}}$.

($\Rightarrow$), Prenons un dST2W M cohérent avec S_φ et définissons l'ensemble d'arbres suivants (pour $i \in \{1, \dots, n\}$) :

$$t_i^+ = c(x_1(\circ,\circ), \dots, x_{i-1}(\circ,\circ), x_i(\bullet,\circ), x_{i+1}(\circ,\circ), \dots, x_n(\circ,\circ)),$$
$$t_i^- = c(x_1(\circ,\circ), \dots, x_{i-1}(\circ,\circ), x_i(\circ,\bullet), x_{i+1}(\circ,\circ), \dots, x_n(\circ,\circ)).$$

L'arbre t_i^+ correspond à la clause x_i seulement et t_i^- correspond à la clause $\neg x_i$ seulement. Le domaine de M étant clos par chemin et puisqu'il contient les arbres de S_φ, M accepte donc les arbres t_i^+ et t_i^-. Les exemples (1) et

(2) assurent que, pour chaque $i \in \{1, \ldots, n\}$, nous avons soit $[\![M]\!](t_i^+) = \mathbf{t}$ et $[\![M]\!](t_i^-) = \varepsilon$, soit $[\![M]\!](t_i^+) = \varepsilon$ et $[\![M]\!](t_i^-) = \mathbf{t}$. Par conséquent, la fonction suivante est une évaluation bien définie des variables :

$$V_M(x_i) = \begin{cases} \mathit{true} & \text{si } [\![M]\!](t_i^+) = \mathbf{t}, \\ \mathit{false} & \text{false } [\![M]\!](t_i^-) = \mathbf{t}. \end{cases}$$

V_M temoigne du fait que $\varphi \in \mathtt{SAT}_{\mathtt{ONE\text{-}IN\text{-}THREE}}$, en se reposant sur les exemples (3). □

6.3 Modèle d'apprentissage

L'impossibilité de tester la cohérence des dST2Ws nous oblige à restreindre notre algorithme à une forme où il peut décider, en temps polynomial dans la taille de l'échantillon, de ne pas sortir de résultat. Dans ce cas, l'algorithme produira le symbole spécial *Null* représentant l'absence de résultat. Pour éviter de tomber dans des algorithmes triviaux produisant *Null* à la moindre difficulté, il est indispensable d'assurer l'apprentissage d'un edST2W, pour toute transformation τ de la classe des $\mathcal{STW}s$ à partir d'un échantillon contenant assez d'information, appelé *échantillon caractéristique*. L'algorithme devant être polynomial en temps et taille dans la taille de l'échantillon, nous devons imposer l'existence d'un échantillon caractéristique dont le nombre d'éléments est bornée polynomialement par le nombre de classes d'équivalences de τ.

Un autre obstacle vient d'un problème inhérent aux $d\text{TA}^{\downarrow}s$, squelette des dST2Ws, qui ne peuvent pas être appris à partir d'exemples uniquement positifs. Il existe plusieurs moyens d'éviter ce problème (comme nous avons pu le voir dans les chapitres précédents) tels que l'ajout d'exemples négatifs sous forme d'arbres ou de chemins ne devant pas apparaitre dans le domaine du transducteur appris. Nous préférons avoir le domaine directement sous forme d'un $d\text{TA}^{\downarrow}$ D. L'apprentissage de ce domaine peut être le résultat d'un pré-apprentissage utilisant les méthodes existantes, ce qui intégré directement ici ne ferait que compliquer inutilement la définition d'un échantillon caractéristique et l'élaboration d'un algorithme d'apprentissage.

Une classe de transformation respectant chacune des propriétés précédentes est dite apprenable *avec restriction*. La suite de ce chapitre cherche à montrer le théorème suivant :

Théorème 18. *Toute transformation de la classe des $\mathcal{STW}$, représentée par un edST2W, est apprenable polynomialement en temps et taille.*

6.4 Algorithme d'apprentissage

Pour prouver le théorème 18 , nous proposons un algorithme d'apprentissage, proche de la construction d'un transducteur canonique. Cet algorithme ne prend pas une transformation en entrée mais un échantillon fini la représentant.

L'algorithme principal prend en entrée un échantillon S cohérent avec la transformation cible τ, et le domaine $dom(\tau)$ sous la forme d'un $d\text{TA}^{\downarrow}$ D.

L'algorithme 3 est divisé en deux parties principales. La première partie, de la ligne 1 à 13, cherche à identifier les classes d'équivalence sur les chemins de la transformation, et ainsi identifier les différents états du transducteur canonique. Pour cela, il créé une fonction *tat* associant à chaque chemin le chemin minimal appartenant à sa classe d'équivalence, partageant un même résiduel. Cette étape repose principalement sur le test $\simeq_{S,D}$, émulant l'équivalence de Myhill-Nérode $\equiv_\tau$ sur un ensemble fini d'exemples de τ. Nous pouvons remarquer que si $\simeq_{S,D}$ se comporte comme $\equiv_\tau$ et que chaque plus petit chemin représentant un résiduel de τ est également présent dans $chemins(dom(S))$, alors cette procédure produit exactement l'ensemble des états Q de $can(\tau)$. Nous reviendrons sur l'implémentation de $\simeq_{S,D}$ dans la suite de ce chapitre.

La seconde partie, à partir de la ligne 14, construit les autres éléments du transducteur canonique, à savoir, son axiome initial (ligne 14), et ses règles (lignes 15 à 19). Cette construction se base sur les fonctions décomp et résiduel, reposant toutes deux sur le calcul d'une décomposition inspirée de la définition de décomposition pour les $\mathcal{STW}s$, fonctions sur lesquelles on reviendras par la suite.

Finalement, l'algorithme renverra le transducteur ainsi créé, se gardant le droit de renvoyer *Null* si le transducteur n'est pas cohérent avec l'échantillon ou le domaine donné en entrée.

Si, comme nous le montrerons par la suite, les procédures $\simeq_{S,D}$, décomp, et résiduel existent et se calculent en temps polynomial dans la taille de S, alors le lemme suivant est trivial.

Lemme 38. *Pour tout échantillon S et $d\text{TA}^{\downarrow}$ D, $learner_D(S)$ produit un edST2W M en temps et taille polynomiale dans la taille de S, ou s'abstient de répondre (Null).*

6.5 Décompositions, résiduels et équivalence

L'algorithme d'apprentissage repose, comme le calcul de résiduels d'une transformation, sur la possibilité de décomposer l'échantillon. La fonction suivante prend en entrée un échantillon S, supposé être *représentatif* d'une trans-

Algorithme 2: Apprentissage d'un *ed*ST2W.

learner$_D(S)$
début
1 $P \coloneqq chemins(dom(S))$
2 $Q \coloneqq \varnothing$
3 état := **new** $hashtable\langle chemins, chemins\rangle()$
4 **tant que** $P \neq \varnothing$ **faire**
5 $p \coloneqq \min_{chemins}(P)$
6 $P' \coloneqq \{p' \in Q \mid p \simeq_{S,D} p'\}$
7 **si** $P' \neq \varnothing$ **alors**
(*p *can be merged**)
8 $P \coloneqq P \setminus \{p' \in P \mid p$ is prefix of $p'\}$
9 état$[p] \coloneqq \min_{chemins}(P')$
10 **sinon**
11 $P \coloneqq P \setminus \{p\}$
12 $Q \coloneqq Q \cup \{p\}$
13 état$[p] \coloneqq p$
14 $init \coloneqq Gauche(S) \cdot$ état$[\varepsilon] \cdot Droite(S)$
15 **pour** $p \in Q$ **faire**
16 **pour** $f \in \Sigma$ *tel que* $\exists i,\ p \cdot (f,i) \in chemins(dom(S))$ **faire**
17 **pour** $i \in 1, \ldots, k$ **faire**
$p_i =$ état$[p \cdot (f,i)]$
18 $(u_0, \_, u_1, \ldots, u_k) \coloneqq$ décomp(résiduel$(S,p), f)$
19 $p \xrightarrow{f} u_0 \cdot p_1 \cdot u_1 \cdot \ldots \cdot p_k \cdot u_k \in rul$
20 $M \coloneqq (\Sigma, \Delta, Q, init, rul)$
21 **si** $S \subseteq [\![M]\!]$ **et** $dom([\![M]\!]) \subseteq [\![D]\!]$ **alors**
22 **retourner** M
23 **sinon**
24 **retourner** *Null*

formation τ, et un symbole $f^{(k)}$; f étant la racine d'au moins un des arbres de S. À partir de cela, la fonction produira une séquence $(u_0, S_1, u_1 \ldots, S_k, u_k)$ qui idéalement sera la décomposition de S au regard de τ.

Fonction 3: Etape de décomposition.

$\mathsf{décomp}(S, f^{(k)})$:
début
1 $S_f \coloneqq \{(t, w) \in S \mid t$ a pour racine $f\}$
2 $s = f(s_1, \ldots, s_k) \coloneqq \min_{Arbre}(dom(S_f))$
3 **pour** $i \coloneqq 1, \ldots, k$ **faire**
4 $D_i \coloneqq \{t_i \mid f(s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_k) \in dom(S_f)\}$
5 $u_0 \coloneqq lcp(im(S_f))$
6 $w_s \coloneqq S(s)$
7 $prefixe_0 = u_0$
8 **pour** $i \coloneqq 1, \ldots, k$ **faire**
9 $prefixe_i \coloneqq lcp\{w \mid \exists t_{i+1}, \ldots, t_k.\ (f(s_1, \ldots, s_i, t_{i+1}, \ldots, t_k), w) \in S_f\}$
10 $suffixe_i \coloneqq prefixe_i^{-1} \cdot w_s$
11 $S'_i \coloneqq \varnothing$
12 **pour** $t \in D_i$ **faire**
13 $w \coloneqq prefixe_{i-1}^{-1} \cdot S(f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_k)) \cdot suffixe_i^{-1}$
14 $S'_i \coloneqq S'_i \cup \{(t, w)\}$
15 $u_i \coloneqq lcs(im(S'_i))$
16 $S_i \coloneqq \{(t, w \cdot u_i^{-1}) \mid (t, w) \in S'_i\}$
17 **retourner** $(u_0, S_1, u_1 \ldots, S_k, u_k)$

Le but de cette fonction est de construire une décomposition valide, ce qui repose avant tout sur une cohérence avec l'échantillon S. Plus formellement, on souhaite que $u_0 \cdot S_1(t_1) \cdot \ldots \cdot S_k(t_k) \cdot u_k = S(f(t_1, \ldots, t_k))$ pour chaque arbre de S_f, tout particulièrement pour le plus petit arbre de $s = f(s_1, \ldots, s_k)$ de S_f. Nous pouvons remarquer que cette décomposition n'est possible que s'il existe au moins un arbre dans l'ensemble S_f.

Le calcul des u_i, puis des sous échantillons, se fait itérativement de gauche à droite en se basant sur la production $w_s = S(s)$ associée au plus petit arbre. Le mot u_0 correspond à $Gauche(S_f)$. Ensuite, pour chaque i, $prefixe_i$ est construit (ligne 9) de sorte qu'il soit égal à $u_0 \cdot S_1(s_1) \cdot \ldots \cdot S_i(s_i) \cdot u_i$, ce qui s'obtient en calculant le préfixe commun de la sortie associée à chaque arbre $f(s_1, \ldots, s_i, t_{i+1}, \ldots, t_k)$. Si $prefixe_i$ contient bien la cible, il est facile de calculer $suffixe_i = S_{i+1} \cdot u_{i+1} \cdot \ldots \cdot S_k \cdot u_k = prefixe_i^{-1} \cdot w_s$ (ligne 10).

Grâce à ces chaines, nous pouvons, pour chaque arbre t, égal à $f(s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_k)$

de S_f, calculer la portion de chaîne correspondant à $S'_i(t_i) = S_i(t_i) \cdot u_1 = \mathit{prefixe}_{i-1}^{-1} \cdot S(t) \cdot \mathit{suffixe}_i^{-1}$ (ligne 11 à 14). En supposant que l'échantillon soit assez riche, notion que nous développerons dans la partie traitant de l'échantillon caractéristique, u_i s'obtient en calculant le suffixe commun à toute sortie de S'_i, i.e. $u_i = \mathit{lcs}(\mathit{im}(S'_i))$, nous permettant de calculer $S_i(t_i)$ (ligne 15 et 16).

Il faut souligner, une fois de plus, que chaque étape repose sur le fait que l'échantillon soit assez riche pour que les différents préfixes et suffixes puissent être calculés correctement (en nous basant sur les propriétés propres aux $\mathcal{STW}s$). Si ce n'est pas le cas, la fonction arriverait tout de même à produire un résultat, possiblement non cohérent avec S_f, en temps polynomial, erreur qui serait détectée dans le test final de l'algorithme principal.

Une fois la fonction décomp obtenue, le résiduel $p^{-1}S$ se définit récursivement, de la manière que $p^{-1}\tau$ est calculé pour une transformation $\mathcal{STW}$ τ. Après avoir réduit l'échantillon, il suffit de suivre le chemin p en appliquant la fonction décomp.

Fonction 4: Construction de résiduels.

résiduel(S, p) :
début
1 $S' := \{(t, w') \mid (t, w) \in S,\ w' = \mathit{Gauche}(S)^{-1} \cdot w \cdot \mathit{Droite}(S)^{-1}\}$
2 **tant que** $p \neq \varepsilon$ **faire**
3 Soit $p = (f^{(k)}, i) \cdot p'$
4 $(u_0, S_1, u_1, \ldots, S_k, u_k) :=$ décomp(S', f)
5 $S' := S_i$
6 $p := p'$
7 **retourner** S

La fonction décomp s'exécutant en temps polynomial dans la taille de S, il en va de même pour le calcul du résiduel.

Maintenant que nous avons pu identifier les résiduels d'un échantillon, il est possible d'émuler la relation d'équivalence $\equiv_\tau$ d'une transformation $\mathcal{STW}$. Nous rappelons que deux chemins p_1 et p_2 sont équivalents dans τ, $p_1 \equiv_\tau p_2$, si et seulement si $p_1^{-1}\tau = p_2^{-1}\tau$, cette égalité reposant tant sur le domaine que sur la transformation définie. Pour ce qui est de l'apprentissage, $S_1 =$ résiduel(S, p_1) et $S_2 =$ résiduel(S, p_2) sont représentés par des échantillons finis, et le fait qu'ils ne partagent pas un même domaine n'implique pas qu'ils ne soient pas équivalents. Le prédicat $\simeq_{S,D}$ repose donc à la fois sur l'échantillon, vérifiant qu'il n'y ai pas deux transformations possibles pour un même arbre, et sur le domaine donné par un $d\mathrm{TA}^{\downarrow}$ D. Plus formellement,

Définition 47. *Pour tout échantillon S, $d\text{TA}^{\downarrow}$ D, et pour tous chemins p_1, $p_2 \in chemins(dom(S))$, $p_1 \simeq_{S,D} p_2$ si :*
- $p_1^{-1}D = p_2^{-1}D$,
- $\forall t \in dom(S_1) \cap dom(S_2)$, $S_1(t) = S_2(t)$.

Une fois de plus toutes les opérations nécessaires à ce prédicat se calculent en temps polynomial dans la taille de D et S.

L'ensemble de l'apprentissage repose sur le choix de l'échantillon. Dans la section suivante nous allons voir quelles propriétés sur les échantillon peuvent assurer l'apprentissage d'un transducteur, et d'une transformation, cible.

6.6 Echantillon caractéristique

Avant de décrire la construction la notion d'échantillon caractéristique d'une transformation pour la classe des $\mathcal{STW}$ remarquons quelques propriétés des ordres pour les chemins, arbre, et contextes nécessaires par la suite :

1. $<_{chemins}$ est compatible avec la relation de préfixe i.e., $p <_{chemins} p \cdot p'$ pour tout chemin p et tout chemin non vide p'.
2. $<_{\mathcal{T}}$ est compatible avec la taille i.e., $t_1 <_{\mathcal{T}} t_2$ si $|t_1| < |t_2|$.
3. $<_{\mathcal{T}}$ est compatible avec la structure de l'arbre i.e., $C[t_1] <_{\mathcal{T}} C[t_2]$ pour $t_1 <_{\mathcal{T}} t_2$.

Pour le reste de cette section, il est également nécessaire d'introduire quelques notations. Par τ, nous représentons une transformation $\mathcal{STW}$. Sans instancier réellement le transducteur $ed\text{ST2W}$ canonique $can(\tau)$ de τ, il est utilisé implicitement pour définir la notion de relation d'équivalence de Myhill-Nerode $\equiv_\tau$ sur les chemins de τ. La classe d'équivalence d'un chemin $p \in chemins(dom(\tau))$ suivant $\equiv_\tau$ est notée $[p]_\tau$. Nous identifions par A_τ le $d\text{TA}^{\downarrow}$ canonique représentant le domaine $dom(\tau)$. Il est également nécessaire d'identifier, pour chaque chemin $p \in chemins(dom(\tau))$, le contexte minimal du domaine de τ ayant un trou en p, que l'on note c_p. Plus précisément pour chaque chemin $p \in chemins(dom(\tau))$:

$$c_p = \min_{Ctx}(\{c \in C_\Sigma \mid c \text{ a son } x \text{ en } p \text{ et } \exists t \in T_\Sigma \text{ tel que } c[t] \in dom(\tau)\ ).$$

L'algorithme traitant les chemins dans un ordre compatible avec $<_{chemins}$, ce représentant est le plus petit chemin menant à ce résiduel. Formellement, pour tout résiduel $p^{-1}\tau$, le représentant est :

$$min_{chemins}(\{p' \in chemins(dom(\tau)) \mid p' \equiv_\tau p\}).$$

Le prédicat $\simeq_{S,D}$ permet de vérifier si ce chemin appartient à une classe d'équivalence déjà représentée, dans quel cas il est associé au chemin et à la

classe correspondante. Si ce chemin n'appartient pas à une classe pré-existante, il est *représentatif* d'un nouveau résiduel. L'échantillon doit être cohérent, dans tous les cas, avec la transformation cible. Le rôle de l'échantillon caractéristique va être principalement d'assurer que les inégalités de la relation d'équivalence $\equiv_\tau$ se retrouvent dans $\simeq_{S,D}$. Il est nécessaire que tout chemin représentant un nouveau résiduel ne puisse être rattaché pendant l'apprentissage à un résiduel pré-existant.

L'algorithme de test se base sur les raisonnements suivants : deux chemins p et p_0 ne sont pas équivalents si et seulement si les résiduels $p^{-1}\tau$ et $p_0^{-1}\tau$ ont un domaine différent, ou s'ils autorisent deux sorties différentes pour un même arbre (du domaine commun), i.e. $p^{-1}\tau(t) \neq p_0^{-1}\tau$. La première condition peut être facilement contrôlée à l'aide de l'automate représentant le domaine ; la deuxième suppose qu'il existe un contre exemple dans l'échantillon.

Avant d'entrer plus dans les détails, identifions quels ensembles de chemins nécessitent de tels contre-exemples. Dans un premiers temps nous définissons l'ensemble des plus *courts chemins* atteignant chaque résiduel de τ :

$$resChemins(\tau) = \{\min_{chemins}([p]_\tau) \mid p \in chemins(dom(\tau))\}.$$

Les chemins de l'ensemble $resChemins(\tau)$ peuvent être également vus comme les chemins minimaux représentant les différents états de $can(\tau)$.

Nous définissons également les *chemins étendus* de τ, ajoutant aux chemins de $resChemins(\tau)$ leurs extensions d'un pas

$$extChemins(\tau) = resChemins(\tau) \cup \{p \cdot (f,i) \in chemins(dom(\tau)) \mid p \in resChemins(\tau)\}.$$

Intuitivement, cet ensemble représente chaque état de $can(\tau)$ ainsi que les transitions qui en sortent.

Pour assurer que l'échantillon caractéristique permette à l'algorithme de marcher efficacement, il faut, pour chaque paire de chemins $p, p_0 \in extChemins(\tau)$ ne partageant une même classe d'équivalence, des contre-exemples appropriés. Nous pouvons remarquer que l'algorithme ne considérera pas d'autres chemins que ceux de $extChemins(\tau)$ car toute continuation de p_0, chemin considéré, est rejeté dès lors que p_0 est fusionné à p. En effet, aucune des continuations $p_0 \cdot w$ possibles ne pourra représenter un résiduel puisque le chemin $p \cdot w$, plus petit, mène déjà à ce résiduel.

Pour finir, l'algorithme mime la décomposition d'une transformation pour identifier les règles de transitions pour un résiduel donné. L'échantillon caractéristique nécessite donc un ensemble d'exemples assez riche pour identifier toutes les transitions entre les différents résiduels, i.e. les transitions entre les états de $can(\tau)$.

Nous pouvons maintenant identifier l'ensemble d'exemples nécessaire pour décrire chaque résiduel de τ. Même si la sortie joue un rôle prédominant dans

la conception de cet échantillon, nous cherchons ici à identifier uniquement l'ensemble d'arbres d'entrée ; la sortie pouvant être retrouvée, la transformation τ étant connue.

Définition 48. *Prenons une transformation τ de $\mathcal{STW}$, un chemin de $p \in extChemins(\tau)$, et un ensemble d'arbres $T \subseteq T_\Sigma$. L'ensemble T est dit* significatif *pour τ au regard de p si et seulement si les conditions suivantes sont respectés :*

(REP) *Pour tout $p_0 \in resChemins(\tau)$ tel que l'ensemble*

$$T_{p,p_0} = \{t \in dom(p^{-1}\tau) \cap dom(p_0^{-1}\tau) \mid p^{-1}\tau(t) \neq p_0^{-1}\tau(t)\}$$

est non vide, l'arbre $\min_{Arbre}(T_{p,p_0})$ appartient à T.

L'ensemble T est dit représentatif structurellement *pour τ au regard de p si et seulement si les conditions suivantes sont respectées :*

($\mathbf{RS_0}$) *L'arbre $\min_{Arbre}(dom(p^{-1}\tau))$ appartient à T ;*

($\mathbf{RS_1}$) *$lcp(p^{-1}\tau(T)) = \varepsilon$ et $lcs(p^{-1}\tau(T)) = \varepsilon$;*

($\mathbf{RS_2}$) *$lcp(im(p^{-1}\tau) \setminus \{\varepsilon\}) = lcp(p^{-1}\tau(T) \setminus \{\varepsilon\})$.*

Pour un chemin p, la condition **(REP)** permet d'identifier l'ensemble d'arbres assurant que le test d'équivalence entre chemins de $extChemins(\tau)$ correspond à $\equiv_\tau$. Les conditions **($\mathbf{RS_0}$)**, **($\mathbf{RS_1}$)** et **($\mathbf{RS_2}$)** assurent que l'ensemble T contient les éléments nécessaires à une bonne décomposition des transformations résiduelles pour tout chemin $p \in resChemins(\tau)$, ce que nous prouverons par la suite. Il reste à s'assurer de la présence des arbres de T dans l'échantillon, en les intégrant dans les contextes adaptés.

Définition 49. *Soit τ une transformation $\mathcal{STW}$, un chemin $p \in extChemins(\tau)$, et un échantillon $S \subseteq T_\Sigma \times \Delta^*$. S est dit* caractéristique *pour τ et un chemin p si $S \subseteq p^{-1}\tau$, et pour tout chemin p_0, tel que $p \cdot p_0 \in extChemins(\tau)$, l'ensemble d'arbres $c_{p_0}^{-1}dom(S)$ est discriminant et structurellement représentatif pour τ et le chemin $p \cdot p_0$.*

Un échantillon S est caractéristique *pour une transformation τ de la classe $\mathcal{STW}$ ssi S est caractéristique pour τ et le chemin ε.*

L'algorithme d'apprentissage doit avoir une certaine robustesse dans son apprentissage en n'acceptant plus que l'échantillon caractéristique. Tout échantillon contenant l'échantillon caractéristique d'une transformation τ, cohérent avec τ, reste caractéristique pour τ. Cette propriété est formalisée par la proposition suivante.

Proposition 28. *Soit S un échantillon caractéristique pour $\tau \in \mathcal{STW}$. Tout échantillon S' étendant S en restant cohérant avec τ, i.e. $S \subseteq S' \subseteq \tau$, est caractéristique pour τ.*

Le lemme suivant assure que l'existence d'un échantillon caractéristique dont la cardinalité ne devient pas trop importante, et reste apprenable.

Lemme 39. *Pour tout ed*ST2W *M, il existe un échantillon $carac_M$ caractéristique pour $[\![M]\!]$ de cardinalité polynomiale dans la taille de M.*

La preuve de ce lemme repose sur la construction du plus petit échantillon caractéristique et de comparer sa cardinalité au nombre de résiduels distincts de $[\![M]\!]$. La propriété **(REP)** nécessite, au plus, un nombre quadratique d'arbres. $\mathbf{(RS_0)}$ nécessite exactement un arbre par résiduel. $\mathbf{(RS_1)}$ quant à elle en nécessite quatre, assurer la valeur d'un *lcp* ou d'un *lcs* nécessitant deux exemples. Et la condition $\mathbf{(RS_2)}$ en nécessite deux. Par conséquent, il existe un échantillon $carac_M$ caractéristique pour $[\![M]\!]$ de cardinalité polynomiale dans la taille de $can([\![M]\!])$ (lemme 27), et donc dans la taille de M ; M étant de taille obligatoirement supérieure ou égale à $can(\tau)$ (Corollaire5).

Il reste à montrer que l'échantillon caractéristique d'une $\mathcal{STW}$ τ assure un bon déroulement de l'algorithme d'apprentissage menant à la production de $can(\tau)$. Pour cela, il faut assurer le lien entre les conditions $\mathbf{(RS_1)}$ et $\mathbf{(RS_2)}$, ne vérifiant que des propriétés locales sur les préfixes et suffixes communs, et les divers décompositions utilisées pour le calcul de résiduels. Nous procéderons par lemmes successifs prouvant certaines propriétés sur les échantillons manipulés pendant l'évaluation de l'algorithme.

Lemme 40. *Si S est un échantillon caractéristique d'une $\mathcal{STW}$ τ (ayant un indice de Myhill-Nerode fini) alors $S' = Gauche(S)^{-1} \cdot S \cdot Droite(S)^{-1}$ est un échantillon caractéristique pour $Noyau(\tau)$.*

Ce lemme repose sur la condition $\mathbf{(RS_2)}$ appliquée à ε puisque, comme on peut le remarquer facilement, $\varepsilon^{-1}\tau = Noyau(\tau)$. Cela implique naturellement que $Gauche(S) = Gauche(\tau)$ et $Droite(S) = Droite(\tau)$.

Intéressons nous maintenant à la procédure de décomposition. L'étape cruciale de cette procédure repose sur le lemme suivant, montrant principalement que chaque *lcp*, requis par décomp, est calculé correctement pour un échantillon caractéristique :

Lemme 41. *Prenons S un échantillon caractéristique de τ, une transformation $\mathcal{STW}$, et du chemin $p \in resChemins(\tau)$, et $(u_0, \tau_1, \ldots, \tau_k, u_k)$ la décomposition de $p^{-1}\tau$ pour $f \in \Sigma^{(k)}$. Pour tout $i \in \{1, \ldots, k\}$ nous obtenons donc que*

$$lcp(\{\tau_i(t_i)\cdot u_i \cdot \ldots \cdot \tau_k(t_k)\cdot u_k \mid t_i \in (f,i)^{-1}dom(S), \ldots, t_k \in (f,k)^{-1}dom(S)\}) = \varepsilon.$$

Preuve. Remarquons tout d'abord que, S étant caractéristique pour τ et p appartenant à $resChemins(\tau)$, $(f,i)^{-1}dom(S)$ est discriminant pour τ et $p{\cdot}(f,i)$, pour tout $1 \leq i \leq k$. En particulier, $s = \min_{Arbre}(dom(S)) = \min_{Arbre}(p^{-1}dom(\tau))$ et de la même manière $s_i = \min_{Arbre}((f,i)^{-1}dom(S)) = \min_{Arbre}(p.(f,i)^{-1}dom(\tau))$ pour $1 \leq i \leq k$.

Nous pouvons maintenant prouver ce lemme par induction sur i, en allant de k à 1. Pour la suite de la preuve, le mot w_i représente $w_i = lcp(\{\tau_i(t_i){\cdot}u_i{\cdot}\ldots{\cdot}\tau_k(t_k)\cdot u_k \mid t_j \in (f,j)^{-1}dom(S)\})$. Nous allons montrer que $q_{i+1} = \varepsilon$ implique que $w_i = \varepsilon$. Pour cela, nous allons nous concentrer sur deux cas dépendant de la valeur de $lcp(im(\tau_i) \setminus \{\varepsilon\})$.

Pour le premier cas, supposons que $lcp(im(\tau_i) \setminus \{\varepsilon\}) = \varepsilon$. En nous basant sur $\mathbf{(RS_2)}$, on obtient que $lcp(im(\tau_i) \setminus \{\varepsilon\}) = lcp(im((f,i)^{-1}(S)) \setminus \{\varepsilon\}) = \varepsilon$. Par conséquent, il existe deux mots non vides $w_1, w_2 \in im((f,i)^{-1}(S))$ tels que $lcp(w_1,w_2) = \varepsilon$, et les deux arbres t_1 et t_2 respectifs dans $(f,i)^{-1}dom(S)$ tels que $\tau_i(t_1) = w_1 \neq \varepsilon$ et $\tau_i(t_2) = w_2 \neq \varepsilon$. Cela implique que $w_i = lcp(w_1 \cdot u_i \cdot w_{i+1}, w_2 \cdot u_i \cdot w_{i+1}, \ldots) = \varepsilon$.

Pour ce qui est du second cas, on suppose que $lcp(im(\tau_i) \setminus \{\varepsilon\}) = w \neq \varepsilon$. En utilisant une fois de plus $\mathbf{(RS_2)}$, on obtient que $lcp(im(\tau_i) \setminus \{\varepsilon\}) = lcp(im((f,i)^{-1}(S)) \setminus \{\varepsilon\}) = w$. D'autre part, grâce à $\mathbf{(RS_1)}$, on sait que $lcp(im(\tau_i)) = lcp(im((f,i)^{-1}(S))) = \varepsilon$. Par conséquent, $\varepsilon \in im(\tau_i)$ et $\varepsilon \in im((f,i)^{-1}(S))$. La propriété $\mathbf{(C2)}$ des décompositions nous dit que $lcp(im(\tau_j){\cdot}u_j{\cdot}\ldots{\cdot}im(\tau_k){\cdot}u_k) = \varepsilon$ pour tout $1 \leq j \leq k$, tout particulièrement pour i et $i+1$. Cela implique que $lcp(im(\tau_i){\cdot}u_i{\cdot}\ldots{\cdot}im(\tau_k){\cdot}u_k)$ est un préfixe de $lcp(w,u_i) = \varepsilon$. Donc, $w_i = lcp(w,u_i) = \varepsilon$.

□

Lemme 42. *Soit S un échantillon caractéristique de la transformation $\tau \in \mathcal{STW}$ pour un chemin $p \in resChemins(\tau)$ et $(u_0,\tau_1,\ldots,\tau_k,u_k)$ la décomposition de $p^{-1}\tau$ pour $f \in \Sigma^{(k)}$, alors $\mathsf{décomp}(S,f) = (u_0,S_1,u_1,\ldots,S_k,u_k)$, où chaque ensemble S_i est un échantillon caractéristique pour τ et $p\cdot(f,i)$, pour tout $i \in \{1,\ldots,k\}$.*

Preuve. Prenons M, un edST2W représentant τ, ayant q_0 pour état initial. Considérons la règle $q_0 \xrightarrow{f} u_0''{\cdot}q_1{\cdot}\ldots{\cdot}q_k{\cdot}u_k''$. La proposition 20 nous permet d'en déduire que $u_i'' = u_i$ et $[\![M]\!]_{q_i} = \tau_i$. Maintenant, notons $(u_0',S_1,u_1',\ldots,S_k,u_k')$ le résultat de $\mathbf{décomp}(S,f)$. Dans un premier temps nous montrons récursivement que $u_i' = u_i'' = u_i$ pour i variant de 1 à k.

Nous pouvons remarquer que $p \in resChemins(\tau)$ implique que $p\cdot(f,i) \in extChemins(\tau)$ pour tout $1 \leq i \leq k$. Soit $p_i = p{\cdot}(f,i)$, rappelons que l'on note c_{p_i} le plus petit contexte de $dom(\tau)$ ayant un trou en p_i. Par définition, S est caractéristique pour τ et p ce qui implique que $c_{p_i}^{-1}dom(S)$ est discriminant pour τ et p_i. Particulièrement, cet ensemble respecte la propriété $\mathbf{(RS_0)}$ nous disant

que chaque arbre s_i identifié par la fonction **décomp** est le plus petit arbre de $((p \cdot (f,i))^{-1} dom(\tau))$. Par conséquent, $p^{-1}c_{p_i} = f(s_1, \ldots, s_{i-1}, \bot, s_{i+1}, \ldots, s_k)$.

Nous pouvons maintenant débuter la récursion. Par construction, $u_0 = lcp(\{w \mid (f(t_1, \ldots, t_k), w) \in S\})$. Puisque $S \in p^{-1}\tau$, $u'_0 = lcp(\{p^{-1}\tau(f(t_1, \ldots, t_k)) \mid f(t_1, \ldots, t_k) \in dom(S)\})$, et u_0 est un préfixe de u'_0, ce qui en utilisant le lemme 41, nous donne que $u'_0 = u_0$.

En se basant sur une même approche et sur le lemme 41, nous pouvons remarquer que chaque $prefixe_i$ calculé par décomp est égal à $u_0 \cdot \tau_1(s_1) \cdot \ldots \cdot \tau_i(s_i) \cdot u_i$, ce qui a pour conséquence directe que $suffixe_i = \tau_{i+1}(s_{i+1}) \cdot \ldots \cdot \tau_k(s_k) \cdot u_k$.

Ainsi, pour tout $t \in D_i$, si $(t,w) \in S'_i$ alors $w = \tau_i(t) \cdot u_i$. Puisque $p^{-1}c_{p\cdot(f,i)} = f(s_1, \ldots, s_{i-1}, \bot, s_{i+1}, \ldots, s_k)$, S est caractéristique pour τ pour le chemin p, et $p \cdot (f,i) \in extChemins(\tau)$, ce qui implique que D_i est représentatif structurellement pour τ et $p \cdot (f,i)$. En particulier, la condition **($\mathbf{RS_1}$)** nous apporte que $lcs(\{\tau_i(t) \mid t \in D_i\}) = \varepsilon$. Par conséquent, $lcs(\{w \mid (t,w) \in S'_i\}) = u_i$.

Nous pouvons également en déduire que $(t,w) \in S_i$ implique que $w = \tau_i(t)$ et donc que S_i est cohérent avec $(p \cdot (f,i))^{-1}\tau$. Les autres propriétés suivent naturellement. S étant caractéristique pour τ et p, pour tout $p' \in extChemins$ tel que p est un préfixe de p', $c_{p'}^{-1} dom(S)$ est discriminant et représentatif structurellement, en particulier pour tout p' préfixé par $p \cdot (f,i)$. Par conséquent, S_i est un échantillon caractéristique pour τ et $p \cdot (f,i)$. □

Le lemme suivant est donc naturellement une conséquence directe du lemme 42.

Lemme 43. *Pour S un échantillon caractéristique de $\tau \in \mathcal{STW}$, pour tout $p \in extChemins(\tau)$, le résultat de résiduel(S,p) est un échantillon caractéristique pour τ et p.*

Maintenant que le calcul des décompositions et résiduels est vérifié, il reste à s'assurer que le prédicat $\simeq_{S,D}$ équivaut à la relation $\equiv_\tau$ pour les chemins représentatifs, si S est caractéristique pour τ.

Lemme 44. *Considérons une $\mathcal{STW}$ τ, S un échantillon caractéristique de τ, et D un $d\text{TA}^{\downarrow}$ décrivant le domaine de τ, i.e. $dom(\tau) = [\![D]\!]$. Alors, pour tous chemins $p_1, p_2 \in extChemins(\tau)$, $p_1 \simeq_{S,D} p_2$ si et seulement si $p_1 \equiv_\tau p_2$.*

Preuve. Les chemins p_1 et p_2 étant dans $extChemins(\tau)$, et en prenant $S_1 =$ résiduel(S,p_1) et $S_2 =$ résiduel(S,p_2), le lemme 43 implique que S_1 (resp. S_2) est caractéristique pour p_1 et τ (resp. p_2 et τ). Par définition de la relation d'équivalence $\equiv_\tau$, $p_1 \equiv_\tau p_2$ si et seulement si $p_1^{-1}\tau$ et $p_2^{-1}\tau$ sont égaux, et particulièrement si ils partagent un même domaine. Le domaine est testable directement à l'aide de D, et la condition **(REP)** nous assure que si deux résiduels ne sont pas équivalent alors S_1 et S_2 contiennent tous les deux dans leur domaine le plus petit arbre t tel que $p_1^{-1}\tau(t) \neq p_2^{-1}\tau(t)$. □

Si l'échantillon donnée en entrée de l'algorithme est caractéristique pour une transformation τ, le lemme suivant montre que le résultat obtenu est bien $can(\tau)$.

Lemme 45. *Considérons une $\mathcal{STW}$ τ, S un échantillon caractéristique de τ, et D un $d\text{TA}^{\downarrow}$ capturant le domaine de τ, alors $learner_D(S)$ renvoie $can(\tau)$.*

Preuve. La preuve se divise en trois parties, à l'image de l'algorithme. La première partie permet de créer l'ensemble des états $Q = resChemins(\tau)$. Cette égalité vient du fait que, pendant tout le processus, les invariants suivants sont respectés : $P \subseteq extChemins(\tau)$, et $Q \subseteq resChemins(\tau)$. Ceci est trivial au début de l'algorithme. Et en se reposant sur le fait que $p \simeq_{S,D} p' \Leftrightarrow p \equiv_\tau p'$ quand p et p' appartiennent à $extChemins(\tau)$ (d'après le lemme 44). L'ensemble *resChemins* étant clos par préfixe, tous les éléments de $resChemins(\tau)$ ont étés vus, et ceci dans l'ordre croissant, donc $Q = resChemins(\tau)$. Tout chemin p appartenant à $extChemins(\tau)$ tel que $p \notin resChemins(\tau)$ a déjà le représentant de sa classe d'équivalence présent dans Q ce qui implique que toutes ces continuations sont ignorées, et que P reste toujours un sous ensemble de $extChemins(\tau)$.

La deuxième étape créé l'axiome initial *init* du transducteur canonique par le lemme 40. Enfin, le lemme 42 assure que chaque règle de M correspond à une règle de $can(\tau)$, ce qui mène à la construction de $can(\tau)$. □

L'algorithme d'apprentissage suivant découle naturellement des lemmes qui le précédent.

Théorème 19. *Les transformations de la classe $\mathcal{STW}$, représentables par des edST2Ws, sont apprenables avec restriction en temps et taille polynomiaux dans la taille de l'échantillon d'apprentissage.*

Preuve. L'algorithme est clairement polynomial. Quand l'échantillon S est caractéristique pour une transformation τ, alors $learner(S, dom(\tau)) = can(\tau)$ d'après le lemme 45. De plus, le lemme 39 assure qu'il existe un échantillon caractéristique de cardinalité polynomiale dans la taille de tout edST2W représentant τ. Enfin, les lignes 29 à 32 de l'algorithme garantissent que l'algorithme retourne un transducteur cohérent avec l'entrée de l'algorithme. □

Chapitre 7
Conclusion

Le but de cette thèse était l'apprentissage d'une classe de transduction permettant de transformer des arbres en mots à l'image de transformations XSLT. L'apprentissage, tel que nous le concevons, repose sur l'existence d'une forme normale unique pour chaque transducteur, la cible de l'apprentissage, et une décidabilité efficace de l'équivalence.

Dans un premier temps, en nous concentrant sur les modèles des NW2Ws, transducteurs de mots imbriqués en mots, nous nous sommes intéressés au problème d'équivalence. En réduisant ce problème à celui de morphismes sur des grammaires algébriques, connu décidable en temps polynomial, nous montrons dans un premier temps que l'équivalence de cette classe est décidable elle aussi en temps polynomial. De plus l'approche utilisée permet d'étendre ce résultat à trois autres classes de transductions séquentielles, sans copie ni réordenancement. La réduction se base essentiellement sur la représentation de l'exécution ne dépendant ni de l'ordre du parcours de l'arbre, ascendant ou descendant, ni de la manière dont cet arbre est représenté, arbre d'arité bornée ou mots imbriqués.

L'équivalence étant maintenant prouvée décidable en temps polynomial pour plusieurs classes de transductions, particulièrement les dST2Ws. Nous avons pu passer à la seconde étape qu'est la définition d'une forme normale assurant, pour chaque transduction de cette classe, l'existence d'un unique transducteur normalisé minimal la représentant. Pour cela nous avons défini les dST2Ws travailleurs, produisant la sortie le plus tôt possible dans la transduction. Nous avons ensuite proposé un algorithme de normalisation, applicable à tout dST2W, permettant de générer un edST2W équivalent en temps polynomial dans la taille du transducteur normalisé. Cette normalisation repose principalement sur la combinatoire de mots et a nécessité l'élaboration d'une méthode efficace de transfert de mots à travers des grammaires. Nous avons également pu établir que la taille d'un dST2W normalisé atteindra au plus une taille double exponentielle dans celle du transducteur de base. Cette explosion vient en partie du fait que tout, ou la majorité, de la production d'un transducteur peut se retrouver projetée dans une seule règle. Grâce à l'introduction d'un algorithme de minimisation de tout edST2W en temps polynomial dans la taille de ce transducteur, nous avons donc assurer l'existence, pour chaque transduction de la classe des dST2Ws, d'un représentant

canonique. ce représentant est la cible de notre futur apprentissage.

Nos résultats nous mènent à un modèle d'apprentissage pour les transformation d'arbres en mots. Après avoir identifié la classe de transformation liée aux dST2Ws en donnant la caractérisation sémantique, les $\mathcal{STW}s$, nous introduisons une relation d'équivalence de Myhill-Nerode pour les résiduels de cette transformation. Cela permet d'adapter le théorème de Myhil-Nerode aux dST2Ws, assurant que toute transformation $\mathcal{STW}$, dont l'index de Myhill-Nerode est fini, peut être représenté par un dST2W et particulièrement par un edST2W canonique. Par le fait de notre limitation aux transformations séquentielles, certains échantillons d ?entrée, même fonctionnels, *peuvent* ne pas être représentables par un edST2W. Nous avons prouvé que décider de l'existence d'un dST2W représentant un échantillon est $NP-complet$. Nous avons donc défini un algorithme d'apprentissage qui, à partir d'un échantillon de couples entrée-sortie et d'un automate d'arbre déterministe représentant le domaine, produit l'unique edST2W minimal représentant la transformation correspondante, ou abandonne (en temps polynomial dans celui de l'échantillon). Nous avons également défini la notion d'échantillon caractéristique représentant un edST2W, et assurant que le résultat de notre apprentissage est bien ce même transducteur pour tout échantillon contenant l'échantillon caractéristique et consistant avec le domaine.

7.1 Perspectives

Comme nous avons pu le remarquer dans le deuxième chapitre, survolant les différents modèles de transducteurs existants, l'intérêt de ce domaine est récent et laisse de nombreuses questions ouvertes. Dans cette thèse nous nous sommes focalisés principalement sur les problèmes de production de sorties sous formes textuelles à partir de données arborées, et de décomposition de ces sorties pour apprendre des transducteurs permettant de représenter ces transformations. Pour cela, nous nous sommes restreints à une utilisation séquentielle de l'entrée ce qui réduit l'expressivité de notre modèle face aux transducteurs d'arbres à arbres qui permettent la restructuration totale de l'entrée pendant la transduction. Maintenant que le traitement de la sortie apparait plus clairement, ainsi que toutes les difficultés qu'elle génère, il serait intéressant de lever peu à peu ces restrictions fortes sur le modèle, en permettant dans un premier temps un réordonnancement de l'entrée dans son utilisation pour la production.

En restreignant les transducteurs de mots imbriqués en mots (NW2Ws) nous obtenons, comme nous avons pu le constater, une expressivité très proche de celles des dST2Ws. L'existence d'une forme normale et d'un possible ap-

prentissage reste une question ouverte. Un des intérêts d'adapter ces résultats aux NW2W*s* est que ces transducteurs peuvent s'appliquer aux documents XML et permettre de produire une sortie au fur et à mesure, ce qui peut être un gain considérable dans le traitement de fichiers volumineux. Ce n'est pas le cas des *d*ST2W*s* dont certaines productions doivent attendre le traitement de sous arbres avant de pouvoir être produits, et doivent donc être gardés en mémoire.

Un travail doit être également effectué sur la représentation de la sortie au sein des transducteurs. Comme nous avons pu le voir, la taille d'un *d*ST2W après normalisation peut exploser exponentiellement. Ceci est principalement dû à la taille que peut atteindre une production réunie au sein d'une même règle. Cette production était présente dans le *d*ST2W de base et pourrait être représentée par une grammaire singleton, i.e. ne représentant qu'un et unique mot.

Les résultats théoriques essentiels étant posés et identifiés, il est maintenant possible d'étudier avec une base fondamentale forte la mise en pratique de nos solutions. Cela permettra également d'observer si les situations menant aux différents problèmes identifiés, et à des explosions exponentielles dans les tailles des transducteurs manipulés, se retrouvent réellement dans ces cas pratiques.

Table des figures

1 Exemple de transformation XSLT basique 3
2 Exemple d'exécution d'un transducteur séquentiel et sa version normalisée 8

1.1 Arbre syntaxique de "$aabb$" par G_3 15
1.2 Automate de mots sur $\{a, b\}$ et sa représentation. Le fait que q_0 est initial et final est représenté respectivement par la flèche entrante et sortante. L'automate reconnait l'ensemble des mots contenant un nombre pair de "a". 16
1.3 Automate A_5 ascendant déterministe et son exécution représentée en bleu. 23
1.4 L'encodage frère-fils. 27
1.5 Automate stepwise pour l'exemple de la figure 1.4 28
1.6 L'encodage frère-fils. 30
1.7 Exemple de mot imbriqué et de la sequence d'arbres d'arité non-bornée correspondante 32
1.8 Parcours en profondeur d'un arbre t et sa linéarisation 32
1.9 Règles et exemple d'exécution de N_{10} 34

2.1 Transducteur de mots $M_{2.1}$: copie de chaque mot se terminant par un a et effacement toutes les occurrences de "a" dans un mot finissant par un "b". 40
2.2 Suppression des ε-transitions 43
2.3 TM $M_{2.3}$ et sa version en lecture lente 45
2.4 règles de compositions 46
2.5 $M_{2.3}$ réduit au domaine $\mathcal{L}(A)$ 49
2.6 Exemples de dTMs 51
2.7 Transducteur non normalisé 56
2.8 Exemple de projection de la sortie 57
2.9 Détection et représentation des groupes connexes 58
2.10 Automate normalisé 60
2.11 DFA et sa version minimale 62
2.12 $M_{2.6(b)}$ en forme normale minimale 71
2.13 transducteur avec anticipation N_1 pour $M_{2.1}$ 77
2.14 arbre et le résultat de la transduction M_{29} 83
2.15 transduction M_{32} 89
2.16 Exemple d'annotation et transduction par N_{33} 92
2.17 Exemple d'exécution de M_{34} sur l'arbre t. 94

2.18 Exemple de transduction de mots imbriqués en mots 101

3.1 Macro-transducteur de l'exemple 34 en XSLT 113

4.1 Relations liants les différentes classes de transductions 123

5.1 Exemple d'exécution de M_{38} et sa version « travailleuse » . . . 127

Bibliographie

Roger Alur et Loris D'Antoni : Streaming tree transducers. *In* Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts et Roger Wattenhofer, éditeurs : *ICALP (2)*, volume 7392 de *Lecture Notes in Computer Science*, pages 42–53. Springer, 2012. ISBN 978-3-642-31584-8. 21, 104

Rajev Alur et P. Madhusudan : Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009. 31, 33

Marie-Pierre Beal et Olivier Carton : Computing the prefix of an automaton. 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.9094`. 57, 58, 60, 61

Jean Berstel : *Transductions and Context-Free Languages*. Teubner Studienbucher, 1979. 37

Adrien Boiret, Aurélien Lemay et Joachim Niehren : Learning rational functions. *In* Hsu C. Yen et Oscar H. Ibarra, éditeurs : *Developments in Language Theory*, volume 7410 de *Lecture Notes in Computer Science*, pages 273–283. Springer, 2012. ISBN 978-3-642-31652-4. 80

Anne Brüggemann-Klein : Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, novembre 1993. 30

Julien Carme, Joachim Niehren et Marc Tommasi : Querying unranked trees with stepwise tree automata. *In 19th International Conference on Rewriting Techniques and Applications*, volume 3091 de *Lecture Notes in Computer Science*, pages 105–118. Springer Verlag, 2004a. 26

Julien Carme, Joachim Niehren et Marc Tommasi : Querying unranked trees with stepwise tree automata. *In 15-th International Conference on Rewriting Techniques and Applications*, 2004b. 27

Jérôme Champavère, Rémi Gilleron, Aurélien Lemay et Joachim Niehren : Efficient inclusion checking for deterministic tree automata and DTDs. *In 2nd International Conference on Language and Automata Theory and Applications*, volume 5196 de *Lecture Notes in Computer Science*, pages 184–195. Springer Verlag, 2008. 121

Christian Choffru : *Contribution a l'etude de quelques familles remarquables de fonctions rationnelles*. Thèse de doctorat, Universite de Paris VII, 1978. 37

C. Choffrut : A generalization of ginsburg and rose's characterisation of g-s-m mappings. *In ICALP 79*, numéro 71 *in* Lecture Notes in Computer Science, pages 88–103. Springer Verlag, 1979. 6, 56

Christian Choffrut : Minimizing subsequential transducers : a survey. *Theoretical Computer Science*, 292(1):131–143, 2003. 56, 63

James Clark : XSL transformations (XSLT) version 1.0. W3C Recommendation, novembre 1999. URL `http://www.w3.org/TR/1999/REC-xslt-19991116`. 2

James Clark et Steve DeRose : XML Path Language (XPath) : W3C Recommendation, 1999. URL `http://www.w3.org/TR/xpath`. 2

Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison et Marc Tommasi : Tree automata techniques and applications. Available online since 1997 : `http://tata.gforge.inria.fr`, octobre 2007b. 22, 24, 25

H. B. Curry et R. Feys : *Combinatory Logic*, volume I. North-Holland, 1958. 26

C. C. Elgot et G. Mezei : On relations defined by generalized finite automata. *IBM J. of Res. and Dev.*, 9:88–101, 1965. 37, 76, 77

J. Engelfriet : Top-down tree transducers with regular look-ahead. *Math. Systems Theory*, 10:289–303, 1977. 91, 92

J. Engelfriet : Some open questions and recent results on tree transducers and tree languages. *In Formal language theory ; perspectives and open problems*. Academic Press, 1980. 6, 93

J. Engelfriet et S. Maneth : Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput.*, 154:34–91, 1999. 95

Joost Engelfriet, Sebastian Maneth et Helmut Seidl : Deciding equivalence of top-down XML transformations in polynomial time. *Journal of Computer and System Science*, 75(5):271–286, 2009. 6, 83, 84, 85, 86, 87, 93

M. J. Fisher : Grammars with macro-like productions. Numéro 9, pages 131–142, 1968. 93

Sylvia Friese : *On Normalization and Type Checking for Tree Transducers*. Thèse de doctorat, Technische Universität München, München, mars 2011. 90

Sylvia Friese, Helmut Seidl et Sebastian Maneth : Minimization of deterministic Bottom-Up tree transducers. *In* Yuan Gao, Hanlin Lu, Shinnosuke Seki et Sheng Yu, éditeurs : *Developments in Language Theory, 14th International Conference DLT 2010*, volume 6224 de *Lecture Notes in Computer Science*, pages 185–196. Springer Verlag, 2010. 90

E. M. Gold : Language identification in the limit. *Inform. Control*, 10:447–474, 1967. 4, 67

T. V. Griffiths : The unsolvability of the equivalence problem for Lambda-Free nondeterministic generalized machines. *Journal of the ACM*, 15(3): 409–413, 1968. 50

John Hopcroft : An n log n algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations*, pages 189–196, 1971. 18

Juhani Karhumäki et Wojciech Plandowski : On the size of independent systems of equations in semigroups. *In* Igor Pr'ivara, Branislav Rovan et Peter Ruzicka, éditeurs : *MFCS*, volume 841 de *Lecture Notes in Computer Science*, pages 443–452. Springer, 1994. ISBN 3-540-58338-6. 143

Stephan Kepser : A simple proof for the Turing-Completeness of XSLT and XQuery. *In Extreme Markup Languages®*, 2004. 2, 112

Stephen C. Kleene : *Automata Studies*, chapitre Representation of Events in Nerve Nets and Finite Automata, pages 3–41. Princeton University Press, 1956. 17

Pavel Labath et Joachim Niehren : A functional language for hyperstreaming XSLT. Rapport technique, INRIA Lille, 2013. 112

Grégoire Laurence, Aurélien Lemay, Joachim Niehren, Slawek Staworko et Marc Tommasi : Normalization of sequential Top-Down Tree-to-Word transducers. *In* Adrian H. Dediu, Shunsuke Inenaga et Carlos M. Vide, éditeurs : *LATA*, volume 6638 de *Lecture Notes in Computer Science*, pages 354–365. Springer, 2011. ISBN 978-3-642-21253-6. 8

Grégoire Laurence, Aurélien Lemay, Joachim Niehren, Slawek Staworko et Marc Tommasi : Learning sequential Tree-to-Word transducers. *In 8th International Conference on Language and Automata Theory and Applications*, Madrid, Espagne, mars 2014. Springer. 9

Aurélien Lemay, Sebastian Maneth et Joachim Niehren : A learning algorithm for Top-Down XML transformations. *In 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 285–296. ACM-Press, 2010. 6, 21, 87, 88

M. Lothaire, éditeur. *Combinatorics on Words*. Cambridge Mathematical Library. Cambridge University Press, 2nd édition, 1997. 145, 146

Wim Martens et Joachim Niehren : On the minimization of XML schemas and tree automata for unranked trees. *Journal of Computer and System Science*, 73(4):550–583, 2007. 25

George H. Mealy : A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 37

Mehryar Mohri : Minimization of sequential transducers. *In Lecture Notes in Computer Science*, pages 151–163, 1994. 57, 58

Edward F. Moore : Gedanken experiments on sequential machines. *In Automata Studies*, pages 129–153. Princeton U., 1956. 37

A. Nerode : Linear automaton transformation. *In Proc. American Mathematical Society*, volume 9, pages 541–544, 1958. 5, 64

J. Oncina et P. Garcia : Inferring regular languages in polynomial update time. *In Pattern Recognition and Image Analysis*, pages 49–61, 1992. 4, 67

J. Oncina, P. Garcia et E. Vidal : Learning subsequential transducers for pattern recognition and interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458, 1993. 6, 67, 76

Ruhsan Onder et Zeki Bayram : XSLT version 2.0 is Turing-Complete : A purely transformation based proof. *In* Oscar H. Ibarra et Hsu C. Yen, éditeurs : *CIAA*, volume 4094 de *Lecture Notes in Computer Science*, pages 275–276. Springer, 2006. ISBN 3-540-37213-X. 2, 112

Christos H. Papadimitriou : *Computational complexity*. Addison-Wesley, 1994. ISBN 978-0-201-53082-7. 138, 163

Wojciech PLANDOWSKI : *The complexity of the morphism equivalence problem for context-free languages*. Thèse de doctorat, Warsaw University. Department of Informatics, Mathematics, and Mechanics, 1995. 7, 53, 55

Michael O. RABIN et Dana SCOTT : Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. 17

Jean-François RASKIN et Frédéric SERVAIS : Visibly pushdown transducers. *In Automata, Languages and Programming, 35th International Colloquium*, volume 5126 de *Lecture Notes in Computer Science*, pages 386–397. Springer Verlag, 2008. 101

C. REUTENAUER et M. P. SCHÜTZENBERGER : Minimalization of rational word functions. *SIAM Journal on Computing*, 20:669–685, 1991. 79, 80, 93

William C. ROUNDS : *Trees, transducers, and transformations*. Thèse de doctorat, Standford University, 1968. 5

Marcel P. SCHÜTZENBERGER : Sur les relations rationnelles. *In* H. BARKHAGE, éditeurs : *Automata Theory and Formal Languages*, volume 33 de *Lecture Notes in Computer Science*, pages 209–213. Springer, 1975. ISBN 3-540-07407-4. 5, 50

Slawek STAWORKO, Grégoire LAURENCE, Aurélien LEMAY et Joachim NIEHREN : Equivalence of nested word to word transducers. *In 17th International Symposium on Fundamentals of Computer Theory*, volume 5699 de *Lecture Notes in Computer Science*, pages 310–322. Springer Verlag, 2009. 7

J. W. THATCHER : Generalized^2 sequential machine maps. *J. Comp. Syst. Sci.*, 4:339–367, 1970. 5

J. W. THATCHER : Tree automata : an informal survey. *In* A. V. AHO, éditeur : *Currents in the theory of computing*, pages 143–178. Prentice Hall, 1973. 22

J. VIRÁGH : Deterministic ascending tree automata I. *Acta Cybernetica*, 5 (1):33–42, 1980. 24