Apprentissage de fonctions reconnaissables d'arbres : vers l'apprentissage de transformations d'arbres

Grégoire Laurence

Mémoire de master recherche
Laboratoire d'Informatique Fondamentale de Lille

Directeurs : Sławek Staworko, Aurélien Lemay, Joachim Niehren, et Marc Tommasi
Equipe Mostrare -INRIA Lille Nord-Europe

Table des matières

Remerciement

Je tient à remercier l'ensemble des encadrants de ce stage qui malgrès leur emploi du temps très chargé ont réussi à être disponible à chaque fois que c'est nécessaire. Je remercie surtout Joachim Niehren et Slawek Staworko pour leur patience, le temps qu'ils m'ont consacrés et la confiance qu'il m'ont accordés.

Je remercie l'ensemble de l'équipe Mostrare qui une fois de plus m'a accueilli chaleureusement dans leur rang ainsi que l'inria pour l'environnement de travail qui a été mis à notre disposition.

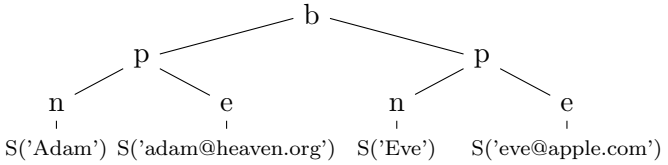
Je remercie Jérome Champavère pour la relecture de ce rapport et l'ensemble des thésards de l'équipe pour leur présence et la bonne ambiance tout au long de ce stage.

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

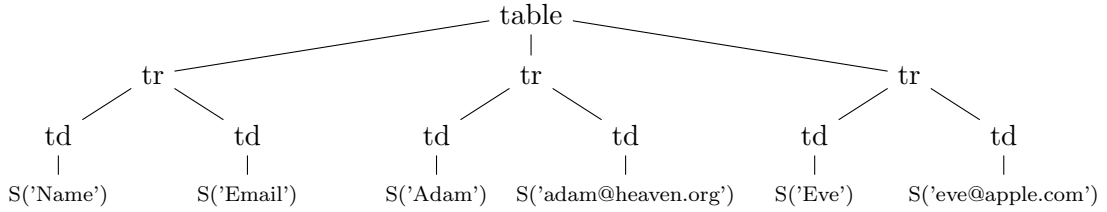
Introduction

Avec l'avancée des technologies liées au web, les données semi-structurées ont un rôle de plus en plus présent. Un des témoins de cette expansion est l'explosion du nombre d'applications traitant le format XML (*eXtensible Markup Language*), et de langages permettant la selection de noeuds (XPath). Ce type de structures s'impose comme un standard pour l'échange de données entre applications ou sur le web. Malgré cela, la grande liberté de structuration laissée par ce format, qui est un de ses points forts, reste un réel frein pour ces échanges. En effet, les documents peuvent être utilisés dans un aspect purement données, ce qui impliquera une structure plus pertinente face aux informations qu'ils contiennent, alors que d'autres ne seront utilisés que dans le but de structurer des documents, auquel cas la structure n'aura qu'un intérêt implicite dans l'interprétation des données. Il est donc indispensable de développer des outils permettant la transformation des documents, qui suivent un modèle qui leur est propre, dans un autre formalisme adapté à leur future utilisation.

Des langages développés dans ce but (comme XSLT, XQuery), ou les langages de programmation habituels permettent de coder ces transformations mais nécessitent une nouvelle définition ad hoc pour chaque source d'information, ce qui est coûteux en temps et nécessite une bonne connaissance de ces langages et des bases de données sur lesquelles on travaille. De plus, ces dépôts évoluant dans le temps, les définitions demanderaient une constante redéfinition.



(a) document XML



(b) un tableau HTML

Fig. 1 – exemple de transformation

Une alternative intéressante pour gérer ce problème serait l'utilisation de *techniques*

d'apprentissage supervisées qui, à partir d'un ensemble d'exemples de transformations, généraliseraient la transformation souhaitée pour l'intégralité d'une base de documents. Cette piste nécessite donc un modèle pouvant représenter ces différents exemples ainsi qu'un outil permettant de les regrouper et de les généraliser. Des solutions ont déjà été apportées dans les cas plus simples ne nécessitant pas une structure arborescente en sortie. C'est le cas de l'adaptation des requêtes monadiques et des requêtes n-aire aux arbres [CGLN07], ce travail étant le point de départ pour notre projet. Les auteurs proposent comme modélisation de ces requêtes, l'annotation des œnuds des arbres par des symboles binaires représentant les éléments choisis par la requête et ceux qui ne le sont pas. Pour reconnaître l'ensemble de ces arbres annotés, ils adaptent le modèle des *automates à pas*, qui sera présenté par la suite, les *transducteurs de sélection de noeuds*. Par la suite, ils utilisent des techniques d'inférence grammaticale pour obtenir un automate fini déterministe à partir des exemples procurés, permettant l'annotation binaire d'arbres d'arité non bornée. Les résultats obtenus motivent la généralisation d'un tel modèle au problème plus complexe qui est la transformation d'arbres. Et les différentes propriétés soulevées, nécessaires à un apprentissage, sont une piste sur les contraintes et les besoins qui s'appliqueront à notre nouveau modèle.

Contributions. Dans cette perspective, nous avons dans un premier temps défini un modèle de transformation d'arbres permettant une représentation et une utilisation simple, tout en essayant de rester le plus complet possible, se basant sur l'idée *d'arbre d'alignements* [ZS89, JWZ94]. Les arbres d'alignements représentent la transformation d'un arbre en un autre sous forme elle aussi arborescente. Il utilise pour cela un ensemble d'opérations de suppression, insertion et renommage des nœuds et permet un large éventail de transformations. Pour permettre la transformation d'un ensemble d'arbres, nous utilisons un *langage régulier d'arbres d'alignements*. Dans une transformation, il est souhaité que pour chaque arbre fourni en entrée, un seul arbre de sortie soit produit. En d'autres termes nous souhaitons pouvoir vérifier la *fonctionnalité* d'un langage régulier d'alignements. Or nous prouvons que cette fonctionnalité est indécidable pour les langages réguliers d'arbres d'alignements. Le modèle tel qu'il est ne peut être utilisé pour définir une transformation.

Nous décidons donc de restreindre les arbres d'alignements aux *superpositions* des arbres d'entrée et des arbres de sortie le plus haut et le plus à gauche possible. La superposition de deux arbres correspond à une relation binaire reconnaissable. Et pour ce modèle nous prouvons que la fonctionnalité peut être vérifiée en temps polynomial.

Perspectives. Le modèle de transformations maintenant défini, il faut adapter un algorithme d'apprentissage à ce langage de transformation. Une des techniques qui paraît appropriée à notre tâche serait l'inférence d'un langage régulier à la limite et plus particulièrement une adaptation de l'algorithme *RPNI* [OG91, OG92] appliqués aux arbres. RPNI construit, à partir d'exemples positifs et négatifs, un automate fini déterministe et le généralise par fusions successives de ses états.

La construction des exemples négatifs nécessaires à cet apprentissage est assez complexe, ce qui nous donne envie d'explorer les sources alternatives pour de tels exemples. Une solution, qui a déjà été utilisée efficacement dans un cas similaire [CLN04], est de remplacer ces exemples par un test d'inclusion du domaine de transformation dans un schéma donnée et de tester à chaque fusion si la fonctionnalité est bien conservée.

Par la suite il faudra essayer d'étendre la classe de transformation tout en gardant la décidabilité de la fonctionnalité. Une première piste serait d'essayer de se limiter aux arbres d'alignements avec la notion de non ambiguïté locale, qu'à chaque étape de la

transformation il n'y ait qu'une unique possibilité de poursuite connaissant le prochain symbole lu en entrée. Ce genre de restriction permettrait de pouvoir tester localement à chaque fusion par l'algorithme RPNI si la non ambiguïté et donc la fonctionnalité est conservée. Une autre piste serait de borner le nombre d'opérations d'insertion qui ne dépendent pas du prochain label du nœud en entrée et qui posent problème.

Travaux relatifs. Dans ce domaine en constante évolution, des solutions commencent à être apportées. C'est le cas de l'annotation d'arbres par des opérations d'édition à l'aide de modèles probabilistes [GJTT06]. Ce modèle, même s'il semble très proche des travaux sur lesquels nous nous basons, restent une branche différente du domaine. Leur approche ne tient compte de la structure arborescente des arbres d'entrées que localement, se limitant aux relations entre un père et deux fils consécutifs pour les annotations, les *cliques*, alors que notre modèle tient compte de l'ensemble de l'arbre parcouru avant d'atteindre le nœud où l'on se trouve. De plus, l'approche probabiliste suppose l'existence d'un grand nombre d'exemples pour chaque label ce qui n'est pas nécessairement notre cas.

De nouvelles classes de transducteurs d'arbres, les *visibly pushdown transducers* [TVY08] sont aussi proposés pour définir les transformations en se basant sur une représentation en flux, une suite de symboles d'ouvertures et de fermeture de nœuds. Ils se rapprochent de l'utilisation d'automates d'arbres et restant de ce fait en marge des transducteurs d'arbres plus classique [CDG$^+$07] étant trop complexes pour une automatisation et une utilisation simple. Ces transducteurs travaillant sur les *visual pushdown languages* [AM04], permettent en plus des symboles d'ouverture et de fermeture, l'utilisation de symboles internes qui sont inutiles pour l'encodage arborescent des documents XML. De plus, ce symbole permet l'expression de langages algébriques (non régulier) par les transformations menant à l'indécidabilité de nombreux problèmes.

Nous pouvons aussi évoquer le *schema matching*, qui même si il ne correspond pas à une transformation d'arbre, définit des relations sur les structures d'entrée et de sortie, ce qui répond au besoin d'échanges et de mise en correspondance des bases de données . Ce genre de techniques reposent surtout sur un typage, une sémantique liée aux données des documents et dépend moins de la structure même de l'arbre. Malgré cela, la définition d'un modèle plus global de transformation d'arbres et de son apprentissage n'est pas encore beaucoup étudiée.

Chapitre 1

Préliminaires

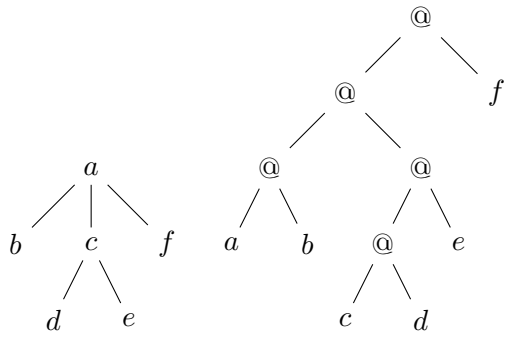
Ce chapitre est consacré à l'introduction du matériel qui sera nécessaire à la bonne compréhension et la cohérence du rapport. Il y sera donc présenté les arbres et les outils disponibles pou leur manipulation ainsi que les différentes normes qui s'y appliquent.

1.1 Arbres

Nous représentons les documents XML par des *arbres ordonnés d'arité non bornée* ayant pour nœuds des labels de l'ensemble fini Σ [Nev02b, Nev02a]. On dénote par T_Σ l'ensemble de tous les arbres définis sur Σ. Ces arbres peuvent être représentés par des termes d'arité non bornée sur l'ensemble Σ.

Un *alphabet d'arité bornée* est un couple $(\Sigma, rank)$ composée d'un ensemble fini Σ contenant les labels des nœuds et de la fonction $rank : \Sigma \to \mathbb{N}$ qui à chaque label de Σ associe son *arité* dans $\mathbb{N}$ l'ensemble des entiers naturels. Dans la plupart des cas, l'arité est définie par le contexte, la notion d'arité bornée sera donc omise et l'alphabet sera dénoté Σ. Pour un alphabet d'arité bornée Σ, t est nommé un *arbre d'arité bornée* sur Σ si pour chaqun de ses nœuds n, si f est le label de ce nœud et que k est l'arité de f alors le nœud n a exactement k fils.

Nous allons maintenant représenter les arbres d'arités bornées en utilisant un des codages binaires standard, le *codage currifié* [CDG$^+$07]. Les arbres seront représentés par des arbres d'arité bornée sur $\Sigma^{@} = \{@\} \cup \Sigma$, où @ est un symbole binaire appelé symbole d'extension et tous les éléments de Σ sont des constantes, i.e. d'arité 0. Quand nous décrirons cette représentation currifiée, nous utiliserons souvent la notation infixe de @. Par exemple la chaîne $a@t_1@t_2$ correspondra à l'arbre $@(@(a, t_1), t_2)$. Nous pouvons passé d'une représentation non ordonnée à un codage binaire (et *vice versa*) en utilisant les deux fonctions suivantes :



(a) arbre d'arité non bornée t (b) codage currifiée de t

FIG. 1.1 – passage d'un arbre t à son codage currifiée

$$\begin{aligned} curry(a) &= a, \qquad a \in \Sigma, \\ curry(a(t_1 \dots t_n)) &= curry(a(t_1 \dots t_{n-1}))@curry(t_n), \end{aligned}$$

$$\begin{aligned} uncurry(a) &= a, \qquad a \in \Sigma, \\ uncurry(t_1@t_2) &= rebuild(t_1, (uncurry(t_2))), \\ rebuild(a, (t_1, \dots, t_n)) &= a(t_1, \dots, t_n), \\ rebuild(t_1@t_2, (t_1, \dots, t_n)) &= rebuild(t_2, (uncurry(t_1), t_1, \dots, t_n)). \end{aligned}$$

1.2 Automates d'arbres d'arité non bornée

Un automate d'arbre fini ascendant (FTA) [CDG+07] A sur Σ est défini par un ensemble fini d'états $states(A)$, et d'un ensemble d'états finaux $final(A) \subseteq states(A)$ et d'un ensemble de règles de transitions $rules(A)$ de la forme $f(q_1, \dots, q_n) \to q$ où $n \geq 0, f \in \Sigma$ d'arité n, et $q, q_1, \dots, q_n \in states(A)$.

Quand $n = 0$, i.e. a est une constante, la règle de transition de la forme $a \to q(a), q \in states(A)$ est appelé une *règle initiale*. L'automate parcourant les arbres des feuilles en remontant jusqu'à la racine est appelé ascendant. Pour un automate d'arbre A défini sur Σ, nous pouvons définir une fonction d'évaluation $eval : T_\sigma \to states(A)$:

$$\begin{aligned} eval_A(a) &= \{q | a \to q \in rules(A)\} \\ eval_A(f(t_1, \dots, t_n)) &= \{q | f(q_1, \dots, q_n) \to q \in rules(A), q - i \in eval_A(t_i) \text{ pour } i \in \{1 \dots n\}\} \end{aligned}$$

L'automate *évalue* l'arbre t dans un ensemble d'états Q si au moins un de ces états appartient à $eval_A(t)$. On peut dénoter par $L_Q(A)$ le langage sur T_Σ, des arbres reconnu aux états composants Q et par $L_q(A)$ la même notion pour un état donnée :

$$\begin{aligned} L_Q(A) &= \{t \in T_\Sigma | eval_A(t) \cap Q \neq \emptyset\}, && \text{pour } Q \subseteq states(A), \\ L_q(A) &= L_{\{q\}}, && \text{pour } q \in states(A). \end{aligned}$$

Un arbre est *reconnu* par l'automate A si il appartient au langage $L_q(A)$ d'un état final $q \in \mathit{final}(A)$. Un automate A défini donc un *langage* d'arbres composé de l'ensemble des arbres qu'il reconnaît :

$$L(A) = L_{\mathit{final}(A)}(A).$$

L'état q d'un automate A est *accessible*, si l'ensemble des arbres évalués dans cet état est non vide, i.e. $L_q(A) \neq \emptyset$. Et il est dit *coaccessible* s'il existe une exécution de A valide passant par q. Un automate A est *émondé* si tous ses états sont à la fois accessibles et coaccessibles. Il est dit *déterministe* s'il n'existe pas deux règles ayant la même partie gauche et une partie droite différente, i.e.

$$\forall f \in \Sigma, \forall q_1, \ldots, q_n \in \mathit{states}(A)\ f(q_1, \ldots, q_n) \to q \wedge f(q_1, \ldots, q_n) \to q' \Rightarrow q = q'.$$

1.2.1 Automates d'arbre à pas

Maintenant nous définissons un modèle d'automate d'arbres reconnaissant les langages d'arbres d'arité non bornées en utilisant la représentation currifiée. Un automate d'arbre *à pas* ascendant sur un alphabet Σ [CNT04] est un automate d'arbre standard défini sur $\Sigma^{@}$. Cet automate est composé d'un ensemble d'états $\mathit{states}(A)$, et des règles de transitions de la forme $a \to q \in \mathit{rules}(A)$ qui à chaque constante $a \in \Sigma$ associe un état $q \in \mathit{states}(A)$, ainsi que des transitions de la forme $q_1@q_2 \to q \in \mathit{rules}(A)$ où $q_1, q_2, q \in \mathit{states}(A)$. Parfois pour une représentation plus proche de l'arborescence nous préférerons la notation préfixée $@(q_1, q_2)$.

On définit la notion d'*exécution* d'un arbre t par un automate à pas déterministe par la succession de différentes étapes d'évaluation permettant d'évaluer l'arbre t dans un état q. Une exécution au sein d'un automate à pas qui reconnaît $t = a@t_1@\ldots@t_n$ à l'état q_i i.e. $t \in L_{q_i}(A)$ peut être représentée par le codage currifié de cet arbre auquel sera ajouté des états $p_i^j, q_i^j \in \mathit{states}(A)$ où $q_i^n = q_i$:

$$(a)^{q_i^0}\ @^{q_i^1}\ (t_1)^{p_i^1}\ @^{q_i^2}\ \ldots\ @^{q_i^n}\ (t_n)^{p_i^n}$$

Les états p_i^j associés aux sous-arbres signifient que $t_j \in L_{p_i^j}(A)$. Et les états associés aux symboles d'extension nous indique qu'il existe les règles $a \to q_i^0$ et que $q_i^{j-1}@p_i^j \to q_i^j$.

1.2.2 Datalog

Datalog est un langage de programmation logique permettant de définir des relations à partir de faits élémentaires et de règles, et permet d'évaluer la complexité de vérification de ces relations. Sa syntaxe est composée d'un ensemble de constantes, de variables et de prédicats. En Datalog les relations sont représentées par des symboles de prédicats et peuvent être sous forme d'un *prédicat* seul ou d'un *atome* $p(x_1, \ldots, x_n)$ composé d'un symbole de prédicat p d'arité n et d'une liste d'arguments $x_i, 1 \leq i \leq n$, appartenant à l'ensemble des constantes et variables.

A chaque relation R, représentée par un prédicat p de l'arité, est associé un ensemble de n-uplets tel que si $(x_1, \ldots, x_n)$ est dans R alors $p(x_1, \ldots, x_n)$ est vrai, sinon il est évalué à faux.

Un programme Datalog est composé d'un ensemble de règles de premier ordre de la forme

$$R_1(u_1) :- R_2(u_2), \ldots, R_n(u_n).$$

avec u_i des n-uplets de constantes et variables tel que toutes les variables de u_1 soient comprises dans un des n-uplets $u_i, 1 < i \leq n$. Cette règle indique que si tous les prédicats

de droite sont vérifiés alors le n-uplet u_1 est ajouté à R_1. S'il n'y a rien à droite la relation R_1 est appelé un *fait* et u_1 est ajouté à R_1.

On applique un mécanisme d'inférence qui à partir des faits enrichit le système avec des nouveaux faits. Un *point fixe* est une étape de l'évaluation tel qu'a l'étape suivante le système reste inchangé. Pour un programme Datalog D, le plus petit point fixe est dénoté par $lfp(D)$.

Théorème 1 *[DEGV97] Pour tout programme Datalog clos D, le plus petit point fixe $lfp(D)$ peut être évalué en temps linéaire $O(|D|)$ où D est le nombre d'atomes dans D.*

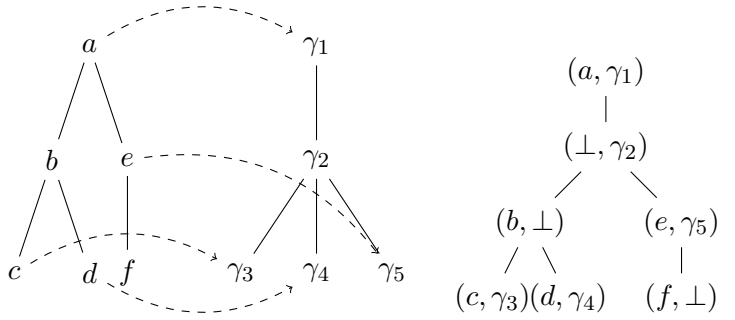
Chapitre 2

Définition de transformation par automates d'arbres

Nous souhaitons définir un modèle de transformation d'arbres ayant les propriétés pour un apprentissage futur dans le but de pouvoir automatiser la transformation de données semi-structurées. Pour cela il faut d'abord trouver une représentation de la transformation d'un arbre en un autre.

2.1 Arbres d'alignements

On propose pour cela une représentation arborescente de cette transformation basé sur l'alignement des données entre l'arbre d'entrée et de sortie, nommé *arbres d'alignement* ou *alignement* (par exemple la figure 2.1). Cette notion d'arbre d'alignement à déjà été introduite pour évaluer la distance d'édition entre deux arbres [ZS89, JWZ94], et étudiée dans divers domaines tel que la bio-informatique [Tou07] mais jamais utilisée comme outil théorique pour la transformation d'arbres. Sa représentation proche de celle des structures sur lesquelles la transformation doivent être effectuées permet une visualisation simple et rapide.



(a) un alignement d'un arbre d'entrée et d'un arbre de sortie

(b) l'arbre d'alignement correspondant

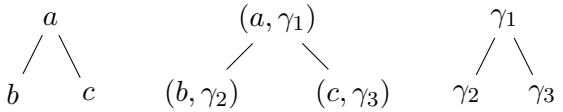
FIG. 2.1 – un exemple pour un arbre d'alignement.

Un alignement représente donc une transformation entre deux arbres que l'on nommera arbre d'entrée et arbre de sortie correspondant à l'arbre passé en paramètre et du résultat de la transformation. Il est défini sur l'alphabet Δ correspondant à $(\Sigma_\perp \times \Gamma_\perp) \setminus \{(\perp, \perp)\}$

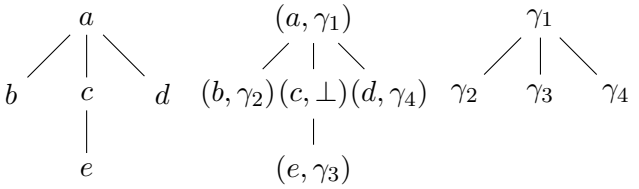
où Σ et Γ sont respectivement les alphabets sur lesquels sont définis l'arbre d'entrée et de sortie, les symboles $\perp$ de nouveaux labels permettant de représenter une notion de vide, et les alphabet ayant ce symbole pour indice correspondent à $\Sigma_\perp = \Sigma \cup \{\perp\}$. En effet si on projette un arbre d'alignement sur l'entrée (resp. la sortie), les symboles $\perp$ correspondront aux nœuds ayant été ajoutés (resp. supprimés) et n'existant pas au sein de cet arbre.

On peut donc voir l'arbre d'alignement comme une application récursive d'opérations d'édition sur les arbres de façon descendante. Ces opérations sont :

- Le renommage qui remplace le précédant label du nœud par un autre sans modifier la structure de l'arbre.



- La suppression qui correspond à un aplatissement de l'arbre. Il supprime le nœud et insère l'ensemble de ses fils à la place qu'il occupait précédemment.



- Et l'insertion correspondant à l'opération inverse de la suppression, à savoir qu'un nœud se crée entre un autre nœud et un sous groupe de ses fils pouvant être vide.

Definition 1 Un arbre d'alignement α est un arbre d'arité non bornée, défini sur Δ tel que sa racine dénote une opération de renommage.

Grâce à ces notions d'opérations d'édition nous pouvons facilement comprendre comment ôter les symboles $\perp$ dans les projections respectives. On peut définir les fonctions $in : T_\Delta \to T_\Sigma$ et $out : T_\Delta \to T_\Gamma$ directement sur l'arbre d'alignement pour retrouver l'arbre d'entrée et de sortie correspondant à cet alignement. Ces fonctions sont définie par :

$$
\begin{aligned}
in((a,\gamma_1)(t_1,t_2,\ldots,t_n)) &= a(in(t_1)\ldots in(t_n)),\\
out((a,\gamma_1)(t_1,t_2,\ldots,t_n)) &= \gamma_1(out(t_1)\ldots out(t_n)),\\
in((a,\perp)(t_1,t_2,\ldots,t_n)) &= a(in(t_1)\ldots in(t_n)),\\
out((a,\gamma_1)(t_1,t_2,\ldots,t_n)) &= out(t_1)\ldots out(t_n),\\
in((\perp,\gamma_1)(t_1,t_2,\ldots,t_n)) &= in(t_1)\ldots in(t_n),\\
out((\perp,\gamma_1)(t_1,t_2,\ldots,t_n)) &= \gamma_1(out(t_1)\ldots out(t_n)).
\end{aligned}
$$

Ces fonctions peuvent produire des *haies* qui est liste d'arbres, un arbre pouvant être décrit comme des haies enracinées.

Remarque *La limitation au renommage pour la racine de l'arbre d'alignement permet d'assurer que l'entrée et la sortie sont bien des arbres.*

Une transformation peut être représentée par un ensemble d'arbres d'alignements, reconnus par un *langage d'alignement.*

Definition 2 Un *langage d'alignement* T est défini tel que :

$$T(t) = \{\tau | \alpha \in T, in(\alpha) = t \wedge out(\alpha) = \tau\}$$

Pour que ce langage représente réellement une transformation, nous avons besoin qu'il respecte la propriété de *fonctionnalité*, i.e. que pour tout arbre d'entrée donné, il existe au plus un arbre résultant de la transformation.

Definition 3 Un langage de transformation T est dit *fonctionnelle* si et seulement si quelque soit $t \in T_\Sigma$, il existe au plus un arbre τ tel que $T(t) = \tau$.

En s'attardant sur le problème de fonctionnalité sur les langages d'alignement qui vient d'être soulevé, on s'aperçoit de la complexité apportée par les symboles $\perp$ internes. En effet les nœuds responsables des opérations d'insertions $((\perp, \gamma))$ ne dépendent de l'entrée que par les nœuds qui les succèdent et les précèdent et empêchent donc d'évaluer la fonctionnalité localement.

2.2 Indécidabilité de la fonctionnalité

Pour prouver l'indécidabilité de la fonctionnalité des transformations définies par un langage d'alignement, on va se ramener au problème de décidabilité de la non ambiguïté d'une grammaire *context-free* que l'on sait indécidable [LP81].

2.2.1 Définitions

Une grammaire algébrique G est définie par un ensemble de symboles non terminaux N et un ensemble de symboles terminaux Σ et d'un symbole source $s_0 \in N$. On suppose que l'ensemble des symboles terminaux et non terminaux sont disjoints, i.e. $N \cap \Sigma = \emptyset$. La grammaire contient aussi un ensemble de règles de transition $trans(G)$ de la forme $R \subseteq N \times N^* \cup N \times N$. Notez que $(a, b_1, ..., b_n) \in trans(G)$ sera représenté par $a \to b_1...b_n$ pour une lecture plus simple.

Un *arbre syntaxique* d'un mot $w \in \Sigma^*$ suivant une grammaire G, est un arbre sur $T_{N \cup \Sigma}$ tel que :

1. les nœuds internes ont un label inclus dans N,
2. les feuilles ont un label appartenant à Σ,
3. pour chaque nœud interne il existe une relation de $trans(G)$ décrivant la relation entre ce nœud et ses enfants,
4. la racine a un label s_0.

La fonction définie sur les arbres, en particulier sur les arbres syntaxiques $yield(t) = w$, où $yield(t)$ est une chaîne sur Σ^* obtenu par la concaténation des labels aux feuilles de t. $parse_G(w)$ dénote l'ensemble des arbres syntaxiques de w par la grammaire G.

Un *langage* $L(G)$ défini par une grammaire G est l'ensemble des mots w sur l'alphabet Σ tel que $parse_G(w) \neq \emptyset$.

Une grammaire algébrique G est *non ambiguë* s'il existe un unique arbre syntaxique pour tout mot $w \in L(G)$.

2.2.2 Indécidabilité

En réduisant le problème de décision de la non ambiguïté d'une grammaire algébrique connu indécidable [LP81] au problème de fonctionnalité d'une transformation d'arbres nous montrons que le test de la fonctionnalité est indécidable lui aussi.

Pour cela, nous pouvons définir un automate d'arbre bottom-up A_G reconnaissant les arbres d'alignement transformant l'ensemble des chaînes $w \in L(G)$ que l'on considérera

comme une haie de symboles de Σ, auquels on ajoute une racine r, dans un des arbres de parse suivant la grammaire G comme définie précédemment. Cet automate A_G est un automate d'arbre ascendant défini sur Δ où :

- $\Delta = \{(r,r)\} \cup \{(\bot,k)|\ k \in N\} \cup \{(a,a)|\ a \in \Sigma\}$, r est un symbole dédié tel que $r \notin N \cup \Sigma$
  $\Delta_0 = \{(a,a)|a \in \Sigma\}$
  $\Delta_k = \{(\bot,M)|M \to M_1,...,M_k \in \mathit{trans}(G)\} \cup \{(r,r)|k=1\}$
- $\mathit{states}(A_G) = \{q_0\} \cup \{q_k|\ k \in (\mathit{trans}(G) \cup \Sigma)\}$
- $\mathit{final}(A_G) = \{q_0\}$
- et l'ensemble des relations de $\mathit{rules}(A_G)$ consiste en :
  - $((\bot,M)(q_{M_1},...,q_{M_k}) \to q_M)$ pour $M \to M_1...M_k$,
  - $(a,a) \to q_a$ pour $a \in \Sigma$,
  - $(r,r)(q_{s_0}) \to q_0$.

Théorème 2 *Tester la fonctionnalité d'une transformation d'arbres est indécidable.*

Lemme 1 *Pour toute chaîne $w \in \Sigma^*$, tout arbre $t \in T_{N\cup\Sigma}$ et tout arbre d'alignement $\alpha \in T_\Delta$ nous avons :*
$w \in L(G) \wedge t \in \mathit{parse}_g(w) \Leftrightarrow \alpha \in L(A_G) \wedge \mathit{in}(\alpha) = r(w) \wedge \mathit{out}(\alpha) = r(t)$.

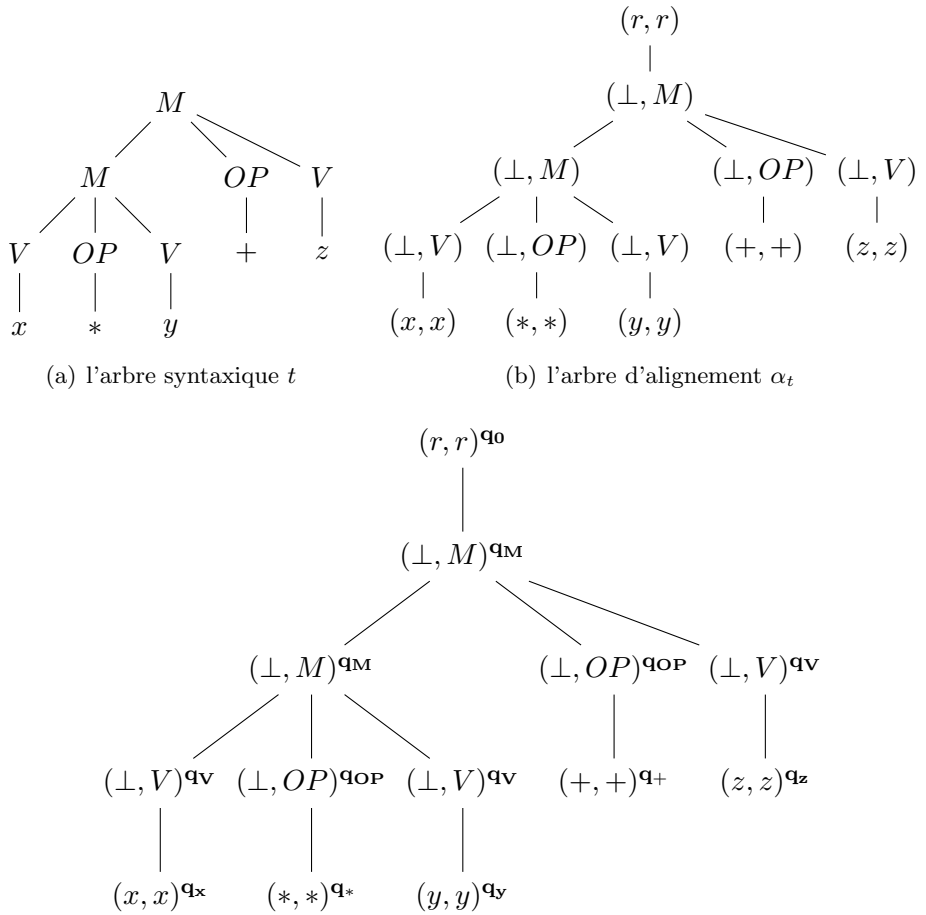
Preuve:
$\boxed{\Rightarrow}$ Si nous prenons un arbre syntaxique t (comme par exemple dans la figure 2.2(a)) d'une chaîne $w \in \Sigma^*$ et un arbre d'alignement $\alpha \in T_{\Sigma'}$ (comme par exemple dans la figure 2.2(b)) tel que $\mathit{in}(\alpha) = r(w)$ et $\mathit{out}(\alpha) = r(t)$, alors α doit être reconnu par A_G.

En effet, pour obtenir $r(t)$ en sortie, donc sur $T_{\Sigma'}$, nous devons remplacer, dans t, chaque nœud interne $M \in N$ par le symbole $(\bot, M)$ et chaque feuille a par le couple (a,a), sans oublier d'ajouter le symbole lié à la racine (r,r). A part cet arbre α, aucun autre arbre ne peut avoir une sortie $r(t)$. L'arbre d'entrée de cet arbre d'alignement correspond bien à $r(w)$.

Il faut que $\alpha \in L(A_G)$, i.e. il existe une exécution de l'automate A_G reconnaissant cet arbre d'alignement (exemple dans la figure 2.2(c)). Chaque règle de G étant traduite dans une transition de A_G et les états finaux étant conservés, il existe une exécution acceptée de l'automate A_G tel que l'arbre α défini ci-dessus soit reconnu.

$\boxed{\Leftarrow}$ Nous prenons α, un arbre d'alignement accepté par une exécution de A_G ; α a donc pour feuilles des couples (a,a) avec $a \in \Sigma$, pour nœuds internes $(\bot, M)$ avec $M \in N$ et pour racine le symbole (r,r). L'arbre d'entrée $\mathit{in}(\alpha)$ est bien de la forme $t(w)$ avec $w \in \Sigma^*$. De plus, chaque transitions de A_G étant construite en suivant les règles de la grammaire G, $(\bot,M)(q_{M_1},...,q_{M_k}) \to q_M$ implique donc qu'il existe une règle $M \to M_1...M_k \in \mathit{trans}(G)$. Le seul état final étant q_0 pour que l'arbre soit accepté nous avons automatiquement que $(r,r)(q_{s_0}) \to q_0$, avec q_{s_0} l'état initial de G, est vérifié.

L'arbre de sortie obtenu par $\mathit{out}(\alpha)$ est un arbre de la forme $r(t)$ tel que t a pour racine q_{s_0}, chaque nœud interne appartient à N et vérifie une relation de $\mathit{trans}(G)$ décrivant la relation entre ce nœud et ses fils, et chaque feuille est dans Σ. Sachant de plus que $\mathit{yield}(t) = w$, nous avons bien que $t = \mathit{parse}_g(w)$. □



(a) l'arbre syntaxique t

(b) l'arbre d'alignement α_t

(c) une exécution valide de A_G sur t

FIG. 2.2 – Exemple d'arbre syntaxique et sa représentation

Chapitre 3

Relations reconnaissables d'arbres

On se limite à une *superposition* de l'arbre d'entrée et de sortie correspondant à la superposition de ces deux arbres le plus en haut et le plus à gauche (voir Figure 3.1(b)). Cette superposition correspond aux relations reconnaissables dans les arbres [CDG$^+$07, GNT08].

L'arbre de superposition est dénoté par l'opérateur binaire infixe $\circledast : T_\Sigma \times T_\Gamma \to T_\Delta$ défini par :

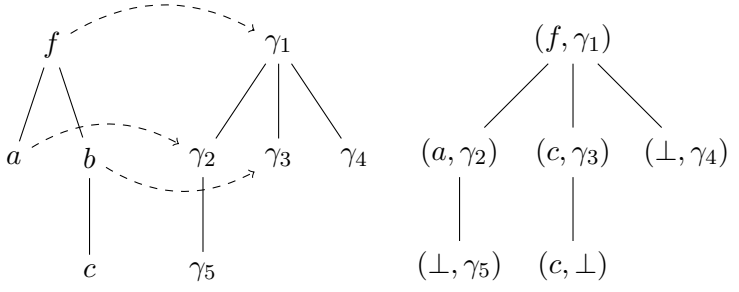
$$a(t_1, \cdots, t_j) \circledast \gamma(\tau_1, \cdots, \tau_k) = \begin{cases} (a,\gamma)(t_1 \circledast \tau_1, \cdots, t_j \circledast \tau_j, \bot \circledast \tau_{j+1}, \cdots, \bot \circledast \tau_k) & \text{si } j < k, \\ (a,\gamma)(t_1 \circledast \tau_1, \cdots, t_k \circledast \tau_k, t_{k+1} \circledast \bot, \cdots, tj \circledast \bot) & \text{sinon,} \end{cases}$$

$$\bot \circledast \gamma(\tau_1, \cdots, \tau_n) = (\bot,\gamma)(\bot \circledast \tau_1, \cdots, \bot \circledast \tau_n),$$

$$a(t_1, \cdots, t_n) \circledast \bot = (a,\bot)(t_1 \circledast \bot, \cdots, t_n \circledast \bot).$$

Dans ce cas, il ne peut avoir plus d'un arbre d'alignement correspondant à un couple d'arbres entrée-sortie. Ce couple est représenter par $t \circledast \tau$. Par la suite nous aurons besoin d'alignements représentant une étape du parcours de l'arbre au quel cas l'un des deux arbres composant la superposition peut être "vide", composé uniquement de symboles $\bot$ ce qui sera dénoté par $t \circledast \bot$ ou $\bot \circledast \tau$. Bien entendu $\bot \circledast \bot = \bot$.

Pour décider de la fonctionnalité d'un langage d'alignement limité aux superpositions il suffit donc de vérifier la non ambiguïté de l'automate le reconnaissant, i.e. que le langage reconnaît au plus un arbre d'alignement pour un arbre d'entrée donné.



(a) arbre d'entrée et de sortie (b) superposition

FIG. 3.1 – exemple de superposition

Pour définir la non fonctionnalité sur un automate d'arbres d'alignements A nous pouvons passer par la définition de prédicats dont un prédicat *non-fonctionnel* et de clauses

les définissant. En inférant ces clauses pour l'automate A, nous pouvons calculer son plus petit point fixe tel que si le prédicat *non-fonctionnel* appartient à ce point fixe alors l'automate est non fonctionnel.

Pour définir ce prédicat *non-fonctionnel* nous passons par le prédicat $o_{\neq}$ (*different output*) définissant la non fonctionnalité de façon locale entre deux états. En d'autres termes on a $o_{\neq}(q_1, q_2)$ pour $q_1, q_2 \in \mathit{states}(A)$ s'ils reconnaissent chacun un arbre d'alignement α_1 et α_2 tel que pour une même entrée, ces arbres produisent une sortie différente. Si cette notion est vérifié entre deux états finaux nous avons *non-fonctionnel*.

$$\mathit{non\text{-}fonctionnel} \Leftrightarrow \exists q_1, q_2 \in \mathit{final}(A) A, o_{\neq}(q_1, q_2)$$
$$o^A_{\neq}(q_1, q_2) = \{(t \circledast \tau_1, t \circledast \tau_2) \mid t \circledast \tau_1 \in L(q_1), t \circledast \tau_2 \in L(q_2), \tau_1 \neq \tau_2\}$$
$$A \models o_{\neq}(q_1, q_2) \Leftrightarrow o^A_{\neq}(q_1, q_2) \neq \emptyset$$

Cette notion de non fonctionnalité locale peut être divisé en deux sous-parties disjointes. D'une part la non fonctionnalité *faible*, apporté par les arbres d'alignements dit *indépendant de l'entrée* (de la forme $\perp \circledast \tau$) qui modifient la sortie sans tenir compte l'entrée. Fonctionnalité qui pour être propagée, nécessite une notion d'*ordre* sur l'arbre de sortie qu'il faudra définir. D'autre part la non fonctionnalité *stricte* qui est gardé tant que les arbres adjoints ont une même entrée. Le prédicat définissant cette égalité sur l'entrée sera également nécessaire.

Il faut introduire le prédicat $\mathit{left}_{\perp}$ pour reconnaître les alignements indépendants de l'entrée, défini par :

$$\mathit{left}^A_{\perp}(q) = \{\alpha = \perp \circledast \tau | \alpha \in L_q(A)\}$$
$$A \models \mathit{left}_{\perp}(q) \Leftrightarrow \mathit{left}^A_{\perp}(q) \neq \emptyset$$

Pour la fonctionnalité faible il est nécessaire d'avoir une notion d'ordre sur les arbres. Pour un alphabet Σ et deux arbres $t, t' \in T_\Sigma$, on dit que α est *inférieur* à α' uniquement s'il existe un ensemble d'arbres $t_1, \ldots, t_n \in T_\Sigma$ tel que t peut être étendu par $t_1 @ \cdots @ t_n$ pour obtenir t'.

$$\forall t, t' \in T_\Sigma, t < t' \Leftrightarrow \exists t_1, \ldots, t_n \in T_\Sigma, n > 0, t' = t @ t_1 @ \cdots @ t_n$$

En étendant cet ordre sur les arbres d'alignements, nous obtenons la notion de $o_<$ (*smaller output*), correspondant à la non fonctionnalité faible, pour un automate A définie par :

$$o^A_<(q_1, q_2) = \{(t \circledast \tau_1, t \circledast \tau_2) \mid \tau_1 < \tau_2 \wedge t \circledast \tau_1 \in L_{q_1}(A) \wedge t \circledast \tau_2 \in L_{q_2}(A)\}$$
$$A \models o_<(q_1, q_2) \Leftrightarrow o^A_<(q_1, q_2) \neq \emptyset$$

On peut dans la lancée parler de la relation $o_>$ (*larger output*) définissant la même propriété que $o_<$ mais dans l'ordre inverse et qui peut être facilement défini par :

$$A \models o_>(q_1, q_2) = o_<(q_2, q_1)$$

Pour la non fonctionnalité, il est nécessaire que l'entrée soit toujours identique, d'où le besoin d'un prédicat $i_=$ dénotant cette égalité et pouvant être définie entre deux états q_1, q_2 de l'automate A par :

$$i^A_=(q_1, q_2) = \{(t \circledast \tau_1, t \circledast \tau_2) \mid t \circledast \tau_1 \in L_{q_1}(A), t \circledast \tau_2 \in L_{q_2}(A)\}$$
$$A \models i_=(q_1, q_2) \Leftrightarrow i^A_=(q_1, q_2) \neq \emptyset$$

La non fonctionnalité stricte entre deux états q_1, q_2 de l'automate A, que l'on appellera $o_{s\neq}(q_1, q_2)$, peut être maintenant définie à partir des autres prédicats par :

$$A \models o_{s\neq}(q_1, q_2) = o_{\neq}(q_1, q_2) \wedge \neg o_{<}(q_1, q_2) \wedge \neg o_{>}(q_1, q_2)$$

Maintenant nous allons décrire les différentes clauses nécessaire à la création du plus petit point fixe. Les clauses sont générés par des observations sur les états et les transitions définies dans l'automate A.

3.1 Indépendance à l'entrée

Dans cette partie nous pouvons inférer le plus petit point fixe contenant l'ensembles des états respectant le prédicat $left^A_\perp(q)$ donc reconnaissant au moins un arbre d'alignement $\perp \circledast \tau$. Ce prédicat peut être inféré indépendement des autres prédicats. Nous dénotons par $lfp(D(A))$ le plus petit point fixe du programme Datalog D défini par les clauses suivantes pour un automate A.

$$\frac{(\perp, \gamma) \rightarrow q}{left_\perp(q).}$$

La projection de l'arbre $(\perp, b) \in L_q(A)$ sur l'entrée produit un arbre $\perp$ défini sur $\in T_{\{\perp\}}$. On a bien $(\perp, b) \in left^A_\perp(q)$.

$$\frac{@(q_1, q_2) \rightarrow q}{left_\perp(q) :- \ left_\perp(q_1), left_\perp(q_2).}$$

$A \models left_\perp(q_1)$ et $A \models left_\perp(q_2)$ nous indiquent qu'il existe les arbres de la forme $(\perp \circledast \tau_1) \in L_{q_1}(A)$ et $(\perp \circledast \tau_2) \in L_{q_2}(A)$, l'ensemble $rules$(A) contenant la transition suivante $@(q_1, q_2) \rightarrow q$ nous pouvons en déduire qu'il existe un arbre $(\perp \circledast \tau_1 @ \tau_2) \in L_q(A)$ correspondant au prédicat $left^A_\perp(q)$.

Il faut maintenant montrer que le prédicat inféré correspond à sa définition pour l'automate A.

Lemme 2 $A \models left_\perp(q) \Leftrightarrow left_\perp(q) \in lfp(D(A))$.

Preuve:
($\Leftarrow$) Comme il est vérifié à chaque ajout d'une clause, $left_\perp(q) \in lfp(D(A))$ implique que $A \models left_\perp(q)$.

($\Rightarrow$) Prenons un arbre d'alignement α tel que $\pi_1(\alpha) \in T_{\{\perp\}}$ et l'exécution de A qui le reconnais :

$$\alpha = (\perp, \gamma_i)^{q_i^0} \ @^{q_i^1} \ \alpha_1^{p_i^1} \ @^{q_i^2} \ \ldots \ @^{q_i^n} \ \alpha_n^{p_i^n}$$

Dans cette exécution, nous savons que le nombre de nœuds composant chaque sous-arbre est strictement inférieur au nombre de nœuds composant l'arbre α. De plus, si α n'est composé que de symboles $\perp$ dans sa projection sur l'entrée alors ses sous-arbres partagent la même propriété, i.e. $\pi_1(\alpha) \in T_{\{\perp\}} \Rightarrow \pi_1(\alpha_i) \in T_{\{\perp\}}$.

Si α n'est composé que d'un nœud, la règle suivante s'applique :

$$\frac{(\perp, b) \rightarrow q}{left_\perp(q).}$$

Sinon, on va prouver l'équivalence par induction sur la taille de α. Pour $weight(\alpha) = k$, si le prédicat est vérifié pour tous les arbres de taille inférieure à k, ce qui implique que $left^A_\perp(p^j_i) \in lfp(D(A))$ alors par induction sur la règle suivante nous obtenons que $left_\perp(q) \in lfp(D(A))$:

$$\frac{@(q_1, q_2) \to q}{left_\perp(q) :- left_\perp(q_1), left_\perp(q_2).}$$

□

3.2 Entrées identiques

Dans cette partie, nous allons inférer le prédicat $i_=$ dénotant l'existence de deux arbres ayant une entrée identique dans les langages de deux états d'un automate A. Comme précédemment $lfp(D(A))$ correspond au plus petit point fixe du programme Datalog D défini par les clauses suivantes pour un automate A.

La notion de $i_=$ entre deux états, sans compter l'automate A, dépend uniquement du prédicat $left_\perp$, préalablement évalué, et se définie grâce aux règles suivantes :

$$\frac{(a, \gamma_1) \to q_1 \qquad (a, \gamma_2) \to q_2 \qquad a \in \Sigma\ \gamma_1, \gamma_2 \in \Gamma_\perp}{i_=(q_1, q_2).}$$

Les transitions montrent que $(a \circledast \gamma_1) \in L_{q_1}(A)$ et $(a \circledast \gamma_2) \in L_{q_2}(A)$ donc $((a \circledast \gamma_1), (a \circledast \gamma_2)) \in i^A_=(q_1, q_2)$.

$$\frac{@(q_1, q_2) \to q_3 \qquad @(q'_1, q'_2) \to q'_3}{i_=(q_3, q'_3) :- i_=(q_1, q'_1), i_=(q_2, q'_2).}$$

Si $i^A_=(q_1, q'_1)$ et $i^A_=(q_2, q'_2)$ sont vérifiés alors il existe les arbres $(t \circledast \tau_1) \in L_{q_1}(A), (t \circledast \tau'_1) \in L_{q'_1}(A), (t' \circledast \tau_2) \in L_{q_2}(A)$, et $(t' \circledast \tau'_2) \in L_{q'_2}(A))$.

Les transitions décrites si dessus appartenant à $rules(A)$, nous avons $(t@t' \circledast \tau_1@\tau_2) \in L_{q_3}(A)$ et $(t@t' \circledast \tau'_1@\tau'_2) \in L_{q'_3}(A)$. Ces alignements, ayant un même arbre d'entrée, appartiennent à l'ensemble $i^A_=(q_3, q'_3)$.

$$\frac{@(q_1, q_2) \to q}{\begin{array}{c} i_=(q_1, q) :- left_\perp(q_2). \\ i_=(q', q) :- i_=(q', q_1), left_\perp(q_2). \end{array}}$$

$left^A_\perp(q_2)$ implique que $\alpha_2 = (\perp, \tau_2) \in L_{q_2}(A)$.

Donc pour tout alignement $\alpha_1 = (t \circledast \tau_1) \in L_{q_1(A)}$, comme $\alpha_1@\alpha_2 \in L_q(A)$, on a $(t \circledast \tau_1, t \circledast \tau_1@\tau_2) \in i^A_=(q_1, q)$.

Si de plus, le prédicat $i^A_=(q', q_1)$ est vérifiée alors il existe un arbre $t' \in T_{\Sigma_\perp}$ tel que $(t' \circledast \tau', t' \circledast \tau'_1) \in i^A_=(q', q_1)$. Comme pour tout alignement $\alpha \in L_{q_1}(A)$ la relation $i^A_=(q, q_1)$ est vérifié alors c'est vrai aussi pour $t' \circledast \tau'_1 \in L_{q_1}(A)$ donc $(t' \circledast \tau', t' \circledast \tau) i^A_=(q', q)$.

$$\frac{q_1, q_2 \in states(A)}{\begin{array}{c} i_=(q_1, q_1). \\ i_=(q_1, q_2) :- i_=(q_2, q_1). \end{array}}$$

Pour tout alignement $(t \circledast \tau) \in L_{q_1(A)}$, nous avons $(t \circledast \tau, t \circledast \tau) \in i^A_=(q_1, q_1)$.

Et $(t \circledast \tau, t \circledast \tau') \in i^A_=(q, q')$ avec $t \circledast \tau \in L_q(A)$ et $t \circledast \tau' \in L_{q'}(A)$ implique que $(t \circledast \tau', t \circledast \tau) \in i^A_=(q', q)$

Lemme 3 $A \models i_{=}(q_1, q_2) \Leftrightarrow i_{=}(q_1, q_2) \in \mathit{lfp}(D(A))$.

Preuve: ($\Leftarrow$)cf. la construction des règles Datalog.
($\Rightarrow$) Prenons deux alignements $t \circledast \tau_1, t \circledast \tau_2$ partageant le même arbre d'entrée $t = a(t_1, \ldots, t_n)$, avec $\tau_1 = \gamma_1(\tau_1^1, \ldots, \tau_1^{n_1})$ et $\tau_2 = \gamma_2(\tau_2^1, \ldots, \tau_2^{n_2})$. Grâce à la règle de commutativité, nous pouvons supposer, sans perte de généralité, que $n_1 \leq n_2$.

$$\frac{q_1, q_2 \in \mathit{states}(A)}{i_{=}(q_1, q_2) :- \ i_{=}(q_2, q_1).}$$

Pour l'exécution de A reconnaissant l'arbre α_i :

$$\alpha = (\bot, \gamma_i)^{q_i^0} \ @^{q_i^1} \ \alpha_1^{p_i^1} \ @^{q_i^2} \ \ldots \ @^{q_i^{m_i}} \ \alpha_n^{p_i^n}$$

$\tau_i^j = \bot$ pour tout $j > n_i$ et $i \in \{1, 2\}$,et si $j > n$ alors $t_j = \bot$. Définissons par m_i la taille de l'arbre d'alignement, i.e. $m_i = \max(n, n_i)$. Notons que $m_1 \leq m_2$ puisque $n_1 \leq n_2$.
Prenons $j < m_2$ le rang auquel on se trouve pendant l'exécution sur α_1 et α_2.
Si $m_2 = 1$, la règle suivante peut être appliquée :

$$\frac{(a, \gamma_1) \to q_1^0 \qquad (a, \gamma_2) \to q_2^0 \qquad a \in \Sigma \ \gamma_1, \gamma_2 \in \Gamma_\bot}{i_{=}(q_1^0, q_2^0).}$$

Sinon, Nous sommes à l'état j. Cela implique que $i_{=}(q_1^j, q_2^j) \in \mathit{lfp}(D(A))$.
Soit $i_{=}(p_1^{j+1}, p_1^{k+1}) \in \mathit{lfp}(D(A))$ et la notion de $i_{=}^A$ est propagée par la règle suivante :

$$\frac{@(q_1^j, p_1^{j+1}) \to q_1^{j+1} \qquad @(q_2^j, p_2^{j+1}) \to q_2^{j+1}}{i_{=}(q_1^{j+1}, q_2^{j+1}) :- \ i_{=}(q_1^j, q_2^j), i_{=}(p_1^{j+1}, p_2^{j+1}).}$$

Soit les états p_1^i pour $j < i < m_2$ sont tels que $\mathit{left}_\bot(p_1^i) \in \mathit{lfp}(D(A))$, ce que l'on peut inférer pour tout i d'après le lemme 2. Donc nous pouvons appliquer les règles suivantes :

$$\frac{@(q_2^{k-1}, p_2^k) \to q_2^k}{\begin{array}{c} i_{=}(q_2^{k-1}, q_2^k) :- \ \mathit{left}_\bot(p_2^k). \\ i_{=}(q_1^j, q_2^k) :- \ i_{=}(q_1^j, q_2^{k-1}), \mathit{left}_\bot(p_2^k). \end{array}}$$

En itérant sur cette clause (récursion sur k), on obtient que $i_{=}(q_1^j, q_2^k) \in \mathit{lfp}(D(A))$ pour tout $j < k < m_2$. Ce qui grâce à la règle de symétrie suivante équivaut à $i_{=}(q_2^k, q_1^j) \in \mathit{lfp}(D(A))$:

$$\frac{q_1, q_2 \in \mathit{states}(A)}{i_{=}(q_1, q_2) :- \ i_{=}(q_2, q_1).}$$

Ce qui donne en itérant sur $j < k < m_1$ sur la règle suivante que $i_{=}(q_1^k, q_2^{m_2}) \in \mathit{lfp}(D(A))$:

$$\frac{@(q_1^{k-1}, p_1^k) \to q_1^k}{\begin{array}{c} i_{=}(q_1^{k-1}, q_1^k) :- \ \mathit{left}_\bot(p_1^k). \\ i_{=}(q_2^{m_2}, q_1^k) :- \ i_{=}(q_2^{m_2}, q_1^{k-1}), \mathit{left}_\bot(p_1^k). \end{array}}$$

Et plus précisément que $i_{=}(q_1^{m_1}, q_2^{m_2}) \in \mathit{lfp}(D(A))$. □

3.3 Non fonctionnalité faible

Maintenant que l'ensemble des états q respectant $left^A_\perp(q)$ et les paires d'états q_1, q_2 tel que $i_=(q_1, q_2)$ sont connus, il nous est possible d'inférer les relations d'ordre $o_<$ et $o_>$ correspondant à la non fonctionnalité faible dépendant de la notion d'ordre précédemment définie.

$$\frac{@(q_1, q_2) \to q}{\begin{array}{c} o_<(q_1, q) :- \mathit{left}_\perp(q_2). \\ o_<(q', q) :- o_<(q', q_1), \mathit{left}_\perp(q_2). \end{array}}$$

Pour $(\perp \circledast \tau_2) \in left^A_\perp(q_2)$, la transition $@(q_1, q_2) \to q$ étant comprise dans l'ensemble $rules(A)$ alors pour tout $t_1 \circledast \tau_1 \in L_{q_1}A$ nous avons $(t_1 \circledast \tau_1, t_1 \circledast \tau_1 @ \tau_2) \in o^A_<(q_1, q)$ car $t_1 @ \perp$ est équivalent à t_1.

Si de plus $(t' \circledast \tau', t' \circledast \tau' @ \tau'_1) \in o^A_<(q', q_1)$, comme pour tout alignement $\alpha \in L_{q_1}(A)$ la relation précédente est vérifié alors c'est vrai aussi pour $t' \circledast \tau' @ \tau'_1 \in L_{q_1}A$ donc $(t' \circledast \tau', t' \circledast \tau' @ \tau'_1 @ \tau'_2) \in o^A_<(q', q)$.

$$\frac{q_1, q_2 \in states(A)}{o_>(q_1, q_2) :- o_<(q_2, q_1).}$$

La relation $o_>$ est par définition l'inverse de $o_<$ donc $o_<(q_2, q_1) \Rightarrow o_>(q_1, q_2)$.

Nous devons maintenant prouver que ce prédicat inférer recouvre le même ensemble que par définition.

Lemme 4 $A \models o_<(q_1, q_2) \Leftrightarrow o_<(q_1, q_2) \in lfp(D(A))$.

Preuve: ($\Leftarrow$) Cette implication se vérifie récursivement sur la création des clauses.

($\Rightarrow$) La preuve se fait par induction sur la structure de t tel qu'il existe deux alignements $t \circledast \tau_1, t \circledast \tau_2$ partageant le même arbre d'entrée $t = a(t_1, \ldots, t_n)$, avec $\tau_1 = \gamma_1(\tau^1_1, \ldots, \tau^{n_1}_1)$ et $\tau_2 = \gamma_1(\tau^1_1, \ldots, \tau^{n_1}_1, \tau^{n_1+1}_2, \ldots, \tau^{n_2}_2)$. Donc nous avons $n_2 > n_1$.

Pour une exécution de A reconnaîssant α_i :

$$(a, \gamma_i)^{q^0_i} \ @^{q^1_i} \ (t_1 \circledast \tau^1_i)^{p^1_i} \ @^{q^2_i} \ \ldots \ @^{q^{m_i}_i} \ (t_{m_i} \circledast \tau^{m_i}_i)^{p^{m_i}_i}$$

Prenons $j = n_1$. L'automate étant déterministe, les états $q^{n_1}_1$ et $q^{n_1}_2$ sont identiques car ils reconnaissent le même alignement $t \circledast \tau_1$. Pour que les deux arbres τ_1, τ_2 gardent le même arbre d'entrée il faut que chaque état p^k_2 pour $n_1 < k < n_2$ soit tel que $left_\perp(p^i_1) \in lfp(D(A))$ ce que nous pouvons inférer pour tout i d'après le lemme 2. Nous pouvons donc appliquer la règle suivante :

$$\frac{@(q^{k-1}_2, p^k_2) \to q^k_2}{\begin{array}{c} o_<(q^{k-1}_2, q^k_2) :- \mathit{left}_\perp(p^k_2). \\ o_<(q^{n_1}_1, q^k_2) :- o_<(q^{n_1}_1, q^{k-1}_2), \mathit{left}_\perp(p^k_2). \end{array}}$$

En appliquant récursivement cette règle pour $n_1 < k < n_2$ on obtient donc que $o_<(q^{n_1}_1, q^{n_2}_2) \in lfp(D(A))$. □

3.4 Non fonctionnalité

Grâce à tous les prédicats définis précédemment et à la notion de non fonctionnalité stricte qui va être définie ici, nous pouvons inférer la non fonctionnalité d'un automate A grâce aux clauses composant le programme Datalog D suivant :

$$\frac{(a,\gamma_1) \to q_1 \qquad (a,\gamma_2) \to q_2 \qquad \gamma_1 \neq \gamma_2 \qquad a \in \Sigma_\perp \ \gamma_1,\gamma_2 \in \Gamma_\perp}{o_{\neq}(q_1,q_2).}$$

D'après la règle de transition $(a \circledast \gamma_1) \in L_{q_1}(A)$ et $(a \circledast \gamma_2) \in L_{q_1}(A)$, avec la propiété que $\gamma_1 \neq \gamma_2$ ce qui implique que $((a \circledast \gamma_1),(a \circledast \gamma_2)) \in o^A_{\neq}(q_1,q_2)$.

$$\frac{(a,\gamma_1) \to q_1 \qquad (a,\gamma_2) \to q_2 \qquad \gamma_1 \neq \gamma_2 \qquad a \in \Sigma \ \gamma_1,\gamma_2 \in \Gamma_\perp}{o_{s\neq}(q_1,q_2).}$$

Comme montré ci dessus, les conditions impliquent que $((a \circledast \gamma_1),(a \circledast \gamma_2)) \in o^A_{\neq}(q_1,q_2)$. De plus, γ_2 ne peut pas s'écrire à partir de γ_1, (resp. pour l'inverse) donc $((a \circledast \gamma_1),(a \circledast \gamma_2)) \notin o^A_{<}(q_1,q_2)$ (resp. $o^A_{>}(q_1,q_2)$), ce qui implique que $((a \circledast \gamma_1),(a \circledast \gamma_2)) \in o^A_{s\neq}(q_1,q_2)$.

$$\frac{q_1,q_2 \in \mathit{states}(A)}{\begin{array}{c} o_{\neq}(q_1,q_2) :- \ o_{\neq}(q_2,q_1). \\ o_{s\neq}(q_1,q_2) :- \ o_{s\neq}(q_2,q_1). \\ o_{\neq}(q_1,q_2) :- \ o_{s\neq}(q_1,q_2). \\ o_{\neq}(q_1,q_2) :- \ o_{<}(q_1,q_2). \end{array}}$$

$((t \circledast \gamma_2),(t \circledast \gamma_1)) \in o^A_{\neq}(q_2,q_1)$ implique que $(t \circledast \gamma_1) \in L_{q_1}(A), (t \circledast \gamma_2) \in L_{q_2}(A)$ et $\gamma_1 \neq \gamma_2$ donc on a bien $((t \circledast \gamma_1),(t \circledast \gamma_2)) \in o^A_{\neq}(q_1,q_2)$. $o_{\neq}$ est symétrique.

$A \models o_{s\neq}(q_2,q_1)$ implique que $o^A_{\neq}(q_2,q_1)$ n'est pas vide alors que $o^A_{<}(q_2,q_1)$ et $o^A_{>}(q_2,q_1)$ le sont. Comme montré précédemment, $o_{\neq}$ est symétrique, de plus $smo^A(q_2,q_1) = lgo^A(q_1,q_2)$ pour tout $q_1,q_2 \in \mathit{states}(A)$ ce qui nous donne que $o^A_{\neq}(q_1,q_2)$ n'est pas vide alors que $o^A_{<}(q_1,q_2)$ et $o^A_{>}(q_1,q_2)$ le sont, donc $A \models o_{s\neq}(q_1,q_2)$ est vérifié.

$(t \circledast \tau_1, t \circledast \tau_2) \in o^A_{<}(q_1,q_2)$ implique que $\tau_1 < \tau_2$ donc τ_1 est différent de τ_2, avec $t \circledast \tau_1 \in L_{q_1}(A)$ et $\tau_2 \in L_{q_2}(A)$. L'appartenance $(t \circledast \tau_1, t \circledast \tau_2) \in o^A_{\neq}(q_1,q_2)$ est vérifiée.

$(\alpha_1,\alpha_2) \in o^A_{\neq}(q_2,q_1)$.

$$\frac{@(q_1,q_2) \to q_3 \qquad @(q'_1,q'_2) \to q'_3}{\begin{array}{c} o_{s\neq}(q_3,q'_3) :- \ o_{s\neq}(q_1,q'_1), i_{=}(q_2,q'_2). \\ o_{s\neq}(q_3,q'_3) :- \ i_{=}(q_1,q'_1), o_{s\neq}(q_2,q'_2). \end{array}}$$

Nous avons $(t' \circledast \tau_2, t' \circledast \tau'_2) \in i_{=}(q_2,q'_2)$ et $(t \circledast \tau_1, t \circledast \tau'_1) \in o^A_{s\neq}(q_1,q'_1)$ donc $\alpha_3 = (t@t' \circledast \tau_1@\tau_2) \in L_{q_3}(A)$ et $\alpha'_3 = (t@t' \circledast \tau'_1@\tau'_2) \in L_{q'_3}(A)$.

Cela implique également que $(\tau_1 \neq \tau'_1)$, et plus particulièrement qu'il n'existe pas d'ensemble $\{\tau_3 \dots \tau_n, \tau_2\}$ tel que $\tau_1 = \tau'_1@\tau_3 \dots @\tau_n@\tau_2$ ou que $\tau'_1 = \tau_1@\tau_3 \dots @\tau_n@\tau_2$.

Nous pouvons en déduire que $(\alpha_3,\alpha'_3) \notin o^A_{<}(q_3,q'_3)$ ainsi que $(\alpha_3,\alpha'_3) \notin o^A_{>}(q_3,q'_3)$ et que $\tau_1@\tau_2 \neq \tau'_1@\tau'_2$ donc $(\alpha_3,\alpha'_3) \in o^A_{s\neq}(q_3,q'_3)$.

Pour la deuxième règle nous avons $(t \circledast \tau_1, t' \circledast \tau'_1) \in i_{=}(q_1, q'_1)$ et $(t \circledast \tau_2, t \circledast \tau'_2) \in o^A_{s\neq}(q_2, q'_2)$. En suivant les règles de transition nous en déduisons que $\alpha_3 = (t@t' \circledast \tau_1@\tau_2) \in L_{q_3}(A)$ et $\alpha'_3 = (t@t' \circledast \tau'_1@\tau'_2) \in L_{q'_3}(A)$. Sachant que $\tau_2 \neq \tau'_2$ alors pour tout $\tau \in T_\Gamma$, particulièrement τ_1 et τ'_1, nous avons $\tau_1@\tau_2 \neq \tau'_1@\tau'_2$ donc $(\alpha_3, \alpha'_3) \in o^A_{\neq}(q_3, q'_3)$. De plus, il n'existe pas d'ensemble $\{\tau_3, \ldots, \tau_n\}$ tel que $\tau_2 = \tau'_2@\tau_3@\ldots@\tau_n$ ou que $\tau'_2 = \tau_2@\tau_3@\ldots@\tau_n$, ce qui reste le cas pour $\tau_1@\tau_2$ et $\tau'_1@\tau'_2$ ce qui implique $(\alpha_3, \alpha'_3) \notin o^A_{<}(q_3, q'_3)$ ainsi que $(\alpha_3, \alpha'_3) \notin o^A_{>}(q_3, q'_3)$. Ce qui nous donne $(\alpha_3, \alpha'_3) \in o^A_{s\neq}(q_3, q'_3)$.

$$\frac{@(q, q_1) \to q_2 \qquad @(q, q'_1) \to q'_2}{o_{s\neq}(q_2, q'_2) :- \; o_{\neq}(q_1, q'_1).}$$

Nous avons $(t_1 \circledast \tau_1, t_1 \circledast \tau'_1) \in o^A_{\neq}(q_1, q'_1)$ donc $\tau_1 \neq \tau'_1$ et pour tout $(t \circledast \tau) \in L_q A$, nous obtenons $\tau@\tau_1 \neq \tau@\tau'_1$. Donc pour $\alpha = t@t' \circledast \tau@\tau_1 \in L_{q_2(A)}$ et $\alpha' = t@t' \circledast \tau@\tau'_1 \in L_{q_2(A)}$ nous avons $(\alpha, \alpha') \in o_{\neq}(q_2, q'_2)$. Comme $\tau_1 \neq \tau'_1$, pour tout $\tau_3, \ldots \tau_n$, $\tau@\tau_1 \neq \tau@\tau'_1@\tau_3@\ldots@\tau_n$ et $\tau@\tau'_1 \neq \tau@\tau_1@\tau_3@\ldots@\tau_n$ ce qui implique que $(\alpha, \alpha') \notin o^A_{<}(q_3, q'_3)$ et $(\alpha, \alpha') \notin o^A_{>}(q_3, q'_3)$ pour n'importe quel couple (α, α') construit comme ci-dessus. $(\alpha, \alpha') \in o^A_{s\neq}(q_2, q'_2)$ est donc vérifié.

$$\frac{@(q_1, q_2) \to q}{o_{s\neq}(q', q) :- \; o_{s\neq}(q', q_1), \mathit{left}_{\perp}(q_2).}$$

$\mathit{left}^A_{\perp}(q_2) \Rightarrow \perp \circledast \tau_2 \in L_{q_2}(A)$ donc pour tout alignement $\in L_q(A)$ et plus particulièrement $\alpha_1 = (t' \circledast \tau_1)$ il existe l'alignement $\alpha = (t' \circledast \tau_1@\tau_2)L_q(A)$.

De plus $(t' \circledast \tau', t' \circledast \tau_1) \in o^A_{s\neq}(q', q_1)$, donc $o^A_{>}(q', q_1) = \emptyset$ impliquant que $\tau' \neq \tau_1@\tau_3 \ldots @\tau_n$ pour tout $\tau_3 \ldots \tau_n \in T_\Gamma$ ce qui est vrai aussi pour τ_2 soit $\tau' \neq \tau_1@\tau_2$. $(\alpha', \alpha) \in o^A_{\neq}(q', q)$ est vérifié. Cela implique aussi l'inégalité $\tau' \neq \tau_1@\tau_2 \ldots \tau_n \in T_\Gamma$, donc $(\alpha', \alpha) \notin o_{>}(q', q)$. Si $o^A_{<}(q', q) \neq \emptyset$,alors il existe $\tau_3 \ldots \tau_n$ tel que $\tau_1@\tau_2 = \tau'@\tau_3 \ldots @\tau_n$, donc $\tau_2 = \tau_n$ et $\tau_1 = \tau'@\tau_3 \ldots @\tau_{n-1}$. Ce qui est impossible car $o^A_{<}(q', q_1) = \emptyset$. On vérifie donc que $o^A_{<}(q', q) = \emptyset$, clôturant les besoins pour que $(\alpha', \alpha) \in o_{s\neq}(q', q)$ soit validé.

Lemme 5 *Prenons deux états $q_1, q_2 \in \mathit{states}(A)$,*
$A \models o_{s\neq}(q_1, q_2) \Leftrightarrow o_{s\neq}(q_1, q_2) \in \mathit{lfp}(D(A))$.

$$\frac{q \in \mathit{final}(A) \qquad q' \in \mathit{final}(A)}{\mathit{non\text{-}fonctionnel} :- \; o_{\neq}(q, q'))}$$

Soit A un automate d'arbre à pas déterministe ascendant, qui reconnaît des arbres d'alignement $t \circledast \tau$ qui couplent des entrées $t \in T_\Sigma$ et des sorties $\tau \in T_\Gamma$.

Preuve: ($\Leftarrow$) L'implication est conservée par toutes les règles Datalog.

($\Rightarrow$) La preuve se fait par induction sur la structure de t.

Prenons deux alignements $t \circledast \tau_1, t \circledast \tau_2$ partageant le même arbre d'entrée $t = a(t_1, \ldots, t_n)$, avec $\tau_1 = \gamma_1(\tau^1_1, \ldots, \tau^{n_1}_1)$ et $\tau_2 = \gamma_2(\tau^1_2, \ldots, \tau^{n_2}_2)$. Grâce à la règle de commutativité, nous pouvons supposer, sans perte de généralité, que $n_1 \leq n_2$.

$\tau^j_i = \perp$ pour tout $j > n_i$ et $i \in \{1, 2\}$, et $t_j = \perp$ $j > n$. Définissons par m_i la taille de l'arbre d'alignement, i.e. $m_i = \max(n, n_i)$. Notons que $m_1 \leq m_2$ depuis que $n_1 \leq n_2$.

$$\frac{q_1, q_2 \in \mathit{states}(A)}{o_{s\neq}(q_1, q_2) : - o_{s\neq}(q_2, q_1).}$$

Pour une exécution de A reconnaîssant α_i :

$$(a,\gamma_i)^{q_i^0}\ @^{q_i^1}\ (t_1 \circledast \tau_i^1)^{p_i^1}\ @^{q_i^2}\ \ldots\ @^{q_i^{m_i}}\ (t_{m_i} \circledast \tau_i^{m_i})^{p_i^{m_i}}$$

Prenons $0 \leq j \leq n_1$ pour la taille minimale tel que $\gamma_1(\tau_1^1, \ldots \tau_1^j) \neq \gamma_2(\tau_2^1, \ldots, \tau_2^j)$. Premièrement nous vérifions que $o_{s\neq}(q_1^j, q_2^j) \in \mathit{lfp}(D(A))$.

Si $j = 0$ alors $\gamma_1 \neq \gamma_2$ et la règle (diff-labels) peut être appliqué.

$$\frac{(a,\gamma_1) \to q_1^0 \in \mathit{rules}(A) \qquad (a,\gamma_2) \to q_2^0 \in \mathit{rules}(A) \qquad \gamma_1 \neq \gamma_2}{o_{s\neq}(q_1^0, q_2^0).}$$

Sinon, $\gamma_1(\tau_1^1, \ldots, \tau_1^{j-1}) = \gamma_2(\tau_2^1, \ldots, \tau_2^{j-1})$ ce qui implique $q_1^{j-1} = q_2^{j-1}$ puisque A est déterministe. De plus, $\tau_1^j \neq \tau_2^j$. Si $\tau_1^j \ s{\neq}\ \tau_2^j$ alors $o_{s\neq}(q_1^j, q_2^j) \in \mathit{lfp}(D(A))$ peut être inféré par hypothèse d'induction (pourvu que t_j est une sous structure correcte de t). Autrement $\tau_1^j < \tau_2^j$ ou $\tau_2^j < \tau_1^j$, ce qui implique $o_<(q_1^j, q_2^j) \in \mathit{lfp}(D(A))$ (resp. $o_<(q_2^j, q_1^j) \in \mathit{lfp}(D(A))$) suivant 4.

Dans tous les cas, une des clauses Datalog suivantes peut être appliquée.

$$\frac{p_1^{j-1}, p_2^{j-1} \in \mathit{states}(A)}{\begin{array}{l} o_{\neq}(p_1^{j-1}, p_2^{j-1}) :- \ o_<(p_1^{j-1}, p_2^{j-1}). \\ o_{\neq}(p_1^{j-1}, p_2^{j-1}) :- \ o_<(p_2^{j-1}, p_1^{j-1}). \\ o_{\neq}(p_1^{j-1}, p_2^{j-1}) :- \ o_{s\neq}(p_1^{j-1}, p_2^{j-1}). \end{array}}$$

Cela implique que $o_{\neq}(p_1^{j-1}, p_2^{j-1}) \in \mathit{lfp}(D(A))$ donc la règle suivante peut s'appliquer :

$$\frac{q_1^{j-1}@p_1^j \to q_1^j \in \mathit{rules}(A) \qquad q_2^{j-1}@p_2^j \to q_2^j \in \mathit{rules}(A) \qquad q_1^{j-1} = q_2^{j-1}}{o_{s\neq}(q_1^j, q_2^j) :- \ o_{\neq}(p_1^{j-1}, p_2^{j-1}).}$$

Par conséquent $o_{s\neq}(q_1^j, q_2^j) \in \mathit{lfp}(D(A))$.

Maintenant nous vérifions que $o_{s\neq}(q_1^k, q_2^k) \in \mathit{lfp}(D(A))$ pour tout $j+1 \leq k \leq m_1$. En remarquant que le prédicat $i_=(p_1^k, p_2^k) \in \mathit{lfp}(D(A))$ est vérifié pour tout k par le lemme 3, nous pouvons appliquer la règle suivante :

$$\frac{q_1^{k-1}@p_1^k \to q_1^k \in \mathit{rules}(A) \qquad q_2^{k-1}@p_2^k \to q_2^k \in \mathit{rules}(A)}{o_{s\neq}(q_1^k, q_2^k) : -o_{s\neq}(q_1^{k-1}, q_2^{k-1}), i_=(p_1^k, p_2^k).}$$

L'itération sur cette clause Datalog (induction sur k) montre que $o_{s\neq}(q_1^k, q_2^k) \in \mathit{lfp}(D(A))$ pour tout $j \leq k \leq m_1$.

Il reste à montrer que $o_{s\neq}(q_1^{m_1}, q_2^k) \in \mathit{lfp}(D(A))$ pour tout $m_1 < k \leq m_2$. Un tel k n'existe que si $n < n_2$. Par conséquent $t_k = \bot$ donc le lemme 2 nous apporte que $\mathit{left}_\bot(p_2^k) \in \mathit{lfp}(D(A))$. La règle :

$$\frac{q_2^{k-1}@p_2^k \to q_2^k \in \mathit{rules}(A)}{o_{s\neq}(q_1^{m_1}, q_2^k) :- \ o_{s\neq}(q_1^{m_1}, q_2^{k-1}), \mathit{left}_\bot(p_2^k).}$$

Par induction sur k, l'application de cette clause Datalog nous montre que $o_{s\neq}(q_1^{m_1}, q_2^k) \in \mathit{lfp}(D(A))$ pour tout $m_1 < k \leq m_2$ ce qui implique que $o_{s\neq}(q_1^{m_1}, q_2^{m_2}) \in \mathit{lfp}(D(A))$, ce qui équivaut à $o_{s\neq}(q_1, q_2) \in \mathit{lfp}(D(A))$. □

3.5 Rappel des règles

$$\frac{(\bot,\gamma)\to q}{left_\bot(q).} \qquad \frac{@(q_1,q_2)\to q}{left_\bot(q) :- left_\bot(q_1), left_\bot(q_2).}$$

$$\frac{(a,\gamma_1)\to q_1 \qquad (a,\gamma_2)\to q_2 \qquad a\in\Sigma\ \gamma_1,\gamma_2\in\Gamma_\bot}{i_=(q_1,q_2).}$$

$$\frac{@(q_1,q_2)\to q_3 \qquad @(q_1',q_2')\to q_3'}{i_=(q_3,q_3') :- i_=(q_1,q_1'), i_=(q_2,q_2').}$$

$$\frac{@(q_1,q_2)\to q}{\begin{array}{c} i_=(q_1,q) :- left_\bot(q_2). \\ i_=(q',q) :- i_=(q',q_1), left_\bot(q_2). \end{array}}$$

$$\frac{q_1,q_2\in states(A)}{\begin{array}{c} i_=(q_1,q_1). \\ i_=(q_1,q_2) :- i_=(q_2,q_1). \end{array}}$$

$$\frac{@(q_1,q_2)\to q}{\begin{array}{c} o_<(q_1,q) :- left_\bot(q_2), \\ o_<(q',q) :- o_<(q',q_1), left_\bot(q_2). \end{array}}$$

$$\frac{q_1,q_2\in states(A)}{o_>(q_1,q_2) :- o_<(q_2,q_1).}$$

$$\frac{(a,\gamma_1)\to q_1 \qquad (a,\gamma_2)\to q_2 \qquad \gamma_1\neq\gamma_2 \qquad a\in\Sigma_\bot\ \gamma_1,\gamma_2\in\Gamma_\bot}{o_{\neq}(q_1,q_2).}$$

$$\frac{(a,\gamma_1)\to q_1 \qquad (a,\gamma_2)\to q_2 \qquad \gamma_1\neq\gamma_2 \qquad a\in\Sigma\ \gamma_1,\gamma_2\in\Gamma_\bot}{o_{s\neq}(q_1,q_2).}$$

$$\frac{q_1,q_2\in states(A)}{\begin{array}{c} o_{\neq}(q_1,q_2) :- o_{\neq}(q_2,q_1). \\ o_{s\neq}(q_1,q_2) :- o_{s\neq}(q_2,q_1). \\ o_{\neq}(q_1,q_2) :- o_{s\neq}(q_1,q_2). \\ o_{\neq}(q_1,q_2) :- o_<(q_1,q_2). \end{array}}$$

$$\frac{@(q_1,q_2)\to q_3 \qquad @(q_1',q_2')\to q_3'}{\begin{array}{c} o_{s\neq}(q_3,q_3') :- o_{s\neq}(q_1,q_1'), i_=(q_2,q_2'). \\ o_{s\neq}(q_3,q_3') :- i_=(q_1,q_1'), o_{s\neq}(q_2,q_2'). \end{array}}$$

$$\frac{@(q,q_1)\to q_2 \qquad @(q,q_1')\to q_2'}{o_{s\neq}(q_2,q_2') :- o_{\neq}(q_1,q_1').}$$

$$\frac{@(q_1,q_2)\to q}{o_{s\neq}(q',q) :- o_{s\neq}(q',q_1), left_\bot(q_2).}$$

$$\frac{q\in final(A) \qquad q'\in final(A)}{non\text{-}fonctionnel :- o_{\neq}(q,q'))}$$

L'ensemble de ces règles nous permet de décider de la fonctionnalité des représentations currifiées de superpositions d'arbres reconnus par un automate à pas en temps polynomial puisqu'elles sont évaluable par une successions de programmes Datalog. On peut donc s'intéresser maintenant au problème de son apprentissage.

Chapitre 4

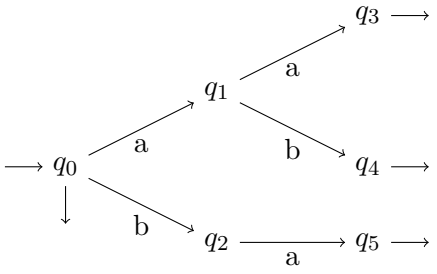
Apprentissage de fonctions reconnaissables

Nous allons nous intéresser ici aux moyens d'apprendre ce langage d'alignements. Une des possibilités qui nous est offerte est l'apprentissage par inférence grammatical à partir d'exemples de transformation. Et plus précisément nous allons nous intéresser à l'algorithme d'apprentissage RPNI [OG92] d'abord développé pour l'apprentissage de langage de mots.

4.1 Apprentissage de mots

Nous allons présenter l'algorithme **RPNI** pour l'apprentissage d'un langage L de mots définis sur Σ^*. L'algorithme RPNI prend en entrée un ensemble d'exemples positifs $S^+ = \{aa, \varepsilon, ab, ba\}$ et négatifs $S^- = \{a, aaa, b\}$.

A partir de l'ensemble S^+, on crée l'arbre *préfixe*, un automate de forme arborescente reconnaissant l'ensemble des mots en les regroupant par préfixe commun :

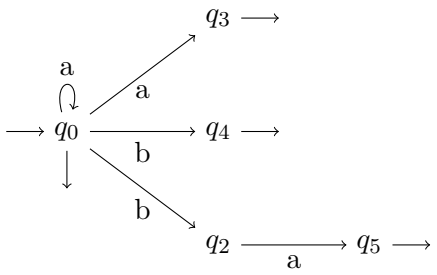


Les états initiaux sont représentés par une flèche entrante et les états finaux par une flèche sortante et une transition par une flèche reliant deux états.

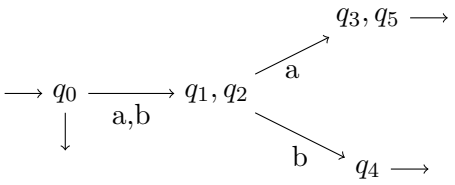
L'ordre de numérotation des états correspond à l'ordre lexicographique des mots qu'il reconnaissent.

RPNI va construire l'automate minimal déterministe reconnaissant L en effectuant une succession de *fusions déterministes* d'états (det-merge), i.e. que si une fusion de deux états entraîne un non déterminisme de l'automate, on fusionne les états responsables de ce non déterminisme récursivement. Ensuite on utilise l'ensemble des exemples négatifs S^- pour tester si l'automate obtenu reconnaît toujours le langage cible.

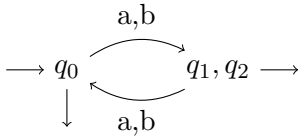
Dans notre exemple, on fusionne les états q_0 et q_1. L'automate ainsi construit reconnaît le mot $aaa \in S^-$ donc la fusion est annulée. De même pour la fusion de q_0 et q_2 qui ajoute b dans le langage reconnu.



On fusionne l'état q_1 et q_2, ce qui entraîne la fusion de q_3 et q_5. Le langage reconnu ne contient pas de mots de l'ensemble négatif, on peut donc garder cette fusion et continuer. Pour garder l'ordre associé aux états le rang des états fusionné est égal à celui du plus petit des deux états fusionnés.



Les fusions (q_0, q_3), (q_0, q_4) sont acceptées et donnent l'automate minimal reconnaissant le langage cible, qui dans notre cas était l'ensemble des mots de longueur paire sur $\{a, b\}$.



4.2 Adaptation aux arbres

L'algorithme RPNI a déjà été adapté aux arbres [OG94]. Dans le cas des arbres, il travaille sur les automates d'arbres ascendants déterministes et comme son équivalent pour les mots, opère par fusions successives tout en contrôlant la validité des automates obtenus. Nous nous baserons sur ce modèle pour définir l'algorithme d'apprentissage pour les superpositions d'arbres.

On sait que qu'un langage d'arbre régulier représenté par un automate d'arbre ascendant est polynomialement apprenable par données fixées [OG94]. Cette polynomialité dépend de la taille de l'échantillon caractéristique du langage, étant défini pour un langage L pour un algorithme a par un ensemble de données S tel que si $S \subseteq S'$ $L(a(S')) = L$.

Dans notre cas, l'apprentissage d'un langage d'arbres d'alignements, nous commençons par la création d'un automate à pas ascendant déterministe qui reconnaît l'ensemble des exemples positifs. N'ayant pas d'exemples négatifs explicites, une solution pour tester que l'automate reste dans le domaine d'apprentissage souhaité est d'utiliser la propriété de fonctionnalité de cet automate. Mais nous devons aussi contrôler que le langage des arbres d'entrée reconnu par notre transformation est cohérent par rapport à la transformation souhaité. Pour cela on vérifie l'inclusion de ce langage dans un schéma décrivant le formalisme de l'entrée. Un schéma de données définit par un ensemble de règles la structure d'un document arborescent (par exemple les DTD pour XML).

Pour que ces propriétés puissent être utilisés dans RPNI nous devons conserver la contrainte de polynomialité nécessaire à cet algorithme. Comme nous l'avons prouvé précédemment, la fonctionnalité d'un langage d'alignement limité aux superposition est décidable en temps polynomial. Et on sait que le test d'inclusion est lui aussi de compléxité polynomiale.

Algorithm 1 RPNI pour un langage régulier d'alignements

1: S^+
2: $A \leftarrow$ un automate à pas t.q. $L(A) = S^+$
3: **for** $i = 1$ to $|states(A)|$ **do**
4: **for** $j \leftarrow 0, \ldots, i-1$ **do**
5: $A \leftarrow$ `det-merge`(q_i, q_j)
6: **if** A est fonctionnel et l'inclusion de l'entrée est vérifiée **then**
7: $A \leftarrow A'$
8: **return** A

end function

Conclusion

Notre tâche a consisté à proposer un modèle de transformation ayant les propriétés nécessaires pour définir la plus grande possibilité de transformation et à un futur apprentissage.

Nous avons dans un premier temps définit les arbres d'alignements en se basant sur les opérations d'édition dans les arbres, ce modèle essayant de rester le plus global et de permettre la plupart des transformations pouvant être définie au sein des documents semi-structurés.

En s'intéressant aux propriétés nécessaires à une réelle transformation de documents ainsi qu'au différentes techniques d'apprentissage automatique, il nous est apparu que la notion de fonctionnalité était un besoin prédominant pour ce modèle. En étudiant cette contrainte nous nous sommes aperçu de son indécidabilité pour ce type de transformation. Indécidabilité que nous prouvons en nous rapprochant au problème d'ambiguïté de grammaires algébriques connue indécidable.

Nous avons donc décider d'imposer des restrictions afin de définir un langage de transformation où la fonctionnalité est décidable, ce qui nous a amené aux relations reconnaissables d'arbres. Par la suite, nous avons prouvé que la fonctionnalité est décidable pour les langages de superposition d'arbres ce qui a ouvert des perspectives pour un futur apprentissage. Des algorithmes adaptés aux arbres existent déjà, mais il est nécessaire d'opérer quelques adaptations pour pouvoir les appliquer à notre modèle.

Grâce aux différentes propriétés obtenues sur ce modèle, nous appréhendons mieux les problèmes liés aux transformations de données arborescentes ainsi que les contraintes nécessaires à leurs apprentissage.

Il faut maintenant appliquer ce modèle et l'apprentissage adapté sur des données réelles pour l'évaluer, et essayer d'élargir l'éventail des transformations possibles en se basant sur notre modèle, sachant que le langage d'alignement représente une borne supérieure à cette extension.

Bibliographie

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages, 2004.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http ://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[CGLN07] Julien Carme, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren. Interactive learning of node selecting tree transducers. *Machine Learning*, 66(1) :33–67, 2007.

[CLN04] J. Carme, A. Lemay, and J. Niehren. Learning node selecting tree transducers from completely annotated examples. In *International Colloquium on Grammatical Inference (ICGI 2004)*, pages 91–102, 2004.

[CNT04] J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *International Conference on Rewriting Techniques and Applications (RTA 2004)*, pages 105–118, 2004.

[DEGV97] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. In *IEEE Conference on Computational Complexity*, pages 82–101, 1997.

[GJTT06] Rémi Gilleron, Florent Jousse, Isabelle Tellier, and Marc Tommasi. Xml document transformation with conditional random fields. In *INEX 2006*, number 4518 in Lecture Notes in Computer Science. Springer Verlag, 2006.

[GNT08] Olivier Gauwin, Joachim Niehren, , and Sophie Tison. Bounded delay and concurrency for earliest query answering. *Information Processing Letters*, 2008. Accepted for publication.

[JWZ94] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of trees - an alternative to tree edit. In *CPM '94 : Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 75–86, London, UK, 1994. Springer-Verlag.

[LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, New York, 1981.

[Nev02a] F. Neven. Automata, logic, and XML. In *Conference for Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.

[Nev02b] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

[OG91] J. Oncina and P. Gracia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, 1991.

[OG92] J. Oncina and P. Gracia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108, 1992.

[OG94] J. Oncina and P. Garcia. Inference of rational tree sets. Technical Report DSIC-ii-1994-23, Departamento de Sistemas Informaticos y Computaction, Universidad Politecnica de Valencia, 1994.

[Tou07] Hélène Touzet. Comparing similar ordered trees in linear-time. *J. of Discrete Algorithms*, 5(4) :696–705, 2007.

[TVY08] Alex Thomo, Srinivasan Venkatesh, and Ying Ying Ye. Visibly pushdown transducers for approximate validation of streaming xml. In Sven Hartmann and Gabriele Kern-Isberner, editors, *FoIKS*, volume 4932 of *Lecture Notes in Computer Science*, pages 219–238. Springer, 2008.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6) :1245–1262, 1989.