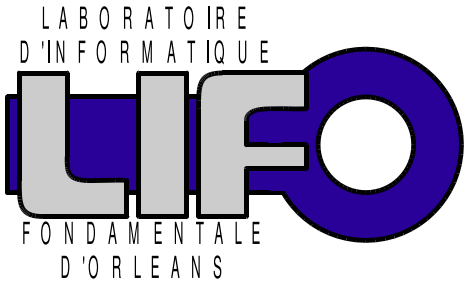


4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Comparaison des langages d'arbres dans le cadre de la substitution de services web

Joshua Amavi
LIFO, Université d'Orléans

Rapport n^o **RR-2010-13**

Comparaison des langages d'arbres dans le cadre de la substitution de services web

Rapport de Stage

Joshua AMAVI

Master 2 IRAD option Recherche

Encadrants : Mirian Halfeld-Ferrari & Pierre Réty

Laboratoire d'Informatique Fondamentale d'Orléans

Département Informatique - Université d'Orléans
Année Universitaire 2009 - 2010

1^{er} Septembre 2010

Table des matières

Remerciements

Dans le cadre de mon stage que j'ai effectué dans l'Equipe PRV du Laboratoire d'Informatique Fondamentale d'Orléans :

Je tiens à remercier mes encadrants, Mirian Halfeld-Ferrari et Pierre Réty, pour tout le temps qu'ils m'ont consacré malgré leur emploi du temps chargé, pour leur patience et la confiance qu'ils m'ont accordé.

Et aussi je remercie ma famille pour leur soutien qu'ils m'ont apporté tout au long du déroulement de mon stage.

Je remercie pour finir l'ensemble de mes collègues master recherche qui m'ont permis de passer une bonne année.

Résumé. Dans ce rapport, nous proposons un algorithme qui étend l'inclusion relâchée sur les arbres à l'inclusion relâchée sur les langages d'arbres. Notre algorithme est utile dans le contexte de la substitution de service web, et plus précisément pour la comparaison de schéma XML. L'algorithme repose sur les langages synchronisés de couples d'arbres. L'algorithme a été prouvé correct et implémenté en Prolog.

Mots clés : Langage d'arbres, inclusion relâchée de schéma XML, service web, langage synchronisé de couples d'arbres, expression régulière.

Abstract. In this paper, we propose algorithm that extend tree inclusion to tree language inclusion. Our algorithm is useful in the context of web service substitution, and precisely in XML schema comparison. The algorithm used the synchronized-contextFree tree-tuple languages. The algorithm is proven correct and an implementation has done.

Keys words : Tree language, inclusion of XML schema, web service, synchronized-contextFree tree-tuple languages, regular expression.

Introduction

Avec l'interconnexion des ordinateurs en réseau et en particulier à travers internet, il devient possible de faire fonctionner des applications sur des machines distantes. Les services web représentent un mécanisme de communication entre applications distantes à travers le réseau internet indépendant de tout langage de programmation et de toute plate-forme d'exécution. Les services web emploient une syntaxe basée sur la notation XML [1] pour décrire les appels de fonctions distantes et les données échangées. Le développement et l'adoption des technologies associées aux services web permettent aux entreprises d'implanter de nouvelles applications en composant des services existants. Ainsi il est possible de créer un service à partir de plusieurs services. Que se passe t'il si l'un des services S_1 de la composition tombe en panne ? Pour que la nouvelle application remarche à nouveau il faudra trouver un nouveau service qui fait la même chose que S_1 pour le remplacer. Pour cela nous devons être capable de comparer deux services web pour savoir s'ils font la même chose ou si l'un est contenu dans l'autre. Pour comparer deux services web, nous comparons les schémas DTD [2] qui décrivent les fichiers XML retournés par ces services.

Dans ce stage nous étendons les travaux sur l'inclusion relâchée d'arbres [10, 8] aux langages d'arbres. Ainsi nous proposons un algorithme qui permet de comparer deux langages d'abres. Nous utilisons les travaux sur les langages synchronisés de couple d'arbres [2] pour pouvoir comparer les langages d'arbres. Des implémentations de l'algorithme que nous proposons ont été faites en Prolog.

Le reste du rapport est organisé comme suit : dans le Chapitre 1 nous introduisons les notions préliminaires nécessaires pour la compréhension des travaux ; le Chapitre 2 présente l'objectif du stage et les différentes étapes de l'algorithme que nous proposons ; les Chapitres 3 et 4 introduisent les langages synchronisés de couple d'arbres et détaillent les étapes de l'algorithme. Enfin le Chapitre 5 discute sur les travaux liés.

1. XML pour eXtensible Markup Language
2. DTD pour Document Type Descriptors

Chapitre 1

Préliminaires

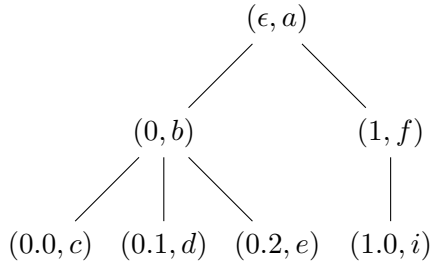
Ce chapitre est consacré à l'introduction du matériel qui sera nécessaire à la bonne compréhension et la cohérence du rapport. Il y sera donc présenté les arbres et les relations sur les arbres dont nous avons besoin.

1.1 Définitions

1.1.1 Arbre d'arité non borné

Definition 1. *[12] Soit Σ un ensemble d'étiquettes. Soit U l'ensemble des chaînes de caractères composées d'entiers positifs ou nuls (ϵ désigne la chaîne vide), et considérons un ensemble $Pos(t) \subseteq U$. On suppose que $Pos(t)$ est clos par préfixe (c'est à dire si $u \leq v$ et $v \in Pos(t)$ cela implique que $u \in Pos(t)$) et vérifie $(j \in \mathbb{N}, uj \in Pos(t), 0 \leq i \leq j) \Rightarrow ui \in Pos(t)$.*
Un arbre enraciné, ordonné et non borné t est une application $t : Pos(t) \to \Sigma$. Les élements de $Pos(t)$ sont appelés positions de t.
Nous notons $t(v) = a$ pour indiquer que l'étiquette de v ($v \in Pos(t)$) est a.

Example 1. Voici un exemple d'arbre enraciné en a et ordonné :



Légende : (x, y) correspond à (position,étiquette)

FIGURE 1.1 – Exemple d'arbre d'arité non borné a(b(c,d,e),f(i))

Definition 2. *Soit Σ l'ensemble des étiquettes. Nous définissons T_Σ comme l'ensemble de tous les arbres d'arité non borné sur Σ.*

Definition 3. *Soient p et $q \in Pos(t)$, p est ancêtre de q ssi il existe une séquence d'entiers r telle que $p.r = q$.*
Nous noterons $p < q$ ssi p est ancêtre de q.

Definition 4. *Soient p et $q \in Pos(t)$, p est à gauche de q ssi il existe des séquences d'entiers u, v, w et deux entiers i, j telle que $p = u.i.v$, $q = u.j.w$, et $i < j$.*
Nous noterons $p \prec q$ ssi p est à gauche de q.

Definition 5. *(Inclusion relâché) Considérons deux arbres P, T. P est inclus relâché dans T (ou est un sous-arbre relâché de T) ssi il existe une application f de $Pos(P)$ vers $Pos(T)$ telle que :*

1. $\forall v \in Pos(P) : P(v) = T(f(v))$,
2. $\forall v_1, v_2 \in Pos(P) : v_1 < v_2 \Longrightarrow f(v_1) < f(v_2)$,
3. $\forall v_1, v_2 \in Pos(P) : v_1 \prec v_2 \Longrightarrow f(v_1) \prec f(v_2)$.

Nous noterons $P \lhd T$ ssi P est inclus relâché dans T.

Les arbres suivants sont inclus relâchés dans l'arbre de la Figure 1 :

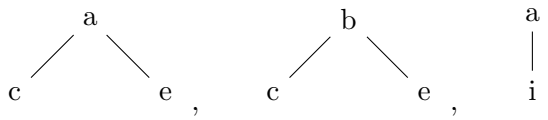


FIGURE 1.2 – Exemple d'arbres inclus relâchés dans l'arbre de la Figure 1

Les arbres suivants ne sont pas inclus relâchés dans l'arbre de la Figure 1. Le premier car le nœud b n'a pas de fils i et le second parce que le nœud c n'est pas à droite de i dans la Figure 1.

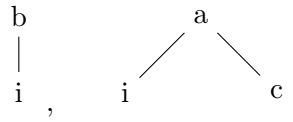


FIGURE 1.3 – Exemple d'arbres non inclus relâchés dans l'arbre de la Figure 1

1.1.2 Arbre binaire

Le codage que nous utilisons pour transformer un arbre d'arité non borné en arbre binaire est le codage first-child next-sibling.

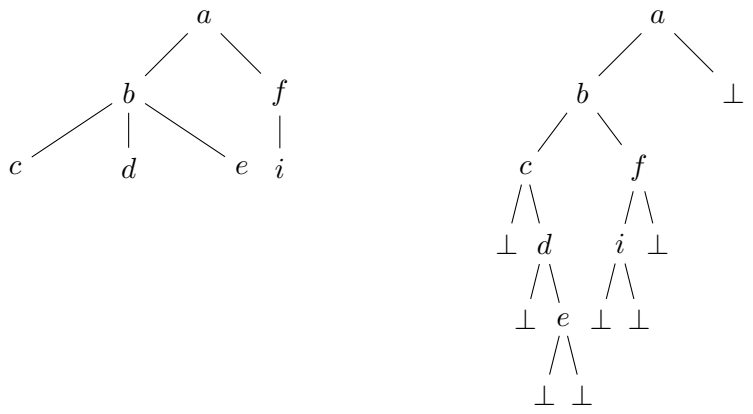


FIGURE 1.4 – L'arbre $a(b(c,d,e),f(i))$ et son codage first-child next-sibling $a(b(c(\bot,d(\bot,e(\bot,\bot))),f(i(\bot,\bot),\bot)),\bot)$

Dans l'arbre binaire ci-dessus, le fils gauche de b est le premier fils de b dans l'arbre d'arité non borné. Son fils droit est son frère immédiat dans l'arbre d'arité non borné.

Soit t un arbre d'arité non borné. Nous notons par $fcns(t)$ le codage en first-child next-sibling de l'arbre t. Si t est défini sur Σ alors $fcns(t)$ est défini sur $\Sigma' = \Sigma \cup \{\bot\}$.
Dans la suite nous reformulons les notions de "ancêtre", "à gauche de" et "inclusion relâchée" en considérant le codage first-child next-sibling d'un arbre t.

Definition 6. *Soient p et $q \in Pos(fcns(t))$, $p <_b q$ ssi il existe une séquence d'entiers r telle que $p.0.r = q$.*

Definition 7. *Soient p et $q \in Pos(fcns(t))$, $p \prec_b q$ ssi il existe des séquences d'entiers u, v, w telle que $p = u.0.v$ ou $p = u$, et $q = u.1.w$.*

Definition 8. *(Inclusion relâché pour les arbres binaires) Soient P et T deux arbres binaires $P \lhd_b T$ ssi il existe une application f de $Pos(P)$ vers $Pos(T)$ telle que :*

1. $\forall v \in Pos(P) : P(v) = T(f(v))$,
2. $\forall v_1, v_2 \in Pos(P) : v_1 <_b v_2 \Longrightarrow f(v_1) <_b f(v_2)$,
3. $\forall v_1, v_2 \in Pos(P) : v_1 \prec_b v_2 \Longrightarrow f(v_1) \prec_b f(v_2)$.

Si de plus $f(\epsilon) = \epsilon$ on dira qu'il s'agit d'une inclusion relachée avec racines communes, et on notera $P \lhd_b^r T$.

Remarque 9. Soient t, t' deux arbres d'arité non bornée. $t \lhd t' \Longleftrightarrow fcns(t) \lhd_b fcns(t')$.

Chapitre 2

Objectif et méthode utilisée

Nous présentons dans ce chapitre le problème d'inclusion relâché de schéma et les différentes parties de la solution que nous proposons pour le résoudre.
Dans la suite, S_1 désigne à la fois le service web S_1, et le langage d'arbres accepté par S_1 (c'est à dire le schéma de S_1). Il en est de même pour S_1'. Le symbole $\bowtie$ désigne la jointure naturelle.

2.1 Objectif du stage

Pour savoir si le service S_1' peut remplacer un autre service S_1, nous avions besoin de décider que $S_1 \lhd S_1'$. Si on arrive à décider que $S_1 \lhd S_1'$, cela veut dire que S_1' contient toutes les informations de S_1 et contient d'autres informations en plus. Puisque S_1' contient toutes les informations de S_1, alors il peut remplacer S_1 jusqu'à ce que S_1 redevienne opérationnel. Pour que le remplacement soit effectif il faudra aussi mettre un filtre entre S_1' et les autres services avec lesquels S_1 communiquait.

Un document XML est vu comme un arbre d'arité non borné. Les DTD qui stucturent les documents XML sont donc des langages d'arbres. Considérons deux langages d'arbres S_1 et S_1' définit sur l'ensemble des étiquettes Σ. Ce stage propose un algorithme pour décider si $S_1 \lhd S_1'$.
Formellement voici la définion de l'inclusion relâchée sur les langages d'arbres.

Definition 10. *Soient S et S' deux langages d'arbres. Nous avons $S \lhd S'$ (S inclus relâché dans S') ssi $\forall\ t \in S, \exists\ t' \in S'$ tel que $t \lhd t'$.*

2.2 Méthode utilisée

Pour décider que $S_1 \lhd S'_1$, nous procédons comme suit :

1. D'abord nous tranformons les grammaires d'arbres d'arité non borné S_1 et S'_1 en grammaires d'arbres binaires pour obtenir $bin(S_1)$ et $bin(S'_1)$. Le codage utilisé pour cette transformation est le codage first-child next-sibling.
2. Ensuite nous exprimons par une grammaire non régulière, une relation R qui définit l'ensemble de tous les couples (t, t') sur T^2_Σ tels que $t \lhd_b t'$. Cet ensemble est appelé relation R à cause de la relation d'inclusion relâchée qu'il y a entre t et t'.
3. Maintenant que nous avons la relation R, nous allons calculer la projection régulière suivante : $\Pi_1(R \bowtie bin(S'_1))$. L'ensemble $(R \bowtie bin(S'_1))$ nous permet d'avoir tous les couples $(t, t') \in R$ tels que $t' \in bin(S'_1)$. En calculant $\Pi_1(R \bowtie bin(S'_1))$ c-a-d la projection sur la première composante des couples de $(R \bowtie bin(S'_1))$ ou plus précisément en gardant les arbres t des couples de $(R \bowtie bin(S'_1))$, on obtient automatiquement tous les arbres t qui sont inclus relâchés dans les arbres t' de $bin(S'_1)$.
4. Si l'on a $bin(S_1) \subseteq \Pi_1(R \bowtie bin(S'_1))$ alors on a bien décidé que $S_1 \lhd S'_1$.

La grammaire de la relation R est exprimée par un langage synchronisé de couples d'arbres d'arité fixe [2]. Nous utilisons ce type de langage pour profiter de sa puissance car il exprime des grammaires qui sont dans une classe qui contient la classe des grammaires algébriques. Ces langages synchronisés de couples possèdent des propriétés intéressantes comme la jointure, le test du vide d'un langage, le test de l'appartenance d'un élément à un langage, qui nous seront utiles dans nos travaux. Nous transformons les grammaires d'arbres d'arité non borné en grammaires d'arbres binaire car les langages synchronisés ne sont définis que pour des arbres d'arité fixe. Le problème d'inclusion classique (qu'on utilise au point 4) entre deux langages réguliers A et B est connu. Décider que $A \subseteq B$ revient à vérifier que $A \cap \overline{B} = \emptyset$.

Chapitre 3

Le calcul de la relation R et la projection régulière

Dans ce chapitre nous allons introduire les langages synchronisés de couples d'arbres. Ensuite nous expliquerons comment la relation R peut être obtenue. Nous terminerons par la projection régulière.

3.1 Les langages synchronisés de couples d'arbres

Les langages synchronisés de couples d'arbres contiennent la classe des langages algébriques. Ils ont donc un pouvoir d'expressivité plus fort que les langages algébriques. Un programme de langage synchronisé de couples d'arbres est un programme logique dont les prédicats ont des arguments de deux sortes : les arguments de sortie et les arguments d'entrée. Les arguments de sortie sont couverts par un chapeau.
Notation : Soit B une liste d'atomes. $Input(B)$ désigne le tuple composé des arguments d'entrées de B, et $Output(B)$ désigne le tuple composé des arguments de sortie de B.
$VarIn(B)$ désigne l'ensemble des variables qui apparaissent dans $Input(B)$, et $VarOut(B)$ désigne l'ensemble des variables qui apparaissent dans $Output(B)$.

Definition 11. *Soit $B = A_1, \ldots, A_n$ une liste d'atomes. On dit que $A_j > A_k$ si $\exists\ y \in VarIn(A_j) \cap VarOut(A_k)$. Plus simplement une entrée de A_j dépend d'une sortie de A_k.*
On dit que B comporte un cycle si on a $A_j >^+ A_j$ pour un atome A_j. ($>^+$ est la fermeture transitive de $>$).
On dit qu'une clause $H \leftarrow B$ a un cycle si B a un cycle.

Example 2. Pour $B = P(\widehat{x}, \widehat{y}, z), Q(\widehat{z}, t)$, $Output(B) = (x, y, z)$ et $Input(B) = (z, t)$.

Example 3. $Q(\widehat{x}, s(y)), R(\widehat{y}, s(x))$ a un cycle car $Q > R > Q$.

Definition 12. *Un programme de langage synchronisé de couples d'arbres Prog est un programme logique dont les clauses $H \leftarrow B$ satisfont :*
- *$Input(H).Output(B)$ (. est la concaténation de tuple) est un tuple linéaire de variables (c-a-d ne comporte pas de doublons)*
- *et B ne comporte pas de cycle.*

Etant donné un prédicat P sans arguments d'entrés, le langage de tuples d'arbres généré par P est $L(P) = \{\vec{t} \in (T_\Sigma)^{|Output(P)|} \mid P(\vec{t}) \in Mod(Prog)\}$, où $Mod(Prog)$ est le plus petit modèle d'Herbrand de $Prog$.

Example 4. Soit le programme suivant :

$Prog = \{S(\overset{\frown}{\underset{x\ \ y}{c}}) \overset{1}{\leftarrow} P(\widehat{x}, \widehat{y}, a, b).\ \ P(\overset{\frown}{\underset{x}{f}}, \overset{\frown}{\underset{y}{g}}, x', y') \overset{2}{\leftarrow} P(\widehat{x}, \widehat{y}, \underset{x'}{h}, \underset{y'}{i}).\ \ P(\widehat{x}, \widehat{y}, x, y) \overset{3}{\leftarrow}\}$

En prenant :

- la clause 1, $Input(H).Output(B) = (x, y)$ est linéaire et ne contient que des variables
- la clause 2, $Input(H).Output(B) = (x', y', x, y)$ est linéaire et ne contient que des variables
- la clause 3, $Input(H).Output(B) = (x, y)$ est linéaire et ne contient que des variables

Et aussi aucune des clauses n'a de cycle. Ce programme est bien un programme de langage synchronisé de couples d'arbres.

Le langage généré par S est $L(S) = \{ \underset{\begin{array}{cc} f^n & g^n \\ | & | \\ h^n & i^n \\ | & | \\ a & b \end{array}}{c} \mid n \in \mathbb{N}\}$. Nous pouvons remarquer que ce langage n'est pas un langage algébrique.

Maintenant nous allons définir la réécriture close sur les programmes logiques. La réécriture close nous sera utile pour démontrer les lemmes et théorèmes de la section suivante.

Definition 13. *Soit un programme logique $Prog$ et une liste d'atomes L, nous avons :*

- *L se dérive en une liste d'atomes L' par une étape de résolution s'il existe une clause $H \leftarrow B$ dans $Prog$ et un atome $A \in L$ tel que A et H sont unifiables par le plus général unificateur σ (donc $A\sigma = H\sigma$) et $L' = (L\sigma)[A\sigma \leftarrow B\sigma]$. On la note $L \rightsquigarrow^\sigma L'$.*
- *L se réécrit en L' s'il existe une clause $H \leftarrow B$ dans $Prog$, un atome $A \in L$, et une substitution σ tel que $A = H\sigma$ (A n'est pas instantié par σ), $B\sigma$ est clos, et $L' = (L[A \leftarrow B\sigma])$. On la note $L \twoheadrightarrow^\sigma L'$.*
  Remarquons que si L est clos alors L' aussi est clos.

Nous considérons la fermeture réflexive et transitive $\rightsquigarrow^*$ de $\rightsquigarrow$ et $\twoheadrightarrow^*$ de $\twoheadrightarrow$.

Theorem 1. *Soit A un atome clos. $A \in Mod(Prog)$ ssi $A \twoheadrightarrow^*_{Prog} \emptyset$, avec $Mod(Prog)$ le plus petit modèle d'Herbrand de $Prog$ et $\emptyset$ est une liste vide d'atomes.*

Example 5. En reprenant le programme $Prog$ de l'exemple 4, regardons comment dériver un arbre $t = c(f(h(a)), g(i(b))) \in L(S)$:

$S(c(f(h(a)), g(i(b)))) \overset{1}{\twoheadrightarrow} P(f(h(a)), g(i(b)), a, b) \overset{2}{\twoheadrightarrow} P(h(a), i(b), h(a), i(b)) \overset{3}{\twoheadrightarrow} \emptyset.$

Dans [2], nous pouvons trouver l'algorithme qui permet de faire l'intersection (ou jointure) entre un programme de langage synchronisé de couples d'arbres et un programme régulier d'arbre, l'algorithme pour le test du vide d'un langage et l'algorithme du test d'appartenance d'un élément à un langage. Par contre il reste la projection régulière que j'ai eu à étudier durant mon stage. Cette projection consiste, à partir d'un programme de langage synchronisé

de couples d'arbres, à trouver le programme régulier d'arbres qui correspond à la projection sur un argument, à condition que cette projection soit bien un langage régulier. Elle sera traitée dans la dernière section de ce chapitre.

3.2 La Relation R

Pour répondre à notre question de savoir si un langage est inclus relâché dans un autre, nous allons exprimer une relation R par un langage synchronisé de couples d'arbres d'arité fixe [2]. Dans le but d'écrire un article avec ces travaux, les preuves des lemmes et propriétés sont rédigés en anglais. Les preuves des lemmes sont en Annexes [1].

Definition 14. *La relation R est définie comme suit :* $R = \{(t, t') \mid (t, t') \in T^2_\Sigma, t \lhd_b t'\}$.

Cette relation définie l'ensemble de tous les couples d'arbres (t, t') sur T^2_Σ tels que $t \lhd_b t'$ avec t et t' ayant la même racine.

Algorithme 1. Le programme logique qui permet d'exprimer la relation R par un langage synchronisé de couples d'arbres binaires est le suivant :

$Prog_1 = \{(1)\ Q_0(\widehat{\begin{smallmatrix}\alpha\\ /\ \backslash\\ X\ \ \bot\end{smallmatrix}}, \widehat{\begin{smallmatrix}\alpha\\ /\ \backslash\\ X'\ \bot\end{smallmatrix}}) \leftarrow Q(\widehat{X}, \widehat{X'}, \bot).\mid \alpha \in \Sigma\}$

$Prog_2 = \{(2)\ Q(\widehat{X}, \widehat{Y}, X) \leftarrow A(\widehat{Y}).$

$(3)\ Q(\widehat{\begin{smallmatrix}\alpha\\ /\ \backslash\\ X\ \ Y\end{smallmatrix}}, \widehat{\begin{smallmatrix}\alpha\\ /\ \backslash\\ X'\ Y'\end{smallmatrix}}, R) \leftarrow Q(\widehat{X}, \widehat{X'}, \bot), Q(\widehat{Y}, \widehat{Y'}, R).$

$(4)\ Q(\widehat{X}, \widehat{\begin{smallmatrix}\alpha\\ /\ \backslash\\ Y\ \ Z\end{smallmatrix}}, R) \leftarrow Q(\widehat{X}, \widehat{Y}, R'), Q(\widehat{R'}, \widehat{Z}, R).\mid \alpha \in \Sigma\}$

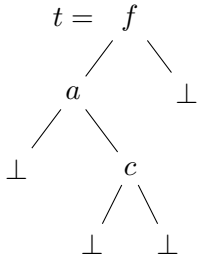
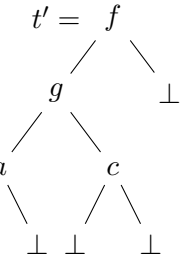
$Prog_3 = \{(5)\ A(\widehat{\bot}) \leftarrow .\quad (6)\ A(\widehat{\begin{smallmatrix}\alpha\\ /\ \backslash\\ X\ \ Y\end{smallmatrix}}) \leftarrow A(\widehat{X}), A(\widehat{Y}).\mid \alpha \in \Sigma\}$

$Prog = Prog_1 \cup Prog_2 \cup Prog_3$

Dans cette section, nous allons voir que le langage généré par A est $L(A) = T_\Sigma$, et aussi que le langage généré par Q_0 est $L(Q_0) = R$.

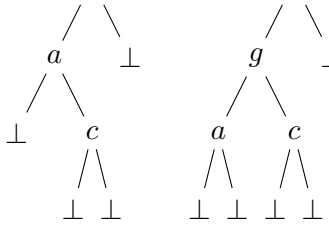
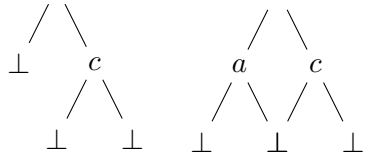
Les prédicats du programme logique $Prog$ ci-dessus permettent d'analyser les arbres de façon descendante en partant de Q_0. Pour dire que deux arbres t et t' sont tels que $t \lhd_b t'$, on analyse $Q_0(t, t')$. Voyons d'abord un exemple.

Example 6. Soit $\Sigma = \{f, g, a, c, \bot\}$.

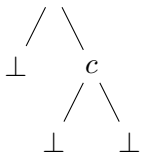
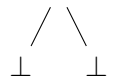
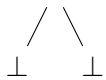
Considérons $t =$  et $t' =$  $\in T_\Sigma$ tel que $t \lhd_b t'$.

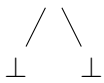
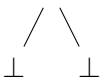
1. Les preuves des lemmes sont en Annexes mais font partie intégrante du rapport.

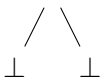
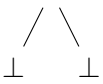
A chaque étape, le prédicat souligné est celui qui est dérivé.
Regardons comment dériver $Q_0(t, t')$:

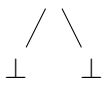
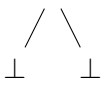
$\underline{Q_0}($  $) \xrightarrow{1}\!\!\!\!\twoheadrightarrow \underline{Q}($  $, \bot)$

en prenant $t'' \in T_\Sigma$ (la valeur de t'' sera précisée plus loin)

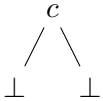
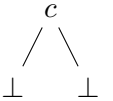
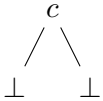
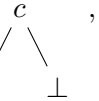
$\xrightarrow{4}\!\!\!\!\twoheadrightarrow \underline{Q}($  $,$  $, t'') \; , \; Q(t'',$  $, \bot)$

$\xrightarrow{3}\!\!\!\!\twoheadrightarrow \underline{Q}(\bot, \bot, \bot) \; , \; Q($  $, \bot, t'') \; , \; Q(t'',$  $, \bot)$

$\xrightarrow{2}\!\!\!\!\twoheadrightarrow \underline{A}(\bot) \; , \; Q($  $, \bot, t'') \; , \; Q(t'',$  $, \bot)$

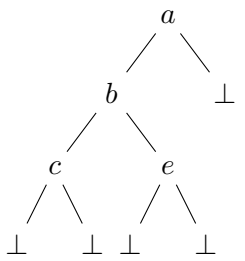
$\xrightarrow{5}\!\!\!\!\twoheadrightarrow \underline{Q}($  $, \bot, t'') \; , \; Q(t'',$  $, \bot)$

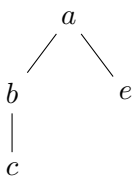
en appliquant la clause 2 avec $t'' = c(\bot, \bot)$

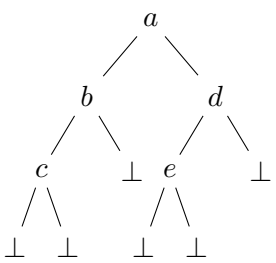
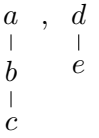
$\xrightarrow{2}\!\!\!\!\twoheadrightarrow \underline{A}(\bot) \; , \; Q($  $,$  $, \bot) \xrightarrow{5}\!\!\!\!\twoheadrightarrow \underline{Q}($  $,$  $, \bot)$

$\xrightarrow{3}\!\!\!\!\twoheadrightarrow \underline{Q}(\bot, \bot, \bot), Q(\bot, \bot, \bot) \xrightarrow{2}\!\!\!\!\twoheadrightarrow \underline{A}(\bot), Q(\bot, \bot, \bot) \xrightarrow{5}\!\!\!\!\twoheadrightarrow \underline{Q}(\bot, \bot, \bot) \xrightarrow{2}\!\!\!\!\twoheadrightarrow \underline{A}(\bot) \xrightarrow{5}\!\!\!\!\twoheadrightarrow \emptyset.$

□

Regardons maintenant plus en détail l'algorithme 1. Dans $Prog_1$, le prédicat Q_0 permet de vérifier que les traduction des deux arbres en arbre d'arité non borné sont bien des arbres et non des forêts. Par exemple l'arbre binaire  est la traduc-

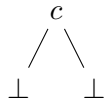
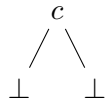
tion first-child next-sibling de  qui est bien un arbre alors que l'arbre binaire

 est la traduction first-child next-sibling de () qui est une forêt. La racine d'un arbre binaire bien formé (c-a-d que sa traduction en arbre d'arité non borné n'est pas une forêt) a $\perp$ comme fils droit.

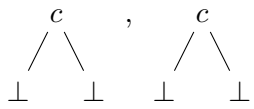
La règle (1) $Q_0(\widehat{\alpha(X,\perp)}, \widehat{\alpha(X',\perp)}) \leftarrow Q(\widehat{X},\widehat{X'},\perp)$ nous dit que si $\alpha(X,\perp) \lhd_b \alpha(X',\perp)$ alors $X \lhd_b X'$ (le sous-arbre gauche de t est inclus relâché dans le sous-arbre gauche de t'). Nous pouvons remarquer que Q a trois arguments : les deux premiers correspondent aux couples d'arbres de Q_0) et le troisième est la mémoire dont nous verrons l'utilité plus tard.

Dans $Prog_2$, la règle (3) $Q(\widehat{\alpha(X,Y)}, \widehat{\alpha(X',Y')}, R) \leftarrow Q(\widehat{X},\widehat{X'},\perp), Q(\widehat{Y},\widehat{Y'},R)$ nous dit que si nous avons deux arbres qui ont la même racine alors la relation d'inclusion relâchée est vérifiée entre leurs sous-arbres gauches X et X' et droits Y et Y'.

Pour expliquer la règle (4) $Q(\widehat{X}, \widehat{\alpha(Y,Z)}, R) \leftarrow Q(\widehat{X},\widehat{Y},R'), Q(\widehat{R'},\widehat{Z},R)$, nous rappelons que $\lhd_b$ est une application des positions du premier arbre vers celles du deuxième arbre. Remarquons encore que la règle (3) nous permet d'associer les racines α de deux sous arbres. La règle (4) traite des cas où cette association entre racine n'est pas possible. Dans cette situation, nous allons procéder à une recherche d'association en trois étapes :

- Nous essayons d'associer la racine de X avec une position de Y (qui est le sous-arbre gauche de $\alpha(Y,Z)$). Tant que cela est possible la procédure continue de manière descendante.
- Si nous arrivons à une position p de X qui ne peut pas être associée à une position de Y, après k itérations, la règle (2) sera utilisée pour transférer dans la mémoire le sous-arbre dont la racine est p. Cela va se faire via l'unification des littéraux *instanciés* $Q(\widehat{X},\widehat{Y},R')$ de la règle (4) et $Q(\widehat{X},\widehat{Y},X)$ de la règle (2). L'unification fera que $R' = X$. La dérivation faite dans l'exemple 6, illustre cette situation, car une fois que nous avons obtenu $\underline{Q}($  $,\perp,t'')$, $Q(t'',$  $,\perp)$

  la suite correspond à l'utilisation de la règle (2) pour obtenir

$\underline{A}(\bot)$, $Q($  , $,\bot)$ et nous voyons bien que le sous-arbre $c(\bot,\bot)$ est stocké dans la mémoire t''.

– La recherche d'associations peut alors continuer avec le deuxième atome du corps de la règle (4), à savoir, $Q(\widehat{R'},\widehat{Z},R)$. En d'autres termes nous cherchons maintenant l'association de la position p avec une position de Z (qui est le sous-arbre droit de ).

Nous remarquons que si $Q(t,t',t'')$ est vrai cela *n'indique pas* que $t \lhd_b t'$ *mais qu'il existe* un arbre t_1 - construit à partir d'un sous-arbre de t en supprimant sa branche droite t'' - tel que $t_1 \lhd_b t'$.

Pour mieux observer ce qui se passe, considérons encore l'exemple 6. Voici les arbres t et t' :

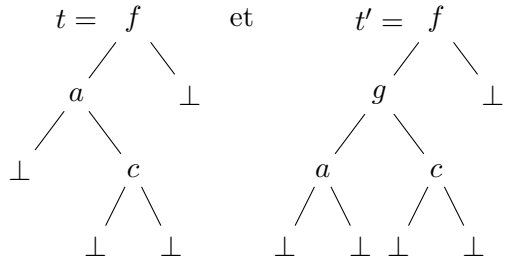


FIGURE 3.1 – Arbres t et t' de l'exemple 6

et leurs arbres d'arité non borné correspondants :

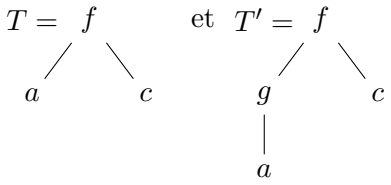
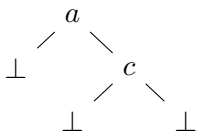
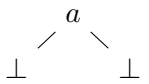
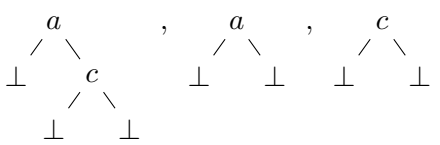


FIGURE 3.2 – Traduction en arbre d'arité non borné des arbres t et t' de la figure 2.1

En descendant sur l'arbre t à la recherche des associations pour toutes les positions, nous sommes bloqués sur la position de c. Notons que a et c sont des frères dans l'arbre d'arité non-borné correspondant à T (Figure 3.2). Dans t', comme nous ne trouvons pas une association de c sur le sous-arbre $a(\bot,\bot)$, cela indique que a n'a pas c comme son frère. Néanmoins, cela n'empêche pas que c soit un nœud à droite de a (comme c'est le cas dans la Figure 3.2). Nous voulons donc dans la version binaire des arbres continuer nos recherches. Nous le faisons en cherchant c sur le sous-arbre $c(\bot,\bot)$ de t', comme expliqué ci-dessus. Notons que effec-

tivement, dans notre exemple 6,  n'est pas inclu relaché dans 

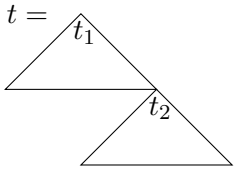
mais que $Q($  $)$ est vrai.

Il est donc clair que pour savoir si $t \lhd_b t'$ nous avons besoin de vérifier la propriété $Prop(t, t', t'')$ qui indique que l'arbre t_1 obtenu en enlevant t'' dans la branche droite de t, est inclus relâché dans t'. Pour l'exprimer de manière formelle, nous présentons les défintions ci-dessous.

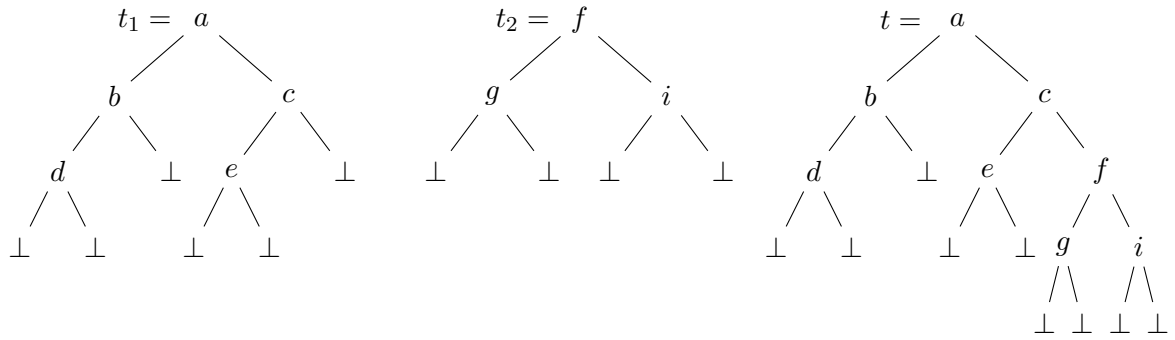
Definition 15. *Soit $t \in T_\Sigma$, rightmost-innermost($Pos(t)$) est la position $p \in Pos(t)$ telle que :*

- *$\forall w$ une séquence d'entier, $w \neq \epsilon \Longrightarrow p.w \notin Pos(t)$ (p est une position de feuille)*
- *p est de la forme $p = 1 \ldots 1$ ou $p = \epsilon$*

Definition 16. *Soit $t_1, t_2 \in T_\Sigma$, L'arbre $t = t_1 \natural t_2$ est défini tel que t_2 est rattaché à t_1 à la position rightmost-innermost($Pos(t_1)$).*



Example 7. Pour $t_1 = a(b(d(\bot, \bot), \bot), c(e(\bot, \bot), \bot))$ et $t_2 = f(g(\bot, \bot), i(\bot, \bot))$ on a $t = t_1 \natural t_2 = a(b(d(\bot, \bot), \bot), c(e(\bot, \bot), f(g(\bot, \bot), i(\bot, \bot))))$.



La définition suivante introduit la propriété *Prop*.

Definition 17. *$\forall\ t, t', t'' \in T_\Sigma$, $Prop(t, t', t'')$ est définie par : $\exists\ t_1 \in T_\Sigma$, $t = t_1 \natural t''$, $t_1 \lhd_b t'$.*

Dans la suite nous allons prouver que le langage généré par Q_0 est $L(Q_0) = R$. Pour cela nous devons prouver que notre programme logique est correct et complet.

Pour prouver la correction de notre programme $Prog$ nous devons prouver que si $Q(t, t', t'') \twoheadrightarrow^* \emptyset$ alors $Prop(t, t', t'')$. Les lemmes suivants sont nécessaires dans cette preuve.

Lemma 2. $\forall\ t \in T_\Sigma,\ A(t) \twoheadrightarrow^* \emptyset$.

The proofs of the following lemmas are in the appendix.

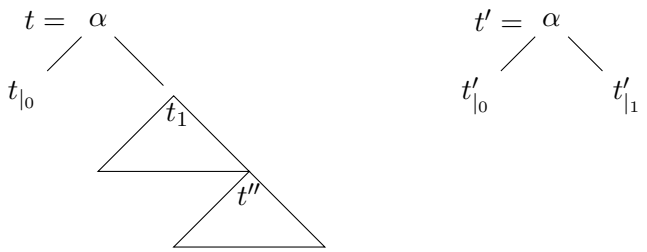
Lemma 3. *Let* $t_1, t'_1, t_2, t'_2 \in T_\Sigma,\ \alpha \in \Sigma$.
If $t_1 \lhd_b t'_1$ *and* $t_2 \lhd_b t'_2$ *then* $\alpha(t_1, t_2) \lhd_b \alpha(t'_1, t'_2)$.

Lemma 4. *Let* $t_1, t'_1, t_2, t'_2 \in T_\Sigma,\ \alpha \in \Sigma$.
If $t_1 \lhd_b t'_1$ *and* $t_2 \lhd_b t'_2$ *then* $t_1 \natural t_2 \lhd_b \alpha(t'_1, t'_2)$.

Property 5. *(Soundness) Let* $t, t', t'' \in T_\Sigma$.
If $Q(t, t', t'') \twoheadrightarrow^* \emptyset$ *then* $Prop(t, t', t'')$.

Démonstration. Let $t, t', t'' \in T_\Sigma$, We have to prove : if $Q(t, t', t'') \twoheadrightarrow^* \emptyset$ then $Prop(t, t', t'')$. The proof is done by induction on the length n of the rewrite derivation $Q(t, t', t'') \twoheadrightarrow^* \emptyset$. This derivation necessarily contains at least one step.

- If the first step uses the clause 2 : $Q(\widehat{X}, \widehat{Y}, X) \leftarrow A(\widehat{Y})$ of $Prog_2$, from Lemma 2 we have : $Q(\widehat{t}, \widehat{t'}, t) \twoheadrightarrow A(\widehat{t'}) \twoheadrightarrow^* \emptyset$
  We deduce that $t = \perp \natural t$ and $\forall\, t' \in T_\Sigma, \perp \lhd_b t'$. Hence we have $Prop(t, t', t)$.
- If the first step uses the clause 3 : $Q(\overbrace{\begin{array}{c}\alpha\\ /\ \backslash\\ X\ \ Y\end{array}}, \overbrace{\begin{array}{c}\alpha\\ /\ \backslash\\ X'\ Y'\end{array}}, R) \leftarrow Q(\widehat{X}, \widehat{X'}, \perp), Q(\widehat{Y}, \widehat{Y'}, R)$ of $Prog_2$, we have : $Q(\widehat{t}, \widehat{t'}, t'') \twoheadrightarrow Q(\widehat{t_{|_0}}, \widehat{t'_{|_0}}, \perp), Q(\widehat{t_{|_1}}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$
  From $Q(\widehat{t_{|_0}}, \widehat{t'_{|_0}}, \perp), Q(\widehat{t_{|_1}}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$, we can obviously extract the subderivations
  $(a)\, Q(\widehat{t_{|_0}}, \widehat{t'_{|_0}}, \perp) \twoheadrightarrow^* \emptyset$ and $(b)\, Q(\widehat{t_{|_1}}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$ whose length are strictly less than n.
  If we apply the induction hypothesis on the subderivation :
  – (a) we obtain $Prop(t_{|_0}, t'_{|_0}, \perp)$ and then $t_{|_0} \lhd_b t'_{|_0}$ (because $t_{|_0} = t_{|_0} \natural \perp$)
  – (b) we obtain $Prop(t_{|_1}, t'_{|_1}, t'')$ therefore $\exists\ t_1 \in T_\Sigma$ such that $t_{|_1} = t_1 \natural t''$ and $t_1 \lhd_b t'_{|_1}$.
  In summary on a drawing we have :

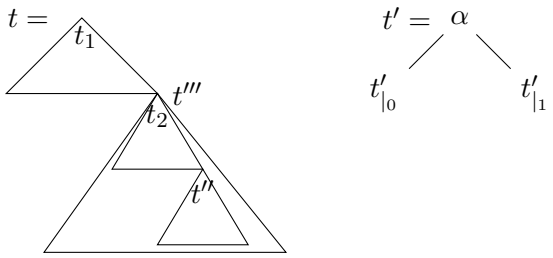


  Let $t_2 = \begin{array}{c}\alpha\\ /\ \backslash\\ t_{|_0}\ \ t_1\end{array}$, from Lemma 3 we have $t_2 \lhd_b t'$ (because $t_{|_0} \lhd_b t'_{|_0}$ and $t_1 \lhd_b t'_{|_1}$). We have found a tree $t_2 \in T_\Sigma$ such that $t = t_2 \natural t''$ and $t_2 \lhd_b t'$, therefore we have $Prop(t, t', t'')$.

- If the first step uses the clause 4 : $Q(\widehat{X}, \overset{\widehat{\alpha}}{\underset{Y\ \ Z}{/\ \backslash}}, R) \leftarrow Q(\widehat{X}, \widehat{Y}, R'), Q(\widehat{R'}, \widehat{Z}, R)$ of $Prog_2$,

  we have : $Q(\widehat{t}, \widehat{t'}, t'') \twoheadrightarrow Q(\widehat{t}, \widehat{t'_{|_0}}, t'''), Q(\widehat{t'''}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$

  From $Q(\widehat{t}, \widehat{t'_{|_0}}, t'''), Q(\widehat{t'''}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$, we can obviously extract the subderivations $(a)\, Q(\widehat{t}, \widehat{t'_{|_0}}, t''') \twoheadrightarrow^* \emptyset$ and $(b)\, Q(\widehat{t'''}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$ whose length are strictly less than n. If we apply the induction hypothesis on the subderivation :
  – (a) we obtain $Prop(t, t'_{|_0}, t''')$ therefore $\exists\ t_1 \in T_\Sigma$ such that $t = t_1 \natural\, t'''$ and $t_1 \lhd_b t'_{|_0}$.
  – (b) we obtain $Prop(t''', t'_{|_1}, t'')$ therefore $\exists\ t_2 \in T_\Sigma$ such that $t''' = t_2 \natural\, t''$ and $t_2 \lhd_b t'_{|_1}$.
  In summary on a drawing we have :



  Let $t_3 = t_1 \natural\, t_2$, from Lemma 4 we have $t_3 \lhd_b t'$ (because $t_1 \lhd_b t'_{|_0}$ and $t_2 \lhd_b t'_{|_1}$). We have found a tree $t_3 \in T_\Sigma$ such that $t = t_3 \natural\, t''$ and $t_3 \lhd_b t'$, therefore we have $Prop(t, t', t'')$. □

Notation : Let $t \in T_\Sigma$. $t_{|_0}$ denotes the left subtree of t, and $t_{|_1}$ denotes the right subtree of t.

The proof of the following lemma is in the appendix.

Lemma 6. *Let $t, t' \in T_\Sigma$, if $t \lhd_b t'$ and $t' \neq \perp$ then :*
– *$t \neq \perp$, $t(\epsilon) = t'(\epsilon)$, $t_{|_0} \lhd_b t'_{|_0}$ and $t_{|_1} \lhd_b t'_{|_1}$*
– *or $\exists\ t_1, t_2 \in T_\Sigma$, $t = t_1 \natural\, t_2$, $t_1 \lhd_b t'_{|_0}$ and $t_2 \lhd_b t'_{|_1}$*

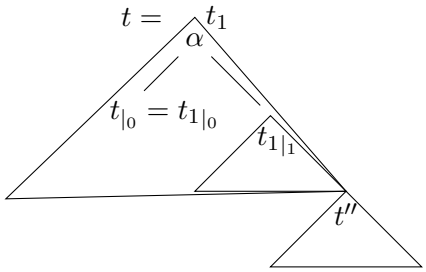
Property 7. *(Completeness) Let $t, t', t'' \in T_\Sigma$. If $Prop(t, t', t'')$ then $Q(t, t', t'') \twoheadrightarrow^* \emptyset$.*

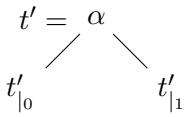
Démonstration. Let $t, t', t'' \in T_\Sigma$, We have to prove : if $Prop(t, t', t'')$ then $Q(t, t', t'') \twoheadrightarrow^* \emptyset$. The proof is done by induction on the size n of t'.

- if $t' = \perp$: since we have $Prop(t, \perp, t'')$ and therefore $\exists\ t_1 \in T_\Sigma$ such that $t = t_1 \natural\, t''$ and $t_1 \lhd_b \perp$. With $t_1 \lhd_b \perp$ we deduce that $t_1 = \perp$ and $t = \perp \natural\, t'' = t''$. We can derive $Q(\widehat{t}, \widehat{t'}, t'')$ by : $Q(\widehat{t}, \widehat{\perp}, t) \xrightarrow[clause\,2]{} A(\widehat{\perp}) \xrightarrow[clause\,5]{} \emptyset$.

- else $t' = \alpha(t'_{|_0}, t'_{|_1})$: since we have $Prop(t, t', t'')$, therefore $\exists\ t_1 \in T_\Sigma$ such that $t_1 \lhd_b t'$, and $t = t_1 \natural\, t''$. From Lemma 6, we have 2 cases for $t_1 \lhd_b t'$:

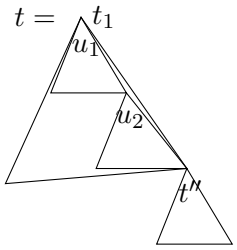
  1. either $t_{1|_0} \lhd_b t'_{|_0}$, $t_{1|_1} \lhd_b t'_{|_1}$, $t_1(\epsilon) = t'(\epsilon)$, and $t_1 \neq \perp$ then $t(\epsilon) = t_1(\epsilon) = t'(\epsilon) = \alpha$. On a drawing we have :

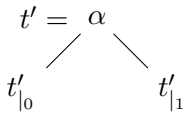




We can apply the clause 3 : $Q(\widehat{t}, \widehat{t'}, t'') \twoheadrightarrow Q(\widehat{t_{|_0}}, \widehat{t'_{|_0}}, \bot), Q(\widehat{t_{|_1}}, \widehat{t'_{|_1}}, t'')$. We know that :
– $t_{|_0} = t_{1|_0} = t_{1|_0} \natural \bot$ and $t_{1|_0} \lhd_b t'_{|_0}$ thus we deduce $Prop(t_{|_0}, t'_{|_0}, \bot)$
– and also $t_{|_1} = t_{1|_1} \natural t''$ and $t_{1|_1} \lhd_b t'_{|_1}$ thus we deduce $Prop(t_{|_1}, t'_{|_1}, t'')$
Since $|t'_{|_0}| < |t'|$ and $|t'_{|_1}| < |t'|$, we can apply the induction hypothesis on $Prop(t_{|_0}, t'_{|_0}, \bot)$ and $Prop(t_{|_1}, t'_{|_1}, t'')$ and we deduce that :
(a) $Q(\widehat{t_{|_0}}, \widehat{t'_{|_0}}, \bot) \twoheadrightarrow^* \emptyset$ and (b) $Q(\widehat{t_{|_1}}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$. With (a) and (b) we conclude that $Q(\widehat{t}, \widehat{t'}, t'') \twoheadrightarrow Q(\widehat{t_{|_0}}, \widehat{t'_{|_0}}, \bot), Q(\widehat{t_{|_1}}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$.

2. or $\exists\ u_1, u_2 \in T_\Sigma$ such that $t_1 = u_1 \natural u_2, u_1 \lhd_b t'_{|_0}$ and $u_2 \lhd_b t'_{|_1}$. On a drawing we have :





Let $v \in T_\Sigma$, we can apply the clause 4 : $Q(\widehat{t}, \widehat{t'}, t'') \twoheadrightarrow Q(\widehat{t}, \widehat{t'_{|_0}}, v), Q(\widehat{v}, \widehat{t'_{|_1}}, t'')$.
By setting $v = u_2 \natural t''$, we can write $t = u_1 \natural v$. We know that :
– $t = u_1 \natural v$ and $u_1 \lhd_b t'_{|_0}$ thus we deduce $Prop(t, t'_{|_0}, v)$
– and also $v = u_2 \natural t''$ and $u_2 \lhd_b t'_{|_1}$ thus we deduce $Prop(v, t'_{|_1}, t'')$
Since $|t'_{|_0}| < |t'|$ and $|t'_{|_1}| < |t'|$, we can apply the induction hypothesis on $Prop(t, t'_{|_0}, v)$ and $Prop(v, t'_{|_1}, t'')$ and we deduce that :
(a) $Q(\widehat{t}, \widehat{t'_{|_0}}, v) \twoheadrightarrow^* \emptyset$ and (b) $Q(\widehat{v}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$. With (a) and (b) we conclude that $Q(\widehat{t}, \widehat{t'}, t'') \twoheadrightarrow Q(\widehat{t}, \widehat{t'_{|_0}}, v), Q(\widehat{v}, \widehat{t'_{|_1}}, t'') \twoheadrightarrow^* \emptyset$.

□

D'après Property 5 et Property 7, nous avons :

Theorem 8. *Soient $t, t', t'' \in T_\Sigma$. $Q(t, t', t'') \in Mod(Prog)$ ssi $Prop(t, t', t'')$.*

Corollary 9. *Soient $t, t' \in T_\Sigma$. $Q_0(t, t') \in Mod(Prog)$ ssi $t \lhd_b^r t'$. Le langage généré par Q_0 est $L(Q_0) = R$.*

Démonstration. Soient $t, t' \in T_\Sigma$. $Q_0(t,t') \twoheadrightarrow Q(t,t',\bot)$ avec $t(\epsilon) = t'(\epsilon)$, $f(\epsilon) = \epsilon$ et $t_{|_1} = t'_{|_1} = \bot$.
Or d'après le théorème 8, nous savons que $Q_0(t,t',\bot) \twoheadrightarrow^* \emptyset$ ssi $Prop(t,t',\bot)$. Et par définition $Prop(t,t',\bot)$ signifie $\exists\, t_1 \in T_\Sigma$, $t = t_1 \natural \bot$, $t_1 \lhd_b t'$. Comme $t = t_1 \natural \bot$ alors $t = t_1$ donc $t \lhd_b t'$. $f(\epsilon) = \epsilon$ et $t \lhd_b t'$ donc on peut conclure que $t \lhd_b^r t'$. □

3.3 Etude de la projection régulière

Même si la projection sur une composante d'un programme de langages synchronisés d'arbres ne présente pas de grandes difficultés, il n'est pas évident de pouvoir trouver un programme régulier pour cette projection.

Un **programme logique** est **régulier** si tous les prédicats n'ont qu'un seul argument de sortie, et pour toutes les clauses de la forme $H \leftarrow B$, $Output(B)$ est linéaire et ne contient que des variables. Un programme régulier génère un langage régulier d'arbres. Nous référençons [9] pour plus de détails sur les langages réguliers.

Nous remarquons que la recherche du programme régulier n'est possible que si la projection sur l'argument est un langage régulier. En analysant le programme 1, nous pensons que la projection sur le premier argument après intersection du 2^{eme} argument avec un langage régulier, génère bien un langage régulier.

L'intersection de $L(Q_0)$ avec un langage régulier sera exprimée par un programme $Prog'$ similaire à *Prog*. Nous pouvons trouver l'algorithme qui calcule l'intersection d'un langage synchronisé d'arbres avec un langage régulier dans [2]. Les clauses du programme $Prog'$ sont de la même forme que les clauses de $Prog$ avec des prédicats ayant de nouveaux noms.

Dans cette section nous n'expliquons pas comment l'intersection est faite mais nous présentons un algorithme qui, à partir du programme $Prog'$, calcule le programme régulier $\Pi(Prog)$ correspondant à la projection sur le premier argument de l'axiome de $Prog'$.
Voici l'algorithme qui calcule la projection :
Dans l'algorithme ci-dessous, $\widehat{q}$ correspond à la projection de q sur son premier argument (de sortie), alors que $\overline{q}$ correspond à la génération de l'argument d'entrée de q.

Algorithme 2. Pour chaque clause du programme $Prog'$, si la clause est de la forme :

1. $p(\widehat{X},\widehat{Y}) \leftarrow q(\widehat{X},\widehat{Y},\bot)$ alors les clauses
   (a) $\widehat{p}(\widehat{X}) \leftarrow \widehat{q}(\widehat{X})$
   (b) $\overline{q}(\bot) \leftarrow$
   appartiennent au programme $\Pi(Prog)$
2. $q(\overbrace{\begin{array}{c}\alpha\\ /\ \backslash\\ X\ \ Y\end{array}}^{\widehat{\ }}, \overbrace{\begin{array}{c}\alpha\\ /\ \backslash\\ X'\ Y'\end{array}}^{\widehat{\ }}, R) \leftarrow q'(\widehat{X},\widehat{X'},\bot), q''(\widehat{Y},\widehat{Y'},R)$ alors les clauses
   (a) $\widehat{q}(\overbrace{\begin{array}{c}\alpha\\ /\ \backslash\\ X\ \ Y\end{array}}^{\widehat{\ }}) \leftarrow \widehat{q'}(\widehat{X}), \widehat{q''}(\widehat{Y})$
   (b) $\overline{q'}(\bot) \leftarrow$
   (c) $\overline{q''}(\widehat{R}) \leftarrow \overline{q}(\widehat{R})$
   appartiennent au programme $\Pi(Prog)$

3. $q(\widehat{R}, \widehat{X}, R) \leftarrow a(\widehat{X})$ alors la clause
   (a) $\widehat{q}(\widehat{R}) \leftarrow \overline{q}(\widehat{R})$

   appartient au programme $\Pi(Prog)$
4. $q(\widehat{X}, \overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ Y\ \ Z\end{array}}, R) \leftarrow q'(\widehat{X}, \widehat{Y}, R_1), q''(\widehat{R_1}, \widehat{Z}, R)$ alors les clauses
   (a) $\widehat{q}(\widehat{X}) \leftarrow \widehat{q'}(\widehat{X})$
   (b) $\overline{q'}(\widehat{R_1}) \leftarrow \widehat{q''}(\widehat{R_1})$
   (c) $\overline{q''}(\widehat{R}) \leftarrow \overline{q}(\widehat{R})$

   appartiennent au programme $\Pi(Prog)$

□

Lorsqu'on veut tranformer une clause $H \leftarrow B$ du programme $Prog'$ pour obtenir le(s) clause(s) du programme régulier $\Pi(Prog)$ on fait les changements suivants :
- soit on garde la clause sans la changer,
- soit on échange le prédicat de la tête de la clause avec l'un des prédicats du corps de la clause.

Dans ces deux cas des modifications sont effectués sur les prédicats de H et B.
Les prédicats de $Prog'$ qui ont la forme $q(\widehat{t_1}, \widehat{t_2}, t_3)$ sont transformés de deux façons :
- soit $q(\widehat{t_1}, \widehat{t_2}, t_3)$ se transforme en $\widehat{q}(\widehat{t_1})$: lorsque le prédicat q se trouve dans la tête de clause H (resp B) de $Prog'$ et il reste dans H (resp B) de $\Pi(Prog)$. Les arguments sont supprimés sauf l'argument sur lequel l'on fait la projection.
- soit $q(\widehat{t_1}, \widehat{t_2}, t_3)$ se transforme en $\overline{q}(t_3)$: lorsque le prédicat q se trouve dans la tête de clause H (resp B) de $Prog'$ et il est dans B (resp H) de $\Pi(Prog)$. Les arguments sont supprimés sauf l'argument d'entrée.

Les prédicats de $Prog'$ qui ont la forme $p(\widehat{t_1}, \widehat{t_2})$ sont transformés d'une seule façon $\widehat{p}(\widehat{t_1})$ car il ne possède pas d'argument d'entrée.

Remarque 18. $\Pi(Prog)$ est un programme régulier.

Example 8. En prenant le programme de la relation R que voici :
$Prog = \{Q_0(\overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ X\ \ \bot\end{array}}, \overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ X'\ \bot\end{array}}) \leftarrow Q(\widehat{X}, \widehat{X'}, \bot).$
$Q(\widehat{X}, \widehat{Y}, X) \leftarrow A(\widehat{Y}).$
$Q(\overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ X\ \ Y\end{array}}, \overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ X'\ Y'\end{array}}, R) \leftarrow Q(\widehat{X}, \widehat{X'}, \bot), Q(\widehat{Y}, \widehat{Y'}, R).$
$Q(\widehat{X}, \overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ Y\ \ Z\end{array}}, R) \leftarrow Q(\widehat{X}, \widehat{Y}, R'), Q(\widehat{R'}, \widehat{Z}, R).$
$A(\widehat{\bot}) \leftarrow .\quad A(\overbrace{\begin{array}{c}\alpha\\ /\ \backslash \\ X\ \ Y\end{array}}) \leftarrow A(\widehat{X}), A(\widehat{Y}). \mid \alpha \in \Sigma\}$

nous allons calculer la projection sur le premier argument du prédicat Q_0. Avec l'algorithme de projection décrit ci-dessus on obtient :
$\Pi_1(Prog) = \{\widehat{Q_0}(\widehat{X}) \leftarrow \widehat{Q}(\widehat{X}).$ (* Tranformation 1a *)

$\overline{Q}(\bot) \leftarrow .$ (* Tranformation 1b *)
$\overline{Q}(\widehat{R'}) \leftarrow \widehat{Q}(\widehat{R'}).$ (* Tranformation 4b *)
$\widehat{Q}(\overset{\frown}{\underset{X\ \ Y}{\overset{\alpha}{/\ \backslash}}}) \leftarrow \widehat{Q}(\widehat{X}), \widehat{Q}(\widehat{Y}).$ (* Tranformation 2a *)
$\widehat{Q}(\widehat{R}) \leftarrow \overline{Q}(\widehat{R}).$ (* Tranformation 3a *) $\mid \alpha \in \Sigma\}$

En observant le langage $L(Q_0)$ obtenu via $\Pi_1(Prog)$, nous pouvons remarquer que $L(Q_0) = T_\Sigma$. En effet, la relation R est l'ensemble de tous les couples $(t, t') \in T_\Sigma^2$ tel que $t \lhd_b t'$, et en ne retenant que les arbres t des couples (t, t') on obtient bien l'ensemble T_Σ.

Conjecture 10. $L_{\Pi(Prog)}(\widehat{p}) = \{t \mid \exists t', (t, t') \in L_{prog}(P)\}$.

L'algorithme de projection a été implémanté, ce qui nous a permis de vérifier la conjecture par des tests. Pour faute de temps nous n'avons pas pu écrire les preuves pour l'algorithme de projection.

Chapitre 4

Transformation de grammaires

Il est important de rappeler que la méthode de vérification de l'inclusion relâchée introduite dans ce travail est fondée sur les arbres binaires, alors que notre but est de pouvoir comparer des schémas XML, c-à-d, des langages d'arbre d'arité non bornée. Nous avons donc besoin de transformer des grammaires d'arbres d'arité non borné en grammaires d'arbres binaires correspondantes.

Pour implémenter nos algorithmes de transformation, nous utilisons des fonctions permettant de manipuler les Expressions Régulières. Dans la suite nous prenons les définitions présentées dans [6] pour les fonctions introduites dans [1, 14].

4.1 Les expressions régulières

Les expressions régulières permettent de décrire les langages réguliers. A partir d'un alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_2\}$, l'ensemble des expressions régulières sur Σ est défini par induction comme ceci : $E ::= \emptyset \mid \epsilon \mid \sigma_i \mid E + E \mid E.E \mid E^+ \mid E^* \mid E? \mid (E)$. A partir d'une expression régulière E sur un alphabet Σ, nous définissons l'expression régulière avec indice $\overline{E}$ en numérotant chaque symbole de E avec un entier positif unique. Cette opération nous permet d'identifier chaque position de E. Comme exemple si nous avons l'expression régulière $E = (a.(b + c)^*)^*.a$, son expression régulière avec indice est $\overline{E} = (a_1.(b_2 + c_3)^*)^*.a_4$. Nous pouvons supprimer le symbole associé à chaque position car étant identifié par sa position. Dans la suite nous allons travailler avec seulement les expressions régulières formées par les positions. A chaque fois que le symbole sera utile, nous allons le retrouver par $symb_E(p)$, où p est la position du symbole dans l'expression régulière E. Comme exemple si nous avons l'expression régulière $E = (a.(b + c)^*)^*.a$, nous allons l'écrire $E = (1.(2 + 3)^*)^*.4$ tout en assumant que $symb_E(1) = a$, $symb_E(2) = b$, $symb_E(3) = c$, $symb_E(4) = a$. L'ensemble des positions d'une expression régulière E est noté $Pos(E)$. Etant donné un mot w, $|w|$ est la longueur du mot w. Par exemple si $w = abc$ alors $|w| = 3$. $L(E)$ désigne le langage reconnu par l'expression régulière E. Les fonctions utilisées dans notre algorithme sont donc les suivantes :

Definition 19. *L'opération $Null_E$ est égal à $\{\epsilon\}$ si $\epsilon \in L(E)$ et $\emptyset$ sinon. L'opération $Null_E$ est défini par induction comme ceci :*

$$Null_\emptyset = \emptyset \qquad Null_\varepsilon = \{\varepsilon\} \qquad Null_a = \emptyset$$
$$Null_{F+G} = Null_F \cup Null_G \qquad Null_{F.G} = Null_F \cap Null_G \qquad Null_{F^+} = Null_F$$
$$Null_{F^*} = \{\varepsilon\} \qquad Null_{F?} = \{\varepsilon\} \qquad Null_{(E)} = Null_E \ \square$$

$Null_E$ permet de savoir si $\epsilon \in L(E)$. Il peut être calculé en temps linéaire sur la taille de E.

Definition 20. *La fonction $First(\overline{E})$ est l'ensemble des positions initiales de $\overline{E}$, correspondant à tous les mots du langage $L(E)$.*

$$First(\overline{E}) = \{x \in Pos(\overline{E}) \mid \exists u \in \Sigma^* : \alpha u \in L(E) \wedge symb(x) = \alpha\}$$

REMARK : *Cette fonction est défini sur les expressions régulières avec indice. Par abus de notation, nous allons écrire $First(E)$ au lieu de $First(\overline{E})$. nous utiliserons la même convention pour les autres fonctions qui seront définies sur les expressions régulières.*

La fonction $First(E)$ est défini par induction comme ceci :

$$First(\emptyset) = \emptyset \qquad First(\varepsilon) = \emptyset$$
$$First(\alpha_x) = x \qquad First(F + G) = First(F) \cup First(G)$$
$$First(F.G) = First(F) \cup Null_F.First(G) \qquad First(F^+) = First(F)$$
$$First(F^*) = First(\ (F)\ ) = First(F) \qquad First(F?) = First(F) \qquad \square$$

$First(E)$ peut être calculé en temps linéaire sur la taille de E.

Definition 21. *Etant donné une expression régulière E, $Last(E)$ est l'ensemble des positions finales correspondant aux mots du langage $L(E)$.*

$$Last(E) = \{x \in Pos(E) \mid \exists u \in \Sigma^* : u\alpha \in L(E) \wedge symb(x) = \alpha\}$$

La fonction $Last(E)$ est défini par induction comme ceci :

$$Last(\emptyset) = \emptyset \qquad Last(\varepsilon) = \emptyset$$
$$Last(\alpha_x) = x \qquad Last(F + G) = Last(F) \cup Last(G)$$
$$Last(F.G) = Last(G) \cup Null_G.Last(F) \qquad Last(F^+) = Last(F)$$
$$Last(F^*) = Last(\ (F)\ ) = Last(F) \qquad Last(F?) = Last(F) \qquad \square$$

$Last(E)$ peut être calculé en temps linéaire sur la taille de E.
Pour chaque ensemble X, on note par $\mathcal{I}_X$ la fonction de X vers $\{\{\varepsilon\}, \emptyset\}$ définie par :
$\mathcal{I}_X(x) = \emptyset$ si $x \notin X$ et $\mathcal{I}_X(x) = \varepsilon$ si $x \in X$.
Cette fonction sera utilisée dans les prochaines définitions, pour sélectionner les positions d'une expression régulière.

Definition 22. *$Follow(E, x)$ est l'ensemble des positions qui suivent immédiatement la position x dans l'expression E. Si x n'est pas une position de E alors $Follow(E, x) = \emptyset$.*

$$Follow(E, x) = \{y \in Pos(E) \mid \exists v, w \in \Sigma^* : v\alpha_1\alpha_2 w \in L(E) \wedge symb(x) = \alpha_1 \wedge symb(y) = \alpha_2\}$$

La fonction $Follow(E, x)$ est défini par induction comme ceci :

$$\begin{aligned}
Follow(\emptyset, x) &= Follow(\varepsilon, x) = Follow(a, x) = \emptyset \\
Follow(F + G, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\
Follow(F.G, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\
&\quad \cup \mathcal{I}_{Last(F)}(x).First(G) \\
Follow(F^+, x) &= Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F) \\
Follow(F^*, x) &= Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F) \\
Follow(F?, x) &= Follow(\ (F)\ , x) = Follow(F, x)
\end{aligned}$$ □

Definition 23. *$Previous(E, x)$ est l'ensemble des positions qui précèdent immédiatement la position x dans l'expression E. Si x n'est pas une position de E alors $Previous(E, x) = \emptyset$.*

$$\begin{aligned}
Previous(E, x) = \{y \in Pos(E) | \exists v, w \in \Sigma^* : v\alpha_2\alpha_1 w \in L(E) \wedge \\
symb(x) = \alpha_1 \wedge symb(y) = \alpha_2\}
\end{aligned}$$

La fonction $Previous(E, x)$ est défini par induction comme ceci :

$$\begin{aligned}
Previous(\emptyset, x) &= \emptyset \\
Previous(\varepsilon, x) &= \emptyset \\
Previous(a, x) &= \emptyset \\
Previous(F + G, x) &= \mathcal{I}_{Pos(F)}(x).Previous(F, x) \cup \mathcal{I}_{Pos(G)}(x).Previous(G, x) \\
Previous(F.G, x) &= \mathcal{I}_{Pos(F)}(x).Previous(F, x) \cup \mathcal{I}_{Pos(G)}(x).Previous(G, x) \\
&\quad \cup \mathcal{I}_{First(G)}(x).Last(F) \\
Previous(F^+, x) &= Previous(F, x) \cup \mathcal{I}_{First(F)}(x).Last(F) \\
Previous(F^*, x) &= Previous(F, x) \cup \mathcal{I}_{First(F)}(x).Last(F) \\
Previous(F?, x) &= Previous(\ (F)\ , x) = Previous(F, x)
\end{aligned}$$ □

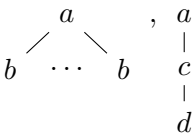
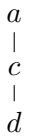
Les fonctions *Follow* et *Previous* peuvent être calculées en temps linéaire.

4.2 Grammaire d'arbres d'arité non borné

Definition 24. *Une grammaire d'arbres d'arité non borné A est définie par $A = (Q, \Sigma, q_{root}, \Delta_A)$ où :*

- *Q est l'ensemble des non terminaux*
- *Σ est l'aphabet*
- *$q_{root} \in Q$ est l'axiome*
- *Δ_A est l'ensemble des règles*

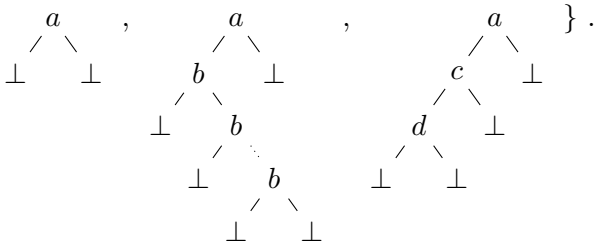
Les règles de production sont de la forme $q \to \alpha, E$ où $q \in Q$, $\alpha \in \Sigma$ et E est une expression régulière formée des non terminaux de Q.

Example 9. Soit la grammaire d'arité non borné $A = (\{X, Y, Z, D\}, \{a, b, c, d\}, X, \Delta)$ avec $\Delta = \{X \to a, [Y^* + Z];\ Y \to b, [\varepsilon];\ Z \to c, [D];\ D \to d, [\epsilon]\}$.
Le langage reconnu par A est $L(A) = \{$  ,  $\}$.

4.3 Grammaire d'arbres binaire

Definition 25. *Une grammaire d'arbres binaire B est définie par $B = (Q, \Sigma, q_{root}, \Delta_B)$ où :*
- *Q est l'ensemble des non terminaux*
- *Σ est l'aphabet*
- *$q_{root} \in Q$ est l'axiome*
- *Δ_B est l'ensemble des règles*

Les règles de production sont de la forme $q \to \alpha, (q_g, q_d)$ où $q, q_g, q_d \in Q$ et $\alpha \in \Sigma$.

Example 10. Soit la grammaire binaire $B = (\{X, Y, Z, D, Q_\perp\}, \{a, b, c, d, \perp\}, X, \Delta)$ avec $\Delta = \{X \to a, (Q_\perp, Q_\perp);\ X \to a, (Y, Q_\perp);\ X \to a, (Z, Q_\perp);\ Y \to b, (Q_\perp, Q_\perp);$
$Y \to b, (Q_\perp, Y);\ Z \to c, (D, Q_\perp);\ D \to d, (Q_\perp, Q_\perp);\ Q_\perp \to \perp\}$.
Le langage reconnu par B est $L(B) = \{$  $\}$.

4.4 Algorithme de transformation d'une grammaire d'arbres d'arité non borné en grammaire d'arbres binaires

Soit $\mathcal{A} = (Q, \Sigma, q_{root}, \Delta_{\mathcal{A}})$ une grammaire d'arbres d'arité non borné. Nous voulons construire la grammaire d'arbres binaires $\mathcal{B} = (Q, \Sigma, Q_f, \Delta_{\mathcal{B}})$ tel que $L(\mathcal{A}) = \{t \mid fcns(t) \in L(\mathcal{B})\}$ où $fcns(t)$ est le codage en first-child next-sibling de l'arbre t. Voici l'algorithme que nous proposons pour passer de $\mathcal{A}$ à $\mathcal{B}$.

NOTES :
- Dans la suite nous allons considérer seulement les règles de production pour les nœuds internes (pas les feuilles). Les règles de production pour les feuilles sont de la forme $q_a \to a, q_\perp$.
- Soit $\alpha \in \Sigma$. On définit par q_α le non terminal qui se trouve à gauche de la flèche d'une règle où α se trouve à droite, c-a-d, $q_\alpha \to \alpha, E$.
- Nous considérons les grammaires dans leur forme normale donc une règle de production pour chaque non terminal q à gauche de la flèche.
- $Symb(k) = \alpha_k$ et q_{α_k} est le non terminal se trouvant à gauche de la règle de production ayant α_k à sa droite.

Algorithme 3. L'algorithme de transformation est le suivant :

Entrée : $\mathcal{A}$: grammaire d'arbres d'arité non borné
Sortie : $\mathcal{B}$: grammaire d'arbres binaire
//Règle de production concernant l'axiome : $q_{root} \to root, E_0$
pour chaque $k \in First(E_0)$ **faire**
    Ajouter $q_{root} \to root, (q_{\alpha_k}, q_\perp)$ **à** $\mathcal{B}$
si $(Null(E_0) = \{\epsilon\})$ **alors**
  Ajouter $q_{root} \to root, (q_\perp, q_\perp)$ **à** $\mathcal{B}$

pour chaque règle r_u dans $\mathcal{A}$ ayant la forme $\boxed{r_u : q \to a, E}$ (où $E \neq q_\perp$) **faire** {
  pour chaque position i dans E **faire** {
    Considérer la règle $\boxed{q_{\alpha_i} \to \alpha_i, E'}$ dans $\mathcal{A}$
    // Règle ayant le non terminal q_{α_i} à gauche

    pour chaque $l \in Follow(E, i)$ (Définissons $\alpha_l = \alpha_\perp$ quand $Follow(E, i) = \emptyset$) **faire** {
    // Pour chaque position l suivant la position i.
    // Note : Chaque position l determine un possible frère immédiat de α_i
    // (en suivant les contraintes établies par E).
      Pour chaque $j \in First(E')$ **faire** {
      // Pour chaque possible premier fils de α_i.
        Ajouter $q_{\alpha_i} \to \alpha_i, (q_{\alpha_j}, q_{\alpha_l})$**à** $\mathcal{B}$
        si $(i \in Last(E)$ et $\alpha_l \neq \alpha_\perp)$ **alors**
          Ajouter $q_{\alpha_i} \to \alpha_i, (q_{\alpha_j}, q_\perp)$ **à** $\mathcal{B}$
      }
      si $(Null(E') = \{\epsilon\}$ et $\alpha_l \neq \alpha_\perp)$ **alors** **Ajouter** $q_{\alpha_i} \to \alpha_i, (q_\perp, q_{\alpha_l})$ **à** $\mathcal{B}$
      si $(i \in Last(E)$ et $Null(E') = \{\epsilon\})$ **alors** **Ajouter** $q_{\alpha_i} \to \alpha_i, (q_\perp, q_\perp)$ **à** $\mathcal{B}$
    }
  }
}

Pour créer les règles de production de la grammaire binaire $\mathcal{B}$, l'algorithme considère chaque règle de la grammaire d'arité non borné $\mathcal{A}$. Pour une règle de $\mathcal{A}$ sous la forme $q \to a, E$, l'on parcourt toutes les positions de l'expression régulière E. Pour chaque position i de E, on cherche dans $\mathcal{A}$ la règle correspondant au non terminal qui se trouve à la position i dans E. Comme la grammaire $\mathcal{A}$ est dans sa forme normale, il n'existe qu'une seule règle correspondant à ce non terminal qui sera $q_{\alpha_i} \to \alpha_i, E'$. La prochaine étape consiste à calculer :

- Les non terminaux q_{α_l}, correspondant aux positions de $Follow(E, i)$, sont les non terminaux des frères immédiats pour le non terminal q_{α_i}.
  Plus précisement, comme dans $L(\mathcal{A})$, un arbre ayant un nœud associé au symbole α_i peut avoir un frère associé à α_l, nous nous intéressons au non terminaux q_{α_l}.
- Les non terminaux q_{α_j}, correspondant aux positions de $First(E')$, sont les non terminaux des premiers fils pour le non terminal q_{α_i}.
  Plus précisement, comme dans $L(\mathcal{A})$, un arbre ayant un nœud associé au symbole α_i peut avoir un premier fils associé à α_j, nous nous intéressons au non terminaux q_{α_j}.

Enfin en prenant les non terminaux correspondant aux positions des ensembles $Follow(E, i)$ et $First(E')$ on crée les règles $q_{\alpha_i} \to \alpha_i, (q_{\alpha_j}, q_{\alpha_l})$ pour la grammaire $\mathcal{B}$. Les autres règles de $\mathcal{B}$ sont déduites des cas particuliers, comme la règle de l'axiome.

Example 11. En utilisant l'algorithme 3 pour transformer la grammaire d'arbres d'arité non borné de l'exemple 9, on obtient la grammaire d'arbres binaire de l'exemple 10.

Chapitre 5

Etat de l'art

Le problème d'inclusion relâchée de langages d'arbres comme nous l'abordons dans ce rapport, n'a pas encore été étudié. Cependant pour répondre à la substitution de services web, il existe une solution qui est l'inclusion de schéma ou plus précisément l'inclusion de langages d'arbres. Si l'on veut remplacer un service S_1, il faudra trouver un service S_1' qui fournissent les mêmes informations, c'est à dire les mêmes documents XML. Ceci revient à faire une comparaison ensembliste. Il existe des travaux sur l'inclusion de schéma. Les travaux [4, 11] ont cherché à réduire la complexité. Notre méthode est plus large que celle-ci car elle nous permet de remplacer un service S_1 par un autre service S_1' qui fournit des documents XML qui contiennent plus d'informations que les documents de S_1.
Avant d'étudier la comparaison de langages d'arbres, je me suis d'abord intéressé à la comparaison d'arbres que nous allons voir en détails dans ce chapitre. Les travaux liés à la comparaison d'arbres peuvent être trouvés dans [10, 8, 5, 13].

5.1 Comparaison d'arbres

Dans cette partie, nous allons parler des différents algorithmes d'inclusions d'arbres et les comparer suivant leur complexité. Nous considérons deux arbres enracinés et ordonnés P et T. P est l'arbre pattern (le sous-arbre) et T l'arbre cible.

Dans le papier de Richter [10], l'auteur décrit un algorithme d'inclusion relâché d'arbres qui consiste à calculer une fonction de mapping entre l'arbre pattern P et l'arbre cible T. Si le mapping entre les nœuds des deux arbres est réussi alors P est inclus dans T. L'algorithme s'arrête dès qu'il trouve une fonction de mapping f (car il peut avoir plusieurs mapping entre les nœuds de P et T).

Example 12. Les nœuds de même couleur correspondent à l'antécédent dans P et l'image dans T.

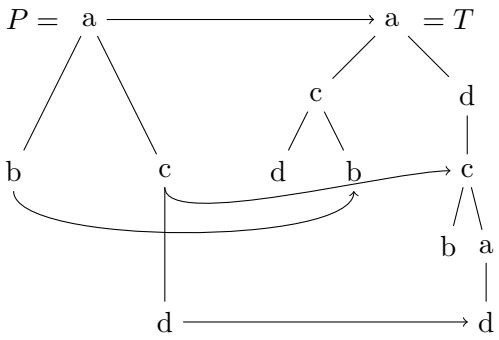


FIGURE 5.1 – Exemple de fonction de mapping

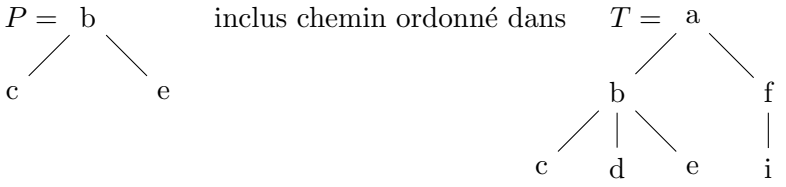
L'algorithme proposé par Richter a une complexité de $O(|\Sigma_P|.|T|+\#matches.DEPTH(T))$ (où Σ_P est l'ensemble des étiquettes de l'arbre pattern et $\#matches$ est le nombre de pairs $(v, w) \in P*T$ avec $Label(v) = Label(w)$). Cet algorithme améliore celui proposé par Mannila et Kilpelainen [8] ayant une complexité de $O(|P|.|T|)$. Mannila et Kilpelainen [8] sont les premiers à proposer un algorithme en temps polynomial qui résoud ce problème. Ces deux articles [10, 8] proposent d'appliquer leurs algorithmes à la recherche de sous-arbres, dans l'arbre d'analyse syntaxique d'une phrase en langage naturel. On peut ainsi extraire une ou plusieurs sous-phrases de la phrase initiale, en ne gardant que les informations souhaitées par l'utilisateur.

Il existe trois autres définitions d'inclusions d'arbres que nous pouvons trouver dans [10]. Voici donc ces définitions avec des exemples :

Definition 26. *(Inclusion chemin ordonné) P est inclus chemin ordonné dans T s'il existe une application f des nœuds de P (c'est à dire des positions de P) vers les nœuds de T, telle que :*

1. $\forall v \in P$ *: v et $f(v)$ ont la même étiquette,*
2. $\forall v_1, v_2 \in Pos(P)$ *: v_1 est le parent de $v_2 \Longrightarrow f(v_1)$ est le parent de $f(v_2)$,*
3. $\forall v_1, v_2 \in Pos(P)$ *: v_1 est à gauche de $v_2 \Longrightarrow f(v_1)$ est à gauche de $f(v_2)$.*

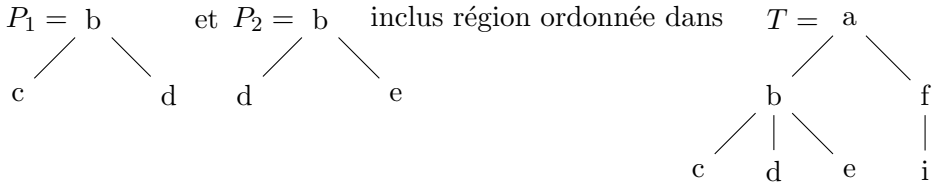
Example 13. Dans la figure ci-dessous, P est inclus chemin ordonné dans T.



Definition 27. *(Inclusion région ordonnée) P est inclus région ordonnée dans T s'il existe une application f des nœuds de P (c'est à dire des positions de P) vers les nœuds de T, telle que :*

1. $\forall v \in P$ *: v et $f(v)$ ont la même étiquette,*
2. $\forall v_1, v_2 \in Pos(P)$ *: v_1 est le parent de $v_2 \Longrightarrow f(v_1)$ est le parent de $f(v_2)$,*
3. $\forall v_1, v_2 \in Pos(P)$ *: v_1 est le frère de $v_2 \Longrightarrow f(v_1)$ est le frère de $f(v_2)$.*

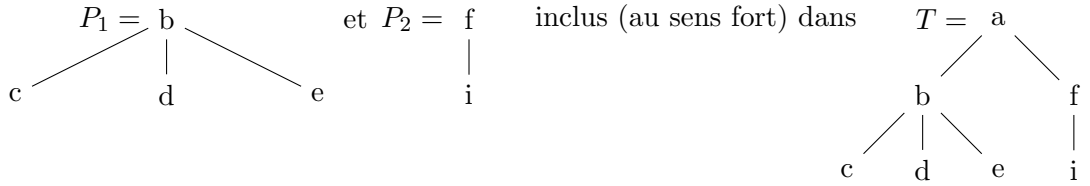
Example 14. Dans la figure ci-dessous, P_1 et P_2 sont inclus région ordonnée dans T.



Definition 28. *(Inclusion forte) P est inclus (au sens fort) dans T s'il existe une application f des nœuds de P (c'est à dire des positions de P) vers les nœuds de T, telle que :*

1. $\forall v \in P$ *: v et $f(v)$ ont la même étiquette,*
2. $\forall v \in P$ *: si v n'est pas une feuille alors le $k^{ième}$ enfant de v correspond au $k^{ième}$ enfant de $f(v)$.*

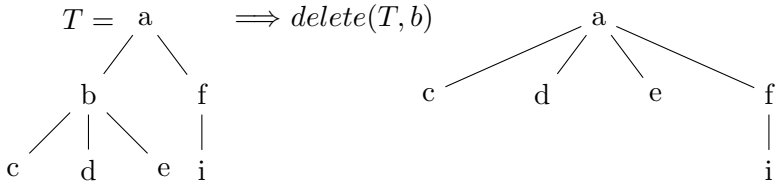
Example 15. Dans la figure ci-dessous, P_1 et P_2 sont inclus (au sens fort) dans T.



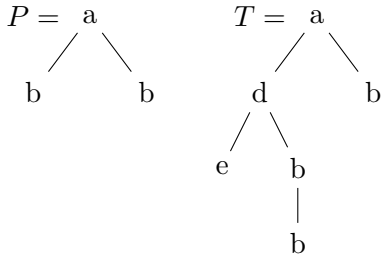
Il existe une hiérarchie entre ces trois définitions d'inclusion et l'inclusion relâchée. Une solution au problème d'inclusion forte est aussi une solution au problème d'inclusion région ordonnée. Une solution au problème d'inclusion région ordonnée est aussi une solution au problème d'inclusion chemin ordonné. Et enfin une solution au problème d'inclusion chemin ordonné est aussi une solution au problème d'inclusion relâchée.

Dans [10], il est montré que le problème d'inclusion d'arbres non ordonnés est NP-complet. Nous abordons plutôt le problème d'inclusion d'arbres ordonnés car nos études sont focalisés sur XML qui a une représentation sous forme d'arbres ordonnés. L'ordre à l'horizontal est intéressante lorsque nous supposons des documents XML soumis à des contraintes d'intégrité où l'ordre horizontal à un rôle.

Un autre algorithme d'inclusion d'arbres ordonnés qui améliore celui proposé dans [10], a été proposé dans le papier de Yangjun Chen et Yibin Chen [5]. Dans ce papier une opération nommée $delete(T, v)$ est définie sur les arbres et consiste à supprimer le nœud v de l'arbre T. Lorsque le nœud v est supprimé, on obtient un nouvel arbre tel que tous les enfants de v seront maintenant les enfants du parent de v dans T. L'opération $delete(T, b)$ appliquée à l'arbre ci-dessous, nous donne le résultat suivant :



L'idée de l'algorithme est de supprimer des nœuds à partir de l'arbre cible T pour obtenir l'arbre pattern P.
Soit les arbres P et T de l'exemple suivant :



Pour savoir si P est inclus relâché dans T, on supprime les nœuds d, e et b de T pour obtenir P.

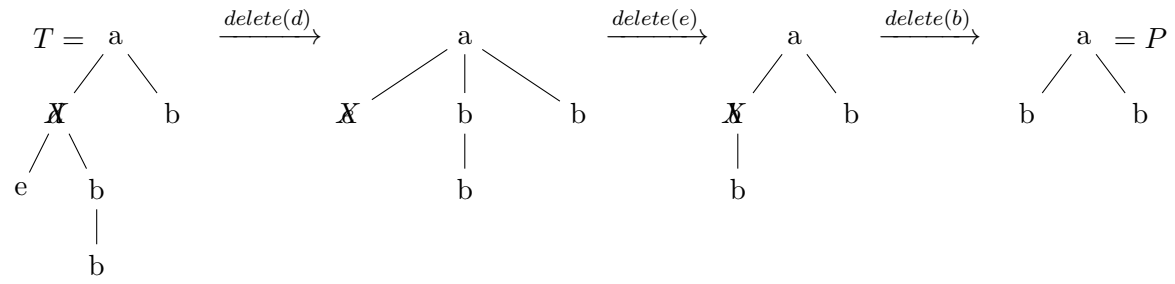


FIGURE 5.2 – Exemple d'inclusion avec suppression de nœuds

Cet algorithme basé sur la suppression des nœuds a une complexité de $O(|T|.min\{DEPTH(P), |leaves(P)|\})$ et est plus performant que celui de Richter.

Le problème d'inclusion relâchée d'arbres apparaît aussi dans le domaine de l'apprentissage. Nous allons aborder maintenant deux algorithmes d'apprentissage sur les arbres ordonnés enracinés et étiquetés.
Le premier qui est décrit dans [13] porte sur l'inclusion d'arbres. Il étudie la fréquence de sous-arbres dans une collection d'arbres par rapport à un certain seuil nommé *min_sup*. Un arbre est fréquent si son support (nombre d'apparitions dans chacun des arbres de la collection) est supérieur à *min_sup* (qui est une donnée du problème). La problématique ici est différente de celle qu'on a vu dans les algorithmes précédents. L'algorithme construit les sous-arbres et garde ensuite ceux qui sont fréquents. La construction des sous-arbres est incrémentale suivant la taille des sous-arbres. Les sous-arbres fréquents de taille $k+1$ sont construits à partir des sous-arbres fréquents de taille k. Une application pour cet algorithme permet de trouver les pages webs les plus visitées d'un site web à partir des données issues des fichiers logs du site web. Ces données sont dans un format XML spécial nommé LOGML. A partir des fichiers logs, on construit des arbres qui correspondent aux différents clics que les utilisateurs ont fait en gardant le lien entre la page à partir duquel le clic a été efféctué et la nouvelle page. Lorsque l'utilisateur fait un retour arrière, on remonte d'un niveau dans l'arbre. L'apprentissage des pages les plus visitées, et des chemins de navigation les plus utilisés pour accéder à ces pages, se fera donc sur l'ensemble des arbres qui a été créé.
Le deuxième algorithme [7] parle de la fréquence de paires de frères ou voisins (notion de cousin entre les nœuds d'un arbre) dans une collection d'arbres non ordonnés. Ce cas ne nous intéresse pas directement.

Conclusion

Nous avons vu dans ce rapport la description de l'algorithme proposé pour la comparaison de langages d'arbres. Nous utilisons cette comparaison pour pouvoir substituer un service web lorsqu'il tombe en panne dans une composition de plusieurs services web. Cette technique permet de gagner du temps en attendant que le service en panne soit réparé, et donc indispensable pour permettre à la composition de continuer à marcher. Pour résumer l'algorithme de comparaison de deux services S_1 et S'_1, comme la technique que nous utilisons est basée sur les arbres binaires, nous transformons les langages d'arbres d'arité non borné (schéma XML) en langage d'arbres binaires $bin(S_1)$, $bin(S'_1)$. Ensuite nous exprimons par un langage synchronisé de couples d'arbres, l'ensemble R de tous les couples d'arbres (t, t') sur un alphabet Σ tel que t soit "inclus relâché dans" t'. Par un algorithme d'intersection, on calcule l'intersection de R avec $bin(S'_1)$. Le nouvel ensemble R' obtenu contient donc les couples (t, t') appartenant à R tel que $t' \in bin(S'_1)$. A partir de R' on calucle $\Pi_1(R')$ qui est la projection régulière sur le premier composant t des couples de R'. Cet ensemble contient les arbres t qui sont "inclus relâché dans" les arbres t' de $bin(S'_1)$. La dernière étape consiste à vérifier que tous les éléments de $bin(S_1)$ se trouvent dans $\Pi_1(R')$. Ceci se fait par une inclusion d'ensemble. Si $bin(S_1) \subseteq \Pi_1(R')$ alors on vient de décider que S_1 est "inclus relâché" dans S'_1.

Les algorithmes ont été implémenté en Prolog. Tous les objectifs du stage ont été rempli. Nous avons obtenu des propriétés intéressantes sur les arbres ce qui nous a permis d'écrire les preuves de l'algorithme qui calcule la relation R.

Perpectives : Nous pouvons citer deux autres façons d'aborder ce problème de substitution de services.

- la première consiste à proposer une extension (minimale, lisible et gardant certaines propriétés) S_2, afin que S_2 puisse accepter les données provenant de S_1 et S'_1. Autrement dit, trouver S_2 tel que $S_1 \cup S'_1 \subseteq S_2$. Plus de détails sur l'extension de schéma dans [3].
- la seconde consiste à travailler directement sur les grammaires d'arité non borné en utilisant la même méthode basée sur les grammaires binaires décrite dans ce rapport.

Bilan : Ce stage m'a apporté des connaisances sur les arbres, les langages d'arbres, et aussi la manipulation des expression régulières. J'ai appris à formuler des théorèmes et à pouvoir écrire les preuves correspondantes. En plus de la théorie, j'ai eu à implémenter les algorithmes pour bien vérifier leurs calculs.

Bibliographie

[1] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theor. Comput. Sci. (TCS)*, 233(1-2) :75–90, 2000.

[2] Jacques Chabin, Jing Chen, and Pierre Réty. Synchronized ContextFree Tree-tuple Languages. Research Report RR-2006-13, LIFO, 2006. Rapport de recherche LIFO.

[3] Jacques Chabin, Mirian Halfeld Ferrari Alves, Pierre Réty, and Martin A. Musicante. Minimal Extensions of Tree Languages : Application to XML Schema Evolution. Research Report RR-2009-06, LIFO, 2009. Rapport de recherche LIFO.

[4] Jérôme Champavère, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren. Efficient Inclusion Checking for Deterministic Tree Automata abd DTDs. In *Second International Conference on Languages and Automata Theory and Applications*, volume 5196 of *LNCS*, pages 184–195. Springer, 2008.

[5] Yangjun Chen and Yibin Chen. A new tree inclusion algorithm. *Information Processing Letters 98(2006) 253-262*, 2006.

[6] Robson da Luz, Mírian Halfeld Ferrari, and Martin A. Musicante. Regular expression transformations to extend regular languages (with application to a datalog XML schema validator). *Journal of Algorithms (Special Issue)*, 62(3-4) :148–167, 2007.

[7] Jason T. L. Wang Dennis Shasha and Sen Zhang. Unordered Tree Mining with Applications to Phylogeny.

[8] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput. 24 340-356*, 1995.

[9] S. Limet and G. Salzer. Basic rewriting via logic programming, with an application to the reachability problem. *Journal of Automata, Languages and Combinatorics*, 2006.

[10] Thorsten Richter. A New Algorithm for the Ordered Tree Inclusion Problem. 1997.

[11] Helmut Seidl. Deciding Equivalence of Finite Tree Automata. 52(2), 1994.

[12] W. Thomas. Automata of infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[13] Mohammed Javeed Zaki. Efficiently mining frequent trees in a forest. In *International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2002.

[14] D Ziadi, J. L. Ponty, and J.M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-deterministe. *Bull. Belg. Math. Soc*, 4 :177–203, 1997.

Annexe A

Proofs of lemmas

Proof of Lemma 3

Let $t_1, t'_1, t_2, t'_2 \in T_\Sigma$, We have to prove : if $t_1 \lhd_b t'_1$ and $t_2 \lhd_b t'_2$ then $t = \overset{\alpha}{\underset{t_1\ t_2}{/\ \backslash}} \lhd_b t' = \overset{\alpha}{\underset{t'_1\ t'_2}{/\ \backslash}}$.

By hypothesis we have the applications $f_1 : Pos(t_1) \to Pos(t'_1)$ and
$f_2 : Pos(t_2) \to Pos(t'_2)$ such that

- $\forall u \in Pos(t_1) : t_1(u) = t'_1(f_1(u))$ and $\forall u \in Pos(t_2) : t_2(u) = t'_2(f_2(u))$,
- $\forall u, v \in Pos(t_1) : u <_b v \Longrightarrow f_1(u) <_b f_1(v)$ and
  $\forall u, v \in Pos(t_2) : u <_b v \Longrightarrow f_2(u) <_b f_2(v)$,
- $\forall u, v \in Pos(t_1) : u \prec_b v \Longrightarrow f_1(u) \prec_b f_1(v)$ and
  $\forall u, v \in Pos(t_2) : u \prec_b v \Longrightarrow f_2(u) \prec_b f_2(v)$.

Let the application $f_3 : Pos(t) \to Pos(t')$ such that

- $f_3(\varepsilon) = \varepsilon$
- $\forall u \in Pos(t_1),\ f_3(0.u) = 0.f_1(u)$
- $\forall v \in Pos(t_2),\ f_3(1.v) = 1.f_2(v)$

For prooving that $t = \overset{\alpha}{\underset{t_1\ t_2}{/\ \backslash}} \lhd_b t' = \overset{\alpha}{\underset{t'_1\ t'_2}{/\ \backslash}}$, we will prove the three goals :

1. $\forall u \in Pos(t) : t(u) = t'(f_3(u))$
2. $\forall u, v \in Pos(t) : u <_b v \Longrightarrow f_3(u) <_b f_3(v)$
3. $\forall u, v \in Pos(t) : u \prec_b v \Longrightarrow f_3(u) \prec_b f_3(v)$

We will analyse all positions in $Pos(t)$.

1. Given a position $w \in Pos(t)$.
   (a) if $w = \epsilon$, we have $t(w) = \alpha$ and $t'(f_3(w)) = t'(\varepsilon) = \alpha$ so $t(w) = t'(f_3(w))$
   (b) if $w = 0.u$, we have $t(w) = t_1(u)$ and
   $t'(f_3(w)) = t'(f_3(0.u)) = t'(0.f_1(u)) = t'_1(f_1(u)) = t_1(u)$ so $t(w) = t'(f_3(w))$
   (c) if $w = 1.v$, we have $t(w) = t_2(v)$ and
   $t'(f_3(w)) = t'(f_3(1.v)) = t'(1.f_2(v)) = t'_2(f_2(v)) = t_2(v)$ so $t(w) = t'(f_3(w))$
2. Given two positions $u, v \in Pos(t)$ such that $u \prec_b v$. By definition 7 we have $u = a.0.b$ and $v = a.1.c$ or $u = a$ and $v = a.1.c$.

(a) if $u = 0.u_1$ and $v = 0.v_1$, then $a = 0.d$ and $u = 0.d.0.b$ or $u = 0.d$. We deduce that $u_1 = d.0.b$ or $u_1 = d$. Otherwise $v = 0.d.1.c$ then $v_1 = d.1.c$ and we have $u_1 \prec_b v_1$ and $f_1(u_1) \prec_b f_1(v_1)$ by hypothesis.
$f_3(u) = 0.f_1(u_1)$ and $f_3(v) = 0.f_1(v_1)$; $f_1(u_1) \prec_b f_1(v_1)$ means that
$f_1(u_1) = a'.0.b'$ or $f_1(u_1) = a'$ and $f_1(v_1) = a'.1.c'$. By setting $a'' = 0.a'$ we have $f_3(u) = a''.0.b'$ or $f_3(u) = a''$ and $f_3(v) = a''.1.c'$ thus $f_3(u) \prec_b f_3(v)$.

(b) if $u = 1.u_1$ and $v = 1.v_1$: it is the same as (a)

(c) if $u = \varepsilon$ and $v = 1.w$, $f_3(u) = \varepsilon$ and $f_3(v) = 1.f_2(w)$ then $f_3(u) \prec_b f_3(v)$.

(d) if $u = 1.u'$ and $v = 0.v'$, it is not possible

(e) if $u = 0.u'$ and $v = 1.v'$, $f_3(u) = 0.f_1(u') \prec_b f_3(u) = 1.f_2(v')$.

3. Given two positions $u, v \in Pos(t)$ such that $u <_b v$. By definition 6 we have $v = u.0.a$.

(a) if $u = 0.u_1$ and $v = 0.v_1$, then $v_1 = u_1.0.a$ and we have $u_1 <_b v_1$ and
$f_1(u_1) <_b f_1(v_1)$ by hypothesis.
$f_3(u) = 0.f_1(u_1)$ and $f_3(v) = 0.f_1(v_1)$; $f_1(u_1) <_b f_1(v_1)$ means that
$f_1(v_1) = f_1(u_1).0.a'$.
We have $f_3(v) = 0.f_1(v_1) = 0.f_1(u_1).0.a' = f_3(u).0.a'$ then we can conclude that $f_3(u) <_b f_3(v)$.

(b) if $u = 1.u_1$ and $u = 1.v_1$: it is the same as (a)

(c) if $u = \varepsilon$ and $v = 0.w$, $f_3(u) = \varepsilon$ and $f_3(v) = 0.f_1(w) = f_3(u).0.f_1(w)$ then $f_3(u) <_b f_3(v)$.

(d) if $u = \varepsilon$ and $v = 1.w$, it is not possible.

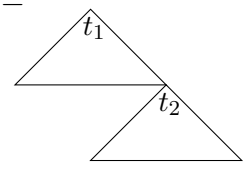
(e) if $v = \varepsilon$ and $u = 1.w$, it is not possible.

(f) if $v = \varepsilon$ and $u = 0.w$, it is not possible.

(g) if $u = 1.u'$ and $v = 0.v'$, it is not possible.

(h) if $u = 0.u'$ and $v = 1.v'$, it is not possible.

Proof of Lemma 4

Let $t_1, t'_1, t_2, t'_2 \in T_\Sigma$, We have to prove : if $t_1 \lhd_b t'_1$ and $t_2 \lhd_b t'_2$ then
$t =$  $\lhd_b t' =$  .

By hypothesis we have the applications $f_1 : Pos(t_1) \to Pos(t'_1)$ and
$f_2 : Pos(t_2) \to Pos(t'_2)$ such that
- $\forall u \in Pos(t_1) : t_1(u) = t'_1(f_1(u))$ and $\forall u \in Pos(t_2) : t_2(u) = t'_2(f_2(u))$,
- $\forall u, v \in Pos(t_1) : u <_b v \Longrightarrow f_1(u) <_b f_1(v)$ and
  $\forall u, v \in Pos(t_2) : u <_b v \Longrightarrow f_2(u) <_b f_2(v)$,
- $\forall u, v \in Pos(t_1) : u \prec_b v \Longrightarrow f_1(u) \prec_b f_1(v)$ and
  $\forall u, v \in Pos(t_2) : u \prec_b v \Longrightarrow f_2(u) \prec_b f_2(v)$.

Let z =rightmost-innermost($Pos(t_1)$)
Let the application $f_3 : Pos(t) \to Pos(t')$ such that

– $\forall u \in Pos(t_1),\ f_3(u) = 0.f_1(u)$
– $\forall v \in Pos(t_2),\ f_3(z.v) = 1.f_2(v)$

For prooving that $t = t_1 \natural t_2 \ \lhd_b \ t' = \begin{array}{c} \alpha \\ / \ \backslash \\ t'_1 \ \ t'_2 \end{array}$, we will prove the three goals :

1. $\forall u \in Pos(t) : t(u) = t'(f_3(u))$
2. $\forall u, v \in Pos(t) : u <_b v \Longrightarrow f_3(u) <_b f_3(v)$
3. $\forall u, v \in Pos(t) : u \prec_b v \Longrightarrow f_3(u) \prec_b f_3(v)$

We will analyse all positions in $Pos(t)$.

1. Given a position $w \in Pos(t)$.
   (a) if $w \in Pos(t_1) \setminus \{z\}$, we have $t(w) = t_1(w)$ and $t'(f_3(w)) = t'(0.f_1(w)) = t'_1(f_1(w)) = t_1(w) = t(w)$ so $t(w) = t'(f_3(w))$
   (b) if $w = z.v$ and $v \in Pos(t_2)$, we have $t(w) = t(z.v) = t_2(v)$ and $t'(f_3(w)) = t'(f_3(z.v)) = t'(1.f_2(v)) = t'_2(f_2(v)) = t_2(v) = t(z.v) = t(w)$ so $t(w) = t'(f_3(w))$
2. Given two positions $u, v \in Pos(t)$ such that $u \prec_b v$. By definition 7 we have $u = a.0.b$ and $v = a.1.c$ or $u = a$ and $v = a.1.c$.
   (a) if $u \in Pos(t_1)$ and $v \in Pos(t_1)$, we have $f_1(u) \prec_b f_1(v)$ by hypothesis.
   $f_1(u) \prec_b f_1(v)$ means that $f_1(u) = a'.0.b'$ or $f_1(u) = a'$ and $f_1(v) = a'.1.c'$.
   $f_3(u) = 0.f_1(u_1)$ and $f_3(v) = 0.f_1(v_1)$; By setting $a'' = 0.a'$ we have $f_3(u) = a''.0.b'$ or $f_3(u) = a''$ and $f_3(v) = a''.1.c'$ thus $f_3(u) \prec_b f_3(v)$.
   (b) if $u = z.u_1$ and $v = z.v_1$, we have $a = z.d$ then $u = z.d.0.b$ or $u = z.d$. We deduce that $u_1 = d.0.b$ or $u_1 = d$. Otherwise $v = z.d.1.c$ then $v_1 = d.1.c$ and we have $u_1 \prec_b v_1$ and $f_2(u_1) \prec_b f_2(v_1)$ by hypothesis.
   $f_3(u) = f_3(z.u_1) = 1.f_2(u_1)$ and $f_3(v) = f_3(z.v_1) = 1.f_2(v_1)$;
   $f_2(u_1) \prec_b f_2(v_1)$ means that $f_2(u_1) = a'.0.b'$ or $f_2(u_1) = a'$ and $f_2(v_1) = a'.1.c'$.
   By setting $a'' = 1.a'$ we have $f_3(u) = a''.0.b'$ or $f_3(u) = a''$ and $f_3(v) = a''.1.c'$ thus $f_3(u) \prec_b f_3(v)$.
   (c) if $u \in Pos(t_1)$ and $v = z.v_1$, we have $f_3(u) = 0.f_1(u)$ and $f_3(v) = f_3(z.v_1) = 1.f_2(v_1)$ then $f_3(u) \prec_b f_3(v)$.
   (d) if $v \in Pos(t_1)$ and $u = z.u_1$, it is not possible.
3. Given two positions $u, v \in Pos(t)$ such that $u <_b v$. By definition 6 we have $v = u.0.a$.
   (a) if $u \in Pos(t_1)$ and $v \in Pos(t_1)$, we have $f_1(u) <_b f_1(v)$ by hypothesis.
   $f_1(u) <_b f_1(v)$ means that $f_1(v) = f_1(u).0.a'$.
   $f_3(u) = 0.f_1(u)$ and $f_3(v) = 0.f_1(v)$;
   We have $f_3(v) = 0.f_1(v) = 0.f_1(u).0.a' = f_3(u).0.a'$ then we can conclude that $f_3(u) <_b f_3(v)$.
   (b) if $u = z.u_1$ and $v = z.v_1$, we have $v = u.0.a = z.u_1.0.a$, with $v = z.v_1$, $v_1 = u_1.0.a$ then $u_1 <_b v_1$ and $f_2(u_1) <_b f_2(v_1)$ by hypothesis.
   $f_2(u_1) <_b f_1(v_1)$ means that $f_2(v_1) = f_2(u_1).0.a'$.
   $f_3(u) = 1.f_2(u_1)$ and $f_3(v) = 1.f_2(v_1)$;
   We have $f_3(v) = 1.f_2(v_1) = 1.f_2(u_1).0.a' = f_3(u).0.a'$ then we can conclude that $f_3(u) <_b f_3(v)$.
   (c) if $u \in Pos(t_1) \setminus \{z\}$ and $v = z.v_1$, it is not possibe.
   (d) if $v \in Pos(t_1) \setminus \{z\}$ and $u = z.u_1$, it is not possible.

Proof of Lemma 6

Let $t, t' \in T_\Sigma$, We have to prove : if $t \lhd_b t'$ et $t' \neq \bot$ then :
- $t \neq \bot$, $t(\epsilon) = t'(\epsilon)$, $t_{|_0} \lhd_b t'_{|_0}$ and $t_{|_1} \lhd_b t'_{|_1}$
- or $\exists\ t_0, t_1 \in T_\Sigma$, $t = t_0 \natural t_1$, $t_0 \lhd_b t'_{|_0}$ and $t_1 \lhd_b t'_{|_1}$

By hypothesis we have the application $f : Pos(t) \to Pos(t')$ such that

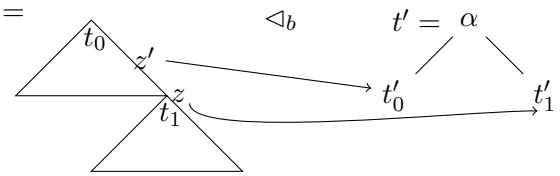
1. $\forall u \in Pos(t) : t(u) = t'(f(u))$
2. $\forall u, v \in Pos(t) : u <_b v \Longrightarrow f(u) <_b f(v)$
3. $\forall u, v \in Pos(t) : u \prec_b v \Longrightarrow f(u) \prec_b f(v)$

We have two cases :

1. If $f(\varepsilon) = \varepsilon$ then we have $t(\varepsilon) = t'(\varepsilon) = \alpha$.
   $t = \begin{array}{c} \alpha \\ / \ \backslash \\ t_{|_0}\ t_{|_1} \end{array}$, $t' = \begin{array}{c} \alpha \\ / \ \backslash \\ t'_{|_0}\ t'_{|_1} \end{array}$.
   Let $t_0 = t_{|_0}$, $t'_0 = t'_{|_0}$, $t_1 = t_{|_1}$, $t'_1 = t'_{|_1}$.
   Since $f(\varepsilon) = \varepsilon$ and f is stable by $<_b$ and $\prec_b$, $\exists\ w, w'$ such that :
   – $\forall u \in Pos(t_0)$, $f(0.u) = 0.w$
   – $\forall v \in Pos(t_1)$, $f(1.v) = 1.w'$
   Therefore, we can define the applications
   $f_0 : Pos(t_0) \to Pos(t'_0)$ and $f_1 : Pos(t_1) \to Pos(t'_1)$ such that :
   – $\forall u \in Pos(t_0)$, $f(0.u) = 0.f_0(u)$
   – $\forall v \in Pos(t_1)$, $f(1.v) = 1.f_1(v)$
   We know that $t \neq \bot$, $t(\epsilon) = t'(\epsilon)$ and for proving $t_{|_0} \lhd_b t'_{|_0}$ and $t_{|_1} \lhd_b t'_{|_1}$ we will prove the three goals :
   (a) $\forall u \in Pos(t_0) : t_0(u) = t'_0(f_0(u))$ and $\forall u \in Pos(t_1) : t_1(u) = t'_1(f_1(u))$,
   (b) $\forall u, v \in Pos(t_0) : u <_b v \Longrightarrow f_0(u) <_b f_0(v)$ and
   $\forall u, v \in Pos(t_1) : u <_b v \Longrightarrow f_1(u) <_b f_1(v)$,
   (c) $\forall u, v \in Pos(t_0) : u \prec_b v \Longrightarrow f_0(u) \prec_b f_0(v)$ and
   $\forall u, v \in Pos(t_1) : u \prec_b v \Longrightarrow f_1(u) \prec_b f_1(v)$.
   We will analyse all positions in $Pos(t_0)$ and $Pos(t_1)$.
   (a) Given a position $w_0 \in Pos(t_0)$ and a position $w_1 \in Pos(t_1)$.
   $t_0(w_0) = t(0.w_0) = t'(f(0.w_0)) = t'(0.f_0(w_0)) = t'_0(f_0(w_0))$.
   $t_1(w_1) = t(1.w_1) = t'(f(1.w_1)) = t'(1.f_1(w_1)) = t'_1(f_1(w_1))$.
   (b) Given two positions $u, v \in Pos(t_0)$ such that $u \prec_b v$. By definition 7 we have $u = a.0.b$ and $v = a.1.c$ or $u = a$ and $v = a.1.c$.
   We have $u' = 0.u \in Pos(t)$ and $v' = 0.v \in Pos(t')$. By setting $a' = 0.a$, $u' = a'.0.b$ or $u' = a'$, and $v' = a'.1.c$, we have $u' \prec_b v'$ and by hypothesis $f(u') \prec_b f(v')$.
   $f(u') \prec_b f(v')$ means that $f(u') = a''.0.b''$ or $f(u') = a''$, and $f(v') = a''.1.c''$.

However $f(u') = f(0.u) = 0.f_0(u)$ and $f(v') = f(0.v) = 0.f_0(v)$. By setting $a'' = 0.a'''$ we have $f_0(u) = a'''.0.b''$ or $f_0(u) = a'''$, and $f_0(v) = a'''.1.c''$ then $f_0(u) \prec_b f_0(v)$.
Idem for $f_1(u) \prec_b f_1(v)$.

(c) Given two positions $u, v \in Pos(t_0)$ such that $u <_b v$. By definition 6 we have $v = u.0.a$
We have $u' = 0.u \in Pos(t)$ and $v' = 0.v \in Pos(t')$.
$v = u.0.a \Rightarrow 0.v = 0.u.0.a \Rightarrow v' = u'.0.a$, hence $u' <_b v'$ and by hypothesis $f(u') <_b f(v')$. $f(u') <_b f(v')$ means that $f(v') = f(u').0.a'$.
However $f(u') = f(0.u) = 0.f_0(u)$ and $f(v') = f(0.v) = 0.f_0(v)$.
We have $f(v') = f(u').0.a' \Rightarrow 0.f_0(v) = 0.f_0(u).0.a' \Rightarrow f_0(v) = f_0(u).0.a'$, hence $f_0(u) <_b f_0(v)$.
Idem for $f_1(u) <_b f_1(v)$.

2. else let $t'_0 = t'_{|_0}$, $t'_1 = t'_{|_1}$, $t = t_0 \natural t_1$ and the position z defined by :
$z = 1^*$, $f(z) = 1.w$ and $\neg(\exists\ z' \in Pos(t),\ f(z') = 1.w',\ z = z'.v$ and $v \neq \varepsilon)$.
$t =$ 

In this case where in t', the position ε don't have an antecedent by f, we have : $\exists\ w, w'$ such that
$\diamond$ $\forall u \in Pos(t_0) \setminus \{z\},\ f(u) = 0.w$
$\circ$ $\forall v \in Pos(t_1),\ f(z.v) = 1.w'$
The proof of that has done in three points :
$\diamond$ if $z = \epsilon$ then $t_0 = \perp$ and $Pos(t_0) \setminus \{z\} = \emptyset$.
$\diamond$ if $z \neq \varepsilon$, we have $z = z'.1$. By definition of z, we have $f(z') \neq 1.w$. Since ε don't have antecedent by f, $f(z') = 0.w'$.
Let $u \in Pos(t_0) \setminus \{z, z'\}$, $u <_b z'$ (resp $u \prec_b z'$) $\Longrightarrow f(u) <_b f(z')$ (resp $f(u) \prec_b f(z')$) then $f(u) = 0.w''$.
$\circ$ $\forall\ v \neq \varepsilon$, $z <_b z.v$ (resp $z \prec_b z.v$) then $f(z) = 1.w <_b f(z.v)$ (resp $f(z) = 1.w \prec_b f(z.v)$) then $f(z.v) = 1.w.w' = 1.w''$.

Therefore, we can define the applications
$f_0 : Pos(t_0) \to Pos(t'_0)$ and $f_1 : Pos(t_1) \to Pos(t'_1)$ such that :
– $\forall u \in Pos(t_0) \setminus \{z\},\ f(u) = 0.w$
– $\forall v \in Pos(t_1),\ f(z.v) = 1.w'$
For proving $t_0 \lhd_b t'_0$ and $t_1 \lhd_b t'_1$, we will prove the three goals :

(a) $\forall u \in Pos(t_0) : t_0(u) = t'_0(f_0(u))$ and $\forall u \in Pos(t_1) : t_1(u) = t'_1(f_1(u))$,

(b) $\forall u, v \in Pos(t_0) : u <_b v \Longrightarrow f_0(u) <_b f_0(v)$ and
$\forall u, v \in Pos(t_1) : u <_b v \Longrightarrow f_1(u) <_b f_1(v)$,

(c) $\forall u, v \in Pos(t_0) : u \prec_b v \Longrightarrow f_0(u) \prec_b f_0(v)$ and
$\forall u, v \in Pos(t_1) : u \prec_b v \Longrightarrow f_1(u) \prec_b f_1(v)$.

We will analyse all positions in $Pos(t_0)$ and $Pos(t_1)$.

(a) Given a position $w_0 \in Pos(t_0) \setminus \{z\}$ and a position $w_1 \in Pos(t_1)$.
$t_0(w_0) = t(w_0) = t'(f(w_0)) = t'(0.f_0(w_0)) = t'_0(f_0(w_0))$.
$t_1(w_1) = t(z.w_1) = t'(f(z.w_1)) = t'(1.f_1(w_1)) = t'_1(f_1(w_1))$.

(b) Given two positions $u, v \in Pos(t_0) \setminus \{z\}$ such that $u \prec_b v$. We have $u, v \in Pos(t_0)$ and then By hypothesis $f(u) \prec_b f(v)$ because $u \prec_b v$.
$f(u) \prec_b f(v)$ means that $f(u) = a.0.b$ or $f(u) = a$ and $f(v) = a.1.c$. However $f(u) = 0.f_0(u)$ and $f(v) = 0.f_0(v)$. By setting $a' = 0.a$ we have $f_0(u) = a'.0.b$ or $f_0(u) = a'$ and $f_0(v) = a'.1.c$ then $f_0(u) \prec_b f_0(v)$.
Idem for $f_1(u) \prec_b f_1(v)$.

(c) Given two positions $u, v \in Pos(t_0) \setminus \{z\}$ such that $u <_b v$. We have $u, v \in Pos(t_0)$ and then By hypothesis $f(u) <_b f(v)$ because $u <_b v$. $f(u) <_b f(v)$ means that $f(v) = f(u).0.a$. However $f(u) = 0.f_0(u)$ and $f(v) = 0.f_0(v)$.
We have $f(v) = f(u).0.a \Rightarrow 0.f_0(v) = 0.f_0(u).0.a \Rightarrow f_0(v) = f_0(u).0.a$, hence $f_0(u) <_b f_0(v)$.
Idem for $f_1(u) <_b f_1(v)$.