Camille Vacher

Accessibilité Inverse dans les Automates d'Arbres à Mémoire d'Ordre Supérieur

Research Report LSV-07-35

October 2007

Laboratoire Spécification et Vérification

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Accessibilité Inverse dans les Automates d'Arbres à Mémoire d'Ordre Supérieur

Camille Vacher

22 octobre 2007

Laboratoire de Spécification et de Vérification
ENS Cachan

Stage de recherche du MPRI

Sous la direction de : Florent JACQUEMARD

Remerciements

Je souhaiterais tout d'abord remercier Florent Jacquemard de m'avoir encadré. Il m'a initié à de nombreux axes de recherche et a su me guider parmi eux, en prenant toujours en compte mes goûts et mes aptitudes. L'ambiance qui règne au sein du LSV a largement contribué au bon déroulement de ce stage. Je remercie tous les membres du LSV, notamment ceux qui ont organisé les séminaires, groupes de travails et les événements de la vie du laboratoire. Merci également aux autres thésards et stagiaires pour les repas animés et pour les nombreux conseils d'ordre scientifique, administratif ou ludique. Merci notamment à ceux avec qui j'ai partagé le bureau. Enfin, je remercie mes colocataires, au sens large, avec qui j'ai passé l'essentiel de mon temps hors-laboratoire durant ces cinq mois.

1 Introduction

Les systèmes logiciels occupent une place toujours plus importante dans le monde. Ils deviennent plus complexes, plus variés et sont en charges de tâches plus critiques. Face à cette croissance rapide, il est nécessaire de s'assurer que ces systèmes s'exécutent correctement, qu'ils disposent de suffisamment de ressources, et que les informations qu'ils produisent sont correctes. Le Laboratoire de Spécification de Vérification concentre ainsi ses activités sur le développement de méthodes formelles pour la vérification de tels systèmes.

La première méthode utilisée pour vérifier le bon déroulement d'un logiciel est le test. Cette pratique, encore largement répandue, s'est beaucoup développée : aujourd'hui, on génére des tests automatiquement qui peuvent par exemple deviner les valeurs critiques dans certaines parties d'un logiciel. Le principal inconvénient des tests est qu'ils ne sont jamais exhaustifs, et de ce fait, n'apportent jamais une preuve du bon fonctionnement d'un logiciel. Les méthodes formelles à l'inverse permettent, en étudiant d'une part le code d'un système logiciel et d'autre part les propriétés qu'il est censé vérifier, de démontrer si oui ou non ces propriétés sont vérifiées par le logiciel, ou sous quelles conditions, et fournissent les preuves de ces propriétés. Ceci se fait tout d'abord en spécifiant correctement le logiciel : en effet, il faut pouvoir manipuler le code du logiciel et exprimer les propriétés que l'on souhaite. Il faut savoir trouver un compromis entre l'expressivité des propriétés que l'on peut écrire, et la facilité et la rapidité avec lesquelles on pourra les vérifier.

Vient ensuite la phase de vérification proprement dite. En fonction du type de système étudié, des propriétés voulues, des spécifications utilisées, il existe différentes méthodes de vérifications, comme le model-checking ou l'interprétation abstraite. Beaucoup de ces méthodes utilisent des automates ou des automates d'arbres : ils modélisent en général l'ensemble des états qu'un système peut théoriquement atteindre. On se pose alors entre autres des questions d'accessibilité : peut-on atteindre un ou plusieurs états qui signalent par exemple une réussite de la procédure de vérification ? Au contraire, est-ce qu'un calcul sur un automate donné évite systématiquement un ou plusieurs états qui représentent un échec de la procédure, ou une erreur critique ? Après ces questions élémentaires, on peut vouloir vérifier des propriétés plus complexes. Supposons qu'un système tolère certaines fautes, et tente de relancer son exécution après une erreur. On souhaiterait alors savoir quels états sont accessibles ou non depuis un autre état, ou ensemble d'états. A l'inverse, on peut considèrer qu'un état n'est valide que si

auparavant on est passé par un autre état (ou par un état quelconque d'un ensemble donné) : auquel cas, on souhaite calculer l'ensemble des états à partir desquels on peut atteindre un certain ensemble d'états.

Différents automates ont été étudiés, plus ou moins expressifs, plus ou moins spécifiques, et vérifiant certaines propriétés particulières. Après les automates finis classiques sont vites apparus les automates à pile : ils permettent ainsi d'avoir une mémoire potentiellement infinie, mais au comportement limité : on ne considère jamais que le dessus de la pile, et les opérations ne permettent que d'empiler ou de dépiler des éléments sur cette pile. Ces automates ont encore été étendus pour donner les automates à pile d'ordre supérieur Ce sont des automates à pile, mais où la simple pile est remplacée par une pile de piles, ou une pile de piles de piles ... Les opérations d'empilage consistent alors à copier la pile qui est déjà en tête. Ils permettent ainsi de modéliser de manière assez précise le comportement des logiciels dans la mémoire de l'ordinateur. En effet, quand un logiciel fait un appel de fonction, il fait une copie du tas, voire de certaines informations dans les registres, et tant qu'il n'est pas sorti de la fonction, il ne travaille que sur ces copies, laissant intactes les versions originales. Ceci est très similaire au comportement des automates à pile d'ordre supérieur. On peut ainsi envisager de vérifier des propriétés sur la manipulation de la mémoire par les logiciels.

Suites aux récents articles de Bouajani et Meyer [BM04] et de Hague et Ong [HO07], on dispose de résultats d'accessibilité assez puissants sur ces automates à pile d'ordre supérieur. Notamment, ils ont prouvé que le Pre^* d'un ensemble régulier de configurations, c'est-à-dire l'ensemble des états à partir desquels on peut atteindre l'ensemble régulier de configurations par une suite finie quelconque de transitions, est lui aussi régulier. Nous avons ici généraliser ce résultat aux automates d'arbres.

De manière similaire aux automates à pile, il existe les automates d'arbres à mémoire. La mémoire est un arbre également, et lors d'une transition d'automate d'arbres, on construit la nouvelle mémoire en fonction des mémoires des fils. On généralise tout d'abord la notion d'ordre supérieur à ces automates d'arbres. On définit les arbres d'ordre supérieur comme des arbres qui sont étiquetés par d'autres arbres. On montre alors qu'on peut coder tous ces arbres d'ordre supérieur comme des arbres classiques. On définit ensuite la notion de Pre^* pour ces automates d'arbres, et on généralise le résultat de Hague et Ong à ces automates.

Il est intéressant d'étendre ce résultat aux arbres car ils peuvent modéli-

ser des systèmes plus complexes. Les arités supérieures des arbres permettent par exemple de modéliser des processus s'exécutant en parallèles, ou différentes exécutions possibles d'un même processus. Mais, surtout, on propose un cadre général commun pour le traitement de ces questions. En effet, les automates utilisés dans les articles [BM04] et [HO07] pour définir les ensembles réguliers sont à chaque fois différents, et sont très dépendants de l'ordre de la pile de l'automate considéré. Notre généralisation aux arbres, qui n'ont plus besoin d'être d'ordre supérieur, permet d'avoir un formalisme d'automate classique indépendant de l'ordre.

2 Préliminaires

Nous proposons ici une introduction succinte aux automates d'arbres et aux automates à pile d'ordre supérieur.

2.1 Arbres et automates d'arbres

On notera N l'ensemble des entiers naturels, et N^* l'ensemble des suites finies sur N (* est l'étoile de Kleene). Un **alphabet à arité borné** est un couple (Σ, ar) où Σ est un ensemble fini (l'alphabet) et où ar est une fonction de Σ dans N. On note $ar(f)$ l'arité du symbole $f \in \Sigma$, et Σ_n l'ensemble des éléments de Σ d'arité n. Les symboles d'arité 0 sont appelés constantes : on supposera qu'un alphabet Σ a toujours au moins une constante. La plupart du temps, nous écrirons alphabet au lieu d'alphabet à arité borné et on supposera l'existence implicite d'une fonction d'arité.

Un **arbre fini t sur un alphabet** Σ est une fonction d'un ensemble de positions $Pos(t) \subseteq N^*$ clos par préfixe vers Σ telle que

$$\forall p \in Pos(t), ar(t(p)) = n \geq 1 \Rightarrow \{j \in N | pj \in Pos(t)\} = \{1, \ldots, n\}.$$

$$\forall p \in Pos(t), ar(t(p)) = 0 \Rightarrow \{j \in N | pj \in Pos(t)\} = \emptyset.$$

Un élément de $Pos(t)$ est appelée une position. Un sous-arbre $t|_p$ d'un arbre t à la position p est un arbre défini par
- $Pos(t|_p) = \{j \in N^* | pj \in Pos(t)\}$
- $\forall q \in Pos(t|_p), t|_p(q) = t(pq)$

D'autre part, on notera $t[u]_p$ l'arbre obtenu en remplaçant, dans t, le sous-arbre $t|_p$ par l'arbre u.

Un **automate d'arbre** sur Σ est un quadruplet $A = (Q, \Sigma, Q_f, \Delta)$ où Q est un ensemble fini d'états, $Q_f \subseteq Q$ est un ensemble d'états finaux et Δ

est un ensemble de règles de transitions de la forme

$$f(q_1(x_1), \dots q_n(x_n)) \rightarrow q(f(x_1, \dots, x_n))$$

avec $n \geq 0$, $f \in \Sigma$ un symbole d'arité n, $q, q_1, \dots q_n \in Q$ et $x_1 \dots x_n$ des variables distinctes. Un *run* d'un automate A sur un arbre t est un arbre r sur Q tel que $Pos(r) = Pos(t)$, et tel que

- $\forall p \in Pos(r), t(p) = f$, si $ar(f) > 0$ alors $f(r(p1)(x_1), \dots, r(pn)(x_n)) \rightarrow r(p)(f(x_1, \dots, x_n)) \in \Delta$
- $\forall p \in Pos(r)$, $t(p) = a$, si $ar(a) = 0$ alors $\rightarrow r(p)(a) \in \Delta$

Un arbre t est reconnu par un automate d'arbres A si et seulement si il existe un *run* acceptant, c'est à dire tel que $r(\epsilon) \in Q_f$, de A sur t. On note $L(A)$ le langage des arbres reconnus par A. On notera également $L(A, q)$ l'ensemble des arbres t tels qu'il existe un *run* de A sur t tels que $r(\epsilon) = q$. Ainsi, on a $L(A) = \bigcup_{q_f \in Q_f} L(A, q_f)$. On trouvera une introduction compléte aux automates d'arbres dans [CDG+97]

D'autre part il existe des automates d'arbres à mémoires. Nous allons en faire une présentation rapide et non-détaillée, puisqu'on introduira un type d'automates d'arbres à mémoire assez spécifique plus tard. Un automate d'arbre à mémoire est un automate d'arbre dont les transitions, au lieu d'être de simples tuples d'états associés à un symbole, dépendent également de l'état de la mémoire de chacun des fils de l'arbre considéré.

Plus précisément le calcul de l'automate se fera sur un arbre et sa mémoire, c'est à dire qu'à tout moment du calcul, la partie de l'arbre considérée sera toujours associée à une mémoire, qui sera un arbre sur un nouvel alphabet, l'alphabet de mémoire. Pour les transitions initiales, on associera un arbre vide ($\perp$). Puis, les règles de transitions, permettront de construire de nouvelles mémoires pour les états suivants. Formellement, les transitions seront de la forme :

$$f(q_1(y_1), q_2(y_2), \dots q_n(y_n)) \rightarrow q(f(op(y_1, \dots, y_n)))$$

où les y_i sont en fait des mémoires, donc des arbres sur l'alphabet de mémoire, et *op* est une opération qui permet de construire une nouvelle mémoire à partir des précédentes. Les contraintes possibles pour ces transitions sont les éléments en tête de chacune des mémoires. Des constructions classiques pour *op* seraient par exemple de choisir une seule des mémoires, ou de créer un arbre dont la racine est étiquetée par un élément quelconque, et dont les fils sont les différentes mémoires. On trouvera une description plus précise des automates d'arbres à une mémoire dans [CCM01].

2.2 Automates à pile d'ordre supérieur

Une pile d'ordre 1 sur un alphabet Σ est un mot de Σ^*. On désigne par "tête" la première lettre de la pile. On peut effectuer deux types d'opérations sur les piles :

- POP_1 qui prend une pile non vide et et enlève l'élément de tête $[\omega_1\omega_2\dots\omega_n] \to [\omega_2\omega_3\dots\omega_n]$
- $PUSH_\omega$ qui ajoute un élément $\omega \in \Sigma$ en tête de pile $[\omega_1\omega_2\dots\omega_n] \to [\omega\omega_1\dots\omega_n]$

Une pile d'ordre $n > 1$ est une pile de piles d'ordre $n-1$, c'est à dire une suite finie et ordonnée, potentiellement vide, de piles d'ordre $n-1$. On appelle "tête" la première pile d'ordre $n-1$ de la suite. On peut effectuer deux types d'opérations sur les piles d'ordre n :

- POP_n qui enlève la tête de la pile : $[\alpha_1\alpha_2\dots\alpha_n] \to [\alpha_2\dots\alpha_n]$ avec les α_i piles d'ordre $n-1$
- $PUSH_n$ qui ajoute une copie de la tête au début de la pile : $[\alpha_1\alpha_2\dots\alpha_n] \to [\alpha_1\alpha_1\alpha_2\dots\alpha_n]$

On note l'ensemble des opérations $POP_1, PUSH_\omega, POP_2, PUSH_2 \dots POP_n, PUSH_n$ par Op_n. On définit la tête d'ordre i, d'une pile d'ordre n s, que l'on notera $top_n(s)$, par récurrence :

- le premier élément de la pile si $i = n$
- la tête d'ordre i du premier élément de la pile sinon

Exemple 1 *Voici une pile d'ordre* 3 *sur l'alphabet* $\{a, b\}$

$$[[[aba][aab][ba]][[ab][bbb]]]$$

Et les têtes respectivement d'ordre $3, 2$ *puis* 1 *qui correspondent :*

$$[[aba][aab][ba]]$$

$$[aba]$$

$$a$$

Un automate à pile d'ordre supérieur, auquel on fixe un ordre n, est un tuple $A = (Q, \Sigma, q_0, s_0, Q_f, \Gamma, \Delta)$ où Q est un ensemble fini d'états, $q_0 \in Q$ est l'état initial, s_0 est une pile d'ordre n sur Γ, $Q_f \subseteq Q$ est un ensemble d'états finaux, Σ est l'alphabet d'entrée, Γ est l'alphabet de pile, et Δ est un ensemble de règles de transitions de la forme : (q, a, γ, q', op) avec $q, q' \in Q$, $a \in \Sigma$, $\gamma \in Gamma$ et $op \in Op_n$.

Une configuration d'un automate à pile d'ordre supérieur est une paire (q, s) constituée d'un état de l'automate ainsi que d'une pile d'ordre n. On peut passer d'une configuration (q, s) à une configuration (q, s') en lisant une lettre $a \in \Sigma$ si et seulement si il existe une transition $(q, a, \gamma, q', op) \in Delta$ telle que la tête d'ordre 1 de s est définie et égale à γ et telle que $s' = op(s)$. Un mot de Σ^* est reconnu par A si, en partant de la configuration initiale (q_0, s_0) et en lisant chacune des lettres du mots, on peut atteindre une configuration (q, s) avec $q \in Q_f$.

L'accessibilité dans les automates à pile d'ordre supérieur n'est pas évidente. Dans [BM04] Bouajani et Meyer montrent que l'accessibilité en avant est difficile : en effet, étant donné une configuration ou un ensemble régulier de configurations, le successeur immédiat est contextuel (reconnaissable par une grammaire contextuelle) ce qui est notamment dû au fait que les opérations $PUSH_n$ font des copies de la tête. Aucun processus de décision n'est connu pour l'instant pour le $Post^*$ (l'ensemble des successeurs par un nombre quelconque d'opérations). Par contre, dans ce même article, ils commencent à montrer que le Pre^* d'un ensemble régulier est lui-même régulier. Ils le montrent dans le cas d'un automate à pile d'ordre supérieur n'ayant qu'un état dans Q. Dans [HO07], Hague et Ong généralisent le résultat à l'ensemble des automates à pile d'ordre supérieur. On détaillera dans la section suivante la généralisation de la notion d'automate à pile d'ordre supérieur aux automates d'arbres ainsi que la généralisation de ce résultat à ces nouveaux automates.

3 Automates d'arbres à mémoire d'ordre supérieur

3.1 Arbres d'ordre supérieur

Soit Σ un alphabet fini. Un arbre d'ordre 1 sur Σ est un arbre binaire dont les noeuds sont étiquetés par Σ. Un arbre d'ordre $n, n > 1$ sur Σ est un arbre binaire dont les noeuds sont étiquetés par un arbre d'ordre $n - 1$. Formellement :

$$\begin{aligned} t_1 &:= a \mid a(t_1, t_1), a \in \Sigma \\ t_n &:= t_{n-1} \mid t_{n-1}(t_n, t_n), \forall n > 1 \end{aligned}$$

À partir de maintenant, quand on parlera d'arbre d'ordre 0, on fera référence à un élément de Σ. Pour tout arbre d'ordre n sur Σ et pour tout $i, 0 \leq i \leq n - 1$, on définit la fonction $head_i$ par :

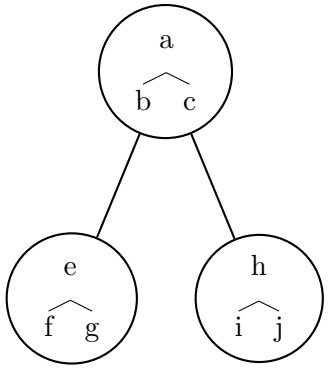


FIG. 1 – Exemple d'arbre d'ordre 2

- Si $n = 1$ et $i = 0$
  - si $t_1 = a \in \Sigma$, $head_0(t_1) = a$
  - si $t_1 = a(t'_1, t''_1)$, $head_0(t_1) = a$
- si $n > 1$ et $i = n - 1$
  - si $t_n = t_{n-1}$, $head_{n-1}(t_n) = t_{n-1}$
  - si $t_n = t_{n-1}(t'_n, t''_n)$, $head_{n-1}(t_n) = t_{n-1}$
- si $n > 1$ et $i < n - 1$
  - si $t_n = t_{n-1}$, $head_i(t_n) = head_i(t_{n-1})$
  - si $t_n = t_{n-1}(t'_n, t''_n)$, $head_i(t_n) = head_i(t_{n-1})$

Intuitivement, $head_i$ associe à un arbre d'ordre n l'arbre d'ordre i qui étiquette la racine de l'arbre d'ordre i qui étiquette la racine de l'arbre d'ordre $i + 2$ ... qui étiquette la racine de l'arbre d'ordre n. On appelle $head_i(t_n)$ la tête d'ordre i de t_n. Par convention on estimera que $head_n(t_n) = t_n$.

D'autre part, soit t_n un arbre d'ordre n et $t_i, 1 \leq i < n$ un arbre d'ordre i, on notera $t_n[t_i]_i$ l'arbre t_n dont la tête d'ordre i est remplacée par t_i.

3.2 Automates à mémoire d'ordre supérieure

On définit ici une nouvelle classe d'automates d'arbres à mémoire. Il s'agit d'automates d'arbres ascendants, donc opérant sur des arbres finis, dont la mémoire est un arbre d'ordre n. Les opérations que l'on peut appliquer sur la mémoire sont des PUSH ou des POP d'ordre $\leq n$. Les PUSH vont par ailleurs sélectionner une des deux mémoires considérées, et les POP vont choisir une des deux mémoires ainsi qu'une direction (droite ou gauche).

- Un PUSH d'ordre i va construire un arbre d'ordre i dont la racine sera

la tête d'ordre $i-1$ de la mémoire sélectionnée (ou un élément de Σ dans le cas d'un PUSH d'ordre 1), et dont les fils seront les têtes d'ordre i des deux mémoires. On remplace alors, dans la mémoire sélectionnée, la tête d'ordre i par cette construction. Dans le cas d'un PUSH d'ordre n, le résultat sera donc la construction elle-même.
- Un POP d'ordre i remplace, dans la mémoire sélectionnée, la tête d'ordre i par un des fils (gauche ou droit) de cette même tête.

Définition 1 *Un* Automate d'arbre à mémoire d'ordre n *sur* Σ *est un tuple* (Γ, Q, Q_f, Δ) *avec* Γ *un alphabet fini,* Q *un ensemble fini d'états,* $Q_f \subseteq Q$ *un ensemble d'états finaux et* Δ *un ensemble de règles de transition de la forme* $(Q \times Q \times \Gamma \times O \times Q)$ *où* O *est l'ensemble des opérations sur les mémoires suivant :*
INT *ou* $\{PUSH_1^a, PUSH_2^a\}_{a\in\Gamma}$ *ou* $\{PUSH_1^i, PUSH_2^i\}_{2\leq i\leq n}$ *ou* $\{POP_{11}^i, POP_{12}^i, POP_{21}^i, POP_{22}^i\}_{1\leq i\leq n}$.

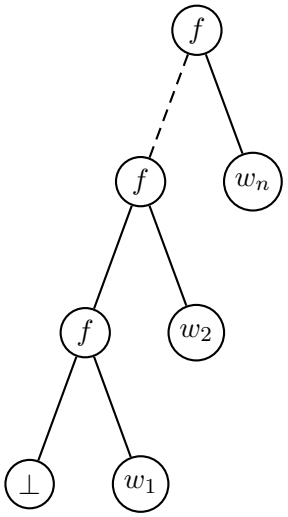
Les opérations sur les mémoires donnent l'ensemble des transitions possibles suivantes :

INT	a		$\rightarrow q(\bot)$
PUSH_1^{a}	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_1[a(head_1(y_1), head_1(y_2))]_1)$
PUSH_2^{a}	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_2[a(head_1(y_1), head_1(y_2))]_1)$
PUSH_1^{i}	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_1[head_{i-1}(y_1)(head_i(y_1), head_i(y_2))]_i)$
PUSH_2^{i}	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_2[head_{i-1}(y_2)(head_i(y_1), head_i(y_2))]_i)$
$\text{POP}_{11}^{\text{i}}$	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_1[left(head_i(y_1))]_i)$
$\text{POP}_{12}^{\text{i}}$	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_1[right(head_i(y_1))]_i)$
$\text{POP}_{21}^{\text{i}}$	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_2[left(head_i(y_2))]_i)$
$\text{POP}_{22}^{\text{i}}$	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_2[right(head_i(y_2))]_i)$

où $a \in \Sigma$, $\bot$ est vu comme l'arbre d'ordre n dont toutes les têtes d'ordre $i < n$ ont un seul noeud et dont la tête d'ordre 1 est $\bot$, q_1, q_2 et $q \in Q$, et y_1 et y_2 sont des arbres d'ordre n. Les transitions $(q_1, q_2, \gamma, \text{PUSH}, q)$ et $(q_1, q_2, \gamma, \text{POP}, q)$ ne peuvent se produirent que si la tête d'ordre 0 de la mémoire concernée est γ.

3.3 Simulation des automates à pile d'ordre supérieur

On peut facilement simuler le comportement d'une pile d'ordre n avec un arbre d'ordre n et par la-même reproduire le calcul d'un automate à pile d'ordre supérieur avec un automate d'arbre à mémoire d'ordre supérieur. Il faut tout d'abord pouvoir représenter tout mot d'entrée de l'automate $w = w_1 w_2 \dots w_k$ par un arbre binaire. On utilise le codage suivant :



L'ensemble des états de l'automate d'arbre est composé des états de l'automate à pile, plus un état Q_a pour tout a de l'alphabet d'entrée. Les états finaux sont les mêmes. Pour chaque état initial q_0 on crée la transition :

$$\text{INT } \bot \rightarrow q_0(\bot)$$

Pour tout a de l'alphabet d'entrée, on crée la transition :

$$\text{INT } a \rightarrow q_a(\bot)$$

Ensuite on associe à chaque transition de l'automate à pile, une nouvelle transition de l'automate d'arbre.

$$\begin{array}{rcl} (q_j, a, \gamma, push_i, q_k) & \Longrightarrow & (q_j, q_a, \gamma, PUSH_1^i, q_k) \\ (q_j, a, \gamma, pop_i, q_k) & \Longrightarrow & (q_j, q_a, \gamma, POP_{11}^i, q_k) \end{array}$$

Il est facile de montrer que tout calcul réussi de l'automate à pile sur un mot w correspond à un calcul réussi de l'automate d'arbre à mémoire sur l'arbre codant w, et réciproquement.

3.4 Codage des arbres d'ordre supérieur

Les arbres d'ordre supérieur permettent une généralisation assez intuitive des automates à pile d'ordre supérieur des automates d'arbre à mémoire. Cependant, le formalisme est un peu lourd, et nous ne disposons pas d'automate pour manipuler ces nouveaux outils. Pour résoudre ces problèmes, on

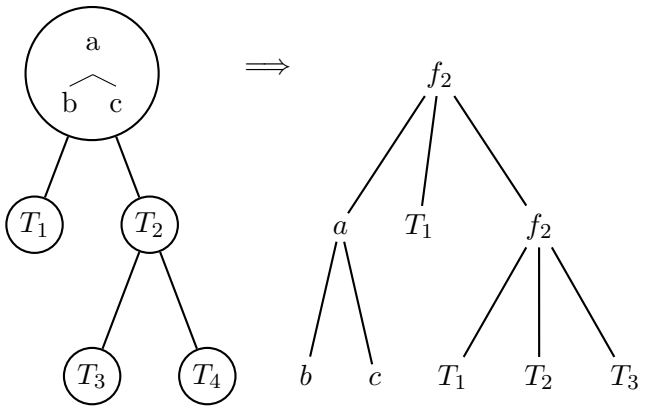


FIG. 2 – Exemple d'encodage d'un arbre d'ordre 2

va représenter les arbres d'ordre quelconque par des arbres standards d'arité bornée. L'idée principale étant d'augmenter la parité des noeuds pour pouvoir coder les différents ordres.

On considère des arbres d'ordre n sur Σ. On peut déclarer que tous les symboles de Σ sont d'arité 2 sauf les symboles de feuille qui sont d'arité 0. Les arbres d'ordre $\leq n$ seront représentés par des arbres classiques d'arité borné étiquetés par $\Sigma \bigcup \{f_i\}_{2 \leq i \leq n}$ où les f_i sont des symboles d'arité 3. Les arbres d'ordre 1 restent inchangés dans cette représentation. Un arbre d'ordre i est défini par induction : $f_i(h, l, r)$ où h est une représentation d'un arbre d'ordre $< i$ qui représente la tête d'ordre $i-1$ de l'arbre, et où l et r sont des arbres d'ordres $\leq i$ qui représentent les fils gauches et droits d'ordre i. Si un arbre d'ordre i n'a qu'un noeud (une racine sans fils), on se contente de le représenter comme un arbre d'ordre $i-1$. On saura interpréter que cet arbre est d'ordre i dont le seul noeud a pour tête d'ordre $i-1$ l'arbre effectivement représenté.

La première chose à remarquer, c'est qu'il est très facile d'obtenir la tête d'ordre i. Il suffit pour celà de parcourir l'arbre en prenant toujours le premier fils (la racine) tant que l'on a un symbole $f_j, j > i$. Le premier que l'on croise qui ne vérifie pas c'est forcément un sous-arbre d'ordre $\leq i$ qui représente la tête d'ordre i. Comme la racine est toujours un arbre d'ordre strictement inférieur à l'arbre d'origine, on descend au plus $n-1$ fois dans l'arbre. Enfin, l'opération qui consiste à remplacer la tête d'ordre i par un autre arbre d'ordre i (obtenir $t_n[t_i]_i$) n'est pas plus difficile : on descend

comme pour obtenir la tête d'ordre i, mais à la place de l'extraire, on la remplace par le sous-arbre voulu. Encore une fois, le nombre de fois qu'il faudra descendre est borné par n. Ainsi, toutes les opérations PUSH et POP peuvent être réalisées en un temps borné linéairement par n. Surtout, elles peuvent être décrites de manière formelle en utilisant des termes de profondeur au plus n.

4 Le problème du Pre^*

Dans [HO07], Hague et Ong montrent que pour tout ensemble régulier C_{Init} de configurations d'un automate à piles d'ordre supérieur alternant, on peut calculer l'ensemble $Pre^*(C_{Init})$ des configurations depuis lesquelles on peut atteindre l'ensemble C_{Init}. La notion de régularité utilisée dépend d'un nouveau type d'automate appelé (multi)-automate de pile d'ordre n. Le mode opératoire de ces automates peut être facilement interprété comme un automate classique sur des mots finis. Aussi, la régularité d'un ensemble de configurations est équivalente à la régularité d'un ensemble de mots finis. Si l'on représente les piles d'ordre supérieur par des arbres, on s'attend donc à ce qu'un ensemble régulier de pile, au sens de Hague et Ong, devienne un ensemble régulier d'arbre, au sens classique des automates d'arbres.

On va montrer que c'est le cas pour notre représentation, en associant à tout multi-automate de pile d'ordre n un automate d'arbre ascendant équivalent. On va ensuite généraliser le résultat du Pre^* en étendant cette notion aux arbres d'ordre supérieur.

4.1 Construction de l'automate reconnaissant Pre^*

Soit un ensemble C de configurations (q, t) d'un automate à mémoire d'ordre n, $A = (\Gamma, Q, Q_f, \Delta)$. On note $Pre(C)$ l'ensemble des configurations (q', t'), tel qu'il existe une transition $(q', q'', o, a, \gamma, q)$ telle que $top(t') = \gamma$ et $o(t') = t$. On note $Pre^*(C)$ la clôture reflexive transitive de C par Pre.

On va montrer que si C est régulier, alors $Pre^*(C)$ l'est aussi. On va construire une suite d'automates d'arbres $A_i, 0 \leq i$ tels que $C = L(A_0)$ et tels que $L(A_i) \subset L(A_{i+1}) \subset Pre^*(C)$. On montrera qu'on obtient ainsi un point fixe après un nombre fini d'étape qui reconnait exactement $Pre^*(C)$.

4.1.1 Construction de l'automate A_0

On va s'intéresser au cas où A est un automate à mémoire d'ordre 2, pour étudier le cas général plus tard. Soit un automate d'arbre ascen-

dant $B = (P, \Gamma \bigcup \{f_2\}, P_f, \Delta')$. On impose, sans perte de généralité que $P_f = \{p_{f_1}, p_{f_2}, \ldots, p_{f_n}\}$ où $n = |Q|$ et qu'il n'existe pas dans Δ de règle de transition ayant un état p_{f_i} dans ses prémisses. Plus tard, on notera souvent q_i au lieu de p_{f_i} quand le contexte permet d'éviter les confusions. Une configuration (q_i, t) de l'automate A est reconnue par B si et seulement si $t \in L(B, p_{f_i})$.

Les automates de la suite A_i reconnaitront des arbres sur $\Gamma \bigcup \{f_2\}$ et auront pour ensemble d'état $(P \bigcup P^1 \bigcup P^2) \times \Gamma$. On définit $P_1 = (P_f^1 \times (P \setminus P_f)^2) \bigcup P_f^1$ et $P_2 = P_f^2$ où P_f^1 et P_f^2 sont deux copies de P_f. On indiquera un état de $P_{1|2}$ avec la notation $q^1_{i,p',p''}$ ou $q_i^{1|2}$. On justifiera plus tard l'interêt de ces états annexes et de leurs formes.

On va associer à toute transition de B un ensemble de transitions qui seront des transitions de A_0. On ajoute les règles de A_0 définies dans le tableau ci-dessous pour tous $\gamma_1, \gamma_2, \gamma_3, \gamma \in \Gamma$, et pour tous $p_1, p_2, p_3, p \in P$.

Transition de B	Transition de A_0
$\rightarrow p(\gamma)$	$\rightarrow (p, \gamma)(\gamma)$
$\gamma(p_1(x), p_2(y)) \rightarrow p(\gamma(x, y))$	$\gamma((p_1, \gamma_1)(x), (p_2, \gamma_2)(y))$
	$\rightarrow (p, \gamma)(\gamma(x, y))$
$f_2(p_1(x), p_2(y), p_3(z))$	$f_2((p_1, \gamma_1)(x), (p_2, \gamma_2)(y), (p_3, \gamma_3)(z))$
$\rightarrow p(f_2(x, y, z))$	$\rightarrow (p, \gamma_1)(f_2(x, y, z))$

On considère comme ensemble d'états finaux $\{(p_f, \gamma) | p_f \in P_f\}$. Il est assez évident, et on le montrera par la suite, que cet automate reconnaît exactement $L(B)$. On souhaite, à chaque étape, rajouter un ensemble de règles dans A_i pour pouvoir reconnaître $Pre(L(A_i))$. Si l'on s'y prend directement en rajoutant des règles de manière intuitive pour chaque transition dans l'automate à mémoire, on s'apercoit qu'on a besoin, pour les règles $POP_1 et POP_2$, d'ajouter plusieurs règles. Il nous faut alors ajouter des états intermédiaires entre ces règles.

Ces états devront être indépendants des états "classiques" ainsi que les uns par rapport aux autres. C'est la raison d'être des ensembles P_1 et P_2 et de leurs formes. En effet, pour éviter de créer des nouvelles transitions qui utilisent des états intermédiaires, et risquent ainsi de faire reconnaître des arbres incorrects, on va éviter un maximum de créer de telles transitions. Cela va donc poser un problème pour reconnaître un arbre qui est un prédecesseur par un nombre quelconque de règles, dont plusieurs POP_i, d'un arbre de B. Aussi, pour permettre de simuler une application d'un nombre quelconque de règles de POP_i, on va directement ajouter des transitions sur

ces états intermédiaires. Aussi, si par la suite on peut reconnaître un arbre qui représente le prédecesseur par une règle POP_i d'un autre arbre déjà reconnaissable, l'ensemble des transitions que l'on va maintenant ajouter, vont permettre de reconnaitre tout prédecesseur de ce même arbre en remplaçant la simple règle POP_i par un nombre fini de règles POP_i dont la dernière est l'originale.

Ainsi pour chaque règle POP_1 dans A de la forme

$$q_{j_1}(f_2(a(x,y),l,r)), q_{j_2}(z) \to q_k(f_2(x,l,r))$$

on ajoute l'ensemble des règles suivantes à l'automate A_0 :

$$\langle q^1_{k,p',p''}, b_1\rangle(x), \langle p_2, b_2\rangle(y) \to \langle p^1_{j_1,p',p''}, a\rangle(a(x,y))$$

pour tous $p', p'' \in P$ et tous p_2, b_2, tels que $L(A_0, \langle p_2, b_2\rangle) \neq \emptyset$.

Ensuite, pour chaque règle POP_1 de A de la forme

$$q_{j_1}(a(x,y)), q_{j_2}(z) \to q_k(x)$$

on ajoute l'ensemble des règles suivantes à l'automate A_0 :

$$\langle q^1_k, b_1\rangle(x), \langle p_2, b_2\rangle(y) \to \langle q^1_{j_1}, a\rangle(a(x,y))$$

ainsi que l'ensemble des règles

$$\langle q^1_k, b_1\rangle(x), \langle p_2, b_2\rangle(y) \to \langle q_{j_1}, a\rangle(a(x,y))$$

pour tous p_2, b_2, tels que $L(A_0, \langle p_2, b_2\rangle) \neq \emptyset$.

De même pour les règles POP_2 de A de la forme

$$q_{j_1}(f_2(b(x,y),l_1,r_1)), q_{j_2}(f_2(z,l_2,r_2)) \to q_k(l_1)$$

on rajoute l'ensemble des règles

$$\langle p_1, b\rangle(x), \langle q^1_k, b_2\rangle(y), \langle p_3, b_3\rangle(y) \longrightarrow \langle q^1_{j_1}, b\rangle$$

pour tous $p_1, p_3 \in P$ et $b_3 \in \Gamma$.

4.1.2 Construction de A_i

On suppose qu'on dispose d'un automate d'arbre A_{i-1} tel que $L(B) \subseteq L(A_{i-1}) \subseteq Pre^*(L(B))$, et on va étendre cet automate pour obtenir A_i tel

que $L(A_{i-1}) \subseteq L(A_i) \subseteq Pre^*(L(B))$. On va créer un automate A_i en considérant une transition de l'automate d'arbre à mémoire A, et en ajoutant des règles à A_{i-1} pour qu'il reconnaisse les prédécesseurs, par cette transition, des arbres qu'il reconnaissait déjà. Traitons au cas par cas, les différentes formes de chaque type de règle :

PUSH$_\mathbf{a}$:
$\boldsymbol{q_{j_1}(f_2(b(x,y),l_1,r_1)), q_{j_2}(f_2(z,l_2,r_2)) \to q_k(f_2(a(b(x,y),z),l_1,r_1))}$
pour chaque couple de règles

$$\langle p_1, a\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q_k, a\rangle(f_2(x,y,z))$$

et

$$\langle p'_1, b\rangle(x), \langle p'_2, c'_2\rangle(y) \to \langle p_1, a\rangle(a(x,y))$$

on ajoute la règle

$$\langle p'_1, b\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q_{j_1}, b\rangle(f_2(x,y,z))$$

$\boldsymbol{q_{j_1}(b(x,y)), q_{j_2}(f_2(z,l_2,r_2)) \to q_k(a(b(x,y),z))}$
pour chaque couple de règles

$$\langle p_1, b\rangle(x), \langle p_2, c_2\rangle(y) \to \langle q_k, q, a\rangle(a(x,y))$$

et

$$\langle p'_1, c'_1\rangle(x), \langle p'_2, c'_2\rangle \to \langle p_1, b\rangle(b(x,y))$$

on ajoute la règle

$$\langle p'_1, c'_1\rangle(x), \langle p'_2, c'_2\rangle(y) \to \langle q_{j_1}, b\rangle(b(x,y))$$

PUSH$_\mathbf{2}$:
$\boldsymbol{q_{j_1}(f_2(b(x,y),l_1,r_1)), q_{j_2}(z) \to q_k(f_2(b(x,y),f_2(b(x,y,l_1,r_1)),z))}$
pour chaque couple de règles

$$\langle p_1, b\rangle(x), \langle p_2, b\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q_k, b\rangle(f_2(x,y,z))$$

et

$$\langle p_1, b\rangle(x), \langle p'_2, c'_2\rangle(y), \langle p'_3, c'_3\rangle(y) \to \langle p_2, b\rangle(f_2(x,y,z))$$

on ajoute la règle

$$\langle p_1, b\rangle(x), \langle p'_2, c'_2\rangle(y), \langle p'_3, c'_3\rangle(z) \to \langle q_{j_1}, b\rangle(f_2(x,y,z))$$

$\mathbf{POP_1}$:
Pour les règles de la forme $\boldsymbol{q_{j_1}(f_2(b(x,y),l_1,r_1)), q_{j_2}(z) \to q_k(f_2(x,l_1,r_1))}$
pour chaque transition

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y)\langle p_3, c_3\rangle \to \langle q_k, c_1\rangle(f_2(x,y,z))$$

on ajoute l'ensemble des transitions

$$\langle p_1, c_1\rangle(x), \langle p'_2, c'_2\rangle(y) \to \langle q^1_{j_1,p_2,p_3}, b\rangle(b(x,y))$$

tels que $L(A_{i-1}, \langle p'_2, c'_2\rangle) \neq \emptyset$ ainsi que les transitions

$$\langle q^1_{l,p_2,p_3}, c'_1\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q_l, c'_1\rangle(f_2(x,y,z))$$

pour tout $1 \leq l \leq |Q|$.

si la règle POP_1 est de la forme $\boldsymbol{q_{j_1}(b(x,y)), q_{j_2}(z) \longrightarrow q_k(x)}$
pour chaque transition

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y) \to \langle q_k, c_3\rangle(c_3(x,y))$$

on ajoute la transition

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y) \to \langle q^1_{j_1}, c_3\rangle(c_3(x,y))$$

ainsi que l'ensemble des transitions

$$\langle q^1_l, c'_1\rangle(x), \langle p'_2, c'_2\rangle(y) \to \langle q_l, b\rangle(b(x,y))$$

pour tout $1 \leq l \leq |Q|$ et telles que $L(A_{i-1}, \langle p'_2, c'_2\rangle \neq \emptyset$.

Et pour chaque transition initiale

$$\longrightarrow \langle q_k, c\rangle(c)$$

on ajoute la transition

$$\longrightarrow \langle q^1_{j_1}, b\rangle(b)$$

et l'ensemble des transitions

$$\langle q^1_l, c_1\rangle(x), \langle p_2, c_2\rangle(y) \longrightarrow \langle q_l, b\rangle(b(x,y))$$

pour tout $1 \leq l \leq |Q|$ et pour tous p_2, c_2 tels que $L(A_{i-1}, \langle p_2, c_2\rangle) \neq \emptyset$.

$\mathbf{POP_2}$:
$\boldsymbol{q_{j_1}(f_2(b(x,y),l_1,r_1)), q_{j_2}(f_2(z,l_2,r_2)) \to q_k(l_1)}$
pour chaque règle

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q_k, c_1\rangle(f_2(x,y,z))$$

on ajoute la règle

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q^2_{j_1}, c_1\rangle(f_2(x, y, z))$$

ainsi que l'ensemble des règles

$$\langle p'_1, c'_1\rangle(x), \langle q^2_{j_1}, c_1\rangle(y), \langle p'_3, c'_3\rangle(z) \to \langle q_i, c'_1\rangle(f_2(x, y, z))$$

pour tous $p'_1, p'_3 \in P$ et tous $c'_1, c'_3 \in$ tels que $L(A_i, \langle p'_1, c'_1\rangle) \neq \emptyset$ et $L(A_i, \langle p'_3, c'_3\rangle) \neq \emptyset$.

4.1.3 Correction de la construction

Théorème 1 *L'union des automates A_i reconnaît $Pre^*(L(B))$.*

Lemme 1 $L(B) = L(A_0)$

preuve : clairement, on peut ignorer les règles contenant les états de P^1 et P^2 dans A_0 car aucun de ces états n'est accessible ni co-accessible dans cet automate.

Soit $t \in L(B)$ et r un *run* acceptant de B sur t. On va montrer qu'il existe un run r' de A_0 sur t tel que pour tout $d \in Pos(t), r'(d) = \langle r(d), \alpha\rangle$ où α est la tête d'ordre 1 de $t|_d$. Par construction, on sait que les règles utilisées sur les feuilles sont uniquement de la forme $\to \langle p, \alpha\rangle(\alpha)$, où $\to p(\alpha)$ est une transition de B. Aussi toutes les feuilles respectent bien la condition requise. On va montrer par induction sur la hauteur des sous-arbres que tous les autres sous-arbres de t aussi. Soit $d \in Pos(t)$

- si $t|_d$ est de la forme $a(x, y)$ alors, pour tous c_1, c_2, et soit $p_1(x), p_2(y) \to p(a(x, y))$ la règle utilisée dans r, il existe la règle $\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y) \to \langle p, a\rangle(a(x, y))$. Ainsi on garde le même état p et la tête d'ordre 1 est mise à jour. Par induction, la condition était déjà vérifiée pour x, y et leurs sous-arbres.
- si $t|_d$ est de la forme $f_2(x, y, z)$, on sait par induction qu'il existe un calcul de A_0 sur $t|_{d.1}$ tel que $r'(d.1) = \langle r(d.1), \alpha\rangle$ où α est la tête d'ordre 1 de $t|_{d.1}$. La transition utilisée dans r existe dans A_0 quelques soient les composantes Γ des états en x, y et z. La transition transmet juste la composante Γ de x à $f_2(x, y, z)$, c'est à dire ici α, or ces deux sous-arbres ont justement la même tête d'ordre 1. Comme par induction les sous-arbres $t|_{d.1}, t|_{d.2}$ et $t|_{d.3}$ respectaient la condition, $t|_d$ la respecte aussi.

Enfin, comme $r(\epsilon) \in P_f$, nécessairement $r'(\epsilon)$ est un état final de A_0. Donc $t \in L(A_0)$.

La réciproque est évidente : on considère un calcul r' de A_0 sur t. Comme chaque transition de A_0 correspond à une transition de B sans les composantes Γ, on remplace les transitions de manière unique, et on obtient un calcul réussi r de B sur t

$$\bigcup_i \boldsymbol{L(A_i) \subseteq Pre^*(L(B))}$$

Soit t un arbre reconnu par A_i. Si $i = 0$, on a montré précédemment que $L(A_0) = L(B)$, aussi, $t \in L(B)$. Si $i > 0$, soit r un *run* acceptant de A_i sur t. Si l'ensemble des transitions utilisées dans r est inclus dans l'ensemble des transitions de A_{i-1}, clairement $t \in L(A_{i-1})$, et par induction $t \in Pre^*(L(B))$. Sinon, il existe dans r des transitions de $A_i \setminus A_{i-1}$. Or toutes les transitions ajoutées ont comme conclusion un état de P_f, P_f^1 ou P_f^2. Procédons en fonction de la nature des transitions ajoutées pour passer de A_{i-1} à A_i.

- **$\mathbf{PUSH_a}$**
  Les transitions ajoutées sont de la forme

  $$\langle p_1', b\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \longrightarrow \langle q_{i'}, b\rangle(f_2(x, y, z))$$

  Aussi, comme $\langle q_{i'}, b\rangle$ et un état final et qu'il n'y a pas de transitions ayant un état final comme prémisse, il n'y a forcément qu'une de ces règles utilisée dans le *run* r, et il s'agit forcément de la dernière, appliquée à la racine de t. On sait donc que $t = f_2(t_1, t_2, t_3)$ avec $t_1 \in L(A_{i-1}, \langle p_1', b\rangle), t_2 \in L(A_{i-1}, \langle p_2, c_2\rangle)$ et $t_3 \in L(A_{i-1}, \langle p_3, c_3\rangle)$. D'autre part, si cette transition a été générée pour obtenir A_i depuis A_{i-1}, c'est qu'il y avait dans A_{i-1} le couple de règles suivant :

  $$\langle p_1, a\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \longrightarrow \langle q_k, a\rangle(f_2(x, y, z))$$

  et

  $$\langle p_1', b\rangle(x), \langle p_2', c_2'\rangle(y) \longrightarrow \langle p_1, a\rangle(a(x, y))$$

  Soit $t_4 \in L(A_{i-1}, \langle p_2', c_2'\rangle)$, comme $t_1 \in L(A_{i-1}, p_1')$, on sait que $a(t_1, t_4) \in L(A_{i-1}, \langle p_1, a\rangle)$ et donc que $t' = f_2(a(t_1, t_4), t_2, t_3) \in L(A_{i-1}, \langle q_k, a\rangle)$. Ainsi $(t', q_k) \in Pre^*(L(B))$. Or, clairement, $PUSH_a(t) = t'$, et comme la règle $\mathrm{PUSH_a} : q_i(f_2(b(x, y), l_1, r_1)), q_j(f_2(z, l_2, r_2)) \longrightarrow q_k(f_2(a(b(x, y), z), l_1, r_1))$ est dans A, on a $(t, q_i) \in Pre(t')$, donc $(t, q_i) \in Pre^*(L(B))$.

– **$\mathbf{PUSH_2}$**
Les transitions ajoutées sont de la forme

$$\langle p_1, b\rangle(x), \langle p'_2, c'_2\rangle(y), \langle p'_3, c'_3\rangle(z) \longrightarrow \langle q_i, b\rangle(f_2(x, y, z))$$

Comme précedemment, seule une de ces règles peut être utilisée et uniquement à la fin du *run*. Ainsi, $t = f_2(t_1, t_2, t_3)$ avec $t_1 \in L(A_{i-1}, \langle p_1, b\rangle), t_2 \in L(A_{i-1}, \langle p'_2, c'_2\rangle)$ et $t_3 \in L(A_{i-1}, \langle p'_3, c'_3\rangle)$. Comme cette règle a été générée pour A_i, les transitions

$$\langle p_1, b\rangle(x), \langle p_2, b\rangle(y), \langle p_3, c_3\rangle(z) \longrightarrow \langle q_k, b\rangle(f_2(x, y, z))$$

et

$$\langle p_1, b\rangle(x), \langle p'_2, c'_2\rangle(y), \langle p'_3, c'_3\rangle(y) \longrightarrow \langle p_2, b\rangle(f_2(x, y, z))$$

sont présentes dans A_{i-1}. Aussi, $f_2(t_1, t_2, t_3) \in L(A_{i-1}, \langle p_2, b\rangle)$ et en considérant $t_4 \in L(A_{i-1}, \langle p_3, c_3\rangle)$, on a $t' = f_2(t_1, f_2(t_1, t_2, t_3), t_4) \in L(A_{i-1}, \langle q_k, b\rangle)$. Donc $(t', q_k) \in Pre^*(L(B))$. Comme $PUSH_2(t) = t'$ et que la règle PUSH_2 :

$$q_i(f_2(b(x, y), l_1, r_1)), q_j(z) \longrightarrow q_k(f_2(b(x, y), f_2(b(x, y, l_1, r_1)), z))$$

est présente dans A, alors $(t, q_i) \in Pre(t')$, donc $(t, q_i) \in Pre^*(L(B))$.

– **$\mathbf{POP_1}$**
Plusieurs types de transitions peuvent être ajoutées. Considérons d'abord le cas où l'on ajoute une transition de la forme

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y) \longrightarrow \langle q^1_{j_1}, c\rangle(c(x, y))$$

Clairement, toute utilisation d'une telle transition de la première forme, implique l'utilisation par la suite d'une règle de la forme

$$\langle q^1_k, b_1\rangle(x), \langle p_2, b_2\rangle(y) \longrightarrow \langle q_j, a\rangle(a(x, y))$$

ajoutée lors de la construction de A_0i. En effet, les seules transitions ayant des états de P^1_f dans les prémisses, sont les règles ajoutées lors de la construction de A_0, lesquelles ont deux formes possibles. Or les transitions de l'autre forme (la première présentée) ont comme état sortant un état de P^1_f également, et comme aucun de ces états n'est final, on utilise forcément une transition ajoutée de la deuxième forme, telle que présentée ici.

Si les transitions de la première forme sont utilisées, elles le sont donc de la manière suivante : une transition telle que celle ajoutée lors du traitement d'une règle POP_1, suivie d'un nombre quelconques de transitions de A_0 ayant un état sortant de P_f^1 (qu'on appelera transitions intermédiaires), terminé par une transition de la deuxième forme. Montrons, par récurrence sur le nombre de transitions intermédiaires utilisées, qu'un arbre reconnu par A_i appartient bien à $Pre^*(L(B))$.

Tout d'abord, si aucune de ces transitions n'est utilisée, il y a donc une transition qui vient d'être ajoutée suivie d'une transition de "sortie". Soit A_i le premier automate contenant ces transitions (c'est à dire que la transition ajoutée est dans $A_i \setminus A_{i-1}$. L'arbre reconnu est de la forme $t = b(c(t_1, t_2), t_3)$ avec $t_1 \in L(A_{i-1}, \langle p_1, c_1\rangle)$, $t_2 \in L(A_{i-1}, \langle p_2, c_2\rangle)$ et $t_3 \in L(A_{i-1}, \langle p_2', c_2'\rangle)$. D'après la construction, on n'a pu ajouté ces nouvelles transitions dans A_i que si il existait dans A_{i-1} une transition de la forme :

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y) \longrightarrow \langle q_k, c\rangle(c(x, y))$$

ainsi qu'une règle POP_1 de la forme

$$q_{j_1}(b(x, y)), q_{j_2}(z) \longrightarrow q_k(x)$$

dans A.
Ainsi, l'arbre $t' = c(t_1, t_2)$ est reconnu par A_{i-1} dans l'état q_k. Par induction, $t' \in Pre^*(L(B))$, et clairement, $t \in Pre(t')$ car t' est obtenu en appliquant la règle POP_1 ci-dessus à t. Donc t appartient bien à $Pre^*(L(B))$.

Par hypothése de récurrence, supposons que, dans A_i, l'utilisation de $m - 1$ règles intermédiaires entre la règle ajoutée et celle de sortie conduit toujours à la reconnaissance d'un arbre de $Pre^*(L(B))$, et soit t un arbre reconnu qui utilise m règles intermédiaires. Il existe un état $q_{j_1}^1$ tel que l'arbre est de la forme $t = b(c(t_1, t_2), t_3)$ avec $t_1 \in L(A_i, \langle q_{j_1}^1, c\rangle)$. On suppose donc que le *run* réussi de A_i sur t utilise une transition ajoutée de la première forme suivie de $m - 1$ transitions intermédiaires (toutes ces transitions sont faites dans le sous-arbre t_1) puis d'une dernière transition intermédiaire (sur le sous-arbre $c(t_1, t_2)$) avant de finir avec une transition de sortie. La dernière transition intermédiaire est nécessairement de la forme

$$\langle q_{j_1}^1, b_1\rangle(x), \langle p_2, b_2\rangle(y) \longrightarrow \langle q_{j_2}^1, c\rangle(c(x, y))$$

avec $t_2 \in L(A_{i-1}, \langle p_2, b_2 \rangle)$. D'après la construction de A_0, cette règle intermédiaire a forcément été ajoutée en même temps que la règle de sortie suivante :

$$\langle q^1_{j_1}, b_1 \rangle(x), \langle p_2, b_2 \rangle(y) \longrightarrow \langle q_{j_1}, c \rangle(c(x, y))$$

Aussi, l'arbre $t' = c(t_1, t_2) \in L(A_i, \langle q_{j_1}, c)$ et est reconnu par A_i en utilisant seulement $m-1$ transitions intermédiaires. Par hypothése de récurrence, $t' \in Pre^*(L(B)$.
Or la règle de sortie utilisée pour le *run* réussi de t a été ajoutée car il y avait dans l'automate d'arbre à mémoire A une règle POP_1 de la forme :

$$q_{j_2}(b(x, y)), q_{j_3}(z) \longrightarrow q_{j_1}(x)$$

t étant bien le prédecesseur de t' par cette règle POP_1, on a ainsi montré que $t \in Pre^*(L(B))$.

Si la règle ajoutée est de la sorte suivante :

$$\langle p_1, c_1 \rangle(x), \langle p'_2, c'_2 \rangle(y) \longrightarrow \langle q^1_{k,p_2,p_3}, b \rangle(b(x, y))$$

le raisonnement reste similaire. La différence principale, c'est que l'on va travailler sur une (succession de) règle(s) POP_1 appliquées à un arbre d'ordre 2. Aussi, seule la tête de l'arbre est modifiée, le reste étant identique : c'est pourquoi c'est l'état de l'arbre au moment de la première règle POP_1 qu'il faut conserver. C'est ce que permettent de faire les éléments supplémentaires dans les états de la forme q^1_{k,p_2,p_3}. C'est aussi pourquoi les états de sortie ne sont générées qu'au fur et à mesure de la construction des A_i et non pas lors de la construction de A_0 : on ne doit pouvoir finir un *run* que si le reste de l'arbre est identique à ce qu'il était au début de la succession des règles POP_1. Donc mis à part les états supplémentaires que l'on sauvegarde pendant le *run*, on peut appliquer ici un raisonnement identique au précédent : en particulier, si un arbre t est reconnu en utilisant une transition de sortie après m transitions intermédiaires, il existera forcément une transition de sortie correspondant à l'application des $m-1$ premières transitions intermédiaires uniquement. Donc le passage de A_{i-1} à A_i en ajoutant une de ces règles conserve l'appartenance à $Pre^*(L(B))$ pour ce cas également.

– **POP_2**
Si on a ajouté une règle de la forme

$$\langle p_1, c_1 \rangle(x), \langle p_2, c_2 \rangle(y), \langle p_3, c_3 \rangle(z) \longrightarrow \langle q^2_{j_1}, c_1 \rangle(f_2(x, y, z))$$

pour construire A_i à partir de A_{i-1} on a aussi ajouté l'ensemble des règles suivantes

$$\langle p'_1, c'_1\rangle(x), \langle q^2_{j_1}, c_1\rangle(y), \langle p'_3, c'_3\rangle(z) \longrightarrow \langle q_{j_1}, c'_1\rangle(f_2(x,y,z))$$

pour tous $p'_1, p'_3 \in P$ et tous $c'_1, c'_3 \in$ tels que $L(A_i, \langle p'_1, c'_1\rangle) \neq \emptyset$ et $L(A_i, \langle p'_3, c'_3\rangle) \neq \emptyset$. L'utilisation d'une règle de la première forme dans un *run* conduit nécessairement à l'utilisation ultérieure d'une règle de la seconde forme. Soit un arbre t reconnu par A_i et dont le *run* acceptant utilise ces deux formes de règles à la suite. Cet arbre est donc de la forme $f_2(t'_1, f_2(t_1,t_2,t_3), t'_3)$ avec $t_1 \in \langle p_1, c_1\rangle, t_2 \in \langle p_2, c_2\rangle, t_3 \in \langle p_3, c_3\rangle, t'_1 \in \langle p'_1, c'_1\rangle$ et $t'_3 \in \langle p'_3, c'_3\rangle$. Ces nouvelles transitions ont été ajoutées dans A_i uniquement si il y avait dans A une règle POP_2 de la forme

$$q_{j_1}(f_2(b(x,y), l_1, r_1)), q_{j_2}(f_2(z, l_2, r_2)) \longrightarrow q_k(l_1)$$

et dans A_{i-1} une transition de la forme

$$\langle p_1, c_1\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \longrightarrow \langle q_k, c_1\rangle(f_2(x,y,z))$$

Ainsi, l'arbre $t' = f_2(t_1,t_2,t_3)$ est reconnu par A_{i-1} et donc par hypothèse d'induction, $t' \in Pre^*(L(B))$. On remarque que t est le prédécesseur de t' par la règle POP_2 ci-dessus, et ainsi $t \in Pre^*(L(B))$. Ici aussi, le calcul peut passer par un nombre quelconques de transitions intermédiaires comme dans le cas des règles POP_1 : le raisonnement est similaire, on utilise des états distincts (de la forme q^2_k) et les transitions ajoutées permettent de conserver le reste de l'état de l'arbre en train d'être traité comme vu auparavant. La preuve se généralise donc facilement à ce cas.

$\boldsymbol{Pre^*(L(B)) \subseteq \bigcup_j L(A_j)}$

Pour tout $t \in Pre^*(L(B))$, et n le plus petit entier tel que $t \in Pre^n(L(B))$. On va prouver par récurrence sur n que pour tout $t \in Pre^n(L(B))$ il existe i tel que t est reconnu par A_i.

Si $n = 0$, on a montré précedemment que $L(A_0) = L(B)$, donc $t \in L(A_0)$. Si $n > 0$, on sait qu'il existe un arbre $t' \in Pre^{n-1}(L(B))$ et une transition Op de l'automate d'arbre à mémoire A tels que $t' \xrightarrow{Op} t$. Par récurrence, il existe i tel que t' est reconnu par A_i. Soit j le plus petit entier tel que $j > i$ et tel qu'on passe de A_{j-1} à A_j en considérant la règle Op. On va montrer que

t est reconnu par A_j : t et t' sont très proches, et on peut donc appliquer les règles d'A_i pour reconnaître t sauf aux parties de l'arbre qui ont été modifiées par l'application de la règle Op. Ces modifications sont toujours proches de la racine de l'arbre, mais dépendent de la nature de la règle Op, c'est pourquoi il faut procéder au cas par cas. Le raisonnement à appliquer est complétement symétrique à celui déjà suivi dans le premier sens de la preuve. Dans le premier sens, on considérait un *run* acceptant de A_i sur t, et on montrait que les nouvelles transitions qui étaient utilisées impliquaient la préexistence de transitions dans A_{i-1} qui permettaient de reconnaître t'. Ici, à l'inverse, on sait que t' est reconnu par A_{j-1}, et on va montrer que les règles utilisées lors du *run* acceptant de A_{j-1} vont générer, lors du traitement de l'opération pour produire A_j, l'apparition de nouvelles règles qui vont permettre de reconnaître t. Il suffit donc de reprendre le traitement au cas par cas précédent à l'envers. Pour l'exemple, nous détaillons ici le cas des opérations $PUSH_a$.

On suppose que t est le prédécesseur de t' par la règle suivante :

$$q_{j_1}(f_2(b(x,y),l_1,r_1)), q_{j_2}(f_2(z,l_2,r_2)) \to q_k(f_2(a(b(x,y),z),l_1,r_1))$$

on a donc $t = f_2(b(t_1,t_2),t'_2,t'_3)$ et $t' = f_2(a(b(t_1,t_2),t_3),t'_2,t'_3)$. t' est reconnu par A_i donc par A_{j-1} puisqu'on a uniquement ajouté des règles lors de la construction des $A_l, i < l \leq j-1$. En particulier il existe dans A_i une transition de la forme

$$\langle p'_1, b\rangle(x), \langle p'_2, c'_2\rangle(y) \to \langle p_1, a\rangle(a(x,y))$$

avec $b(t_1,t_2) \in L(A_i, \langle p'_1, b\rangle)$ et $t_3 \in L(A_i, \langle p'_2, c'_2\rangle)$ et une transition de la forme

$$\langle p_1, a\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \to \langle q_k, a\rangle(f_2(x,y,z))$$

Comme ces deux règles sont dans A_{j-1} le traitement de l'opération $PUSH_1$ ci-dessus a conduit à l'ajout de la règle

$$\langle p'_1, b\rangle(x), \langle p_2, c_2\rangle(y), \langle p_3, c_3\rangle(z) \longrightarrow \langle q_i, b\rangle(f_2(x,y,z))$$

dans A_j. Ainsi t est reconnu par A_j.

5 Conclusion

On a ainsi réussi à généraliser les automates à pile d'ordre supérieur aux arbres, et à généraliser le résultat du Pre^* de Hague et Ong. Ainsi, on

pourrait étendre le champ d'application de ce résultat. Les arbres permettraient par exemple de simuler des processus s'exécutant en parallèle, et nos résultats d'accessibilité seraient un outil pour vérifier des propriétés assez simples. Mais surtout, cette généralisation permet d'utiliser un cadre plus large et plus simple, d'une part en permettant l'utilisation d'un automate classique (un simple automate d'arbre), d'autre part en travaillant sur un cadre plus puissant qu'avant (les arbres plutôt que les mots). Nous avons également tenté de poursuivre nos investigations sur des résultats d'accessibilité en avant, espérant pouvoir résoudre pour partie le $Post$ et le $Post^*$. Cependant les résultats de Bouajani et Meyer [BM04] sur la difficulté dans le cas des mots laissent peu d'espoir à une solution, même partielle, dans ce cas. Toutefois, dans [Car05] Arnaud Carayol montre la régularité d'un sous-ensemble du $Post^*$. En effet, plutôt que de considèrer l'ensemble des configurations accessibles depuis un ensemble régulier de configurations à l'aide une succession quelconque d'un nombre fini de règles de transitions, il ne considère qu'une succession régulière de ces règles. La succession des règles de transitions appliquées est ainsi elle-même reconnaissable par un automate. Sous ces conditions, et en utilisant des formes normales spécifiques pour la représentation des piles d'ordre supérieures, il montre que le $Post^*$ d'un ensemble régulier de configurations est lui aussi régulier. Il serait intéressant qu'une telle notion, ou une plus restrictive mais de même inspiration, puisse être étendue au cas des arbres et apporter ainsi encore plus de possibilités de preuves de propriétés.

Références

[BM04] A. Bouajani and A. Meyer. Symbolic reachability analysis of higher-order context-free processes. *FSTTCS 2004*, pages 135–147, 2004.

[Car05] Arnaud Carayol. Regular sets of higher-order pushdown stacks. In *MFCS*, pages 168–179, 2005.

[CCM01] Hubert Comon, Véronique Cortier, and John Mitchell. Tree automata with one memory, set constraints, and ping-pong protocols. In *ICALP '01 : Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 682–693, London, UK, 2001. Springer-Verlag.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : http ://www.grappa.univ-lille3.fr/tata, 1997.

[HO07] Matthew Hague and C.-H. Luke Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. In *FoSSaCS*, pages 213–227, 2007.