Inférence de requêtes régulières dans les arbres et applications à l'extraction d'information sur le Web

Julien Carme

Inférence de requêtes régulières dans les arbres et applications à l'extraction d'information sur le Web

THÈSE

présentée et soutenue publiquement le 23 Septembre 2005

pour l'obtention du

Doctorat de l'Université Charles de Gaulle - Lille 3

(spécialité informatique)

par

Julien Carme

Composition du jury

Rapporteurs :	Colin de la Higuera, Professeur	Université Jean Monnet - St-Etienne
	Marie-Christine Rousset, Professeur	Université Paris Sud
Examinateurs :	Boris Chidlovskii, Chercheur	Xerox Research Centre Europe
	Joachim Niehren, Directeur de Recherche	INRIA Futurs
	Sophie Tison, Professeur	Université des Sciences et Technologies de Lille
	Marc Tommasi, Maître de Conférence	Université Charles de Gaulle - Lille 3
Directeur :	Rémi Gilleron, Professeur	Université Charles de Gaulle - Lille 3

UNIVERSITÉ LILLE 3
Groupe de Recherche sur l'APPrentissage Automatique

Mis en page et composé avec LaTeX

Remerciements

Tout d'abord, je tiens à remercier Rémi Gilleron, mon directeur de thèse, pour la confiance et le soutien qu'il m'a accordé au commencement de cette thèse et tout au long de mes recherches, pour la qualité de ses conseils et le recul qu'il m'a aidé a apporter à mes travaux.

Merci à Aurélien Lemay, Joachim Niehren, Alain Terlutte et Marc Tommasi pour la qualité des échanges que j'ai eu avec eux au cours de ces années de thèse. La collaboration avec chacun d'entre eux a été une expérience extrêmement enrichissante, et je considère comme une grande chance pour moi d'avoir pu travailler au sein d'une telle équipe.

Je voudrais également remercier Marie-Christine Rousset et Colin de la Higuera pour avoir accepté de relire et de commenter ma thèse.

Je tiens également à remercier l'ensemble du Grappa, pour la qualité de l'ambiance de travail qu'il fait régner à la "petite maison", particulièrement le vendredi en fin d'après-midi.

Enfin, je voudrais remercier Emilie pour avoir supporté sans faillir la dure condition de femme de thésard pendant toutes ces années.

Table des matières

Table des Symboles

f_n	Symbole f d'arité n
V, F	Symboles booléens vrai et faux
q	requête monadique
t	arbre quelconque
β	arbre booléen
$t \times \beta$	arbre annoté par β
T	symbole représentant une branche coupée
Σ	Alphabet à arité fixe
Γ	Alphabet à arité arbitraire
$\Gamma_@$	Alphabet de construction associé à Γ
Bool	Alphabet booléen : $\{\mathsf{vrai}, \mathsf{faux}\}$
@	Symbole de l'extension d'arbres
$@^s$	Opérateur d'extension des arbres à arité arbitraire
$@^a$	Opérateur d'extension des arbres de construction
$\beta_0(t)$	Annotation de t dont toutes les valeurs sont à faux
T_Σ	Arbres à arité fixe sur Σ
$\mathsf{T}^\mathsf{s}_\Gamma$	Arbres à arité arbitraire sur Γ
$\mathsf{T}^\mathsf{a}_\Gamma$	Arbres de construction sur $\Gamma_@$
$\equiv_{tsnRPNI}$	Relation d'équivalence entre TSN pour l'algorithme d'apprentissage tsnRPNI
$\in_{tsnRPNI}$	Relation de consistance entre un exemple et un TSN pour l'algorithme d'apprentissage tsnRPNI
$\mathsf{annot}_\mathsf{A}(\mathsf{t})$	Annotation d'un arbre t par un TSN A
$\mathsf{c}_\mathsf{arbre}$	Fonction de correspondance entre arbres à arité arbitraire et arbres de construction
$\mathsf{c}_\mathsf{dom}(t)$	Fonction de correspondance entre le domaine étendu de t et le domaine de $\mathsf{c}_\mathsf{arbre}(t)$
$\mathsf{c}_\mathsf{requete}$	Fonction de correspondance entre les requêtes dans un arbre à arité arbitraire et les requêtes dans son arbre de construction
$\mathsf{caracterise}_x(L)$	Ensemble caractéristique du langage L pour l'algorithme d'inférence grammaticale x
$\mathsf{compat}(t_1, t_2)$	Relation de compatibilité entre les arbres élagués t_1 et t_2
$\mathsf{complete}(t)$	Opérateur de complétion d'un langage sur $\Sigma \times$ Bool (voir 3.1.2)

$\mathsf{dom}(t)$	Domaine de t (ensemble de ses nœuds)
$\mathsf{dom}^{\mathsf{e}}(t)$	Domaine étendu de t (ensemble de ses nœuds et de ses arêtes)
$\mathsf{elag}(t)$	Ensemble des élagages d'un arbre t
$\mathsf{eval}_A(t)$	Évaluation de l'arbre t par l'automate A
QCA	Question de Correction d'Annotation
$\mathsf{runs_A(t)}$	Ensemble des runs de A sur t
$\mathsf{sruns_A(t)}$	Ensemble des runs réussis de A sur t
$\mathsf{trans_A(t)}$	Ensemble des images de t par la transduction associée à A
QE	Question d'Equivalence

Introduction

1 Motivation et objectifs

Parmi l'ensemble des sites Web existants, nombreux sont ceux qui peuvent être vus comme des bases de données librement accessibles, donnant la possibilité aux utilisateurs d'accéder à un ensemble d'informations structurées. Les exemples sont innombrables : annuaires internet, catalogues de magasins en ligne, moteurs de recherche, sites donnant accès à des informations en temps réel (météo, bourse...). Cependant, il y a une différence majeure entre un site contenant un ensemble d'informations et une base de données contenant les mêmes informations. Alors qu'une base de données est faite pour être interrogée par programme, un site Web est pensé pour être consulté par un être humain. Les données qu'il contient sont organisées au moyen du langage HTML, dont l'objectif est de décrire le rendu des documents et non d'en structurer les informations en vue de leur exploitation automatique.

Par exemple, pour chercher sur le Web, les prix, type, marque et fournisseur d'ordinateurs portables, on effectue une recherche par mot-clé dans un moteur de recherche, puis on parcourt les différents sites proposés. Si on disposait d'une base de données, on effectuerait une requête SQL de la forme "affiche les quadruplets prix, type, marque, fournisseur à partir des expressions de tables avec des critères de sélection". Pour être capable d'interroger le Web avec ce type de requête, il faut récupérer les informations sur les différents sites et les collecter dans une base de données. C'est le rôle des programmes d'extraction que d'effectuer cette tâche. Nous nous situons dans le cadre de la conception de programmes d'extraction d'informations à partir de pages Web. Ces programmes sont communément appelés *wrappers* ou *butineurs*. Ils peuvent être écrits manuellement à l'aide de langages de scripts adaptés aux documents du Web tels que Perl, Python, ... Cependant leur écriture est une tâche trop complexe pour un utilisateur ne possédant pas d'expertise particulière. De plus, les tâches de maintenance peuvent s'avérer complexe au vu de l'évolution continue des sites Web.

Les solutions apportées à ces difficultés se divisent en deux catégories : la spécification assistée et l'inférence de wrappers. Les systèmes fonctionnant par spécification assistée offrent à l'utilisateur un ensemble d'outils, généralement au sein d'une interface graphique, qui lui permettent de spécifier un wrapper sans maîtriser le formalisme utilisé pour le définir. Ces systèmes sont les plus fiables et les plus utilisés actuellement. Ils restent cependant relativement difficiles à utiliser pour un utilisateur inexpérimenté car ils nécessitent une bonne compréhension des formats utilisés pour le Web et une connaissance du processus de calcul associé au formalisme. Les systèmes fonctionnant par inférence, au contraire, ne demandent aucune expertise de la part de l'utilisateur : son seul rôle est d'annoter quelques documents en

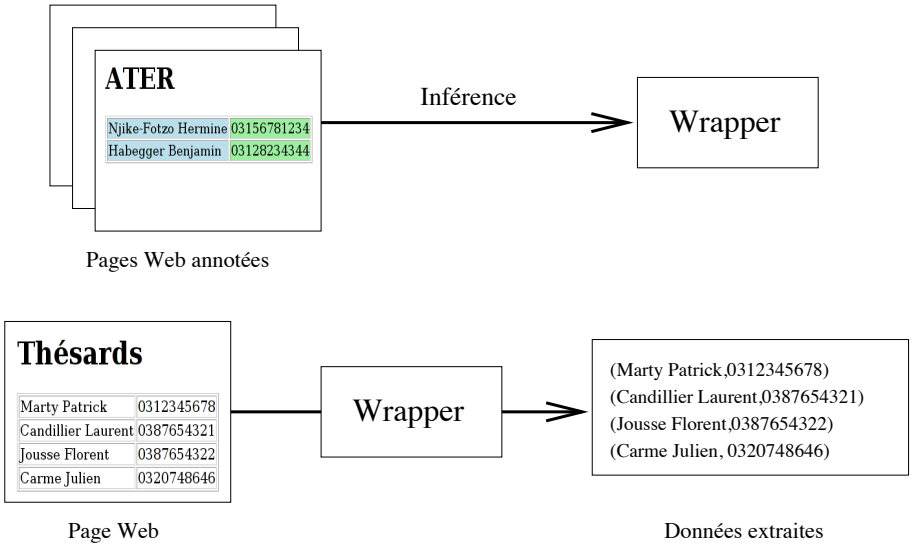


Figure 1: Schéma général d'un système d'extraction d'information par inférence de wrappers. Un wrapper est inféré à partir d'un ensemble de pages dans lesquelles l'utilisateur a spécifié les données à extraire. Il peut ensuite être utilisé pour extraire les données d'une page similaire.

spécifiant manuellement les éléments à extraire, le wrapper correspondant étant ensuite inféré automatiquement (voir Fig. 1). Par contre, leur expressivité est encore limitée.

Les formalismes de définition de wrappers utilisés par les systèmes existants se divisent également en deux catégories : les wrappers textuels et les wrappers structurels. Les wrappers textuels considèrent les pages Web comme des chaînes de caractères ; leur rôle est alors de localiser des sous-chaînes dans ces chaînes. Les wrappers structurels considèrent les pages Web comme des arbres dont la structure est définie à partir de l'imbrication des balises HTML (voir Fig. 2). Le rôle des wrappers structurels est alors de localiser des noeuds dans des structures arborescentes. Intuitivement, il semble raisonnable de penser que l'approche structurelle est pertinente, car même si le rôle de la structure HTML est de décrire le rendu d'une page, il existe tout de même un lien fort entre cette structure et l'organisation des données qu'elle contient.

En observant les systèmes existants, il apparaît clairement que les systèmes de spécification assistée utilisent généralement des requêtes dans les arbres (XWrap [Liu et al., 2000], W4F [Sahuguet and Azavant, 2001], Lixto [Baumgartner et al., 2001]), alors que les systèmes d'inférence utilisent généralement des wrappers textuels (WIEN [Kushmerick, 1997], BWI [Freitag and Kushmerick, 2000], Stalker [Muslea et al., 1998]). Seuls quelques travaux [Kosala et al., 2003, Cohen et al., 2003], sur lesquels nous reviendrons, se sont intéressés à l'inférence de programmes d'extraction d'information sur le Web en utilisant la structure arborescente des documents du Web.

```
<HTML xmlns="http://www.w3.org/1999/xhtml">
  <BODY>
    <H1> Thésards </H1>
    <TABLE border="true">
      <TR>
        <TD> Marty Patrick </TD>
        <TD> <I> 0312345678 <I> </TD>
      </TR>
      <TR>
        <TD> Candillier Laurent</TD>
        <TD> <I> 0387654321 <I> </TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

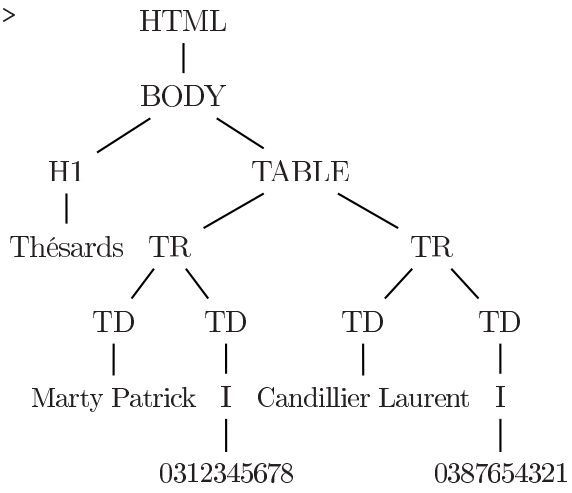


Figure 2: Vue textuelle et vue structurelle d'une page HTML

2 Contribution

Cette thèse se place donc dans le cadre de l'inférence de programmes d'extraction d'information à partir du Web. Elle soutient les deux idées suivantes :

- l'utilisation de la structure arborescente des documents du Web permet de définir des programmes d'extraction expressifs et efficaces ;
- les techniques d'inférence grammaticale sur les arbres sont bien adaptées pour l'inférence de programmes d'extraction d'information.

Cette thèse sera défendue avec des arguments théoriques par le développement d'un formalisme de requêtes et la proposition d'algorithmes d'apprentissage adaptés. Elle sera également défendue par le développement d'un programme Squirrel, prototype d'outil d'extraction d'information sur le Web que nous avons développé et dont les performances sur les benchmarks du domaine et sur des problèmes réels sont tout à fait encourageantes.

Avant de préciser notre approche, nous fixons les limites de notre travail en précisant la classe des problèmes d'extraction considérés dans ce mémoire. L'exemple introductif présenté en Fig. 1 consistait en l'extraction des couples identité et numéro de téléphone. Dans ce mémoire nous ne considérerons que des problèmes d'extraction monadiques dans lesquels on extrait une seule composante, par exemple l'identité ou le numéro de téléphone. Cette restriction est faite par la plupart des systèmes, les wrappers monadiques étant utilisés comme briques de base pour construire des wrappers à sortie complexe (tuples ou documents au format XML) [Gottlob and Koch, 2002a, Niehren et al., 2005]. Également, nous ne considérons que le cas où l'information à extraire correspond à un noeud de l'arbre. Autrement dit, nous ne considérons pas le cas où l'information à extraire est incluse dans le contenu d'une feuille, ni le cas où l'information à extraire est répartie sur plusieurs feuilles. Ces cas pourraient être résolus par des techniques de combinaison de wrappers.

Nous précisons maintenant l'approche retenue pour ce mémoire. Nous considérons les wrappers comme des *requêtes de sélection de noeuds* dans les arbres, que nous appellerons plus

simplement *requêtes* dans les arbres. Cette notion de requête a été largement étudiée dans le domaine de l'interrogation de documents XML (voir [Neven, 1999] pour plus de détail sur le sujet). Nous nous consacrons donc à la conception et à l'analyse d'une méthode d'inférence de requêtes dans les arbres adaptée à l'extraction d'information dans les pages Web. Cette méthode se fonde sur l'idée de représenter les requêtes par des langages d'arbres et de les apprendre en utilisant des algorithmes d'inférence grammaticale d'automates d'arbres.

Pour donner une première description globale de notre système, il nous faut introduire la notion de *langage de requête*, fondamentale dans nos travaux. On appellera *arbre annoté* par une requête un arbre dans lequel chaque noeud est associé à un booléen qui indique si le noeud est sélectionné ou non par la requête. Par exemple l'arbre annoté de la Fig. 2 pour la requête de sélection des identités sera cet arbre dont les fils gauches de TR sont annotés par vrai et tous les autres noeuds par faux. On appellera *langage d'une requête* le langage formé par l'ensemble de tous les arbres annotés par cette requête. Par exemple, le langage des arbres HTML dont les fils gauches de TR sous un TABLE à droite d'un H1 sont annotés par vrai, tous les autres noeuds étant annotés par faux.

Notre système d'inférence de requêtes fonctionne ainsi : l'utilisateur annote des pages HTML en spécifiant dans chacune d'entre elles les noeuds sélectionnés par la requête qu'il veut inférer. Il constitue ainsi un échantillon fini d'arbres annotés. Le système utilise alors un algorithme d'inférence grammaticale pour généraliser cet échantillon en un langage régulier d'arbres qui, idéalement, est le langage de la requête voulue par l'utilisateur. Le système renvoie un automate d'arbres représentant ce langage et donc la requête.

Ce système, simple dans son principe, pose un certain nombre de problèmes qui seront résolus dans ce mémoire :

Le problème de l'arité. Les automates d'arbres dans leur définition classique reconnaissent des langages d'arbres à arité fixe, c'est-à-dire dans lesquels chaque symbole est associé à une ou à un nombre fini d'arités. Les arbres représentant des pages Web contiennent des symboles dont l'arité des symboles n'est pas bornée. Par exemple, on ne peut pas borner le nombre de TR sous un symbole TABLE. Pour résoudre ce problème, nous introduirons une nouvelle classe d'automates d'arbres adaptés aux langages d'arbres à arité arbitraire, les automates à pas. Cette classe d'automates définit les requêtes régulières monadiques qui ont le même pouvoir d'expression que la logique monadique du second ordre et de la logique Datalog monadique.

L'évaluation de requêtes. Un automate d'arbres représentant une requête reconnaît les arbres correctement annotés par rapport à la requête. Ayant un document Web à partir duquel il faut extraire l'information, soit encore un arbre HTML auquel il faut appliquer la requête, il n'est pas imaginable de tester toutes les annotations possibles pour rechercher une annotation correcte. Par conséquent, nous introduirons un algorithme efficace de calcul de requête à partir d'un automate d'arbres reconnaissant le langage de cette requête.

L'apprentissage à partir d'un échantillon de documents annotés. Pour obtenir une expressivité satisfaisante, sur laquelle nous reviendrons dans nos travaux, nous utilisons un algorithme d'inférence grammaticale capable d'identifier l'ensemble des langages réguliers d'arbres. Cet algorithme nécessite un échantillon d'exemples positifs et d'exemples négatifs, c'est-à-dire d'arbres appartenant au langage cible pour les positifs et n'y appartenant pas pour les négatifs.

En extraction d'information, on ne dispose naturellement que d'exemples positifs qui sont les pages annotées par l'utilisateur. Pour pallier l'absence d'exemples négatifs, nous utiliserons une information implicite : une page ayant été annotée, toute annotation différente de cette page est incorrecte, et fournit donc un exemple négatif. L'utilisation efficace de cette source d'exemples négatifs nécessite une adaptation de l'algorithme utilisé, que nous détaillerons.

L'apprentissage à partir de documents partiellement annotés. La méthode que nous venons de décrire implique de la part de l'utilisateur une annotation *complète* de chaque document fourni en exemple. Ce n'est pas satisfaisant dans la pratique, car dans le cas de pages présentant un grand nombre d'éléments à extraire, l'annotation complète d'un document peut devenir une tâche pénible pour l'utilisateur. Il faut donc admettre des annotations partielles dans lesquelles seules un petit nombre des informations à extraire sont annotées. Nous proposerons une solution à l'inférence dans le cadre d'annotations partielles fondée sur l'élagage des arbres. Cette partie de notre travail est d'autant plus importante qu'elle va nous permettre de mettre en place un protocole d'apprentissage interactif par corrections successives qui sera à la base du fonctionnement de Squirrel.

Les problèmes de performance. Dans les systèmes d'extraction, les performances sont : la rapidité d'extraction, la rapidité de construction du wrapper. Nous avons déjà signalé que le calcul d'une requête sera démontré efficace (en temps linéaire). Pour la conception d'un wrapper, le système doit être capable d'inférer correctement une requête à partir d'un nombre d'exemples très réduit. Pour améliorer les performances de notre algorithme, nous allons mettre en place des heuristiques spécifiques au problème de l'extraction d'information dans les pages Web permettant de réduire le nombre d'exemples nécessaires à l'inférence d'une requête. Nos algorithmes d'inférence seront prouvés polynomiaux et nous montrerons qu'ils donnent des temps de calcul réalistes sur des problèmes réels. De plus, pour diminuer le nombre d'annotations, nous introduirons un cadre actif d'inférence dans lequel le concepteur du programme d'extraction peut valider ou corriger les propositions d'extraction faites par le système.

3 Travaux existants

Comme nous l'avons signalé dans la première partie de cette introduction, l'essentiel des travaux qui ont été entrepris dans le domaine de l'extraction d'information l'ont été soit dans le domaine de l'inférence de wrappers textuels, soit dans celui de la spécification de requêtes dans les arbres. Nous n'effectuerons pas dans cette thèse de rappel exhaustif des différents systèmes existants (pour cela, on pourra se référer à [Kosala, 2003]). En revanche, nous allons nous intéresser aux quelques travaux plus récents qui abordent le problème de l'utilisation de la structure dans l'apprentissage de wrappers.

L'une de ces approches consiste à modifier un algorithme d'inférence de wrappers textuels en y intégrant une composante structurelle pour en améliorer ses performances ([Cohen et al., 2003]). Il est intéressant de remarquer que les auteurs de ces travaux ont constaté une amélioration considérable des performances de leur système par rapport aux versions exclusivement textuelles, ce qui confirme l'idée de la pertinence de l'approche structurelle de l'inférence de wrappers.

Les travaux les plus proches de ceux présentés dans cette thèse sont ceux de Kosala *et al.* [Kosala et al., 2003], qui sont également basés sur l'inférence de langages d'arbres. Une première différence entre leurs travaux et les nôtres réside dans le type de langage appris. Leur système est basé sur des algorithmes d'apprentissage de langages *k-testables* qui sont des sous-classes de la classe des langages réguliers. Le notre est basé sur l'apprentissage de langages réguliers, ce qui permet potentiellement d'inférer toutes les requêtes régulières monadiques. Une seconde différence concerne la rapidité d'extraction. Nous avons défini un algorithme de calcul linéaire pour les requêtes inférées dans notre système. Dans le système présenté dans [Kosala et al., 2003], un calcul doit être effectué pour chaque feuille de l'arbre pour savoir si cette feuille doit être extraite ou non ce qui limite les performances du système lors de l'extraction.

Enfin, il est intéressant de noter que des travaux sont en cours pour intégrer à Lixto [Baumgartner et al., 2001] une composante apprentissage qui permettrait de résoudre en partie le problème de la difficulté de spécification des requêtes.

4 Plan

Nous commencerons par un chapitre préliminaire rappelant les notions d'arbres, de langages et d'automates que nous utiliserons par la suite. Nous introduirons en particulier la distinction entre les arbres à arité fixe et arbres à arité arbitraire.

Dans le *chapitre 2*, nous introduirons les *automates à pas*, automates présentant la particularité de pouvoir être utilisés indifféremment sur les arbres à arité arbitraire et sur les *arbres de construction*, un codage des arbres à arité arbitraire en arbres à arité fixe que nous introduirons. Cette notion nous permettra ensuite d'utiliser des algorithmes adaptés aux arbres à arité fixe dans les arbres à arité arbitraire.

Dans le *chapitre 3*, nous introduirons les *transducteurs de sélection de noeuds (*TSN*)*, automates d'arbres définissant des requêtes monadiques. Nous étudierons les propriétés particulières de ces automates, et nous montrerons comment les utiliser pour calculer des requêtes. Nous montrerons ensuite l'intérêt de l'élagage des arbres pour la définition de requêtes, et nous présenterons les *transducteurs de sélection de noeuds élagueurs (*TSNe*)*, TSN adaptés à l'élagage.

Le *chapitre 4* sera consacré à l'apprentissage des TSN et des TSNe. Nous présenterons un algorithme l'apprentissage des TSN et nous montrerons qu'il est capable d'identifier les requêtes *régulières* (c'est-à-dire les requêtes dont le langage d'arbre associé est régulier) dans le modèle d'apprentissage par données fixées. Nous présenterons ensuite un algorithme d'apprentissage des TSNe, et nous montrerons comment l'intégrer dans un algorithme d'apprentissage actif des TSNe qui sera à la base de notre outil d'apprentissage interactif de requêtes, Squirrel.

Dans le *chapitre 5*, nous discuterons des difficultés posées par le passage de la théorie à la pratique, nous présenterons des heuristiques permettant d'améliorer les performances de nos algorithmes, et nous décrirons le prototype d'outil interactif d'extraction d'information que nous avons développé, Squirrel.

Nous terminerons dans le *chapitre 6* par l'exposé des résultats expérimentaux que nous avons obtenus. Nous commencerons par discuter de la constitution d'un corpus et des pro-

blèmes liés à l'évaluation des systèmes d'extraction d'information, et ensuite nous présenterons les deux modes d'évaluation que nous avons retenus. D'abord nous évaluerons notre système dans le cadre classique de l'évaluation de systèmes d'extraction d'information à partir d'un ensemble de documents annotés pour des soucis de comparaison avec les systèmes existants. Ensuite, nous tenterons d'évaluer l'efficacité réelle de notre système en simulant le comportement d'un utilisateur de Squirrel et en évaluant la quantité d'interaction nécessaire à la définition d'une requête.

Enfin, vous trouverez en annexe deux travaux réalisés en début de thèse et publiés. Ils sont basés sur l'idée d'étendre les résultats obtenus dans l'équipe [Denis et al., 2002b, Denis et al., 2004] sur les automates à états résiduels (une classe d'automates non déterministes possédant de bonnes propriétés pour l'apprentissage) du cas des mots au cas des arbres. Un premier travail concerne la définition des automates d'arbres résiduels ascendants et descendants et l'étude de leurs propriétés [Carme et al., 2003a]. Un second travail concerne l'apprentissage d'une classe d'automates d'arbres à états résiduels [Carme et al., 2003b]. En raison de la création du projet MOSTRARE, cette piste de travail a été abandonnée au profit des travaux sur l'extraction d'information qui constituent le sujet essentiel de ce mémoire.

5 Bibliographie personnelle

Les travaux présentés dans cette thèse on donné lieu à des publications dans les actes de différentes conférences :

1. **2005** Julien Carme, Rémi Gilleron, Aurélien Lemay et Joachim Niehren, *Interactive Learning of Node Selection Queries in Tree Structured Documents,* IJCAI Workshop on Grammatical Inference
2. **2004** [1] Julien Carme, Joachim Niehren et Marc Tommasi, *Querying Unranked Trees with Stepwise Tree Automata,* 19th International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science 3091, 105 – 118
3. **2004** Julien Carme, Aurélien Lemay et Joachim Niehren, *Learning Node Selecting Tree Transducer from Completely Annotated Examples*, 7th International Colloquium on Grammatical Inference, Lecture Notes in Artificial Intelligence 3264, 91–102
4. **2003** J. Carme, R. Gilleron, A. Lemay, A. Terlutte et M. Tommasi, *Residual Finite Tree Automata*, 7th International Conference on Developments in Language Theory, Lecture Notes in Computer Science 2710, 171 – 182
5. **2003** J. Carme, A. Lemay et A. Terlutte, *Identification à la limite de langages réguliers d'arbres à résiduels premiers disjoints*, Proceedings of CAP'03, 217 – 232

[1] Version étendue soumise à Machine Learning Journal

Chapitre 1

Langages et automates d'arbres

Ce chapitre préliminaire est consacré à la présentation des notions d'arbres, de langages d'arbres et d'automates d'arbres, qui seront à la base des travaux de cette thèse. Il s'agit de notions classiques dont nous ne ferons qu'un rapide survol. Pour plus d'informations sur ce sujet, consultez [Comon et al., 1997, Bruggemann-Klein et al., 2001].

1.1 Arbres

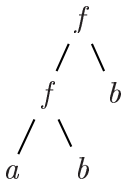
Nous allons commencer par présenter la notion d'arbre va nous servir de modèle pour les pages HTML. Nous allons d'abord définir la notion de structure d'arbre, avant de donner les deux définitions récursives d'arbres que nous utiliserons par la suite : les arbres à arité fixe et les arbres à arité arbitraire.

1.1.1 Structure d'arbre

Un *arbre* t est défini par un domaine et un *étiquetage* :

- Le *domaine* de t, noté $\mathsf{dom}(t)$, est un ensemble de nœuds formant la structure de l'arbre t. On représente chaque nœud par un mot sur $\mathbb{N}$ décrivant le chemin qui y mène, en partant de la racine et en descendant vers le i-ème fils à la lecture d'un symbole i. Par exemple, la racine d'un arbre est représentée par le mot vide ε, et le deuxième fils de son premier fils par le mot 1.2.
- L'étiquetage d'un arbre est une fonction de son domaine vers un ensemble d'étiquettes E. On notera $t(n)$ l'étiquette du nœud π dans t.

Exemple 1.1.



Le domaine de cet arbre est $\{\varepsilon, 1, 1.1, 1.2, 2\}$*, et son étiquetage est :* $t(\varepsilon) = f$*,* $t(1) = f$*,* $t(1.1) = a$*,* $t(1.2) = b$ *et* $t(2) = b$*.*

Un *sous-arbre* t' d'un arbre t en position π, avec $\pi \in \mathsf{dom}(t)$, est un arbre noté $t_{|\pi}$, défini par :
- $\mathsf{dom}(t') = \{\pi' | \pi.\pi' \in \mathsf{dom}(t)\}$
- $t'(\pi') = t(\pi.\pi')$

On dit que t' est un sous-arbre de t s'il existe un nœud π dans $\mathsf{dom}(t)$ telle que π' soit un sous-arbre de t en position π.

1.1.2 Arbres définis sur un alphabet à arité fixe

Un *alphabet à arité fixe* est un couple (Σ, Arit) où Σ est un ensemble fini de symboles et Arit une fonction de Σ dans $\mathbb{N}$. On appelle *arité* d'un symbole la valeur de cette fonction pour ce symbole. Les symboles d'arité 0 sont les *constantes*, les symboles d'arité n sont les symboles n-aires. Lorsque le contexte exigera de préciser l'arité d'un symbole, nous noterons f_n un symbole f d'arité n. Nous noterons Σ un alphabet à arité fixe (Σ, Arit).

L'ensemble des *termes clos* sur un alphabet à arité fixe Σ, noté T_Σ, est le plus petit ensemble tel que :
- $\{a \in \Sigma | \mathsf{Arit}(a) = 0\} \subset \mathsf{T}_\Sigma$
- $\{f(t_1, \ldots, t_n) | (f \in \Sigma) \wedge (\mathsf{Arit}(f) = n) \wedge (t_1 \ldots t_n \in \mathsf{T}_\Sigma)\} \subset \mathsf{T}_\Sigma$

Un arbre est défini sur un alphabet à arité fixe Σ s'il coïncide avec un terme clos sur Σ, c'est-à-dire si l'étiquette de chaque nœud est un symbole de Σ dont l'arité correspond au nombre de fils de ce nœud. Formellement, on doit avoir pour tout élément n de $\mathsf{dom}(t)$:

$$\{n.i | n.i \in \mathsf{dom}(t)\} = \{n.1, \ldots, n.\mathsf{Arit}(n)\}$$

Par la suite, nous appellerons *arbres à arité fixe* les arbres définis sur un alphabet à arité fixe. Un *langage d'arbres à arité fixe* est un ensemble d'arbres définis sur un alphabet à arité fixe.

Dans nos travaux, lorsque nous parlerons d'alphabets, d'arbres ou de langages d'arbres sans précision supplémentaire, il s'agira d'alphabets, d'arbres, et de langages d'arbres à arité fixe.

1.1.3 Arbres définis sur un alphabet à arité arbitraire

Un *alphabet à arité arbitraire* Γ est un ensemble de symboles. L'ensemble des *termes clos* sur un alphabet à arité arbitraire Γ, noté T_Γ, est le plus petit ensemble tel que :
- $\{a \in \Gamma\} \subset \mathsf{T}_\Gamma$
- $\{f(t_1, \ldots, t_n) | f \in \mathsf{T}_\Gamma \wedge n \geq 0 \wedge t_1 \ldots t_n \in \mathsf{T}_\Gamma\} \subset \mathsf{T}_\Gamma$

Un arbre est défini sur un alphabet à arité arbitraire Γ s'il coïncide avec un terme clos sur Γ, c'est-à-dire si toutes les étiquettes de ses nœuds sont des symboles de Γ.

De façon analogue au cas de l'arité fixe, nous utiliserons les termes d'*arbre* et de *langage d'arbre à arité arbitraire*

Exemple 1.2. *Soit* $\mathsf{T}_\Gamma = \{f, a\}$. *Les arbres* $a(a, a)$, $f(a)$ *et* a *sont des exemples d'arbres définis sur* T_Γ.

L'ensemble $\{f(\underbrace{a, \ldots, a}_{i fois})_{i>0}\}$ *est un langage non fini sur* T_Γ.

1.2 Automates d'arbres

Nous allons présenter ici les notions d'automates d'arbres pour les arbres à arité fixe, puis d'automates à haie, automates d'arbres pour les arbres à arité arbitraire.

1.2.1 Automates d'arbres pour les arbres à arité fixe

Il existe des automates d'arbres ascendants et descendants. Dans nos travaux, nous nous limiterons aux automates d'arbres ascendants, c'est-à-dire effectuant les évaluations des arbres des feuilles vers la racine.

Un automate d'arbres est un objet défini sur un alphabet Σ capable de reconnaître des arbres sur Σ. Ainsi, un automate d'arbres définit le langage des arbres qu'il reconnaît.

Un automate d'arbre A est défini par :
- Un ensemble d'états, noté $\mathsf{etats}(A)$
- Un ensemble d'états finaux, noté $\mathsf{finaux}(A)$, tel que $\mathsf{finaux}(A) \subseteq \mathsf{etats}(A)$
- Un ensemble de règles de transition, du type :
  - $a \to q$ avec a une constante de Σ et q un état de A
  - $f(q_1, \ldots, q_n) \to q$ avec f un symbole n-aire de Σ, et $q_1 \ldots q_n, q$ des états de A

Évaluation d'arbres

Un automate d'arbres A sur Σ est associé à une *fonction d'évaluation*, définie de T_Σ dans l'ensemble des parties de $\mathsf{etats}(A)$:

$$\begin{array}{lllll} \mathsf{eval}_A() : & \mathsf{T}_\Sigma & \to & 2^{\mathsf{etats}(A)} \\ & \mathsf{eval}_A(a) & = & \{q \mid a \to q \in \mathsf{regles}(A)\} \\ & \mathsf{eval}_A(f(t_1, \ldots, t_n) & = & \{q \mid q_1 \in \mathsf{eval}_A(t_1), \ldots,\ q_n \in \mathsf{eval}_A(t_n), \\ & & & f(q_1, \ldots, q_n) \to q \in \mathsf{regles}(A)\} \end{array}$$

On dit que A *évalue* un arbre t en q si q est inclue dans $\mathsf{eval}_A(t)$.

Un arbre t est *reconnu* par A si et seulement si l'évaluation de t par A contient un état final. Un automate sur Σ définit donc un langage d'arbres sur Σ, qui est l'ensemble des arbres qu'il reconnaît :

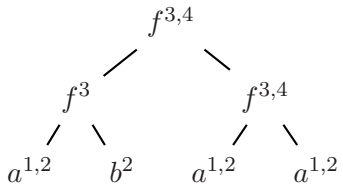
$$\mathsf{langage}(A) = \{t \in \mathsf{T}_\Sigma | \exists q \in \mathsf{finaux}(A),\ q \in \mathsf{eval}_A(t)\}$$

Un langage sur Σ est régulier si et seulement s'il existe un automate d'arbres sur Σ le reconnaissant.

Exemple 1.3. *Considérons l'automate d'arbres A défini sur l'alphabet $\{a_0, b_0, f_2\}$ ainsi :*

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{1,2,3,4\} \\ \mathsf{finaux}(A) & = & \{4\} \\ \mathsf{regles}(A) & = & \{a \to 1, a \to 2, b \to 2, \\ & & f(1,2) \to 3, f(1,1) \to 4, f(3,4) \to 4, f(3,3) \to 3\} \end{array}$$

L'évaluation se fait par évaluation récursive de chacun des sous-arbres. Prenons l'exemple de l'évaluation de l'arbre t ci-dessous par l'automate A. Chaque évaluation de chaque sous-arbre est indiquée ici en indice de la racine du sous-arbre correspondant :



La racine est évaluée en 4, qui est un état final, donc cet arbre est reconnu par l'automate A.

Runs sur les arbres

Un *run* d'un automate d'arbres A sur un arbre t est un étiquetage des nœuds de t par des états de A conformément aux règles de A. Cela correspond à une évaluation au cours de laquelle on ne choisit qu'un état par sous-arbre récursivement calculé et on l'associe à la racine de ce sous-arbre. Un run est *réussi* s'il étiquette la racine de l'arbre avec un état final. On a alors une autre caractérisation des arbres reconnus par un automate d'arbres A : un arbre t est reconnu par un automate A si et seulement si il existe un run réussi de A sur t.

Formellement, un run r de A sur t est une fonction de $\mathsf{dom}(t)$ dans $\mathsf{etats}(A)$ telle que pour tout nœud π de fils $\pi_1 \dots \pi_n$ et d'étiquette f, on ait $f(r(\pi_1), \dots, r(\pi_n)) \to r(\pi) \in \mathsf{regles}(A)$.

Le calcul d'un run s'effectue de manière récursive, à partir des feuilles jusqu'à la racine. Pour calculer un run sur un arbre $f(t_1, \dots, t_n)$, on calcule un run sur chacun des $t_1, \dots, t_n$ puis on applique un règle de l'automate pour associer un état à la racine.

On note $\mathsf{runs_A(t)}$ l'ensemble des runs de A sur l'arbre t, et $\mathsf{sruns_A(t)}$ l'ensemble des runs réussis de A sur t.

Les notions de runs et d'évaluation sont liées par cette propriété fondamentale :

$$\mathsf{eval}_A(t) = \{q | \exists r \in \mathsf{runs_A(t)}\ (r(\varepsilon) = q\}$$

Exemple 1.4. *Reprenons l'exemple 1.5. Voici les deux runs de l'automate A sur l'arbre t :*

Parmi ces deux runs, seul celui de droite est réussi, car seul 4 *est un état final*

Automates d'arbres déterministes et non-déterministes

Un automate d'arbres est *déterministe* s'il ne contient pas de règles distinctes ayant le même membre gauche. L'évaluation d'un arbre par un automate déterministe contient au plus un élément.

Les langages d'arbres exprimables par les automates d'arbres déterministes sont exactement les langages d'arbres exprimables par les automates d'arbres non-déterministes. Il est possible de transformer un automate d'arbres non-déterministe en automate d'arbre déterministe équivalent, par une opération dite de *déterminisation*. La déterminisation se fait par une *subset construction*, opération consistant à construire un état de l'automate déterministe pour chaque ensemble d'états de l'automate non-déterministe évaluant le même arbre. La taille de l'automate résultant peut être exponentiellement plus grande que la taille de l'automate de départ.

Automates d'arbres et ε-transistions

Une ε-transition est une transition d'un état vers un autre notée du type $q_1 \xrightarrow{\varepsilon} q_2$. Elle s'interprète ainsi : tout arbre évalué en q_1 peut également être évalué en q_2. L'utilisation d'ε-transition permet d'exprimer plus facilement de nombreuses opérations sur les automates d'arbres. Ce sera le cas dans nos travaux pour la conversion automates à haie/automates à pas.

On peut ainsi définir une extension des automates d'arbres, les *automates d'arbres avec ε-transitions*, qui contiennent en plus des règles déjà définies un ensemble d'ε-transitions, et dont la fonction d'évaluation associée est adaptée.

Les automates d'arbres avec ε-transitions ont la même expressivité que les automates d'arbres sans ε-transition : il est possible de supprimer les ε-transitions sans changer le langage reconnu par l'automate par une opération dite d'ε-clôture. Cette opération se fait en temps quadratique, et le nombre d'états de l'automate ne change pas.

Automates d'arbres émondés

Un état *coaccessible* est un état q pour lequel il existe un arbre t, un nœud π de t, et un run réussi r de A sur t tel que $r(\pi) = q$. Un automate d'arbres A émondé est un automate d'arbres pour lequel tout état est *coaccessible*.

Les états non coaccessibles sont inutiles : leur retrait ne change pas le langage reconnu par l'automate.

1.2.2 Automates à haies

Les automates à haies sont les automates les plus classiquement utilisés pour représenter des langages d'arbres à arité arbitraire. Ils sont également appelés automates d'arbres à arité arbitraire.

Un automate à haies est un objet défini sur un alphabet à arité arbitraire Γ capable de reconnaître des arbres à arité arbitraire sur Γ.

Un automate à haies H est défini par :
- Un ensemble d'états, noté $\mathsf{etats}(H)$
- Un ensemble d'états finaux, noté $\mathsf{finaux}(H)$, tel que $\mathsf{finaux}(H) \subseteq \mathsf{etats}(H)$
- Une collection de langages réguliers représentés par des automates de mots ou des expressions régulières $H_{a,q}$ sur $\mathsf{etats}(A)$, pour tout a dans Γ et q dans $\mathsf{etats}(A)$.

Chaque automate de mot $H_{a,q}$, représente l'ensemble des règles $a(q_1 \dots q_n) \to q$ telles que $q_1 \dots q_n$ soit reconnu par $H_{a,q}$

Évaluation d'arbres avec les automates à haie

L'évaluation d'un arbre à arité arbitraire par un automate à haie est un calcul récursif ascendant similaire à celui d'un automate classique. La seule différence est dans l'utilisation des automates $H_{a,q}$ pour représenter des ensembles non finis de règles :

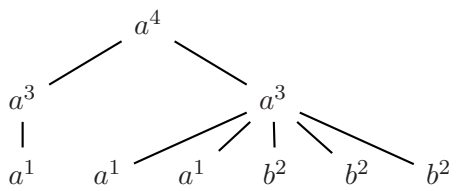
$$\mathsf{eval}_H(f(t_1,\dots,t_n)) = \{q | \exists q_1 \in \mathsf{eval}_H(t_1), \dots, \exists q_n \in \mathsf{eval}_H(t_n),\ q_1 \dots q_n \in L(H_{f,q})\}$$

Exemple 1.5. *Considérons l'automate à haie H défini sur l'alphabet à arité arbitraire $\{a, b\}$ ainsi :*

$$\begin{aligned} \mathsf{etats}(A) &= \{1,2,3,4\} \\ \mathsf{finaux}(A) &= \{4\} \\ H_{a,1} &= \varepsilon \\ H_{b,2} &= \varepsilon \\ H_{a,3} &= 1^+.2^* \\ H_{a,4} &= 3.3 \end{aligned}$$

Les $H_{w,q}$ non spécifiés sont vides. Les deux premières définitions de $H_{w,q}$ nous indiquent que les feuilles d'étiquette a peuvent être évaluées en 1 et les feuilles d'étiquette b peuvent être évaluées en 2. La troisième nous indique qu'un arbre $a(t_1,\dots,t_n)$ est évalué en 3 s'il existe i entre 1 et n tel que les arbres $t_1,\dots,t_i$ sont évalués en 1 et les arbres $t_{i+1}\dots t_n$ sont évalués en 2. Enfin, la quatrième nous indique qu'un arbre $a(t_1,t_2)$ est évalué en 4 si t_1 et t_2 sont évalués en 3.

Voici un exemple d'évaluation d'un arbre à arité arbitraire sur $\{a, b\}$ par l'automate à haie H :



Chapitre 2

Les automates à pas

L'objet de ce chapitre est de mettre en place un formalisme d'automates d'arbres adapté au traitement d'arbres à arité arbitraire, de manière à pouvoir par la suite utiliser ces automates pour définir des requêtes dans les pages HTML.

Les automates d'arbres classiques fonctionnent sur des *arbres à arité fixe*, c'est-à-dire sur des arbres définis sur des alphabets dans lesquels chaque symbole a une *arité* donnée. Ce type d'automate n'est pas capable de reconnaître des langages dans lesquels l'arité des symboles n'est pas bornée, ce qui n'est pas satisfaisant dans le cadre du traitement de pages HTML. Par exemple, la famille d'arbres $F = \{f(\underbrace{a, \ldots, a}_{i fois})_{i>0}\}$, ne pourrait être exprimée par un langage d'arbres sur un alphabet à arité fixe fini. En effet, dans $f(a)$, le symbole à la racine serait f_1, dans $f(a, a)$, ce serait $f_2 \ldots$ et il faudrait une infinité de symboles à arité fixe pour représenter ce langage.

Pour résoudre ce problème, il y a deux solutions :
- Travailler sur les codages binaires des arbres à arité arbitraire. C'est généralement le classique codage frère-fils qui est utilisé, par example dans les *selection automata* [Koch, 2003, Gottlob and Koch, 2002b, Frick et al., 2003].
- Définir un nouveau type d'automates, capables de travailler directement sur les arbres à arité arbitraire. C'est le cas des *automates à haies* [Bruggemann-Klein et al., 2001, Neumann and Seidl, 1998].

Les *automates à haies*, dont nous reparlerons plus tard dans ce chapitre, sont une méthode élégante pour modéliser les langages d'arbres à arité arbitraire, mais qui ne semble pas satisfaisante pour l'apprentissage. L'utilisation de codages binaires présente l'inconvénient de ne pas pouvoir permettre l'utilisation directe, sans encodage préalable, des automates sur les arbres dont on dispose.

Nous allons présenter dans ce chapitre un formalisme, les automates à pas, qui concilie les deux solutions présentées. Les automates à pas peuvent être vus à la fois comme des automates fonctionnant directement sur les arbres à arité arbitraire et comme des automates classiques sur un codage particulier, dit *codage de construction*.

Nous allons introduire l'opérateur d'extension d'arbre, un opérateur qui permet de construire récursivement tout arbre à arité arbitraire. Ensuite, nous allons montrer qu'à par-

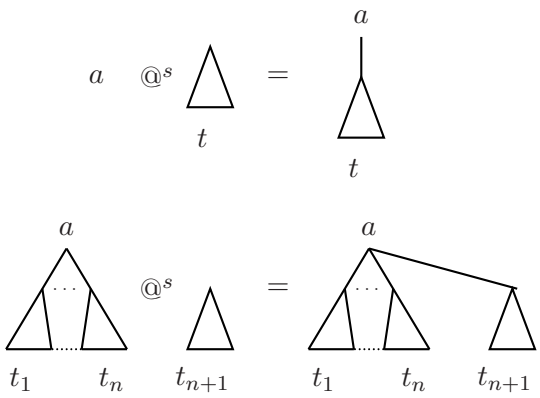


Figure 2.1: Extension d'arbres

tir de cette opérateur, il est possible de définir un codage des arbres à arité arbitraire en arbres binaires que nous appellerons codage de construction. Puis nous introduirons les automates à pas, automates d'arbres travaillant simultanément sur les arbres à arité arbitraire et leurs codages de construction. Enfin, nous montrerons que les automates à pas ont la même expressivité que les automates à haies.

2.1 L'opérateur d'extension d'arbre

L'extension d'arbre est un opérateur binaire de construction d'arbres à arité arbitraire. Il reçoit deux arbres t_1 et t_2 en paramètres, et renvoie le résultat de l'ajout de t_2 en dernière position dans les fils de t_1. Son fonctionnement est illustré Fig. 2.1.

Définition 2.1. *Soit $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ l'ensemble des arbres à arité arbitraire sur l'alphabet à arité arbitraire Γ. Soit $a(t_1, \ldots, t_n)$ avec $n \geq 0$ et t' deux éléments de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$.*

L'extension d'arbre $@^s$ est définie ainsi :

$$\begin{array}{rlcl} @^s : & \mathsf{T}^{\mathsf{s}}_{\Gamma} \times \mathsf{T}^{\mathsf{s}}_{\Gamma} & \rightarrow & \mathsf{T}^{\mathsf{s}}_{\Gamma} \\ & @^s(a(t_1, \ldots, t_n), t') & = & a(t_1, \ldots, t_n, t') \end{array}$$

En particulier, dans le cas où $n = 0$, $@^s(a, t') = a(t')$.

Par la suite, dans un but de lisibilité, nous utiliserons la syntaxe $t@^s t'$ plutôt que $@^s(t, t')$. L'opérateur $@^s$ n'étant pas associatif, lorsque nous utiliserons la syntaxe $t_1@^s t_2@^s t_3$, la priorité sera donné aux opérateurs de gauche, c'est-à-dire que $t_1@^s t_2@^s t_3$ s'interprétera $(t_1@^s t_2)@^s t_3$

2.2 Les arbres de construction

Nous allons utiliser l'opérateur d'extension pour définir une correspondance entre l'ensemble des arbres à arité arbitraire sur Γ et l'ensemble des *arbres de construction*, arbres définis sur un alphabet binaire que nous allons définir.

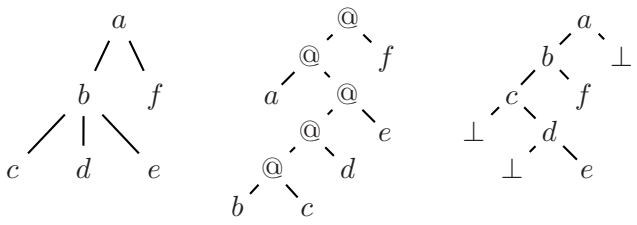


Figure 2.2: A gauche, un arbre à arité arbitraire, au milieu, son arbre de construction, et à droite, à titre de comparaison, son codage binaire selon le classique codage fils-frère.

Le but de cette correspondance est de définir un cadre de travail dans lequel toute manipulation d'arbres à arité arbitraire pourra se faire de manière analogue dans les arbres de construction. Cela nous permettra de travailler sur les arbres à arité arbitraire en utilisant des résultats existant sur les arbres à arité fixe.

Pour cela, il nous faut d'abord définir cette correspondance, et établir qu'il s'agit d'un codage, c'est-à-dire qu'elle est bijective. Mais il faut également établir une correspondance entre le domaine de chaque arbre à arité arbitraire et de son codage de construction. En effet, nous serons amenés par la suite à manipuler les éléments de ces domaines, notamment pour la définition des runs d'automates et pour le calcul de requêtes. Pour pouvoir définir de façon analogue des opérations sur les éléments des domaines des arbres à arité arbitraire et sur ceux des arbres de construction, nous allons avoir besoin d'une correspondance entre ces domaines.

2.2.1 Une correspondance entre les arbres à arité fixe et arbitraire

On peut remarquer qu'en utilisant l'opérateur d'extension, $a(t_1, \ldots, t_n)$ peut s'exprimer $a@^s t_1 @^s \ldots @^s t_n$. On peut en déduire par une récurrence immédiate que n'importe quel arbre sur Γ peut s'exprimer par une expression parenthésée ne comportant que des symboles de Γ et l'opérateur $@^s$. Par exemple, $a(b(c, d, e), f)$ peut s'écrire $a@^s(b@^s c@^s d@^s e)@^s f$.

Cette expression des arbres à arité arbitraire n'utilisant que les symboles et un unique opérateur binaire induit de façon naturelle un codage binaire des arbres à arité arbitraire (voir Fig. 2.2), qui est la description de la construction des arbres à arité arbitraire avec l'opérateur d'extension. C'est pour cela que nous appellerons ces codages *arbres de construction*.

Définition 2.2. *On appelle* alphabet de construction $\Gamma_@$ *associé à l'alphabet à arité arbitraire* Γ *l'alphabet à arité fixe défini ainsi :*

$$\Gamma_@ = \{a_0 | a \in \Gamma\} \cup \{@_2\}$$

Les arbres de construction sont des arbres à arité fixe définis sur l'alphabet de construction. On notera $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ l'ensemble des arbres de construction définis sur $\Gamma_@$. Par analogie avec l'opération d'extension d'arbres à arité arbitraire, on peut définir *l'extension d'arbre de construction* ainsi :

$$@^a(c_1, c_2) = @(c_1, c_2)$$

Il s'agit ici simplement de la classique construction récursive d'un arbre à partir d'un symbole binaire et de deux arbres. Notons tout de même que dans notre cas, nous distinguons

volontairement $@^a$, l'opérateur d'extension d'arbres de construction, et $@$, le symbole binaire lui correspondant dans l'alphabet de construction.

Nous allons maintenant définir la *fonction de construction*, fonction associant à un arbre à arité arbitraire son codage de construction, et montrer qu'il s'agit bien d'un codage, c'est-à-dire qu'elle est bien bijective.

Définition 2.3. *On appelle* $\mathsf{c}_{\mathsf{arbre}}$ *la* fonction de construction, *qui associe à tout arbre sur* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *l'arbre de construction sur* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ *correspondant :*

$$\begin{array}{lllll} \mathsf{c}_{\mathsf{arbre}} : & \mathsf{T}^{\mathsf{s}}_{\Gamma} & \rightarrow & \mathsf{T}^{\mathsf{a}}_{\Gamma} & \\ & \mathsf{c}_{\mathsf{arbre}}(a) & = & a & \text{avec } a \in \Gamma \\ & \mathsf{c}_{\mathsf{arbre}}(t_1 @^s t_2) & = & \mathsf{c}_{\mathsf{arbre}}(t_1) @^a \mathsf{c}_{\mathsf{arbre}}(t_2) & \text{avec } t_1 \in \mathsf{T}^{\mathsf{s}}_{\Gamma} \\ & & & & \text{et } t_2 \in \mathsf{T}^{\mathsf{s}}_{\Gamma} \end{array}$$

On appelle arbre de construction de t *l'arbre* $\mathsf{c}_{\mathsf{arbre}}(t)$.

Propriété 2.1. *La fonction* $\mathsf{c}_{\mathsf{arbre}}$ *est une bijection de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *dans* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$

Preuve : Nous utiliserons dans cette preuve ainsi que dans plusieurs autres preuves de ce chapitre une "récurrence sur la construction des arbres". On prouve une hypothèse H pour les arbres de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *de hauteur 1, puis on montre que si elle est vraie pour deux arbres* t_1 *et* t_2, *alors elle l'est également pour* $t_1 @^s t_2$. *On en déduit qu'elle est vraie pour tout arbre de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$. *Le même raisonnement peut être fait dans* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$, *il s'agit alors d'une simple récurrence sur la hauteur des arbres.*

Montrons que $\mathsf{c}_{\mathsf{arbre}}$ *est une injection de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *dans* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$.

Soit t *et* t' *deux arbres de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$, *tels que* $\mathsf{c}_{\mathsf{arbre}}(t) = \mathsf{c}_{\mathsf{arbre}}(t')$. *Montrons par récurrence sur la construction de* t *que* $t = t'$.

Si $t = a$, *comme* $\mathsf{c}_{\mathsf{arbre}}(t') = a$ *alors* $\mathsf{c}_{\mathsf{arbre}}(t) = a$, *donc* $t' = a$, *donc* $t' = t$.

Si $t = t_1 @^s t_2$, *on a* $\mathsf{c}_{\mathsf{arbre}}(t) = \mathsf{c}_{\mathsf{arbre}}(t_1) @^a \mathsf{c}_{\mathsf{arbre}}(t_2)$ *par définition de* $\mathsf{c}_{\mathsf{arbre}}$, *donc* $\mathsf{c}_{\mathsf{arbre}}(t') = \mathsf{c}_{\mathsf{arbre}}(t_1) @^a \mathsf{c}_{\mathsf{arbre}}(t_2)$ *par hypothèse. Donc* $\mathsf{c}_{\mathsf{arbre}}(t')$ *n'est pas une feuille, donc* $\mathsf{c}_{\mathsf{arbre}}(t') = \mathsf{c}_{\mathsf{arbre}}(t'_1) @^a \mathsf{c}_{\mathsf{arbre}}(t'_2)$, *avec* $t' = t'_1 @^s t'_2$. *Donc* $\mathsf{c}_{\mathsf{arbre}}(t'_1) = \mathsf{c}_{\mathsf{arbre}}(t_1)$ *et* $\mathsf{c}_{\mathsf{arbre}}(t'_2) = \mathsf{c}_{\mathsf{arbre}}(t_2)$. *Par hypothèse de récurrence,* $t_1 = t'_1$ *et* $t_2 = t'_2$, *donc* $t = t'$. *On en déduit que l'hypothèse est vérifiée pour toute paire* t, t' *de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$.

Montrons que $\mathsf{c}_{\mathsf{arbre}}$ *est une surjection de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *dans* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$.

Soit c' *un arbre de* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$. *Montrons par récurrence sur la construction de* c' *qu'il existe* t *tel que* $\mathsf{c}_{\mathsf{arbre}}(t) = c'$.

Si $c' = a$, *comme* $\mathsf{c}_{\mathsf{arbre}}(a) = a$, *l'hypothèse est vérifiée.*

Si $c' = c'_1 @^a c'_2$, *on suppose par récurrence qu'il existe* t_1 *et* t_2 *tels que* $t_1 = \mathsf{c}_{\mathsf{arbre}}(c'_1)$ *et* $t_2 = \mathsf{c}_{\mathsf{arbre}}(c'_2)$. *En posant* $t = t_1 @^s t_2$, *on a bien* $\mathsf{c}_{\mathsf{arbre}}(t) = c'$. *On en déduit que l'hypothèse est vérifiée pour tout* c' *de* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$. ◄

L'étude des relations entre définitions algèbrique et syntaxiques des langages d'arbres est un sujet en dehors du cadre de cette thèse ([Thatcher and Wright, 1968, Courcelle, 1992, Niehren and Podelski, 1993]). Cependant, il est intéressant de remarquer que la fonction de

construction peut être vue comme un isomorphisme entre les algèbres des arbres à arité arbitraire et des arbres de construction.

Une Γ-algèbre $A = (E, \Gamma)$ est défini par un ensemble E d'éléments et un ensemble Γ de symboles n-aires interprétables comme des fonctions de E^n dans E.

Propriété 2.2. *L'ensemble des arbres à arité arbitraire* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *et l'alphabet de construction* $\Gamma_{@}$ *forment la* $\Gamma_{@}$*-algèbre* $(\mathsf{T}^{\mathsf{s}}_{\Gamma}, \Gamma_{@})$.

Preuve : L'alphabet de construction $\Gamma_{@}$ *est constitué d'un ensemble de symboles d'arité nulle et du symbole binaire* @. *Chacun des symboles d'arité nulle a peut s'interpréter comme une fonction d'arité nulle à valeur dans* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$, *dont la valeur constante est l'arbre constitué de la feuille a. Le symbole binaire* @ *peut s'interpréter comme la fonction de* $\mathsf{T}^{\mathsf{s}}_{\Gamma} \times \mathsf{T}^{\mathsf{s}}_{\Gamma}$ *dans* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *que nous avons déja définie :* $@^s$. ◄

Propriété 2.3. *L'ensemble des arbres de construction* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ *et l'alphabet de construction* $\Gamma_{@}$ *forment une* $\Gamma_{@}$*-algèbre* $(\mathsf{T}^{\mathsf{a}}_{\Gamma}, \Gamma_{@})$

Preuve : L'algèbre est définie de façon similaire à celle de la propriété précédente. Seule change l'interprétation de @ *qui ici est* $@^a$. ◄

Propriété 2.4. *La fonction de construction* $\mathsf{c}_{\mathsf{arbre}}$ *est un isomorphisme d'algèbre de* $(\mathsf{T}^{\mathsf{s}}_{\Gamma}, \Gamma_{@})$ *vers* $(\mathsf{T}^{\mathsf{a}}_{\Gamma}, \Gamma_{@})$.

Preuve : La définition de $\mathsf{c}_{\mathsf{arbre}}$ *implique que c'est un morphisme. Or* $\mathsf{c}_{\mathsf{arbre}}$ *est une bijection de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *vers* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$. *Donc c'est un isomorphisme d'algèbre de* $(\mathsf{T}^{\mathsf{s}}_{\Gamma}, \Gamma_{@})$ *vers* $(\mathsf{T}^{\mathsf{a}}_{\Gamma}, \Gamma_{@})$. ◄

2.2.2 Une correspondance entre les domaines des arbres

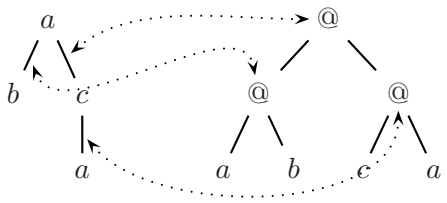
Par la suite, pour la définition des runs puis pour le calcul de requêtes, nous aurons besoin de manipuler non pas des arbres en tant que termes, mais des arbres en tant qu'ensemble structuré de nœuds (voir chapitre 2). Dans la mesure où notre but est d'avoir une approche homogène des arbres à arité arbitraire et de leur arbres de construction, il serait intéressant d'établir une correspondance entre les nœuds des arbres à arité arbitraire et les nœuds des arbres de construction, de manière à ce que toute manipulation de nœuds sur une arbre à arité arbitraire puisse être définie de manière analogue dans l'arbre de construction correspondant.

Il est clair qu'une telle correspondance est impossible, car le nombre de nœuds d'un arbre à arité arbitraire est différent du nombre de nœuds d'un arbre de construction.

En observant la définition de la fonction de construction, on peut pourtant voir qu'elle induit naturellement une correspondance entre les éléments des arbres à arité arbitraire et ceux de leurs arbres de construction :

- Chaque feuille d'un arbre de construction décrit la construction d'un nœud de l'arbre à arité arbitraire correspondant.
- Chaque nœud interne d'un arbre de construction décrit la construction d'un arbre $t_1 @^s t_2$ à partir des arbres t_1 et t_2, et donc l'ajout de l'arête qui relie ces deux arbres.

Nous avons donc mis en évidence que la fonction de construction induit une correspondance entre l'ensemble des nœuds et des arêtes des arbres à arité arbitraire et l'ensemble des nœuds de leurs arbres de construction.



$$
\begin{array}{rclrcl}
\mathsf{c}_{\mathsf{dom}}(\epsilon) & = & 1.1 & & & \\
\mathsf{c}_{\mathsf{dom}}(1) & = & 1.2 & \mathsf{c}_{\mathsf{dom}}(\mathsf{arete}(1)) & = & 1 \\
\mathsf{c}_{\mathsf{dom}}(2) & = & 2.1 & \mathsf{c}_{\mathsf{dom}}(\mathsf{arete}(2)) & = & \epsilon \\
\mathsf{c}_{\mathsf{dom}}(2.1) & = & 2.2 & \mathsf{c}_{\mathsf{dom}}(\mathsf{arete}(2.1)) & = & 2
\end{array}
$$

Figure 2.3: Correspondance entre les domaines d'un arbre à arité arbitraire et de son arbre de construction.

Pour pouvoir formaliser cette correspondance, nous allons définir le *domaine étendu* d'un arbre à arité arbitraire, l'ensemble de ses nœuds et de ses arêtes. Toute arête descendant d'un nœud π_1 vers un nœud π_2 sera notée $\mathsf{arete}(\pi_2)$. On définit ainsi le domaine étendu d'un arbre à arité arbitraire t :

$$\mathsf{dom}^{\mathsf{e}}(t) = \mathsf{dom}(t) \cup \{\mathsf{arete}(\pi) | (\pi \in \mathsf{dom}(t)) \wedge (\pi \neq \epsilon)\}$$

Définition 2.4. *La* correspondance *de domaine d'un arbre t est une fonction du domaine étendu de t vers le domaine de $\mathsf{c}_{\mathsf{arbre}}(t)$. Elle est définie ainsi :*

$$
\begin{array}{lll}
\mathsf{c}_{\mathsf{dom}}(t) : \ \mathsf{dom}^{\mathsf{e}}(t) & \rightarrow & \mathsf{dom}(\mathsf{c}_{\mathsf{arbre}}(t)) \\
\mathsf{c}_{\mathsf{dom}}(a)(\epsilon) & = & \epsilon \\
\mathsf{c}_{\mathsf{dom}}(t_1 @^s t_2)(\pi) & = & \mathsf{Si}\ \pi\ \in \mathsf{dom}^{\mathsf{e}}(t_1)\ \mathsf{Alors}\ \mathsf{c}_{\mathsf{dom}}(t_1)(\pi) \\
 & & \mathsf{Si}\ \pi\ \in \mathsf{dom}^{\mathsf{e}}(t_2)\ \mathsf{Alors}\ \mathsf{c}_{\mathsf{dom}}(t_2)(\mathsf{descente}(\pi)) \\
 & & \mathsf{Sinon}\ \pi = \mathsf{derniere\_arete}(t)
\end{array}
$$

L'opérateur $\mathsf{descente}$ *renvoie respectivement u et $\mathsf{arete}(u)$ pour les éléments $n.u$ et $\mathsf{arete}(n.u)$. Il n'est pas défini pour ϵ. Il permet de calculer l'adresse d'un nœud d'un arbre t dans un sous-arbre t' de t dont la racine est un fils de t. La fonction* $\mathsf{derniere\_arete}(t)$ *renvoie l'arête située entre la racine de t et son dernier fils.*

Cette correspondance est illustrée Fig.2.3.

Propriété 2.5. *Soit t un arbre de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$. La fonction $\mathsf{c}_{\mathsf{dom}}(t)$ est une bijection de $\mathsf{dom}^{\mathsf{e}}(t)$ vers $\mathsf{dom}(\mathsf{c}_{\mathsf{arbre}}(t))$.*

Preuve : Nous allons prouver cette propriété par récurrence sur la construction de t.

Si $t = a$, $\mathsf{dom}^{\mathsf{e}}(t)$ et $\mathsf{dom}(\mathsf{c}_{\mathsf{arbre}}(t))$ n'ont qu'un seul élément, donc $\mathsf{c}_{\mathsf{dom}}(t)$ est une bijection.

Si $t = t_1@^s t_2$, *supposons par récurrence que* $\mathsf{c_{dom}}(t_1)$ *et* $\mathsf{c_{dom}}(t_2)$ *soient des bijections.*

On peut diviser le domaine de t *en trois parties disjointes :* $\mathsf{dom^e}(t_1)$, $\mathsf{dom^e}(t_2)$ *et* $\{\mathsf{derniere\_arete}(t)\}$, *conformément à la définition du domaine que nous avons donnée. De même, on peut diviser le domaine de* $\mathsf{c_{arbre}}(t)$ *en trois parties disjointes :* $\mathsf{dom}(\mathsf{c_{arbre}}(t_1))$, $\mathsf{dom}(\mathsf{c_{arbre}}(t_2))$ *et* ϵ.

La fonction $\mathsf{c_{dom}}(t_1@^s t_2)$ *envoie* $\mathsf{dom^e}(t_1)$ *sur* $\mathsf{dom}(\mathsf{c_{arbre}}(t_1))$. *Or sa restriction à* $\mathsf{dom^e}(t_1)$ *est égale à* $\mathsf{c_{dom}}(t_1)$, *donc c'est une bijection de* $\mathsf{dom^e}(t_1)$ *vers* $\mathsf{dom}(\mathsf{c_{arbre}}(t_1))$. *De même, la restriction de* $\mathsf{c_{dom}}(t_1@^s t_2)$ *est une bijection de* $\mathsf{dom^e}(t_2)$ *vers* $\mathsf{dom}(\mathsf{c_{arbre}}(t_2))$. *Enfin, la restriction de* $\mathsf{c_{dom}}(t_1@^s t_2)$ *restreinte au singleton* $\{\mathsf{derniere\_arete}(t_1@^s t_2\}$ *est une bijection vers le singleton* $\{\epsilon\}$.

Donc $\mathsf{c_{dom}}(t_1@^s t_2)$ *est une bijection de* $\mathsf{dom^e}(t_1@^s t_2)$ *vers* $\mathsf{dom}(\mathsf{c_{arbre}}(t_1@^s t_2))$. ◀

2.3 Les automates à pas

Nous allons utiliser les correspondances que nous avons définies entre arbres à arité arbitraire et arbres de construction pour définir un nouveau type d'automate adapté aux arbres à arité arbitraire : les automates à pas. L'idée est d'utiliser les mêmes automates pour les arbres de construction et pour les arbres à arité arbitraire, et ainsi de transposer le formalisme classique des arbres à arité fixe aux arbres à arité arbitraire.

Pour introduire les automates à pas, nous allons en donner la définition formelle, puis décrire leur comportement, c'est-à-dire le traitement des arbres qui leur est associé.

Définition 2.5. *Un* automate à pas *sur la signature à arité arbitraire* Γ *est un automate d'arbre classique sur la signature de construction* $\Gamma_@$. *Il est défini par :*

- *Un ensemble d'états, noté* $\mathsf{etats}(A)$
- *Un ensemble d'états finaux, noté* $\mathsf{finaux}(A)$, *tel que* $\mathsf{finaux}(A) \subseteq \mathsf{etats}(A)$
- *Un ensemble de règles de transition, du type :*
  - $a \to q$, *avec* $a \in \Gamma$ *et* $q \in \mathsf{etats}(A)$
  - $@(q_1, q_2) \to q$, *avec* $q_1 \in \mathsf{etats}(A)$, $q_2 \in \mathsf{etats}(A)$ *et* $q \in \mathsf{etats}(A)$. *Nous utiliserons plutôt la syntaxe* $q_1@q_2 \to q$ *pour spécifier ces règles.*

L'originalité des automates à pas ne réside pas dans leur définition, mais dans leur capacité à traiter à la fois les arbres de construction et les arbres à arité arbitraire.

Pour définir le traitement d'un automate sur un arbre, on dispose de deux notions distinctes : l'évaluation et le calcul de runs. Pour les automates à pas, nous allons donner deux définitions de chacune de ces deux notions : l'une pour les arbres de construction, qui coïncidera avec la définition classique, et l'autre pour les arbres à arité arbitraire. Nous montrerons que les deux définitions *correspondent*, au sens de la correspondance définie dans la partie 2.2.

Nous terminerons ensuite notre étude des automates à pas par une comparaison de leur expressivité avec celle des autres formalismes d'automates d'arbres à arité arbitraire existants.

2.3.1 Évaluation d'arbres avec les automates à pas

L'évaluation d'arbres de construction par des automates à pas est identique à l'évaluation d'arbres binaires par des automates classiques. Cette définition est fournie dans les préliminaires. Pour distinguer cette évaluation et celle des arbres à arité arbitraire que nous allons définir après, nous la noterons $\mathsf{eval^a}_A()$.

Dans le cas des arbres à arité arbitraire, nous allons définir une évaluation qui se calculera directement sur les arbres à arité arbitraire, mais qui sera *correspondante*, c'est-à-dire qu'elle donnera le même résultat pour les arbres à arité arbitraire et leurs arbres de construction.

Nous allons commencer par la définir formellement avant d'illustrer son fonctionnement sur un exemple.

Définition 2.6. *L'*évaluation *d'un arbre de* $\mathsf{T^s_\Gamma}$ *par un automate à pas sur une signature* Γ *est définie ainsi :*

$$\begin{array}{lllll} \mathsf{eval^s}_A() : & \mathsf{T^s_\Gamma} & \to & 2^{\mathsf{etats}(A)} \\ & \mathsf{eval^s}_A(a) & = & \{q \mid a \to q \in \mathsf{regles}(A)\} \\ & \mathsf{eval^s}_A(t_1 @^\alpha t_2) & = & \{q \mid q_1 \in \mathsf{eval^s}_A(t_1),\ q_2 \in \mathsf{eval^s}_A(t_2), q_1 @ q_2 \to q \in \mathsf{regles}(A)\} \end{array}$$

La correspondance entre l'évaluation des arbres à arité arbitraire et celle des arbres de construction s'exprime simplement par la propriété suivante :

Propriété 2.6. *Pour tout arbre à arité arbitraire* t *sur* Γ *et tout automate à pas* A *sur* Γ*,*

$$\mathsf{eval^a}_A(t) = \mathsf{eval^s}_A(\mathsf{c_{arbre}}(t))$$

Preuve : La preuve se fait par récurrence sur la construction de t*.* ◄

Si les évaluations des arbres à arité arbitraire et de leurs arbres de construction ont des définitions tout à fait analogue, le calcul de ces évaluations ne se déroule pas exactement de la même manière.

Dans le cas des arbres de construction, il s'agit d'une classique récurrence sur les sous-arbres : l'évaluation d'un arbre $t = t_1 @^a t_2$ se fait en calculant récursivement les évaluations de t_1 et de t_2, puis en appliquant les règles de l'automate selon la définition 2.6.

Dans le cas des arbres à arité arbitraire, c'est un peu plus compliqué, car dans l'expression $t = t_1 @^s t_2$, t_1 ne correspond pas à un sous-arbre de t. Les objets récursivement évalués lors de l'évaluation d'un arbre t sont les *sous-arbres partiels* de t, c'est-à-dire les sous-arbres de t amputés d'un nombre quelconque des fils les plus à droite de leur racine. L'exemple qui suit illustre cette notion en explicitant l'ensemble des sous-arbres partiels d'un arbre à arité arbitraire.

L'évaluation d'un arbre à arité arbitraire se passe ainsi : chaque sous-arbre partiel $f(t_1, \ldots, t_n)$ est évalué par une évaluation récursive de $f(t_1, \ldots, t_{n-1})$ et de t_n, puis par l'application des règles de l'automate selon la définition 2.6.

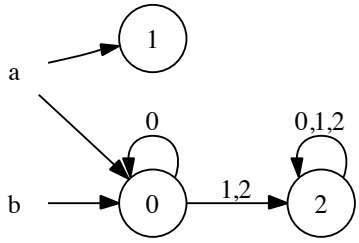


Figure 2.4: Une représentation graphique de l'automate A_a

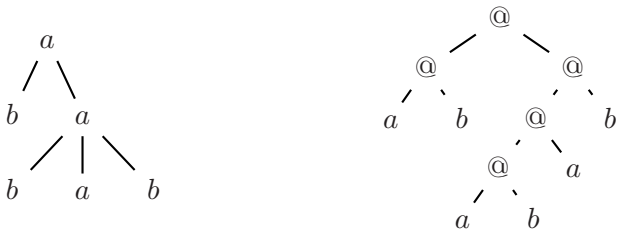


Figure 2.5: L'arbre t et son arbre de construction.

Exemple 2.1. *Nous allons illustrer par un exemple le fonctionnement d'un automate à pas sur les arbres de construction et sur les arbres binaires.*

Nous allons définir un automate à pas A_a qui reconnaît exactement les arbres à arité arbitraire définis sur l'alphabet $\Gamma = \{a, b\}$ possédant au moins une feuille étiquetée par a.

L'automate A_a est défini ainsi :

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{0,1,2\} \\ \mathsf{finaux}(A) & = & \{1,2\} \\ \mathsf{regles}(A) & = & \{a \to 1,\ a \to 0,\ b \to 0, \\ & & 0@0 \to 0,\ 0@1 \to 2,\ 0@2 \to 2, \\ & & 2@0 \to 2,\ 2@1 \to 2,\ 2@2 \to 2\} \end{array}$$

Nous donnons une représentation graphique de cet automate Fig. 2.4. Chaque transition $q_1 \xrightarrow{q_2} q_3$ représente une règle $q_1@q_2 \to q_3$. Notons que les transitions sont étiquetées par des états, ce qui rend l'interprétation de cette représentation assez difficile.

Nous allons donner un exemple d'évaluation d'arbre avec l'automate A_a. L'arbre que nous allons évaluer est donné Fig. 2.5 avec son arbre de construction. L'ensemble des sous-arbres récursivement évalués et de leurs arbres de construction, ainsi que des résultats de ces évaluation est fourni Fig. 2.6.

Une fois la notion d'évaluation définie, la reconnaissance d'un arbre par un automate en découle immédiatement : un arbre est reconnu par un automate si et seulement si son évaluation par cet automate contient un état final.

Il en découle la notion de *langage d'un automate d'arbre*, l'ensemble des arbres reconnus par un automate. La capacité des automates à pas à évaluer à la fois des arbres à arité arbitraire et des arbres de construction nous amène à établir deux définitions des langages d'automates à pas, l'une dans les arbres à arité arbitraire, l'autre dans les arbres de construction :

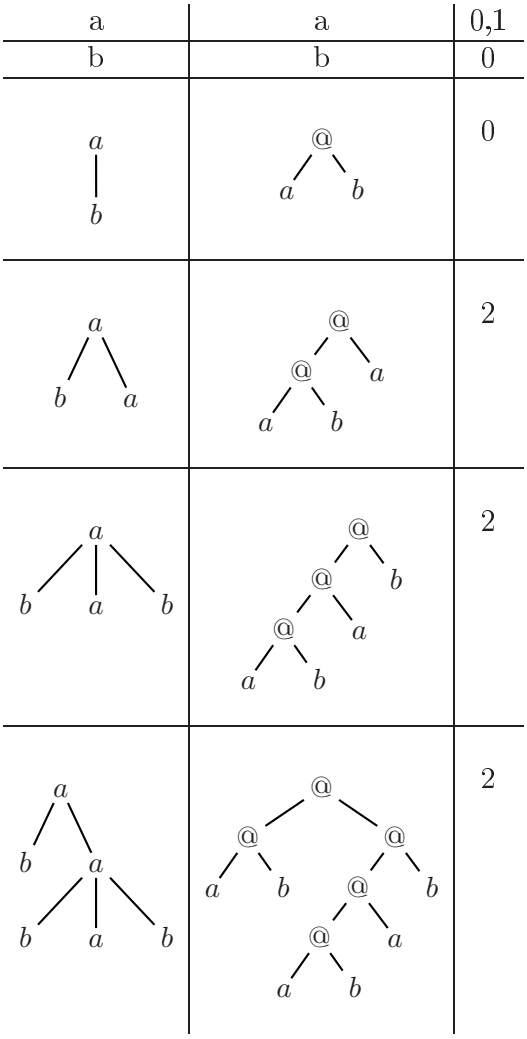


Figure 2.6: Les sous-arbres partiels de t, les sous-arbres de $\mathsf{c}_{\mathsf{arbre}}(t)$ correspondant, les résultats de leurs évaluations

Définition 2.7. *Le* langage des arbres à arité arbitraire d'un automate à pas A *est défini ainsi :*

$$\mathsf{langage}^{\mathsf{s}}(A) = \{t | \exists q \in \mathsf{finaux}(A)\ (q \in \mathsf{eval}^{\mathsf{s}}{}_A(t))\}$$

Définition 2.8. *Le* langage des arbres de construction d'un automate à pas A *est défini ainsi :*

$$\mathsf{langage}^{\mathsf{a}}(A) = \{c | \exists q \in \mathsf{finaux}(A)\ (q \in \mathsf{eval}^{\mathsf{a}}{}_A(c))\}$$

Et ces langages correspondent, au sens défini dans la partie 2.2 Cela peut s'exprimer par cette propriété, conséquence directe de la propriété 2.6 :

Propriété 2.7. *Pour tout automate à pas A sur Γ*

$$\mathsf{langage^a}(A) = \mathsf{c_{arbre}}(\mathsf{langage^s}(A))$$

Preuve :

$$\begin{aligned}
\mathsf{langage^a}(A) &= \{c | \exists q \in \mathsf{finaux}(A)\ (q \in \mathsf{eval^a}_A(c))\} && (2.1)\\
&= \{c | \exists q \in \mathsf{finaux}(A)\ (q \in \mathsf{eval^s}_A(\mathsf{c_{arbre}}^{-1}(c)))\} && (2.2)\\
&= \{\mathsf{c_{arbre}}(t) | \exists q \in \mathsf{finaux}(A)\ (q \in \mathsf{eval^s}_A(t))\} && (2.3)\\
&= \mathsf{c_{arbre}}(\mathsf{langage^s}(A)) && (2.4)
\end{aligned}$$

Le passage de la ligne 2.1 à la ligne 2.2 se déduit de la propriété 2.6. La ligne 2.3 se déduit ensuite de la définition de $\mathsf{eval^s}_A()$*, et enfin la ligne 2.4 se déduit de la définition de* $\mathsf{langage^s}$*.*

◀

2.3.2 Calcul de runs avec des automates à pas

Comme pour l'évaluation, la définition des runs des automates à pas sur les arbres de construction est identique à celle de runs des automates d'arbres classiques sur les arbres binaires.

En revanche, la transposition de la notion de run dans les arbres à arité arbitraire va poser un problème. En effet, le principe d'un run est d'associer le résultat d'un calcul récursif à un élément du domaine de l'arbre. Dans le cas des arbres binaire, on associe un état évaluant un sous-arbre à la racine de ce sous-arbre. Dans le cas des arbres à arité arbitraire, les évaluations récursives portent sur des sous-arbres partiels, et il n'est pas possible d'associer un nœud différent à chaque sous-arbre partiel. Par exemple, l'arbre $f(a, b, c)$ contient 4 nœuds et possède 7 sous-arbres partiels : f, a, b, c, $f(a)$, $f(a, b)$ et $f(a, b, c)$.

Pour résoudre ce problème, nous allons utiliser la notion de domaine étendu que nous avons introduite en 2.2.2. On peut associer à chaque sous-arbre partiel exactement un élément du domaine étendu : chaque sous-arbre partiel réduit à un nœud sera associé à ce nœud, et chaque sous-arbre partiel qui n'est pas réduit à un nœud sera associé à l'arête située entre sa racine et son dernier fils. Dans l'exemple précédent, les sous-arbres partiels f, a, b, c seront associés aux nœuds correspondants, et les sous-arbres partiels $f(a)$, $f(a, b)$ et $f(a, b, c)$ seront respectivement associés aux arêtes séparant le nœud f et les nœuds a, b et c. On appellera *tête* d'un sous-arbre partiel l'élément de son domaine étendu qui lui est associée. La *tête* d'un sous-arbre partiel correspond à la racine du sous-arbre correspondant dans l'arbre de construction.

Définition 2.9. *Soit* t *un arbre à arité arbitraire sur l'alphabet* Γ*. La* tête *de* t *est un élément de* $\mathsf{dom^e}(t)$ *défini ainsi :*

- *Si* $t = a$ *alors* $\mathsf{tete}(t) = \mathsf{racine}(t)$
- *Si* $t = f(t_1, \ldots, t_n)$*, alors* $\mathsf{tete}(t) = \mathsf{derniere\_arete}(t)$

Cette définition est illustrée Fig.2.7.

Cela nous permet de donner une définition du run d'un automate à pas pour les arbres à arité arbitraire et pour les arbres de construction :

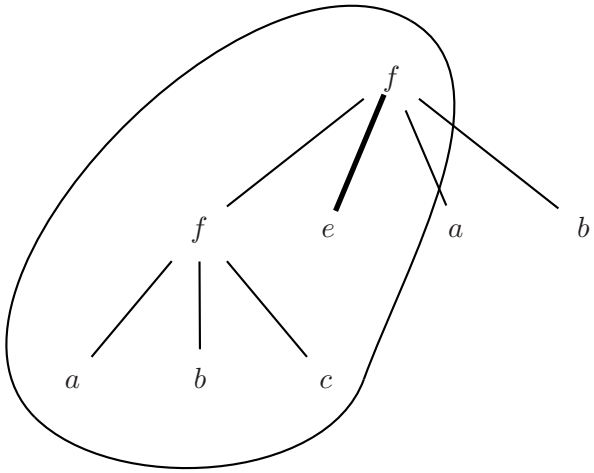


Figure 2.7: L'ensemble entouré est un sous-arbre partiel, l'arête en gras est sa tête, c'est-à-dire l'arête qui lui est associé.

Définition 2.10. *Soit A un automate à pas sur un alphabet Γ. Un run de A sur un arbre t de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ est une fonction $r : \mathsf{dom}^{\mathsf{e}}(t) \to \mathsf{etats}(A)$ qui satisfait les règles de A, c'est-à-dire que :*

- *Si $t = a$, alors $(a \to r(\mathsf{tete}(t))) \in \mathsf{regles}(A)$*
- *Si $t = t_1 @^s t_2$, alors :*
  - *$r_{|\mathsf{dom}^{\mathsf{s}}(t_1)}$ est un run de A sur t_1*
  - *$r_{|\mathsf{dom}^{\mathsf{s}}(t_2)}$ est un run de A sur t_2*
  - *$(r(\mathsf{tete}(t_1))@r(\mathsf{tete}(t_2)) \to r(\mathsf{tete}(c))) \in \mathsf{regles}(A)$*

On note $\mathsf{runs}_{\mathsf{A}}(\mathsf{t})$ l'ensemble des runs de A sur t.

On peut remarquer que cette définition est similaire à la définition classique d'un run sur un arbre binaire, excepté qu'on associe un état à la *tête* de chaque sous-arbre partiel, au lieu de l'associer à la racine de chaque sous-arbre.

Pour donner un exemple de run d'un automate sur un arbre, reprenons l'automate A_a et l'arbre t de l'exemple 2.1. La figure 2.8 donne un exemple de run de A_a sur t et sur son arbre de construction. Il s'agit en fait du seul run réussi de cet automate sur cet arbre.

Cet exemple met en évidence la correspondance entre les deux runs, l'un sur l'arbre à arité arbitraire, l'autre sur l'arbre de construction, affectant les mêmes états aux nœuds correspondants des arbres correspondants. Cette correspondance peut se formaliser par la propriété suivante :

Propriété 2.8. *Soit A un automate à pas, t un arbre de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ et r une fonction de $\mathsf{dom}^{\mathsf{e}}(t)$ dans $\mathsf{etats}(A)$*

$$r \in \mathsf{runs}_{\mathsf{A}}(\mathsf{t}) \Leftrightarrow r \circ \mathsf{c}_{\mathsf{dom}}(t)^{-1} \in \mathsf{runs}_{\mathsf{A}}(\mathsf{c}_{\mathsf{arbre}}(\mathsf{t}))$$

Enfin, la validité de cette définition du run peut être mise en évidence par cette propriété fondamentale du run qui lie runs et évaluation. Là encore, il s'agit de la transposition d'une

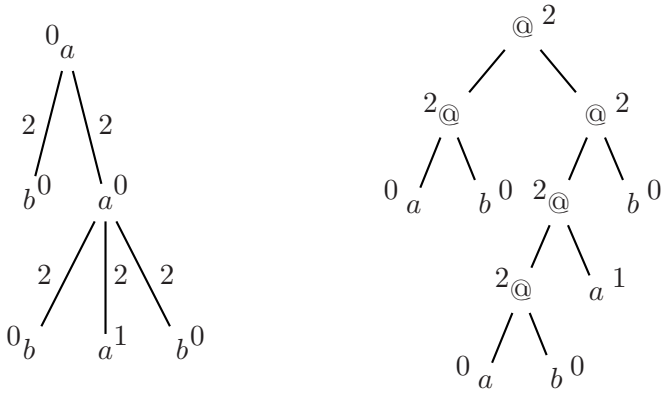


Figure 2.8: L'unique run réussit de l'automate A_a (Fig. 2.4) sur l'arbre t.

propriété fondamentale liant runs et évaluation dans les arbres binaires, utilisant la *tete* au lieu de la racine comme élément de domaine représentant un arbre.

Propriété 2.9. *Soit A un automate à pas et t un arbre de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$.*

$$\mathsf{eval}^{\mathsf{s}}{}_A(t) = \{r(\mathsf{tete}(t)) | r \in \mathsf{runs_A(t)}\}$$

2.3.3 Expressivité des automates à pas

L'expressivité des automates à pas sur les arbres de construction ne nécessite pas d'étude particulière : ils reconnaissent par définition l'ensemble des langages d'arbres réguliers à arité fixe sur $\Gamma_@$.

Dans le cas des langages à arité arbitraire, il n'y a pas de définition normalisée des langages réguliers. Cependant, d'autres formalismes ont déjà été employés pour traiter les arbres à arité arbitraire avec des automates d'arbres, et il est intéressant de comparer l'expressivité des automates à pas avec celles de ces formalismes.

Automates à pas et automates à haies

Les automates à haies sont la transposition la plus intuitive du formalisme des automates d'arbres classiques dans les arbres à arité arbitraire. L'idée et de remplacer les règles à arité fixe du type $a(q_1, \ldots, q_n) \to q$ par des règles à arité arbitraire du type $a(L) \to q$, avec L langage régulier de mots d'états. Pour des informations complètes sur les automates à haies, consulter [Bruggemann-Klein et al., 2001, Thatcher, 1967].

Théorème 2.1. *Les automates à pas et les automates à haies ont la même expressivité pour décrire les langages d'arbres à arité arbitraire.*

La preuve complète de ce théorème est fournie en annexe. Elle est fondée sur la construction d'un opérateur de conversion d'automate à haie en automate à pas équivalent, et d'un opérateur inverse.

Automates à pas et automates sur les codages binaires

Il est prouvé dans [Neven and Schwentick, 2002] que les automates sur les codages frère-fils ont une expressivité équivalente aux automates à haies pour décrire des langages à arité arbitraire. Il en résulte le corollaire suivant du théorème 2.1 :

Corollaire 2.1. *Les automates binaires sur les codages frère-fils et les automates à pas ont la même expressivité pour décrire les arbres à arité arbitraire.*

Chapitre 3

Les transducteurs de sélection de nœuds

Nous allons introduire dans ce chapitre les transducteurs de sélection de nœuds, automates d'arbres définissant des requêtes monadiques, c'est-à-dire des fonctions de sélection de nœuds, dans les arbres. Ce sont ces objets que nous utiliserons dans les chapitres suivants dans nos travaux sur l'apprentissage de requêtes.

L'utilisation d'automates d'arbres pour sélectionner des nœuds dans les arbres n'est pas une idée nouvelle. On peut par exemple citer les *Selection automata* [Frick et al., 2003] ou les *Query automata* [Neven and Schwentick, 2002]. Dans un cas comme dans l'autre, c'est l'aspect *calculatoire* des automates qui est mis en valeur dans ces approches. Les automates sont enrichis de règles de sélection, et ainsi l'évaluation d'un arbre s'accompagne d'une sélection de certains de ses nœuds, permettant ainsi le calcul de requêtes.

Dans nos travaux, c'est en tant que reconnaisseurs de langages d'arbres que les automates d'arbres vont avant tout nous intéresser. En effet, toute requête monadique peut être identifiée à un langage d'arbres, et donc à l'automate d'arbres reconnaissant ce langage si la requête est régulière. Cela nous permettra ensuite d'apprendre des requêtes en apprenant ces automates.

Dans la mesure ou une requête monadique s'exprime naturellement par une formule logique, on peut trouver l'origine des idées développées ici dans l'étude des correspondances entre logique et automates [Thatcher and Wright, 1968, Comon et al., 1997]. Nous reviendrons plus en détail sur ce sujet lorsque nous étudierons l'expressivité des requêtes que nous allons définir. Par ailleurs, l'idée d'apprendre des langages d'arbres annotés pour définir des requêtes avait déjà été étudié dans [Kosala et al., 2003].

Dans ce chapitre, nous allons commencer par établir ces ponts entre requêtes, langages et automates, introduire et étudier les Transducteurs de Sélection de Nœuds (TSN) dans les arbres binaires. Nous allons ensuite nous intéresser à l'élagage des arbres pour la définition de requêtes, ce qui nous amènera à introduire les Transducteurs de Sélection de Nœuds élagueurs (TSNe), TSN définis dans les arbres élagués. Enfin, nous montrerons comment utiliser les résultats du premier chapitre pour définir des TSN dans les arbres à arité arbitraire. Nous nous interesserons également à l'expressivité des TSN, à la fois dans les arbres binaires et dans les arbres à arité arbitraire.

3.1 Des requêtes aux automates

Nous allons commencer par montrer comment représenter des requêtes par des automates d'arbres. Cela va se faire en deux étapes :

- Dans un premier temps, nous allons introduire la notion de langage de requête. Pour cela, nous allons montrer qu'une requête peut être vue comme une fonction d'annotation d'arbres, et donc qu'une requête peut être identifiée à l'ensemble des arbres qu'elle annote.
- Dans un deuxième temps, nous verrons que cela induit naturellement la représentation des requêtes dites *régulières* par des automates d'arbres.

3.1.1 Des requêtes aux langages

Nous allons travailler sur l'alphabet à arité fixe Σ tel que tous les symboles de Σ ait une arité égale à 0 ou 2. Ainsi, l'ensemble des arbres à arité fixe T_Σ défini sur Σ est un ensemble d'arbres binaires.

Commençons par donner la définition des requêtes monadiques dans les arbres :

Définition 3.1. *Une* requête monadique *sur* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ *est une fonction* q *qui associe à tout arbre* t *un sous-ensemble* $\mathsf{q}(t)$ *de* $\mathsf{dom}(t)$.

Le principe de la représentation des requêtes par des langages d'arbres repose sur l'idée que sélectionner des éléments d'un domaine revient à les annoter par des booléens : vrai pour les éléments sélectionnés, faux pour les autres. L'ensemble des arbres annotés par une requête forme un langage d'arbre qui caractérise cette requête.

Soit $\mathsf{T}_{\mathsf{Bool}}$ l'ensemble des arbres binaires booléens, c'est-à-dire définis sur l'alphabet $\{\mathsf{vrai}, \mathsf{faux}\}$ dans lequel chaque symbole peut prendre l'arité 0 ou 2. Nous appellerons *annotation de* t tout arbre β défini sur l'alphabet $\mathsf{T}_{\mathsf{Bool}}$ tel que $\mathsf{dom}(\beta) = \mathsf{dom}(t)$. Une requête définit une annotation de t pour chaque t de T_Σ. Nous pouvons formaliser cela en définissant la *fonction d'annotation* d'une requête :

Définition 3.2. *Soit* q *une requête sur* T_Σ.

La fonction d'annotation de q *est la fonction de* T_Σ *dans* $\mathsf{T}_{\mathsf{Bool}}$ *définie ainsi :*

Pour tout t *dans* T_Σ, $\mathsf{annot_q(t)}$ *est définie par :*

- $\mathsf{dom}(\mathsf{annot_q(t)}) = \mathsf{dom}(t)$
- *Pour tout* π *dans* $\mathsf{dom}(t)$ *:*
  - *Si* π *est dans* $\mathsf{q}(t)$*, alors* $\mathsf{annot_q(t)}(\pi) = \mathsf{vrai}$
  - *Sinon* $\mathsf{annot_q(t)}(\pi) = \mathsf{faux}$

En "fusionnant" un arbre t sur Σ et une annotation β de t, on obtient un *arbre annoté*, c'est-à-dire un arbre défini sur l'alphabet $\Sigma \times \mathsf{Bool}$, pour lequel chaque nœud est constitué d'un symbole de Σ et de son annotation dans β. On note cet arbre annoté $t \times \beta$. Formellement, $t \times \beta$ est ainsi défini :

- $\mathsf{dom}(t \times \beta) = \mathsf{dom}(t)$
- Pour tout π dans $\mathsf{dom}(t \times \beta)$, $(t \times \beta)(\pi) = (t(\pi), \beta(\pi))$.

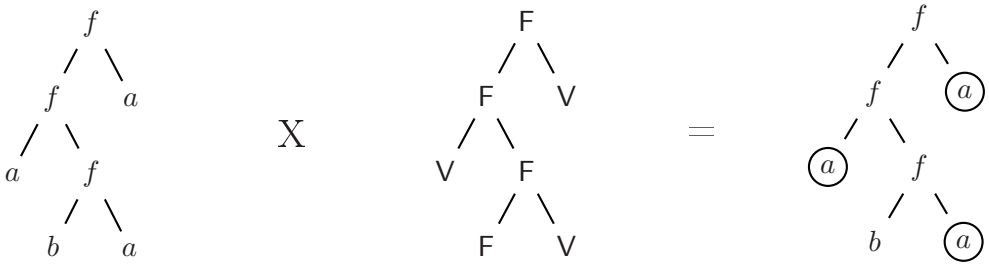


Figure 3.1: L'arbre binaire t, son annotation β par la requête q, et l'arbre annoté $t \times \beta$. Pour caractériser les annotations, on utilisera toujours la convention suivante : les nœuds annotés par vrai seront entourés d'un cercle

Exemple 3.1. *Soit* q *la requête sélectionnant tous les nœuds étiquetés par* a*. La figure 3.1 représente un arbre binaire* t*, son annotation par la requête* q *et l'arbre annoté par la requête* q*.*

Une requête peut être représentée par l'ensemble des arbres annotés par elle. Cela nous amène à définition suivante des langages de requêtes :

Définition 3.3. *Soit* q *une requête sur* T_Σ*. Le langage de la requête* q *est un langage de* $\mathsf{T}_{\Sigma \times \mathsf{Bool}}$ *défini ainsi :*

$$\mathsf{langage(q)} = \{t \times \mathsf{annot_q(t)} | t \in \mathsf{T}_\Sigma\}$$

3.1.2 Des langages aux automates

Nous avons vu qu'il était possible de représenter simplement une requête monadique dans les arbres par un langage d'arbre. Il est alors naturel de penser à représenter une requête par un automate d'arbre reconnaissant ce langage. C'est ce que nous allons faire d'abord en introduisant la notion de *description stricte* d'une requête par un automate, qui est la définition qui découle naturellement de l'explication que nous venons de donner, puis nous introduirons la notion de *description* d'une requête par un automate, notion plus souple et plus intéressante dans la pratique.

Avant tout, nous allons nous limiter aux requêtes que l'on peut décrire avec les automates d'arbres. Seuls les langages d'arbres *réguliers* sont descriptibles par des automates d'arbres. Nous appellerons *régulières* les requêtes monadiques correspondantes :

Définition 3.4. *Une requête monadique* q *sur* T_Σ *est* régulière *si* $\mathsf{langage(q)}$ *est régulier, c'est-à-dire si il existe un automate d'arbres* A *tel que* $\mathsf{langage}(A) = \mathsf{langage}(q)$*.*

Il existe bien évidemment des requêtes non régulières. On peut en créer facilement en reprenant les exemples classiques de langages non réguliers d'arbres. Typiquement, les requêtes impliquant un comptage non borné sont non régulières. Par exemple, "sélectionner tous les a dans les arbres qui comptent autant de a que de b" n'est pas une requête régulière. Dans la pratique, les requêtes régulières seront assez expressives pour nos besoins en extraction

d'information : leur expressivité est la même que celle de la logique monadique du second ordre. Nous reviendrons en détail sur ce sujet en 3.2.4.

Considérons maintenant que nous manipulons exclusivement des requêtes monadiques régulières.

Définition 3.5. *Soit* q *une requête sur* T_Σ. *Un automate* A décrit strictement la requête q *si et seulement si* $\mathsf{langage}(A) = \mathsf{langage}(\mathsf{q})$

La notion de *description stricte* d'une requête sur T_Σ par un automate est tout à fait satisfaisante d'un point de vue théorique, mais un peu contraignante, car elle impose à l'automate d'être *complet*, c'est-à-dire de reconnaître toutes les annotations des arbres de T_Σ, y compris les annotations *nulles*. Les annotations nulles sont les annotations des arbres ne contenant aucune valeur vrai. On note $\beta_0(t)$ l'annotation nulle de t.

Définition 3.6. *Un automate d'arbre* A *défini sur* $\Sigma \times \mathsf{Bool}$ *est* complet *si et seulement si :*

$$\forall t \in \mathsf{T}_\Sigma \; \exists \beta \in \mathsf{T}_{\mathsf{Bool}} \; (t \times \beta \in \mathsf{langage}(A)$$

Il n'est pas nécessaire qu'un automate sur $\Sigma \times \mathsf{Bool}$ soit complet pour décrire une requête : il suffit à un automate de reconnaître les annotations *non nulles* associées à une requête pour spécifier cette requête. Cela va nous permettre d'introduire une définition plus souple de la description d'une requête par un automate. Pour la formaliser, nous allons introduire la fonction $\mathsf{complete}$, définie de $2^{\mathsf{T}_{\Sigma \times \mathsf{Bool}}}$ dans $2^{\mathsf{T}_{\Sigma \times \mathsf{Bool}}}$, qui complète un langage d'annotation en ajoutant les annotations nulles nécessaires :

$$\mathsf{complete}(L) = L \cup \{t \times \beta_0(t) | (t \in \mathsf{T}_\Sigma) \wedge \neg(\exists \beta \in \mathsf{T}_{\mathsf{Bool}} \; (t \times \beta \in L))\}$$

Définition 3.7. *Soit* q *une requête sur* T_Σ. *Un automate* A décrit la requête q *si et seulement si* $\mathsf{complete}(\mathsf{langage}(A)) = \mathsf{langage}(\mathsf{q})$

Nous allons illustrer ces deux définitions par un exemple.

Exemple 3.2. *Soit* $\Sigma = \{f_2, a_0, b_0\}$. *Considérons la requête* q *sélectionnant l'ensemble des feuilles étiquetées par* a *dans les arbres de hauteur 2.*

Soit A *l'automate ainsi défini :*

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{1, 2\} \\ \mathsf{finaux}(A) & = & \{2\} \\ \mathsf{regles}(A) & = & \{(a, \mathsf{V}) \to 1, (b, \mathsf{F}) \to 1, (f, \mathsf{F})(1, 1) \to 2\} \end{array}$$

L'automate A *décrit la requête* q*, mais ne la décrit pas strictement. En effet,* A *ne reconnaît pas les annotations nulles des arbres de hauteur supérieure à 2. Pour décrire strictement la requête* q*, il faut également reconnaître toutes les annotations nulles, ce qui ne peut être fait que par un automate nettement plus complexe :*

$$\begin{array}{rcl}
\mathsf{etats}(A) & = & \{1,2,3,4,5\} \\
\mathsf{finaux}(A) & = & \{2,3,5\} \\
\mathsf{regles}(A) & = & \{(a,\mathsf{V}) \to 1, (b,\mathsf{F}) \to 1, (f,\mathsf{F})(1,1) \to 2 \\
 & & (a,\mathsf{F}) \to 3, (b,\mathsf{F}) \to 3, (f,\mathsf{F})(3,3) \to 4, \\
 & & (f,\mathsf{F})(4,3) \to 5, (f,\mathsf{F})(3,4) \to 5, (f,\mathsf{F})(4,4) \to 5, \\
 & & (f,\mathsf{F})(4,5) \to 5, (f,\mathsf{F})(5,4) \to 5, (f,\mathsf{F})(5,5) \to 5\}
\end{array}$$

Cet assouplissement de la définition de requête ne change rien à l'expressivité des automates pour décrire des requêtes :

Propriété 3.1. *L'ensemble des requêtes sur* T_Σ *décrites par des automates sur* $\Sigma \times \mathsf{Bool}$ *est l'ensemble des requêtes régulières.*

Preuve : Soit L *un langage régulier sur* $\mathsf{T}_{\Sigma\times\mathsf{Bool}}$. *Posons* $L_1 = \{t \times \beta_0(t) | \exists \beta\ t \times \beta \in L\}$, $L_2 = \mathsf{T}_{\Sigma\times\mathsf{faux}} - L_1$. *Si* L *est régulier, alors* L_1 *est régulier, donc* L_2 *aussi. Or* $\mathsf{complete}(L) = L \cup L_2$, *donc* $\mathsf{complete}(L)$ *est régulier aussi. Donc si* L *est un langage régulier, alors* $\mathsf{complete}(L)$ *est également régulier.*

Il en résulte que si A *décrit une requête* q *sur* T_Σ, *alors il existe un automate* A' *qui reconnaît le langage* $\mathsf{complete}(\mathsf{langage}(A))$ *et donc qui reconnaît strictement* q.

La réciproque est immédiate, car pour toute requête régulière q *sur* T_Σ, *il existe un automate* A *qui décrit strictement* q, *ce qui implique qu'il décrit* q. ◄

3.2 Les transducteurs de sélection de nœuds (TSN)

Nous avons vu qu'il était possible de définir des requêtes monadiques dans les arbres de T_Σ avec des automates d'arbres. Cependant ces objets ne sont pas entièrement adaptés à la manipulation de requêtes pour deux raisons essentielles :

- Les automates d'arbres sur $\Sigma \times \mathsf{Bool}$ ne décrivent en général pas de requêtes au sens où on l'a définit. En effet, un langage d'arbre sur $\Sigma \times \mathsf{Bool}$ peut contenir deux annotations inconsistantes du même arbre sur Σ. Il serait plus satisfaisant de contraindre nos objets pour qu'ils définissent nécessairement des requêtes.
- Ils ne sont pas opérationnels : un automate d'arbre sur $\Sigma \times \mathsf{Bool}$ ne permet pas directement d'appliquer la requête qu'il définit, c'est à dire de calculer $\mathsf{q}(t)$ pour t arbre quelconque de T_Σ.

Nous allons étudier les conditions sous lesquelles un automate d'arbres sur $\Sigma \times \mathsf{Bool}$ définit une requête, puis montrer comment calculer cette requête. Cela nous permettra d'introduire les objets que nous utiliserons par la suite pour représenter des requêtes : les transducteurs de sélection de nœuds.

3.2.1 Propriétés des automates décrivant des requêtes

Transductions associées à un automate d'arbres sur $\Sigma \times \mathsf{Bool}$

Pour discuter des propriétés des automates d'arbres décrivant des requêtes, nous allons nous intéresser à la *transduction associée à un automate d'arbre sur* $\Sigma \times \mathsf{Bool}$, c'est-à-dire à la transduction prenant en entrée des arbres de T_Σ et donnant en sortie des arbres de $\mathsf{T}_{\mathsf{Bool}}$ telle que la fusion d'une entrée et d'une sortie soit un arbre reconnu par l'automate. Soit A un automate d'arbres sur $\Sigma \times \mathsf{Bool}$. Notons $\mathsf{trans_A(t)}$ l'ensemble des images de t par la transduction associée à A. Formellement, elle se définit ainsi :

$$\mathsf{trans_A(t)} =_{\mathsf{def}} \{\beta | t \times \beta \in \mathsf{langage}(A)\}$$

Lorsque l'automate décrit strictement une requête, la transduction qui lui est associée définit exactement sa fonction d'annotation. Calculer cette transduction revient donc à calculer une requête décrite par un automate d'arbre. Nous verrons comment effectuer ce calcul dans la section 3.2.2.

Propriété des transductions associées aux automates d'arbres sur $\Sigma \times \mathsf{Bool}$

Nous allons caractériser les transductions associées à des automates d'arbres qui définissent des requêtes. Cette caractérisation nous permettra de définir une condition nécessaire et suffisante pour qu'un automate d'arbres sur $\Sigma \times \mathsf{Bool}$ définisse une requête.

Propriété 3.2. *Soit* q *une requête et* A *un automate d'arbre sur* $\Sigma \times \mathsf{Bool}$ *qui décrit cette requête.*

La transduction associée à A *est fonctionnelle :*

$$\forall t \in \mathsf{T}_\Sigma \ (|\mathsf{trans_A(t)}| \leq 1)$$

Preuve : Une requête q *est une fonction, c'est-à-dire qu'elle définit un et un seul sous-ensemble du domaine de chaque arbre. Donc* $\mathsf{langage}(q) = \{\mathsf{annot_q(t)} | t \in \mathsf{T}_\Sigma\}$ *ne peut contenir deux éléments* $t \times \beta$ *et* $t \times \beta'$ *avec* $\beta \neq \beta'$*. Donc la transduction associée à un automate à pas qui décrit* q *est fonctionnelle.* ◄

Par extension, nous qualifierons un automate d'arbres A sur $\Sigma \times \mathsf{Bool}$ de *fonctionnel* quand la transduction associée à A sera fonctionnelle.

Propriété 3.3. *Soit* A *un automate d'arbre fonctionnel sur* $\Sigma \times \mathsf{Bool}$*.*

Il existe une requête q *sur* T_Σ *telle que* A *décrit* q*.*

Preuve : Soit A *un automate d'arbre fonctionnel sur* $\Sigma \times \mathsf{Bool}$*.*

Soit q *la requête sur* T_Σ *définie par :*

$$\forall t \in \mathsf{T}_\Sigma \ \mathsf{q}(t) = \{\pi | \exists \beta \in \mathsf{T}_{\mathsf{Bool}} \ \exists t \times \beta \in \mathsf{langage}(A) \ (t \times \beta)(\pi) = \mathsf{vrai}\}$$

Montrons que $\mathsf{langage(q)} = \mathsf{complete}(\mathsf{langage}(A))$.

Soit t un arbre de T_Σ. *Montrons que* $\mathsf{annot_q(t)}$ *est un arbre de* $\mathsf{complete(langage}(A))$.

La définition de $\mathsf{q}(t)$ *est un ensemble de nœuds dépendant de l'existence de β tel que $t \times \beta$ soit un élément de* $\mathsf{langage}(A)$. *Deux cas se présentent :*

- *β existe. Comme A est fonctionnel, β est unique, donc* $\mathsf{q}(t) = \{\pi | (t \times \beta)(\pi) = \mathsf{vrai}\}$. *Donc* $\mathsf{annot_q(t)}$ *est un arbre de* $\mathsf{langage}(A)$.
- *β n'existe pas. Donc* $\mathsf{q}(t) = \emptyset$ *et $t \times \beta_0(t)$ est un arbre de* $\mathsf{complete(langage}(A))$. *Donc* $\mathsf{annot_q(t)}$ *est un arbre de* $\mathsf{complete(langage}(A))$.

Réciproquement, soit $t \times \beta$ un arbre de $\mathsf{complete(langage}(A))$. *Deux cas se présentent :*

- *$t \times \beta \in$* $\mathsf{langage}(A)$. *Dans ce cas, comme A est fonctionnel, il n'existe pas de β' tel que $t \times \beta'$ soit dans* $\mathsf{langage}(A)$, *donc* $\mathsf{q}(t) = \{\pi | (t \times \beta)(\pi) = \mathsf{vrai}\}$, *donc* $\mathsf{annot_q(t)} = t \times \beta$.
- *$t \times \beta \notin$* $\mathsf{langage}(A)$. *Dans ce cas, $\beta = \beta_0(t)$ et il n'existe pas de β' tel que $t \times \beta'$ soit dans* $\mathsf{langage}(A)$, *donc* $\mathsf{q}(t) = \emptyset$. *Donc* $\mathsf{annot_q(t)} = t \times \beta_0(t) = t \times \beta$.

Donc $\mathsf{langage(q)} = \mathsf{complete(langage}(A))$, *donc A décrit* q.

◀

Cela nous donne une condition nécessaire et suffisante pour qu'un automate d'arbres A sur $\Sigma \times \mathsf{Bool}$ décrive une requête. Nous allons manipuler des requêtes avec des automates d'arbres satisfaisant cette condition. Comme ces automates peuvent être vus comme des transducteurs et que c'est cet aspect que nous voulons mettre en valeur, nous allons appeler ces automates les *transducteurs de sélection de nœuds*.

Définition 3.8. *Un* transducteur de sélection de nœuds *sur Σ est un automate d'arbre fonctionnel sur* $\Sigma \times \mathsf{Bool}$.

Comme tout TSN représente une requête, on peut définir la fonction d'annotation d'un TSN comme la fonction d'annotation de la requête qu'il représente :

$$\mathsf{annot_A(t)} = \beta \Leftrightarrow t \times \beta \in \mathsf{complete(langage}(A))$$

3.2.2 Le calcul de requêtes avec les TSN

Nous avons défini la fonction d'annotation d'un TSN, nous allons maintenant montrer comment la calculer.

L'algorithme prend en entrée un TSN A sur Σ et un arbre t sur Σ. Il se décompose en deux phases :

- Une première phase ascendante dans laquelle on collectionne les états *potentiellement* associés à chaque nœud. Plus précisément, pour chaque sous-arbre t' de t, on calcule l'ensemble des évaluations possibles d'arbres annotés $t' \times \beta'$ et on l'associe au nœud correspondant à la racine de t' dans t. Le résultat de cette phase est un arbre sur $\Sigma \times 2^{\mathsf{etats}(A)}$ contenant les listes de ces états potentiels.
- Une deuxième phase descendante dans laquelle pour chaque nœud π, on sélectionne parmi les états sélectionnés dans la première phase ceux qui font partie d'un run réussi de A sur l'unique annotation β de t induite par A. Pour cela, on sélectionne uniquement les états finaux à la racine, puis on redescend en sélectionnant récursivement les états qui ont pu mener aux états sélectionnés (dans l'algorithme ci-après, les états sélectionnés sont ceux qui sont fournis en argument à la fonction auxiliaire aux). Les règles utilisées

pour redescendre nous donnent l'annotation β. Si deux règles de descente induisent deux annotations différentes du même nœud, cela veut dire que l'automate n'est pas fonctionnel.

Un exemple de déroulement de cet algorithme est fourni juste après sa description.

Calcul de requête, phase 1

Fonction $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})$

Entrée : Un TSN A sur Σ
Un arbre t sur Σ

Sortie : Un arbre t sur $\Sigma \times 2^{\mathsf{etats}(A)}$

Match t Avec

a :
  $S \leftarrow \{q | \exists b \in \mathsf{Bool}\ (a,b) \rightarrow q \in \mathsf{regles}(A)\}$
  Retour (a, S)

$f(t_1, t_2)$:
  $t'_1 \leftarrow \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_1})$
  $t'_2 \leftarrow \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_2})$
  Soit $(f_1, S_1) = t'_1(\mathsf{racine})$
  Soit $(f_2, S_2) = t'_2(\mathsf{racine})$
  $S \leftarrow \{q | \exists b \in \mathsf{Bool}\ \exists q_1 \in S_1\ \exists q_2 \in S_2\ (f,b)(q_1,q_2) \rightarrow q \in \mathsf{regles}(A)\}$
  Retour $(f, S)(t'_1, t'_2)$

Calcul de requête, phase 2

Fonction $\mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})$
Entrée : Un TSN A sur Σ
Un arbre t sur $\Sigma \times 2^{\mathsf{etats}}(A)$
Sortie : Un arbre t sur $\Sigma \times \mathsf{Bool}$

Fonction $\mathsf{aux}(t, S')$
Entrée : Un arbre t sur $\Sigma \times 2^{\mathsf{etats}}(A)$
Un ensemble d'états S
Sortie : Un arbre t sur $\Sigma \times \mathsf{Bool}$
Match t Avec
(a, S) :
$bSet \leftarrow \{b | \exists q \in S' \ ((a, b) \rightarrow q \in \mathsf{regles}(A))$
Si $(\mathsf{vrai} \in bSet) \wedge (\mathsf{faux} \in bSet)$
Alors **Erreur :** Entrée inconsistante (Automate non fonctionnel)
Sinon $b \leftarrow (\mathsf{vrai} \in bSet)$
Retour (a, b)
$(f, S)(t_1, t_2)$:
Soit $(f_1, S_1) = t_1(\mathsf{racine})$
Soit $(f_2, S_2) = t_2(\mathsf{racine})$
$S'_1 \leftarrow \emptyset$
$S'_2 \leftarrow \emptyset$
$bSet \leftarrow \emptyset$
PourTout $((f, b)(q_1, q_2) \rightarrow q) \in \mathsf{regles}(A) | q_1 \in S_1,\ q_2 \in S_2,\ q \in S'$
$S'_1 \leftarrow S'_1 \cup \{q_1\}$
$S'_2 \leftarrow S'_2 \cup \{q_2\}$
$bSet \leftarrow bSet \cup \{b\}$
$t'_1 \leftarrow \mathsf{aux}(t_1, S'_1)$
$t'_2 \leftarrow \mathsf{aux}(t_2, S'_2)$
Si $(\mathsf{vrai} \in bSet) \wedge (\mathsf{faux} \in bSet)$
Alors **Erreur :** Entrée inconsistante (Automate non fonctionnel)
Sinon $b \leftarrow (\mathsf{vrai} \in bSet)$
Retour $(f, b)(t'_1, t'_2)$

Soit $(f, S) = t(\mathsf{racine})$
$t' \leftarrow aux(t, S \cap \mathsf{finaux}(A))$
Retour t'

Exemple 3.3. *Nous allons illustrer l'algorithme décrit ci-dessus par un exemple. Soit* $A_{impaire}$, TSN *qui sélectionne les feuilles d'étiquette* a *et de profondeur impaire dans les arbres de* T_Σ, *avec* $\Sigma = \{f_2, a_0, b_0\}$. $A_{impaire}$ *est défini ainsi :*

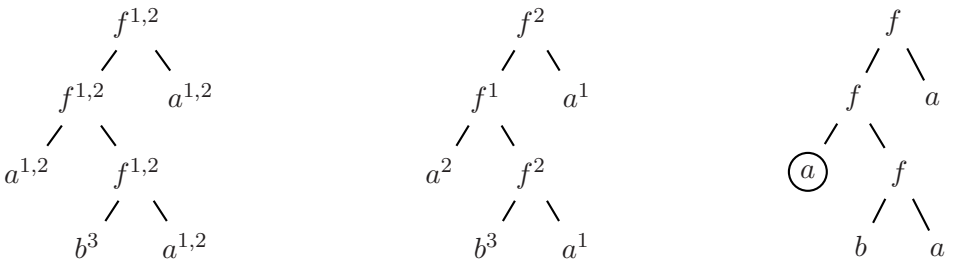


Figure 3.2: L'annotation d'un arbre binaire par le TSN $A_{impaire}$ (décrit dans l'exemple 3.3). A gauche, les états sélectionnés par la phase ascendante, au milieu, les états sélectionnés par la phase descendante, à droite, l'annotation résultante.

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{1,2,3\} \\ \mathsf{finaux}(A) & = & \{2,3\} \\ \mathsf{regles}(A) & = & \{(a,\mathsf{F}) \to 1,\ (a,\mathsf{V}) \to 2,\ (b,\mathsf{F}) \to 3 \\ & & (f,\mathsf{F})(1,1) \to 2, (f,\mathsf{F})(2,2) \to 1, \\ & & (f,\mathsf{F})(1,3) \to 2, (f,\mathsf{F})(2,3) \to 1, \\ & & (f,\mathsf{F})(3,1) \to 2, (f,\mathsf{F})(3,2) \to 1\} \end{array}$$

Nous allons appliquer ce TSN *sur un arbre binaire défini sur* Σ *en utilisant notre algorithme en deux phases. Les deux phases de calcul puis l'annotation d'un arbre par l'automate* A_{impair} *sont représentées Fig. 3.3.*

Lors de la phase ascendante, on applique les règles aux feuilles, puis progressivement à chacun de nœuds de l'arbre. Ici, les feuilles a*, puis les nœuds internes sont associées aux états* 1 *et* 2*. Intuitivement, cela correspond à l'idée qu'il n'est pas possible de savoir si une feuille est à une profondeur paire ou impaire avant d'être remonté jusqu'à la racine. La phase descendante commence à la racine. Parmi les états qui sont associés à la racine, seul* 2 *est final. Parmi les états qui sont associés aux enfants respectifs de la racine, seuls* 1 *et* 1 *ont pu mener à* 2 *pour la racine, grâce à la règle* $f(1,1) \to 2$*. On peut ainsi retrouver récursivement l'ensemble des états associés à chaque nœud dans un run réussi de* A *sur l'arbre annoté par* A*. On peut en déduire l'annotation correspondante.*

Théorème 3.1. *Soit* A *un* TSN *sur* Σ *et* t *un arbre de* T_Σ*. Le calcul de* $\mathsf{annot_A(t)}$ *se fait en* $O(|\mathsf{regles}(A)| \times |t|)$

Preuve : La première phase de l'algorithme que nous venons de fournir se fait en calculant récursivement pour chacun des nœuds de l'arbre, l'ensemble S *à partir des ensembles* S_1 *et* S_2*, selon cette formule :*

$$\{q | \exists b \in \mathsf{Bool}\ \exists q_1 \in S_1\ \exists q_2 \in S_2\ (f,b)(q_1,q_2) \to q \in \mathsf{regles}(A)\}$$

En utilisant une structure de donnée adaptée, ce calcul peut se faire en $O(|\mathsf{regles}(A)|)$*. La complexité en temps de la première phase est donc* $O(|\mathsf{regles}(A)| \times |t|)$*.*

La deuxième phase se fait en choisissant pour chaque paire de nœuds frère la paire d'état qui convient parmi les états choisis dans la première phase. Il faut tester l'existence d'une

règle pour chacune de ces paires, ce qui se fait en $O(|\mathsf{etats}(A)|^2)$. *La complexité en temps de la deuxième phase est donc* $O(|\mathsf{etats}(A)|^2 \times |t|)$.

On peut optimiser cet algorithme en mémorisant les règles qui mènent à chaque nœud dans la phase montante, au lieu de ne mémoriser que les résultats des évaluations. Cela nous permet de savoir directement quelle règle choisir dans la phase descendante, ce qui réduit la complexité en temps de l'ensemble des deux phases à $O(|\mathsf{regles}(A)| \times |t|)$. ◄

3.2.3 Le test d'appartenance à la classe des TSN

Dans le chapitre suivant nous mettrons en place des algorithmes d'inférence de TSN. Ce sont des algorithmes fonctionnant en travaillant à partir d'un TSN très spécifique, et en effectuant des généralisations successives. A chaque généralisation, nous allons avoir besoin de vérifier que l'automate sur lequel nous travaillons est un TSN, c'est-à-dire que la transduction qui lui est associée est fonctionnelle.

Théorème 3.2. *La fonctionnalité d'un automate d'arbre* A, *défini sur* $\Sigma \times \mathsf{Bool}$ *se vérifie en* $O|\mathsf{etats}(A)|.|\mathsf{regles}(A)|^2$.

Preuve : L'algorithme que nous allons montrer s'inspire de travaux sur la détection d'ambiguïté dans les automates de mots introduits dans [Coste and Fredouille, 2000].

Pour déterminer la fonctionnalité d'un automate d'arbre A *sur* $\Sigma \times \mathsf{Bool}$ A, *nous allons calculer pour toute paire d'états les relations suivantes :*

- $\mathsf{sim}_A(q, q')$, *vérifiée si et seulement si* q *et* q' *font partie de l'évaluation d'un même arbre annoté* $t \times \beta$.
- $\mathsf{drst}_A(q, q')$, *vérifiée si et seulement si* q *et* q' *font respectivement partie des évaluations de* $t \times \beta$ *et* $t \times \beta'$, *où* t *est un arbre quelconque sur* Σ, β *et* β' *deux annotations distinctes de* t.

Le lemme 3.1 montre en quoi on peut déduire la fonctionnalité d'un automate d'arbre A *sur* $\Sigma \times \mathsf{Bool}$ *du calcul de* drst *pour l'ensemble des paires d'états de* A.

Le lemme A.3 fournit une expression récursive de drst *et* sim *qui nous permet d'en déduire un algorithme de calcul de ces relations.*

Enfin, le lemme 3.3 montre que ce calcul sur toutes les paires d'états d'un automate d'arbre A *sur* $\Sigma \times \mathsf{Bool}$ *se fait en* $O|\mathsf{etats}(A)|.|\mathsf{regles}(A)|^2$.

En conséquence, le test de fonctionnalité d'un automate d'arbres sur $\Sigma \times \mathsf{Bool}$ *s'effectue en* $O|\mathsf{etats}(A)|.|\mathsf{regles}(A)|^2$ ◄

Lemme 3.1. *Soit* A *un automate d'arbre sur* $\Sigma \times \mathsf{Bool}$. *L'automate* A *est fonctionnel si et seulement si il n'existe pas de paire d'états finaux* q *et* q' *telle que* $\mathsf{drst}_A(q, q')$.

Preuve : L'existence de q *et* q' *finaux tels que* $\mathsf{drst}_A(q, q')$ *est équivalente à l'existence de* $t \times \beta$ *et* $t \times \beta'$ *acceptés par* A *avec* $\beta \neq \beta'$, *et donc équivalente à la non-fonctionnalité de l'automate.* ◄

Lemme 3.2. *Soit* A *un automate d'arbre émondé sur* $\Sigma \times \mathsf{Bool}$, q *et* q' *deux états de* A. *En omettant les quantifications existentielles pour plus de lisibilité, les équivalences suivantes sont vérifiées :*

$$\mathsf{sim}_A(q,q') \Leftrightarrow$$
$$\begin{array}{ll} & ((a,b) \to q \wedge (a,b) \to q') \\ \vee & ((f,b)(q_1,q_2) \to q \wedge (f,b)(q'_1,q'_2) \to q' \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)) \end{array}$$

$$\mathsf{drst}(q,q') \Leftrightarrow$$
$$\begin{array}{ll} & ((a,b) \to q \wedge (a,\neg b) \to q') \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_1,q'_1) \wedge \mathsf{drst}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{drst}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,\neg b)(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)) \end{array}$$

La preuve de ce lemme est donnée en annexe.

Nous allons utiliser ce lemme pour définir un algorithme de calcul de drst. Le principe de l'algorithme est d'affecter drst et sim pour les cas de base, ne dépendant pas d'une autre valeur de drst ou de sim, c'est-à-dire pour les cas correspondant aux règles initiales. A chaque affectation, on va effectuer l'ensemble des nouvelles affectations à effectuer récursivement, en cascade, selon les règles récursives exprimées dans la propriété A.3.

Test de fonctionnalité d'un automate sur $\Sigma \times$ Bool

Entrée : Un automate sur $\Sigma \times$ Bool
Sortie : vrai si l'automate est fonctionnel, faux sinon
Fonction affecte_sim$(q,q') =$
  sim$(q,q') \leftarrow$ vrai
  PourTout $((f,b)(q,q_2) \to q_0) \in$ regles(A)
    PourTout $((f,b')(q',q'_2) \to q'_0) \in$ regles(A)
      Si sim$(q_2,q'_2) \wedge b = b'$ Alors affecte_sim(q_0,q'_0)
      Si (sim$(q_2,q'_2) \wedge b \neq b') \vee$ drst(q_2,q'_2) Alors affecte_drst(q_0,q'_0)
  PourTout $((f,b)(q_1,q) \to q_0) \in$ regles(A)
    PourTout $((f,b')(q'_1,q') \to q'_0) \in$ regles(A)
      Si sim(q_1,q'_1) Alors affecte_sim(q_0,q'_0)
      Si (sim$(q_1,q'_1) \wedge b \neq b') \vee$ drst(q_1,q'_1) Alors affecte_drst(q_0,q'_0)

Fonction affecte_drst$(q,q') =$
  drst$(q,q') \leftarrow$ vrai
  PourTout $((f,b)(q,q_2) \to q_0) \in$ regles(A)
    PourTout $((f,b')(q',q'_2) \to q'_0) \in$ regles(A)
      Si sim$(q_2,q'_2) \vee$ drst(q_2,q'_2) Alors affecte_drst(q_0,q'_0)
  PourTout $((f,b)(q_1,q) \to q_0) \in$ regles(A)
    PourTout $((f,b')(q'_1,q') \to q'_0) \in$ regles(A)
      Si sim$(q_1,q'_1) \vee$ drst(q_1,q'_1) Alors affecte_drst(q_0,q'_0)

PourTout (q, q')
  drst$(q, q') \leftarrow$ faux
  sim$(q, q') \leftarrow$ faux
PourTout $((a, b) \rightarrow q) \in$ regles(A)
  PourTout $((a, b') \rightarrow q') \in$ regles(A)
    Si $b \neq b'$ Alors affecte_drst(q, q') Sinon affecte_sim(q, q')
PourTout $((f, b)(q_1, q_2) \rightarrow q) \in$ regles(A)
  PourTout $((f, b')(q_1, q_2) \rightarrow q') \in$ regles(A)
    Si $b \neq b'$ Alors affecte_drst(q, q') Sinon affecte_sim(q, q')

PourTout $q \in$ finaux(A)
  PourTout $q' \in$ finaux(A)
    Si drst(q, q') Alors Retour faux
Retour vrai

Lemme 3.3. *Le calcul de* sim *et de* drst *pour l'ensemble des paires d'états dans un automate d'arbre sur* $\Sigma \times$ Bool*, se fait en* $O|\text{etats}(A)|.|\text{regles}(A)|^2$.

Preuve : On peut majorer le nombre d'appels récursifs à affecte_drst *par* $|\text{etats}(A)|^2$*, et de même pour* affecte_sim*. Hormis les appels récursifs, l'exécution de chacune de ces fonctions est en* $O(|\text{regles}(A)|^2)$*. Donc en première approximation la complexité en temps du test de fonctionnalité est en* $O(|\text{etats}(A)|^2 * |\text{regles}(A)|^2)$.

Nous allons établir une borne supérieure de la complexité de façon un peu plus fine. Fixons q' *et considérons l'ensemble des appels à* affecte_drst(q, q')*. Pour chaque* q*,* affecte_drst(q, q') *n'est exécuté qu'une fois, et donc la boucle sur les règles* $((f, b)(q, q_2) \rightarrow q_0)$ *ne peut être exécuté qu'une fois par règle. Le nombre d'exécutions de cette boucle pour* q' *fixé est donc majoré par* $|\text{regles}(A)|$*, donc le nombre total d'exécution de cette boucle est majoré par* $|\text{etats}(A)|.|\text{regles}(A)|$*. Symétriquement, le nombre d'exécutions de la boucle suivante est majoré de la même manière. Chacune de ces boucles contient elle-même une boucle sur un ensemble de règles, contenant un test effectué en temps constant. On peut faire le même raisonnement avec* affecte_sim(q, q').

La complexité en temps totale de l'ensemble des appels à affecte_drst*, et donc de l'algorithme dans son ensemble, est donc en* $O(|\text{etats}(A)|.|\text{regles}(A)|^2)$.

◄

3.2.4 L'expressivité des TSN

Les TSN expriment exactement les requêtes que nous avons définies comme *régulières*, c'est-à-dire les requêtes dont les langages associés, selon notre définition, sont réguliers. Nous allons maintenant comparer cette expressivité à celle d'un formalisme classique de définition des requêtes monadiques : la logique monadique du second ordre (MSO) sur les arbres. La MSO

est une référence en terme d'expressivité pour les requêtes dans les arbres : c'est le formalisme qu'utilise Lixto [Gottlob and Koch, 2002a, Baumgartner et al., 2001], et de nombreux formalismes de définition de requêtes dans les arbres, fondés notamment sur les automates d'arbres [Neven and Bussche, 2002, Neven and Schwentick, 2002, Berlea and Seidl, 2004], lui sont équivalent en expressivité.

Nous allons commencer par introduire la définition de requêtes monadiques dans les arbres en MSO, puis montrerons que les TSN et la MSO ont la même expressivité.

Requêtes et MSO

Le principe de la définition de requêtes dans les arbres en MSO est d'utiliser des formule MSO à valeur dans le domaine des arbres pour caractériser les nœuds sélectionnés par une requête. Pour plus de détail sur le sujet, consulter [Thomas, 1997, Gottlob and Koch, 2002b]. Nous allons commencer par définir la MSO dans les arbres, puis nous allons montrer comment définir une requête en MSO.

Les formules MSO sont définies à partir d'une *signature*, c'est-à-dire un ensemble de prédicats élémentaires, puis sont construites selon la syntaxe suivante :

$$\phi ::= B_n(x_1, \ldots, x_n) \mid p(x) \mid \phi \wedge \phi' \mid \neg\phi \mid \exists x\phi \mid \exists p\phi$$

où B_n est un des prédicats élémentaires d'arité n de la signature de la MSO utilisée, x une variable du premier ordre et p une variable du second ordre.

Une MSO sur les arbre est une MSO dont les variables sont à valeurs dans le domaine des arbres. Il faut donc définir une signature adaptée pour décrire propriétés et relations des éléments du domaine des arbres. La signature classiquement utilisée pour décrire les structures d'arbres binaires sur Σ est la suivante :

$$\mathsf{sig}^{\mathsf{b}} = \{\mathsf{fils}_1, \mathsf{fils}_2, \mathsf{racine}, \mathsf{feuille}\} \cup \{\mathsf{etiq}_a \mid a \in \Sigma\}$$

La sémantique des prédicats est la suivante : $\mathsf{fils}_1(x, y)$ signifie que y est le fils gauche de x, $\mathsf{fils}_2(x, y)$ signifie que y est le fils droit de x, $\mathsf{racine}(x)$ que x est la racine de l'arbre, $\mathsf{feuille}(x)$ que x est une feuille, $\mathsf{etiq}_a(x)$ que x est étiqueté par a. Nous appellerons $\mathsf{MSO}^{\mathsf{b}}$ la MSO définie sur la signature $\mathsf{sig}^{\mathsf{b}}$.

On peut définir une requête monadique dans les arbres de T_Σ avec une formule $\mathsf{MSO}^{\mathsf{b}}$ à une variable libre, les nœuds sélectionnés seront ceux qui satisfont cette formule :

$$\mathsf{qu}^{\mathsf{b}}_{\phi(\mathsf{x})}(\mathsf{t}) = \{\sigma(x) | t, \sigma \models \mathsf{MSO}^{\mathsf{b}}\phi\}$$

La $\mathsf{MSO}^{\mathsf{b}}$ est un moyen de définir des requêtes de façon simple et intuitive. Par exemple, la requête "sélectionner toutes les feuilles d'étiquette a" s'écrira simplement $\phi(x) = \mathsf{etiq}_a(x) \wedge \mathsf{feuille}(x)$. La requête "sélectionner toute les feuilles dont le fils gauche est étiqueté par a" s'écrira $\phi(x) = \exists y\ (\mathsf{etiq}_a(y) \wedge \mathsf{fils}_1(y, x)$.

Expressivité comparée des TSN et de la MSO

Théorème 3.3. *Les* TSN *et la* $\mathsf{MSO}^{\mathsf{b}}$ *expriment les même requêtes monadiques dans les arbres binaires.*

Preuve : Cette preuve se fonde sur les résultats d'équivalence en expressivité entre la MSO *et les automates d'arbres décrits dans [Thatcher and Wright, 1968] et détaillés dans [Comon et al., 1997].*

Le principe de cette preuve est le suivant : le théorème de Thatcher et Wright induit une équivalence pour les langages d'arbres sur $\Sigma \times \mathsf{Bool}$ *entre régularité et définissabilité en* $\mathsf{MSO}^{\mathsf{b}}$*. Cependant, la correspondance entre automates d'arbre et* $\mathsf{MSO}^{\mathsf{b}}$ *induite par cette équivalence n'est pas exactement celle que nous cherchons à mettre en évidence entre les* TSN *et les requêtes* $\mathsf{MSO}^{\mathsf{b}}$*. En effet, les* TSN *sélectionnent les noeuds d'un arbre donné simultanément, c'est-à-dire qu'il reconnaissent uniquement les annotations complètes de cet arbre. La formule* $\mathsf{MSO}^{\mathsf{b}}$ définissant *ce langage est une formule* $\mathsf{MSO}^{\mathsf{b}}$ *à une variable libre du second ordre, qui, pour chaque arbre* t*, est satisfaite pour l'unique ensemble contenant exactement les noeuds sélectionnés par la requête. Les requêtes* $\mathsf{MSO}^{\mathsf{b}}$*, au contraire, sont des formules à une variable libre du premier ordre satisfaites dans chaque arbre pour chaque noeud sélectionné par la requête. Nous allons montrer l'équivalence en expressivité entre* TSN *et* $\mathsf{MSO}^{\mathsf{b}}$ *en montrant qu'il est possible de convertir une formule* $\mathsf{MSO}^{\mathsf{b}}$ *sélectionnant simultanément tous les noeuds de chaque arbre en une formule* $\mathsf{MSO}^{\mathsf{b}}$ *les sélectionnant un par un, et réciproquement.*

Commençons par rappeler la notion de définissabilité utilisée dans le théorème de Thatcher et Wright placé dans notre contexte.

Soit t *un arbre de* T_{Σ}*,* p *une variable ensembliste à valeur dans* $\mathsf{dom}(t)$ *et* σ *une affectation de* p *dans* $\mathsf{dom}()$*. Appelons* annotation de t par σ*, noté* $\mathsf{annot}_{\sigma}(\mathsf{t})$*, l'arbre de* $\mathsf{T}_{\Sigma \times \mathsf{Bool}}$ *défini par :*

– $\mathsf{dom}(\mathsf{annot}_{\sigma}(\mathsf{t})) = \mathsf{dom}(t)$

– *Pour tout* π*,* $\mathsf{annot}_{\sigma}(\mathsf{t})(\pi) = (t(\pi), b)$*, avec* $b = \mathsf{vrai}$ *si* $\pi \in \sigma(p)$*, et* $b = \mathsf{faux}$ *sinon.*

Soit $\phi(p)$ *une formule de* $\mathsf{MSO}^{\mathsf{b}}$ *à une variable libre. Appelons* langage défini par ϕ*, noté* $L(\phi)$*, le langage sur* $\Sigma \times \mathsf{Bool}$ *formé par l'ensemble des annotations des arbres de* T_{Σ} *par des affectations de* p *satisfaisant* ϕ *:*

$$L(\phi) = \{\mathsf{annot}_{\sigma}(\mathsf{t}) | t, \sigma \models \phi\}$$

On dit qu'un langage L *sur* $\Sigma \times \mathsf{Bool}$ *est* définissable *en* $\mathsf{MSO}^{\mathsf{b}}$ *si et seulement si il existe une formule à une variable libre* ϕ *qui définit* L*.*

Le théorème de Thatcher et Wright ([Thatcher and Wright, 1968]), restreint aux formules à une variable ensembliste libre, nous dit que les langages définissables sur $\Sigma \times \mathsf{Bool}$ *sont exactement les langages réguliers sur* $\Sigma \times \mathsf{Bool}$*.*

Il en résulte que pour tout langage régulier L *sur* $\Sigma \times \mathsf{Bool}$*, il existe une formule* $\mathsf{MSO}^{\mathsf{b}}$ $\phi(p)$ *telle que* L *est le langage défini par* $\phi(p)$*. En particulier, dans le cas où* L *décrit strictement une requête* q*, pour tout arbre* t*,* $\phi(p)$ *est satisfaite par l'unique affectation* σ *telle que* $\sigma(p)$ *soit l'ensemble des nœuds de* t *sélectionnés par la requête* q*. Donc il existe une formule* $\mathsf{MSO}^{\mathsf{b}}$

qui définit la requête q*, que l'on peut écrire* $\psi(x) = \exists p, p(x) \wedge \phi(p)$*. Donc pour toute requête définie par un* TSN*, il existe une formule* $\mathsf{MSO}^{\mathsf{b}}$ *qui la définit.*

Réciproquement, soit $\psi(x)$ *une formule de* $\mathsf{MSO}^{\mathsf{b}}$ *définissant une requête* q*. Soit* $\phi(p) = (p(x) \Leftrightarrow \psi(x))$*. Pour un arbre* t *donné, la formule* $\phi(p)$ *n'est satisfaite que si* p *contient exactement l'ensemble des noeuds satisfaisant* ψ*, elle est donc satisfaite pour l'unique affectation* σ *de* t *telle que* $\sigma(p)$ *soit exactement l'ensemble des nœuds sélectionnés par* q*. Le langage défini par* ϕ *est donc exactement le langage de la requête* q*. Donc le langage de la requête* q *est régulier, donc il existe un* TSN *qui décrit la requête* q*. Donc pour toute requête définie par une formule* $\mathsf{MSO}^{\mathsf{b}}$*, il existe un* TSN *qui la décrit.*

◀

3.3 Les TSN dans les arbres à arité arbitraire

Nous allons utiliser le formalisme des automates à pas, défini dans le chapitre 2, pour définir des requêtes dans les arbres à arité arbitraire avec des TSN.

Jusque là, nous avons défini les TSN dans des langages d'arbres binaires. Nous allons utiliser ces résultats pour définir des TSN dans les arbres de construction, et, grâce aux correspondances entre arbres de construction et arbres à arité arbitraire définies dans le chapitre 2, les mêmes TSN définirons des requêtes dans les arbres à arité arbitraire.

3.3.1 TSN et automates à pas

Notre but est de définir des requêtes dans les documents semi-structurés, qui peuvent être modélisés par des arbres à arité arbitraire, comme nous le verrons dans le chapitre 5. Jusque là, nous n'avons parlé que de TSN dans les arbres binaires. Nous allons utiliser le formalisme des automates à pas, décrit dans le chapitre 2, pour décrire des requêtes dans les arbres à arité arbitraire.

Les TSN dans les arbres de construction

Les arbres de construction étant des arbres binaires, l'utilisation de TSN pour décrire des requêtes dans les arbres de construction semble directe.

Cependant, en utilisant les définitions que nous avons vues, un TSN sur les arbres de construction serait un automate d'arbre sur $\Sigma_{@} \times \mathsf{Bool}$, alphabet contenant deux symboles binaires $(@, \mathsf{V})$ et $(@, \mathsf{F})$. Cela pose problème, pour deux raisons :

- Ce TSN n'est pas un automate à pas, selon la définition que nous avons donnée.
- On peut définir des requêtes qui sélectionnent des nœuds internes, dont les éléments correspondants dans les arbres à arité arbitraire correspondants sont des arêtes. Nous verrons par la suite que nous ne voulons pas de ce type de requêtes.

Pour résoudre ce problème, nous allons simplement limiter les requêtes dans les arbres de construction aux requêtes ne sélectionnant que des feuilles. Nous les appellerons *requêtes monadiques sur les feuilles.*

Pour décrire les requêtes monadiques sur les feuilles dans les arbres de construction, nous allons transposer le formalisme des automates à pas dans les TSN.

Définition 3.9. *Un* TSN *à pas sur* Σ *est un automate à pas fonctionnel et déterministe sur* $\Sigma \times \mathsf{Bool}$.

Propriété 3.4. *Les* TSN *à pas sur* Σ *décrivent exactement les requêtes monadiques régulières sur les feuilles dans les arbres de construction de* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$.

Preuve : Les automates binaires sur $\Sigma_{@} \times \mathsf{Bool}$ *décrivent exactement les requêtes régulières sur* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$. *En assimilant le symbole* $(@, \mathsf{F})$ *à* $@$, *les langages réguliers d'arbres de constructions sur* $\Sigma \times \mathsf{Bool}$ *sont exactement les langages d'arbres binaires sur* $\Sigma_{@} \times \mathsf{Bool}$ *ne contenant pas le symbole* $(@, \mathsf{V})$. *Donc les requêtes régulières décrites par les* TSN *à pas sont exactement celles qui ne sélectionnent que des feuilles.* ◄

Les TSN dans les arbres à arité arbitraire

Il y a entre les requêtes dans les arbres de construction et les requêtes dans les arbres à arité arbitraire une notion de correspondance qui est l'extension naturelle des correspondances décrites dans le chapitre 2 : deux requêtes correspondent si elles extraient les éléments correspondants des arbres correspondants.

Définition 3.10. *Soit* q *une requête sur* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$. *La requête correspondante* $\mathsf{c}_{\mathsf{requete}}(\mathsf{q})$ *est une requête sur* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ *définie ainsi :*

$$\mathsf{c}_{\mathsf{requete}}(\mathsf{q})(t) = \mathsf{c}_{\mathsf{dom}}(\mathsf{c}_{\mathsf{arbre}}{}^{-1}(t))(\mathsf{q}(\mathsf{c}_{\mathsf{arbre}}{}^{-1}(t)))$$

Propriété 3.5. *Soit* q *une requête sur* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$. *La requête* $\mathsf{c}_{\mathsf{requete}}(\mathsf{q})$ *est une requête monadique sur les feuilles de* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ *si et seulement si la requête* q *est une requête monadique sur* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$.

Preuve : Les requêtes correspondantes définissent des éléments correspondants. Or les feuilles des arbres de construction correspondent aux nœuds des arbres à arité arbitraire (voir propriété 2.2.2). ◄

Nous avons vu que les TSN à pas peuvent définir des requêtes monadiques sur les feuilles des arbres de construction. Comme les automates à pas ont la capacité de traiter à la fois des arbres de constructions et des arbres à arité arbitraire, les même TSN peuvent en même temps définir des requêtes dans les arbres à arité arbitraire. En fait, le couple de requêtes définit par un TSN est un couple de requête correspondantes :

Propriété 3.6. *Un automate à pas A décrit une requête* q *sur* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *si et seulement si il décrit la requête* $\mathsf{c}_{\mathsf{requete}}(\mathsf{q})$ *sur* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$.

Preuve : Cette propriété se déduit directement de la propriété 2.7 et de la proposition suivante, pour q *requête sur* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *:*

$$\mathsf{langage}(\mathsf{c}_{\mathsf{requete}}(\mathsf{q})) = \mathsf{c}_{\mathsf{arbre}}(\mathsf{langage}(\mathsf{q}))$$

Montrons cette propriété.

Soit t un arbre de $\mathsf{T}^{\mathsf{a}}_{\Gamma}$. *Montrons que* $\mathsf{annot}^{\mathsf{a}}_{\mathsf{c_{requete}(q)}}(\mathsf{c_{arbre}(t)}) = \mathsf{c_{arbre}}(\mathsf{annot}^{\mathsf{s}}_{\mathsf{q}}(\mathsf{t}))$.

Ces deux arbres ont le même domaine, celui de $\mathsf{c_{arbre}}(t)$. *Soit* π' *une feuille de* $\mathsf{c_{arbre}}(t)$. *Posons* $\pi = \mathsf{c_{dom}}^{-1}(\pi')$.

On a $\mathsf{c_{arbre}}(\mathsf{annot}^{\mathsf{s}}_{\mathsf{q}}(\mathsf{t}))(\pi') = \mathsf{annot}^{\mathsf{s}}_{\mathsf{q}}(\mathsf{t})(\pi)$ *et on peut déduire de la définition de* $\mathsf{c_{requete}}$ *et de celle de* $\mathsf{annot}^{\mathsf{a}}()$ *que* $\mathsf{annot}^{\mathsf{a}}_{\mathsf{c_{requete}(q)}}(\mathsf{c_{arbre}(t)})(\mathsf{c_{dom}}(\pi)) = \mathsf{annot}^{\mathsf{s}}_{\mathsf{q}}(\mathsf{t})(\pi)$. *Montrons que Donc* $\mathsf{annot}^{\mathsf{a}}_{\mathsf{c_{requete}(q)}}(\mathsf{c_{arbre}(t)})(\pi') = \mathsf{c_{arbre}}(\mathsf{annot}^{\mathsf{s}}_{\mathsf{q}}(\mathsf{t}))(\pi')$.

Donc $\mathsf{annot}^{\mathsf{a}}_{\mathsf{c_{requete}(q)}}(\mathsf{c_{arbre}(t)})(\pi') = \mathsf{c_{arbre}}(\mathsf{annot}^{\mathsf{s}}_{\mathsf{q}}(\mathsf{t}))(\pi')$. *Donc* $\mathsf{langage}^{\mathsf{a}}(\mathsf{c_{requete}(q)})$ *et* $\mathsf{c_{arbre}}(\mathsf{langage}^{\mathsf{s}}(\mathsf{q}))$ *ont même domaine et même valeurs sur leurs feuilles, donc ils sont égaux.* ◀

En utilisant un TSN à pas pour définir une requête sur les feuilles des arbres de $\mathsf{T}^{\mathsf{a}}_{\Gamma}$, on définit simultanément une requête sur les nœuds des arbres correspondants de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$. C'est ainsi que les TSN à pas nous permettent d'exprimer des requêtes dans les arbres à arité arbitraire. La partie suivante est consacrée à l'expressivité des TSN pour décrire ces requêtes.

3.3.2 Expressivité des TSN à pas

Nous avons vu en 3.2.4 que les TSN dans les arbres binaires avaient la même expressivité que la MSO définie sur la signature $\mathsf{sig}^{\mathsf{b}}$ pour définir des requêtes monadiques.

Dans les arbres de construction, nous avons vu que les TSN à pas peuvent exprimer l'ensemble des requêtes régulières ne sélectionnant que des feuilles. Il en résulte que les TSN à pas peuvent exprimer l'ensemble des requêtes MSO définies sur la signature $\mathsf{sig}^{\mathsf{b}}$ ne sélectionnant que des feuilles.

Étudions maintenant le cas des arbres à arité arbitraire. En utilisant les correspondances entre requêtes que nous venons de définir, nous en arrivons à la conclusion suivante : les TSN à pas expriment dans les arbres à arité arbitraire les requêtes correspondant au requêtes MSO dans les arbres de construction ne sélectionnant que des feuilles.

Pour exprimer leur expressivité dans une MSO définie directement dans les arbres à arité arbitraire, nous allons transcrire les formules MSO dans les arbres de construction en formules MSO dans les arbres à arité arbitraire. Le problème, c'est que les formules MSO dans les arbres de construction sont définies sur une signature, $\mathsf{sig}^{\mathsf{b}}$, qui est définie sur l'ensemble des nœuds des arbres de construction. La transcription de ces formules dans les arbres à arité arbitraire sera donc nécessairement définie sur une signature manipulant les éléments correspondant à ces nœuds dans les arbres à arité arbitraire, c'est-à-dire les éléments du domaine étendu : les nœuds et les arêtes. Voici celle que nous avons choisie :

$$\mathsf{sig}^{\mathsf{s}} = \{\mathsf{premiere\_arete}, \mathsf{arete\_suivante}, \mathsf{derniere\_arete}, \mathsf{cible}, \mathsf{racine}, \mathsf{feuille}\} \cup \{\mathsf{etiq}_a \mid a \in \Sigma\}$$

La sémantique de ces prédicats est la suivante : $\mathsf{premiere\_arete}(x, y)$ signifie que y est l'arête reliant le nœud x à son premier fils. $\mathsf{arete\_suivante}(x, y)$ signifie que x et y sont des arêtes reliant un même nœud à deux nœuds consécutifs. $\mathsf{derniere\_arete}(x, y)$ signifie que y est l'arête reliant x et son dernier fils. $\mathsf{cible}(x, y)$ signifie que y est la cible de l'arête x, c'est-à-dire que l'arête x relie le père de y et y. Enfin, les prédicats unaires ont leur signification naturelle.

Nous appellerons $\mathsf{MSO}^{\mathsf{s}}$ la MSO définie sur cette signature, et nous appellerons $\mathsf{MSO}^{\mathsf{a}}$ la $\mathsf{MSO}^{\mathsf{b}}$ sur l'alphabet de construction $\Sigma_{@}$.

Nous voulons maintenant montrer que les requêtes définies par des formules exprimées en $\mathsf{MSO}^{\mathsf{s}}$ correspondent exactement aux requêtes sur les feuilles définies par des formules exprimées en $\mathsf{MSO}^{\mathsf{a}}$, au sens de la correspondance entre requêtes telle que nous l'avons définie.

Dans les arbres de construction comme dans les arbres à arité arbitraire, nous imposons une restriction sur les requêtes définies par des formules MSO qui va nous poser problèmes pour établir cette équivalence : d'un coté on se limite aux formules à une variable libre ne satisfaisant que des feuilles alors que les variables sont à valeur dans l'ensemble des nœuds, et de l'autre on se limite aux formules ne satisfaisant que des nœuds alors que les variables sont à valeur dans le domaine étendu de l'arbre, c'est-à-dire l'ensemble des nœuds et des feuilles.

Pour les besoins de la démonstration, nous allons introduire la notion de *requête étendue* dans les arbres à arité arbitraire, c'est-à-dire de requêtes sélectionnant des éléments quelconques du domaine étendu. Nous allons montrer que les requêtes étendues définies en $\mathsf{MSO}^{\mathsf{s}}$ correspondent exactement aux requêtes définies en $\mathsf{MSO}^{\mathsf{a}}$. Il suffira alors de faire une simple restriction pour en déduire que les requêtes définies en $\mathsf{MSO}^{\mathsf{s}}$ correspondent exactement aux requêtes sur les feuilles définies en $\mathsf{MSO}^{\mathsf{a}}$. Notons enfin que nous étendons ici la définition de la fonction $\mathsf{c}_{\mathsf{requete}}$ aux requêtes étendues.

Lemme 3.4. *La fonction de correspondance de requête* $\mathsf{c}_{\mathsf{requete}}$ *est une bijection entre l'ensemble des requêtes étendues exprimée en* $\mathsf{MSO}^{\mathsf{s}}$ *et l'ensemble des requêtes exprimée en* $\mathsf{MSO}^{\mathsf{a}}$*.*

Preuve : La preuve se fait par un encodage des formules $\mathsf{MSO}^{\mathsf{a}}$ *en formule* $\mathsf{MSO}^{\mathsf{s}}$*, et réciproquement. Elle est détaillée en annexe.* ◄

Théorème 3.4. *Les* TSN *et la* $\mathsf{MSO}^{\mathsf{s}}$ *expriment les même requêtes monadiques dans les arbres à arité arbitraire.*

Preuve : Les TSN *à pas et la* $\mathsf{MSO}^{\mathsf{a}}$ *expriment les même requêtes sur les feuilles dans les arbres de construction (voir le théorème 3.3 et la propriété 3.4).*

En utilisant la propriété 3.6 et le lemme 3.4, que les TSN *et la* $\mathsf{MSO}^{\mathsf{s}}$ *expriment les mêmes requêtes monadiques dans les arbres à arité arbitraire.* ◄

3.4 Les TSN dans les arbres élagués

Nous allons présenter ici une extension des TSN, les Transducteurs de Selection de Nœuds élagueurs (TSNe), qui permet de définir des requêtes dans des arbres élagués et de les appliquer dans des arbres non élagués.

L'ensemble des idées et des résultats exprimés ici vont l'être dans le cadre des requêtes dans les arbres binaires. Cependant, leur transposition dans le cadre des requêtes dans les arbres à arité arbitraire se fait de façon similaire à celle des TSN, telle qu'elle est décrite dans la section 3.3.

Nous allons commencer par présenter l'élagage des arbres de manière informelle, en montrant en quoi il peut permettre de grandement simplifier la définition de requête par des TSN

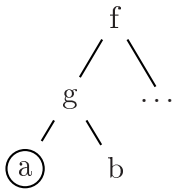
dans le cas déterministe. Puis, nous entreprendrons une introduction formelle de l'élagage et des TSNe.

3.4.1 Pourquoi élaguer ?

C'est avant tout lorsque nous parlerons de l'apprentissage de requêtes que l'élagage prendra tout son intérêt. Nous allons cependant donner ici une première description des avantages à utiliser des TSN capables de gérer l'élagage.

Un des problème des TSN lors de leur utilisation pratique est la nécessité de modéliser tout l'arbre pour définir une requête qui n'en concerne qu'une petite partie. Dans la grande majorité des cas qui se présentent dans la pratique, seule une petite partie de chaque arbre est intéressante pour définir la requête. Nous allons voir ici qu'on peut résoudre le problème simplement en utilisant un "état puits". Nous appellerons "état puits" un état qui reconnaît n'importe quel arbre et qui peut donc être associé aux sous-arbres qui ne nous intéressent pas. Par la suite, nous allons être amenés à manipuler exclusivement des TSN déterministes. Ceux-ci ont la même expressivité que les TSN non déterministes, mais dans certains cas, ils peuvent être bien plus complexes que leurs équivalents non déterministes. En particulier, la gestion d'un "état puits" s'adapte mal aux automates déterministes. Nous allons préciser ce problème et présenter une solution : les TSNe.

Prenons un exemple. Considérons l'alphabet $\Sigma = \{f_2, g_2, a_0, b_0, c_0, d_0\}$. Supposons qu'on veuille sélectionner la feuille a dans un arbre de T_Σ présentant cette structure :



Pour pouvoir décrire cette requête, nous allons construire un "état puits" qui évalue n'importe quel arbre de T_Σ n'ayant aucun élément sélectionné. Le TSN obtenu aura d'un côté des règles décrivant la structure des parties significatives de l'arbre pour la requête, et de l'autre des règles décrivant l'état puits (0) :

$$\begin{aligned}
\mathsf{etats}(A) &= \{1,2,3,4,0\} \\
\mathsf{finaux}(A) &= \{4\} \\
\mathsf{regles}(A) &= \{(a,\mathsf{V}) \to 1, (b,\mathsf{F}) \to 2, (g,\mathsf{F})(1,2) \to 3, (f,\mathsf{F})(3,0) \to 4, \\
&\quad (a,\mathsf{F}) \to 0, (b,\mathsf{F}) \to 0, (c,\mathsf{F}) \to 0, (d,\mathsf{F}) \to 0, \\
&\quad (f,\mathsf{F})(0,0) \to 0, (g,\mathsf{F})(0,0) \to 0\}
\end{aligned}$$

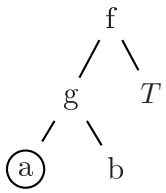
C'est lorsqu'on veut définir cette requête avec un TSN déterministe que les choses se compliquent. Le problème vient du b, qui ne peut pas être à la fois dans l'état 2 et dans l'état puits. Il faut donc exclure b de l'état puits et le traiter indépendamment, au prix d'un grand nombre de règles spécifiques supplémentaires :

$$\begin{array}{rcl}
\mathsf{etats}(A) & = & \{1,2,3,4,0\} \\
\mathsf{finaux}(A) & = & \{4\} \\
\mathsf{regles}(A) & = & \{(a,\mathsf{V}) \to 1, (b,\mathsf{F}) \to 2, (g,\mathsf{F})(1,2) \to 3, (f,\mathsf{F})(3,0) \to 4, \\
 & & (a,\mathsf{F}) \to 0, (c,\mathsf{F}) \to 0, (d,\mathsf{F}) \to 0, \\
 & & (f,\mathsf{F})(0,0) \to 0, (g,\mathsf{F})(0,0) \to 0 \\
 & & (f,\mathsf{F})(0,2) \to 0, (g,\mathsf{F})(0,2) \to 0 \\
 & & (f,\mathsf{F})(2,0) \to 0, (g,\mathsf{F})(2,0) \to 0\}
\end{array}$$

Ce problème est un problème majeur dans la définition de requêtes par des automates déterministes ascendant, qui peut s'exprimer ainsi : Étant donné un sous-arbre potentiellement significatif pour la requête que l'on décrit, l'est-il réellement ou fait-il partie d'une zone non significative ? Intuitivement, ce problème met en évidence l'aspect non-déterministe des requêtes définies par un processus ascendant dans les arbres.

La solution que nous allons apporter est très simple dans le principe. Nous allons ajouter à l'alphabet sur lequel les TSN sont définis un caractère spécial, T, qui a pour particularité de pouvoir remplacer n'importe quel arbre. Nous appellerons Transducteurs de Sélection de Nœuds élagueurs les TSN ainsi enrichis.

Ainsi, le problème précédent se ramènera à la définition d'un TSNe reconnaissant l'unique arbre :



Ce TSNe déterministe est défini par :

$$\begin{array}{rcl}
\mathsf{etats}(A) & = & \{1,2,3,4,T\} \\
\mathsf{finaux}(A) & = & \{4\} \\
\mathsf{regles}(A) & = & \{(a,\mathsf{V}) \to 1, (b,\mathsf{F}) \to 2, (g,\mathsf{F})(1,2) \to 3, T \to T, (f,\mathsf{F})(3,T) \to 4\}
\end{array}$$

Il est clair que le problème soulevé précédemment ne semble pas résolu par cette nouvelle définition. Le non-déterminisme évoqué est toujours présent implicitement dans ce TSNe, à travers le choix entre le remplacement ou non d'un sous-arbre par T. C'est le principe même des TSNe déterministes : ils sont déterministes dans leur définition, c'est-à-dire en temps qu'automates déterministes sur l'alphabet $\Sigma \times \mathsf{Bool} \cup T$, ce qui va nous permettre de les apprendre comme des TSN déterministes (nous verrons cela en détail dans le chapitre suivant). Mais ils sont non-déterministes dans leur utilisation, quand il s'agit de calculer des requêtes sur des arbres définis sur Σ.

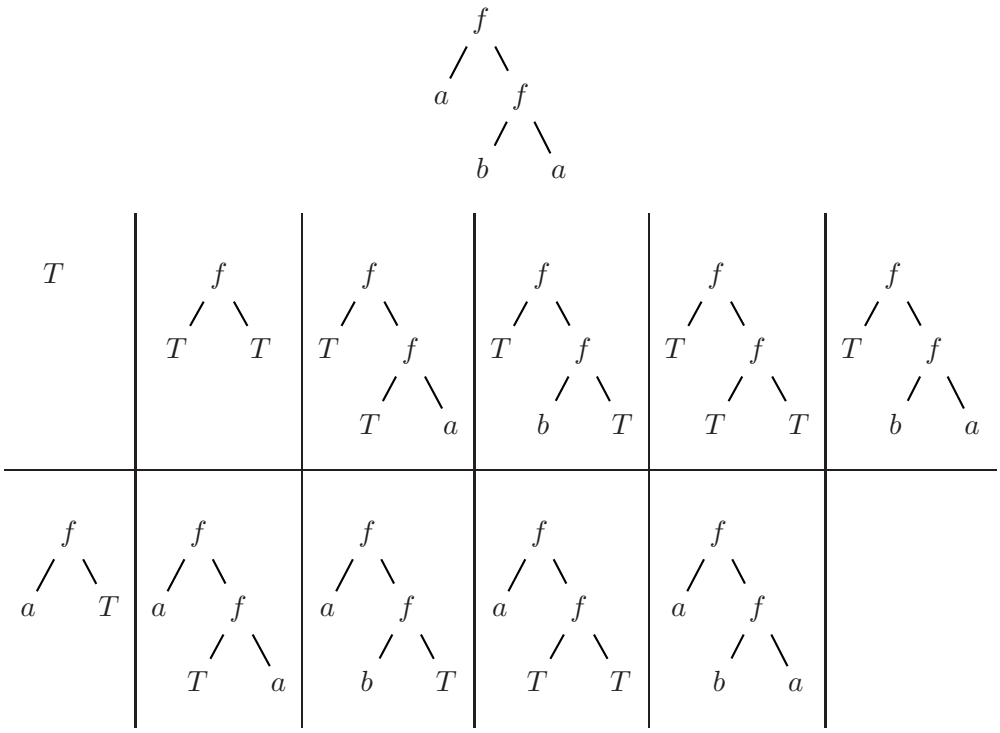


Figure 3.3: Un arbre sur $\Sigma \times \mathsf{Bool}$ (en haut). L'ensemble de ses élagages (en bas)

3.4.2 Les TSNe

L'élagage d'arbres

L'élagage d'un arbre consiste à en couper des branches et à les remplacer par le symbole spécial T.

Formellement, on peut définir l'ensemble des élagages d'un arbre ainsi :

Définition 3.11. *Soit t un arbre défini sur $\Sigma \times \mathsf{Bool}$. L'ensemble des élagages de t, noté $\mathsf{elag}(t)$, est l'ensemble des arbres t' tels que pour tout nœud π dans $\mathsf{dom}(t)$, l'une de ces trois propositions soit juste :*

- $(\pi \in \mathsf{dom}(t')) \wedge (t'(\pi) = t(\pi))$
- $(\pi \in \mathsf{dom}(t')) \wedge (t'(\pi) = T) \wedge (\forall u \in \mathbb{N}^+ \ (\pi.u \notin \mathsf{dom}(t'))$
- $(\pi \notin \mathsf{dom}(t')) \wedge \exists u \in \mathbb{N}^* \ \exists v \in \mathbb{N}^+ (\pi = u.v \wedge t'(u) = T)$

La Fig. 3.3 fournit un exemple d'ensemble des élagages d'un arbre sur $\Sigma \times \mathsf{Bool}$

Définir des requêtes avec des automates d'arbres élagués

La définition de l'élagage des arbres que nous venons de donner nous permet de définir des langages d'arbres élagués, et donc des automates dans les arbres élagués définissant des requêtes dans les arbres élagués.

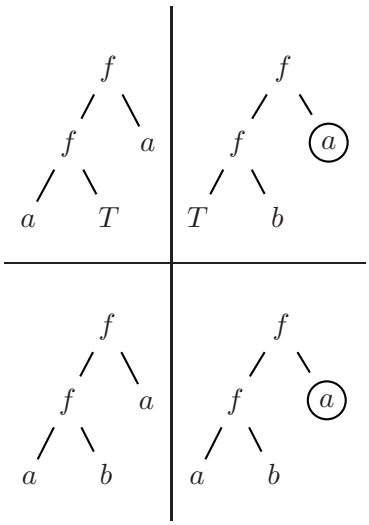


Figure 3.4: Deux élagage différents (en haut) donnant deux annotations inconsistantes du même arbre (en bas). On pourrait construire un automate d'arbres élagués qui reconnaît exactement ces deux arbres élagués. Il serait fonctionnel, et pourtant il ne serait pas satisfaisant pour décrire des requêtes sur les arbres non élagués.

C'est l'idée que nous avions déjà développée dans la partie 3.4.1. Dans de nombreux cas, il est plus simple d'exprimer des requêtes dans des arbres élagués, notamment pour traduire le fait qu'une partie de l'arbre ne nous intéresse pas pour définir la requête. Notre but est d'utiliser ces automates définis sur les arbres élagués pour définir des requêtes sur les arbres non élagués. Pour cela, nous allons simplement considérer que les nœuds d'un arbre non élagué t sélectionnés sont ceux qui sont sélectionnés dans un élagage de t.

De même que nous avons défini les TSN en caractérisant les automates d'arbres sur $\Sigma \times \mathsf{Bool}$ décrivant des requêtes, nous allons définir les TSNe en caractérisant les automates d'arbres sur $\Sigma \times \mathsf{Bool} \cup T$ qui définissent des requêtes sur Σ. La première chose que l'on peut remarquer est que la fonctionnalité de l'automate n'est pas une condition suffisante. En effet, il est possible d'avoir deux annotations distinctes du même arbre qui donnent des indications inconsistantes. La Fig. 3.4 en donne un exemple.

Le rôle de la fonctionnalité dans les TSN est d'empêcher une inconsistance entre deux annotations du même arbre. Pour caractériser cela dans le cas des arbres annotés, nous allons introduire la notion de *compatibilité* entre arbres élagués. Deux arbres sont compatibles si ils sont élagages du même arbre. Il y a inconsistance entre annotations quand un automate annote deux arbres élagués compatibles en deux annotations incompatibles.

Nous dirons qu'un automate sur $\Sigma \times \mathsf{Bool} \cup \{T\}$ est *pseudo-fonctionnel* si il annote toujours des arbres sur $\Sigma \times \mathsf{Bool} \cup \{T\}$ compatibles en annotations compatibles.

Définition 3.12. *Soit Γ un alphabet quelconque, t_1 et t_2 définis sur $\Gamma \cup \{T\}$.*

Les arbres t_1 et t_2 sont compatibles *si et seulement si il existe un arbre t défini sur Γ tel que $t_1 \in \mathsf{elag}(t)$ et $t_2 \in \mathsf{elag}(t)$. On note cette relation* $\mathsf{compat}(t_1, t_2)$.

Définition 3.13. *Soit A un automate sur $(\Sigma \times \mathsf{Bool}) \cup \{T\}$. A est* pseudo-fonctionnel *si et seulement si :*

$$\forall t_1 \times \beta_1 \in \mathsf{langage}(A)\ \forall t_2 \times \beta_2 \in \mathsf{langage}(A)\ (\mathsf{compat}(t_1, t_2) \Rightarrow \mathsf{compat}(\beta_1, \beta_2))$$

On peut remarquer qu'en l'absence d'occurrence d'un symbole T dans la définition d'un automate sur $\Sigma \times \mathsf{Bool} \cup T$, la compatibilité entre arbres se réduit à une égalité, et la pseudo-fonctionnalité à la fonctionnalité.

Nous avons défini la contrainte à imposer aux automates sur $\Sigma \times \mathsf{Bool} \cup \{T\}$ pour qu'ils définissent des requêtes sur Σ, donc nous pouvons maintenant définir les TSNe.

Définition 3.14. *Un Transducteur de Sélection de Nœuds élagueur (*TSNe*) sur Σ est un automate d'arbre pseudo-fonctionnel sur $\Sigma \times \mathsf{Bool} \cup \{T\}$.*

3.4.3 Calcul de requêtes avec les TSNe

Un TSNe sur Σ est un automate d'arbre fonctionnel sur $\Sigma \times \mathsf{Bool} \cup \{T\}$, il est donc capable de sélectionner des nœuds dans les arbres élagués, définis sur $\Sigma \cup \{T\}$. Soit t un arbre sur Σ. Un nœud π de t est sélectionné par une requête définie par un TSNe A si il existe un élagage de t dans lequel π est sélectionné par A.

Pour calculer une requête sur un arbre t défini sur Σ avec un TSNe, on pourrait simplement générer l'ensemble des élagages de t et appliquer l'algorithme décrit dans la section 3.2.2. Mais ce serait trop coûteux, car le nombre d'élagages possibles croît exponentiellement avec la taille des arbres.

Nous allons donc présenter une modification de l'algorithme de calcul de requête décrit en 3.2.2 qui permet de calculer directement une requête sur un arbre défini sur Σ avec un TSNe sur Σ. Il y a une seule modification à l'algorithme : pendant la phase ascendante, les états évaluant le symbole T sont associés à tout les nœuds de l'arbre. Toute les évaluations de tous les sous-arbres élagués possibles sont donc associées aux nœuds correspondants. Cela donne l'algorithme suivant pour la phase ascendante :

Calcul de requête, phase 1

Fonction $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})$
  Entrée : Un TSN A sur Σ
    Un arbre t sur Σ
  Sortie : Un arbre t sur $\Sigma \times 2^{\mathsf{etats}(A)}$
  Match t Avec
  a :
    $S \leftarrow \{q | \exists b \in \mathsf{Bool}\ (a, b) \rightarrow q \in \mathsf{regles}(A)\} \cup \{q | T \rightarrow q \in \mathsf{regles}(A)\}$
    Retour (a, S)
  $f(t_1, t_2)$:
    $t'_1 \leftarrow \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_1})$
    $t'_2 \leftarrow \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_2})$

Soit $(f_1, S_1) = t'_1(\mathsf{racine})$
Soit $(f_2, S_2) = t'_2(\mathsf{racine})$
$S \leftarrow \{q | \exists b \in \mathsf{Bool}\ \exists q_1 \in S_1\ \exists q_2 \in S_2\ (f,b)(q_1,q_2) \rightarrow q \in \mathsf{regles}(A)\} \cup \{q | T \rightarrow q \in \mathsf{regles}(A)\}$
Retour $(f, S)(t'_1, t'_2)$

La phase descendante est identique à la phase descendante du calcul de requête avec les TSN.

3.4.4 Test d'appartenance à l'ensemble des TSNe

Pour tester l'appartenance d'un arbre à l'ensemble des TSNe, il suffit de tester sa pseudo-fonctionnalité.

Le test de pseudo-fonctionnalité est en fait très proche du test de fonctionnalité. Rappelons que celui-ci était fondé sur le calcul des relations sim et drst définies en section 3.2.3. Nous allons définir les relations $\mathsf{sim_e}$ et $\mathsf{drst_e}$, analogues à sim et drst dans les arbres élagués.

Soit A un TSNe, q et q' deux états de A :
- $\mathsf{sim_e}_A(q, q')$ si et seulement si q et q' font respectivement partie des évaluations de $t \times \beta$ et $t' \times \beta'$, avec $\mathsf{compat}(t, t')$ et $\mathsf{compat}(\beta, \beta')$.
- $\mathsf{drst_e}_A(q, q')$ si et seulement si q et q' font respectivement partie des évaluations de $t \times \beta$ et $t' \times \beta'$, avec $\mathsf{compat}(t, t')$ et $\neg\mathsf{compat}(\beta, \beta')$.

De même que pour les TSN, la pseudo-fonctionnalité se déduit immédiatement du calcul de $\mathsf{drst_e}$:

Propriété 3.7. *Soit A un automate d'arbres sur $\Sigma \times \mathsf{Bool}$. L'automate A est pseudo-fonctionnel si et seulement si il n'existe pas de paire d'états finaux q et q' (pas nécessairement distincts) telle que $\mathsf{drst_e}_A(q, q')$.*

Preuve : L'existence de q et q' finaux tels que $\mathsf{drst_e}_A(q, q')$ est équivalente à l'existence de $t \times \beta$ et $t' \times \beta'$ acceptés par A avec $\mathsf{compat}(t, t')$ et $\neg\mathsf{compat}(\beta, \beta')$, et donc exactement la négation de la pseudo-fonctionnalité de l'automate. ◄

Les relations $\mathsf{sim_e}$ et $\mathsf{drst_e}$ peuvent se définir de façon récursive.

$$\mathsf{sim_e}_A(q, q') \Leftrightarrow$$
$$\begin{array}{ll} & ((a,b) \rightarrow q \wedge (a,b) \rightarrow q') \\ \vee & ((f,b)(q_1,q_2) \rightarrow q \wedge (f,b)(q'_1,q'_2) \rightarrow q' \wedge \mathsf{sim_e}_A(q_1,q'_1) \wedge \mathsf{sim_e}_A(q_2,q'_2)) \\ \vee & (T \rightarrow q) \\ \vee & (T \rightarrow q') \end{array}$$

$$\mathsf{drst_e}(q, q') \Leftrightarrow$$
$$\begin{array}{ll} & ((a,b) \rightarrow q \wedge (a,\neg b) \rightarrow q') \\ \vee & ((f,b)(q_1,q_2) \rightarrow q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \rightarrow q' \in \mathsf{regles}(A) \wedge \mathsf{drst_e}_A(q_1,q'_1) \wedge \mathsf{drst_e}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \rightarrow q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \rightarrow q' \in \mathsf{regles}(A) \wedge \mathsf{sim_e}_A(q_1,q'_1) \wedge \mathsf{drst_e}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \rightarrow q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \rightarrow q' \in \mathsf{regles}(A) \wedge \mathsf{drst_e}_A(q_1,q'_1) \wedge \mathsf{sim_e}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \rightarrow q \in \mathsf{regles}(A) \wedge (f,\neg b)(q'_1,q'_2) \rightarrow q' \in \mathsf{regles}(A) \wedge \mathsf{sim_e}_A(q_1,q'_1) \wedge \mathsf{sim_e}_A(q_2,q'_2)) \end{array}$$

On remarquera la proximité entre ces expressions et les expressions de sim et drst. La seule différence se situe dans le traitement particulier du symbole T : tout état évaluant T est similaire à n'importe quel autre état. Cela rejoint la remarque que nous avons déjà faite préalablement selon laquelle pseudo-fonctionnalité et fonctionnalité se confondaient dans le cas d'un TSNe dont les règles ne contiennent aucune occurrence du caractère T.

L'algorithme de test de pseudo-fonctionnalité d'un automate d'arbres sur $(\Sigma \times \mathsf{Bool}) \cup \{T\}$ est tout-à-fait similaire à l'algorithme de test de fonctionnalité décrit dans la section 3.2.3. Il faut simplement ajouter les règles concernant le symbole T au début du programme principal :

Test de fonctionnalité d'un automate sur $\Sigma \times \mathsf{Bool}$

Entrée : Un automate sur $\Sigma \times \mathsf{Bool}$
Sortie : vrai si l'automate est fonctionnel, faux sinon
Fonction affecte_$\mathsf{sim_e}(q, q') =$
    $\mathsf{sim_e}(q, q') \leftarrow$ vrai
    PourTout $((f, b)(q, q_2) \rightarrow q_0) \in \mathsf{regles}(A)$
        PourTout $((f, b')(q', q'_2) \rightarrow q'_0) \in \mathsf{regles}(A)$
            Si $\mathsf{sim_e}(q_2, q'_2) \wedge b = b'$ Alors affecte_$\mathsf{sim_e}(q_0, q'_0)$
            Si $(\mathsf{sim_e}(q_2, q'_2) \wedge b \neq b') \vee \mathsf{drst_e}(q_2, q'_2)$ Alors affecte_$\mathsf{drst_e}(q_0, q'_0)$
    PourTout $((f, b)(q_1, q) \rightarrow q_0) \in \mathsf{regles}(A)$
        PourTout $((f, b')(q'_1, q') \rightarrow q'_0) \in \mathsf{regles}(A)$
            Si $\mathsf{sim_e}(q_1, q'_1)$ Alors affecte_$\mathsf{sim_e}(q_0, q'_0)$
            Si $(\mathsf{sim_e}(q_1, q'_1) \wedge b \neq b') \vee \mathsf{drst_e}(q_1, q'_1)$ Alors affecte_$\mathsf{drst_e}(q_0, q'_0)$

Fonction affecte_$\mathsf{drst_e}(q, q') =$
    $\mathsf{drst_e}(q, q') \leftarrow$ vrai
    PourTout $((f, b)(q, q_2) \rightarrow q_0) \in \mathsf{regles}(A)$
        PourTout $((f, b')(q', q'_2) \rightarrow q'_0) \in \mathsf{regles}(A)$
            Si $\mathsf{sim_e}(q_2, q'_2) \vee \mathsf{drst_e}(q_2, q'_2)$ Alors affecte_$\mathsf{drst_e}(q_0, q'_0)$
    PourTout $((f, b)(q_1, q) \rightarrow q_0) \in \mathsf{regles}(A)$
        PourTout $((f, b')(q'_1, q') \rightarrow q'_0) \in \mathsf{regles}(A)$
            Si $\mathsf{sim_e}(q_1, q'_1) \vee \mathsf{drst_e}(q_1, q'_1)$ Alors affecte_$\mathsf{drst_e}(q_0, q'_0)$

PourTout (q, q')
    $\mathsf{drst_e}(q, q') \leftarrow$ faux
    $\mathsf{sim_e}(q, q') \leftarrow$ faux

——Partie ajoutée par rapport au test de fonctionnalité——
PourTout $(T \rightarrow q) \in \mathsf{regles}(A)$
    PourTout $q' \in \mathsf{etats}(A)$
        affecte_$\mathsf{sim_e}(q, q')$
——Fin de la partie ajoutée par rapport au test de fonctionnalité——

PourTout $((a, b) \rightarrow q) \in \mathsf{regles}(A)$
    PourTout $((a, b') \rightarrow q') \in \mathsf{regles}(A)$
        Si $b \neq b'$ Alors affecte_$\mathsf{drst_e}(q, q')$ Sinon affecte_$\mathsf{sim_e}(q, q')$

PourTout $((f,b)(q_1,q_2) \to q) \in \mathsf{regles}(A)$
  PourTout $((f,b')(q_1,q_2) \to q') \in \mathsf{regles}(A)$
    Si $b \neq b'$ Alors affecte_$\mathsf{drst_e}(q,q')$ Sinon affecte_$\mathsf{sim_e}(q,q')$

PourTout $q \in \mathsf{finaux}(A)$
  PourTout $q' \in \mathsf{finaux}(A)$
    Si $\mathsf{drst_e}(q,q')$ Alors Retour faux
Retour vrai

Théorème 3.5. *Le test de pseudo-fonctionnalité d'un automate sur* $\Sigma \times \mathsf{Bool}$*, c'est à dire le test d'appartenance à la classe des* TSNe*, se fait en* $O|\mathsf{etats}(A)|.|\mathsf{regles}(A)|^2$.

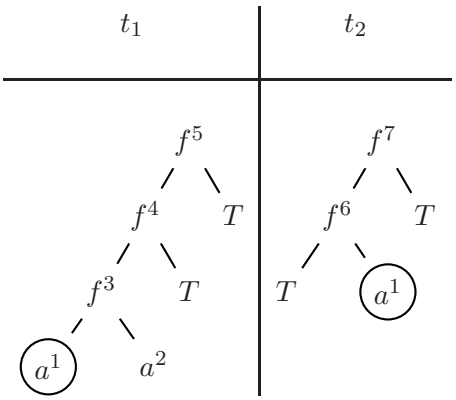
Ce résultat est similaire à celui du théorème 3.1, l'algorithme utilisé étant similaire.

Nous allons maintenant illustrer le fonctionnement des TSNe sur un exemple.

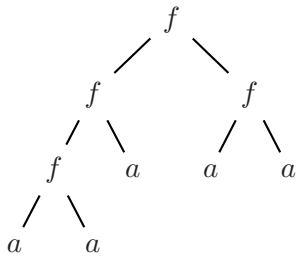
Exemple 3.4. *Nous allons considérer le* TSNe A *défini ainsi :*

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{1,2,3,4,5,6,7\} \\ \mathsf{finaux}(A) & = & \{5,7\} \\ \mathsf{regles}(A) & = & \{T \to T,\ (a,\mathsf{V}) \to 1,\ (a,\mathsf{F}) \to 2, \\ & & (f,\mathsf{F})(1,2) \to 3,\ (f,\mathsf{F})(3,T) \to 4,\ (f,\mathsf{F})(4,T) \to 5, \\ & & (f,\mathsf{F})(T,1) \to 6,\ (f,\mathsf{F})(6,T) \to 7\} \end{array}$$

Ce TSNe *reconnaît exactement les arbres élagués suivants (les états correspondants sont indiqués en exposants) :*

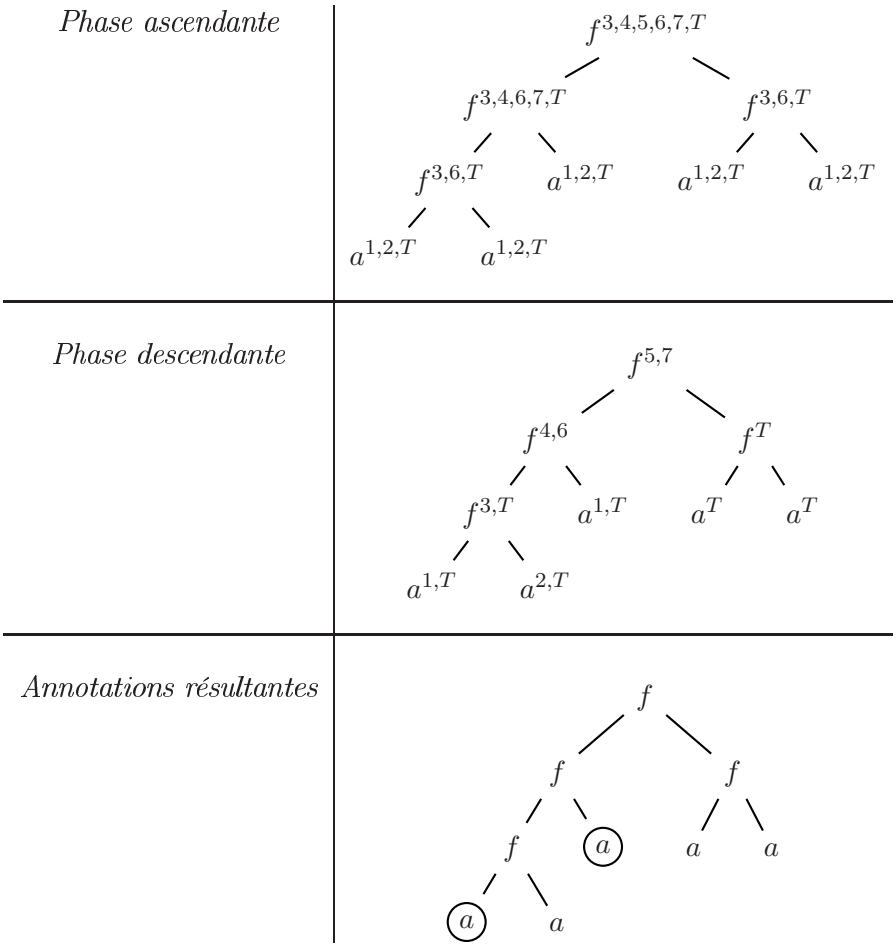


Nous allons calculer l'annotation par ce TSNe *d'un arbre* t *défini sur* $\Sigma \times \mathsf{Bool}$ *:*



Il est intéressant de voir que t_1 et t_2 sont tous les deux des élagages de t, et qu'ils vont engendrer la sélection de nœuds différents dans t. Cependant, il n'y a pas d'inconsistance car les annotations de t_1 et de t_2 sont compatibles. *En effet, aucun nœud n'est sélectionné dans t_1 sans l'être dans t_2. Quand un nœud est sélectionné dans t_1, soit il l'est également dans t_2, soit il fait partie d'une branche coupée dans t_2.*

Voici les arbres correspondant à la phase ascendante, la phase descendante puis l'annotation de t par le TSNe *A :*



La phase ascendante se passe exactement comme pour un TSN, *sauf que l'on retrouve l'état T partout. Pour la phase descendante, on remarquera que les nœuds de la branche la plus à droite sont associés exclusivement à l'état T.*

Chapitre 4

Apprentissage de requêtes

Les travaux que nous avons effectués jusque là nous ont permis d'introduire des objets décrivant des requêtes, les TSN et les TSNe. Si nous avons introduit ce formalisme, fondé sur l'idée qu'une requête dans les arbres s'apparente à un langage d'arbres, c'est dans l'idée d'en arriver à l'objet essentiel de nos travaux, à savoir l'apprentissage de requêtes. En effet, l'apprentissage de langages d'arbres est un sujet qui a déjà été largement étudié en inférence grammaticale, et il nous est ainsi possible de nous fonder sur ces travaux pour écrire des algorithmes d'inférence de requêtes.

Nous allons commencer ce chapitre par une rapide introduction à l'apprentissage automatique, que nous placerons rapidement dans le cadre de l'inférence grammaticale. Nous présenterons deux modèles d'apprentissage classiques en inférence grammaticale, ainsi que l'algorithme tRPNI, algorithme d'apprentissage de langages d'arbres qui servira de base à nos travaux.

Ensuite, nous présenterons tsnRPNI, notre algorithme d'apprentissage de TSN déterministes, et nous établirons les résultats d'apprentissage qui lui sont associés. La partie suivante sera consacrée à tsn_eRPNI, adaptation de tsnRPNI à l'apprentissage de TSNe déterministes, c'est-à-dire d'automates déterministes décrivant des requêtes sur des arbres élagués.

Enfin, nous montrerons que les algorithmes présentés peuvent se placer dans le cadre de l'apprentissage actif, cadre d'apprentissage se rapprochant davantage de l'apprentissage de requêtes par interaction avec un utilisateur, comme dans l'outil que nous avons développé et que nous présenterons dans le chapitre 5 : Squirrel.

4.1 Apprentissage de langages d'arbres

Cette partie du chapitre ne présente aucune contribution personnelle. Elle consiste en une présentation succincte de l'inférence grammaticale, des modèles d'apprentissage que nous allons utiliser, et de l'algorithme d'apprentissage de langages d'arbres à partir duquel nous allons écrire nos propres algorithmes.

4.1.1 Généralités sur l'apprentissage

Nous allons présenter ici le problème de l'apprentissage supervisé, que nous placerons directement dans le cadre qui nous concerne ici, celui de l'inférence grammaticale.

Commençons par décrire un modèle du problème de l'apprentissage supervisé. On considère un ensemble de données et un ensemble de classes. On dispose d'un processus, dont on ne connaît pas le fonctionnement interne, qui associe une classe à chaque donnée. En observant le comportement du processus sur un nombre fini de données, on cherche à prédire le comportement du processus sur d'autres données, en minimisant la probabilité d'erreur.

Reformulons ce problème de façon plus spécifique, en l'adaptant à l'inférence grammaticale. On considère une classe d'objets C, une classe d'exemples E, une relation d'équivalence sur les objets, une relation de consistance entre les exemples et les objets. On considère un objet L que l'on appelle la cible. A partir d'un nombre fini d'exemples consistants avec L, on cherche à identifier un objet équivalent à L connaissant C.

Typiquement, considérons comme classe d'objets l'ensemble des langages réguliers sur Σ, comme classe d'exemples les mots sur Σ accompagnés d'un booléen indiquant si il s'agit d'un exemple ou d'un contre-exemple, comme relation d'équivalence l'égalité, et comme relation de consistance l'appartenance pour les exemples positifs et la non appartenance pour les exemples négatifs. On obtient le problème, classique en inférence grammaticale, de l'identification des langages réguliers par exemples positifs et négatifs.

4.1.2 Modèles d'apprentissage

Le principe d'un modèle d'apprentissage est de décrire formellement le contexte dans lequel on fait fonctionner un algorithme et la capacité de cet algorithme à identifier une cible dans ce contexte.

Nous allons nous intéresser à deux modèles d'apprentissages : le modèle d'apprentissage à la limite et le modèle d'apprentissage par données fixées. Le premier parce que c'est le modèle fondateur et qu'il est la base des autres modèles, et le deuxième parce qu'il permet de définir des contrainte d'efficacité qui le rendent plus intéressant dans la pratique.

Identification à la limite

Ce modèle est le premier modèle formel d'apprentissage, introduit par Gold en 1967 [Gold, 1967].

Ce modèle place l'apprentissage dans le cadre de l'observation d'une séquence infinie d'exemples. Une classe d'objet C est identifiable à la limite à partir d'une classe d'exemples E si il existe un algorithme capable d'identifier la cible L au bout d'un nombre fini d'observations d'exemples consistants avec L, quelle que soit la cible et la présentation. La séquence observée doit être une *présentation complète* des exemples, c'est-à-dire que pour tout exemple consistant avec L, il existe un rang i tel que cet exemple est le i-ème exemple de la séquence.

Dans ce modèle d'apprentissage, les langages réguliers sont identifiables à partir d'exemples positifs et négatifs [Gold, 1967]

Lorsque l'on s'intéresse à l'inférence grammaticale par exemples positifs et négatifs, ce modèle est trop permissif pour être intéressant. En effet, il est montré dans [Gold, 1967] que toute classe de récursivement énumérable de langages récursifs est identifiable à la limite par exemples positifs et négatifs. L'algorithme permettant d'effectuer cette identification est un algorithme par énumération : à chaque nouvel exemple observé, en cas d'inconsistance, on énumère les cibles potentielles jusqu'à en trouver une consistante avec tous les exemples déjà observés.

Identification par données fixées

Ce modèle, introduit par Gold en 1978 [Gold, 1978], est une variante de l'identification à la limite qui va nous permettre d'introduire des contraintes d'efficacité sur les algorithmes d'apprentissage.

On appelle *échantillon de L* tout ensemble fini d'exemples consistants avec L. Une classe d'objets C est identifiable par données fixées à partir d'une classe d'exemples E si il existe un algorithme capable de déterminer un objet L de C à partir de tout échantillon contenant S_L, l'*échantillon caractéristique* de L.

Sans contraintes supplémentaires, ce modèle est équivalent au modèle d'identification à la limite. Mais il nous permet facilement de définir des contraintes de polynomialité : polynomialité du temps de réponse de l'algorithme d'apprentissage par rapport à la taille de l'échantillon d'entrée, et polynomialité de la taille de l'échantillon caractéristique de chaque objet par rapport à sa représentation la plus petite. Notons que cela implique que les résultats d'apprenabilité selon ce modèle dépendent de la représentation des objets.

Voici la définition formelle de ce modèle d'apprentissage, telle qu'elle est formalisée dans [de la Higuera, 1997] :

Définition 4.1. *Soit* exemples *et* classe *deux ensembles, $\sim$ une relation de consistance entre* exemples *et* classe, $\equiv$ *une relation d'équivalence sur* classe, rep *une représentation des objets de* classe *et* echantillons *l'ensemble des sous-ensembles finis de* exemples. *Un échantillon est consistant avec une classe si tous ses éléments le sont.*

Les éléments de classe *représentés par* rep *sont polynomialement identifiables à la limite à partir de* exemples *si il existe deux fonctions* identifie : echantillons $\rightarrow$ classe *et* caracterise : classe $\rightarrow$ echantillons *tels que :*

- caracterise(L) *est toujours consistant avec L. On dit que* caracterise(L) *est* l'échantillon caractéristique *de L.*
- *Pour tout échantillon E et toute classe L, si E est consistant avec L et si E contient* caracterise(L), *alors* identifie(E) $\equiv L$.
- identifie(E) *se calcule en temps polynomial en la taille de E.*
- *La cardinalité de* caracterise(L) *est polynomiale en la taille de L. Dans le modèle décrit en [de la Higuera, 1997], c'est* caracterise(L) *qui doit être polynomial en la taille de* rep(L). *Nous verrons dans la partie 4.1.3 pourquoi nous avons dû effectuer ce léger changement dans le modèle.*

Par exemple, il est montré dans [Gold, 1978] que les langages réguliers représentés par des automates déterministes sont polynomialement identifiables par données fixées à partir

d'exemples positifs et négatifs. Si on représente les langages par des automates non déterministes, ce résultat n'est plus vrai [de la Higuera, 1997].

4.1.3 Apprentissage de langages d'arbres

Dans les modèles d'apprentissages précédemment décrits, nous avons pris comme exemple l'apprentissage de langages réguliers de mots, car il s'agit du cas le plus étudié en inférence grammaticale.

Le plus classique des algorithmes d'identification de langages réguliers de mots est RPNI [Oncina and Garcia, 1992]. Cet algorithme identifie les langages réguliers de mots représentés par des automates déterministes par exemples positifs et négatifs selon le modèle d'identification à la limite. Nous allons présenter ici tRPNI, adaptation de RPNI aux langages d'arbres. Cet algorithme a déjà été introduit dans [Oncina and García, 1993].

Description de tRPNI

L'idée générale de tRPNI est de partir d'un automate très spécifique, qui reconnaît exactement les exemples positifs de l'échantillon d'entrée, puis de procéder par généralisations successives en fusionnant les états de cet automate tant qu'il reste consistant avec les exemples négatifs de l'échantillon d'entrée.

L'automate initial est l'équivalent pour les ensembles d'arbres de l'arbre préfixe pour les ensembles de mots. Il s'agit du plus grand automate déterministe émondé qui reconnaît exactement l'ensemble des exemples positifs. Il est construit simplement en créant un noeud par sous-arbre des exemples positifs, et en construisant les règles associées.

---

initial(S^+)

Entrée : Ensemble d'exemples positifs S^+
Soit $t_1, \ldots, t_n$ ensemble des sous-arbres des arbres de S^+
Soit A automate d'arbres tel que :

$$\begin{aligned} \text{etats}(A) &= \{q_1, \ldots, q_n\} \\ \text{finaux}(A) &= \{q_i | t_i \in S^+\} \\ \text{regles}(A) &= \{f(q_{i_1}, \ldots, q_{i_k}) \rightarrow q_j | t_j = f(t_{i_1}, \ldots, t_{i_k})\} \end{aligned}$$

Sortie : Automate A

Une fusion d'état fusion(A, q_1, q_2) consiste à remplacer toutes les occurrences de q_2 dans les règles de A par q_1. L'automate doit rester déterministe après la fusion. Pour cela, d'autres fusions peuvent se succéder en cascade. L'ensemble de cette séquence de fusions est appelée *fusion déterministe*.

---

fusion_deterministe(A, q_a, q_b)

Entrée : Un automate d'arbres A, deux états q_a et q_b
PourTout $r \in$ regles(A)
    Remplacer les occurrences de q_b par q_a dans r
PourTout $(f(q_1, \ldots, q_n) \rightarrow q, f(q_1, \ldots, q_n) \rightarrow q')$ tel que $q \neq q'$
    fusion_deterministe(A, q, q')
Sortie : A

L'algorithme principal consiste à partir de l'automate initial et à tenter l'ensemble des fusions déterministes possibles selon un ordre respectant ces critères :

- L'ordre doit être fixé au départ.
- L'ordre doit respecter l'ordre de hauteur des états dans l'automate initial. On associe à chaque état une hauteur qui correspond à la hauteur de l'unique arbre qu'il reconnaît dans l'automate initial. L'ordre des fusions doit respecter la règle suivante : la fusion de deux états q_1 et q_2 de hauteur h_1 et h_2 doit être précédée de l'ensemble des fusions possibles entre états de hauteur strictement inférieure à $max(h_1, h_2)$

Ces contraintes sur l'ordre des fusions sont nécessaires pour obtenir les résultats d'apprenabilité que nous allons décrire par la suite. Dans la pratique, il est possible d'améliorer les performances de l'algorithme en s'affranchissant de ces contraintes et en ajoutant des règles heuristiques pour déterminer l'ordre des fusions [Lang et al., 1998]. Nous reverrons cela en détail dans le chapitre suivant.

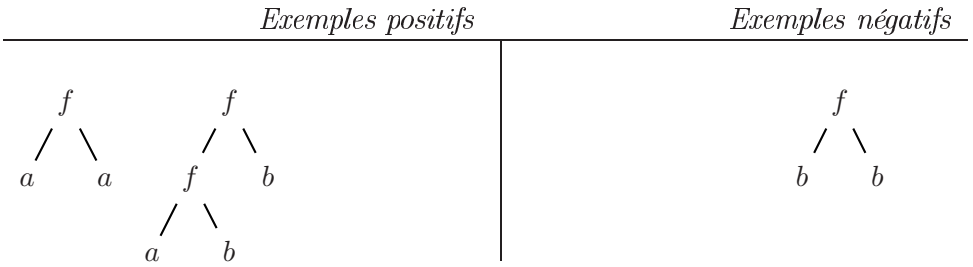
---

tRPNI (S^+, S^-)

Entrée : Un échantillon d'exemples positifs S^+
Un échantillon d'exemples négatifs S^-
$A \leftarrow$ initial(S^+)
Soit $F = \{(q_{i_1}, q_{j_1}), \ldots, (q_{i_n}, q_{j_n})\}$ séquence ordonnée des paires d'états de A
    respectant l'ordre de hauteur des états dans A
PourTout (q_i, q_j) dans F
    $A' \leftarrow$ fusion_deterministe(A, q_i, q_j)
    Si A' est consistant avec S^-
        $A \leftarrow A'$
Sortie : A'

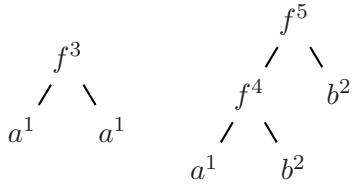
Exemple 4.1. *Nous allons illustrer le fonctionnement de* tRPNI *par un exemple.*

Considérons l'ensemble d'apprentissage suivant :



L'automate initial se construit par étiquetage de chacun des sous-arbres des exemples positifs :

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{1,2,3,4,5\} \\ \mathsf{finaux}(A) & = & \{3,5\} \\ \mathsf{regles}(A) & = & \{a \to 1, b \to 2, \\ & & f(1,1) \to 3, f(1,2) \to 4, f(4,2) \to 5\} \end{array}$$



L'apprentissage se fait ensuite par fusion d'états. On tente d'abord de fusionner 1 *et* 2*, mais cette fusion est interdite car l'automate obtenu reconnaîtrait l'exemple négatif. On fusionne alors* 1 *et* 3*, puis* 1 *et* 4*, et enfin* 1 *et* 5*. Plus aucune fusion n'est possible ensuite. L'automate obtenu est le suivant :*

$$\begin{array}{rcl} \mathsf{etats}(A) & = & \{1,2\} \\ \mathsf{finaux}(A) & = & \{1\} \\ \mathsf{regles}(A) & = & \{a \to 1, b \to 2, \\ & & f(1,1) \to 1, f(1,2) \to 1\} \end{array}$$

L'algorithme a inféré à partir des exemples donnés le langage suivant : tous les arbres définis sur $\{f_2, a_0, b_0\}$ *dans lesquels* b *n'est jamais un fils gauche sont acceptés.*

Résultat d'apprentissage sur tRPNI

Il est prouvé dans [Oncina and García, 1993] que les langages réguliers d'arbres représentés par des automates d'arbres déterministes sont polynomialement identifiables par données fixées à partir d'exemples positifs et négatifs en utilisant l'algorithme tRPNI.

On peut remarquer que ce résultat n'est vrai que dans le modèle par données fixées relâché tel que nous l'avons décrit. En effet, dans [Oncina and García, 1993], il est montré que la cardinalité de l'ensemble caractéristique pour tRPNI d'un langage est polynomiale en la taille de la représentation de ce langage. Or, dans le modèle d'apprentissage par données fixées classique, c'est la taille de l'échantillon caractéristique d'un langage qui doit être polynomiale en la taille de la représentation de ce langage, ce qui est une condition strictement plus forte. Voici un exemple qui montre que dans ce modèle classique, tRPNI n'est pas capable d'identifier les langages réguliers d'arbres représentés par des automates d'arbres déterministes.

Exemple 4.2. *Soit L_n le langage d'arbre sur $\Sigma = \{f_2, a_0\}$ contenant l'unique arbre équilibré de hauteur n défini sur Σ. Par exemple, $L_3 = \{f(f(a,a), f(a,a))\}$.*

L'algorithme tRPNI *nécessite au moins un exemple positif, donc l'échantillon caractéristique du langage L_n doit contenir son unique élément. Or la taille de cet élément est exponentielle en n, alors que la taille de l'automate déterministe minimal reconnaissant L_n est polynomiale en n. Donc, au sein de la classe de langage réguliers $\{L_n\}_{n>0}$, la taille des ensembles caractéristiques des langages est exponentielle en la taille des automates déterministes minimaux représentant les langages.*

4.2 Apprentissage de requêtes représentées par des TSN déterministes

Le cadre d'apprentissage que nous avons présenté s'adapte facilement au problème de l'apprentissage de requêtes monadiques régulières dans les arbres. Les cibles sont les requêtes régulières, et les données les arbres étiquetés par ces requêtes. Ce cadre correspond à l'apprentissage de requêtes tel qu'on pourrait l'imaginer dans la pratique : on dispose d'un ensemble de documents dans lesquels les éléments à sélectionner sont annotés, et on cherche à inférer à partir de ces documents la requête correspondante. Une fois cette requête calculée, on peut l'utiliser pour sélectionner des éléments dans des documents non annotés.

Les objets que nous avons introduits dans le chapitre précédent nous fournissent des représentations de requêtes adaptées à l'apprentissage : nous avons défini les langages de requêtes, langages d'arbres décrivant les requêtes dans les arbres, et les TSN, puis les TSNe, automates d'arbres définissant ces langages. Or nous disposons d'algorithmes permettant l'apprentissage de langages d'arbres représentés par des automates d'arbres déterministes. Dans cette partie, nous allons nous intéresser à l'apprentissage de requêtes représentées par des TSN déterministes, d'abord en présentant un algorithme dérivé de tRPNI, puis en introduisant un résultat d'apprenabilité lui correspondant. Dans la partie suivante, nous nous intéresserons à l'apprentissage de requêtes représentées par des TSNe déterministes.

4.2.1 Algorithme d'apprentissage des TSN déterministes

En 4.1.3, nous avons introduit tRPNI, algorithme d'apprentissage de langages d'arbres représentés par des automates d'arbres déterministes. On pourrait directement utiliser cet algorithme pour apprendre des langages de requêtes, mais cela ne conviendrait pas à l'apprentissage de requêtes pour une raison simple : tRPNI a besoin d'exemples "positifs", c'est-à-dire d'arbres inclus dans le langage cible, et d'exemples "négatifs", c'est-à-dire d'arbres exclus du langage cible. Dans le cas des langages de requête, les exemples positifs sont des arbres complètement annotés, et les exemples négatifs sont des arbres dont l'annotation est incorrecte. Dans la pratique, il est difficile d'imaginer une source pertinente d'exemples négatifs dans ce contexte.

Nous allons introduire ici tsnRPNI, un algorithmes qui utilise les propriétés spécifiques des TSN pour les apprendre à partir d'exemples positifs. Cet algorithme repose sur le fait que les TSN ne reconnaissent qu'une annotation possible de chaque arbre. Ainsi, chaque arbre

annoté apporte une information "positive" (cet arbre annoté est reconnu par le TSN), et une information "négative" (aucune autre annotation de cet arbre n'est reconnue par le TSN). Pour différencier cet apprentissage de l'apprentissage par exemples positifs seuls au sens classique du terme, nous parlerons ici d'*apprentissage par exemples complètement annotés*.

Description de l'algorithme

L'algorithme que nous présentons ici, tsnRPNI, prend en entrée un échantillon complètement annoté d'un langage de requêtes, c'est-à-dire un ensemble d'arbres définis sur $\Sigma \times \mathsf{Bool}$, et donne en sortie un TSN déterministe sur Σ consistant avec cet échantillon.

---

tsnRPNI (S)

Soit A l'automate préfixe de S
Soit l une séquence ordonnée des paires d'éléments de $\mathsf{etats}(A)$
PourTout (q_1, q_2) de l
    $A' \leftarrow \mathsf{fusion\_deterministe}(A, q_1, q_2)$
    Si A' est fonctionnel Alors
        $A \leftarrow A'$
Sortie : A

Similitude avec tRPNI

L'algorithme tsnRPNI est similaire à tRPNI en tout point sauf dans le test d'acceptation des fusions d'états. Au lieu d'autoriser une fusion d'états quand l'automate résultant reste consistant avec les exemples négatifs, on va l'autoriser quand l'automate résultant reste fonctionnel. Comme tRPNI, tsnRPNI est sensible à l'ordre dans lequel les fusions sont tentées, et les contraintes sur cet ordre pour obtenir les résultats d'apprenabilité sont les mêmes. Nous verrons dans la partie expérimentale de nos travaux qu'il est possible d'améliorer les performances de l'algorithme en changeant cet ordre.

Capacité de généralisation de tsnRPNI

Il est facile de voir que tRPNI renvoie un TSN déterministe : chaque étape maintient le déterminisme et la fonctionnalité. Il est également clair que ce TSN est consistant avec l'échantillon d'entrée : comme tRPNI, tsnRPNI part d'un automate reconnaissant exactement l'échantillon d'entrée, puis peu à peu *généralise* à chaque fusion d'état en *augmentant* le langage reconnu par l'automate. Ainsi, la requête décrite par le TSN généré est au départ capable d'effectuer uniquement les extractions décrites dans l'échantillon d'entrée, puis au fur et à mesure des fusions accroît sa capacité d'extraction de noeuds à un nombre croissant d'arbres. La Fig. 4.1 (page 75) illustre cette capacité de généralisation en montrant comment à partir d'un seul exemple il est possible d'inférer la requête "Sélectionner toutes les feuilles de profondeur impaire".

D'un point de vue théorique, on peut caractériser la capacité de généralisation d'un algorithme en prouvant un résultat d'apprenabilité selon un modèle d'apprentissage. Ce sera l'objet de la section 4.2.2.

D'un point de vue pratique, les conditions imposées par ce modèle d'apprentissage sont rarement réunies. Nous verrons dans le chapitre 5, qui concerne la partie expérimentale de notre travail, comment notre algorithme se comporte face à des données réelles.

Complexité de tsnRPNI

Théorème 4.1. *Le calcul de* tsnRPNI *se fait en temps polynomial par rapport à la taille de l'échantillon d'entrée. En posant n la taille de l'échantillon d'entrée, la complexité en temps de* tsnRPNI *est en $O(n^5)$.*

Preuve :

Lors de la construction de l'automate initial, on ajoute une règle et un état par sous-arbre d'arbres de l'échantillon d'entrée distincts. Il en résulte que n majore le nombre d'états et le nombre de règles de l'automate initial.

Chaque exécution de la boucle de tsnRPNI *est soit une réussite (l'automate reste fonctionnel), soit un échec (l'automate n'est plus fonctionnel).*

Le nombre de réussites est majoré par n, car on enlève un état de l'automate à chaque réussite. Le nombre d'échecs est majoré par n^2, car on ne réessaye jamais une fusion qui a échoué. Nous allons évaluer indépendamment la complexité de l'ensemble des réussites et celle de l'ensemble des échecs.

Chaque échec consiste en une succession de fusions déterministes terminé par un test de fonctionnalité. Le nombre de fusions successives est majoré par n, et chacune d'entre elles consiste en une mise à jour de toutes les règles de l'automate. La complexité en temps de la fusion déterministe est donc en $O(n^2)$. Or celle du test de fonctionnalité est en $O(n^3)$ (voir théorème 3.1). La complexité en temps de chaque échec est donc en $O(n^3)$, et donc celle de l'ensemble des échecs en $O(n^5)$.

Chaque réussite est également une succession de fusions déterministes terminée par un test de fonctionnalité. Ici, le nombre total de fusions au cours de toutes les fusions déterministes acceptées est majoré par n, donc le nombre total de tests de fonctionnalité est majoré par n. De plus, chaque test de fonctionnalité se fait sur un automate plus général que le précédent. En reprenant l'algorithme du théorème 3.1, on peut voir qu'il est facile d'en faire une version incrémentale. *Les* drst *affectés au cours d'un test de fonctionnalité resteront vrais pour le test suivant. Ainsi, la complexité de n tests successifs de fonctionnalité sur des automates à chaque fois plus généraux est également en $O(n^3)$. Ainsi, la complexité de l'ensemble des fusions réussies par* tsnRPNI *est en $O(n^3)$.*

La complexité en temps de tsnRPNI *est donc en $O(n^5)$.* ◄

Cette complexité peut sembler très élevée, mais dans la pratique l'algorithme fonctionne bien plus vite que ce résultat théorique ne pourrait le laisser penser. En effet, le nombre d'échecs de fusions est bien plus faible que $|\mathsf{etats}(A)|^2$, car les heuristiques de choix de l'ordre des fusions que nous présenterons dans le chapitre 5, qui tentent de "deviner" les fusions

les plus intéressantes, devinent en général des fusions qui seront acceptées. De plus, chaque réussite induit une diminution du nombre d'états, et donc du nombre d'échecs possibles. Cela n'est pas quantifiable en terme de complexité, mais nous pouvons donner une idée du gain qu'induit une réduction du nombre d'échecs en évaluant la complexité de l'algorithme en fonction du nombre d'échecs.

Notons err le nombre de fusions échouées. Même si err n'est pas une donnée d'entrée, il est intéressant de voir que si on l'utilise comme paramètre de la complexité en temps de tsnRPNI, alors on obtient le résultat plus raisonnable de $O(n^3 + err.n^3)$. A titre indicatif, dans la pratique le nombre d'erreur est souvent inférieur à 10 pour inférer une requête non triviale sur un ensemble de page extrait d'un des sites Web de notre corpus.

4.2.2 Identification des TSN par données fixées

Nous allons tenter ici de caractériser la capacité de généralisation de tsnRPNI d'un point de vue théorique en nous intéressant à l'apprenabilité des TSN. Nous allons nous placer dans le modèle d'apprentissage par données fixées ([Gold, 1967]) et montrer que tsnRPNI est capable d'apprendre les requêtes représentées par des TSN à partir d'exemples complètement annotés dans ce modèle.

Il est établi que les automates d'arbres sont polynomialement identifiables par données fixées à partir d'exemples positifs et négatifs en utilisant l'algorithme tRPNI. Nous allons montrer que les TSN sont polynomialement identifiables par données fixées à partir d'exemples complètement annotés en utilisant l'algorithme tsnRPNI.

Pour cela, nous allons procéder en deux étapes :
- Tout d'abord, nous allons montrer que tRPNI et tsnRPNI se comportent de façon identique si on dispose des échantillons caractéristiques et si on cible un TSN complet (voir def. 3.6). C'est l'objet du lemme 4.1.
- Ensuite, nous en déduirons qu'en redéfinissant de façon satisfaisante les relations de consistance et d'équivalence entre les TSN (définitions 4.2 et 4.3), on peut en déduire que les requêtes définies par des TSN sont identifiables par données fixées en utilisant tsnRPNI (propriété 4.1 et théorème 4.2)

Les automates d'arbres que nous apprenons étant fonctionnels, chaque exemple complètement annoté induit un exemple positif, lui-même, et un ensemble d'exemples négatifs, les autres annotations du même arbre. Ainsi, il est possible de définir un échantillon d'exemples positifs et négatifs $\mathsf{pn}(E)$ implicitement induits par un échantillon d'exemples annotés E :

$$\mathsf{pn}(E) = \{(t \times \beta', b) | \exists \beta \; t \times \beta \in E \wedge (b \Leftrightarrow (\beta = \beta')\}$$

Lemme 4.1. *Soit A un* TSN *complet, et E un sous-ensemble de* langage(A). *Soit* $\mathsf{char}_{\mathsf{tRPNI}}(A)$ *l'ensemble caractéristique de* langage(A) *pour* tRPNI. *Si* $\mathsf{char}_{\mathsf{tRPNI}}(A) \subseteq \mathsf{pn}(E)$, *alors* $\mathsf{tsnRPNI}(E) = A$.

Preuve : On sait que $\mathsf{tRPNI}(\mathsf{pn}(E)) = A$. *Il faut montrer que* $\mathsf{tRPNI}(\mathsf{pn}(E)) = \mathsf{tsnRPNI}(E)$. *Comme ces algorithmes sont identiques au test de validation des fusions près, il faut vérifier qu'à chaque étape de l'apprentissage, le test de consistance entre A et* $\mathsf{pn}(E)$ *est équivalent au test de fonctionnalité de A.*

Remarquons tout d'abord que les fusions d'états successives de tRPNI augmentent *le langage reconnu par l'automate, et ne le réduisent jamais. Il en résulte que si à une étape t l'automate n'est pas fonctionnel, c'est-à-dire qu'il reconnaît deux arbres $t \times \beta$ et $t \times \beta'$ avec $\beta \neq \beta'$, alors quelles que soient les étapes d'apprentissage suivantes il ne sera pas fonctionnel à la fin de l'apprentissage.*

Si E contient $\mathsf{char}_{\mathsf{tRPNI}}(A)$, *alors* $\mathsf{tRPNI}(pn(E)) = A$ *et donc puisque A est fonctionnel, d'après la remarque précédente, cela implique qu'à chaque étape d'apprentissage, lorsqu'une fusion est acceptée, alors l'automate est fonctionnel.*

Réciproquement, si à une étape d'apprentissage une fusion n'est pas acceptée, cela veut dire qu'il existe un exemple négatif $t \times \beta'$ dans $\mathsf{pn}(E)$ *qui est reconnu par A. Par définition de* pn, *cela implique qu'il existe β tel que $\beta \neq \beta'$ et $t \times \beta \in E$, donc $t \times \beta$ est reconnu par A, car A est complet. Donc A n'est pas fonctionnel.*

Sous les hypothèses que nous avons décrites, le test de consistance sur $\mathsf{pn}(E)$ *et le test de fonctionnalité de A sont donc équivalents. Donc* $\mathsf{tRPNI}(\mathsf{pn}(E)) = \mathsf{tsnRPNI}(E)$. ◄

Ce lemme semble nous amener naturellement à réduire les résultats d'apprenabilité de tRPNI à un résultat d'apprenabilité de tsnRPNI. Cependant, il ne s'applique que pour les TSN complets. Il n'est pas possible d'obtenir un résultat identique pour les TSN en général, car si un TSN n'est pas complet, alors il est possible que son ensemble caractéristique pour tRPNI contienne des exemples négatifs qu'on ne puisse pas induire à partir d'exemples complètement annotés.

Or, appliqué à un échantillon quelconque, tsnRPNI ne génère pas nécessairement de TSN complets. Ce lemme n'est donc pas directement suffisant pour réduire l'identification par données fixées à partir d'exemple positifs et négatifs avec tRPNI en identification par données fixées à partir d'exemples complètement annotés avec tsnRPNI.

Pour résoudre ce problème, nous allons redéfinir la relation d'équivalence et la relation de consistance associées à tsnRPNI. Rappelons que chaque instance du modèle d'apprentissage par donnée fixées est associée à une relation d'équivalence et une relation de consistance. Dans le cas de tRPNI, la relation d'équivalence est : $A \equiv A' \Leftrightarrow \mathsf{langage}(A) = \mathsf{langage}(A')$, et la relation de consistance est $t \in A \Leftrightarrow t \in \mathsf{langage}(A)$. Pour tsnRPNI, nous allons redéfinir ces relations ainsi : deux TSN sont équivalent si ils décrivent la même requête, et un arbre annoté est consistant avec un TSN si son annotation est celle de la requête décrite par le TSN. Formellement, cela donne les définitions suivante de l'équivalence et de la consistance (voir 3.1.2) :

Définition 4.2. *La relation d'équivalence entre deux* TSN *est définie ainsi : Soit A et A' deux* TSN *sur Σ.*

$$A \equiv_{tsnRPNI} A' \Leftrightarrow \mathsf{complete}(\mathsf{langage}(A)) = \mathsf{complete}(\mathsf{langage}(A'))$$

Définition 4.3. *La relation de consistance entre un exemple et un* TSN *est définie ainsi : Soit $t \times \beta$ un arbre de* $\mathsf{T}_{\Sigma \times \mathsf{Bool}}$, *et A un* TSN *sur Σ.*

$$t \times \beta \in_{tsnRPNI} A \Leftrightarrow t \times \beta \in \mathsf{complete}(\mathsf{langage}(A))$$

Cela résout notre problème, car on peut maintenant considérer que tsnRPNI apprend des TSN dans le cas général, et en particulier apprend les représentants complets de leur classe d'équivalence quand on lui fournit des échantillons caractéristiques.

Nous allons maintenant définir la fonction caractéristique associée à tsnRPNI, c'est-à-dire la fonction associant à chaque TSN son échantillon caractéristique pour tsnRPNI. Si A est un TSN complet, alors il s'agit du plus petit ensemble d'exemples complètement annotés induisant implicitement l'échantillon caractéristique de A pour tRPNI. Si A n'est pas un TSN complet, il s'agit de l'échantillon caractéristique du TSN complet représentant sa classe d'équivalence.

Définition 4.4. *La fonction caractéristique de* tsnRPNI *est définie ainsi :*

$$\begin{aligned}\mathsf{char}_{\mathsf{tsnRPNI}}(A) \quad = \quad & \{t \times \beta \in \mathsf{langage}(A') | \\ & \exists A' \ \exists (t \times \beta', b) \in \mathsf{char}_{\mathsf{tRPNI}}(A') \ (\mathsf{complete}(\mathsf{langage}(A)) = \mathsf{langage}(A'))\}\end{aligned}$$

Propriété 4.1. *Les* TSN *sont identifiables par données fixées à partir d'exemples complètement annotés, en utilisant la fonction d'identification* tRPNI*, la fonction de caractérisation* $\mathsf{char}_{\mathsf{tsnRPNI}}$*, la relation d'équivalence* $\equiv_{tsnRPNI}$ *et la relation de consistance* $\in_{tsnRPNI}$*.*

Preuve : Appelons $\mathsf{class}_{\mathsf{TSN}}$ *l'ensemble des* TSN *sur* Σ*,* $\mathsf{echantillons}_{\mathsf{TSN}}$ *l'ensemble des échantillons, c'est-à-dire l'ensemble des sous-ensembles de langages de* TSN*.*

Il y a plusieurs éléments à montrer :

- tsnRPNI *est une fonction de* $\mathsf{echantillons}_{\mathsf{TSN}}$ *dans* $\mathsf{class}_{\mathsf{TSN}}$*. La sortie de* tsnRPNI *est un automate déterministe sur* $\Sigma \times \mathsf{Bool}$*, et elle est fonctionnelle, car sa fonctionnalité est vérifiée à chaque étape.*
- $\mathsf{char}_{\mathsf{tsnRPNI}}(A)$ *est toujours consistant avec* A*, pour* A *élément de* $\mathsf{echantillons}_{\mathsf{TSN}}$*.*
  Cela se déduit de façon immédiate des définitions de $\mathsf{char}_{\mathsf{tsnRPNI}}$ *et de* $\in_{tsnRPNI}$*.*
- *Pour tout échantillon* E *et tout* TSN A*, si* $E \in_{tsnRPNI} A$ *et si* $\mathsf{char}_{\mathsf{tsnRPNI}}(A) \subseteq E$*, alors* $\mathsf{tsnRPNI}(E) \equiv_{tsnRPNI} A$*.*
  Soit A *un* TSN *et* E *un échantillon tel que* $E \in_{tsnRPNI} A$ *et* $\mathsf{char}_{\mathsf{tsnRPNI}}(A) \subseteq E$*. Soit* A' *représentant complet de la classe d'équivalence de* A*, c'est-à-dire* TSN *sur* Σ *tel que* $\mathsf{complete}(\mathsf{langage}(A)) = \mathsf{langage}(A')$*. Remarquons que* $\mathsf{char}_{\mathsf{tRPNI}}(A') \subseteq \mathsf{pn}(\mathsf{char}_{\mathsf{tsnRPNI}}(A'))$*, donc que* $\mathsf{char}_{\mathsf{tRPNI}}(A') \subseteq \mathsf{pn}(E)$*. Le lemme 4.1 nous indique que* $\mathsf{tsnRPNI}(E) = A'$*, donc* $\mathsf{tsnRPNI}(E) \equiv_{tsnRPNI} A$*.*
- *La fonction* tsnRPNI *se calcule en temps polynomial par rapport à la taille de l'échantillon* S*.*
  Chaque test de fonctionnalité s'effectue en temps polynomial (voir théorème 3.1), et il y en a au plus $|\mathsf{etats}(A)|$ *à effectuer, avec* A *automate préfixe de* S*.*
- *La taille de* $\mathsf{char}_{\mathsf{tsnRPNI}}(A)$ *est polynomiale par rapport à* A*.*
  La taille de $\mathsf{char}_{\mathsf{tsnRPNI}}(A)$ *est majorée par la taille de* $\mathsf{char}_{\mathsf{tRPNI}}(A)$*, qui elle-même est polynomiale en la taille de* A*.*

◄

Théorème 4.2. *Les requêtes monadiques dans les arbres représentées par des* TSN *sont polynomialement identifiables par données fixées à partir d'exemples complètement annotés.*

Preuve : Chaque requête monadique correspond à une classe d'équivalence de TSN *au sens où on l'a définie. Ce théorème se déduit donc directement de la propriété 4.1.* ◄

4.3 Apprentissage des requêtes représentées par des TSNe

Nous allons maintenant nous intéresser à l'apprentissage de TSNe, c'est-à-dire d'automates d'arbres décrivant des requêtes sur des arbres élagués. L'apprentissage de TSNe à partir d'un ensemble d'apprentissage va se faire en deux phases :

- L'élagage de chaque arbre de l'échantillon, selon une heuristique donnée indépendante de l'apprentissage.
- L'apprentissage du TSNe à partir de l'ensemble d'arbres élagués, selon un algorithme proche de l'algorithme d'apprentissage des TSN déjà étudié.

Une fois le TSNe appris, on pourra l'utiliser pour calculer la requête qu'il définit dans les arbres non-élagués, comme on l'a vu dans la section 3.4.3.

L'objectif est donc le même que pour l'apprentissage de TSN : on apprend un automate à partir d'un échantillon d'entrée composé d'arbres annotés, et on utilise cet automate pour annoter d'autres arbres. Cependant, l'apprentissage de TSNe s'avère être beaucoup plus efficace que l'apprentissage de TSN. Il y a trois raisons à cela :

- Une même requête peut être beaucoup plus simple à exprimer avec un TSNe déterministe qu'avec un TSN déterministe (voir section 3.4.1).
- L'élagage permet de cibler les éléments significatifs pour la requête avant l'apprentissage, ce qui diminue d'autant le travail de discrimination des éléments significatifs implicitement effectué par l'apprentissage.
- L'apprentissage peut se faire à partir d'exemples partiellement annotés, ce qui est beaucoup plus intéressant dans la pratique.

En revanche, l'apprentissage de TSNe pose deux problèmes majeurs :

- L'efficacité de l'algorithme dépend énormément de l'heuristique d'élagage. Nous verrons dans le détail les caractéristiques que celle-ci doit avoir pour que l'apprentissage fonctionne correctement.
- Nous n'avons pas de résultats théoriques d'apprenabilité des TSNe, car la classe de requêtes que nous pouvons apprendre dépend de l'heuristique d'élagage utilisée. Cette dépendance reste à étudier.

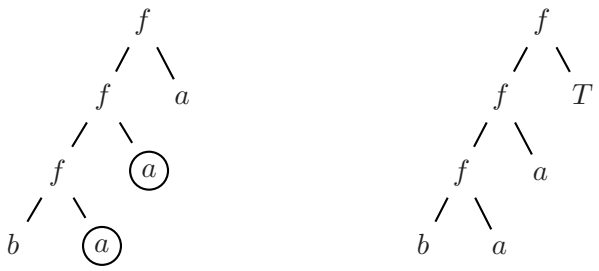
4.3.1 Apprentissage à partir d'exemples partiellement annotés

L'idée la plus évidente pour apprendre des TSNe est de commencer par élaguer les arbres de l'ensemble d'apprentissage, puis d'appliquer sur les arbres élagués un algorithme similaire à tsnRPNI en remplaçant le test de fonctionnalité par un test de pseudo fonctionnalité.

Cependant, alors que tsnRPNI garantissait la consistance entre l'ensemble d'apprentissage et le TSN appris, l'adaptation de tsnRPNI aux TSNe que nous venons de décrire ne la garantit pas. En effet, il est possible qu'un noeud appartenant à une branche élaguée d'un arbre soit sélectionné par un TSNe obtenu au cours du processus d'apprentissage.

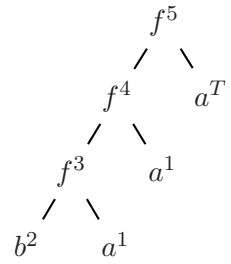
Exemple 4.3. *Nous allons prendre l'exemple d'un apprentissage à partir d'un unique arbre élagué.*

Voici cet arbre et l'élagage à partir duquel nous allons effectuer notre apprentissage :



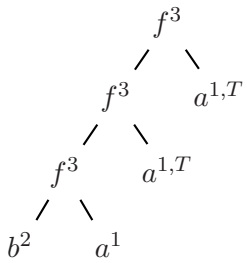
Au début de l'apprentissage, le TSNe *ne reconnaît que l'arbre élagué de l'ensemble d'apprentissage. Voici l'automate initial ainsi que son unique run sur l'arbre de l'ensemble d'apprentissage :*

$$
\begin{array}{rcl}
\mathsf{etats}(A) & = & \{1,2,3,4,5,T\} \\
\mathsf{finaux}(A) & = & \{5\} \\
\mathsf{regles}(A) & = & \{a \to 1, b \to 2, T \to T, \\
 & & f(2,1) \to 3, f(3,1) \to 4, f(4,T) \to 5\}
\end{array}
$$

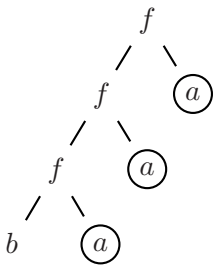


Supposons maintenant que la procédure d'apprentissage fusionne les états 3 *et* 4*, puis* 3 *et* 5*. L'automate obtenu, qui est toujours pseudo-fonctionnel, est :*

$$
\begin{array}{rcl}
\mathsf{etats}(A) & = & \{1,2,3,T\} \\
\mathsf{finaux}(A) & = & \{3\} \\
\mathsf{regles}(A) & = & \{a \to 1, b \to 2, T \to T, \\
 & & f(2,1) \to 3, f(3,1) \to 3, f(3,T) \to 3\}
\end{array}
$$



L'annotation correspondant au run de cet automate sur l'arbre de départ est donc :



Il est possible de résoudre le problème en ajoutant un test de consistance après chaque fusion déterministe, en plus du test de pseudo-fonctionnalité. Le principe du test de consistance est simple : pour chaque arbre $t \times \beta$ de l'ensemble d'apprentissage, on calcule l'annotation β' de t par le TSNe courant, et si β' est différent de β, alors la fusion est refusée.

D'un autre point de vue, il est possible d'utiliser cette capacité de l'algorithme à généraliser en sélectionnant des noeuds dans les parties élaguées pour faire de l'apprentissage à partir d'exemples partiellement annotés. Dans l'exemple précédent, on peut voir l'arbre d'entrée comme un arbre partiellement annoté, et le résultat de la fusion comme une généralisation de l'annotation partielle de l'arbre d'entrée.

Si l'on considère un arbre sur $t \times \beta$ comme un arbre partiellement annoté, alors se pose le problème de l'absence d'information négative apportée par cet arbre. En effet, lorsqu'on considère $t \times \beta$ comme un arbre complètement annoté, pour chaque noeud π, $\beta(\pi)$ nous indique qu'un noeud est annoté, si il vaut vrai, ou qu'il ne l'est pas, si il vaut faux. Lorsqu'on considère $t \times \beta$ comme une annotation partielle, alors il n'est plus possible de spécifier qu'un noeud ne doit pas être sélectionné. Pour tout π tel que $\beta(\pi) = \text{faux}$, on ne sait pas si π est sélectionné ou ne l'est pas.

Pour résoudre ce problème nous allons donner une définition plus riche des arbres partiellement annotés. Une annotation partielle d'un arbre sera défini par deux ensembles de noeuds disjoints : les noeuds annotés positivement (c'est-à-dire qui doivent être sélectionnés par la requête) et les noeuds annotés négativement (c'est-à-dire qui ne doivent pas être sélectionnés par la requête). Les noeuds de l'arbre qui ne seront présent ni dans un ensemble ni dans l'autre seront les noeuds dont on ne sait pas s'il sont sélectionnés ou s'il ne le sont pas.

Pour pouvoir apprendre des TSNe à partir de ce type d'arbres partiellement annotés, nous allons reprendre l'idée du test de consistance que nous avons précédemment exposé, en limitant la vérification aux noeuds spécifiés comme non sélectionnés. Une fusion sera refusée quand le TSNe obtenu sélectionnera dans les arbres de l'ensemble d'apprentissage un noeud spécifiquement indiqué comme non sélectionné. Nous reprendrons les choses formellement lors de la description de l'algorithme.

4.3.2 Élagage des arbres

Dans l'apprentissage des TSNe, l'heuristique d'élagage devient une partie déterminante pour l'efficacité de l'algorithme. Idéalement, l'élagage doit :

- Conserver les éléments significatifs pour la requête.
- Élaguer les éléments non significatifs pour la requête.
- Élaguer les parties non annotées de l'arbre. En effet, l'algorithme ne peut sélectionner des noeuds supplémentaires que dans les zones élaguées.

Il s'agit d'une tâche difficile. Dans un certain sens, en imposant de telles contraintes, on peut penser qu'on a reporté les difficultés implicites de l'apprentissage (quelle généralisation effectuer ?) dans l'heuristique d'élagage (quelle partie des arbres conserver ?).

Il n'est donc pas possible d'espérer atteindre facilement cet "élagage idéal". Devant la difficulté du problème notre choix s'est porté sur des heuristiques simples et génériques, qui ne répondent pas toujours de façon satisfaisante aux critères que nous avons établis, mais qui se sont avérées relativement efficaces dans la pratique. Nous détaillerons ces heuristiques dans la partie expérimentale de notre travail.

4.3.3 Algorithme d'apprentissage

L'algorithme d'apprentissage des TSNe, que nous allons appeler $\mathsf{tsn_eRPNI}$, est une variante de tsnRPNI.

Tout d'abord, l'algorithme prend en entrée des arbres partiellement annotés $< t, p^+, p^- >$, avec p^+ ensemble de noeuds annotés positivement et p^- ensemble de noeuds annotés négativement. L'apprentissage va se faire sur les *annotations minimales* correspondant à ces annotations partielles, c'est-à-dire sur les arbres annotés ainsi définis :

$$\mathsf{annot_{min}}(< t, p^+, p^- >) = \{t \times \beta | \forall \pi \ ((\pi \in p^+) \Leftrightarrow (\beta(\pi) = \mathsf{vrai}))\}$$

Ensuite, ces arbres annotés vont être élagués par une fonction d'élagage qui n'est pas spécifiée ici.

Enfin, le test de fonctionnalité est remplacé par un test de pseudo-fonctionnalité (voir partie 3.4.4), assorti d'un test de consistance (voir partie 4.3.1), dont le rôle est de vérifier que les noeuds de chaque ensemble p^- ne sont pas sélectionnés par le TSNe courant :

$$\mathsf{consist}(A, S) = (\forall < t, p^+, p^- > \in S \forall \pi \in p^- \ (\mathsf{annot_A(t)}(\pi) = \mathsf{faux}))$$

Voici l'algorithme obtenu :

---

tsnRPNI (S)

$S' \leftarrow \{\mathsf{annot_{min}}(elagage(t)) | t \in S\}$
Soit A l'automate préfixe de S'
Soit l une séquence ordonnée des paires d'éléments de $\mathsf{etats}(A)$
PourTout (q_1, q_2) de l
    $A' \leftarrow \mathsf{fusion\_deterministe}(A, q_1, q_2)$
    Si A' est pseudo-fonctionnel
    Et $\mathsf{consist}(A', S)$ Alors
        $A \leftarrow A'$
Sortie : A

L'algorithme $\mathsf{tsn_eRPNI}$ présente la caractéristique intéressante de se comporter exactement comme tsnRPNI lorsqu'on utilise une fonction d'élagage qui n'élague rien. On peut donc le considérer comme une extension de tsnRPNI, qui présente le double avantage d'avoir une capacité de généralisation supérieur du fait de l'élagage, et d'être capable de fonctionner à partir de document partiellement annotés. C'est grâce à ce dernier point que nous allons pouvoir utiliser $\mathsf{tsn_eRPNI}$ dans un cadre interactif, dans lequel l'utilisateur pourra effectuer une annotation partielle qui sera complétée par le programme.

4.4 Apprentissage actif des TSNe

Les modèles d'apprentissage que nous avons présentés dans la partie 4.1 sont des modèles dans lesquels l'apprentissage se fait par observation de données fournies à l'algorithme. En ap-

prentissage actif, ce ne sont pas les données qui sont fournies à l'algorithme, mais l'algorithme qui interroge une source de données, qu'on appelle "un oracle".

Il est clair que l'apprentissage actif présente un cadre plus proche de la réalité d'un apprentissage de requête par interaction avec un utilisateur, comme dans le logiciel Squirrel que nous allons présenter dans le chapitre 5.

4.4.1 Un modèle pour l'apprentissage actif de requêtes

Nous allons ici commencer par introduire un modèle classique d'apprentissage actif en inférence grammaticale, le modèle MAT [Angluin, 1987]. Puis, nous introduirons une variante de ce modèle spécifiquement adaptée à l'apprentissage de requêtes.

Dans le modèle MAT, l'algorithme effectue une succession de questions à un oracle, et détermine ainsi la cible. Les questions posée à l'oracle peuvent être de deux types :

- Question d'appartenance : l'algorithme d'apprentissage propose un objet, et l'oracle répond vrai si l'objet est consistant avec la cible, faux sinon.
- Question d'équivalence : l'algorithme d'apprentissage propose une cible potentielle, et l'oracle répond vrai si il s'agit bien de la cible, et faux sinon. Dans ce cas, l'oracle propose un contre-exemple.

Il est montré dans [Sakakibara, 1990, Drewes and Hogberg, 2003] que les automates d'arbres déterministes sont identifiables dans ce modèle d'apprentissage.

Nous allons proposer un modèle d'apprentissage qui est une variante de ce modèle adapté à l'apprentissage de requêtes. La différence essentielle tient dans la possibilité de demander à l'oracle de corriger une erreur d'annotation. Les questions que l'algorithme peut poser à l'oracle dans ce modèle sont les suivantes :

- Question de correction d'annotation (QCA) : l'algorithme d'apprentissage propose un arbre annoté, et l'oracle répond vrai si l'annotation est correcte. Dans le cas contraire, l'oracle répond faux et indique un noeud pour lequel l'annotation n'est pas correcte.
- Question d'équivalence (QE) : l'algorithme d'apprentissage propose une requête, et l'oracle répond vrai si il s'agit bien de la requête cible. Dans le cas contraire, l'oracle répond faux et propose un arbre que la requête proposée et la requête cible n'annotent pas de la même manière.

4.4.2 Un algorithme d'apprentissage actif de TSNe

L'algorithme que nous présentons ici, que nous appellerons Squirrel, du nom de l'outil l'implémentant que nous avons développé, se place dans le cadre de l'apprentissage de requêtes par questions présenté en 4.4.1. Dans l'application que nous présenterons par la suite, l'oracle sera l'utilisateur, qui répondra aux questions du programme à travers l'interface du logiciel.

L'idée générale de l'algorithme est de constituer un ensemble d'apprentissage pour l'algorithme $\mathsf{tsn_e}$RPNI (voir 4.3.1) en posant des questions du type QE ou QCA définies en 4.4.1 . Cet ensemble d'apprentissage est constitué d'un ensemble de n arbres complètement annotés que nous noterons S, et d'un unique arbre partiellement annoté, que nous appellerons arbre courant et que nous noterons $< t, p^+, p^- >$. Chaque étape d'apprentissage consiste à apprendre

un TSNe sur l'ensemble d'apprentissage courant, à le tester sur l'arbre courant, à interroger l'oracle et à enrichir l'ensemble d'apprentissage en fonction de la réponse de l'oracle.

Plus précisément, à chaque étape l'algorithme va interroger l'oracle par une QCA sur le TSNe courant (qui est vide au départ) et l'arbre courant (qui est fournit par QE sur un TSNe vide). Si l'oracle répond que l'annotation est incorrecte, il va fournir un noeud de l'arbre pour lequel l'annotation est fausse, ce qui va permettre d'enrichir l'annotation de l'arbre courant et de recommencer le processus. Si l'oracle répond que l'annotation est correcte, alors l'arbre courant est un arbre complètement annoté qui peut être ajouté à l'ensemble des arbres complètement annotés. On interroge alors l'oracle par une QE pour savoir si l'apprentissage est terminé. Si il ne l'est pas, l'arbre courant devient l'arbre fourni par l'oracle et on recommence le processus.

Squirrel

$A \leftarrow$ TSNe vide
$S \leftarrow \emptyset$
TantQue QE$(A) \neq$ vrai
    $t \leftarrow sortie$(QE)
    $p^+ \leftarrow \emptyset$
    $p^- \leftarrow \emptyset$
    TantQue QCA(annot$_\mathsf{A}$(t)) $\neq$ vrai
        Soit $\pi_{err} = sortie$(QCA)
        Si annot$_\mathsf{A}$(t)$(\pi_{err}) =$ vrai
            Alors $p^- \leftarrow p^- \cup \{\pi_{err}\}$
            Sinon $p^+ \leftarrow p^+ \cup \{\pi_{err}\}$
        $A \leftarrow$ tsn$_\mathsf{e}$RPNI$(S \cup \{< t, p^+, p^-\})$
Sortie : A

---

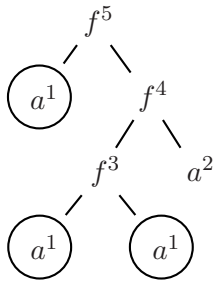
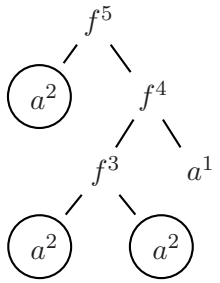
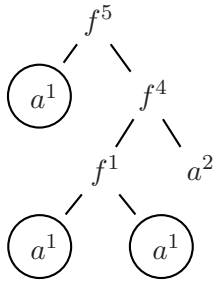
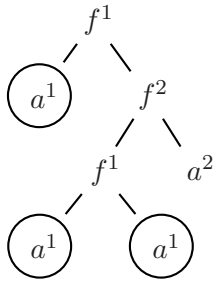
1. Tout d'abord on construit l'automate initial. Il s'agit du plus grand automate déterministe émondé reconnaissant exactement l'échantillon caractéristique.	$(a, \mathsf{F}) \to 1$ $(a, \mathsf{V}) \to 2$ $(f, \mathsf{F})(2, 2) \to 3$ $(f, \mathsf{F})(3, 1) \to 4$ $(f, \mathsf{F})(2, 4) \to 5$ final : $\{5\}$	
2. On tente de fusionner 1 et 2. L'automate obtenu n'est pas fonctionnel (il est facile de voir qu'on pourrait indifféremment étiqueter chacune des feuilles par V ou F). On revient donc à l'automate de l'étape 1.	$(a, \mathsf{F}) \to 1$ $(a, \mathsf{V}) \to 1$ $(f, \mathsf{F})(1, 1) \to 3$ $(f, \mathsf{F})(3, 1) \to 4$ $(f, \mathsf{F})(1, 4) \to 5$ final : $\{5\}$	
3. On fusionne 1 et 3. L'automate reste fonctionnel.	$(a, \mathsf{F}) \to 1$ $(a, \mathsf{V}) \to 2$ $(f, \mathsf{F})(2, 2) \to 1$ $(f, \mathsf{F})(1, 1) \to 4$ $(f, \mathsf{F})(2, 4) \to 5$ final : $\{5\}$	
4. On fusionne 4 et 2. Pour que l'automate reste déterministe, on doit également fusionner 5 et 1. L'automate reste fonctionnel et plus aucune fusion n'est possible, donc l'algorithme se termine.	$(a, \mathsf{F}) \to 1$ $(a, \mathsf{V}) \to 2$ $(f, \mathsf{F})(2, 2) \to 1$ $(f, \mathsf{F})(1, 1) \to 2$ final : $\{1\}$	

Figure 4.1: L'algorithme tsnRPNI appliqué à un échantillon ne contenant qu'un exemple. La requête inférée est : sélectionner toutes les feuilles d'étiquette a et de profondeur impaire.

Chapitre 5

Application à l'extraction d'information dans les pages HTML

Jusque là, nous nous sommes intéressés à la définition, puis à l'apprentissage des TSN, objets représentant des requêtes monadiques dans les arbres. Ce chapitre est consacré à l'application des algorithmes que nous avons définis au problème concret de l'apprentissage de requêtes dans les pages HTML.

Dans un premier temps, nous allons introduire l'ensemble des techniques que nous avons développées pour appliquer efficacement nos algorithmes à des données réelles. On peut diviser ces travaux en deux parties : le pré-traitement des données et les heuristiques accompagnant les algorithmes. Dans un deuxième temps, nous présenterons Squirrel, le prototype d'outil d'apprentissage de requêtes interactif que nous avons développé.

5.1 Apprentissage de requêtes dans des pages HTML

L'aspect arborescent des pages HTML est facile à percevoir : la syntaxe HTML, fondée sur l'utilisation de balises, structure l'ensemble d'un document en éléments imbriqués les uns dans les autres et donc induit naturellement une représentation arborescente. Nous appellerons cette représentation l'arbre DOM en référence à l'API DOM de gestion des documents XML et HTML qui travaille sur ce type de représentation.

Malgré cela, l'application de notre algorithme à l'inférence de requêtes dans des pages HTML ne va pas être immédiate. La difficulté majeure à laquelle nous allons être confrontés sur des données réelles est le manque d'exemples dont nous allons disposer. Notre objectif étant de permettre à un utilisateur de générer des requêtes en annotant manuellement des pages, notre programme doit être capable d'apprendre correctement une requête à partir d'un nombre très restreint de documents, voire à partir d'un seul document.

Or, pour un document annoté donné, il existe de nombreuses requêtes régulières différentes consistantes avec cette annotation. Parmi ces requêtes, certaines définissent des tâches d'extraction intuitivement pertinentes, et d'autres non. Or rien dans nos algorithmes ne nous laisse supposer que les requêtes qui semblent les plus pertinentes dans le cadre de l'extraction d'information ont plus de chance d'être apprises que les autres.

Pour obtenir de bons résultats, nous allons chercher à guider notre algorithme d'apprentissage grâce à des heuristiques spécifiques à notre tâche intervenant à différents niveaux :

- Au moment de l'abstraction des documents HTML en arbres, nous allons essayer d'épurer ces documents, d'obtenir des arbres qui représentent la structure du document sans contenir d'information inutile.
- Au moment de l'élagage des arbres, nous allons essayer d'éliminer au maximum les parties de cette structure qui ne sont généralement pas utiles aux requêtes dans les pages HTML.
- Au moment de l'apprentissage, nous allons choisir un ordre des fusions favorisant les généralisations qui semblent correspondre à celles qu'on retrouve le plus souvent dans la définition de requêtes dans les pages HTML.

Chacune de ces heuristiques contient un certain nombre de choix subjectifs qui dépendent de l'intuition que l'on se fait des tâches d'extraction d'information les plus courantes. Le problème de ces choix est qu'ils ne peuvent être justifiés que de manière qualitative. En effet, pour pouvoir effectuer des tests statistiquement valables de comparaison entre heuristiques ou entre paramétrages d'heuristiques, il faudrait disposer d'un corpus de documents suffisamment grand pour être statistiquement représentatif des tâches d'extractions dans les pages HTML, ce qui n'est pas possible, ou du moins hors de notre portée. Nous reviendrons sur les problèmes de constitution de corpus dans le chapitre suivant.

Nous allons maintenant détailler chacune des étapes dans lesquelles interviennent nos heuristiques.

5.1.1 Des pages HTML aux arbres

La première tâche à effectuer avant de pouvoir appliquer nos algorithmes à des pages HTML est de transformer ces pages en arbres à arité arbitraire. Ceci est réalisé par un pré-traitement ayant deux objectifs :

- transformer les arbres DOM par un traitement des attributs en arbres à arité arbitraire, c'est-à-dire en termes sur un alphabet à arité arbitraire fini Γ ;
- épurer les informations qu'ils contiennent, de manière à faciliter le travail des algorithmes d'apprentissage.

Nous allons effectuer ce pré-traitement en deux phases : la première traitera les informations aux noeuds des arbres, la deuxième effectuera un filtrage de noeuds. La Fig. 5.1 donne un exemple de document HTML sur lequel nous allons travailler tout au long de ce chapitre. La Fig. 5.2 illustre le pré-traitement de ce document, que nous allons maintenant détailler.

Le traitement des informations aux noeuds

Les arbres DOM sont constitués de deux types de noeuds :

- Des noeuds éléments, qui sont caractérisés par le nom de leur balise et associés à une liste de paires $(attribut, valeur)$.
- Des noeuds de données, qui contiennent une chaîne de caractère de taille quelconque.

Lorsque l'on doit transformer ces arbres DOM en arbres définis sur un alphabet fini se pose le problème de ce qu'on va faire des informations associées à chaque noeud. Pour chaque

```
<HTML xmlns="http://www.w3.org/1999/xhtml">
  <HEAD>
    <TITLE>Un exemple de page Web</title>
    <META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
  </HEAD>
  <BODY>
    <H1> Un exemple de page Web </H1>
    <P id="résumé">
      <FONT>
        Cette page est constituée d'un cours résumé et d'un ensemble de
        données dans un tableau. Nous allons inférer une <I>requête</I> qui
        extrait l'ensemble des adresses e-mail de chacun.
      </FONT>
    </P>
    <TABLE id="données">
      <TR>
        <TD align="left"> <B>Gilleron</B> Rémi </TD>
        <TD align="center"> 0312345678</TD>
        <TD align="right"> gilleron@univ-lille3.fr </TD>
      </TR>
      <TR>
        <TD align="left"> <B>Carme</B> Julien </TD>
        <TD align="center"> <I>0320748646</I> </TD>
        <TD align="right"> carme@univ-lille3.fr </TD>
      </TR>
      <TR>
        <TD align="left"> Lemay Aurélien </TD>
        <TD align="center"> 0387654321</TD>
        <TD align="right"> email:<I>lemay@univ-lille3.fr</I> </TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

Figure 5.1: Un exemple de document HTML. L'arbre DOM de ce document est présenté Fig. 5.2

information, il y a deux possibilités : soit on l'ignore, soit on l'encode dans un symbole de l'alphabet sur lequel on va travailler.

Si on conserve trop d'information, l'alphabet sera trop grand (voire infini pour les attributs à valeur réelle) et les possibilités de généralisation trop restreintes. La section 5.1.3 détaille les difficultés d'apprentissage à partir d'un faible nombre d'exemples, et il est facile de comprendre au vu de ces explications que l'augmentation de la taille de l'alphabet aggrave les choses.

Si on conserve trop peu d'information, on risque de supprimer un élément crucial pour définir correctement la requête. Par exemple, si on veut extraire toutes les feuilles dont le père

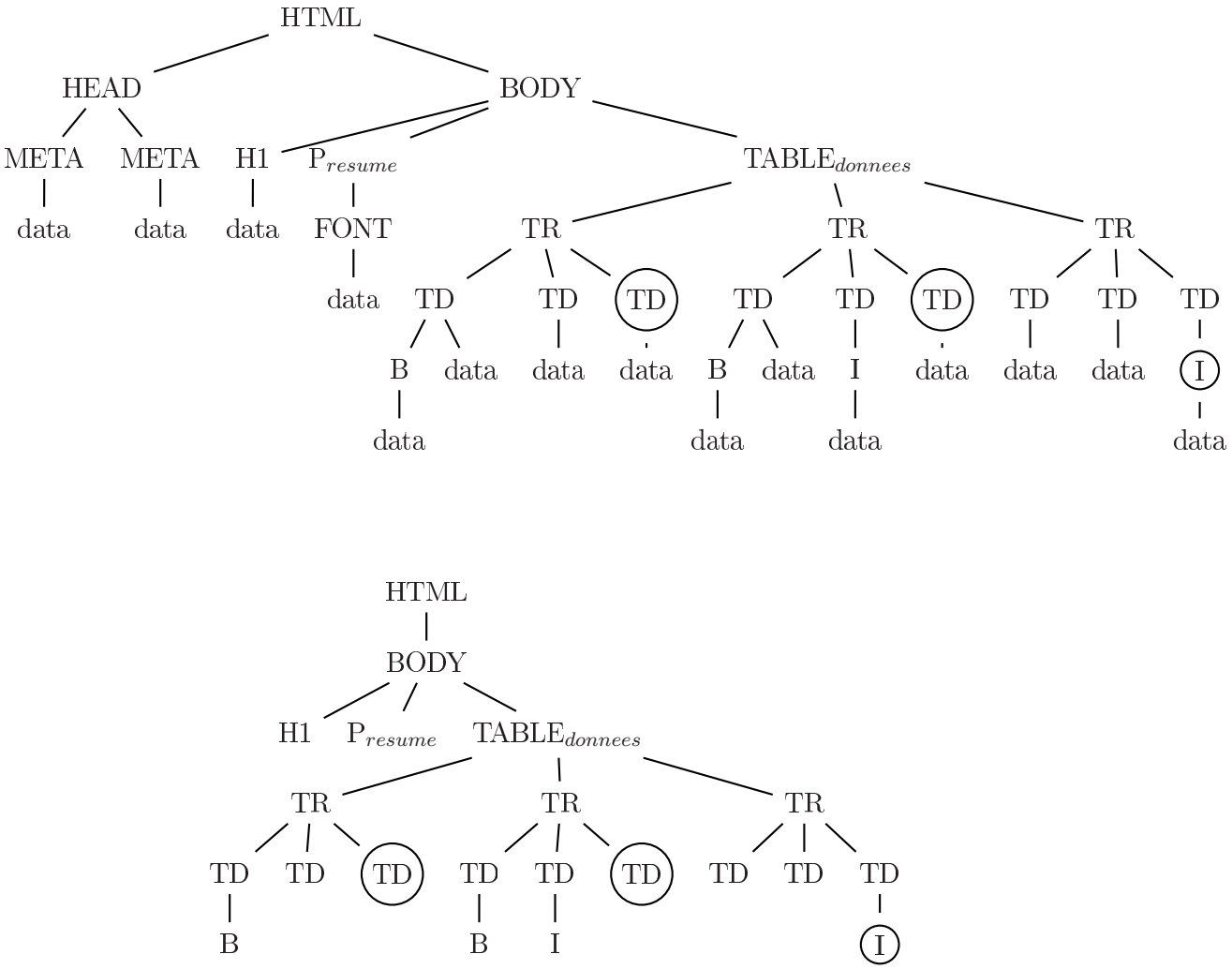


Figure 5.2: En haut, l'arbre DOM correspondant au document de la Fig. 5.1 après pré-traitement des noeuds. En bas, le même arbre après pré-traitement sur la structure, prêt à être donné en entrée à notre algorithme.

possède l'attribut *color* et qu'on applique un pré-traitement qui supprime les attributs, on a déjà échoué avant même de commencer l'apprentissage.

Il n'y a pas de solution simple à ce problème. Dans la mesure où l'objet de nos travaux est de démontrer la pertinence d'une approche de l'extraction d'information fondée sur la structure des documents, il est apparu raisonnable de garder un minimum d'informations aux étiquettes. Voici les règles que nous nous sommes fixées :

- Les noms de balises sont conservés.
- Les attributs sont oubliés, sauf **CLASS** et **ID**.
- Les contenus textuels sont oubliés.

Ainsi, l'alphabet sur lequel nous travaillerons sera composé de l'ensemble des symboles s où s est une balise présente dans l'échantillon, de l'ensemble des symboles s_i où s est une balise présente dans l'échantillon et i est une valeur de l'attribut **ID** ou **CLASS**, et enfin du symbole *data* caractérisant les noeuds de données. La Fig. 5.2 (en haut) fournit l'arbre défini sur cet alphabet correspondant au document de la Fig. 5.1.

Le filtrage des noeuds

Le but de ce filtrage est de simplifier les arbres en supprimant les noeuds que nous considérons comme n'apportant aucune information structurelle pertinente. Nous voulons obtenir un arbre dont la structure reste celle du document, mais qui contiennent le moins de noeuds possible, pour des raisons d'efficacité.

Pour cela, nous allons utiliser deux règles de filtrage :

- Nous allons supprimer l'ensemble des noeuds de données. Les noeuds de données ne structurent pas le document : ce sont des feuilles qui contienne une information textuelle. Comme nous ne conservons pas cette information textuelle, les noeuds de données ne nous sont plus utiles.
- Nous allons supprimer les noeuds éléments correspondant à des balises non structurantes. En effet, parmi les balises HTML, certaines structurent le document (**H1**, **H2**, **TABLE**, **UL** ...), alors que d'autre contiennent des informations non structurelles (**HEAD**, **META**,**SCRIPT** ...). Nous avons fait le choix de conserver les balises suivantes : **HTML**, **BODY**, **H1**, **H2**,**H3**,**H4**,**H5**,**H6**, **P**, **UL**, **OL**, **LI**, **TABLE**, **TR**, **TD**, **A**, **DIV**, **SPAN**.
  Dans le cas où la tâche d'extraction concerne une balise qui ne fait pas partie de cette liste, alors cette balise est ajoutée à la liste pour le traitement de cette tâche.

Pour la suppression d'un noeud interne les fils du noeud supprimés sont remontés au niveau de son père en respectant l'ordre.

5.1.2 L'élagage des arbres

Comme nous l'avons vu dans le chapitre précédent, l'apprentissage de TSNe dépend d'une heuristique d'élagage, dont le rôle est de supprimer les parties des arbres non significatives pour la requête.

Le principal problème dans la conception de ce type d'heuristique est qu'elle se fait de manière empirique, en observant des documents et des tâches d'extraction associées. Le risque est alors de concevoir des heuristiques ad-hoc, efficaces pour les documents observés mais trop spécialisées. Pour limiter ce risque nous avons développé une heuristique simple et générique. Elle se fonde sur l'idée que l'essentiel de l'information importante pour la requête se situe à proximité du chemin menant de la racine aux noeuds sélectionnés. Cette technique se rapproche des techniques de fenêtrage utilisée pour l'inférence de wrappers textuels dans [Chidlovskii, 2001] et [Freitag and Kushmerick, 2000].

Elle consiste à garder l'ensemble des noeuds situés sur les chemins menant aux noeuds sélectionnés, ainsi que les descendants de ces noeuds jusqu'à une profondeur n. Comme la notion de chemin ne se préserve pas correctement lors du passage de l'arbre à arité arbitraire

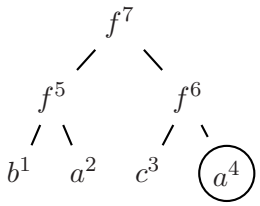
au codage de construction, l'élagage se fait sur l'arbre à arité arbitraire, avant l'encodage en arbre de construction.

La figure 5.3 montre l'arbre de la figure 5.2 élagué selon cette heuristique avec $n = 0$, ainsi que l'arbre de construction correspondant. La figure 5.4 montre l'élagage du même arbre avec $n = 1$. Il est important de noter que même pour $n = 0$, l'information restante sur l'arbre n'est pas réduite aux chemins menant aux noeuds sélectionnés, car l'ensemble des symboles T sur l'arbre élagué indique la position de chaque noeud restant parmi ses frères respectifs, ainsi que les embranchements entre les différents chemins menant aux noeuds sélectionnés. Dans la pratique, le choix de n ne peut être laissé à l'utilisateur et les tentatives de choix heuristique de n ne se sont pas avérées concluantes. Au sein de l'outil que nous avons développé, nous nous sommes fixés sur le choix $n = 0$. Ce choix est le meilleur avec la représentation des arbres actuellement choisie (non prise en compte des attributs autres que ID et CLASS, non prise en compte des contenus textuels). Nous n'avons pas, en effet, trouvé en pratique de cas pour lequel le choix $n = 1$ fournissait de meilleurs résultats. Cependant, le choix du paramètre d'élagage n sera à revoir lorsque nous étendrons l'expressivité du système en conservant des contenus textuels qui pourraient être pertinents pour l'extraction. C'est le cas lorsque l'on souhaite extraire une feuille lorsque la feuille à sa gauche contient un texte particulier, par exemple "Name :".

5.1.3 Ordre des fusions

Le choix d'un ordre de fusion satisfaisant est primordial pour obtenir de bonnes généralisations. Comme on ne dispose que d'un petit nombre d'exemples, il est fréquent qu'une fusion qui aurait été refusée si on avait eu suffisamment d'exemples ne le soit pas. Même si la généralisation qui en découle ne pose pas de problème, il est possible qu'elle empêche par la suite une généralisation nécessaire pour définir correctement la requête. Ce cas de figure est illustré par l'exemple suivant :

Exemple 5.1. *Supposons que nous essayions d'inférer la requête "Sélectionner les noeuds d'étiquette a qui sont frères droits de noeuds d'étiquette c". On dispose de l'arbre annoté suivant :*



Dans l'ordre ascendant de fusion, l'une des premières fusions tentée sera celle des états 1 *et* 3. *Rien ne s'oppose à celle-ci, et une fois ces deux états fusionnés, il sera impossible d'inférer la requête attendue, car b et c joueront alors le même rôle.*

Pour inférer cette requête, il faudrait, par exemple, disposer également de l'arbre suivant :

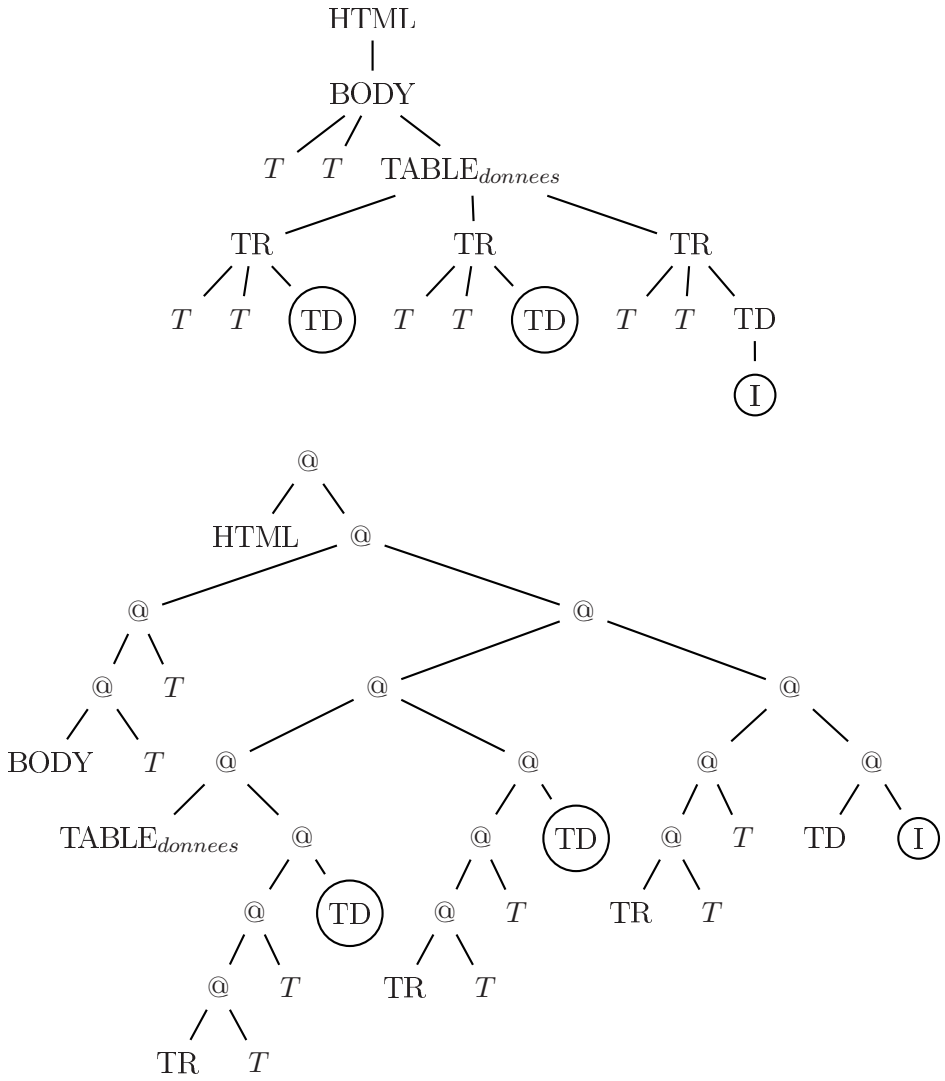
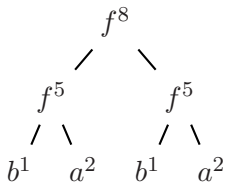


Figure 5.3: Élagage de l'arbre de la Fig. 5.2 avec une profondeur d'élagage $n = 0$ (en haut), on rappelle que le symbole T correspond à un sous-arbre élagué. L'arbre de construction correspondant, qui va nous servir pour l'apprentissage (en bas).



Une fusion des états 1 *et* 3 *devient alors impossible car elle induirait un automate non-fonctionnel : le deuxième arbre serait reconnu par l'automate, que la feuille a la plus à droite soit sélectionnée ou non. En déroulant l'algorithme complètement, on vérifierait que c'est bien la requête cible qui est inférée.*

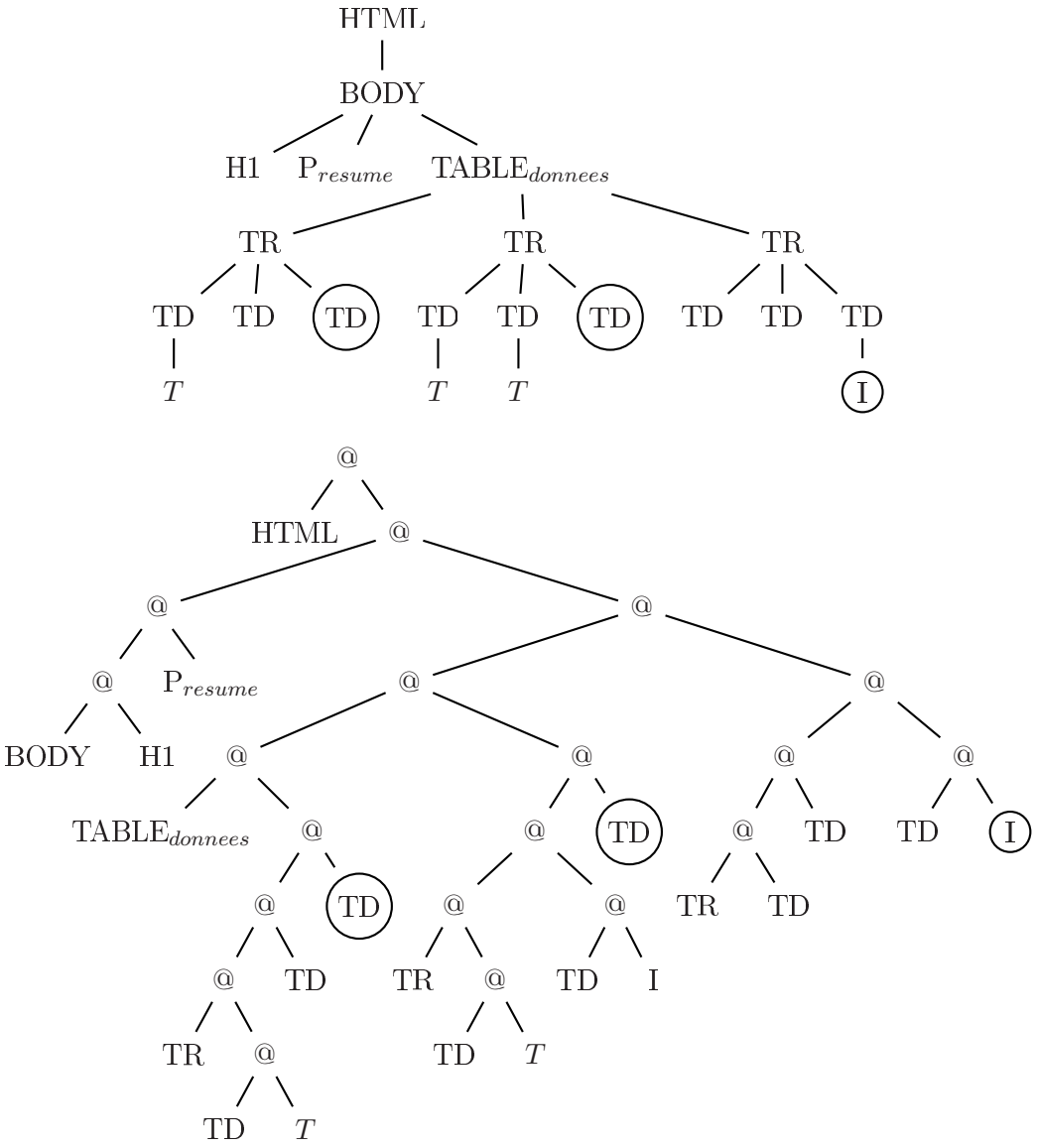
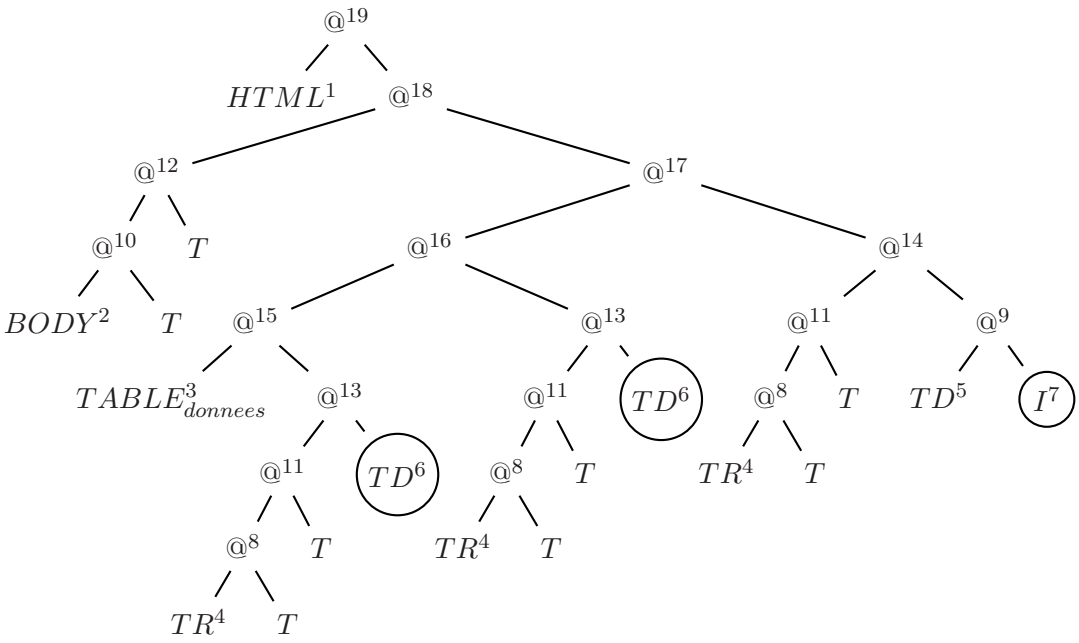


Figure 5.4: Élagage de l'arbre de la Fig. 5.2 avec une profondeur d'élagage $n = 1$ (en haut). L'arbre de construction correspondant (en bas).

Il est facile de comprendre qu'un ordre judicieux des fusions d'états peut améliorer les choses quand on a peu d'exemples. Si on commence par faire les fusions les plus pertinentes, alors on limite les risques qu'une généralisation non pertinente bloque par la suite une généralisation nécessaire à l'inférence de la requête cible. Il s'agit d'un procédé classique pour améliorer les algorithmes par fusion d'états [Lang et al., 1998].

L'heuristique que nous allons présenter n'est pas fondée sur des travaux existants, car à notre connaissance, ceux-ci ont toujours portés sur des variations de RPNI dans les mots, et non dans les arbres. De plus, elle est fortement spécifique à notre tâche, c'est-à-dire à l'inférence de requêtes dans les pages HTML.



Étape de fusion des états	Groupes d'états à fusionner
1	{13, 14}
2	{5,9}, {4,8,11,13}, {4,8,11,15}, {3,15,16,17}, {2,10,12,18}, {1,19}

Figure 5.5: Les différentes étapes de l'apprentissage d'un TSNe à partir de l'arbre élagué de la Fig. 5.3. Chaque noeud de cet arbre de construction est associé ici à un état de l'automate préfixe (en exposant). Le tableau présente les groupes d'états au sein desquels les fusions sont tentées, dans l'ordre, c'est-à-dire que toutes les fusions possibles sont tentées au sein de {13,15}, puis de {5,9}, etc... L'étape 3 n'est pas précisée, car elle consiste simplement à tenter les fusions d'état restantes. L'automate résultant est présenté Fig. 5.2.

Nous allons diviser la succession de fusions d'états en trois phases, illustrée sur un exemple Fig. 5.5 :

Dans la première phase, nous allons fusionner certains états qui correspondent dans l'ensemble d'apprentissage à des sous-arbres que nous considérons comme *interchangeables* selon une règle heuristique simple. L'idée de cette règle est que deux sous-arbres dont les racines sont soeurs dans l'arbre HTML et contenant au moins un élément sélectionné sont *interchangeables*, c'est-à-dire que leur rôles sont similaires pour définir une requête. Cette règle est illustrée par la fusion des états 13 et 14 dans l'exemple de la Fig. 5.5. Cela induit une limitation dans l'expressivité des requêtes que nous pouvons apprendre, mais il s'agit d'une limitation tout-à-fait raisonnable. Par exemple, il n'est plus possible de définir une requête qui sélectionne les noeuds d'étiquette A dans les sous-arbres de la racine de rang pair et les noeuds d'étiquette B dans les sous-arbres de la racine de rang impair. En revanche, il est toujours possible de tenir compte du rang pour décider si il faut ou non sélectionner un noeud. Par exemple, il est

toujours possible de définir une requête qui sélectionne les noeuds A dans les sous-arbres de la racine de rang pair et seulement dans ces sous-arbres là.

Dans une deuxième phase, nous allons procéder par ordre croissant de hauteur des sous-arbres correspondant aux états dans l'automate préfixe, mais en se limitant aux fusions correspondant aux généralisations *horizontales*, c'est-à-dire aux fusions entre états associés à des noeuds frères dans les arbres HTML de l'ensemble d'apprentissage. Cette phase est illustrée par la fusion des états 4,8, 11 et 13, ou des états 3, 15, 16 et 17 dans l'exemple de la Fig. 5.5.

Replaçons nous dans le cadre des TSN à pas. Prenons un sous-arbre quelconque $(f, b)(t_1, \ldots, t_n)$ d'un arbre de notre ensemble d'apprentissage. Le codage de construction de ce sous-arbre est $(\ldots(((f, b)@^a t_1)@^a t_2)\ldots@^a t_n)$. Appelons $q_0 \ldots q_n$ l'ensemble des états associés à (f, b), $((f, b)@^a t_1)$, $(((f, b)@^a t_1)@^a t_2)$, $\ldots$, $(\ldots(((f, b)@^a t_1)@^a t_2)\ldots@^a t_n)$. Toute fusion entre deux états q_i et q_j correspond exactement à une généralisation *horizontale*, c'est-à-dire à une généralisation du langage des sous-arbres acceptés en dessous d'un noeud évalué en q_0, ou encore du langage de la *haie* associé à q_0, dans le vocabulaire des automates à haie. D'une manière générale, ce sont ces généralisations qui nous intéressent en priorité. En effet, dans l'immense majorité des cas, les variations entre les documents d'une même famille porte sur la présence ou non de tel ou tel sous-arbre, sur le nombre et l'emplacement de tel ou tel type de sous-arbre, et les généralisations permettant d'inférer des requêtes valables sur des documents présentant ce type de variations sont des généralisations horizontales.

Nous allons donc former des ensembles d'états (non disjoints) correspondant aux "étages" des arbres de l'ensemble d'apprentissage, et tenter les fusions d'état au sein de chacun de ces groupes, dans l'ordre croissant de hauteur de ces étages. Cette heuristique correspond bien aux documents HTML et XML où les récursions horizontales sont plus fréquentes que les récursions verticales.

La troisième phase consiste à essayer de faire des fusions d'état entre les états restants, une fois la deuxième phase terminée. Dans la pratique, ces fusions n'apportent pas de généralisations intéressantes, et le TSNe obtenu gagne en lisibilité quand on ne les effectue pas.

Exemple 5.2. *Nous allons terminer cette partie en donnant un exemple de* TSNe *inférée par notre algorithme en utilisant les techniques d'abstraction et d'élagage décrites. La Fig. 5.2 montre le* TSNe *généré à partir de l'unique document de la Fig. 5.1 en utilisant l'élagage de la Fig. 5.3, et selon les fusions décrites dans la figure 5.5.*

Rappelons que chaque transition $a \xrightarrow{b} c$ s'interprète : un sous-arbre évalué en a qu'on étend avec un sous-arbre évalué en b s'évalue en c. Ainsi, ce TSNe *s'interprète : La balise* ***TD*** *est sélectionnée si c'est une feuille, que elle est la dernière fille de* ***TR****, et qu'elle est au bout du chemin* ***HTML, BODY, TABLE, TR, TD****. La balise* ***I*** *est sélectionné si c'est une feuille, si elle est la dernière fille de* ***TD****, qui elle-même est la dernière fille de* ***TR****, et si elle est au bout du chemin* ***HTML, BODY, TABLE, TR, TD, I****.*

Même si il s'agit d'un exemple simple, il est intéressant de remarquer que la requête inférée est vraisemblablement celle qu'un utilisateur aurait intuitivement spécifiée à partir du document fourni en exemple.

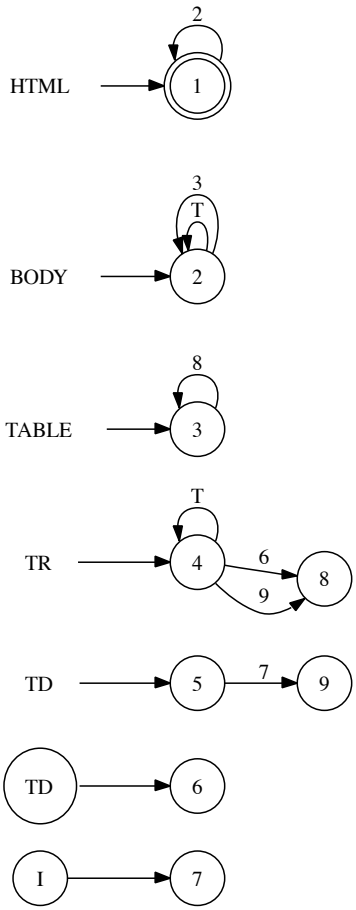


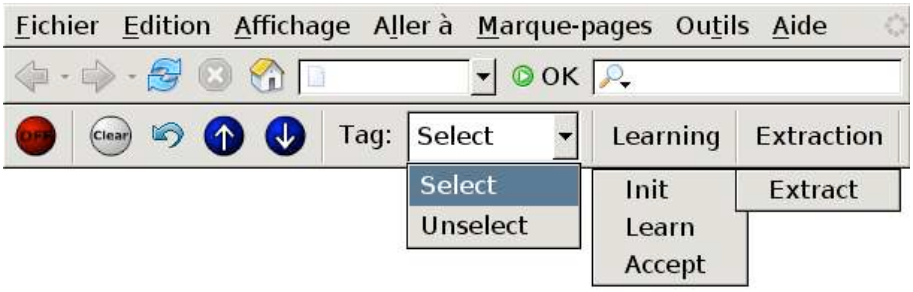
Figure 5.6: Le TSNe obtenu par application de tsn_eRPNI avec l'élagage 5.3 et selon les étapes d'apprentissage décrites dans la figure 5.5. La sémantique de ce TSNe est détaillée dans l'exemple 5.2.

5.2 Squirrel : un outil interactif d'apprentissage de requêtes

5.2.1 Description de Squirrel

Squirrel est un prototype d'outil d'extraction d'information par apprentissage interactif de requêtes que nous avons développé à partir des algorithmes présentés dans cette thèse. Embarqué au sein du navigateur Web Firefox, il permet à un utilisateur de générer des requêtes dans les documents HTML en quelques clics de souris à partir de quelques exemples. Une fois la requête générée, il est possible de l'appliquer à d'autres pages pour vérifier sa validité. L'utilisation pratique des requêtes générées, comme la constitution automatique de bases de données à partir de sites Web, n'a pas été implantée.

Squirrel se présente comme une barre d'outils ajoutée à Firefox :



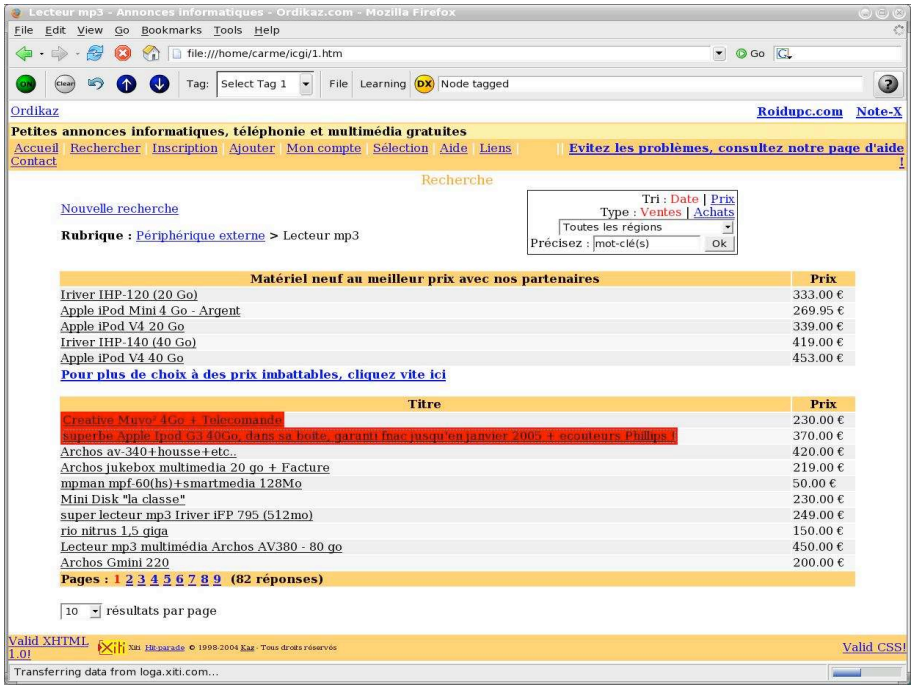
Nous allons commencer par décrire les différentes commandes de Squirrel, avant d'illustrer son fonctionnement sur un exemple.

L'interface de Squirrel donne la possibilité à l'utilisateur d'effectuer les opérations suivantes :

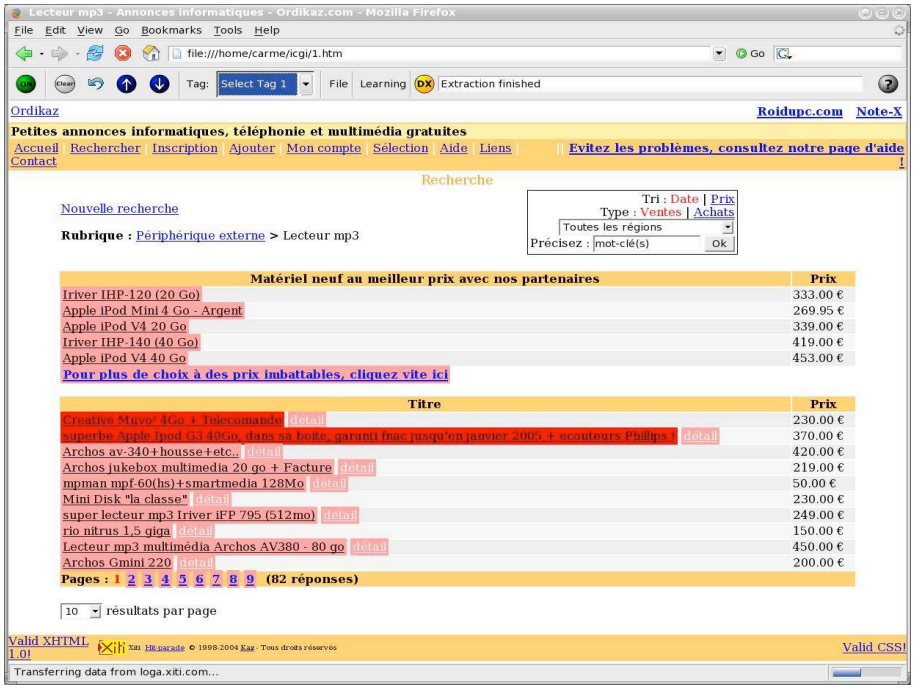
- Démarrer l'apprentissage d'une requête (bouton *Init*). La requête courante est alors réinitialisée.
- Sélectionner (par un simple clic de souris) des éléments dans la page Web courante, en mode *select* (exemple positif) ou *unselect* (exemple négatif).
- Lancer l'apprentissage (bouton *Learn*). L'apprentissage se fait sur l'annotation partielle de la page courante et sur l'ensemble des annotations complètes déjà validées. Il est suivi d'un calcul de la requête apprise sur la page courante.
- Valider l'annotation courante (bouton *Accept*). L'annotation courante est alors considérée comme complète et sauvegardée, elle pourra ensuite être utilisée pour continuer l'apprentissage.
- Appliquer la requête courante sur la page courante (bouton *Extract*). La requête courante est calculée sur la page courante.

Exemple 5.3. *Prenons l'exemple de l'apprentissage d'une requête sélectionnant une liste de noms de produit sur un site Web de vente par correspondance. Chaque page contient deux listes, l'une pour les produits neufs, l'autre pour les produits d'occasion, et nous ne voulons sélectionner que les produits d'occasion.*

L'utilisateur commence par sélectionner un nombre quelconque d'éléments à extraire dans une page.

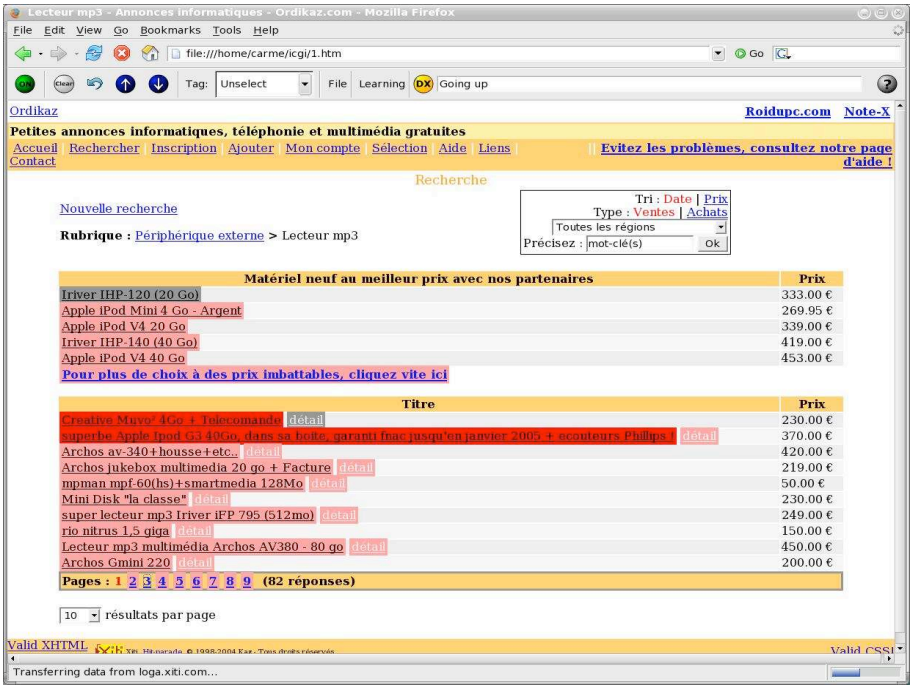


Ensuite, l'utilisateur lance la procédure d'apprentissage, qui s'effectue à partir de cette unique page partiellement annotée. Une fois la procédure d'apprentissage terminée, la requête inférée est automatiquement lancée sur cette même page.

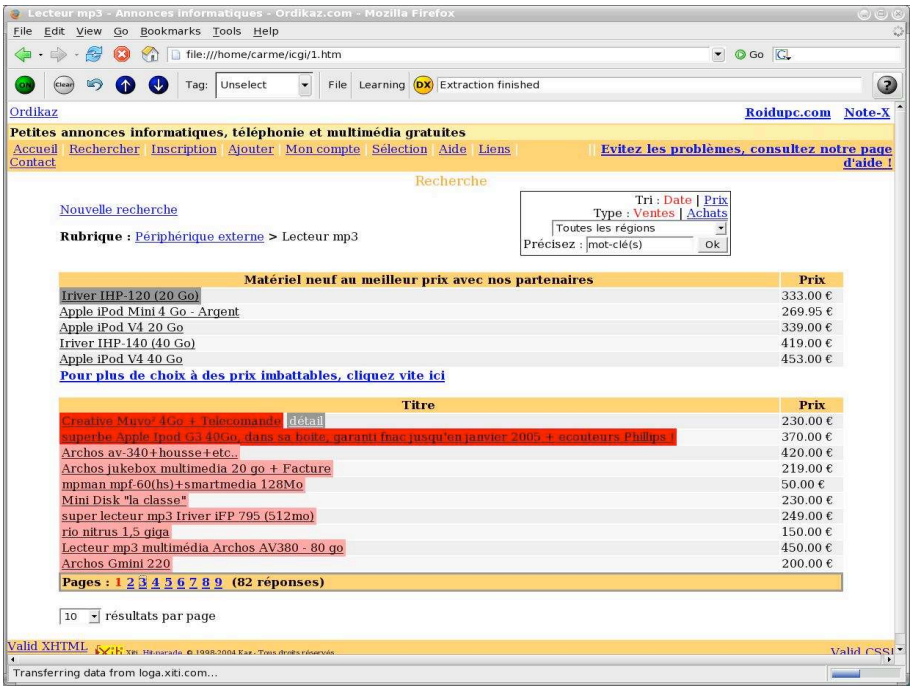


L'utilisateur constate que le résultat n'est pas satisfaisant, car certains éléments ont été sélectionnés alors qu'ils n'auraient pas dû l'être. L'utilisateur peut alors enrichir l'annotation

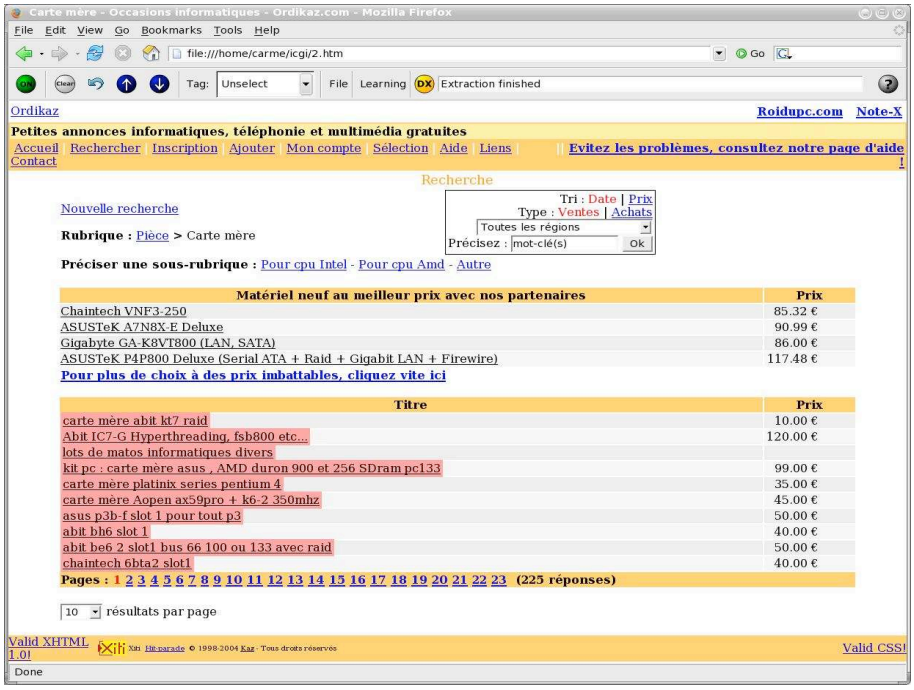
partielle en ajoutant des annotations négatives, *c'est-à-dire en spécifiant des éléments comme étant* non sélectionnés.



Il relance alors la procédure d'apprentissage. La requête inférée est alors lancée sur la page.



L'utilisateur constate alors que la requête inférée a correctement sélectionné tous les éléments voulus de la page. Il peut alors tester la requête apprise sur une autre page.



A chaque fois que le résultat de la requête n'est pas satisfaisant, l'utilisateur peut recommencer la procédure d'apprentissage que l'on vient de décrire. Le programme disposera alors d'un ensemble d'exemples complètement annotés (les pages sur lesquelles l'interaction est arrivée à son terme) et d'un unique exemple partiellement annoté (la page courante).

5.2.2 Fonctionnement de Squirrel

L'algorithme de fonctionnement de l'outil est Squirrel, décrit en 4.4.2. L'interaction entre l'utilisateur et le programme peut aisément s'interpréter du point de vue du programme comme une succession de réponses à des questions QCA et QE telles qu'elles sont décrites en 4.4.2 :

- La première annotation partielle fournie par l'utilisateur correspond à la réponse à la QE initiale.
- Chaque information supplémentaire (positive ou négative) fournie par l'utilisateur correspond à une réponse à une QCA. Dans la pratique, l'utilisateur peut faire plusieurs corrections à la fois, ce qui est un petit écart par rapport au modèle.
- Une fois la page courante correctement annotée, le programme pose implicitement une QE à l'utilisateur. Soit celui-ci décide d'arrêter l'apprentissage, ce qui correspond à une réponse positive, soit celui-ci trouve une page sur laquelle la requête ne se comporte pas correctement, ce qui correspond à une réponse négative.

Il y a tout de même une différence majeure entre l'algorithme Squirrel et le fonctionnement du programme : en pratique, l'utilisateur ne sait pas quand la requête est correcte. Il arrête l'apprentissage lorsqu'il estime qu'il a vérifié la validité de la requête courante sur un nombre de pages suffisant.

5.2.3 Vers un choix automatisé des documents

Dans Squirrel, les pages sur lesquelles les requêtes sont apprises et testées sont choisies par l'utilisateur. On peut supposer que l'ordre de ces pages est dicté par l'organisation du site et non par un choix qui optimiserait l'apprentissage.

On pourrait imaginer d'utiliser Squirrel dans un cadre un peu différent, dans lequel le programme choisirait lui-même l'ordre dans lequel les pages sont présentées à l'utilisateur. En essayant de présenter en priorité les documents sur lesquels la requête courante semble ne pas se comporter correctement, le programme minimiserait le nombre total de documents à présenter à l'utilisateur pour apprendre une requête correcte.

Nous avons développé une heuristique qui tente de deviner les documents sur lesquels la requête ne se comporte pas correctement. Pour cela, nous partons de la supposition que pour une tâche d'extraction donnée, le nombre d'éléments à extraire est une fonction affine de la taille du document, en nombre de noeuds. Cette supposition repose sur l'idée que les documents sont souvent construits sur le schéma suivant : un ensemble d'éléments associés à chaque page (entête, pied de page, menus...), et un ensemble d'éléments associés à chaque éléments à extraire (ligne d'un tableau, paragraphe...). Ce modèle est bien évidemment simpliste, mais souvent, il n'est pas très éloigné de la réalité. Notons également que ce modèle fonctionne aussi quand le nombre d'éléments à extraire est constant.

Le choix de la page à présenter à l'utilisateur se fait par le test de Grubbs [Grubbs, 1 21]. On évalue le nombre d'éléments extraits par la requête courante dans chaque document, on calcule la droite de régression linéaire des points de coordonnées (x, y) où x est le nombre de noeuds dans le document et y est le nombre de noeuds sélectionnés par la requête dans le document. Enfin, on choisit le document correspondant au point le plus éloigné de cette droite.

Nous n'avons pas intégré cette heuristique au sein de notre outil, pour une raison pratique simple : le programme ne dispose pas au départ d'un ensemble de pages, c'est l'utilisateur qui les lui fournit au fur et à mesure de l'apprentissage. Cependant, pour mettre en évidence l'intérêt de ce type de technique dans des travaux futurs, nous avons implanté cette heuristique dans le cadre non réaliste où le programme dispose d'un corpus de pages Web. Les résultats expérimentaux correspondants sont décrits dans le chapitre suivant. Une perspective sera d'étendre Squirrel pour qu'il intègre cette heuristique avec en entrée une description des URLs sur lesquelles le wrapper généré doit s'appliquer. Il faudra donc intégrer des capacités de navigation et de récupération de contenus.

Chapitre 6

Évaluation expérimentale du système d'apprentissage de requêtes

Ce chapitre présente les expériences que nous avons effectuées pour évaluer les performances de notre programme.

Nous allons d'abord présenter le corpus de documents annotés à partir duquel nous allons effectuer nos expériences. Ensuite nous présenterons et analyserons les résultats obtenus par notre programme sur ce corpus selon deux jeux d'expériences : l'un évaluant les performances d'un algorithme par une méthode standard de validation, et l'autre l'évaluant dans les conditions de l'apprentissage interactif.

Le premier jeu d'expériences à pour objectif de montrer que notre programme obtient de bonnes performances, égales ou supérieures aux autres programmes d'extraction d'information. Le deuxième jeu d'expériences va nous permettre de montrer que le nombre d'interactions nécessaires pour apprendre une requête est suffisamment faible pour que Squirrel soit utilisable dans la pratique.

6.1 Corpus

La constitution d'un corpus est un problème particulièrement difficile en extraction d'information. En effet, le seul moyen de constituer un corpus est d'annoter manuellement des documents. Ce travail a déjà été fait dans d'autres travaux, mais le nombre de jeux de données disponibles reste limité. Pour nos expériences, nous allons utiliser ces jeux de données existants et des jeux de données que nous avons construits à partir de sites usuels.

6.1.1 Corpus existant

Le seul corpus librement disponible adapté à nos expériences est le corpus RISE [2], constitué par Muslea afin de comparer les systèmes de génération de wrappers ou d'extraction d'information. Ce corpus est divisé en différents jeux de données. Chaque jeu de données est

[2] http ://www.isi.edu/info-agents/RISE/repository.html

constitué d'un ensemble de pages souvent très similaires, appartenant à un même site, et est associé à une ou plusieurs tâches d'extraction dans ces pages.

Un grand nombre de ces jeux de données ne peuvent pas être utilisés par nos algorithmes, car même si les documents sont des pages HTML, l'information à extraire n'est pas délimitée par des balises, et donc les tâches d'extraction définies ne sont pas des tâches de sélection de noeuds. Typiquement, l'extraction d'un mot donné dans une phrase est un problème qui sort du cadre de nos travaux.

Nous avons donc utilisé les jeux de données restants :

- 18 des jeux de données utilisés par Nicholas Kushmerick lors de la conception du système d'extraction d'information Wien. Lorsque Kushmerick a écrit sa thèse sur l'inférence de wrappers, il a expérimenté ses algorithmes sur un corpus d'une centaine de sites Web choisis aléatoirement. Ceux-ci ont ensuite été réutilisés dans d'autres travaux sur l'inférence de wrappers. Muslea, notamment, en a sélectionné une trentaine pour ses propres expériences, ceux qu'il a considérés comme les plus difficiles. Nous avons choisi parmi cette sélection de Muslea les 18 jeux de données pour lesquels les tâches d'extraction associées sont des tâches de sélection de noeuds. Chaque jeu de données est constitué de 8 pages du même site, annotées pour une à quatre tâches d'extraction. Dans les résultats, ces jeux de données sont notés S-XX
- Bigbook, un ensemble de 250 pages d'un annuaire. Les tâches associées sont l'extraction des noms et celles de adresses dans chaque page.
- Okra, un ensemble de 250 pages de sortie d'un moteur de recherche de personnes. Les tâches associées sont l'extraction des noms, des adresses mail et des scores dans chaque page.

6.1.2 Corpus créé

Pour compléter ce corpus, nous avons constitué un ensemble de jeux de données à partir d'un certain nombre de sites Web connus [3]. Dans la mesure du possible, nous avons cherché à choisir des sites présentant des tâches d'extraction variées et suffisamment complexes :

- Yahoo Directories. Chaque page contient un ensemble de liens pour un thème donné. Le jeu de données que nous avons constitué contient 80 pages. La tâche que nous avons définie est l'extraction de ces liens. La difficulté de cette tâche réside dans la variabilité de la structure générale de chaque page : chacune d'entre elles est divisée en rubriques : thèmes connexes, sélection des liens les plus populaires, liens donnés par ordre alphabétique ... Seuls les liens de cette dernière rubrique nous intéressent, mais leur position dépend de la présence ou non des autres rubriques.
- Google. Chaque page contient dix paragraphes, contenant chacun un lien vers un site extérieur et un paragraphe résumant ce site. Notre jeu de données est constitué de 40 pages. La tâche est d'extraire les liens vers les sites. La difficulté réside dans la confusion possible entre ces liens et d'autres liens annexes les précédant ou les suivant : conversion HTML, PDF, cache...
- Ebay. Chaque page contient un tableau, chaque ligne du tableau contenant les informations relatives à un produit. Notre jeu de données est constitué de 40 pages. La tâche

[3]Corpus disponible à l'adresse suivante : http ://www.grappa.univ-lille3.fr/ carme/corpus.html

est d'extraire le nom du produit dans chacune de ces lignes. La mise en page de chacune de ces lignes varie d'un produit à l'autre.
- New York Times. Chaque page contient un article. Notre jeu de données est constitué de 40 pages. Ici, il n'y a qu'un élément à extraire dans chaque page : le corps du texte de l'article.
- Citeseer. Chaque page contient un ensemble d'informations relatives à un article. Notre jeu de données est constitué de 40 pages. Les élément à extraire sont les citations. Ces élément sont regroupés en éléments consécutifs d'un sous-ensemble de liste HTML.

6.2 Expériences à partir de pages annotées

6.2.1 Protocole

Nous allons commencer par évaluer nos algorithmes en travaillant à partir de documents complètement étiquetés par validation croisée. Ce protocole, classique en apprentissage supervisé, va nous permettre de comparer nos algorithmes avec d'autres méthodes d'extraction d'information dans des pages Web.

Le principe de ces expériences est le suivant : on travaille à partir de jeux de données constitués d'un ensemble de pages Web étiquetées pour une tâche d'extraction déterminée. On divise chaque jeu de données en n parties, et on effectue n expériences successives, dans lesquelles on constitue un *ensemble d'apprentissage* à partir de $n-1$ parties, et un *ensemble de test* à partir de la partie restante. Plus précisément, pour la i^{me} expérience, on utilise les parties $1, \ldots, i-1, i+1, \ldots n$ comme ensemble d'apprentissage, et la partie i comme ensemble de test.

Chaque expérience se déroule ainsi : on apprend une requête à partir de l'ensemble d'apprentissage, puis on applique la requête sur les pages de l'ensemble de test après avoir retiré leurs annotations. On compare alors les noeuds sélectionnés par la requête inférée aux noeuds sélectionnés dans les documents annotés, en mesurant les trois valeurs suivantes :
- VP, le nombre de vrais positifs, c'est-à-dire de noeuds sélectionnés par la requête inférée à juste titre.
- FP, le nombre de faux positifs, c'est-à-dire le nombre de noeuds sélectionnés alors qu'ils n'auraient pas du l'être.
- FN, le nombre de faux négatifs, c'est-à-dire le nombre de noeuds qui auraient du être sélectionnés et qui ne l'ont pas été

On peut ensuite évaluer la qualité de l'extraction en mesurant la *précision* (P), le *rappel* (R) et la *F-mesure* (F) :
- $P = \frac{VP}{VP+FP}$
- $R = \frac{VP}{VP+FN}$
- $F = \frac{2R.P}{R+P}$

Pour chaque jeu de données, on calcule la moyenne de ces valeurs pour l'ensemble des n expériences.

	Précision	Rappel	F-mesure
S-4	100	100	100
S-5	100	100	100
S-6	85	91	88
S-7	100	100	100
S-8	100	100	100
S-9	100	100	100
S-10	100	100	100
S-11	45	55	50
S-12	100	100	100
S-13	100	100	100
S-14	100	100	100
S-19	100	100	100
S-20	100	100	100
S-22	100	100	100
S-23	100	100	100
S-24	100	100	100
S-25	100	100	100
S-28	100	100	100
S-30	100	100	100
Okra	100	100	100
BigBook	100	100	100

Figure 6.1: Résultats de Squirrel sur le corpus WIEN.

6.2.2 Résultats

Les tableaux 6.1 et 6.2 montrent l'ensemble des résultats que nous obtenons pour l'ensemble de notre corpus. Dans le cas des jeux de données S-XX, les valeurs données sont des moyennes entre les différentes tâches d'extraction définies sur chaque jeu de données.

La comparaison de nos résultats avec d'autres programmes n'est pas facile, car il ne nous a pas été possible d'utiliser ces programmes et nous avons donc du utiliser les résultats publiés. Or les conditions expérimentales dans lesquelles ces résultats sont obtenus sont à chaque fois différentes et ne sont pas toujours clairement spécifiées.

Plutôt que de présenter des données précises mais incomparables, nous allons suivre l'approche de Muslea *et al.* dans [Muslea et al., 2002] et établir une comparaison qualitative des résultat de Wien, Stalker et Squirrel pour l'ensemble du corpus de Wien. Cette comparaison est présentée Fig. 6.3.

	Précision	Rappel	F-mesure
Yahoo	100	97.5	98.7
Google	100	100	100
NYTimes	100	100	100
E-Bay	100	100	100
CiteSeer	24	31	27

Figure 6.2: Résultats de Squirrel sur notre corpus

	Stalker	Wien	Squirrel
S-4	OK	OK	OK
S-5	OK	OK	OK
S-6	Approx	Echec	Echec
S-7	OK	Echec	OK
S-8	OK	OK	OK
S-9	Approx	Echec	OK
S-10	OK	OK	OK
S-11	Approx	Echec	Echec
S-12	OK	OK	OK
S-13	OK	OK	OK
S-14	OK	OK	OK
S-19	OK	OK	OK
S-20	OK	OK	OK
S-22	OK	OK	OK
S-23	OK	OK	OK
S-24	Approx	Echec	OK
S-25	OK	OK	OK
S-28	OK	OK	OK
S-30	OK	OK	OK
Okra	OK	OK	OK
BigBook	OK	OK	OK

Figure 6.3: Comparaison qualitative entre Stalker, Wien et Squirrel sur le corpus Wien. *Echec* signifie une F-mesure inférieure à 90%, *Approx* signifie une F-mesure strictement comprise entre 90% et 100%, et *OK* signifie une F-mesure égale à 100%

6.2.3 Analyse

Bilan

Ces résultats montrent clairement que notre programme présente des résultats tout à fait satisfaisants, comparables ou même supérieurs en performances à Wien et Stalker, les deux programmes auxquels nous avons pu nous comparer.

Au delà de cette constatation, il faut relativiser la portée de ces résultats, pour deux raisons essentielles.

Tout d'abord, le corpus RISE a été constitué en 1997, et ses pages ont une structure bien plus simple que celle des sites Web d'aujourd'hui. Dans de nombreux cas, les tâches d'extraction sont triviales lorsqu'on considère la structure, ce que fait Squirrel, mais que ne font ni Wien et Stalker. A mon sens, il faudrait renouveler l'ensemble du corpus standard de tests de programmes d'extraction d'information. Le corpus que nous avons créé va dans ce sens, en présentant des tâches bien plus réalistes et plus complexes.

Ensuite, la validation croisée est un protocole satisfaisant pour tester les algorithmes d'apprentissage supervisé en général, mais dans le cas de l'extraction d'information, il n'est peut-être pas le plus adapté. En effet, étant donné la forte similitude entre les structures des pages d'un même jeu de donnée, il est intuitivement clair que la tâche qui consiste à déduire les éléments à extraire sur 10% des pages à partir de 90% de pages complètement annotées est une tâche relativement facile.

La meilleure interprétation de chacune de ces expériences est la suivante : quand la F-mesure atteint 100, ou pratiquement 100, cela prouve que notre algorithme est capable d'apprendre la bonne requête, sans nous donner d'information satisfaisante sur le nombre d'exemples nécessaires à l'apprentissage. Quand la F-mesure est nettement inférieure à 100, cela nous montre que notre algorithme échoue, et ceci quel que soit le nombre d'exemples d'apprentissage. Nous allons maintenant détailler les causes de ces échecs.

Analyse des échecs

Nous avons obtenu de mauvais résultats sur trois jeux de données : S-6, S-11 et Citeseer. L'analyse de ces echecs nous a montré qu'ils illustrent parfaitement les problèmes potentiels de notre approche que nous avons déjà évoqués :

- Dans S-6 et S-11, les structures ne sont pas suffisantes pour définir correctement les requêtes. En effet, l'élément à extraire est caractérisé par un texte le précédant, et sa position dans la structure de la page varie. Dans S-6, les résultats semblent acceptables, car l'élément à extraire est souvent, mais pas toujours, à un emplacement donné.
- Dans CiteSeer, c'est l'élagage qui est en cause. Les éléments à extraire sont situés dans une liste et délimités par une balise particulière. Or notre élagage supprime cette balise, rendant impossible la définition d'une requête correcte dans les arbres élagués.

6.3 Expériences en mode interactif

6.3.1 Protocole

Le but des expérimentations que nous allons présenter ici est d'évaluer la quantité d'interactions nécessaires pour inférer une requête. Pour cela, nous allons interfacer l'outil interactif que nous avons présenté en 5.2 avec un programme simulant le comportement d'un utilisateur. Autrement dit, nous allons utiliser un programme qui fera office d'oracle en répondant aux questions QCA et QE de l'algorithme Squirrel (voir 4.4.2) à partir d'un jeu de données annotées. Rappelons que l'apprentissage se fait à partir d'exemples partiellement annotés, et que ces annotations partielles ne se font dans le modèle d'apprentissage actif qu'à travers les QCA : le nombre de QCA représente donc le nombre total d'éléments annotés ou corrigés.

Selon notre modèle d'apprentissage actif, l'oracle doit effectuer des choix lorsqu'il répond aux QCA et aux QE. Il nous faut donc spécifier le comportement de notre programme pour chacune de ces questions.

A chaque QCA posée à l'oracle, celui-ci doit répondre *oui* si la requête se comporte correctement sur la page courante, et *non* sinon. Comme notre oracle dispose d'un jeu de données complètement annotées contenant la page courante, cela ne pose pas de problème. En cas de réponse négative, l'oracle doit choisir un noeud pour lequel la requête renvoie une annotation fausse et renvoyer cette information au programme. Le choix que nous avons fait pour notre oracle est proche de celui que ferait un utilisateur : le noeud renvoyé est le premier noeud du document pour lequel l'annotation est fausse, dans l'ordre de lecture du document, que ce soit un faux positif (noeud sélectionné alors qu'il ne devrait pas l'être) ou un faux négatif (noeud non sélectionné alors qu'il aurait du l'être).

Pour les QE, le problème est un peu plus complexe. En effet, celui-ci doit répondre *oui* si la requête proposée par le programme est la requête cible, *non* dans le cas contraire. Or notre oracle ne dispose pas explicitement de cette requête cible, mais seulement d'un ensemble de données étiquetées. Nous allons créer deux oracles qui vont se comporter de manière différente dans leurs réponses aux QE :

- *total* va tester la requête sur chacune des pages du jeu de données jusqu'à ce qu'il en trouve une sur laquelle la requête ne se comporte pas correctement ou jusqu'à ce que l'ensemble du jeu de données ait été testé. Si la requête s'est comportée correctement sur toutes les pages, alors il répond *oui*. Sinon, il répond *non* et renvoie la page sur laquelle la requête s'est trompée, privée de son annotation. Il s'agit de l'approximation la plus précise que nous puissions faire du comportement d'un oracle qui connaîtrait la requête cible.
- *partiel* va tester la requête sur chacune des pages du jeu de données jusqu'à ce qu'il en trouve une sur laquelle la requête ne se comporte pas correctement ou jusqu'à ce que n pages aient été observées durant toute la procédure d'apprentissage. Si la requête s'est comportée correctement sur toutes les pages observées, alors il répond *oui*. Sinon, il répond *non* et renvoie la page sur laquelle la requête s'est trompée, privée de son annotation. Il s'agit d'un comportement qui se rapproche de celui d'un utilisateur, car dans la pratique, les utilisateurs disposent d'un grand nombre de pages mais interrompent l'apprentissage après avoir constaté un comportement correct de la requête sur un nombre de pages suffisant.

Ces deux oracles vont être utilisés dans des expériences différentes. L'oracle *total* va nous permettre d'estimer la quantité de questions QE et QCA nécessaires pour apprendre la bonne requête. L'oracle *partiel* va nous permettre d'estimer la qualité d'une requête apprise à partir d'un nombre donné de pages. Il nous suffira pour cela de calculer la F-mesure obtenue sur l'ensemble du jeu de données une fois l'apprentissage terminé.

Il reste un point à préciser : l'ordre dans lequel les documents sont traités par l'oracle lors d'une QE. Nous avions déjà évoqué ce problème en 5.2.3. C'est un problème particulièrement important pour l'oracle *partiel* : à partir du moment où on arrête l'apprentissage lorsqu'on a vérifié le comportement de la requête sur n pages, il est clair que le choix de ces pages est important. Nous allons distinguer l'oracle $partiel_{alea}$, dans lequel l'ordre des pages traitées est aléatoire, et $partiel_{heur}$, dans lequel l'ordre des pages traitées se fait selon l'heuristique décrite en 5.2.3.

	QE	QCA
Okra-names	1.6	3.48
Bigbook-addresses	1	3.02
Yahoo	6.18	11.36
E-bay	1.06	2.62
NYTimes	1.44	1.44
Google	1.86	4.78

Figure 6.4: Nombre de QE et de QCA nécessaires pour apprendre les requêtes définies sur chaque jeu de données.

	1	2	5	10
	ordre aléatoire			
Yahoo	71.8	80.9	82.1	93.7
NYTimes	91.2	92.4	95.3	100
Google	98.7	99.1	99.5	99.8
	ordre heuristique			
Yahoo	71.8	86.1	98.4	99.4
NYTimes	91.2	100	100	100
Google	98.7	100	100	100

Figure 6.5: Efficacité d'une requête apprise à partie de 1,2,5 ou 10 documents pour chaque jeu de données. En haut, les documents sont choisis aléatoirement, en bas, ils sont choisis selon l'heuristique correspondant au test de Grubbs.

Le hasard rentrant en compte dans le comportement des oracles, chaque expérience est répétée 100 fois, et les résultats donnés sont des moyennes des résultats de chaque expérience.

6.3.2 Résultats

Le but de nos expériences n'étant plus ici d'établir des comparaisons avec des programmes existants, nous avons exclu le corpus Wien dont les jeux de données présentent assez peu d'intérêt. Nous avons également exclu CiteSeer, car notre algorithme n'étant pas capable d'apprendre une requête satisfaisante pour ce jeu de données, cela n'a pas de sens d'essayer d'évaluer la vitesse avec laquelle il peut l'apprendre.

La Fig. 6.4 donne le nombre de QE et le nombre de QCA nécessaires pour inférer une requête annotant correctement l'ensemble de chaque jeu de données en utilisant l'oracle *total*. La Fig. 6.5 mesure les performances des requêtes inférées par $partiel_{alea}$ et $partiel_{heur}$ à partir de 1, 2, 5 ou 10 pages.

6.3.3 Analyse

Les expériences utilisant l'oracle *total* mettent en évidence le nombre remarquablement faible de questions posées à l'oracle nécessaires pour inférer correctement une requête. Dans la majorité des cas, moins de 5 QCA et 1 à 2 QE sont suffisantes pour déterminer la bonne

requête. Rappelons que le nombre de QE ne représente pas le nombre de pages traitées par l'oracle. La première QE correspond à la première page présentée. La deuxième QE correspond à la deuxième page présentée sur laquelle la requête ne sélectionne pas les bons éléments. Même si un apprentissage n'a nécessité que 2 QE, il se peut que l'utilisateur ait eu à tester la requête sur un grand nombre de pages pour répondre à la deuxième QE.

Les expériences utilisant l'oracle *partiel* nous apportent deux enseignements. Tout d'abord, pour tous les jeux de données sauf Yahoo, les résultats sont très bons même avec un nombre de documents utilisés pour l'apprentissage très faible. Cela montre que dans les conditions d'utilisation de Squirrel, un utilisateur peut apprendre correctement une requête en très peu de temps. Ensuite, en comparant les résultats de $partiel_{alea}$ et de $partiel_{heur}$, on constate que le choix heuristique de l'ordre de présentation des documents définie dans la section 5.2.3 présentés augmente encore considérablement l'efficacité de Squirrel. En effet, il suffit que le concepteur annote deux pages pour obtenir un programme d'extraction parfait sur les jeux de données Google et NYTimes. Le jeu de données Yahoo est hétérogène et plus difficile mais on peut constater que l'heuristique améliore les performances.

Conclusion

L'objectif du mémoire était de montrer l'importance de l'utilisation des structures arborescentes pour l'extraction d'information sur le Web et de montrer la pertinence de l'inférence grammaticale pour l'inférence de programmes d'extraction d'information.

D'un point de vue théorique, nous avons introduit un formalisme expressif et efficace pour représenter les requêtes régulières : les TSN. Formalisme bien adapté au cadre de l'inférence car les TSN ont été démontrés apprenables à partir d'exemples complètement annotés dans le modèle d'apprentissage par données fixées, avec des algorithmes polynomiaux. Pour prendre en compte les contraintes d'un système d'inférence de programmes d'extraction d'information sur le Web, nous avons étendu ces résultats dans deux directions. Tout d'abord, avec les TSNe, nous avons intégré la possibilité d'apprendre des requêtes à partir d'arbres partiellement annotés car la tâche d'annotation peut s'avérer très lourde. Ensuite, nous avons étendu nos algorithmes à un cadre interactif dans lequel le concepteur de programmes annote partiellement des pages et corrige des propositions du système.

D'un point de vue pratique, le développement de Squirrel a été extrêmement instructif. D'abord, la confrontation de nos algorithmes à la réalité d'un tâche concrète nous a imposé une étude détaillée des forces et des faiblesses de notre approche, et un travail approfondi sur l'ensemble des techniques qui permettent de passer d'un algorithme résultant de travaux théoriques à un système opérationnel.

En conclusion, nous pensons que le résultat obtenu, en l'occurrence le programme Squirrel, est suffisamment convaincant pour venir étayer notre thèse, c'est-à-dire pour mettre en évidence la pertinence de l'utilisation d'inférence grammaticale de langages d'arbres en extraction d'information sur le Web. Il reste cependant de nombreux points à développer à partir des résultats présentés dans ce mémoire :

- Il serait intéressant d'étudier la classe de langage apprise par $\mathsf{tsn_e RPNI}$ en fonction de la politique d'élagage choisie. Cela pourrait peut-être déboucher sur un nouvel éclairage du choix de la politique d'élagage. Il sera également important de caractériser la classe apprise dans les autres formalismes comme par exemple la logique datalog monadique.
- Même si notre approche purement structurelle de l'extraction d'information est visiblement pertinente, il ne fait aucun doute que pour obtenir une efficacité optimale, il faudrait s'orienter vers une approche mixte textuelle/structurelle. Or nous ne savons pas comment utiliser efficacement des données textuelles dans notre système.
- Nous ne nous sommes intéressés qu'aux requêtes monadiques. Pour des requêtes dont les résultats sont des tuples ou des données structurées selon un schéma XML un important travail reste à effectuer. Deux pistes de travail sont possibles : trouver une décomposition

de la requête complexe en requêtes monadiques, apprendre les requêtes monadiques, puis composer les résultats pour obtenir le programme d'extraction ; apprendre directement la requête complexe ce qui reviendrait, dans le cadre de l'inférence grammaticale, à s'orienter vers l'inférence de transducteurs d'arbres.

En plus de ces différentes pistes de travail, nous allons prochainement utiliser les résultats que nous avons obtenus pour établir une collaboration avec Georg Gottlob et l'équipe de Lixto, dans le but de mettre en place un système d'inférence de requêtes dans les arbres, utilisant simultanément spécification et apprentissage et combinant les points de vue logique et automates.

Annexe A

Preuves

A.1 Expressivité des automates à pas

Théorème A.1. *Les automates à pas et les automates à haies ont la même expressivité pour décrire les langages d'arbres à arité arbitraire.*

Pour démontrer ce théorème, nous allons montrer qu'il est possible de convertir un automate à haie en un automate à pas reconnaissant le même langage d'arbre à arité arbitraire, et réciproquement. La conversion d'un automate à haie en automate à pas se fait en temps linéaire, et la conversion inverse de fait en temps quadratique. Nous allons définir les opérateurs de conversion correspondants : pas et haie.

Lemme A.1. *Soit H un automate à haies. Soit $\mathsf{pas}(H)$ l'automate à pas ainsi défini :*

$$\begin{array}{rcl}
\mathsf{etats}(\mathsf{pas}(H)) & = & \mathsf{etats}(H) \uplus \biguplus_{a\in\Gamma, q\in\mathsf{etats}(H)} \mathsf{etats}(H_{a,q}) \\
\mathsf{regles}(\mathsf{pas}(H)) & = & \bigcup_{a\in\Gamma, q\in\mathsf{etats}(H)} \{q_1@q_2 \to q_3 | q_1 \xrightarrow{q_2} q_3 \in \mathsf{regles}(H_{a,q})\} \\
 & \cup & \{p \xrightarrow{\epsilon} q \mid p \in \mathsf{finaux}(H_{a,q}), q \in \mathsf{etats}(H), a \in \Gamma\} \\
 & \cup & \{a \to p \mid p \in \mathsf{initiaux}(H_{a,q}), q \in \mathsf{etats}(H)\} \\
\mathsf{finaux}(\mathsf{pas}(H)) & = & \mathsf{finaux}(H)
\end{array}$$

On a alors :

$$\mathsf{langage}(H) = \mathsf{langage}^{\mathsf{s}}(\mathsf{pas}(H))$$

Preuve : Remarquons que l'automate à pas $\mathsf{pas}(H)$ contient maintenant des ϵ-transition, car cela simplifie l'écriture de l'opérateur de conversion. Rappelons que les epsilon-transitions ne changent pas l'expressivité des automates et peuvent être supprimées par un traitement supplémentaire [Comon et al., 1997].

Contrairement aux automates à pas, les automates à haies fonctionnent à deux niveau distincts : l'un horizontal, à travers un ensemble d'automates de mots sur les états, et l'autre vertical, à travers un système d'évaluation ascendante proche de celle des automates d'arbre classiques.

Pour convertir un automate à haie H en automate à pas $\mathsf{pas}(H)$, il suffit de fusionner tous les automates "horizontaux" $H_{a,q}$ et l'automate principal H en un seul, puis d'ajouter les transitions pour simuler les règles "verticales".

L'automate à pas $\mathsf{pas}(H)$ contiendra des états issus de l'automate principal (c'est-à-dire les états de $\mathsf{etats}(H)$) et des états issus des automates $H_{a,q}$. Appelons les premiers les états principaux *de $\mathsf{pas}(H)$ et les seconds les* états secondaires *de $\mathsf{pas}(H)$. Nous allons montrer que les évaluations d'un arbre par un automate à haie H et par un automate à pas $\mathsf{pas}(H)$ coïncident sur les états principaux.*

Soit H un automate à haies. Soit t un arbre à arité arbitraire sur Γ. Nous allons prouver que :

$$\mathsf{eval}_H(t) = \mathsf{eval}^{\mathsf{s}}{}_{\mathsf{pas}(H)}(t) \cap \mathsf{etats}(H)$$

Soit $q \in \mathsf{eval}_H(t)$. On a $q \in \mathsf{etats}(H)$. Montrons que $q \in \mathsf{eval}^{\mathsf{s}}{}_{\mathsf{pas}(H)}(t)$ par récurrence sur la hauteur de t. Posons $A = \mathsf{pas}(H)$

Si $t = a$, alors le mot vide ϵ est dans $\mathsf{langage}(H_{a,q})$. On en déduit qu'il existe p dans $\mathsf{etats}(H_{a,q})$ tel que p soit à la fois initial et final. Il en résulte que A contient $a \to p$ et $p \xrightarrow{\epsilon} q$. Donc $\mathsf{eval}^{\mathsf{s}}{}_A(a)$.

Si $t = f(t_1, \ldots, t_n)$, on suppose par récurrence que pour tout i entre 1 et n, $q_i \in \mathsf{eval}_H(t_i) \Rightarrow q_i \in \mathsf{eval}^{\mathsf{s}}{}_A(t_i)$.

Si $q \in \mathsf{eval}_(f)(t_1, \ldots, t_n))$, alors il existe $q_1 \ldots q_n$ dans $\mathsf{langage}(H_{f,q})$ tel que pour tout i entre 1 et n, q_i soit dans $\mathsf{eval}_(t)_i)$. Cela implique l'existence des états $q'_1, \ldots, q'_{n+1}$ dans $H_{f,q}$ tels que q'_1 soit un état initial, q'_{n+1} soit un état final et $q'_i \xrightarrow{q_i} q'_{i+1}$ pour tout i entre 1 et n soient des règles de $H_{f,q}$.

On en déduit l'existence des règles suivantes dans A : $f \to q'_1$, $q'_i @ q_i \to q'_{i+1}$ pour tout i entre 1 et n, et enfin $q'_{n+1} \xrightarrow{epsilon} q$.

En appliquant successivement chacune de ces règles et en appliquant l'hypothèse de récurrence à chacun des q_i, on en déduit que :

$$q \in \mathsf{eval}^{\mathsf{s}}{}_A(f(t_1, \ldots, t_n))$$

On en déduit par récurrence que $q \in \mathsf{eval}_H(t) \Rightarrow q \in \mathsf{eval}^{\mathsf{s}}{}_{\mathsf{pas}(H)}(t) \cap \mathsf{etats}(H)$ est vérifiée pour tout arbre à arité arbitraire t.

Réciproquement, soit q un état de $\mathsf{eval}^{\mathsf{s}}{}_{\mathsf{pas}(H)}(t) \cap \mathsf{etats}(H)$. Montrons par récurrence sur la hauteur de t que q est un état de $\mathsf{eval}_H(t)$. Posons $A = \mathsf{pas}(H)$.

Si $t = a$, alors a a été évalué en q par une transition du type $a \to q$. Comme A contient des ϵ-transitions, cela veux dire que $\mathsf{regles}(A)$ contient une séquence du type $a \to p_1$, $p_1 \xrightarrow{\epsilon} p_2, \ldots, p_m \xrightarrow{\epsilon} q$.

Or dans toutes les règles du type $a \to p$ de A, p est un état secondaire, et dans toutes les ϵ-transitions $p_1 \xrightarrow{epsilon} p_2$, p_1 est un état secondaire et p_2 un état principal. L'état q étant principal, car $q \in \mathsf{etats}(H)$, la séquence qui mène de a à q est nécessairement $a \to p$, $p \xrightarrow{\epsilon} q$. Cela implique que p est un état initial et un état final de $H_{a,q}$. Donc le mot vide est reconnu par $H_{a,q}$, donc q est un état de $\mathsf{eval}_H(t)$.

Si $t = f(t_1, \ldots, t_n)$, on suppose par récurrence que pour tout i entre 1 et n, $\mathsf{eval}^s{}_A(t_i) \Rightarrow q_i \in \mathsf{eval}_H(t_i)$. L'arbre $f(t_1, \ldots, t_n)$ a été évalué en q par une séquence de transition du type $f \to q'_1$, $(q'_i @ q_i \to q'_{i+1})_{i \in [1\ldots(n-1)]}$, $q'_n @ q_n \to q$, avec $q_i \in \mathsf{eval}^s{}_A(t_i)$ pour tout i entre 1 et n. Les états q'_i étant secondaires et les états q_i ainsi que l'état q principaux, on doit ajouter exactement une ϵ-transition à cette séquence, pour des raisons identiques à celles décrite dans le paragraphe précédent. Cela donne la séquence de règles suivante : $f \to q'_1$, $(q'_i @ q_i \to q'_{i+1})_{i \in [1\ldots n]}$, $q'_{n+1} \xrightarrow{\epsilon} q$. Cela implique que le mot $q_1 \ldots q_n$ est reconnu par l'automate $H_{f,q}$. Or par hypothèse de récurrence, q_i est dans $\mathsf{eval}_H(t_i)$ pour tout i entre 1 et n. Donc q est dans $\mathsf{eval}_($(f)(t_1 \ldots t_n))$.

On en déduit par récurrence que $q \in \mathsf{eval}^s{}_{\mathsf{pas}(H)}(t) \cap \mathsf{etats}(H) \Rightarrow q \in \mathsf{eval}_H(t)$ est vérifiée pour tout arbre à arité arbitraire t.

Les états finaux de H et de $\mathsf{pas}(H)$ sont des états de H. Les automates H et $\mathsf{pas}(H)$ ont donc les mêmes états finaux et leurs évaluations coïncident pour ces états, donc ils reconnaissent le même langage.

◀

Lemme A.2. *Soit A un automate à pas. Soit $\mathsf{haie}(A)$ l'automate à haie ainsi défini :*

$$\begin{aligned}
\mathsf{etats}(\mathsf{haie}(A)) &= \mathsf{etats}(A) \\
\mathsf{finaux}(\mathsf{haie}(A)) &= \mathsf{finaux}(A) \\
\textit{Pour tout } a \in \Gamma \textit{ et } q \in \mathsf{etats}(A) &: \\
\mathsf{etats}(\mathsf{haie}(A)_{a,q}) &= \mathsf{etats}(A) \\
\mathsf{regles}(\mathsf{haie}(A)_{a,q}) &= \{q_1 \xrightarrow{q_2} q_3 \mid q_1 @ q_2 \to q_3 \in \mathsf{regles}(A)\} \\
\mathsf{initiaux}(\mathsf{haie}(A)_{a,q}) &= \{q' \mid a \to q' \in \mathsf{regles}(A)\} \\
\mathsf{finaux}(\mathsf{haie}(A)_{a,q}) &= \{q\}
\end{aligned}$$

On a alors :

$$\mathsf{langage}(\mathsf{haie}(A)) = \mathsf{langage}^s(A)$$

Preuve : L'idée de cette conversion est de réutiliser l'automate à pas dans son ensemble pour chacun des automates "horizontaux" de l'automate à haie. Seuls les états initiaux et finaux changent de manière à cibler la partie de l'automate à pas concernée.

Soit A un automate à pas, t un arbre à arité arbitraire. Montrons que :

$$\mathsf{eval}^s{}_A(t) = \mathsf{eval}_{\mathsf{haie}(A)}(t)$$

Soit $q \in \mathsf{eval}^s{}_A(t)$. Montrons par récurrence sur t que $q \in \mathsf{eval}_{\mathsf{haie}(A)}(t)$. Posons $H = \mathsf{haie}(A)$

Si $t = a$, alors $a \to q$ est une règle de A, donc on peut déduire de la définition de haie que q est un état de $\mathsf{initiaux}(H_{a,q})$. Or q est également un état final de $H_{a,q}$, donc $H_{a,q}$ reconnaît le mot vide, donc $q \in \mathsf{eval}_H(a)$.

Si $t = f(t_1, \ldots, t_n)$, on suppose par récurrence que pour tout i entre 1 et n, $q_i \in \mathsf{eval}^s{}_A(t_i) \Rightarrow q_i \in \mathsf{eval}_H(t_i)$.

Si $q \in \mathsf{eval}^s{}_A(f(t_1, \ldots, t_n))$, alors il existe q_i dans $\mathsf{eval}^s{}_A(()t_i)$ pour tout i entre 1 et n, et A contient les règles suivantes : $f \to q'_1$, $(q'_i @ q_i \to q'_{i+1})_{1 \leq i < n}$, $q'_n @ q_n \to q$. On peut déduire de

la définition de haie *que* q'_1 *est un état initial de* $H_{f,q}$*, que* q *est un état final de* $H_{f,q}$*, et que* $H_{f,q}$ *contient les règles* $(q'_i \xrightarrow{q_i} q'_{i+1})_{1\leq i<n}$ *ainsi que* $q'_n \xrightarrow{q_n} q$*. Il s'ensuit que le mot* $q_1 \dots q_n$ *est reconnu par* $H_{f,q}$*, donc* $q \in \mathsf{eval}_H(t)$*.*

On en déduit par récurrence que quel que soit t*,* $q \in \mathsf{eval^s}_A(t) \Rightarrow q \in \mathsf{eval}_{\mathsf{haie}(A)}(t)$*.*

Réciproquement, soit $q \in \mathsf{eval}_{\mathsf{haie}(A)}(t)$*. Montrons par récurrence sur* t *que* $q \in \mathsf{eval^s}_A(t)$*. Posons* $H = \mathsf{haie}(A)$*.*

Si $t = a$*, alors le mot vide est dans* $\mathsf{langage}(H_{a,q})$*. Le seul état final de* $H_{a,q}$ *étant* q*,* q *en est un état initial, donc* $a \to q$ *est un règle de* A*. Donc* $q \in \mathsf{eval^s}_A(a)$*.*

Si $t = f(t_1, \dots, t_n)$*, on suppose par récurrence que pour tout* i *entre* 1 *et* n*,* $q_i \in \mathsf{eval}_H(t_i) \Rightarrow q_i \in \mathsf{eval^s}_A(t_i)$*.*

Si $q \in \mathsf{eval}_H(f(t_1, \dots, t_n))$*, alors il existe* $q_1 \dots q_n$ *dans* $\mathsf{langage}(H_{f,q})$ *tel que pour tout* i *entre* 1 *et* n*,* q_i *soit dans* $\mathsf{eval}_(t)_i)$*. Cela implique l'existence des états* $q'_1, \dots, q'_n$ *dans* $H_{f,q}$ *tels que* q'_1 *soit un état initial et que* $(q'_i \xrightarrow{q_i} q'_{i+1})_{1\leq i<n}$ *ainsi que* $q'_n \xrightarrow{q_n} q$ *soient des règles de* $H_{f,q}$*.*

D'après la définition de haie*, on peut en déduire que* A *contient les règles* $f \to q'_1$*,* $(q'_i@q_i \to q'_{i+1})_{1\leq i<n}$ *et* $q'_n@q_n \to q$*.*

En appliquant successivement chacune de ces règles et en appliquant l'hypothèse de récurrence à chacun des q_i*, on en déduit que :*

$$q \in \mathsf{eval^s}_A(f(t_1, \dots, t_n))$$

On en déduit par récurrence que $q \in \mathsf{eval}_{\mathsf{haie}(A)}(t) \Rightarrow q \in \mathsf{eval^s}_A(t)$ *est vérifiée pour tout arbre à arité arbitraire* t*.*

Les automates A *et* $\mathsf{haie}(A)$ *ont les même états finaux et leurs évaluations coïncident, donc ils reconnaissent le même langage.*

◄

A.2 Algorithme de calcul de requêtes avec un TSN

Nous allons maintenant démontrer la validité de l'algorithme de la section 3.2.2. Pour cela, nous allons d'abord définir les fonctions phase^1 et phase^2 correspondant aux algorithmes décrits.

L'algorithme phase^1 calcule un arbre sur $\Sigma \times 2^{\mathsf{etats}}A$ à partir d'un arbre sur Σ de domaine équivalent. Nous allons définir la fonction $\mathsf{phase}^1 At$, de $\mathsf{dom}(t)$ dans $2^{\mathsf{etats}}(A)$ qui lui correspond. Soit t un arbre sur Σ et π un noeud de t :

– Cas 1 : π est une feuille. Alors :

$$\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi) = \{q | \exists b \in \mathsf{Bool}\ (t(\pi), b) \to q \in \mathsf{regles}(A)\}$$

– Cas 2 : π n'est pas une feuille. Alors :

$$\begin{aligned}\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi) \quad = \quad & \{q | \exists b \in \mathsf{Bool}\ \exists q_1 \in \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\mathsf{fils}_1(\pi))\ \exists q_2 \in \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t}) \\ & (\mathsf{fils}_2(\pi))\ (t(\pi), b)(q_1, q_2) \to q \in \mathsf{regles}(A)\}\end{aligned}$$

L'algorithme phase^2 calcule un arbre annoté sur $\Sigma \times \mathsf{Bool}$ à partir d'une sortie de l'algorithme phase^1. Ce calcul se fait en une phase descendante sélectionnant des états parmi ceux déjà sélectionnés par phase^1, et en en déduisant l'annotation correspondante au fur et à mesure.

Nous allons définir la fonction $\mathsf{phase}'^2 At$, qui donne en sortie l'ensemble des états sélectionnés pendant la phase descendante (l'ensemble S' donné en argument à la fonction aux). Cette fonction est un peu plus complexe à définir dans la mesure ou ses valeurs pour les des deux enfants de chaque noeuds interne doivent être définis simultanément.

Soit t un arbre sur Σ et π un noeud de t.

– Cas 1 : π est la racine. Alors

$$\mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})(\pi) = \{q | (q \in \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi)) \wedge (q \in \mathsf{finaux}(A))\}$$

– Cas 2 : π n'est pas la racine. Soit π_0, π_1 et π_2 tels que $\pi_1 = \mathsf{fils}_1(\pi_0)$, $\pi_2 = \mathsf{fils}_2(\pi_0)$ et π soit égal à π_1 ou π_2 selon qu'il est fils gauche au fils droit de son père. On définit simultanément $\mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})(\pi_1)$ et $\mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})(\pi_2)$ ainsi :

$$\begin{array}{rcl}(\mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})(\pi_1), \mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})(\pi_2)) & = & \{(q_1,q_2) | \exists (f,b) \in \Sigma \times \mathsf{Bool}\ \exists q_0 \in \mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})(\pi_0) \\ & & (q_1 \in \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_1)) \wedge \\ & & (q_2 \in \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_2)) \wedge \\ & & (f,b)(q_1,q_2) \rightarrow q_0 \in \mathsf{regles}(A)\}\end{array}$$

Nous allons montrer que la fonction $\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})$ calcule l'ensemble des états associé à chaque noeud dans un run réussit de A un arbre annoté $t \times \beta$. Notons que pour un arbre t donné, il peut y avoir plusieurs runs réussis de A sur un arbre annoté $t \times \beta$, mais que cette annotation β est unique, car A est fonctionnel. Cette annotation β est ensuite calculée simplement par $\mathsf{phase}^2_{\mathsf{A}}(\mathsf{t})$.

Nous allons d'abord montrer un lemme auxiliaire, puis caractériser $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})$ (lemme A.4), puis $\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})$ (lemme A.5).

Lemme A.3. *Soit A un* TSN*, t un arbre de T_Σ et t' un sous-arbre de t de racine π.*

$$\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi) = \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t}')(\pi)$$

Preuve : Ce lemme formalise le fait que $\mathsf{phase}^1 A$ est une fonction purement ascendante. En effet, le calcul de $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi)$ ne dépend que de A et récursivement du même calcul sur les enfants de π. Il dépend donc uniquement de A et du sous-arbre de t dont π est racine. ◄

Lemme A.4. *Soit A un* TSN*, t un arbre de T_Σ.*

$$\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\mathsf{racine}(t)) = \{q | \exists \beta \in \mathsf{T}_{\mathsf{Bool}}\ \mathsf{dom}(\beta) = \mathsf{dom}(t) \wedge q \in \mathsf{eval}_A(t \times \beta)\}$$

Preuve : Soit $P_1(t) = \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\mathsf{racine}(t))$ et $P_2(t) = \{q | \exists \beta \in \mathsf{T}_{\mathsf{Bool}}\ \mathsf{dom}(\beta) = \mathsf{dom}(t) \wedge q \in \mathsf{eval}_A(t \times \beta)\}$

Montrons par récurrence sur t que $q \in P_1(t) \Rightarrow q \in P_2(t)$.

Supposons que t soit une feuille de valeur a. Soit q un état de $P_1(t)$. Il existe b tel que $(a,b) \rightarrow q) \in \mathsf{regles}(A)$. Donc q est un état de $\mathsf{eval}_A((a,b))$. Donc q est dans $P_2(t)$.

Supposons maintenant par récurrence que pour tout arbre t' de hauteur strictement inférieure à n on ait $q \in P_1(t') \Rightarrow q \in P_2(t')$. Supposons également que la hauteur de t soit égale à n.

Soit q un état de $P_1(t)$. Soit π la racine de t, π_1 et π_2 les fils de π, t_1 et t_2 les sous-arbres de t de racine respectives π_1 et π_2. Posons $f = t(\pi)$. Comme q est dans $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\mathsf{racine}(t))$, il existe b dans Bool, q_1 et q_2 dans $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_1)$ et $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_2)$ tels que $(f,b)(q_1,q_2) \to q$ soit une règle de A. Comme π_1 est inclus dans t_1, $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_1})(\pi_1)$ est égal à $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_1)$ (lemme A.3, et de même pour π_2. L'hypothèse de récurrence implique alors l'existence de deux arbres β_1 et β_2 sur $\mathsf{T}^{\mathsf{a}}_{\mathsf{Bool}}$ tels que $q_1 \in \mathsf{eval}_A(t_1 \times \beta_1)$ et $q_2 \in \mathsf{eval}_A(t_2 \times \beta_2)$. Soit $\beta = b(\beta_1, \beta_2)$, on peut en déduire que $t \times \beta = (f,b)(t_1 \times \beta_1, t_2 \times \beta_2)$. Comme $(f,b)(q_1,q_2) \to q$ est une règle de A, il s'ensuit que q est un état de $\mathsf{eval}_A(t \times \beta)$. Donc $q \in P_2(t)$. On en déduit par récurrence que quel que soit t, $q \in P_1(t) \Rightarrow q \in P_2(t)$.

Réciproquement, montrons par récurrence sur t que $q \in P_2(t) \Rightarrow q \in P_1(t)$.

Supposons que t soit une feuille de valeur a. Soit q un état de $P_2(t)$. Il existe b booléen, tel que q soit un état de $\mathsf{eval}_A((a,b))$. Donc $(a,b) \to q$ est une règle de A. Donc q est un état de $P_1(t)$.

Supposons maintenant par récurrence que pour tout arbre t' de hauteur strictement inférieure à n on ait $q \in P_2(t') \Rightarrow q \in P_1(t')$. Supposons également que la hauteur de t soit égale à n.

Soit q un état de $P_2(t)$. Cela implique l'existence d'un arbre β de $\mathsf{T}_{\mathsf{Bool}}$ tel que q soit un état de $\mathsf{eval}_A(t \times \beta)$. En posant $t = f(t_1,t_2)$ et $\beta = b(\beta_1,\beta_2)$, on en déduit l'existence de q_1 et q_2, deux états de A tels que q_1 soit un état de $\mathsf{eval}_A(t_1 \times \beta_1)$, q_2 soit un état de $\mathsf{eval}_A(t_2 \times \beta_2)$ et $(f,b)(q_1,q_2) \to q$ soit une règle de A. Soit π_1 la racine de t_1, π_2 la racine de t_2 et π la racine de t. L'hypothèse de récurrence, appliquée à t_1, implique que q_1 soit un état de $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_1})(\pi_1)$. Le lemme A.3 implique que $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_1)$ soit égal à $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t_1})(\pi_1)$, donc q_1 est un état de $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_1)$. De même, q_2 est un état de $\mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi_2)$. Comme $(f,b)(q_1,q_2) \to q$ est une règle de A, il en résulte que $q \in \mathsf{phase}^1_{\mathsf{A}}(\mathsf{t})(\pi)$. Donc $q \in \P_1(t)$. On en déduit par récurrence que quel que soit t, $q \in P_2(t) \Rightarrow q \in P_1(t)$. ◄

Lemme A.5. *Soit A un TSN, t un arbre de T_{Σ}.*

$$\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi) = \{q | \exists \beta \in \mathsf{T}_{\mathsf{Bool}}\ \exists r \in \mathsf{sruns}_{\mathsf{A}}(\mathsf{t} \times \beta)\ r(\pi) = q\}$$

Preuve : Nous allons prouver ce lemme par récurrence descendante sur les noeuds de t. Soit $P_1(\pi) = \mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi)$ et $P_2(\pi) = \{q | \exists \beta \in \mathsf{T}_{\mathsf{Bool}}\ \exists r \in \mathsf{sruns}_{\mathsf{A}}(\mathsf{t} \times \beta)\ r(\pi) = q\}$.

Montrons par récurrence sur les noeuds de t que $q \in P_1(\pi) \Rightarrow q \in P_2(\pi)$.

Supposons que π soit la racine de t. Soit $q \in P_1(\pi)$. L'état q est élément de $\mathsf{phase}'^1_{\mathsf{A}}(\mathsf{t})(\pi)$, donc c'est un élément de l'évaluation de $t \times \beta$, donc il existe un run r de A sur $t \times \beta$ qui associe l'état q à π. Comme q est un état final, r est un run réussit. Donc $q \in P_2(\pi)$.

Supposons maintenant par récurrence que pour tout π' de profondeur strictement inférieure à n dans t, avec $n > 1$, l'implication $q \in P_1(\pi') \Rightarrow q \in P_2(\pi')$ soit vérifiée. Supposons que la profondeur de π soit égale à n.

Soit q un état de $P_1(\pi)$. Soit π_0 le père de π, π_1 et π_2 les fils gauche et droit de π_0. Supposons que $\pi = \pi_1$, et posons $q_1 = q$. Posons $f = t(\pi_0)$. La définition de phase'^2 implique

qu'il existe b booléen et q_2 et q_0 états de A tels que $(f,b)(q_1,q_2) \to q_0$ soit une règle de A, avec $q_0 \in \mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi_0)$ et $q_2 \in \mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi_2)$. Soit t_0, t_1 et t_2 les sous-arbres de t de racines respectives π_1 et π_2. Les états q_1 et q_2 font respectivement partie de $\mathsf{phase}'^1_{\mathsf{A}}(\mathsf{t})(\pi)$ et $\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi_2)$, donc il existe β_1 et β_2 arbres de $\mathsf{T}_{\mathsf{Bool}}$ tels que les évaluations de $t_1 \times \beta_1$ et de $t_2 \times \beta_2$ contiennent respectivement q_1 et q_2. Il existe donc deux runs de A r_1 et r_2 sur $t_1 \times \beta_1$ et $t_2 \times \beta_2$ tels que $r_1(\pi_1) = q_1$ et $r_2(\pi_2) = q_2$. Posons $\beta_0 = b(\beta_1, \beta_2)$. Comme $(f,b)(q_1,q_2) \to q_0$ est une règle de A, on peut définir r_0, run de A sur t_0, tel que $r_{0|\mathsf{dom}(t_1)} = r_1$, $r_{0|\mathsf{dom}(t_2)} = r_2$ et $r_0(\pi_0) = q_0$. L'hypothèse de récurrence, appliquée à π_0, nous indique qu'il existe un étiquetage β de t et un run réussit r de A sur $t \times \beta$ tel que $r(\pi_0) = q_0$. Considérons maintenant l'étiquetage β' de t ainsi défini : pour tout π' dans $\mathsf{dom}(t)$, si π' est dans le domaine de t_0, alors $\beta'(\pi') = \beta_0(\pi')$, sinon $\beta'(\pi') = \beta(\pi')$. Définissons maintenant la fonction r' de $\mathsf{dom}(t)$ dans $\mathsf{etats}(A)$ telle que $r'_{|\mathsf{dom}t_0} = r_0$ et $r'_{|\mathsf{dom}t - \mathsf{dom}t_0} = r$. La fonction r' est un run de A sur $t \times \beta'$ et $r'(\pi_1) = q_1$. Comme $\pi = \pi_1$ et $q = q_1$, $q \in P_2(\pi)$. Si on avait eu $\pi = \pi_2$, on aurait symétriquement obtenu le même résultat. On en déduit par récurrence que quel que soit π noeud de t et q état de A, $q \in P_1(\pi) \Rightarrow q \in P_2(\pi)$.

Réciproquement, montrons que $q \in P_2(\pi) \Rightarrow q \in P_1(\pi)$.

Supposons que π soit la racine de t. Soit q un état de $P_2(\pi)$. Il existe une annotation β de t et un run réussit r de A sur $t \times \beta$ tel que $r(\pi) = q$. Donc q est un élément de $\mathsf{eval}_A(t \times \beta)$ et q est final. Donc $q \in P_1(\pi)$.

Supposons maintenant par récurrence que pour tout π' de profondeur strictement inférieure à n dans t, avec $n > 1$, l'implication $q \in P_2(\pi') \Rightarrow q \in P_1(\pi')$ soit vérifiée. Supposons que la profondeur de π soit égale à n.

Soit q un état de $\P_2(\pi)$. Il existe une annotation β de t et run réussit r de A sur $t \times \beta$ tel que $r(\pi) = q$. Soit π_0 le père de π, π_1 et π_2 les enfants de π_0. Supposons que $\pi = \pi_1$. Soit $q_0 = r(\pi_0)$, $q_1 = r(\pi_1)$ et $q_2 = r(\pi_2)$. Soit $t_0 \times \beta_0$, $t_1 \times \beta_1$ et $t_2 \times \beta_2$ les sous-arbres de t de racines respectives π_0, π_1 et π_2. L'existence de r implique la présence de $r(\pi_1)$ et de $r(\pi_2)$ dans les évaluation par A de $t_1 \times \beta_1$ et de $t_2 \times \beta_2$. Donc q_1 est un élément de $\mathsf{phase}'^1_{\mathsf{A}}(\mathsf{t_1})(\pi_1)$ et q_2 un élément de $\mathsf{phase}'^1_{\mathsf{A}}(\mathsf{t_2})(\pi_2)$. Le lemme A.3 implique que $\mathsf{phase}'^1_{\mathsf{A}}(\mathsf{t})(\pi_1)$ et q_2 un élément de $\mathsf{phase}'^1_{\mathsf{A}}(\mathsf{t})(\pi_2)$. Comme $r(\pi_0) = q_0$, l'hypothèse de récurrence implique que q_0 est un élément de $\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi_0)$. Enfin, l'existence de r implique que, en posant $f = t(\pi_0)$ et $b = \beta(\pi_0)$, alors $(f,b)(q_1,q_2) \to q_0$ est une règle de A. Donc, selon la définition récursive de phase'^2, q_1 et q_2 sont respectivement éléments de $\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi_1)$ et $\mathsf{phase}'^2_{\mathsf{A}}(\mathsf{t})(\pi_2)$. Comme $\pi = \pi_1$ et $q = q_1$, $q \in P_1(\pi)$. Si on avait eu $\pi = \pi_2$, on aurait obtenu symétriquement le même résultat. On en déduit par récurrence que quel que soit π noeud de t et q état de A, $q \in P_2(\pi) \Rightarrow q \in P_1(\pi)$.
◀

Nous avons montré que phase^2 nous donne l'ensemble des états associés à chaque noeud dans un run réussit de A sur l'unique arbre annoté $t \times \beta$ reconnu par A. Il suffit d'observer les règles utilisées pendant la descente pour en déduire β. C'est ce que fait l'algorithme en associant à chaque noeud de l'arbre de sortie le booléen associée aux règles choisies.

A.3 Test d'appartenance à la classe des TSN

Nous allons démontrer ici le lemme , qui donne une expression récursive de drst, relation permettant de déterminer la fonctionnalité d'un automate d'arbre sur $\Sigma \times \mathsf{Bool}$.

Lemme A.6. *Soit A un automate d'arbre émondé sur $\Sigma \times \mathsf{Bool}$, q et q' deux états de A. En omettant les quantifications existentielles pour plus de lisibilité, les équivalences suivantes sont vérifiées :*

$$\mathsf{sim}_A(q,q') \Leftrightarrow$$
$$\begin{array}{ll} & ((a,b) \to q \wedge (a,b) \to q') \\ \vee & ((f,b)(q_1,q_2) \to q \wedge (f,b)(q'_1,q'_2) \to q' \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)) \end{array}$$

$$\mathsf{drst}(q,q') \Leftrightarrow$$
$$\begin{array}{ll} & ((a,b) \to q \wedge (a,\neg b) \to q') \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_1,q'_1) \wedge \mathsf{drst}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{drst}_A(q_2,q'_2)) \\ \vee & ((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,\neg b)(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)) \end{array}$$

Preuve : Nous allons commencer par montrer la première équivalence, celle qui concerne sim. *Soit A un automate d'arbres sur $\Sigma \times \mathsf{Bool}$. Soit q et q' tels que $\mathsf{sim}_A(q,q')$, c'est-à-dire tels que :*

$$\exists r \times \beta \in \mathsf{T}_{\Sigma \times \mathsf{Bool}}\ \exists r \in \mathsf{runs_A}(\mathsf{t} \times \beta)\ \exists r' \in \mathsf{runs_A}(\mathsf{t} \times \beta)\ (r(t \times \beta) = q \wedge r'(t \times \beta) = q')$$

Si $\mathsf{sim}_A(q,q')$, alors il existe t, β, r et r' satisfaisant $r(t \times \beta) = q \wedge r'(t \times \beta) = q'$. Deux cas se présentent :

***Cas 1 :** t **est une feuille.** Dans ce cas, en posant a la valeur de cette feuille et b la valeur de son annotation par β, l'existence de r et r' implique l'existence des règles $(a,b) \to q$ et $(a,b) \to q'$ dans $\mathsf{regles}(A)$. Le premier élément de la disjonction est donc vérifié.*

***Cas 2 :** t **n'est pas une feuille**. On pose alors $t = f(t_1,t_2)$, $\beta = b(\beta_1,\beta_2)$. Soit π_1 la racine de t_1, π_2 la racine de t_2. En posant $q_1 = r(\pi_1)$, $q_2 = r(\pi_2)$, $q'_1 = r'(\pi_1)$ et $q'_2 = r'(\pi_2)$, on peut en déduire que $(f,b)(q_1,q_2) \to q$ et $(f,b)(q'_1,q'_2) \to q'$ sont des règles de A. De plus, les restrictions r_1 et r'_1 de r et r' à $t \times \beta$ sont nécessairement des runs de A sur $t_1 \times \beta$, donc $\mathsf{sim}_A(q_1,q'_1)$. De même, $\mathsf{sim}_A(q_2,q'_2)$. Le deuxième élément de la disjonction est donc vérifié.*

Réciproquement, soit q et q' tels que le membre droit de l'équivalence soit vérifié, montrons que $\mathsf{sim}_A(q,q')$.

***Cas 1 :** $((a,b) \to q \wedge (a,b) \to q')$. Soit $t = (a,b)$. On peut définir deux run r et r' sur t tels que $r(\epsilon) = q$ et $r'(\epsilon) = q'$. Donc $\mathsf{sim}_A(q,q')$.*

***Cas 2 :** $((f,b)(q_1,q_2) \to q \wedge (f,b)(q'_1,q'_2) \to q' \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2))$. Si $\mathsf{sim}_1(q_1,q'_1)$, alors on peut en déduire l'existence d'un arbre $t_1 \times \beta_1$ et de deux runs r_1 et r'_1 de A sur*

$t_1 \times \beta_1$ *tels que* $r_1(\epsilon) = q_1$ *et* $r'_1(\epsilon) = q'_1$. *De même, il existe deux runs* r_2 *et* r'_2 *sur* $t_2 \times \beta_2$ *tels que* $r_2(\epsilon) = q_2$ *et* $r'_2(\epsilon) = q'_2$. *En posant* $t = f(t_1, t_2)$ *et* $\beta = b(\beta_1, \beta_2)$, *on peut en déduire l'existence de runs* r *et* r' *sur* $t \times \beta$ *dont les restrictions à* $t_1 \times \beta$ *sont respectivement* r_1 *et* r'_1, *et les restrictions à* $t_2 \times \beta_2$ *sont respectivement* r_2 *et* r'_2, *tels que* $r(\epsilon) = q$ *et* $r'(\epsilon) = q'$. *Il en résulte que* $\mathsf{sim}_A(q, q')$

Nous allons maintenant montrer la deuxième équivalence. Soit A *un automate d'arbres sur* $\Sigma \times \mathsf{Bool}$. *Soit* q *et* q' *tels que* $\mathsf{drst}(q, q')$, *c'est-à-dire tels que :*

$$\exists t \in \mathsf{T_\Sigma}\ \exists \beta, \beta' \in \mathsf{T_{Bool}}\ \exists r \in \mathsf{runs_A(t)}\ \exists r' \in \mathsf{runs_A(t \times \beta')}\ (\beta \neq \beta' \wedge r(t \times \beta) = q \wedge r'(t \times \beta') = q')$$

Montrons qu'au moins une des disjonctions du membre droit de l'équivalence ci-dessus, que nous numéroterons de 1 à 4, est vérifiée.

Si $\mathsf{drst}_A(q, q')$, *alors il existe* t, β, β', r *et* r' *satisfaisant* $\beta \neq \beta' \wedge r(t \times \beta) = q \wedge r'(t \times \beta') = q'$. *Plusieurs cas se présentent :*

Cas 1 : t ***est une feuille.*** *Dans ce cas* β *est* β' *sont les annotations* b *et* b' *de cette feuille, avec* b *et* b' *deux booléens de valeur différentes. Supposons que cette feuille soit étiquetée par* a. *L'existence des runs* r *et* r' *de* A *sur* $t \times \beta$ *et* $t \times \beta'$ *implique que les règles* $(a, b) \to q$ *et* $(a, b') \to q'$ *soient des règles de* A. *Donc le premier élément de la disjonction est vérifié.*

Cas 2 : t ***n'est pas une feuille.*** *On pose alors* $t = f(t_1, t_2)$, $\beta = b(\beta_1, \beta_2)$ *et* $\beta' = b(\beta'_1, \beta'_2)$, *ainsi que* $q_1 = r(\mathsf{racine}(t_1 \times \beta_1))$, $q'_1 = r'(\mathsf{racine}(t_1 \times \beta'_1))$, $q_2 = r(\mathsf{racine}(t_2 \times \beta_2))$, $q'_2 = r'(\mathsf{racine}(t_2 \times \beta'_2))$. *Remarquons que l'existence de* r *et de* r' *implique que* A *contienne les règles* $(f, b)(q_1, q_2) \to q$ *et* $(f, b')(q'_1, q'_2) \to q$. *Nous allons maintenant diviser ce cas en sous-cas en fonction des valeurs relatives de* β *et* β'.

Cas 2a : $\beta_1 \neq \beta'_1$ ***et*** $\beta_2 \neq \beta'_2$. *Dans ce cas, en considérant les restrictions* r_1 *et* r'_1 *de* r *et* r' *à* t_1, *on déduit de la définition de* drst *que* $\mathsf{drst}_A(q_1, q'_1)$. *De même,* $\mathsf{drst}_A(q_2, q'_2)$. *Or* $(f, b)(q_1, q_2) \to q$ *et* $(f, b')(q'_1, q'_2) \to q$ *sont des règles de* A. *Donc le deuxième élément de la disjonction est vérifié.*

Cas 2b : $\beta_1 \neq \beta'_1$ ***et*** $\beta_2 = \beta'_2$. *De même que dans le cas précédent, on a* $\mathsf{drst}_A(q_1, q'_1)$. *Les annotations* β_2 *et* β'_2 *étant identiques, les restriction de* r *et de* r' *à* $t \times \beta_2$ *et* $t \times \beta'_2$ *sont identiques, donc, selon la définition de* sim, *on a* $\mathsf{sim}_A(q_2, q'_2)$. *Or* $(f, b)(q_1, q_2) \to q$ *et* $(f, b')(q'_1, q'_2) \to q$ *sont des règles de* A, *donc le troisième élément de la disjonction est vérifié.*

Cas 2c : $\beta_1 = \beta'_1$ ***et*** $\beta_2 \neq \beta'_2$ *Par symétrie avec le cas précédent, le quatrième élément de la disjonction est vérifié.*

Cas 2d : $\beta_1 = \beta'_1$ ***et*** $\beta_2 = \beta'_2$. *Les annotations* β *et* β' *sont différentes, donc elles diffèrent nécessairement sur leur racine, donc* $b = \neg b'$. *Les annotations* β_1 *et* β'_1 *étant identiques, les restriction de* r *et de* r' *à* $t \times \beta_1$ *et* $t \times \beta'_1$ *sont identiques, donc* $\mathsf{sim}_A(q_1, q'_1)$. *De même* $\mathsf{sim}_A(q_2, q'_2)$. *Or* $(f, b)(q_1, q_2) \to q$ *et* $(f, b')(q'_1, q'_2) \to q$ *sont des règles de* A, *donc le dernier élément de la disjonction est vérifié.*

Réciproquement, soient q *et* q' *tels que le membre droit de l'équivalence soit vérifié, montrons que* $\mathsf{drst}_A(q, q')$. *Plus précisément, prenons chacun des élément de la disjonction, et montrons qu'il implique* $\mathsf{drst}_A(q, q')$.

Cas 1 : $((a,b) \to q \wedge (a,b') \to q' \wedge b \neq b')$. *On peut définir* r *et* r' *runs de* A *sur* (a,b) *et* (a,b') *tels que* $r(\mathsf{racine}((a,b))) = q$ *et* $r'(\mathsf{racine}((a,b'))) = q'$. *Donc* $\mathsf{drst}_A(q,q')$.

Cas 2 : $(f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_1,q'_1) \wedge \mathsf{drst}_A(q_2,q'_2))$. *Soient* t_1, β_1, β'_1, r_1 *et* r'_1 *les éléments définis existentiellement dans la formule de la définition de* drst *appliquée à* q_1 *et* q'_1, *et de même pour* t_2, β_2, β'_2, r_2 *et* r'_2 *pour les états* q_2 *et* q'_2. *Soient* $t = (f,b)(t_1,t_2)$, $\beta = b(\beta_1,\beta_2)$, $\beta' = b'(\beta'_1,\beta'_2)$. *Soit* r, *fonction de* $\mathsf{dom}(t \times \beta)$ *vers* $\mathsf{etats}(A)$ *telle que* $r_{|t_1 \times \beta_1} = r_1$, $r_{|t_2 \times \beta_2} = r_2$ *et* $r(\mathsf{racine}(t \times \beta)) = q$. *Comme* $r_1(\mathsf{racine}(t_1 \times \beta_1)) = q_1$, $r_2(\mathsf{racine}(t_2 \times \beta_2)) = q_2$ *et* $(f,b)(q_1,q_2) \to q$ *est une règle de* A, *alors* r *est un run de* A *sur* $t \times \beta$. *De même, on peut définir* r', *run de* A *sur* $t \times \beta'$ *tel que* $r'(\mathsf{racine}(t \times \beta')) = q'$. *Comme* $\beta_1 \neq \beta'_1$, $\beta \neq \beta'$. *Donc* $\mathsf{drst}(q,q')$.

Cas 3 : $((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)$ *Soient* t_1, β_1, β'_1, r_1 *et* r'_1 *les éléments définis existentiellement dans la formule de la définition de* drst *appliquée à* $\mathsf{drst}(q_1,q'_1)$. *Soient* t_2, β_2, r_2 *et* r'_2 *les éléments définis existentiellement dans la formule de* sim *appliquée à* $\mathsf{sim}(q_2,q'_2)$. *Soient* $t = (f,b)(t_1,t_2)$, $\beta = b(\beta_1,\beta_2)$, $\beta' = b'(\beta'_1,\beta_2)$. *Soit* r, *fonction de* $\mathsf{dom}(t \times \beta)$ *vers* $\mathsf{etats}(A)$ *telle que* $r_{|t_1 \times \beta_1} = r_1$, $r_{|t_2 \times \beta_2} = r_2$ *et* $r(\mathsf{racine}(t \times \beta)) = q$. *Comme* $r_1(\mathsf{racine}(t_1 \times \beta_1)) = q_1$, $r_2(\mathsf{racine}(t_2 \times \beta_2)) = q_2$ *et* $(f,b)(q_1,q_2) \to q$ *est une règle de* A, r *est un run de* A *sur* $t \times \beta$. *De même, on peut définir* r', *run de* A *sur* $t \times \beta'$ *tel que* $r'(\mathsf{racine}(t \times \beta')) = q'$. *Comme* $\beta_1 \neq \beta'_1$, $\beta \neq \beta'$. *Donc* $\mathsf{drst}(q,q')$.

Cas 4 : $((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,b')(q_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{drst}_A(q_2,q'_2))$. *Ce cas est symétrique au cas précédent.*

Cas 5 : $((f,b)(q_1,q_2) \to q \in \mathsf{regles}(A) \wedge (f,\neg b)(q'_1,q'_2) \to q' \in \mathsf{regles}(A) \wedge \mathsf{sim}_A(q_1,q'_1) \wedge \mathsf{sim}_A(q_2,q'_2)$. *Soient* t_1, β_1, r_1, r'_1 *les éléments définis existentiellement dans la définition de* $\mathsf{sim}(q_1,q'_1)$, *et* t_2, β_2, r_2, r'_2 *les éléments définis existentiellement dans la définition de* $\mathsf{sim}(q_2,q'_2)$. *Soient* $t = f(t_1,t_2)$, $\beta = b(\beta_1,\beta_2)$ *et* $\beta' = (\neg b)(\beta_1,\beta_2)$. *Soit* r, *fonction de* $\mathsf{dom}(t \times \beta)$ *dans* $\mathsf{etats}(A)$ *telle que* $r_{|t_1 \times \beta_1} = r_1$, $r_{|t_2 \times \beta_2} = r_2$ *et* $r(\mathsf{racine}(t \times \beta)) = q$. *Comme* $(f,b)(q_1,q_2) \to q$ *est une règle de* A, r *est un run de* A *sur* $t \times \beta$. *De même, on peut définir* r', *run de* A *sur* $t \times \beta'$ *tel que* $r'(\mathsf{racine}(t \times \beta')) = q'$. *Donc* $\mathsf{drst}(q,q')$.

◀

A.4 Expressivité comparée des automates à pas et de la MSO dans les arbres à arité arbitraire

Nous allons démontrer ici le lemme 3.4, élément manquant à la démonstration présente dans le chapitre 3.

Lemme A.7. *La fonction de correspondance de requête* $\mathsf{c_{requete}}$ *est une bijection entre l'ensemble des requêtes étendues exprimée en* $\mathsf{MSO^s}$ *et l'ensemble des requêtes exprimée en* $\mathsf{MSO^a}$.

Preuve : Nous allons établir une fonction $\mathsf{Enc}()$ *d'encodage des formules de* $\mathsf{MSO^s}$ *en* $\mathsf{MSO^a}$, *telle que pour toute formule* $\phi(x)$ *de* $\mathsf{MSO^s}$, *la requête définie par* $\mathsf{Enc}(\phi(\mathsf{x}))$ *soit la requête correspondant (au sens de* $\mathsf{c_{requete}}$*) à la requête définie par* $\phi(x)$. *Réciproquement, nous allons établir une fonction* $\mathsf{Dec}()$ *de décodage des formules de* $\mathsf{MSO^a}$ *en* $\mathsf{MSO^s}$, *telle que pour toute formule* $\psi(x)$ *de* $\mathsf{MSO^a}$, *la requête définie par* $\psi(x)$ *soit la requête correspondante (au sens de* $\mathsf{c_{requete}}$*) à la requête définie par* $\mathsf{Dec}(\phi(\mathsf{x}))$.

L'opérateur d'encodage Enc() *est défini ainsi :*

$$
\begin{array}{ll}
\mathsf{Enc}(\exists \mathsf{x}\ \psi) & =_{\mathsf{def}}\ \exists x\ \mathsf{Enc}(\psi) \\
\mathsf{Enc}(\exists \mathsf{p}\ \psi) & =_{\mathsf{def}}\ \exists p\ \mathsf{Enc}(\psi) \\
\mathsf{Enc}(\psi \wedge \psi') & =_{\mathsf{def}}\ \mathsf{Enc}(\psi) \wedge \mathsf{Enc}(\psi') \\
\mathsf{Enc}(\neg\psi) & =_{\mathsf{def}}\ \neg\mathsf{Enc}(\psi) \\
\mathsf{Enc}(\mathsf{p}(\mathsf{x})) & =_{\mathsf{def}}\ p(x) \\
 & \\
\mathsf{Enc}(\mathsf{premiere\_arete}(\mathsf{x},\mathsf{y})) & =_{\mathsf{def}}\ \mathsf{fils}_1(y,x) \wedge \mathsf{feuille}(x) \\
\mathsf{Enc}(\mathsf{arete\_suivante}(\mathsf{x},\mathsf{y})) & =_{\mathsf{def}}\ \exists z\ (\mathsf{fils}_1(y,x) \wedge \mathsf{fils}_1(x,z)) \\
\mathsf{Enc}(\mathsf{cible}(\mathsf{x},\mathsf{y})) & =_{\mathsf{def}}\ \exists z\ (\mathsf{fils}_2(x,z) \wedge \mathsf{lar}(y,z)) \\
\mathsf{Enc}(\mathsf{etiq}_\mathsf{a}(\mathsf{x})) & =_{\mathsf{def}}\ \mathsf{etiq}_a(x) \quad (a \in \Sigma) \\
\mathsf{Enc}(\mathsf{derniere\_arete}(\mathsf{x},\mathsf{y})) & =_{\mathsf{def}}\ \mathsf{lar}(x,y) \wedge (x \neq y) \\
\mathsf{Enc}(\mathsf{racine}(\mathsf{x})) & =_{\mathsf{def}}\ \exists y\ (\mathsf{lar}(x,y) \wedge \mathsf{racine}(y)) \\
\mathsf{Enc}(\mathsf{feuille}(\mathsf{x})) & =_{\mathsf{def}}\ \exists y\ \mathsf{fils}_2(y,x) \wedge \mathsf{feuille}(x)
\end{array}
$$

L'encodage utilise la formule auxiliaire lar *définie ainsi :*

$$
\begin{array}{rll}
\mathsf{ar}'(x,p) & =_{\mathsf{def}} & p(x) \wedge \forall y \forall z((\mathsf{fils}_1(y,z) \wedge p(z)) \rightarrow p(y)) \\
\mathsf{ar}(x,p) & =_{\mathsf{def}} & \mathsf{ar}'(x,p) \wedge \forall p'(\mathsf{ar}'(x,p') \rightarrow p \subset p') \\
\mathsf{lar}(x,y) & =_{\mathsf{def}} & \mathsf{feuille}(x) \wedge \exists p.p(y) \wedge \mathsf{ar}(x,p) \wedge (\mathsf{racine}(y) \vee \exists y'\ \mathsf{fils}_2(y',y))
\end{array}
$$

La sémantique de lar *est la suivante :* $\mathsf{lar}(x,y)$ *signifie que* y *est le plus lointain ancêtre de* x *tels que toute la lignée entre* y *et* x *n'est formé que de relations père/fils gauche.*

L'opérateur de décodage Dec() *est défini ainsi :*

$$
\begin{array}{ll}
\mathsf{Dec}(\exists \mathsf{x}\ \psi) & =_{\mathsf{def}}\ \exists x\ \mathsf{Dec}(\psi) \\
\mathsf{Dec}(\exists \mathsf{p}\ \psi) & =_{\mathsf{def}}\ \exists p\ \mathsf{Dec}(\psi) \\
\mathsf{Dec}(\psi \wedge \psi') & =_{\mathsf{def}}\ \mathsf{Dec}(\psi) \wedge \mathsf{Dec}(\psi') \\
\mathsf{Dec}(\neg\psi) & =_{\mathsf{def}}\ \neg\mathsf{Dec}(\psi) \\
\mathsf{Dec}(\mathsf{p}(\mathsf{x})) & =_{\mathsf{def}}\ p(x) \\
 & \\
\mathsf{Dec}(\mathsf{fils}_1(\mathsf{x},\mathsf{y})) & =_{\mathsf{def}}\ \mathsf{premiere\_arete}(y,x) \vee \mathsf{arete\_suivante}(y,x) \\
\mathsf{Dec}(\mathsf{fils}_2(\mathsf{x},\mathsf{y})) & =_{\mathsf{def}}\ \exists z\ (\mathsf{cible}(x,z) \wedge \mathsf{tete}(z,y)) \\
\mathsf{Dec}(\mathsf{etiq}_\mathsf{a}(\mathsf{x})) & =_{\mathsf{def}}\ \mathsf{etiq}_a(x) \quad (a \in \Sigma) \\
\mathsf{Dec}(\mathsf{racine}(\mathsf{x})) & =_{\mathsf{def}}\ \exists z\ (\mathsf{racine}(z) \wedge \mathsf{tete}(z,x)) \\
\mathsf{Dec}(\mathsf{feuille}(\mathsf{x})) & =_{\mathsf{def}}\ \mathsf{racine}(x) \vee \exists z\ \mathsf{cible}(z,x)
\end{array}
$$

L'opérateur de décodage utilise la formule auxiliaire tete *définie ainsi :*

$$\mathsf{tete}(x,y) \quad =_{\mathsf{def}} \quad (\mathsf{feuille}(x) \wedge (x = y)) \vee (\mathsf{derniere\_arete}(x,y))$$

Cette opérateur s'appelle tete *par analogie avec la fonction du même nom défini en 2.9. La formule* $\mathsf{tete}(x,y)$ *est satisfaite quand* y *est la* tête *(au sens de 2.9) du sous-arbre de racine* x*. Nous allons montrer dans le lemme A.9 que la formule* $\mathsf{tete}(x,y)$ *dans un arbre de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *correspond exactement à la formule* $\mathsf{lar}(x,y)$ *dans l'arbre de construction correspondant.*

Nous allons maintenant établir les correspondances entre les requêtes définies par ϕ *et* $\mathsf{Enc}(\phi)$*, avec* ϕ *formule de* $\mathsf{MSO}^{\mathsf{s}}$ *ainsi qu'entre celles définies par* ψ *et* $\mathsf{Dec}(\psi)$*, avec* ψ *formule de* $\mathsf{MSO}^{\mathsf{a}}$*. Pour cela, nous allons d'abord établir les lemmes auxiliaires A.8 et A.9, puis nous établirons les lemmes explicitant ces correspondances A.10 et A.12.*

Lemme A.8. *Soit* t *un arbre de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ *et* t' *un sous-arbre de* t*. La proposition suivante est toujours vraie :*

$$\mathsf{tete}(\mathsf{racine}(t'), \mathsf{c_{dom}}(t)^{-1}(\mathsf{racine}(\mathsf{c_{arbre}}(t'))))$$

Preuve : Si t' *est une feuille, alors* $racine(t') = \mathsf{c_{dom}}(t)^{-1}(\mathsf{racine}(\mathsf{c_{arbre}}(t')))$*, or* π *est une feuille implique* $\mathsf{tete}(\pi, \pi)$*, donc le lemme est vérifié.*

Si t' *n'est pas une feuille, alors* $t' = t_1 @^s t_2$*. L'arête reliant* t_1 *à* t_2 *est la dernière arête en dessous de la racine de* t'*. Or la racine de* $\mathsf{c_{arbre}}(t') = \mathsf{c_{arbre}}(t_1)@^a\mathsf{c_{arbre}}(t_2)$ *est l'élément correspondant à cette arête. Donc* $\mathsf{tete}(\mathsf{racine}(t'), \mathsf{c_{dom}}(t)^{-1}(\mathsf{racine}(\mathsf{c_{arbre}}(t'))))$*.*

◀

Lemme A.9. *Soit* t *un arbre de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$*, et* σ *une affectation des variables* x *et de* y *dans* $\mathsf{dom}^{\mathsf{s}}(t)$*.*

$$t, \sigma \models_{\mathsf{MSO}^{\mathsf{s}}} \mathsf{tete}(x, y) \Leftrightarrow \mathsf{c_{arbre}}(t), \mathsf{c_{dom}} \circ \sigma \models_{\mathsf{MSO}^{\mathsf{a}}} \mathsf{tete}(x, y)$$

Preuve :

Soit π *et* π' *deux éléments de* $\mathsf{dom}(t)$ *tels que* $\mathsf{tete}(\pi, \pi')$*. Montrons que* $\mathsf{lar}(\mathsf{c_{dom}}(t)(\pi), \mathsf{c_{dom}}(t)(\pi'))$*.*

Cas 1 : π ***est une feuille****. On a alors* $\pi = \pi'$*. De plus, si* π *est une feuille, alors* $\mathsf{c_{dom}}(\pi)$ *est soit la racine de* $\mathsf{c_{arbre}}(t)$ *(dans la cas ou* t *est réduit à* π*), soit un fils droit (car toutes les feuilles* a *de* t *sont construites par une opération du type* $t'@^s a$ *lors de la construction de* t*). Dans les deux cas,* π *n'est pas un fils gauche, donc* $\mathsf{lar}(\pi, \pi)$*.*

Cas 2 : π ***n'est pas une feuille****. Le noeud* π *est la racine de* t_0*, sous-arbre de* t*. Soient* $\pi_1, \ldots, \pi_n$ *les enfants de* π*, eux-mêmes racine des sous-arbres* $t_1, \ldots, t_n$*. Soient* $\pi'_1, \ldots, \pi'_n$ *les arêtes reliant* π *à* $\pi_1, \ldots, \pi_n$*, ce qui implique* $\pi' = \pi'_n$*. Soit* $a = t(\pi)$*. On a alors* $t_0 = a@^s t_1 @^s \ldots @^s t_n$*, et donc* $\mathsf{c_{arbre}}(t_0) = \mathsf{c_{arbre}}(a)@^a\mathsf{c_{arbre}}(t_1)@^a \ldots @^a\mathsf{c_{arbre}}(t_n)$*, avec* $\mathsf{c_{dom}}(\pi)$ *racine de* $\mathsf{c_{arbre}}(a)$*,* $\mathsf{c_{dom}}(\pi'_1)$ *racine de* $\mathsf{c_{arbre}}(a)@^a\mathsf{c_{arbre}}(t_1)$*, ...,* $\mathsf{c_{dom}}(\pi'_n)$ *racine de* $\mathsf{c_{arbre}}(a)@^a\mathsf{c_{arbre}}(t_1)@^a \ldots @^a\mathsf{c_{arbre}}(t_n)$*. Les noeuds* $\mathsf{c_{dom}}(\pi)$*,*$\mathsf{c_{dom}}(\pi'_1)$*,...,*$\mathsf{c_{dom}}(\pi'_n)$ *forment donc une séquence telle que deux éléments consécutifs sont en relation fils-gauche/père. Or* $\mathsf{c_{dom}}(\pi)$ *est une feuille, et* $\mathsf{c_{dom}}(\pi'_n)$ *n'est pas un fils gauche, car cela impliquerai l'existence d'un fils supplémentaire* π_{n+1} *à* π*. Donc on a bien* $\mathsf{lar}(\pi, \pi'_n)$*. Comme* $\pi' = \pi'_n$*, cela implique* $\mathsf{lar}(\pi, \pi')$*.*

Réciproquement, posons $c = \mathsf{c_{arbre}}(t)$*. Soient* π *et* π' *deux éléments de* $\mathsf{dom}(c)$ *tels que* $\mathsf{lar}(\pi, \pi')$*. Montrons que* $\mathsf{tete}(\mathsf{c_{dom}}(t)^{-1}(\pi), \mathsf{c_{dom}}(t)^{-1})$*.*

Cas 1 : $\pi = \pi'$*. Dans ce cas,* $\mathsf{lar}(\pi, \pi')$ *implique que* π *soit un fils gauche, donc que* $\mathsf{c_{dom}}^{-1}(\pi)$ *soit une feuille. Si* $\mathsf{c_{dom}}^{-1}(\pi)$ *est une feuille et* $\mathsf{c_{dom}}^{-1}(\pi) = \mathsf{c_{dom}}^{-1}(\pi')$*, alors* $\mathsf{tete}(\mathsf{c_{dom}}^{-1}(\pi), \mathsf{c_{dom}}^{-1}(\pi))$*.*

Cas 2 : $\pi \neq \pi'$*. Si* $\mathsf{lar}(\pi, \pi')$ *avec* $\pi \neq \pi'$*, alors* c *contient une séquence de noeuds* $\pi'_1 \ldots \pi'_n$ *telle que* $\mathsf{fils}_1(\pi'_1, \pi)$ *et* $\mathsf{fils}_1(\pi'_{i+1}, \pi'_i)$ *pour* i *entre* 1 *et* n*, avec* $\pi' = \pi'_n$*. Les* π'_i *ayant des fils*

gauches, ils ont également des fils droits que nous noteront respectivement $\pi_1, \ldots, \pi_n$. *Posons* $\pi'_0 = \pi$ *et* $a = c(\pi)$. *Soient* $c_1, \ldots, c_n$ *les sous-arbres de* c *de racines respectives* $\pi_1, \ldots, \pi_n$, *et* c' *le sous-arbre de* c *de racine* π'. *On a alors* $c' = a@^a c_1 @^a \ldots @^a c_n$. *Soit* t' *le sous-arbre partiel de* t *correspondant à* c', *tel que* $t' = \mathsf{c_{arbre}}^{-1}(c')$. *On a* $\mathsf{c_{dom}}(t)^{-1}(\pi) = \mathsf{racine}(t')$ *et* $\mathsf{c_{dom}}^{-1}(\pi') = \mathsf{c_{dom}}^{-1}(\mathsf{racine}(\mathsf{c_{arbre}}(t)))$. *Donc d'après le lemme A.8,* $\mathsf{tete}(\mathsf{c_{dom}}(t)^{-1}(\pi), \mathsf{c_{dom}}(t)^{-1}(\pi'))$.

◀

Lemme A.10. *Soit* t *un arbre à arité arbitraire,* ϕ *une formule de* $\mathsf{MSO^s}$, *et* σ *une affectation des variables libres de cette formule.*

$$t, \sigma \models_{\mathsf{MSO^s}} \phi \Leftrightarrow \mathsf{c_{arbre}}(t), \mathsf{c_{dom}} \circ \sigma \models_{\mathsf{MSO^a}} \mathsf{Enc}(\phi)$$

Preuve : Ce lemme se montre par récurrence sur la structure de la formule ϕ.

La récurrence se fait sur la profondeur *de la formule, c'est-à-dire sur le nombre d'appels récursifs nécéssaires pour la construction de la formule selon la définition récursive des formules MSO que nous avons donnée.*

Supposons par récurrence que pour toute formule de profondeur inférieure ou égale à n, *la propriété soit vérifiée. Soit* ϕ *une formule de profondeur* $n+1$. *La formule* ϕ *a nécessairement une des syntaxes suivante :* $\exists x \phi'$, $\exists p \phi'$, $\not\phi'$ *ou* $\phi' \wedge \phi''$, *avec* ϕ' *et* ϕ'' *de profondeur inférieure ou égale à* n. *Dans chacun de ces cas, on peut déduire de façon immédiate que si la propriété est vraie pour* ϕ' *(et* ϕ''*), alors elle l'est également pour* ϕ.

Montrons maintenant que la propriété est vérifiée pour les formules de profondeur 1, *c'est-à-dire la formule* $p(x)$ *ou l'un des prédicat élémentaire de la* $\mathsf{MSO^s}$.

Cas 1 : $\phi = p(x)$. p *est une variable du second ordre et* x *une variable du premier ordre. La fonction* $\mathsf{c_{dom}}$ *est une bijection, donc* $\sigma(x) \in \sigma(p)$ *est équivalent à* $\mathsf{c_{dom}}(\sigma(x)) \in \mathsf{c_{dom}}(\sigma(p))$. *Il en résulte que pour tout* t *arbre de* $\mathsf{T^s_\Gamma}$, ϕ *formule de* $\mathsf{MSO^s}$,$\sigma \models p(x) \Leftrightarrow \mathsf{c_{arbre}}(t), \mathsf{c_{dom}} \circ \sigma \models_{\mathsf{MSO^a}} p(x)$.

Cas 2 : $\phi = \mathsf{premiere\_arete}(x, y)$. *Soit* t *un arbre de* $\mathsf{T^s_\Gamma}$ *et* σ *une affectation de* x *et de* y *dans* t *qui satisfait* $\mathsf{premiere\_arete}(x, y)$. *Montrons que* $\mathsf{c_{dom}} \circ \sigma$ *est une affectation de* x *et de* y *dans* $\mathsf{c_{arbre}}(t)$ *qui satisfait* $\mathsf{Enc(premiere\_arete(x, y))}$. *Considérons* $t_0 = t_1 @^s t_2$ *le sous-arbre partiel de* t *tel que* $\sigma(x)$ *est la racine de* t_1 *et* $\sigma(y)$ *l'arête entre* t_1 *et* t_2. *Comme* $\sigma(y)$ *est la première arête en dessous de* $\sigma(x)$, t_1 *est une feuille. Dans* $\mathsf{c_{arbre}}(t)$, $\mathsf{c_{dom}}(t)(\sigma(x))$ *est la racine de* $\mathsf{c_{arbre}}(t_1)$, *donc c'est une feuille, et* $\mathsf{c_{dom}}(t)(\sigma(y))$ *est la racine de* $\mathsf{c_{arbre}}(t_1 @^a t_2)$, *donc* $\sigma(x)$ *est le fils gauche de* $\sigma(y)$.

Réciproquement, soit $\psi = fils_1(y, x) \wedge \mathsf{feuille}(x)$. *Soit* c *un arbre de* $\mathsf{T^a_\Gamma}$ *et* σ *une affection de* x *et de* y *satisfaisant* ψ. *Considerons* $c_0 = c_1 @^a c_2$ *le sous-arbre de* c *tel que* $\sigma(y)$ *est la racine de* c_0 *et* $\sigma(x)$ *la racine de* c_1. *Le noeud* $\sigma(x)$ *étant une feuille,* c_1 *est une feuille, donc* $\mathsf{c_{arbre}}^{-1}(c_1)$ *est un sous-arbre partiel de* $\mathsf{c_{arbre}}^{-1}(c_0)$ *réduit à un unique noeud qui est* $\mathsf{c_{dom}}^{-1}(\sigma(x))$. *Donc,* $\mathsf{c_{arbre}}^{-1}(c_0)$, *c'est-à-dire* $\mathsf{c_{arbre}}^{-1}(c_1) @^s \mathsf{c_{arbre}}^{-1}(c_2)$, *est un arbre dont la racine n'a qu'un seul fils, qui est* $\mathsf{c_{dom}}^{-1}(\sigma(x))$, *et dont l'arête séparant la racine est ce fils est* $\mathsf{c_{dom}}^{-1}(\sigma(y))$. *Comme* $\mathsf{c_{arbre}}^{-1}(c_0)$ *est un sous-arbre partiel de* c, *on en déduit que* $\mathsf{c_{dom}}^{-1}(\sigma(y))$ *est la première arête en dessous du noeud* $\mathsf{c_{dom}}^{-1}(\sigma(x))$.

On en déduit que pour tout t *de* $\mathsf{T^s_\Gamma}$ *et tout* σ *affectation de* x *et de* y *dans* t, *on a* $t, \sigma \models_{\mathsf{MSO^s}} \mathsf{premiere\_arete}(x, y) \Leftrightarrow \mathsf{c_{arbre}}(t), \mathsf{c_{dom}} \circ \sigma \models_{\mathsf{MSO^a}} \mathsf{fils_1}(y, x) \wedge \mathsf{feuille}(x)$.

Cas 3 : $\phi = \mathsf{cible}(x,y)$. *Soit t un arbre de $\mathsf{T}^{\mathsf{s}}_{\Gamma}$ et σ une affectation de x et de y dans t qui satisfait* $\mathsf{cible}(x,y)$. *Montrons que* $\mathsf{c_{dom}} \circ \sigma$ *est une affectation de x et de y dans* $\mathsf{c_{arbre}}(t)$ *qui satisfait* $\exists z\ \mathsf{fils}_2(x,z) \wedge \mathsf{lar}(y,z)$. *Soit t_0 le sous-arbre partiel de t tel que $\sigma(x)$ soit la dernière arête en dessous de t_0 et $\sigma(y)$ soit le dernier fils de t_0, et t_1 le sous-arbre partiel de t dont la racine est $\sigma(y)$. L'arbre* $\mathsf{c_{arbre}}(t_0)$ *est le sous-arbre de* $\mathsf{c_{arbre}}(t)$ *de racine* $\mathsf{c_{dom}}(\sigma(x))$. *L'arbre* $\mathsf{c_{arbre}}(t_1)$ *est le sous-arbre de* $\mathsf{c_{arbre}}(t)$ *de racine z' avec* $\mathsf{fils}_2(\mathsf{c_{dom}}(\sigma(x)), z')$. *En posant* $z = \mathsf{c_{dom}}(t)^{-1}(z')$, *et en appliquant le lemme A.8 à t_1, on en déduit* $\mathsf{tete}(\sigma(y), \sigma(z))$, *ce qui implique, d'après le lemme A.9,* $\mathsf{lar}(\mathsf{c_{dom}}(\sigma(y)), \mathsf{c_{dom}}(\sigma(z)))$.

Réciproquement, soit $\psi = \exists z\ \mathsf{fils}_2(x,z) \wedge \mathsf{lar}(y,z)$. *Soit c un arbre de* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$ *et Σ une affectation de x et de y satisfaisant ψ. Considérons c_0 le sous-arbre de c de racine x, c_1 le sous-arbre de c dont la racine est le fils gauche de x, et c_2 le sous-arbre de c de racine z, fils droit de x. On a $c_0 = c_1 @^a c_2$. On en déduit* $\mathsf{c_{arbre}}^{-1}(c_0) = \mathsf{c_{arbre}}^{-1}(c_1) @^s \mathsf{c_{arbre}}^{-1}(c_2)$, *avec* $\mathsf{c_{dom}}(t)^{-1}(x)$ *arête reliant* $\mathsf{c_{arbre}}^{-1}(c_1)$ *et* $\mathsf{c_{arbre}}^{-1}(c_2)$. *Le lemme A.9 nous indique que* $\mathsf{lar}(y,z)$ *implique* $\mathsf{tete}(\mathsf{c_{dom}}(t)^{-1}(\sigma(y)), \mathsf{c_{dom}}(t)^{-1}(\sigma(z)))$, *et le lemme A.8 nous indique que* $\mathsf{tete}(racine(\mathsf{c_{arbre}}^{-1}(c_2)), \mathsf{c_{dom}}(t)^{-1}(\sigma(z))$. *Comme* $\mathsf{tete}(a,b)$ *et* $\mathsf{tete}(a',b)$ *implique* $a = a'$, *alors y est la racine de* $\mathsf{c_{arbre}}^{-1}(c_2)$, *donc x est l'arête reliant la racine de* $\mathsf{c_{arbre}}^{-1}(c_1)$ *au noeud y. Donc* $\mathsf{cible}(x,y)$.

Le cas $\mathsf{arete\_suivante}(x,y)$ *est très similaire au cas* $\mathsf{premiere\_arete}(x,y)$, *le cas* $\mathsf{derniere\_arete}(x,y)$ *se déduit directement du lemme A.9, le cas* $\mathsf{racine}(x,y)$ *se déduit directement du lemme A.8, et les cas* $\mathsf{etiq}_a(x)$ *et* $\mathsf{feuille}(x)$ *sont triviaux.*

◀

De même que pour $\mathsf{Enc}()$, *nous allons établir dans le lemme que pour toute formule ψ de* $\mathsf{MSO}^{\mathsf{a}}$, $\mathsf{Dec}(\psi)$ *selectionne les noeuds correspondants dans les arbres correspondants. Établissons d'abord ce lemme auxiliaire :*

Lemme A.11. *Soit ψ une formule de* $\mathsf{MSO}^{\mathsf{a}}$.

$$\mathsf{Enc}(\mathsf{Dec}(\psi)) = \psi$$

Preuve : Comme dans la preuve précédente, on va prouver cette propriété par récurrence sur la structure de ψ, et l'induction est triviale. Il faut maintenant vérifier par le calcul la propriété pour chacun des prédicats élémentaires.

Cas 1 : $\psi = \mathsf{fils}_1(x,y)$.

$$\begin{aligned}
\mathsf{Enc}(\mathsf{Dec}(\mathsf{fils}_1(\mathsf{x},\mathsf{y}))) &= \mathsf{Enc}(\mathsf{premiere_arete}(\mathsf{y},\mathsf{x}) \vee \mathsf{arete_suivante}(\mathsf{y},\mathsf{x})) \\
&= \mathsf{Enc}(\mathsf{premiere_arete}(\mathsf{y},\mathsf{x})) \vee \mathsf{Enc}(\mathsf{arete_suivante}(\mathsf{y},\mathsf{x})) \\
&= (\mathsf{fils}_1(x,y) \wedge \mathsf{feuille}(y)) \vee (\exists z \mathsf{fils}_1(x,y) \wedge \mathsf{fils}_1(y,z)) \\
&= \mathsf{fils}_1(x,y) \wedge (\mathsf{feuille}(y) \vee \exists z \mathsf{fils}_1(y,z)) \\
&= \mathsf{fils}_1(x,y)
\end{aligned}$$

Dans l'avant-dernière ligne, soit y est une feuille, soit il a un fils. Donc le membre droit de la conjonction est toujours vrai.

Cas 2 : $\psi = \mathsf{fils}_2(x, y)$. *On utilise le lemme A.9 pour les conversions de* lar *et* tete.

$$\begin{array}{rcl} \mathsf{Enc}(\mathsf{Dec}(\mathsf{fils}_2(\mathsf{x},\mathsf{y}))) & = & \mathsf{Enc}(\exists \mathsf{zcible}(\mathsf{x},\mathsf{z}) \wedge \mathsf{tete}(\mathsf{z},\mathsf{y})) \\ & = & \exists z \mathsf{Enc}(\mathsf{cible}(\mathsf{x},\mathsf{z})) \wedge \mathsf{Enc}(\mathsf{tete}(\mathsf{z},\mathsf{y})) \\ & = & \exists z (\exists z' \mathsf{fils}_2(x, z') \wedge \mathsf{lar}(z, z')) \wedge \mathsf{lar}(z, y) \\ & = & \exists z \mathsf{fils}_2(x, y) \wedge \mathsf{lar}(z, y) \\ & = & \exists z \mathsf{fils}_2(x, y) \end{array}$$

L'égalité entre la ligne 3 et la ligne 4 se justifie car $\mathsf{lar}(z, y)$ *et* $\mathsf{lar}(z, z')$ *implique* $y = z'$. *L'égalité entre la ligne 4 et la ligne 5 se justifie car* y *étant un fils droit, il existe nécessairement un* z *tel que* $\mathsf{lar}(z, y)$.

Cas 3 : $\psi = \mathsf{racine}(x)$.

$$\begin{array}{rcl} \mathsf{Enc}(\mathsf{Dec}(\mathsf{racine}(\mathsf{x}))) & = & \mathsf{Enc}(\exists \mathsf{zracine}(\mathsf{z}) \wedge \mathsf{tete}(\mathsf{z},\mathsf{x})) \\ & = & \exists z \mathsf{Enc}(\mathsf{racine}(\mathsf{z})) \wedge \mathsf{Enc}(\mathsf{tete}(\mathsf{z},\mathsf{x})) \\ & = & \exists z (\exists y \mathsf{racine}(y) \wedge \mathsf{lar}(z, y)) \wedge \mathsf{lar}(z, x) \\ & = & \exists z \mathsf{racine}(x) \wedge \mathsf{lar}(z, x) \mathsf{lar}(z, x) \\ & = & \mathsf{racine}(y) \end{array}$$

L'égalité entre les lignes 3, 4 et 5 s'explique de la même manière que dans le cas précédent.

Cas 4 : $\psi = \mathsf{feuille}(x)$

$$\begin{array}{rcl} \mathsf{Enc}(\mathsf{Dec}(\mathsf{feuille}(\mathsf{x}))) & = & \mathsf{Enc}(\mathsf{racine}(\mathsf{x}) \vee \exists \mathsf{zcible}(\mathsf{z},\mathsf{x})) \\ & = & \mathsf{Enc}(\mathsf{racine}(\mathsf{x})) \vee \exists z\ \mathsf{Enc}(\mathsf{cible}(\mathsf{z},\mathsf{x})) \\ & = & (\exists y \mathsf{racine}(y) \wedge \mathsf{lar}(x, y)) \vee (\exists z \exists z'\ \mathsf{fils}_2(z, z') \wedge \mathsf{lar}(x, z')) \end{array}$$

Cette dernière expression implique $\mathsf{feuille}(x)$ *car* $\mathsf{lar}(x, y)$ *implique* $\mathsf{feuille}(x)$ *quel que soit* y. *Réciproquement,* $\mathsf{feuille}(x)$ *implique qu'il existe un* y *tel que* $\mathsf{lar}(x, y)$, *et tel que cet* y *soit soit un fils droit soit la racine.*

Enfin, les cas $\mathsf{etiq}_a(x)$ *et* $p(x)$ *sont triviaux.* ◄

Lemme A.12. *Soit* c *un arbre de construction,* ψ *une formule de* $\mathsf{MSO}^{\mathsf{a}}$, *et* σ *une affectation des variables libres de cette formule.*

$$c, \sigma \models_{\mathsf{MSO}^{\mathsf{a}}} \psi \Leftrightarrow {\mathsf{c}_{\mathsf{arbre}}}^{-1}(t), {\mathsf{c}_{\mathsf{dom}}}^{-1} \circ \sigma \models_{\mathsf{MSO}^{\mathsf{s}}} \mathsf{Dec}(\psi)$$

Preuve : En posant $\phi = \mathsf{Dec}(()\psi)$ *et en utilisant le lemme A.11, on se ramène immédiatement au lemme A.10.* ◄

Des lemmes A.10 et A.12 on peut déduire les deux propriétés suivantes, pour $\phi(x)$ *formule de* $\mathsf{MSO}^{\mathsf{s}}$, t *arbre de* $\mathsf{T}^{\mathsf{s}}_{\Gamma}$, $\psi(x)$ *formule de* $\mathsf{MSO}^{\mathsf{a}}$, c *arbre de* $\mathsf{T}^{\mathsf{a}}_{\Gamma}$:

$$\mathsf{c}_{\mathsf{requete}}(\mathsf{qu}^{\mathsf{s}}_{\phi(\mathsf{x})}(\mathsf{t})) = \mathsf{qu}^{\mathsf{a}}_{\mathsf{Enc}(\phi(\mathsf{x}))}(\mathsf{c}_{\mathsf{arbre}}(\mathsf{t}))$$

$${\mathsf{c}_{\mathsf{requete}}}^{-1}(\mathsf{qu}^{\mathsf{a}}_{\psi(\mathsf{x})}(\mathsf{c})) = \mathsf{qu}^{\mathsf{s}}_{\mathsf{Dec}(\psi(\mathsf{x}))}({\mathsf{c}_{\mathsf{arbre}}}^{-1}(\mathsf{t}))$$

Ce qui nous permet d'en déduire le lemme 3.4.

◄

Annexe B

Residual Finite Tree Automata

Authors : J. Carme, R. Gilleron, A. Lemay, A. Terlutte, M. Tommasi

Abstract : Tree automata based algorithms are essential in many fields in computer science such as verification, specification, program analysis. They become also essential for databases with the development of standards such as XML. In this paper, we define new classes of non deterministic tree automata, namely residual finite tree automata (RFTA). In the bottom-up case, we obtain a new characterization of regular tree languages. In the top-down case, we obtain a subclass of regular tree languages which contains the class of languages recognized by deterministic top-down tree automata. RFTA also come with the property of existence of canonical non deterministic tree automata.

B.1 Introduction

The study of tree automata has a long history in computer science ; see the survey of Thatcher [Thatcher, 1973], and the texts of F. Gécseg and M. Steinby [Gécseg and Steinby, 1984, Gécseg and Steinby, 1996], and of the TATA group [Comon et al., 1997]. With the advent of tree-based metalanguages (SGML and XML) for document grammars, new developments on tree automata formalisms and tree automata based algorithms have been done [Murata et al., 2001, Neven, 2002]. Also, because of the tree structure of documents, learning algorithms for tree languages have been defined for the tasks of information extraction and information retrieval [Fernau, 2002, Goldman and Kwek, 2002]. We are currently involved in a research project dealing with information extraction systems from semi-structured data. One objective is the definition of classes of tree automata satisfying two properties : there are efficient algorithms for membership and matching, and there are efficient learning algorithms for the corresponding classes of tree languages.

In the present paper, we only consider finite ranked trees. There are bottom-up (also known as frontier to root) tree automata and top-down (also known as root to frontier) tree automata. The top-down version is particularly relevant for some implementations because important properties such as membership[4] can be solved without handling the whole input tree into memory. There are also deterministic tree automata and non-deterministic tree automata. Determinism is important to reach efficiency for membership and other decision properties.

[4]given a tree automaton A, decide whether an input tree is accepted by A.

It is known that non-deterministic top-down, non-deterministic bottom-up, and deterministic bottom-up tree automata are equally expressive and define regular tree languages. But there is a tradeoff between efficiency and expressiveness because some regular (and even finite) tree languages are not recognized by deterministic top-down tree automata. Moreover, the size of a deterministic bottom-up tree automaton can be exponentially larger than the size of a non-deterministic one recognizing the same tree language. This drawback can be dramatic when the purpose is to build tree automata. This is for instance the case in the problem of tree pattern matching and in machine learning problems like grammatical inference.

The process of learning finite state machines from data is referred as grammatical inference. The first theoretical foundations were given by Gold [Gold, 1967] and first applications were designed in the field of pattern recognition. Grammatical inference mostly focused on learning string languages but recent works are concerned with learning tree languages [Sakakibara, 1990, Fernau, 2002, Goldman and Kwek, 2002]. In most works, the target tree language is represented by a deterministic bottom-up tree automaton. This is problematic because the time complexity of the learning algorithm depends on the size of the target automaton. Therefore, again it is crucial to define learning algorithms for non-deterministic tree automata. The reader should note that tree patterns [Goldman and Kwek, 2002] satisfy this property.

Therefore the aim of this article is to define non-deterministic tree automata corresponding to sufficiently expressive classes of tree languages and having nice properties from the algorithmic viewpoint and from the grammatical inference viewpoint. For this aim, we extend previous works from the string case [Denis et al., 2002b] to the tree case and we define residual finite state automata (RFTA). The reader should note that learning algorithms for residual finite string automata have been defined [Denis et al., 2001, Denis et al., 2002c].

In Section B.3, we study the bottom-up case. We define the residual language of a language L w.r.t a ground term t as the set of contexts c such that $c[t]$ is a term in L. We define bottom-up residual tree automata as automata whose states correspond to residual languages. Bottom-up residual tree automata are non-deterministic and recognize regular tree languages. We prove that every regular tree language is recognized by a unique canonical bottom-up residual tree automaton, minimal according to the number of states. We give an example of regular tree languages for which the size of the deterministic bottom-up tree automata grows exponentially with respect to the size of the canonical bottom-up residual tree automata.

In Section B.4, we study the top-down case. We define the residual language of a language L w.r.t a context c as the set of ground terms t such that $c[t]$ is a term in L. We define top-down residual tree automata as automata whose states correspond to residual languages. Top-down residual tree automata are non-deterministic tree automata. Interestingly, the class of languages recognized by top-down residual tree automata is strictly included in the class of regular tree languages and strictly contains the class of languages recognized by deterministic top-down tree automata. We also prove that every tree language in this family is recognized by a unique canonical top-down residual tree automaton; this automaton is minimal according to the number of states.

The definition of residual finite state automata comes with new decision problems. All of them rely on properties of residual languages. It is proved that all residual languages of a given tree language L can be built in both top-down and bottom-up cases. From these constructions we obtain positive answers to decision problems like 'decide whether an automaton is a (canonical) RFTA'. The exact complexity bounds are not given but we conjecture that are identical than in the string case.

The present work is connected with the paper by Nivat and Podelski [Nivat and Podelski, 1997]. They consider a monoid framework, whose elements are called pointed trees (contexts in our terminology, special trees in [Thomas, 1984]), to define tree automata. They define a Nerode congruence in the bottom-up case and in the top-down case. Their work leads to the generalization of the notion of deterministic to l-r-deterministic (context-deterministic in our terminology) for top-down tree automata. They have a minimization procedure for this class of automata. It should be noted that the class of languages recognized by context-deterministic tree automata (also called homogeneous tree languages) is strictly included in the class of languages recognized by residual top-down tree automata.

B.2 Preliminaries

We assume that the reader is familiar with basic knowledge about tree automata. We follow the notations defined in TATA [Comon et al., 1997].

A ranked alphabet is a couple $(\mathcal{F}, \mathit{Arity})$ where $\mathcal{F}$ is a finite set and *Arity* is a mapping from $\mathcal{F}$ into $\mathbb{N}$. The set of symbols of arity p is denoted by $\mathcal{F}_p$. Elements of arity 0, 1, ... p are respectively called constants, unary, ..., p-ary symbols. We assume that $\mathcal{F}$ contains at least one constant. In the examples, we use parenthesis and commas for a short declaration of symbols with arity. For instance, a is a constant and $f(,)$ is a short declaration for a binary symbol f. The set of *terms* over $\mathcal{F}$ is denoted by $\mathcal{T}(\mathcal{F})$. Let $\diamond$ be a special constant which is not in $\mathcal{F}$. The set of *contexts* (also known as pointed trees in [Nivat and Podelski, 1997] and special trees in [Thomas, 1984]), denoted by $\mathcal{C}(\mathcal{F})$, is the set of terms which contains exactly one occurrence of $\diamond$. The expression $c[\diamond]$ denotes a context, we only write c when there is no ambiguity. We denote by $c[t]$ the term obtained from $c[\diamond]$ by replacing $\diamond$ by a term t.

A *bottom-up Finite Tree Automaton* ($\uparrow$-FTA) over $\mathcal{F}$ is a tuple $A = (Q, \mathcal{F}, Q_f, \Delta)$ where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form $f(q_1, \ldots, q_n) \to q$ where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$. In this paper, the size of an automaton refers to its size in number of states, so two automaton which have the same number of states but different number of rules are considered as having the same size. When $n = 0$ a rule is written $a \to q$, where a is a constant. The *move relation* is written $\to_A$ and $\to_A^*$ is the reflexive and transitive closure of $\to_A$. A term t reaches a state q if and only if $t \to_A^* q$. A state q *accepts* a context c if and only if there exists a $q_f \in Q_f$ such that $c[q] \to_A^* q_f$. The automaton A recognizes a term t if and only if there exists a $q_f \in Q_f$ such that $t \to_A^* q_f$. The language recognized by A is the set of all terms recognized by A, and is denoted by $L(A)$.

Two $\uparrow$-FTA are equivalent if they recognize the same tree language. A $\uparrow$-FTA $A = (Q, \mathcal{F}, Q_f, \Delta)$ is *trimmed* if and only if all its states can be reached by at least one term and accepts at least one context. A $\uparrow$-FTA is *deterministic* ($\uparrow$-DFTA) if and only if there are no two rules with the same left-hand side in its set of rules. A tree language is *regular* if and only if it is recognized by a bottom-up tree automaton. As any $\uparrow$-FTA can be changed into an equivalent trimmed $\uparrow$-DFTA, any regular tree language can be recognized by a trimmed $\uparrow$-DFTA.

Let L be a tree language over a ranked alphabet $\mathcal{F}$ and t a term. The bottom-up residual language of L relative to a term t, denoted by $t^{-1}L$, is the set of all contexts in $\mathcal{C}(\mathcal{F})$ such that $c[t] \in L$:

$$t^{-1}L = \{c \in \mathcal{C}(\mathcal{F}) \mid c[t] \in L\}.$$

Note that a bottom-up residual language is a set of contexts, and not a tree language. The Myhill-Nerode congruence for tree languages can be defined by two terms t and t' are equivalent if they define the same residual languages. From the Myhill-Nerode theorem fro tree languages, we get the following result : a tree language is recognizable if and only if the number of residual languages is finite.

A *top-down finite tree automaton* ($\downarrow$-FTA) over $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ where Q is a set of states, $I \subseteq Q$ is a set of initial states, and Δ is a set of rewrite rules of the form $q(f) \to f(q_1, \ldots, q_n)$ where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$. Again, if $n = 0$ the rule is written $q(a) \to a$. The *move relation* is written $\to_A$ and $\to_A^*$ is the reflexive and transitive closure of $\to_A$. A state q accepts a term t if and only if $q(t) \to_A^* t$. A recognizes a term t if and only if at least one of its initial states accepts it. The language recognized by A is the set of all ground terms recognized by A and is denoted by $L(A)$.

Any regular tree language can be recognized by a $\downarrow$-FTA. This means that $\downarrow$-FTA and $\uparrow$-FTA have the same expressive power. A $\downarrow$-FTA is *deterministic* ($\downarrow$-DFTA) if and only if its set of rules does not contain two rules with the same left-hand side. Unlike $\uparrow$-DFTA, $\downarrow$-DFTA are not able to recognize all regular tree languages.

Let L be a tree language over a ranked alphabet $\mathcal{F}$, and c a context of $\mathcal{C}(\mathcal{F})$. The top-down residual language of L relative to c, denoted by $c^{-1}L$, is the set of ground terms t such that $c[t] \in L$:

$$c^{-1}L = \{t \in \mathcal{T}(\mathcal{F}) \mid c[t] \in L\}.$$

The definition of top-down residual languages comes with an equivalence relation on contexts. It is worth noting that it does not define a congruence over terms. Nonetheless, based on [Nivat and Podelski, 1997], it can be shown that a tree language L is regular if and only if the number of top-down residual languages associated with L is finite. In the proof, it is used that the number top-down residual languages is lower than the number of bottom-up residual languages.

B.3 Bottom-up residual finite tree automata

In this section, we introduce a new class of bottom-up finite tree automata, called bottom-up residual finite tree automata ($\uparrow$-RFTA). This class of automata shares some interesting properties with both bottom-up deterministic and non-deterministic finite tree automata which both recognize the class of regular tree languages.

On the one hand, as $\uparrow$-DFTA, $\uparrow$-RFTA admits a unique canonical form, based on a correspondence between states and residual languages, whereas $\uparrow$-FTA does not. On the other hand, $\uparrow$-RFTA are non-deterministic and can be much smaller in their canonical form than their deterministic counter-parts.

B.3.1 Definition and expressive power of bottom-up residual finite tree automata

First, let us precise the nature of this correspondence, then let us give the formal definition of $\uparrow$-residual tree automata and describe their properties.

In order to establish the nature of this correspondence between states and residual languages, let us introduce the notion of state languages. The *state language* C_q of a state q is the set of contexts accepted by the state q :

$$C_q = \{c \in \mathcal{C}(\mathcal{F}) \mid \exists q_f \in Q_f, c[q] \rightarrow_A^* q_f\}.$$

As shown by the following example, state languages are generally not residual languages :

Exemple B.1. *Consider the tree language* $L = \{f(a_1, b_1), f(a_1, b_2), f(a_2, b_2)\}$ *over* $\mathcal{F} = \{f(,), a_1, b_1, a_2, b_2\}$. *This language* L *is recognized by the tree automaton* $A = (\{q_1, q_2, q_3, q_4, q_5\}, \mathcal{F}, \{q_5\}, \Delta)$ *where* $\Delta = \{a_1 \rightarrow q_1, b_1 \rightarrow q_2, b_2 \rightarrow q_3, a_2 \rightarrow q_4, a_1 \rightarrow q_4, f(q_1, q_2) \rightarrow q_5, f(q_4, q_3) \rightarrow q_5\}$. *Residual languages of* L *are* $a_1^{-1}L = \{f(\diamond, b_1), f(\diamond, b_2)\}$, $b_1^{-1}L = \{f(a_1, \diamond)\}$, $b_2^{-1}L = \{f(a_1, \diamond), f(a_2, \diamond)\}$, $a_2^{-1}L = \{f(\diamond, b_2)\}$, $f(a_1, b_1)^{-1}L = \{\diamond\}$. *The state language of* q_1 *is* $\{f(\diamond, b_1)\}$, *which is not a residual language. The tree* a_1 *reaches* q_1, *so each context accepted by* q_1 *is an element of the residual language* $a_1^{-1}L$, *which means that* $C_{q_1} \subset a_1^{-1}L$. *But the reverse inclusion is not true because* $f(\diamond, b_2)$ *is not an element of* C_{q_1}. *The reader should note that this situation is possible because* A *is non-deterministic.*

In fact, it can be proved (the proof is omitted) that residual languages are unions of state languages. For any L recognized by a tree automaton A, we have

$$\forall t \in T(\mathcal{F}), t^{-1}L = \bigcup_{q \in Q,\, t \rightarrow_A^* q} C_q. \tag{B.1}$$

As a consequence, if A is deterministic and trimmed, each residual language is a state language and conversely.

We can define a new class of non-deterministic automata stating that each state language must correspond to a residual tree language. We have seen that residual tree languages are related to the Myhill-Nerode congruence and we will show that minimization of tree automata can be extended in the definition of a canonical form for this class of non-deterministic tree automata.

Définition B.1. *A bottom-up residual tree automaton (*$\uparrow$*-RFTA) is a* $\uparrow$*-FTA* $A = (Q, \mathcal{F}, Q_f, \Delta)$ *such that* $\forall q \in Q$, $\exists t \in T(\mathcal{F})$, $C_q = t^{-1}L(A)$.

According to the above definition and previous remarks, it can be shown that every trimmed $\uparrow$-DFTA is a $\uparrow$-RFTA. As a consequence, $\uparrow$-RFTA have the same expressive power than finite tree automata :

Théorème B.1. *The class of tree languages recognized by* $\uparrow$*-RFTA is the class of regular tree languages.*

As an advantage of $\uparrow$-RFTA, the number of states of an $\uparrow$-RFTA can be much smaller than the number of states of any equivalent $\uparrow$-DFTA :

Proposition B.1. *There exists a sequence* (L_n) *of regular tree languages such that for each* L_n, *the size of the smallest* $\uparrow$*-DFTA which recognizes* L_n *is an exponential function of* n, *and the size of the smallest* $\uparrow$*-RFTA which recognizes* L_n *is a linear function of* n.

Sketch of proof We give an example of regular tree languages for which the size of the ↑-DFTA grows exponentially with respect to the size of the equivalent canonical ↑-RFTA. A path is a sequence of symbols from the root to a leaf of a tree. The length of a path is the number of symbols on the path, except the root. Let $\mathcal{F} = \{f(,), a\}$ and let us consider the tree language L_n which contains exactly the trees with at least one path of length n. Let $A_n = (Q, \mathcal{F}, Q_f, \Delta)$ be a ↑-FTA defined by : $Q = \{q_*, q_0, \ldots, q_n\}, Q_f = \{q_0\}$ and

$$\Delta = \{a \to q_*, a \to q_n, f(q_*, q_*) \to q_*\} \cup \bigcup_{k \in [1,\ldots,n], q \in Q \setminus \{q_0\}}^{n} \{f(q_k, q) \to q_{k-1}, f(q, q_k) \to q_{k-1}, f(q_k, q) \to q_*, f(q, q_k) \to q_*\}$$

Let C_* be the set of contexts which contain at least one path of length n. Let C_i be the set of contexts whose path from the root to $\diamond$ is of length i. Let t_* be a term such that all its paths are of length greater than n. Note that the set of contexts c such that $c[t_*]$ belongs to L_n is exactly the set of contexts C_*. Let $t_0 \ldots t_n$ be terms such that for all $i \leq n$, t_i contains exactly one path of length smaller than n, and the length of this path is $n - i$. Therefore, $t_i^{-1} L_n$ is the set of contexts $C_* \cup C_i$.

One can verify that C_{q_*} is exactly $t_*^{-1} L_n = C_*$, and for all $i \leq n$, C_{q_i} is exactly $t_i^{-1} L_n = C_* \cup C_i$. The reader should note that rules of the form $f(q_k, q) \to q_*$ and $f(q, q_k) \to q_*$ are not useful to recognize L_n but they are required to obtain a ↑-RFTA (because C_i is not a residual language of L_n). So A_n is a ↑-RFTA and recognizes L_n. The size of A_n is $n + 2$.

The construction of the smallest ↑-DFTA which recognizes $L(A_n)$ is left to the reader. But, it can easily be shown that the number of states is in $O(2^n)$ because states must store lengths of all paths smaller than n. □

Unfortunately, the size of a ↑-RFTA can be exponentially larger than the size of an equivalent ↑-FTA.

B.3.2 The canonical form of bottom-up residual tree automata

As ↑-DFTA, ↑-RFTA have the interesting property to admit a canonical form. In the case of ↑-DFTA, there is a one-to-one correspondence between residual languages and state languages. This is a consequence of the Myhill-Nerode theorem for trees.

A similar result holds for ↑-RFTA. In a canonical ↑-RFTA, the set of states is in one-to-one correspondence with a subset of residual languages called prime residual languages.

Définition B.2. *Let L be a tree language. A bottom-up residual language of L is* composite *if and only if it is the union of the bottom-up residual languages that it strictly contains :*

$$t^{-1} L = \bigcup_{t'^{-1} L \subsetneq t^{-1} L} t'^{-1} L.$$

A residual language is prime *if and only if it is not composite.*

Exemple B.2. *Let us consider again the tree languages in the proof of Proposition B.1. Let Q_n be the set of states of A_n. All the $n + 2$ states $q_*, q_0, \ldots, q_n$ of Q_n have state languages which are prime residual languages. The subset construction applied on A_n to build a ↑-DFTA D_n leads to consider states which are subsets of Q. The state language of a state $\{q_{k_1} \ldots q_{k_n}\}$ is a composite residual language. It is the union of $t_{q_{k_1}}^{-1} L \ldots t_{q_{k_n}}^{-1} L$.*

In canonical ↑-RFTAs, all state languages are prime residual languages.

Théorème B.2. *Let L be a regular tree language and let us consider the ↑-FTA $A_{can} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by :*

- *Q is in bijection with the set of all prime bottom-up residual languages of L. We denote by t_q a ground term such that q is associated with $t_q^{-1}L$ in this bijection*
- *Q_f is the set of all elements q of Q such that $t_q^{-1}L$ contains the void context $\diamond$,*
- *Δ contains all the rules $f(q_1, \ldots, q_n) \to q$ such that $t_q^{-1}L \subseteq (f(t_{q_1}, \ldots, t_{q_n}))^{-1}L$ and all the rules $a \to q$ such that $a \in \mathcal{F}_0$ and $t_q^{-1}L \subseteq a^{-1}L$.*

A_{can} is a ↑-RFTA, it is the smallest ↑-RFTA in number of states which recognizes L, and it is unique up to a renaming of its states.

Sketch of proof There are three things to prove in this theorem : the canonical ↑-RFTA $A_{can} = (Q, \mathcal{F}, Q_f, \Delta)$ of a regular tree language L recognizes L, it is a ↑-RFTA, and there cannot be any strictly smaller ↑-RFTA which recognizes L. The three points are proved in this order.

We first have to prove the equality $L(A_{can}) = L$. It follows from the identity $(\circledast)$ $\forall t,\ t^{-1}L = \bigcup_{q \in Q,\, t \to^*_{A_{can}} q} t_q^{-1}L$ which can be proved inductively on the height of t. Using this property, we have :

$$t \in L \Leftrightarrow \diamond \in t^{-1}L \underset{\circledast}{\Leftrightarrow} \diamond \in \bigcup_{q \in Q,\, t \to^*_{A_{can}} q} t_q^{-1}L \Leftrightarrow \exists q_f \in Q_f, t \to^*_{A_{can}} q_f \Leftrightarrow t \in L(A_{can})$$

The equality between L and $L(A_{can})$ helps us to prove the characterization of ↑-RFTA : $t_q^{-1}L = C_q^{A_{can}}$ where $C_q^{A_{can}}$ is the state language of q in A_{can}.

The last point can be proved in such a way. In a ↑-RFTA, any residual language is a union of state languages, and any state language is a residual language. So any prime residual language is a state language, so there is at least as much states in a ↑-RFTA as prime residual languages admitted by its corresponding tree language.

□

The canonical automaton is uniquely defined determined by the tree language under consideration, but there may be other automata which have the same number of states. The canonical ↑-RFTA is unique because it has the maximum number of rules. Even though all its states are associated to prime residual languages, the automaton considered in the proof of Proposition B.1 is not the canonical one because some rules are missing : $\bigcup_{k=1}^{n} \{f(q_k, q_0) \to q_{k-1}, f(q_0, q_k) \to q_{k-1}\}$ and $\bigcup_{q \in Q} \{f(q, q_0) \to q_*, f(q, q_0) \to q_*\}$.

B.4 Top-Down residual finite tree automata

The definition of top-down residual finite tree automata (↓-RFTA) is tightly correlated with the definition of ↑-RFTA. Similarly to ↑-RFTA, ↓-RFTA are defined as non-deterministic tree automata where each state language is a residual language. Any ↓-RFTA can be transformed in a canonical equivalent ↓-RFTA — minimal in the number of states and unique up to state renaming.

The main difference between the bottom-up and the top-down case is in the problem of the expressive power of tree automata. The three classes of bottom-up tree automata, $\uparrow$-DFTA, $\uparrow$-RFTA or $\uparrow$-FTA, have the same expressive power. In the top-down case, deterministic, residual and non-deterministic tree automata have different expressive power. This makes the canonical form of $\downarrow$-RFTA more interesting. Compared to the minimal form of $\downarrow$-DFTA, it can be smaller when both exist, and it exists for a wider class of tree languages.

Let us introduce $\downarrow$-RFTA through their similarity with $\uparrow$-RFTA, then study this specific problem of expressiveness.

B.4.1 Analogy with bottom-up residual tree automata

Let us formally define state languages in the top-down case :

Définition B.3. *Let L be a regular tree language over a ranked alphabet $\mathcal{F}$, let A be a top-down tree automaton which recognizes L, and let q be a state of this automaton. The state language of L relative to q, written L_q, is the set of terms which are accepted by q :*

$$L_q = \{t \in \mathcal{T}(\mathcal{F}) \mid q(t) \rightarrow_A^* t\}.$$

It follows from this definition some properties similar to those already studied in the previous section. Firstly, state languages are generally not residual languages. Secondly, residual languages are unions of state languages. Let us define Q_c :

$$Q_c = \{q \mid q \in Q, \exists q_i \in I, q_i(c[\diamond]) \rightarrow_A^* c[q(\diamond)]\}.$$

We have the following relation between state languages and residual languages.

Lemme B.1. *Let L be a tree language and let $A = (Q, \mathcal{F}, I, \Delta)$ be a top-down tree automaton which recognizes L. Then $\forall c \in \mathcal{C}(\mathcal{F}), \bigcup_{q \in Q_c} L_q = c^{-1}L$.*

These similarities lead us to this definition of top-down residual tree automata :

Définition B.4. *A top-down Residual Finite Tree Automaton ($\downarrow$-RFTA) recognizing a tree language L is a $\downarrow$-FTA $A = (Q, \mathcal{F}, I, \Delta)$ such that : $\forall q \in Q$, $\exists c \in \mathcal{C}(\mathcal{F})$, $L_q = c^{-1}L$.*

Languages defined in the proof of Proposition B.1 are still interesting here to define examples of top-down residual tree automata :

Exemple B.3. *Let us consider again the family of tree languages L_n, and the family of corresponding $\uparrow$-RFTA A_n. For every n, let A'_n be the $\downarrow$-RFTA defined by : $Q = \{q_*, q_0, \ldots, q_n\}, Q_i = \{q_0\}$ and $\Delta = \{q_*(a) \rightarrow a, q_n(a) \rightarrow a, q_*(f) \rightarrow f(q_*, q_*)\} \cup \bigcup_{k=1}^{n} \{q_{k-1}(f) \rightarrow f(q_k, q_*), q_{k-1}(f) \rightarrow f(q_*, q_k)\}$.*

For every $k \leq n$, the state language of q_k is equal to L_{n-k}. And, L_{n-k} is the top-down residual language of c_k, where c_k is a context whose height from the root to the special constant $\diamond$ is k and c_k does not contain any path whose length is smaller or equal to n. The state language of q_* is $\mathcal{T}(\mathcal{F})$. And, $\mathcal{T}(\mathcal{F})$ is the top-down residual language of L_n relative to c_*, where c_* is a context who contains a path whose length is n. So A'_n is a $\downarrow$-RFTA. Moreover, it is easy to verify that A'_n recognizes L_n.

B.4.2 The expressive power of top-down tree automata

Top-down deterministic automata and path-closed languages

A tree language L is *path-closed* if :

$$\forall c \in C(\mathcal{F}), c[f(t_1, t_2)] \in L \wedge c[f(t'_1, t'_2)] \in L \Rightarrow c[f(t_1, t'_2)] \in L.$$

The reader should note that the definition only considers binary symbols, the definition can easily be extended to n-ary symbols. The class of languages that $\downarrow$-DFTA can recognize is the class of path-closed languages [Viragh, 1981].

Context-deterministic automata and homogeneous languages.

Podelski and Nivat in [Nivat and Podelski, 1997] have defined *l-r-deterministic* top-down tree automata. In the present paper, let us call them top-down *context-deterministic* tree automata.

Définition B.5. *A top-down context-deterministic tree automaton ($\downarrow$-CFTA) A is a $\downarrow$-FTA such that for every context $c \in \mathcal{C}(\mathcal{F})$, Q_c is either the empty set or a singleton set.*

An *homogeneous language* is a tree language L satisfying :

$$\forall c \in C(\mathcal{F}), c[f(t_1, t_2)] \in L \wedge c[f(t_1, t'_2)] \in L \wedge c[f(t'_1, t_2)] \Rightarrow c[f(t'_1, t'_2)] \in L.$$

Again, the definition can easily be extended from the binary case to n-ary symbols. They have shown that the class of languages recognized by $\downarrow$-CFTA is the class of homogeneous languages.

The hierarchy

A $\downarrow$-DFTA is a $\downarrow$-CFTA. For $\downarrow$-CFTA and $\downarrow$-RFTA, we have the following result :

Lemme B.2. *Any trimmed $\downarrow$-CFTA is a $\downarrow$-RFTA.*

Preuve : Let $A = (Q, \mathcal{F}, I, \Delta)$ be a trimmed $\downarrow$-CFTA recognizing a tree language L. As A is trimmed, all states are reachable, so for every q, there exists a c such that $q \in Q_c$. Then, by definition of a $\downarrow$-CFTA, for every q, there exists a c such that $\{q\} = Q_c$. Using Lemma B.1, we have :

$$\forall q \in Q, \exists c \in \mathcal{C}(\mathcal{F}), L_q = c^{-1}L.$$

stating that A is a $\downarrow$-RFTA. □ ◄

Therefore, if we denote by $\mathcal{L}_{\mathcal{C}}$ the class of tree languages recognized by a class of automata $\mathcal{C}$, we obtain the following hierarchy :

$$\mathcal{L}_{\downarrow-DFTA} \subseteq \mathcal{L}_{\downarrow-CFTA} \subseteq \mathcal{L}_{\downarrow-RFTA} \subseteq \mathcal{L}_{\downarrow-FTA}$$

The hierarchy is strict

- Let $L = \{f(a,b), f(b,a)\}$. L_1 is homogeneous but not path-closed. Therefore L can be recognized by a $\downarrow$-CFTA, but can not be recognized by a $\downarrow$-DFTA.
- The tree languages L_n in the proof of Proposition B.1 are not recognized by $\downarrow$-CFTA. We can easily verify that L_n is not homogeneous. Indeed, if t is a term which has a path whose length is equal to $n-1$, and t' a term which does not have any path whose length is smaller than n, $f(t,t)$, $f(t,t')$, $f(t',t)$ belong to L_n, but $f(t',t')$ does not. And, we have already shown that L_n is recognized by a $\downarrow$-RFTA.
- Let $L' = \{f(a,b), f(a,c), f(b,a), f(b,c), f(c,a), f(c,b)\}$. L' is a finite language, therefore it is a regular tree language which can be recognized by a $\downarrow$-FTA. L' cannot be recognized by a $\downarrow$-RFTA. To prove that, let us consider A' a $\downarrow$-FTA which recognizes L'. The top-down residual languages of L' are $\{a,b\}$, $\{a,c\}$, $\{b,c\}$ and L'. As A' recognizes L', it recognizes $f(a,b)$. This implies the existence of three states q_1, q_2, q_3 and three rules $q_1(f) \to f(q_2,q_3)$, $q_2(a) \to a$, and $q_3(b) \to b$. If A' was a $\downarrow$-RFTA, then q_2 would accept a residual language. As q_2 accepts a, it would accept either $\{a,b\}$ or $\{a,c\}$. Similarly, q_3 would accept either $\{a,b\}$ or $\{b,c\}$. In these conditions, and thanks to the rule $q_1(f) \to f(q_2,q_3)$, A' would recognize $f(a,a)$, $f(b,b)$ or $f(c,c)$. So A' cannot be a $\downarrow$-RFTA.

Therefore, we obtain the following result :

Théorème B.3. $\mathcal{L}_{\downarrow-DFTA} \subsetneq \mathcal{L}_{\downarrow-CFTA} \subsetneq \mathcal{L}_{\downarrow-RFTA} \subsetneq \mathcal{L}_{\downarrow-FTA}$

So top-down residual tree automata are strictly more expressive than context-deterministic tree automata. But as far as we know, there is no straightforward characterization of the tree languages recognized by $\downarrow$-RFTA.

B.4.3 The canonical form of top-down residual tree automata

The problem of the canonical form of top-down tree automata is similar to the bottom-up case. Whereas there is no way to reduce a non-deterministic top-down tree automaton to a unique canonical form, a top-down residual tree automaton can take such a form. Its definition is similar to the definition of the canonical bottom-up tree automaton.

In the same way that we have defined composite bottom-up residual language, a top-down residual language of L is *composite* if and only if it is the union of the top-down residual languages that it strictly contains and a residual language is *prime* if and only if it is not composite.

Théorème B.4. *Let L be a tree language in the class $\mathcal{L}_{\downarrow-RFTA}$. Let us consider the $\downarrow$-RFTA $A_{can} = (Q, \mathcal{F}, I, \Delta)$ defined by :*

- *Q is a set of state in bijection with the prime residual languages of L. For each of these residual languages, there exists a c_q such that q is associated with $c_q^{-1}L$ in this bijection.*
- *I is the set of prime residuals which are subsets of L.*
- *Δ contains all the rules $q(a) \to a$ such that a is a constant and $c_q[a] \in L$, and all the rules $q(f) \to f(q_1, \ldots, q_n)$ such that for all $t_1 \ldots t_n$ where $t_i \in c_{q_i}^{-1}L$, $c_q[f(t_1, \ldots, t_n)] \in L$.*

A_{can} is a $\downarrow$-RFTA, it is the smallest $\downarrow$-RFTA in number of states which recognizes L, and it is unique up to a renaming of its states.

Sketch of proof

The proof is mainly based on this lemma : $t \in c_q^{-1}L \Leftrightarrow t \in L_q^{A_{can}}$

where $L_q^{A_{can}}$ is the state language of q in the automaton A_{can}.

This lemma is proved by induction on the height of t. This is not a straightforward induction. It involves the rules of a $\downarrow$-RFTA automaton A' which recognizes L. Its existence is granted by the hypothesis of the theorem.

Once this is proved, it can be easily deduced that A_{can} recognizes L and is a RFTA. As there is one state per prime residual in A_{can}, it is minimal in number of states.

□

B.5 Decidability issues

Some decision problems naturally arise with the definition of RFTA. Most of these problems are solved just noting that one can build all residual languages of a given regular language L defined by a non-deterministic tree automaton. In the bottom-up case, the state languages of the minimal $\uparrow$-RFTA which recognizes L are exactly the residual languages of L, and this automaton can be built with the subset construction. In the top-down case, the subset construction does not necessarily gives us an automaton which recognizes exactly L, but it gives us the set of all residual languages. Therefore, knowing whether a tree automaton is a RFTA, whether a residual language is prime or composite, and whether a tree automaton is a canonical RFTA are decidable. These problems have not been deeply studied in terms of complexity, but they are at least as hard as the similar problems with strings, that is they are PSPACE-hard ([Denis et al., 2002b]).

B.6 Conclusion

We have defined new classes of non-deterministic tree automata. In the bottom-up case, we get another characterization of regular tree languages. More interestingly, in the top-down case, we obtain a subclass of the regular tree languages. For both cases, we have a canonical form and the size of residual tree automata can be much smaller than equivalent (when exist) deterministic ones.

We are currently extending these results to the case of unranked trees because our application domain is concerned with html and xml documents. Also, we are designing learning algorithms for residual finite tree automata extending previous algorithms for residual finite string automata [Denis et al., 2001, Denis et al., 2002c].

B.7 Appendix

B.7.1 Proof of Equation (B.1)

Let L be a tree language and $(Q, \mathcal{F}, Q_f, \Delta)$ a $\uparrow$-FTA which recognizes it. We show that $\forall t \in T(\mathcal{F}), t^{-1}L = \bigcup_{t \rightarrow_A^* q} C_q$.

Let $t \in \mathcal{T}(\mathcal{F})$, and $c \in t^{-1}L$. $c[t] \in L$, so there exists $q_f \in Q_f$ and $q \in Q$ such that $c[t] \rightarrow_A^* c[q] \rightarrow_A^* q_f$, where $t \rightarrow_A^* q$ and $c \in C_q$. So $c \in \bigcup_{t \rightarrow_A^* q} C_q$. So $t^{-1}L \subseteq \bigcup_{t \rightarrow_A^* q} C_q$

Let $t \in \mathcal{T}(\mathcal{F})$, and $c \in \bigcup_{t \rightarrow_A^* q} C_q$. There exists a $q \in Q$ such that $t \rightarrow_A^* q$ and $c \in C_q$. So there exists $q_f \in Q_f$ such that $c[t] \rightarrow_A^* c[q] \rightarrow_A^* q_f$. So $c \in t^{-1}L$. So $\bigcup_{t \rightarrow_A^* q} C_q \subseteq t^{-1}L$

B.7.2 Proof of the theorem B.2

Théorème B.5. *The canonical $\uparrow$-RFTA recognizing a regular tree language is the smallest $\uparrow$-RFTA which recognizes it. Therefore, $\uparrow$-RFTA accepts a unique and minimal representation.*

The first point we have to demonstrate in this theorem is that the canonical $\uparrow$-RFTA that we have defined recognizes L.

Before this demonstration, we need to establish two properties of residual languages :

Lemme B.3. *Let L a regular language.*

$$\forall i, 1 \leq i \leq n, t_i^{-1}L \subseteq t_i'^{-1}L \Rightarrow f(t_1, \ldots, t_n)^{-1}L \subseteq f(t'_1, \ldots, t'_n)^{-1}L$$

Preuve :

This lemma can be proven inductively on i. Let $t_1 \ldots t_n$ such that for all i, $t_i^{-1}L$ is a subset of $t_i'^{-1}L$. Let c in $f(t_1, \ldots, t_n)^{-1}L$. Let us assume that $c[f(t'_1, \ldots, t'_{i-1}, t_i, \ldots, t_n)] \in L$. This implies that $c[f(t'_1, \ldots, t'_{i-1}, \diamond, t_{i+1}, \ldots, t_n)] \in t_i^{-1}L$, and therefore $c[f(t'_1, \ldots, t'_{i-1}, \diamond, t_{i+1}, \ldots, t_n)] \in t_i'^{-1}L$.

So $c[f(t'_1, \ldots, t'_i, t_{i+1}, \ldots, t_n)] \in L$. Inductively, $c[f(t'_1, \ldots, t'_n)] \in L$.

So $f(t_1, \ldots, t_n)^{-1}L \subseteq f(t'_1, \ldots, t'_n)^{-1}L$.

□ ◄

Lemme B.4.

$$\forall i, 1 \leq i \leq n, t_i^{-1}L = \bigcup_{j_i} t_{i,j_i}^{-1}L \Rightarrow f(t_1, \ldots, t_n)^{-1}L = \bigcup_{j_1 \ldots j_n} f(t_{1,j_1}, \ldots, t_{n,j_n})^{-1}L$$

Here, $\bigcup_{j_1 \ldots j_n}$ has to be understood as 'the union of all the possible combination of $j_1 \ldots j_n$'.

Preuve :

Let $t_1 \ldots t_n$ and for all $i \leq n$, $t_{i,1} \ldots t_{i,m_i}$ such that $t_i^{-1}L = \bigcup_{1 \leq j_i \leq m_i} t_{i,j_i}^{-1}L$.

$$\forall t_{1,j_1} \ldots t_{n,j_n}, \forall i \leq n, t_{i,j_i}^{-1}L \subseteq t_i^{-1}L \Rightarrow_{lemmaB.3}$$
$$\forall t_{1,j_1} \ldots t_{n,j_n}, f(t_{1,j_1} \ldots t_{n,j_n})^{-1}L \subseteq f(t_1, \ldots, t_n)^{-1}L \Rightarrow$$
$$\bigcup_{j_1 \ldots j_n} f(t_{1,j_1}, \ldots, t_{n,j_n})^{-1}L \subseteq f(t_1, \ldots, t_n)^{-1}L$$

Now, let c in $f(t_1, \ldots, t_n)^{-1}L$.

$$c[f(t_1, \ldots, t_n)] \in L \Rightarrow c[f(\diamond, t_2, \ldots, t_n)] \in t_1^{-1}L$$

As $t_1^{-1}L = \bigcup t_{1,j}^{-1}L$, there exists t_{1,m_1} such that $c[f(\diamond, t_2, \ldots, t_n)] \in t_{1,m_1}^{-1}L$. So $c[f(t_{1,m_1}, t_2, \ldots, t_n)] \in L$.

It can be proven inductively on i that there exists $t_{1,m_1} \dots t_{n,m_n}$ such that $c[f(t_{1,k_1}, \dots, t_{n,m_n})] \in L$. So $c \in f(t_{1,m_1}, \dots, t_{n,m_n})^{-1}L$. So :

$$f(t_1, \dots, t_n)^{-1}L \subseteq \bigcup_{j_1 \dots j_n} f(t_{1,j_1}, \dots, t_{n,j_n})^{-1}L$$

◀ □

Now, we can prove inductively this lemma, which is the main step to prove the equality between L and $L(A_{can})$

Lemme B.5.

$$\forall t, t^{-1}L = \bigcup_{t \to^*_{A_{can}} q} t_q^{-1}L$$

Preuve :

Let us prove this lemma inductively. Let $h(t)$ be the height of t.

Let us assume that $h(t) = 1$, so $t = a$ where a is a constant. A residual is a union of prime residuals, so :

$$a^{-1}L = \bigcup_{t_q^{-1}L \subseteq a^{-1}L} t_q^{-1}L$$

As $t_q^{-1}L \subseteq a^{-1}L$ if and only if A_{can} contains the rule $a \to q$:

$$a^{-1}L = \bigcup_{a \to^*_{A_{can}} q} t_q^{-1}L$$

Now let us assume that for any term t such that $h(t) \leq k$, lemma B.5 is true.

Let $t = f(t_1, \dots, t_n)$ such that $h(t) = k + 1$.

$$h(t) = k+1 \Rightarrow \forall i \leq n, h(t_i) = k \Rightarrow \forall i, t_i^{-1}L = \bigcup_{t_i \to^*_{A_{can}} q_{i,j_i}} t_{q_{i,j_i}}^{-1}L \Rightarrow_{lemmaB.4}$$

$$t^{-1}L = \bigcup_{t_i \to^*_{A_{can}} q_{i,j_i}} f(t_{1,q_1}, \dots, t_{n,q_n})^{-1}L$$

Any residual is a union of prime residuals, so for all $j_1 \dots j_n$:

$$f(t_{q_{1,j_1}}, \dots, t_{q_{n,j_n}})^{-1}L = \bigcup_{t_q^{-1}L \subseteq f(t_{q_{1,j_1}}, \dots, t_{q_{n,j_n}})^{-1}L} t_q^{-1}L$$

So :

$$t^{-1}L = \bigcup_{t_i \to^*_{A_{can}} q_{i,j_i}} f(t_{1,q_1}, \dots, t_{n,q_n})^{-1}L \Rightarrow$$

$$t^{-1}L = \bigcup_{t_i \to^*_{A_{can}} q_{i,j_i}} \left(\bigcup_{t_q^{-1}L \subseteq f(t_{q_{1,j_1}}, \dots, t_{q_{n,j_n}})^{-1}L} t_q^{-1}L \right)$$

As $t_q^{-1}L \subseteq f(t_{q_{1,j_1}}, \ldots, t_{q_{n,j_n}})^{-1}L$ if and only if A_{can} contains the rule $f(q_{1,j_1}, \ldots, q_{n,j_n}) \rightarrow q$,

$$t^{-1}L = \bigcup_{t \rightarrow^*_{A_{can}} q} t_q^{-1}L$$

◀ □

The equality between L and $L(A_{can})$ is formalized as such :

Lemme B.6. *The canonical $\uparrow$-RFTA $\mathcal{A}_{can}$ of a language L recognizes L, that is :*

$$\exists q_f \in Q_f, t \rightarrow^*_{A_{can}} q_f \Leftrightarrow t \in L$$

Preuve :

Let $t \in \mathcal{T}(\mathcal{F})$ and $q_f \in Q_f$ such that $t \rightarrow^*_{A_{can}} q_f$.

$$t \rightarrow^*_{A_{can}} q_f \Rightarrow t_{q_f}^{-1}L \subset \bigcup_{t \rightarrow^*_{A_{can}} q_j} t_{q_j}^{-1}L \Rightarrow_{lemmaB.5} t_{q_f}^{-1}L \subset t^{-1}L$$

As $\diamond \in t_{q_f}^{-1}L$, $\diamond \in t^{-1}L$, so $t \in L$.

Let $t \in L$.

$$\diamond \in t^{-1}L \Rightarrow \exists q_j \mid t \rightarrow^*_{A_{can}} q_j \wedge \diamond \in t_{q_j}^{-1}L \Rightarrow$$
$$\exists q_j \mid t \rightarrow^*_{A_{can}} q_j \wedge q_j \in Q_f$$

◀

Now, we have to prove that the canonical $\uparrow$-RFTA is a $\uparrow$-RFTA. In order to do this, we need to establish this lemma :

Lemme B.7. *Let $t_q^{-1}L$ and $t_{q'}^{-1}L$ be prime bottom-up residual languages of L. Let $C_q^{A_{can}}$ and $C_{q'}^{A_{can}}$ be sets of contexts accepted by q and q' in then canonical automaton of L A_{can}. Then :*

$$t_{q'}^{-1}L \subset t_q^{-1}L \Rightarrow C_{q'}^{A_{can}} \subset C_q^{A_{can}}$$

Preuve : Let t_q and $t_{q'}$ such that $t_{q'}^{-1}L \subset t_q^{-1}L$. For all $t_{q_1} \ldots t_{q_n}$, $f(t_{q_1}, \ldots, t_{q'}, \ldots, t_{q_n})^{-1}L \subset f(t_{q_1}, \ldots, t_q, \ldots, t_{q_n})$ (lemma B.3).

The construction of the set of rules of the canonical automaton implies that :

$$f(q_1, \ldots, q_n) \rightarrow q' \in \Delta \Leftrightarrow t_{q'}^{-1}L \subseteq f(t_{q_1}, \ldots, t_{q_n})^{-1}L$$

So :

$$f(q_1, \ldots, q', \ldots, q_n) \rightarrow q'' \in \Delta \Rightarrow$$
$$t_{q''}^{-1}L \in f(t_{q_1}, \ldots, t_{q'}, \ldots, t_{q_n})^{-1}L \Rightarrow$$
$$t_{q''}^{-1}L \in f(t_{q_1}, \ldots, t_q, \ldots, t_{q_n})^{-1}L \Rightarrow$$

$$f(q_1, \ldots, q, \ldots, q_n) \to q'' \in \Delta$$

So each context accepted by q' is accepted by q.

So $C_{q'}^{A_{can}} \subset C_q^{A_{can}}$. □ ◄

Lemme B.8. *The canonical RFTA A_{can} of a language L is a residual finite tree automata.*

Preuve :

Let $t_q^{-1}L$ be a prime residual language of L. Thanks to lemma B.5 :

$$t_q^{-1}L = \bigcup_{t_q \to^*_{A_{can}} q'} t_{q'}^{-1}L$$

If $t_q^{-1}L$ would strictly contain all the $t_{q'}^{-1}L$ of the union, it would be composite. As it is prime, $t_q^{-1}L$ is itself an element of this union, so $t_q \to^*_{A_{can}} q$.

Equation (B.1) *tells us that :*

$$t_q^{-1}L = \bigcup_{t_q \to^*_{A_{can}} q'} C_{q'}$$

So $C_q \subset t_q^{-1}L$.

For all q' such that $t_q \to^*_{A_{can}} q'$, $t_{q'}^{-1}L \subset t_q^{-1}L$, so $C_{q'} \subset C_q$ (lemma B.7). As the union of all $C_{q'}$ is equal to $t_q^{-1}L$, $t_q^{-1}L \subset C_q$

So $t_q^{-1}L = C_q$, so every prime residual language is accepted by its corresponding state.

So A_{can} is a RFTA.

□ ◄

Lemme B.9. *The canonical RFTA A_{can} of a language L is the smallest RFTA which recognizes L.*

Preuve :

Let A_{can} be the canonical RFTA of a language L, and t such that $\nexists q \in Q, t^{-1}L = C_q$.

Thanks to lemma B.5, $t^{-1}L = \bigcup_{t \to^*_{A_{can}} q} t_q^{-1}L$. As $\nexists q \in Q, t^{-1}L = t_q^{-1}L$, $t^{-1}L$ is a union of residuals that it strictly contains. So $t^{-1}L$ is a composite residual.

So for all prime residuals $t^{-1}L$, there is a q such that $t^{-1}L = C_q$. A_{can} contains as much states as prime residuals in L, so it is the smallest RFTA which recognizes L.

□ ◄

B.7.3 Proof of the theorem B.4

Théorème B.6. *Let L be a language recognized by a $\downarrow$-RFTA. The canonical top-down residual tree automaton of L is the smallest $\downarrow$-RFTA which recognizes L.*

In order to prove this theorem, let us firstly prove these lemma :

Lemme B.10. *Let $A = (Q, \mathcal{F}, I, \Delta)$ be a $\downarrow$-RFTA which recognizes L. For any prime residual $c^{-1}L$, there exists a state $q \in Q$ such that $L_q = c^{-1}L$.*

Preuve :

Let c be a context of L such that $\nexists q \in Q, c^{-1}L = L_q$.

Lemma B.1 implies that $c^{-1}L = \bigcup_{q \in Q_c} L_q$ and none of these L_q are equal to $c^{-1}L$. As $\forall q \in Q, L_q = c_q^{-1}L$, we have $c^{-1}L = \bigcup_{q \in Q_c} c_q^{-1}L$ where none of the $c_q^{-1}L$ are equal to $c^{-1}L$. So $c^{-1}L$ is composite.

□ ◄

Now, let us make the main part of the demonstration : let us prove that each prime residual language is exactly accepted by a state of the canonical $\downarrow$-RFTA.

Lemme B.11. *Let L be a language recognized by a $\downarrow$-RFTA. Let $A_{can} = (Q, \mathcal{F}, I, \Delta)$ be its canonical automaton. For all q in Q, $c_q^{-1}L = L_q$.*

Preuve :

As seen in the definition, Q is in bijection with the set of all residual languages, so for all q there exists a corresponding $c_q^{-1}L$. Let us prove inductively on the height of t that $t \in c_q^{-1}L \Leftrightarrow t \in L_{A_{can},q}$. Let us call $H(n)$ this hypothesis when $h(t) \leq n$.

Firstly, let us prove $H(1)$.

Let t such that $h(t) = 1$ and $t \in c_q^{-1}L$. As $h(t) = 1$, $t = a$ where a is a constant. As $t \in c_q^{-1}L$, $c_q[a] \in L$. So Δ contains the rule $q(a) \to a$, so $t \in L_{A_{can},q}$. Reciprocally, $t \in L_{A_{can},q}$ where $t = a$ implies that Δ contains the rule $q(a) \to a$. This rule exists in the canonical automata if and only if a is a constant and $c_q[a] \in L$. So $c_q[a] \in L$, so $t \in c_q^{-1}L$.

Now, let us assume that $H(l)$ is true when $l < k$. Let us prove that $H(k)$ is true.

Let $t = f(t_1, \ldots, t_n) \in c_q^{-1}L$ such that $h(t) = k$. For all t_i where $1 \leq i \leq n$, $t_i \in c_q[f(t_1, \ldots, t_{i-1}, \diamond, t_{i+1}, \ldots, t_n)]^{-1}L$.

Now, let us consider $A' = (Q', \mathcal{F}, I', \Delta')$ a $\downarrow$-RFTA which recognizes L. As L is recognized by a $\downarrow$-RFTA, A' exists. We will use this automaton to prove the existence of a rule $q \to f(q_1, \ldots, q_n)$ such that for all i, $q_i[t_i] \to_A^* t_i$ in A_{can}.

As $c_q^{-1}L$ is prime, there exists a $q' \in Q'$ such that $L_{A',q'} = c_q^{-1}L$ (lemma B.10). As $t \in c_q^{-1}L$, there exists in Δ' a rule $q' \to f(q'_1, \ldots, q'_n)$ such that for all i, $1 \leq i \leq n$, we have $t_i \in L_{A',q'_i}$.

For all $t'_1 \ldots t'_n$ such that $t'_i \in L_{A',q'_i}$, $f(t'_1, \ldots, t'_n) \in c_q^{-1}L$.

As L_{A',q'_i} is a residual, it is either a prime residual or a composite residual. If it is a prime residual, there exists a $q_i \in Q$ such that $L_{A',q'_i} = c_{q_i}^{-1}L$ and $t_i \in c_{q_i}^{-1}L$. If it is a composite residual, there exists a $q_i \in Q$ such that $c_{q_i}^{-1}L \subset L_{A',q'_i}$ and $t_i \in c_{q_i}^{-1}L$.

So there exists $q_1 \ldots q_n$ such that $t_i \in c_{q_i}^{-1}L \subset L_{A',q'_i}$. So for all $t'_1 \ldots t'_n$ in $c_{q_1}^{-1}L \ldots c_{q_n}^{-1}L$, $f(t'_1, \ldots, t'_n) \in L_{A',q'}^{-1}L = c_q^{-1}L$. So the rule $q(f) \to f(q_1, \ldots, q_n)$ exists in Δ.

For all t_i, $h(t_i) < k$, so as we have assumed that $H(l)$ is right when $l < k$, $H(h(t_i))$ is right. So for all i, $t_i \in L_{A_{can},q_i}$. As $q(f) \to f(q_1, \ldots, q_n)$, $t \in L_{A_{can},q}$.

We have proven that $t \in c_q^{-1}L \Rightarrow t \in L_{A_{can},q}$. *Now let us prove that* $t \in L_{A_{can},q} \Rightarrow t \in c_q^{-1}L$.

Let $t = f(t_1, \ldots, t_n) \in L_{A_{can},q}$ *such that* $h(t) = k$.

There exist $q_1 \ldots q_t$ *such that* $q(f(t_1 \ldots t_n)) \rightarrow_A^* f(q_1(t_1), \ldots, q_n(t_n)) \rightarrow_A^* f(t_1, \ldots, t_n)$. *For all* i, $t_i \in L_{A_{can},q_i}$ *and* $h(t_i) < k$, *so* $H(h(t_i))$ *is assumed to be true, so* $t_i \in c_{q_i}^{-1}L$. *The existence of the rule* $q(f) \rightarrow f(q_1, \ldots, q_n)$ *in* Δ *implies that for all* $t'_1 \ldots t'_n$ *such that* $t'_i \in c_{q_i}^{-1}L$, $c_q[f(t'_1, \ldots, t'_n)] \in L$. *So* $t \in c_q^{-1}L$.

So H(k) is true. We have proven inductively that for any t, $t \in L_{A_{can},q} \Leftrightarrow t \in c_q^{-1}L$.

□ ◄

Lemme B.12. $A_{can} =< Q, \mathcal{F}, Q_i, \Delta >$ *is a* $\downarrow$*-RFTA, recognizes* L, *and is minimal in number of states.*

Preuve :

Let us prove that lemma B.11 implies that $L(A_{can}) = L$. *Let* $t \in L$. $\diamond^{-1}L = L$ *is a residual, so it is a union of prime residuals. So there exists* $q_i \in Q$ *such that* $t \in c_{q_i}^{-1}L$ *and* $c_{q_i}^{-1}L \subseteq L$. *As* $c_{q_i}^{-1}L = L_{A_{can},q_i}$, *we have* $t \in L_{A_{can},q_i}$. $c_{q_i}^{-1}L \subseteq L$, *so* q_i *is initial, so* $t \in L(A_{can})$.

Reciprocally, let $t \in L(A_{can})$. *There exists a* $q_i \in I$ *such that* $t \in L_{A_{can},q_i}$. $c_{q_i}^{-1}L = L_{q_i}$, *so* $t \in c_{q_i}^{-1}L$. *As* q_i *is initial,* $c_{q_i}^{-1}L$ *is a subset of* L. *So* $t \in L$. *So* $L = L(A_{can})$.

So A_{can} *recognizes* L. *For any* q, $L_q = c_q^{-1}L$, *so* A_{can} *is a RFTA. For any prime residual of* L, *there exists a state in the RFTA which recognizes it. As there are one state per prime residual in* A_{can}, A_{can} *is minimal in number of states.*

□ ◄

Annexe C

Identification à la limite de langages réguliers d'arbres à résiduels premiers disjoints

Auteurs : J. Carme, A. Lemay, A. Terlutte

Résumé : Nous définissons deux classes de langages d'arbres. Ces deux classes contiennent strictement la classe des langages d'arbres réversibles. Nous montrons que les deux classes ainsi définies sont identifiables à la limite par exemples positifs seuls.

C.1 Introduction

Nous nous intéressons ici à l'identification à la limite de langages par exemples positifs (modèle de Gold [Gold, 1967]). Cette tâche semble très ardue puisque des classes de langages aussi simples que la classe des langages réguliers n'est pas apprenable dans ce cadre. Par contre certaines sous-classes non triviales le sont. La classe des langages réversibles en particulier l'est [Angluin, 1982]. Si on aborde la théorie des langages sous l'angle de la théorie de Chomsky, la classe des langages algébriques est la classe qui vient après la classe des langages réguliers. S'il est difficile d'inférer des sous-classes de langages réguliers, il l'est encore plus d'inférer des sous-classes de langages algébriques. La tâche devient plus simple si on s'intéresse à l'inférence de ces langages à partir, non pas de mots, mais de mots structurés (les arbres de dérivations de ces grammaires par exemple). Cette problématique peut être intéressante en inférence de données semi-structurées (par exemple, données sous forme HTML ou XML) ou en linguistique (où on peut supposer pouvoir disposer de la structure soit par la prosodie, soit par la sémantique [Dudau-Sofronie et al., 2002]).

Il existe un certain nombre d'algorithmes permettant l'identification à la limite de langages d'arbres à partir d'exemples positifs [Sakakibara, 1992, Kanazawa, 1996, Besombes and Marion, 2002] et ce dans différents formalismes. En particulier, [Besombes and Marion, 2002] présente un algorithme d'inférence de langages d'arbres

dits réversibles qui semble factoriser ces différentes approches. Nous montrons dans la partie C.3 que cette classe de langages est l'extension directe de la classe des langages réversibles de mots [Angluin, 1982] : un langage régulier de mots est réversible si et seulement s'il peut être reconnu par un automate déterministe dont le miroir est déterministe, de même, un langage régulier d'arbres est réversible si et seulement s'il peut être reconnu par un automate déterministe ascendant dont le miroir est déterministe par contexte.

Dans [Denis et al., 2002b], nous avons introduit la classe des Automates Finis à États Résiduels de mots que nous avons étendu aux automates d'arbres ascendants et descendants dans [Carme et al., 2003a]. Dans [Denis et al., 2002a], nous avons montré que la classe des langages reconnaissables par un AFER dont le miroir est déterministe (et ayant certaines autres propriétés) est identifiable efficacement à la limite par exemples positifs. La classe ainsi définie a comme avantage de contenir strictement la classe des langages réversibles et d'introduire du non déterministe.

Nous suivons ici une approche parallèle : de la même manière que nous avons étendu la classe des langages réversibles de mots, nous étendons la classe des langages réversibles d'arbres. Nous obtenons alors deux classes de langages identifiables à la limite par exemples positifs : la classe des langages reconnaissables par des automates déterministes ascendants dont le miroir est un AFAR descendant (ayant certaines propriétés) et celle reconnue par des AFAR ascendants dont le miroir est déterministe par contexte (ayant certaines propriétés). Ces deux classes de langages sont identifiables efficacement à la limite par exemples positifs et contiennent strictement la classe des langages réversibles d'arbres. Par ailleurs, la seconde de ces classes a une certaine pertinence en traitement automatique du langage puisqu'elle permet de tenir compte de certains problèmes d'ambiguïté du langage qui n'étaient pas pris en compte par les algorithmes classiques (voir l'exemple de la section C.5.5).

C.2 Préliminaires

Nous considérons que le lecteur possède déjà des connaissances de base sur les automates d'arbres. Nous utilisons les notations définies dans TATA [Comon et al., 1997].

Un alphabet gradué est un couple $(\mathcal{F}, \mathit{Arité})$ où $\mathcal{F}$ est un ensemble fini et $\mathit{Arité}$ une fonction de $\mathcal{F}$ dans $\mathbb{N}$. L'ensemble des symboles d'arité p se note $\mathcal{F}_p$. Les éléments de $\mathcal{F}_0$ sont appelés constantes. L'ensemble des termes sur $\mathcal{F}$ se note $\mathcal{T}(\mathcal{F})$. La hauteur d'un terme t se définit ainsi : elle vaut 0 si t est une constante, et $n+1$ si $t = f(t_1, \ldots, t_n)$, où n est la hauteur du plus haut t_i. L'ensemble des termes de hauteur n se note $\mathcal{T}_n(\mathcal{F})$, et l'ensemble des termes de hauteur inférieure ou égale à n se note $\mathcal{T}_{\leq n}(\mathcal{F})$. Soit $\diamond$ une constante particulière, qui n'est pas dans $\mathcal{F}$. L'ensemble des *contextes*, noté $\mathcal{C}(\mathcal{F})$, représente l'ensemble des termes qui contiennent exactement une occurence de $\diamond$. Chaque contexte est noté $c[\diamond]$, ou c en l'absence d'ambiguïté. On note $c[t]$ le terme obtenu à partir de $c[\diamond]$ en remplaçant $\diamond$ par t. La hauteur h_t d'un contexte se définit comme la hauteur d'un terme. On utilise également pour les contextes une autre définition de la hauteur, notée $h_\diamond$. Elle correspond à la distance, en nombre de noeuds intermédiaires, entre la racine et le $\diamond$.

Un *Automate Fini d'Arbres ascendant* ($AFA_\uparrow$) sur $\mathcal{F}$ est un quadruplet $A = \langle Q, \mathcal{F}, Q_f, \Delta \rangle$ où Q est un ensemble fini d'états, $Q_f \subseteq Q$ est un ensemble d'états finaux, et Δ un ensemble de règles de transition de la forme $f(q_1, \ldots, q_n) \rightarrow q$ ou $a \rightarrow q$. La *relation de transition*

s'écrit $\rightarrow_A$ et sa cloture transitive et symétrique s'écrit $\rightarrow_A^*$. Les termes reconnus par A sont les termes t tels qu'il existe un état final q_f tel que $t \rightarrow_A^* q_f$. Le langage $L(A)$ reconnu par A est l'ensemble des termes reconnus par A. Un état q *accepte* un contexte c s'il existe un état final q_f tel que $c[q] \rightarrow_A^* q_f$. Un terme t *atteint* un état q si $t \rightarrow_A^* q$.

Un $AFA_\uparrow$ est *émondé* si pour tout état q, il existe un terme t, un contexte c et un état final q_f tels que $c[t] \rightarrow_A^* c[q] \rightarrow_A^* q_f$. Dans cet article, nous considérerons tous les automates comme émondés. Un $AFA_\uparrow$ est *déterministe* ($AFAD_\uparrow$) si et seulement si toutes ses règles ont des membres gauches différents. Un langage d'arbres est *régulier* si et seulement s'il est reconnu par un $AFA_\uparrow$. Tout langage régulier d'arbres est également reconnu par un $AFAD_\uparrow$.

Soit L un langage régulier d'arbres sur un alphabet $\mathcal{F}$, et soit t un terme. Le *langage résiduel ascendant* de L relatif à t, noté $t^{-1}L$, est l'ensemble des contextes c tels que $c[t] \in L$: $t^{-1}L = \{c \in \mathcal{C}(\mathcal{F}) \mid c[t] \in L\}$.

Remarquons qu'un langage résiduel ascendant est un ensemble de contextes, et non un ensemble d'arbres. D'autre part, sa définition dépend uniquement du langage et non d'un quelconque automate d'arbres reconnaissant ce langage. Un langage résiduel ascendant est *composé* s'il est l'union d'autres langages résiduels, *premier* sinon. Le *terme caractéristique* d'un langage résiduel ascendant $t^{-1}L$ est le plus petit terme t' tel que $t'^{-1}L = t^{-1}L$; un exemple d'ordre pour définir le plus petit terme est donné en définition C.2. $R_\uparrow(L)$ est l'ensemble des termes caractéristiques des langages résiduels ascendants de L. $P_\uparrow(L)$ est l'ensemble des termes caractéristiques des langages résiduels ascendants premiers de L.

Un *Automate Fini d'Arbres à Résiduels ascendant* ($AFAR_\uparrow$) est un $AFA_\uparrow$ dans lequel l'ensemble des contextes acceptés par chaque état est exactement un langage résiduel ascendant. Un $AFAR_\uparrow$ n'est pas nécessairement déterministe mais accepte une forme canonique [Carme et al., 2003a]. L'$AFAR_\uparrow$ canonique $A_{can\uparrow}$ d'un langage d'arbres L est un automate $A_{can\uparrow} = \langle Q, \mathcal{F}, Q_f, \Delta\rangle$ tel que $Q = P_\uparrow(L)$, $Q_f = \{t^{-1}L \mid t \in P_\uparrow(L),\ \diamond \in t^{-1}L\}$ et $\Delta = \{a \rightarrow t \mid t^{-1}L \subseteq a^{-1}L\} \cup \{f(t_1, \ldots, t_n) \rightarrow t \mid \forall i,\ t_i \in P_\uparrow(L),\ t \in P_\uparrow(L),\ t^{-1}L \subseteq f(t_1, \ldots, t_n)^{-1}L\}$. Il est unique, minimal en nombre d'états et maximal en nombre de transitions.

Un *Automate Fini d'Arbres descendant* ($AFA_\downarrow$) sur $\mathcal{F}$ est un quadruplet $A = \langle Q, \mathcal{F}, Q_i, \Delta\rangle$ où Q est un ensemble fini d'états, $Q_i \subseteq Q$ est un ensemble d'états initiaux, et Δ un ensemble de règles de transition de la forme $q(f) \rightarrow f(q_1, \ldots, q_n)$ ou $q(a) \rightarrow a$. La *relation de transition* s'écrit $\rightarrow_A$ et sa cloture transitive et symétrique s'écrit $\rightarrow_A^*$. Les termes reconnus par A sont les termes t tels qu'il existe un état initial q_i tel que $q_i(t) \rightarrow_A^* t$. Le langage $L(A)$ reconnu par A est l'ensemble des termes reconnus par A. Un état q *accepte* un terme t si $q(t) \rightarrow_A^* t$. Un contexte c *atteint* un état q s'il existe un état initial q_i tel que $q_i(c) \rightarrow_A^* c[q]$. Un $AFA_\downarrow$ est *émondé* si pour tout état q, il existe un terme t, un contexte c et un état initial q_i tels que $q_i(c[t]) \rightarrow_A^* c[q(t)] \rightarrow_A^* c[t]$. Dans cet article, nous considérerons tous les automates comme émondés. Un $AFA_\downarrow$ est *déterministe* ($AFAD_\downarrow$) si et seulement si toutes ses règles ont des membres gauches différents. Un $AFA_\downarrow$ est *déterministe par contexte* si pour tout contexte c, l'ensemble Q_c des états atteints par ce contexte est soit vide, soit un singleton [Nivat and Podelski, 1997]. Tout automate déterministe est déterministe par contexte. Dans un automate déterministe par contexte, pour tout couple de règles $q(f) \rightarrow f(q_1, \ldots, q_n)$ et $q(f) \rightarrow f(q'_1, \ldots, q'_n)$, s'il existe un indice i tel que $q_i \neq q'_i$, alors il existe un indice $j \neq i$ tel que $L_{q_j} \cap L_{q'_j} = \emptyset$ [Nivat and Podelski, 1997].

Soit L un langage régulier d'arbres sur un alphabet $\mathcal{F}$, et soit c un contexte. Le *langage résiduel descendant* de L relatif à c, noté $c^{-1}L$, est l'ensemble des termes t tels que $c[t] \in L$, c'est à dire $c^{-1}L = \{t \in \mathcal{T}(\mathcal{F}) \mid c[t] \in L\}$. Un langage résiduel descendant est *composé* s'il est l'union d'autres langages résiduels, *premier* sinon. Le *contexte caractéristique* d'un langage résiduel descendant $c^{-1}L$ est le plus petit contexte c' tel que $c'^{-1}L = c^{-1}L$; un exemple d'ordre pour définir le plus petit contexte est donné en définition C.1. $R_\downarrow(L)$ est l'ensemble des contextes caractéristiques des langages résiduels descendants de L. $P_\downarrow(L)$ est l'ensemble des contextes caractéristiques des langages résiduels descendants premiers de L.

Un *Automate Fini d'Arbres à Résiduels descendant* ($AFAR_\downarrow$) est un $AFA_\downarrow$ dans lequel l'ensemble des termes acceptés par chaque état est exactement un langage résiduel descendant. Tout automate déterministe par contexte est à résiduels. L'$AFAR_\downarrow$ canonique $A_{can\downarrow}$ d'un langage d'arbres L est un automate $A_{can\downarrow} = \langle Q, \mathcal{F}, Q_i, \Delta\rangle$ tel que $Q = P_\downarrow(L)$, $Q_i = \{c \mid c \in P_\downarrow(L),\ c^{-1}L \subseteq L\}$ et $\Delta = \{c(a) \to a \mid c \in P_\downarrow(L),\ a \in c^{-1}L\} \cup \{c(f) \to f(c_1, \ldots, c_n) \mid \forall i,\ \forall t_i \in c_i^{-1}L,\ f(t_1, \ldots, t_n) \in c^{-1}L\}$. Il est unique, minimal en nombre d'états et maximal en nombre de transitions.

Contrairement au cas ascendant, les automates déterministes, déterministes par contexte, à residuels et non-déterministes reconnaissent des classes de langages strictement incluses les unes dans les autres, dans cet ordre [Carme et al., 2003a].

Le *miroir* d'un $AFA_\uparrow$ $A = \langle Q, \mathcal{F}, Q_f, \Delta\rangle$ est un $AFA_\downarrow$ $A' = \langle Q, \mathcal{F}, Q_i, \Delta'\rangle$ tel que $Q_i = Q_f$ et Δ' est constitué des règles de Δ inversées. Les automates A et A' reconnaissent le même langage.

Un *automate réversible* [Sakakibara, 1992, Besombes and Marion, 2002] est un $AFAD_\uparrow$ ayant la propriété suivante : pour tout couple de règles différentes $f(q_1, \ldots, q_n) \to q$ et $f(q'_1, \ldots, q'_n) \to q$, il existe au moins deux indices i et j tels que $q_i \neq q'_i$ et $q_j \neq q'_j$.

Le contexte $c_1[\diamond]$ est *préfixe descendant* du contexte $c_2[\diamond]$ s'il existe un contexte $c[\diamond]$ tel que $c_2[\diamond] = c_1[c[\diamond]]$. Le contexte $c[\diamond]$ est *préfixe descendant* du terme t_1 s'il existe un terme t_2 tel que $t_1 = c[t_2]$. On note $Pref_\downarrow(\alpha)$ l'ensemble des préfixes descendants d'un terme ou d'un contexte α, et $Pref_\downarrow(S)$ l'ensemble des préfixes descendants des éléments d'un ensemble fini de termes ou de contextes S. Le terme t_1 est *préfixe ascendant* du terme t_2 s'il existe un contexte c tel que $t_2 = c[t_1]$. On note $Pref_\uparrow(t)$ l'ensemble des préfixes ascendants d'un arbre t, et $Pref_\uparrow(S)$ l'ensemble des préfixes ascendants des termes de l'ensemble S. L'*automate préfixe ascendant* d'un ensemble fini d'arbres S, noté $AP_\uparrow(S)$ est le plus grand automate d'arbres déterministe émondé ascendant qui reconnait exactement l'ensemble S.

Il est possible de définir une relation d'ordre pour les contextes ou pour les termes de différentes façons. Nous indiquons ici les contraintes que doit vérifier une relation d'ordre cohérente avec nos algorithmes d'identification. Il faut simplement que le préfixe d'un contexte soit plus petit, dans cet ordre, que le contexte lui-même. Il faut aussi qu'on puisse énumérer les contextes compris entre deux contextes donnés sans risquer d'explorer des branches infinies. Ces ordres nous serviront pour définir les échantillons caractéristiques et la notion de représentant caractéristique, en prenant le plus petit élément parmi un ensemble d'éléments équivalents. Plus important, dans l'algorithme d'apprentissage $A_\downarrow(S, P)$, les contextes seront traités séquentiellement selon cet ordre qui nous évite d'explorer les contextes en profondeur.

Définition C.1. *(définition partielle)*

Soient c_1 et c_2 deux contextes. On dira que $c_1 \leq c_2$ si l'une de ces trois conditons est remplie :

– $h_\diamond(c_1) + h_t(c_1) < h_\diamond(c_2) + h_t(c_2)$

– $h_\diamond(c_1) + h_t(c_1) = h_\diamond(c_2) + h_t(c_2)$ *et* $h_\diamond(c_1) < h_\diamond(c_2)$

– $h_\diamond(c_1) = h_\diamond(c_2)$, $h_t(c_1) = h_t(c_2)$ *et, dans ce cas, les contextes étant en nombre fini, on fixe des conditions pour ordonner deux contextes de même hauteur totale et de même hauteur de la racine au $\diamond$; par exemple, on pourrait établir une bijection entre les contextes d'une hauteur totale donnée et un ensemble de valeurs entières.*

Lemme C.1. *Soient c_1 et c_2 deux contextes, si c_1 est préfixe descendant de c_2 alors $c_1 < c_2$. Si $c_1, c_2 \in P_\downarrow(L)$ alors le nombre de contextes entre c_1 et c_2 est fini.*

De la même façon, on définit un ordre sur les termes :

Définition C.2. *(définition partielle)*

Soient t_1 et t_2 deux termes de $\mathcal{T}(\mathcal{F})$, on dira que $t_1 < t_2$ si $h_t(t_1) < h_t(t_2)$. Si $h_t(t_1) = h_t(t_2)$, on fixe des conditions pour ordonner des termes de même hauteur.

Dans cet article, nous traiterons de l'apprentissage des langages dans le modèle d'identification à la limite de Gold (voir [Gold, 1967] et [Angluin, 1980]). Nous reprenons ici quelques définitions classiques.

Soit S une présentation positive d'un langage, c'est à dire une présentation telle que chaque élément du langage est assuré de figurer après un temps fini. On note S_t les t premiers éléments de S. Soit M un algorithme qui prend en entrée un ensemble fini de mots et qui donne en sortie une représentation d'un langage (un automate par exemple). On dit que M identifie à la limite une classe de langages $\mathcal{L}$ par exemples positifs si pour tout langage L de $\mathcal{L}$ et pour toute présentation positive de L, il existe un indice T à partir duquel $M(S_t)$ est toujours une représentation de L. On dit qu'une classe est identifiable à la limite par exemples positifs s'il existe un algorithme qui l'identifie à la limite.

C.3 Langages réguliers d'arbres réversibles

La définition de [Besombes and Marion, 2002] qui reprend celle de [Sakakibara, 1992] définit les langages réversibles comme des langages d'arbres reconnus par des automates ascendants déterministes possédant une propriété supplémentaire (appelée *reset-free* chez [Sakakibara, 1992]). Dans les mots, les langages réversibles ont été définis comme étant reconnus par des automates déterministes dont le miroir est lui-même déterministe. Le résultat suivant met en évidence l'analogie entre langages réversibles dans les mots et dans les arbres, en utilisant des automates déterministes ascendants et descendants.

Théorème C.1. *Tout automate d'arbres réversible est un automate d'arbres déterministe ascendant dont le miroir est déterministe par contexte, et inversement, tout automate d'arbres déterministe ascendant dont le miroir est déterministe par contexte est un automate d'arbres réversible.*

Les langages réversibles sont reconnus par des automates dont le miroir est déterministe par contexte. Une autre façon de présenter cette propriété consiste à dire que les langages résiduels associés aux états sont disjoints.

Proposition C.1. *Les langages réversibles sont des langages réguliers d'arbres à résiduels (ascendants ou descendants) disjoints.*

Que ce soit relativement aux contextes ou relativement aux termes, les langages réversibles ne contiennent pas de résiduels composés. Tous les résiduels sont premiers et disjoints deux à deux. Ces résultats permettent de faire le lien entre [Angluin, 1982] et [Besombes and Marion, 2002]. De la même manière que nous avons étendu le résultat d'[Angluin, 1982] dans [Carme et al., 2003a], nous pouvons étendre le résultat de [Besombes and Marion, 2002].

Nous nous intéresserons aux langages ayant des langages résiduels *premiers* disjoints. Une intersection non vide fournit alors des informations sur l'inclusion des résiduels. Le lemme suivant s'applique aussi bien aux résiduels descendants qu'aux résiduels ascendants.

Lemme C.2. *Si L est à résiduels premiers disjoints, si p est un résiduel premier de L et r un résiduel quelconque de L, alors $p \cap r \neq \emptyset \Rightarrow p \subseteq r$.*

C.4 Langages à résiduels descendants premiers disjoints

C.4.1 Classe de langages

Tout comme dans le cas des langages de mots, la classe des langages à résiduels descendants premiers disjoints n'est pas apprenable. Elle le devient si on lui ajoute certaines restrictions.

Définition C.3. *Un langage d'arbres est à résiduels descendants premiers préfixiels si et seulement si, pour tous contextes c et c', si $c[c']^{-1}L$ est premier, alors $c^{-1}L$ l'est également.*

Définition C.4. *Un langage d'arbres est à résiduels descendants composés prévisibles si et seulement si, pour tout contexte c tel que $c^{-1}L$ est composé, alors il existe un contexte $c_p < c$ tel que ${c_p}^{-1}L$ soit premier et ${c_p}^{-1}L \subset c^{-1}L$.*

Nous nous intéressons ici à la classe des langages d'arbres à résiduels descendants premiers disjoints et préfixiels et à résiduels descendants composés prévisibles. Nous noterons cette classe $\mathcal{L}_{\downarrow}$.

Exemple C.1. *Le langage $\{f(a,b), f(a,c), f(d,c), f(e,d), f(b,e)\}$ appartient à $\mathcal{L}_{\downarrow}$ sans appartenir à la classe des langages réversibles.*

Proposition C.2. *Un langage d'arbres est à résiduels descendants premiers disjoints si et seulement s'il est reconnu par un $AFAR_{\downarrow}$ dont le miroir est déterministe.*

C.4.2 Échantillon caractéristique

Soit L un langage de $\mathcal{L}_\downarrow$. On note $K_\downarrow(L)$[5] l'ensemble de termes suivants : $K_\downarrow(L) = P_\downarrow(L) \cup \{c \mid c = c'[c],\ h_\diamond(c) = 1,\ c' \in P_\downarrow(L),\ c < Max(P_\downarrow(L))\}$. L'échantillon caractéristique de L, noté $S_{\downarrow car}(L)$ est le plus petit échantillon tel que :

- pour tout contexte c de $K_\downarrow(L)$, soit t le plus petit terme tel que $c[t] \in L$, $c[t] \in S_{\downarrow car}(L)$,
- pour tout contexte $c \in P_\downarrow(L)$ et pour toute constante $a \in \mathcal{F}_0$, si $c[a] \in L$, $c[a] \in S_{\downarrow car}(L)$,
- pour tous contextes $c \in P_\downarrow(L)$ et $c' \in K_\downarrow(L)$, si $c^{-1}L \cap c'^{-1}L \neq \emptyset$, alors, soit t le plus petit terme de $c^{-1}L \cap c'^{-1}L$, $c[t] \in S_{\downarrow car}(L)$ et $c'[t] \in S_{\downarrow car}(L)$.
- pour tout contexte $c \in P_\downarrow(L)$ et pour tout ensemble de contextes $c_1, \ldots, c_n$ de $K_\downarrow(L)$ et pour tout $f \in \mathcal{F}_n$, s'il existe un terme $t = f(t_1, \ldots, t_n)$ tel que $c[t] \in L$ et pour tout i, $c_i[t_i] \in L$, soit t le plus petit terme permettant ce résultat, $c[t] \in S_{\downarrow car}(L)$ et $c_i[t_i] \in S_{\downarrow car}(L)$,

On peut observer que, du fait du lemme C.1, l'échantillon $S_{\downarrow car}(L)$ est fini. De la définition de $S_{\downarrow car}(L)$, on obtient directement le lemme suivant :

Lemme C.3. *Soit L un langage de $\mathcal{L}_\downarrow$. Soit S un échantillon de termes tel que $S_{\downarrow car}(L) \subset S$. Soient deux contextes $c \in P_\downarrow(L)$ et $c' \in K_\downarrow(L)$. On a $(c^{-1}L \cap c'^{-1}L = \emptyset) \Leftrightarrow (c^{-1}S \cap c'^{-1}S' = \emptyset)$.*

C.4.3 Apprentissage

On note tout d'abord la procédure suivante qui permet de construire l'$AFAR_\downarrow$ canonique d'un langage de $\mathcal{L}_\downarrow$ à partir d'un ensemble de termes et d'un ensemble de contextes.

$A_\downarrow(S, P)$

Entrée : S et P
$\mathcal{F}$ = l'alphabet de S
$Q = P$
$Q_i = \{c_i \in P \mid c_i^{-1}S \subseteq S\} = \{\diamond\}$
Δ contient toutes les transitions de la forme $c(f) \rightarrow f(c_1, \ldots, c_n)$ telles que :
$c \in P$, $f \in \mathcal{F}_n$, pour tout i, $c_i \in P$ et
il existe un terme $t = f(t_1, \ldots, t_n)$ tel que $c[t] \in S$ et, $\forall i$, $c_i[t_i] \in S$
et les transitions de la forme $c(a) \rightarrow a$ si $c[a] \in S$
Sortie : $A = \langle Q, \mathcal{F}, Q_i, \Delta \rangle$

Lemme C.4. *Soit L un langage de $\mathcal{L}_\downarrow$. Pour tout échantillon S de L contenant $S_{\downarrow car}(L)$, l'automate $A_\downarrow(S, P_\downarrow(L))$ est l'$AFAR_\downarrow$ canonique de L.*

Preuve :

On remarque tout d'abord que, soient c et c' deux contextes tels que $c^{-1}L$ soit premier, $c^{-1}L \subseteq c'^{-1}L \Leftrightarrow c^{-1}L \cap c'^{-1}L \neq \emptyset$ car les résiduels premiers de L sont disjoints (lemme C.2). Comme les résiduels sont disjoints, on a aussi $(\forall t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ tels que $c[f(t_1, \ldots, t_n)] \in L$

[5] K pour Kernel

et $\forall i$, $c_i[t_i] \in L) \Leftrightarrow (\exists t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ *tels que* $c[f(t_1, \ldots, t_n)] \in L$ *et* $\forall i$, $c_i[t_i] \in L)$. *Par ailleurs, le choix de* S *fait que :*

- L *étant à résiduels descendants premiers préfixiels,* $\diamond \in P(L)$. *Comme* $\diamond^{-1}L = L$ *et que* L *est un langage à résiduels descendants premiers disjoints,* $\diamond$ *est le seul contexte* c *tel que* $c^{-1}L \subset L$,
- *pour tout* $c \in P_{\downarrow}(L)$ *et pour tout* $a \in \mathcal{F}_0$, $c[a] \in L \Leftrightarrow c[a] \in S$ *et*
- *soit* $n \in \mathbb{N}$, *soit* $f \in \mathcal{F}_n$, *soit* $c \in P_{\downarrow}(L)$ *et soient* $c_1, \ldots, c_n \in P_{\downarrow}(L)$, *alors* $(\exists t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ *tels que* $c[f(t_1, \ldots, t_n)] \in L$ *et* $\forall i$, $c_i[t_i] \in L) \Leftrightarrow (\exists t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ *tels que* $c[f(t_1, \ldots, t_n)] \in S$ *et* $\forall i$, $c_i[t_i] \in S)$.

L'algorithme construit donc bien l'$AFAR_{\downarrow}$ *canonique de* L. ◄

Lemme C.5. *Soit* L *un langage de* $\mathcal{L}_{\downarrow}$. *Soit* S *un échantillon de* L *contenant* $S_{\downarrow car}(L)$ *et* P *un ensemble de termes strictement contenu dans* $P_{\downarrow}(L)$, $A_{\downarrow}(S, P)$ *est un automate qui n'est pas consistant avec* S.

Théorème C.2. *La classe* $\mathcal{L}_{\downarrow}$ *est identifiable à la limite par exemples positifs.*

L'algorithme qui permet ce résultat est le suivant :

Apprend$_{\downarrow}$

Entrée : S
$i = 0$
$Pref_{\downarrow}(S) = \{c_i[\diamond] \mid \exists t_i \in \mathcal{T}(\mathcal{F})$ tel que $c_i[t_i] \in S\}$
// $Pref_{\downarrow}(S)$ est ordonné selon l'ordre de la définition C.1
$P_0 = \{\diamond\}$
Faire
$i \leftarrow i + 1$
Si $\forall c \in Pref_{\downarrow}(c_i) \setminus \{c_i\}$, $c \in P_{i-1}$ et si $\forall c \in P_{i-1}$, $c^{-1}S \cap c_i^{-1}S = \emptyset$ **Alors**
$P_i = P_{i-1} \cup \{c_i\}$
Sinon
$P_i = P_{i-1}$
Fin Si
Tant que $A_{\downarrow}(S, P_i)$ n'est pas consistant avec S et $i < |Pref_{\downarrow}(S)|$
Sortie : $A_{\downarrow}(S, P_i)$ si cet automate est consistant avec S, sinon $AP_{\uparrow}(S)$.

Le théorème C.2 est prouvé par le lemme suivant.

Lemme C.6. *Soit* L *un langage de* $\mathcal{L}_{\downarrow}$. *Soit* S *un échantillon de termes tel que* $S_{\downarrow car}(L) \subset S$ *et* $S \subset L$. *Si l'algorithme Apprend*$_{\downarrow}$ *prend en entrée* S, *il donne en sortie l'*$AFAR_{\downarrow}$ *canonique de* L.

Preuve : Nous montrons tout d'abord qu'à chaque étape i, $P_i = P_{\downarrow}(L) \cap \{c_{j \leq i}\}$. *À l'étape* 0, $P_0 = \{\diamond\}$, L *étant à premiers préfixiels,* $\diamond \in P_{\downarrow}(L)$. *Si à l'étape* $i-1$ *l'hypothèse est vérifiée, montrons qu'elle l'est également à l'étape* i.

Montrons tout d'abord que $P_i \subseteq P_{\downarrow}(L) \cap \{c_{j \leq i}\}$. *Deux cas se présentent :*

– *si* $P_i = P_{i-1}$ *alors* $P_i = P_{i-1} = P_\downarrow(L) \cap \{c_{j\leq i-1}\} \subseteq P_\downarrow(L) \cap \{c_{j\leq i}\}$,
– *si* $P_i \neq P_{i-1}$, *c'est à dire* $c_i \in P_i$, *supposons que* c_i *ne soit pas dans* $P_\downarrow(L)$.
Soit ${c_i}^{-1}L$ *est premier, soit il est composé.*
S'il est premier, alors il existe $c \in P_\downarrow(L)$ *tel que* $c < c_i$ *et* $c^{-1}L = {c_i}^{-1}L$. *Par hypothèse,* $c \in P_{i-1}$. *Par construction de* P_i, *on a* $c_i \notin P_i$. *Contradiction.*
Si ${c_i}^{-1}L$ *est composé, alors, soit il existe un contexte* $c \in Pref_\downarrow(c_i)$ *tel que* $c \notin P_\downarrow(L)$, *dans ce cas,* $c_i \notin P_i$. *S'il n'existe pas de tel contexte, alors par définition de* $K_\downarrow(L)$, $c_i \in K_\downarrow(L)$. *Les langages résiduels composés étant prévisibles, il existe un contexte* $c_p \in P_\downarrow(L)$ *tel que* $c_p < c_i$ *et* ${c_p}^{-1}L \subset {c_i}^{-1}L$. *Par hypothèse,* $c_p \in P_{i-1}$ *et par construction de* P_i, $c_i \notin P_i$. *Contradiction.*
Par conséquent, si $c_i \in P_i$, *on a* $c_i \in P_\downarrow(L)$.

Par conséquent, $P_i \subseteq P_\downarrow(L) \cap \{c_{j\leq i}\}$.

Montrons maintenant que $P_\downarrow(L) \cap \{c_{j\leq i}\} \subseteq P_i$. *Deux cas se présentent, soit* $c_i \in P_\downarrow(L)$, *soit* $c_i \notin P_\downarrow(L)$. *Si* $c_i \notin P_\downarrow(L)$, $P_\downarrow(L) \cap \{c_{j\leq i}\} = P_\downarrow(L) \cap \{c_{j\leq i-1}\}$. *Comme* $P_\downarrow(L) \cap \{c_{j\leq i-1}\} = P_{i-1}$ *par hypothèse de récurrence et* $P_{i-1} \subseteq P_i$, *on a* $P_\downarrow(L) \cap \{c_{j\leq i}\} \subseteq P_i$. *L'autre possibilité est que* $c_i \in P_\downarrow(L)$. *Dans ce cas, les premiers étant préfixiels, tous les éléments de* $Pref_\downarrow(c_i)$ *sont dans* $P_\downarrow(L)$, *comme ceux-ci sont plus petits que* c_i, *ils sont dans* P_{i-1}. *Par ailleurs, les résiduels étant disjoints, il n'existe aucun élément* $c \in P_i = P_{i-1} \cup \{c_i\}$ *(donc aucun de* P_{i-1}*) tel que* $c \neq c_i$ *et* $c^{-1}L \cap {c_i}^{-1}L \neq \emptyset$. *Par conséquent* $c_i \in P_i$. *On a donc* $P_i \supseteq P_\downarrow(L) \cap \{c_{j\leq i}\}$.

L'hypothèse est donc vérifiée, on a donc $P_i = P_\downarrow(L) \cap \{c_{j\leq i}\}$.

Comme en plus $P_\downarrow(L) \subset Pref_\downarrow(S_{\downarrow car}(L))$ *et* $S_{\downarrow car}(L) \subset S$, *les lemmes C.4 et C.5 permettent de montrer que quand l'algorithme s'arrête, il fournit l'*$AFAR_\downarrow$ *canonique de* L. ◄

C.4.4 Exemple

Soit L le langage reconnu par l'automate $A = \langle Q, \mathcal{F}, Q_i, \Delta\rangle$ avec $\mathcal{F} = \{g(,), f(,), h(), a, b\}$, $Q_i = \{q_0\}$ et $\Delta = \{q_0(g) \rightarrow g(q_1, q_2), q_1(f) \rightarrow f(q_1, q_2), q_1(f) \rightarrow f(q_1, q_3), q_1(f) \rightarrow f(q_4, q_2), q_1(f) \rightarrow f(q_4, q_3), q_4(h) \rightarrow h(q_3), q_2(b) \rightarrow b, q_3(a) \rightarrow a\}$. L'automate A n'est pas un $AFAR_\downarrow$, n'est pas déterministe par contexte... Mais il reconnaît un langage appartenant à $\mathcal{L}_\downarrow$ dont les premiers arbres sont $\{g(h(a), b)$, $g(f(h(a), a), b)$, $g(f(h(a), b), b)$, $g(f(f(h(a), a), a), b)$, $g(f(f(h(a), b), a), b)$, $g(f(f(\ h(a), a), b), b)$, $g(f(f(h(a), b), b), b), ...\}$.

Ce langage possède quatre langages résiduels descendants premiers qui sont $\diamond^{-1}L = L, g(\diamond, b)^{-1}L, g(h(a), \diamond)^{-1}L = \{b\}, g(h(\diamond), b)^{-1}L = \{a\}$ et un résiduel composé $g(f(h(a), \diamond), b)^{-1}L = \{a, b\}$.

On peut vérifier que $K_\downarrow(L) = P_\downarrow(L) = \{\diamond, g(\diamond, b), g(h(a), \diamond), g(h(\diamond), b)\}$. Comme cet ensemble contient exactement les premiers préfixes du langage, ceci signifie que l'algorithme ne devrait avoir à examiner que ces contextes pour retrouver l'automate.

L'échantillon caractéristique est très petit car la plupart des contraintes demandées dans l'échantillon caractéristique aboutissent à des arbres identiques. Par exemple, si on doit poursuivre les contextes $\diamond$, $g(\diamond, b)$, $g(h(a), \diamond)$, et $g(h(\diamond), b)$, cela se fait avec le même arbre $g(h(a), b)$. On aboutit donc à l'échantillon caractéristique constitué des trois premiers arbres du langage, à savoir $S_{\downarrow car}(L) = \{g(h(a), b), g(f(h(a), a), b), g(\ f(h(a), b), b)\}$.

En exécutant l'algorithme, on va retrouver $P_3 = K_\downarrow(L) = P_\downarrow(L)$ qui sera suffisant pour reconstruire un automate consistant qui, contrairement à celui qui nous a permis de définir L, sera un $AFAR_\downarrow$.

Pour terminer, donnons un exemple de reconstruction de règles de l'algorithme $A_\downarrow$ (S, P). Les contextes correspondront aux états ; associons q_1 à $c_1 = g(\diamond, b)$ et q_3 à $c_3 = g(h(\diamond), b)$. Nous aurons une règle $q_1(f) \to f(q_1, q_3)$ car $g(f(h(a), a), b)$ appartient à $S_{\downarrow car}(L)$. Cet arbre peut se mettre sous la forme $c_1[f(t_1, t_3)]$ avec $c_1[t_1]$ et $c_3[t_3]$ dans $S_{\downarrow car}(L)$.

L'$AFAR_\downarrow$ reconstruit par l'algorithme possèdera les règles $\{q_0(g) \to g(q_1, q_2)$, $q_1(f) \to f(q_1, q_2)$, $q_1(f) \to f(q_1, q_3)$, $q_1(h) \to h(q_3)$, $q_2(b) \to b$, $q_3(a) \to a\}$.

C.5 Langages à résiduels ascendants premiers disjoints

C.5.1 Graphe de résiduels

Soit L un langage d'arbres. Soit T un ensemble de termes tel que $T \subset Pref_\uparrow(L)$. On appellera graphe de résiduels de L sur T le graphe orienté $G_{T,L} = \langle T, \to_L \rangle$ où $\to_L$ (noté $\to$ pour plus de clarté) est une relation incluse dans $T \times T$ telle que $t \to t'$ si et seulement si $t^{-1}L \cap t'^{-1}L \neq \emptyset$ et si $\{t'' \mid t''^{-1}L \cap t'^{-1}L \neq \emptyset\} \subseteq \{t'' \mid t''^{-1}L \cap t^{-1}L \neq \emptyset\}$. C'est-à-dire que tout langage résiduel représenté dans le graphe ayant une intersection non nulle avec $t'^{-1}L$ a une intersection non nulle avec $t^{-1}L$. Nous montrerons par la suite que, sous certaines conditions, cette relation peut être considérée équivalente à la relation d'inclusion entre langages résiduels.

On dira que $G_{T,L}$ est un graphe de résiduels parfaitement prévisible si pour tout terme $t \in T$ et pour tout terme $t_p \in P_\uparrow(L)$ tel que $t_p^{-1}L \subseteq t^{-1}L$, on a $t_p \in T$. On a alors le théorème suivant :

Proposition C.3. *Soit L un langage régulier d'arbres à résiduels ascendants premiers disjoints. Soit $G_{T,L}$ un graphe de résiduels parfaitement prévisible. Alors, pour tous termes t et $t' \in T$, on a $t \to t' \Leftrightarrow t'^{-1}L \subseteq t^{-1}L$.*

Par ailleurs, la fonction suivante nous permet de retrouver les résiduels premiers présents dans le graphe.

Définition C.5. *Soit L un langage régulier d'arbres. Soit T un ensemble de termes. Soit $G_{T,L}$ un graphe de résiduels parfaitement prévisible. P est une fonction qui prend en entrée un graphe de résiduels et fournit en résultat $P(G_{T,L}) = \{t \in T \mid \forall t' \in T,\ t \to t' \Rightarrow t' \to t\}$.*

Le lemme suivant montre que quand le langage est à résiduels premiers disjoints et que le graphe est parfaitement prévisible, $P(G_{T,L})$ nous donne les termes correspondants à des résiduels premiers dans le graphe.

Lemme C.7. *Soit L un langage régulier d'arbres à résiduels ascendants premiers disjoints. Soit $G_{T,L} = \langle T, \to \rangle$ un graphe de résiduels de L parfaitement prévisible. Pour tout $t \in T$, on a $t \in P(G_{T,L}) \Leftrightarrow t^{-1}L$ premier.*

Preuve : Supposons qu'il existe $t \in P(G_{T,L})$ tel que $t^{-1}L$ soit composé. Comme $G_{T,L}$ est parfaitement prévisible, il existe un résiduel $t_p \in P_\uparrow(L)$ tel que $t_p \in T$, ${t_p}^{-1}L \subset t^{-1}L$ et

$t^{-1}L \not\subset {t_p}^{-1}L$. D'après la proposition C.3, on a donc $t \to t_p$ et $t_p \not\to t$. Donc, pour $t \in T$, on a $t^{-1}L$ composé $\Rightarrow t \notin P(G_{T,L})$ et par conséquent, $t \in P(G_{T,L}) \Rightarrow t^{-1}L$ premier.

Par ailleurs, supposons qu'il existe un terme $t_p \in T$ tel que $t_p^{-1}L$ soit premier et $t_p \notin P(G_{T,L})$. Il existe donc $t \in T$ tel que $t_p \to t$ et $t \not\to t_p$. D'après la proposition C.3, c'est donc que $t^{-1}L$ est inclus strictement dans $t_p^{-1}L$ ce qui est impossible car les résiduels premiers de L sont disjoints. Par conséquent, pour tout $t \in T$, $t^{-1}L$ premier $\Rightarrow t \in P(G_{T,L})$, ce qui termine la preuve. ◄

Notons que $P(G_{T,L})$ peut éventuellement contenir des termes correspondant à des résiduels équivalents. On utilisera la fonction $Carac$ suivante pour s'en débarrasser : $Carac$ est une fonction qui prend en entrée le résultat de la fonction P et fournit en résultat $Carac(P(G_{T,L})) = \{t \in P(G_{T,L}) \mid \nexists t' \in T, t \to t', t < t'\}$. On a alors par conséquence directe des deux lemmes précédents le lemme suivant :

Lemme C.8. *Soit L un langage régulier d'arbres à résiduels premiers disjoints. Soit $G_{T,L} = \langle T, \to \rangle$ un graphe de résiduels de L parfaitement prévisible. Si pour tout terme $t \in T$ tel que $t^{-1}L$ soit premier, il existe $t_p \in T$ tel que t_p soit le terme caractéristique de $t^{-1}L$, alors $Carac(P(G_{T,L})) = P_\uparrow(L) \cap T$.*

C.5.2 Classe de langages

Tout comme la classe des langages réguliers de mots à résiduels premiers disjoints, la classe des langages réguliers d'arbres à résiduels ascendants premiers disjoints n'est pas identifiable à la limite par exemples positifs. On doit lui apporter des restrictions.

Définition C.6. *Un langage d'arbres est à résiduels ascendants composés parfaitement prévisibles si pour tout résiduel composé r de terme caractéristique t et pour tout résiduel premier r' de terme caractéristique t' tel que $r' \subseteq r$, on a $h(t') \leq h(t)$.*

Définition C.7. *Un langage d'arbres est à résiduels ascendants premiers préfixiels si pour tout résiduel ascendant premier r de terme caractéristique t, tout résiduel de la forme ${t'}^{-1}L$ où t' est un sous-terme de t est premier.*

Nous nous intéresserons ici à la classe des langages d'arbres à résiduels ascendants premiers disjoints et préfixiels et à résiduels composés parfaitement prévisibles, notée ici $\mathcal{L}_\uparrow$.

Proposition C.4. *Un langage d'arbres est à résiduels ascendants premiers disjoints si et seulement s'il est reconnu par un $AFAR_\uparrow$ dont le miroir est déterministe par contexte.*

Remarquons que tout langage homogène, c'est à dire reconnu par un automate déterministe par contexte, est un langage régulier. Il est donc reconnu par un $AFAR_\uparrow$. Mais pour autant, tout langage homogène n'est pas à résiduels premiers disjoints. Pour appartenir à la famille qui nous intéresse, le langage doit être reconnu par un $AFAR_\uparrow$ dont le miroir est déterministe par contexte.

C.5.3 Échantillon caractéristique

Soit L un langage de $\mathcal{L}_{\uparrow}$. On définit le noyau de L, noté $K_{\uparrow}(L)$ de la manière suivante : $K_{\uparrow}(L) = P_{\uparrow}(L) \cup \{a \in \mathcal{F}_0\} \cup \{t \mid t = f(t_1, ..., t_n)\},\ f \in \mathcal{F}_n,\ \forall i, t_i \in P_{\uparrow}(L)\}$

On appellera échantillon caractéristique de L l'échantillon $S_{\uparrow car}(L)$ défini tel que $S_{\uparrow car}(L)$ soit le plus petit échantillon où :

- pour tout terme t de $K_{\uparrow}(L)$, soit c le plus petit contexte tel que $c[t] \in L$, $c[t] \in S_{\uparrow car}(L)$,
- pour tous termes t et t' de $K_{\uparrow}(L)$, si $t^{-1}L \cap t'^{-1}L \neq \emptyset$, alors, soit c le plus petit contexte tel que $c[t] \in (t^{-1}L \cap t'^{-1}L)$, $c[t] \in S_{\uparrow car}(L)$ et $c[t'] \in S_{\uparrow car}(L)$.

C.5.4 Apprentissage

On note tout d'abord qu'il est possible, comme dans le cas précédent, de reconstruire l'$AFAR_{\uparrow}$ canonique d'un langage à résiduels de termes premiers disjoints, connaissant $P_{\uparrow}(L)$ et un échantillon contenant $S_{car}(L)$. L'algorithme suivant est le pendant de $A_{\downarrow}(S, P)$:

$A_{\uparrow}(S, P)$

Entrée : S et P
$\mathcal{F}$ = l'alphabet de S
$Q = P$
$Q_f = \{t \in P \mid t \in S\}$
Δ contient toutes les transitions de la forme $f(t_1, \ldots, t_n) \to t$ telles que :
pour tout i, $t_i \in P$, $t \in P$, $f \in \mathcal{F}_n$
il existe un contexte c tel que $c[f(t_1, \ldots, t_n)] \in S$, $c[t] \in S$
Sortie : $A = \langle Q, \mathcal{F}, Q_f, \Delta \rangle$

Le lemme suivant montre que cet algorithme permet de reconstruire l'$AFAR_{\uparrow}$ canonique de L quand $P = P_{\uparrow}(L)$ et que S contient l'échantillon caractéristique.

Lemme C.9. *Soit L un langage régulier d'arbres de $\mathcal{L}_{\uparrow}$. Soit S un échantillon de termes tel que $S_{\uparrow car}(L) \subset S$ et $S \subset L$. Soit $P = P_{\uparrow}(L)$. Alors, $A_{\uparrow}(S, P)$ est l'$AFAR_{\uparrow}$ canonique de L.*

Preuve :

On remarque tout d'abord que soient t et t' deux termes tels que $t^{-1}L$ est premier, $t^{-1}L \subseteq t'^{-1}L \Leftrightarrow t^{-1}L \cap t'^{-1}L \neq \emptyset$ car les résiduels premiers de L sont disjoints (lemme C.2). Par ailleurs, le choix de S et de P fait que :

- *pour tout $t \in P$, $t^{-1}L \subseteq L \Leftrightarrow t^{-1}S \cap S \neq \emptyset$ et*
- *soit $n \in \mathbb{N}$, soit $f \in \mathcal{F}_n$, soit $t \in P$ et soient $t_1, \ldots, t_n \in P$, $t^{-1}L \subseteq f(t_1, \ldots, t_n)^{-1}L \Leftrightarrow t^{-1}S \cap (f(t_1, \ldots, t_n))^{-1}S \neq \emptyset$.*

L'algorithme construit donc bien l'$AFAR_{\uparrow}$ canonique de L. ◄

On peut alors démontrer le théorème suivant :

Théorème C.3. *La classe $\mathcal{L}_{\uparrow}$ est identifiable à la limite par exemples positifs.*

L'algorithme permettant ce résultat se déroule en deux temps : recherche des résiduels premiers puis construction de l'$AFAR_{\uparrow}$ canonique. A chaque itération h, il travaille sur deux ensembles de termes : K_h qui contient l'ensemble des termes de hauteur inférieure ou égale à h utilisés pour construire le graphe et P_h l'ensemble des termes caractéristiques des résiduels premiers rencontrés.

Apprend$_{\uparrow}$

Entrée : S
$h = 0$
$K_0 = \{t \mid t \in Pref_{\uparrow}(S), h(t) = 0\}$
Faire
Construire le graphe $G_{K_h,S}$
$P'_h = P(G_{K_h,S})$
$P_h = Carac(P')$
$h = h + 1$
$K_h = K_{h-1} \cup$ tous les termes de $Pref_{\uparrow}(S)$ de hauteur h
dont tous les sous termes sont dans P_h
Tant Que $K_h \neq K_{h-1}$

Si $A_{\uparrow}(S, P_{h-1})$ est consistant avec S **Alors** sortir $A_{\uparrow}(S, P_{h-1})$ **Sinon** Sortir $AP_{\uparrow}(S)$

Lemme C.10. *Soient L un langage régulier d'arbres de $\mathcal{L}_{\uparrow}$ et S un échantillon de L. Si $S \supset S_{\uparrow car}(L)$, Apprend$_{\uparrow}(S)$ sort l'$AFAR_{\uparrow}$ canonique de L.*

Preuve :

On montre tout d'abord qu'à chaque étape h, on a $K_h = K_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$ et $P_h = P_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$. On montre pour cela les trois points suivants :

– $K_0 = K_{\uparrow}(L) \cap \mathcal{T}_0(\mathcal{F})$.

Par construction, $K_0 = Pref_{\uparrow}(S) \cap \mathcal{T}_0(\mathcal{F})$. Comme $S \supset S_{\uparrow car(L)}$, $K(L) \subseteq Pref_{\uparrow}(S)$. Comme $K_{\uparrow}(L)$ contient $\mathcal{F}_0$, $K_0 = \mathcal{F}_0 = K_{\uparrow}(L) \cap \mathcal{T}_0(\mathcal{F})$,

– Pour tout h, si $K_h = K_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$ et si, soit $h = 0$, soit $P_{h-1} = P_{\uparrow}(L) \cap \mathcal{T}_{\leq h-1}(\mathcal{F})$, alors $P_h = P_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$.

Comme $S \supset S_{\uparrow car}(L)$, si t et $t' \in K_{\uparrow}(L)$, on a $(t^{-1}L \cap t'^{-1}L \neq \emptyset) \Leftrightarrow (t^{-1}S \cap t'^{-1}S \neq \emptyset)$ par construction de $S_{\uparrow car}(L)$. Par conséquent $G_{K_h,S} = G_{K_h,L}$. Par ailleurs, L est parfaitement prévisible. Par conséquent, soit $t \in K_h$ et soit $t_p \in P_{\uparrow}(L)$ tel que $t_p^{-1}L \subset t^{-1}L$, on a $h(t_p) \leq h(t) = h$, donc $t_p \in K_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$ qui est égal à K_h par hypothèse. Le graphe $G_{K_h,L}$ est donc parfaitement prévisible. Le lemme C.8 nous permet alors de déduire que $Carac(P(G_{K_h,L})) = P_{\uparrow}(L) \cap K_h$. Comme, par définition de $K_{\uparrow}(L)$, $P_{\uparrow}(L) \subset K_{\uparrow}(L)$, et que, par hypothèse, $K_h = K_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$, on a donc $P_h = Carac(P(G_{K_h,S})) = Carac(P(G_{K_h,L})) = P_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$.

– Pour tout $h > 0$, si $P_h = P_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$ et si $K_h = K_{\uparrow}(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$, alors $K_{h+1} = K_{\uparrow}(L) \cap \mathcal{T}_{\leq h+1}(\mathcal{F})$.

Tout d'abord, comme $K_{h+1} = K_h \cup \{t \in Pref_\uparrow(S) \mid h(t) = h+1$, *tous les sous-termes* t' *de* t *sont dans* $P_h\}$, *que* $P_h = P_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$ *et que* $K_h = K_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$, *on* $K_{h+1} = (K_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})) \cup \{t \in Pref_\uparrow(S) \mid h(t) = h+1$, *tous les sous-termes* t' *de* t *sont dans* $P_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})\}$, *ce qui, suite au premier point et par définition de* $K_\uparrow(L)$, *est égal à* $K_\uparrow(L) \cap \mathcal{T}_{\leq h+1}(\mathcal{F})$. *Par conséquent,* $K_{h+1} = K_\uparrow(L) \cap \mathcal{T}_{\leq h+1}(\mathcal{F})$.

Ce qui prouve que $K_h = K_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$ *et* $P_h = P_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F})$. *Par ailleurs, si* $K_h = K_{h-1}$, *c'est à dire si* $K_\uparrow(L) \cap \mathcal{T}_{\leq h}(\mathcal{F}) = \emptyset$, *cela implique que* $P_\uparrow(L) \cap \mathcal{T}_h(\mathcal{F}) = \emptyset$. *Comme* L *est préfixiel, cela implique également que* $P_\uparrow(L) \cap \mathcal{T}_{\geq h}(\mathcal{F}) = \emptyset$. *Quand l'algorithme s'arrête, on a donc* $P_{h-1} = P_\uparrow(L)$.

Le lemme C.9 permet alors de conclure. ◄

C.5.5 Exemple

Prenons l'automate $A = \langle Q, \mathcal{F}, Q_f, \Delta\rangle$ décrit par $Q = \{q_{art}, q_{nom}, q_{vt}, q_{vnt}, q_{gn}, q_S\}$, $Q_f = \{q_S\}$, $\mathcal{F} = \{le,\ renard,\ lapin,\ mange,\ dort,\ /(,),\ \backslash(,)\}$[6] et Δ décrit par l'ensemble de règles : $le \to q_{art}$, $renard \to q_{nom}$, $lapin \to q_{nom}$, $mange \to q_{vt}|q_{vnt}$, $dort \to q_{vnt}$, $regarde \to q_{vt}$, $/(q_{art}, q_{nom}) \to q_{gn}$, $/(q_{vt}, q_{gn}) \to q_{vnt}$, $\backslash(q_{gn}, q_{vnt}) \to q_S$.

Le langage L généré par A contient notamment les arbres $\backslash(/(le, lapin), dort)$, $\backslash(/(le,$ $renard), /(mange, /(le, lapin)))$, $\backslash(/(le, lapin), /(regarde, /(le, renard)))$. Les langages résiduels premiers de L sont $C_{q_{art}} = (le)^{-1}L$, $C_{q_{nom}} = (lapin)^{-1}L$, $C_{q_{vt}} = (regarde)^{-1}L$, $C_{q_{vnt}} = (dort)^{-1}L$, $C_{q_{gn}} = /(le, lapin)^{-1}L$, $C_{q_S} = \backslash(/(le,\ lapin),\ dort)^{-1}L$. le langage L a un seul langage résiduel composé : $(mange)^{-1}L = (dort)^{-1} \cup (regarde)^{-1}L$. Par conséquent, le langage n'est pas réversible. Les langages résiduels sont premiers, ils forment un ensemble préfixiel, et le langage résiduel composé de L est parfaitement prévisible. Le langage L appartient donc à $\mathcal{L}_\uparrow$. Notons que A est l'$AFAR_\uparrow$ canonique de L.

L'échantillon caractéristique de L est $S_{\uparrow car}(L) = \{\ \backslash(/(le, lapin), dort)$, $\backslash(/(le, lapin),\ /(regarde,\ /(le,\ lapin))),\ \backslash(/(le, lapin), /(mange, /(le, lapin)))$, $\backslash(/(le, lapin), mange),\ \backslash(/(le, renard), dort)\}$. Prenons $S = S_{\uparrow car}(L)$. L'algorithme fonctionne ensuite de la manière suivante :

1. $K_0 = \{le, lapin, renard, regarde, mange, dort\}$,
2. $P_0 = \{le, lapin, regarde, dort\}$,
3. $K_1 = \{le, lapin, renard, regarde, mange, dort, /(le, lapin)\}$,
4. $P_1 = \{le, lapin, regarde, dort, /(le, lapin)\}$,
5. $K_2 = \{le, lapin, renard, regarde, mange, dort, /(le, lapin)$, $/(/(le, lapin), dort),\ /(regarde, /(le, lapin))\}$,
6. $P_2 = \{le, lapin, regarde, dort, /(le, lapin), /(/(le, lapin), dort)\}$,
7. $K_3 = \{le, lapin, renard, regarde, mange, dort, /(le, lapin)$, $/(/(le, lapin), dort),\ /(regarde, /(le, lapin))\} = K_2$.

[6] Cet exemple se veut proche des exemples utilisés par les grammaires catégorielles et utilise comme elles deux symboles d'arité 2 qui sont / et \.

L'algorithme retourne ensuite l'automate $A_{\uparrow}(S, P_2) = A$. On notera que ce langage d'arbres n'étant pas réversible, il s'agit ici d'un exemple de grammaire qui ne peut pas être inféré par les algorithmes classiques d'inférence de langages d'arbres.

Cet exemple illustre par ailleurs l'intérêt que peut avoir la classe $\mathcal{L}_{\uparrow}$ pour le traitement automatique du langage. Comme il est présenté dans [Besombes and Marion, 2002], les langages d'arbres inférés dans ce cas sont des langages d'arbres réversibles. Cette classe de langages a comme principal désavantage de ne pas pouvoir rendre compte de l'ambiguïté du langage. Un mot ne peut ainsi appartenir qu'à une seule catégorie lexicale. La classe $\mathcal{L}_{\uparrow}$ permet de lever en partie cette contrainte : dans l'exemple ci-dessus, le verbe "manger" est un verbe qui est à la fois transitif et intransitif.

C.6 Conclusion

Nous avons étendu les résultats d'identification à la limite par exemples positifs à des classes de langages contenant strictement les langages d'arbres réversibles. Alors que les automates d'arbres réversibles sont des automates déterministes ascendants et déterministes par contexte descendants, nos extensions introduisent du non déterminisme. L'introduction du non déterminisme nous a permis de prendre en considération des cas intéressants du langage naturel qui n'étaient pas traités par les algorithmes précédents. D'un autre côté, nous allons aussi orienter nos recherches vers une meilleure adaptation aux problèmes pratiques (documents XML). Le travail présenté ici peut alors être considéré comme une étape intermédiaire pour la recherche d'algorithmes d'inférences de langages d'arbres à arité non bornée.

Bibliographie

[Angluin, 1980] Angluin, D. (1980). Inductive inference of formal languages from positive data. *Inform. Control*, 45(2) :117–135.

[Angluin, 1982] Angluin, D. (1982). Inference of reversible languages. *J. ACM*, 29(3) :741–765.

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2) :87–106.

[Baumgartner et al., 2001] Baumgartner, R., Flesca, S., and Gottlob, G. (2001). Visual web information extraction with lixto. In *28th International Conference on Very Large Data Bases*, pages 119–128.

[Berlea and Seidl, 2004] Berlea, A. and Seidl, H. (2004). Binary queries for document trees. *Nordic Journal of Computing*, 11(1) :41–71.

[Besombes and Marion, 2002] Besombes, J. and Marion, J. (2002). Apprentissage des langages réguliers d'arbres et applications. In *CAP'2002*, pages 55–70.

[Bruggemann-Klein et al., 2001] Bruggemann-Klein, A., Wood, D., and Murata, M. (2001). Regular tree and regular hedge languages over unranked alphabets : Version 1.

[Carme et al., 2003a] Carme, J., Gilleron, R., Lemay, A., Terlutte, A., and Tommasi, M. (2003a). Residual finite tree automata. In *7th International Conference on Developments in Language Theory*, number 2710 in Lecture Notes in Computer Science, pages 171 – 182. Springer Verlag.

[Carme et al., 2003b] Carme, J., Lemay, A., and Terlutte, A. (2003b). Identification à la limite de langages réguliers d'arbres à résiduels premiers disjoints. In *Proceedings of CAP'03*, pages 217 – 232. Presses Universitaires de Grenoble.

[Chidlovskii, 2001] Chidlovskii, B. (2001). Wrapping web information providers by transducer induction. In *Proc. European Conference on Machine Learning*, volume 2167 of *Lecture Notes in Artificial Intelligence*, pages 61 – 73.

[Cohen et al., 2003] Cohen, W., Hurst, M., and Jensen, L. (2003). *Web Document Analysis : Challenges and Opportunities*, chapter A Flexible Learning System for Wrapping Tables and Lists in HTML Documents. World Scientific.

[Comon et al., 1997] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (1997). Tree automata techniques and applications. Available on : http ://www.grappa.univ-lille3.fr/tata.

[Coste and Fredouille, 2000] Coste, F. and Fredouille, D. (2000). Efficient ambiguity detection in c-nfa. In *Grammatical Inference : Algorithms and Applications*, volume 1891 of *Lecture Notes in Artificial Intelligence*. Springer Verlag.

[Courcelle, 1992] Courcelle, B. (1992). Recognizable sets of unrooted trees. In Nivat, M. and Podelski, A., editors, *Tree Automata and Languages*. Elsevier Science.

[de la Higuera, 1997] de la Higuera, C. (1997). Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27 :125–137.

[Denis et al., 2001] Denis, F., Lemay, A., and Terlutte, A. (2001). Learning regular languages using rfsa. In *ALT 2001*, number 2225 in Lecture Notes in Artificial Intelligence, pages 348–363. Springer Verlag.

[Denis et al., 2002a] Denis, F., Lemay, A., and Terlutte, A. (2002a). Quelques classes de langages identifiables à la limite par exemples positifs. In *CAP 2002*, pages 43–54.

[Denis et al., 2002b] Denis, F., Lemay, A., and Terlutte, A. (2002b). Residual finite state automata. *Fundamenta Informaticae*, 51(4) :339–368.

[Denis et al., 2002c] Denis, F., Lemay, A., and Terlutte, A. (2002c). Some language classes identifiable in the limit from positive data. In *ICGI 2002*, number 2484 in Lecture Notes in Artificial Intelligence, pages 63–76. Springer Verlag.

[Denis et al., 2004] Denis, F., Lemay, A., and Terlutte, A. (2004). Learning regular languages using rfsas. *Theoretical Computer Science*, 313(2) :267–294.

[Drewes and Hogberg, 2003] Drewes, F. and Hogberg, J. (2003). Learning a regular tree language from a teacher. In *D.L.T. 2003*, volume 2710 of *Lecture Notes in Computer Science*, pages 279–291.

[Dudau-Sofronie et al., 2002] Dudau-Sofronie, D., Tellier, I., and Tommasi, M. (2002). A tool for language learning based on categorial grammars and semantic information. In *proceedings of ICGI'2002, International Colloquium on Grammatical Inference (demo session)*, volume 2484 of *Lecture Notes in Artificial Intelligence*, pages 303–305. Springer Verlag.

[Fernau, 2002] Fernau, H. (2002). Learning tree languages from text. In *Proc. 15th Annual Conference on Computational Learning Theory, COLT 2002*, pages 153 – 168.

[Freitag and Kushmerick, 2000] Freitag, D. and Kushmerick, N. (2000). Boosted wrapper induction. In *AAAI/IAAI*, pages 577–583.

[Frick et al., 2003] Frick, M., Grohe, M., and Koch, C. (2003). Query evaluation on compressed trees. In *18th IEEE Symposium on Logic in Computer Science*, pages 188–197.

[Gold, 1967] Gold, E. (1967). Language identification in the limit. *Inform. Control*, 10 :447–474.

[Gold, 1978] Gold, E. (1978). Complexity of automaton identification from given data. *Inform. Control*, 37 :302–320.

[Goldman and Kwek, 2002] Goldman, S. A. and Kwek, S. S. (2002). On learning unions of pattern languages and tree patterns in the mistake bound model. *Theoretical Computer Science*, 288(2) :237 – 254.

[Gottlob and Koch, 2002a] Gottlob, G. and Koch, C. (2002a). Monadic datalog and the expressive power of languages for web information extraction. In *Symposium on Principles of Database Systems (PODS)*, pages 17–28.

[Gottlob and Koch, 2002b] Gottlob, G. and Koch, C. (2002b). Monadic queries over tree-structured data. In *Proceedings of the* 17^{th} *LICS*, Lecture Notes in Computer Science, pages 189–202, Copenhagen. Springer Verlag.

[Grubbs, 1 21] Grubbs, F. (1969, volume=11, number=1, pages=1-21). Procedures for detecting outlying observations in samples. *Technometrics*.

[Gécseg and Steinby, 1984] Gécseg, F. and Steinby, M. (1984). *Tree Automata*. Akademiai Kiado.

[Gécseg and Steinby, 1996] Gécseg, F. and Steinby, M. (1996). Tree languages. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer Verlag.

[Kanazawa, 1996] Kanazawa, M. (1996). Identification in the limit of categorial grammars. *Journal of Logic, Language, and Information*, 5(2) :115–155.

[Koch, 2003] Koch, C. (2003). Efficient processing of expressive node-selecting queries on xml data in secondary storage : A tree automata-based approach. In *Proc. VLDB 2003*.

[Kosala, 2003] Kosala, R. (2003). *Information Extraction by Tree Automata Inference*. PhD thesis, Katholieke Universiteit Leuven.

[Kosala et al., 2003] Kosala, R., Bruynooghe, M., den Bussche, J. V., and Blockeel, H. (2003). Information extraction from web documents based on local unranked tree automaton inference. In *18th International Joint Conference on Artificial Intelligence*, pages 403–408. Morgan Kaufmann.

[Kushmerick, 1997] Kushmerick, N. (1997). *Wrapper Induction for Information Extraction*. PhD thesis, University of Washington.

[Lang et al., 1998] Lang, K. J., Pearlmutter, B. A., and Price, R. A. (1998). Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. *Lecture Notes in Computer Science*, 1433 :1–12.

[Liu et al., 2000] Liu, L., Pu, C., and Han, W. (2000). XWRAP : An XML-enabled wrapper construction system for web information sources. In *ICDE*, pages 611–621.

[Murata et al., 2001] Murata, M., Lee, D., and Mani, M. (2001). "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Montreal, Canada.

[Muslea et al., 1998] Muslea, I., Minton, S., and Knoblock, C. (1998). Stalker : Learning extraction rules for semistructured, web-based information sources. In *Proceedings of AAAI-98 Workshop on AI and Information Integration*.

[Muslea et al., 2002] Muslea, I., Minton, S., and Knoblock, C. (2002). Active + Semi-supervised Learning = Robust Multi-view Learning. In *Proceedings of ICML-2002*, pages 435–442.

[Neumann and Seidl, 1998] Neumann, A. and Seidl, H. (1998). Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*, pages 134–145.

[Neven, 1999] Neven, F. (1999). *Design and Analysis of Query Languages for Structured Documents. Aformal and logical Approach.* PhD thesis, Limburgs Univ.

[Neven, 2002] Neven, F. (2002). Automata theory for xml researchers. *SIGMOD Rec.*, 31(3) :39–46.

[Neven and Bussche, 2002] Neven, F. and Bussche, J. V. D. (2002). Expressiveness of structured document query languages based on attribute grammars. *Journal of the ACM*, 49(1) :56–100.

[Neven and Schwentick, 2002] Neven, F. and Schwentick, T. (2002). Query automata over finite trees. *Theoretical Computer Science*, 275(1-2) :633–674.

[Niehren et al., 2005] Niehren, J., Planque, L., Talbot, J.-M., and Tison, S. (2005). N-ary queries by tree automata. 19th International Workshop on Unification. Submitted to a conference.

[Niehren and Podelski, 1993] Niehren, J. and Podelski, A. (1993). Feature automata and recognizable sets of feature trees. In Gaudel, M.-C. and Jouannaud, J.-P., editors, *TAPSOFT : Theory and Practice of Software Development : Joint International Conference CAAP/FASE/TOOLS.*, volume 668 of *Lecture Notes in Computer Science*, pages 356–375. Springer Verlag.

[Nivat and Podelski, 1997] Nivat, M. and Podelski, A. (1997). Minimal ascending and descending tree automata. *SIAM Journal on Computing*, 26(1) :39–58.

[Oncina and Garcia, 1992] Oncina, J. and Garcia, P. (1992). Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, pages 49–61.

[Oncina and García, 1993] Oncina, J. and García, P. (1993). Inference of recognizable tree sets. Technical report, Departamento de Sistemas Informáticos y Computación, Universidad de Alicante. DSIC-II/47/93.

[Sahuguet and Azavant, 2001] Sahuguet, A. and Azavant, F. (2001). Building intelligent web applications using lightweight wrappers. *Data Knowl. Eng.*, 36(3) :283–316.

[Sakakibara, 1990] Sakakibara, Y. (1990). Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76 :223 – 242.

[Sakakibara, 1992] Sakakibara, Y. (1992). Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1) :23–60.

[Thatcher, 1973] Thatcher, J. (1973). Tree automata : an informal survey. In Aho, A., editor, *Currents in the theory of computing*, pages 143–178. Prentice Hall.

[Thatcher, 1967] Thatcher, J. W. (1967). Characterizing derivation trees of context-free grammars through a generalization of automata theory. *Journal of Comput. and Syst. Sci.*, 1 :317–322.

[Thatcher and Wright, 1968] Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2 :57–82.

[Thomas, 1984] Thomas, W. (1984). Logical aspects in the study of tree languages. In *Proceedings of the 9th International Colloquium on Trees in Algebra and Programming, CAAP '84*, pages 31 – 50.

[Thomas, 1997] Thomas, W. (1997). Languages, automata and logic. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer Verlag.

[Viragh, 1981] Viragh, J. (1981). Deterministic ascending tree automata. *Acta Cybernetica*, 5 :33–42.

Index