Projections et cohérence de vues dans les grammaires algébriques

Éric BADOUEL et Maurice TCHOUPÉ Tchendji

Inria Rennes - Irisa
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{ebadouel,mtchoupe}@irisa.fr

Maurice Tchoupé bénéficie d'une bourse de thèse offerte par le service de coopération et d'action culturelle de l'ambassade de France au Cameroun et d'un complément financier apporté par le projet SARIMA

RÉSUMÉ. Un document structuré complexe est représenté intentionnellement sous la forme d'une structure arborescente décorée par des attributs. Les structures licites sont caractérisées par une grammaire algébrique abstraite. Nous faisons ici abstraction des attributs ; ces derniers sont liés à des aspects sémantiques qui peuvent être traités séparément des aspects purement structurels qui nous intéressent ici. Cette représentation intentionnelle peut être manipulée de façon indépendante et éventuellement non synchronisée par divers outils d'édition et de manipulation qui opèrent sur des vues partielles distinctes du même document. Pour la re-synchronisation de ces vues partielles nous devons résoudre le problème de leur cohérence : décider s'il existe un document correspondant à ces différentes vues et dans l'affirmative produire un tel document. Nous montrons comment résoudre ce problème dans le cas où chaque vue est associée à un sous-ensemble des symboles grammaticaux : ceux qui correspondent aux catégories syntaxiques visibles. L'algorithme proposé, qui repose fortement sur le mécanisme d'évaluation paresseuse, résout ce problème même dans le cas où chaque vue partielle correspond à un nombre infini de documents possibles.

ABSTRACT. A complex structured document is intentionnally represented as a tree decorated with attributes. The set of legal structures is given by an abstract context-free grammar. We forget about the attributes; they are related with semantical issues that can be treated independently of the purely structural aspects that we address in this article. That intentional representation may be asynchronously manipulated by a set of independent tools each of which operates on a distinct partial view of the whole structure. In order to synchronize these various partial views, we are faced to the problem of their coherence: can we decide whether there exists some global structure corresponding to a given set of partial views and in the affirmative, can we produce such a global structure ? We solve this problem in the case where a view is given by a subset of grammatical symbols, those associated with the so-called visible syntactical categories. The proposed algorithm, that strongly relies on the mechanism of lazy evaluation, produces an answer to this problem even if partial views may correspond to an infinite set of related global structures.

MOTS-CLÉS : DTD, XML, grammaire algébrique, représentation intentionnelle, cohérence de vues, évaluation paresseuse, co-algèbres, automates d'arbres, anamorphisme

KEYWORDS : DTD, XML, Contex-Free Grammar, Intentional Representation, Coherence of Views, Lazy Evaluation, Co-Algebra, Tree Automata, Anamorphism

1. Introduction

Un document structuré, prenant la forme d'un arbre décoré par des attributs, peut être utilisé comme interface entre diverses équipes de concepteurs intervenant sur des aspects distincts d'une spécification hétérogène d'un système complexe. Cette approche est conforme à la démarche préconisée par des travaux récents sur la programmation orientée langages et la programmation générative [34, 13, 11] qui militent en faveur d'une représentation intentionnelle d'un programme, découplée de sa (ou de ses) vue(s) concrète(s) plus ou moins partielle(s), sur laquelle on peut opérer par divers outils de métaprogrammation pour l'éditer, la parcourir, la transformer, en extraire de l'information. L'ensemble des structures licites est donnée par une grammaire algébrique avec des contraintes de cohérence imposées aux valeurs des attributs (ce qui peut par exemple être spécifié à l'aide d'une grammaire attribuée [26, 32]). Nous ferons néanmoins ici abstraction des attributs : ces derniers sont liés à des aspects sémantiques qui peuvent être traités séparément des aspects purement structurels qui nous intéressent ici. Pour la même raison nous ferons abstraction de la syntaxe concrète des documents, puisque celle-ci peut être représentée par un attribut particulier. Au final, les documents licites sont identifiés aux arbres de syntaxe abstraite d'une grammaire algébrique ; la syntaxe concrète et les divers attributs sont supposés pouvoir être calculés par des règles sémantiques dont nous ne tenons pas compte dans ce travail. Nous pouvons donc oublier les symboles terminaux de la grammaire et ne considérer que des grammaires dites abstraites. Ces dernières peuvent être identifiées à des automates d'arbres ce qui nous permet de caractériser les documents licites comme un ensemble régulier d'arbres.

Les différents outils d'édition et de manipulation d'un document structuré opèrent en général sur des vues partielles de son arbre de syntaxe abstraite. Par exemple plusieurs intervenants dans un projet, ayant chacun un domaine d'expertise propre, peuvent travailler par le biais d'interfaces qui reposent sur des langages dédiés liés à leurs domaines d'intervention et associées à diverses vues partielles d'une même structure partagée. On peut imaginer des situations dans lesquelles ces différentes manipulations peuvent se faire de manière asynchrone. Dans un tel cas nous sommes confrontés au problème de la cohérence des vues : existe-t-il toujours une(des) structure(s) globale(s) correspondant aux différentes vues à un moment donné ? Si oui, peut-on les produire ?

Le but de cet article est de proposer une notion de vue partielle d'un document structuré et de fournir une réponse à ce problème de la cohérence de vues. Nous considérons qu'une vue d'un document est obtenue par la projection consistant a ne conserver que les symboles grammaticaux visibles par un utilisateur donné. *A priori* un ensemble infini de documents licites peuvent donner lieu à une même image dans cette projection. Néanmoins nous montrons qu'il est possible d'associer un automate d'arbres à une vue, identifiée au sous-ensemble des symboles grammaticaux visibles, de telle sorte que le langage (régulier) d'arbres reconnu par cet automate à partir d'un état initial associée à cette image coïncide avec l'ensemble des documents licites se projetant sur cette image. Nous observons que les automates d'arbres sont des co-algèbres pour un foncteur dont le point fixe est une structure de données co-inductive dont les éléments (que nous appelons trees) permettent de représenter, de façon paresseuse, des ensembles potentiellement infinis d'arbres. Un trees engendré à partir d'un automate d'arbre (co-algèbre) et un état initial (élément du domaine de cette co-algèbre) est le codage de l'ensemble régulier d'arbres reconnus par cet automate à partir de cet état initial. On peut aisément décider

si l'ensemble représenté par un tel trees (régulier) est non vide et dans un tel cas exhiber certains de ses éléments. La résolution du problème de cohérence est alors facile : il suffit de faire le produit synchrone des différents automates associés à chacune des vues ce qui procure un nouvel automate qui reconnaît l'intersection des langages d'arbres reconnus par chacun d'entre eux.

La notion de vue utilisée dans cet article n'est pas sans rappeler celle qui a été introduite dans le contexte des bases de données lorsqu'il s'agissait de présenter à divers utilisateurs des informations extraites de la base de donnée. La base de données pouvait être distribuée et le problème de la cohérence de ses différentes vues locales s'est posée naturellement. Plus proche de notre problématique, cette notion a aussi émergé dans la communauté XML sous l'appellation *vue XML* pour désigner suivant les auteurs soit un document XML contenant des données provenant d'une base de données [1] et sujette aux opérations d'éditions permettant la mise à jour de la base de données correspondante [9], soit un fragment de document XML obtenu à partir d'un document de base XML par combinaison à partir de certaines opérations de base [10] : la projection (par restriction à certains éléments XML), la sélection (à partir des valeurs de certains attributs), l'échange (certains éléments peuvent avoir un ordre d'apparition dans la vue différent de celui existant dans le document de base). La création d'une vue XML au sens précédent peut se faire aisément en utilisant l'un des langages proposés par le consortium World Wide Web pour l'interrogation des documents XML comme *XSL* [40] ou *XQuery* [39].

Nous n'avons pas connaissance d'autres études portant sur la validation de documents XML sujets à un processus d'édition asynchrone sinon [27, 19] où les auteurs abordent cette question dans un cadre très restrictif en présentant des algorithmes pour la fusion de plusieurs vues en un document global, mais dans lequel toutes les vues ainsi que le résultat de la fusion sont des documents conformes *à une même DTD*. Le cas synchrone est par exemple abordée dans [30] où la cohérence est garantie par le fait que chaque opération d'édition sur la vue (resp. sur le document de base) est directement répercutée sur le document de base (resp. sur la vue) en utilisant des transformations bi-directionnelles d'arbres.

La suite du document est structurée en quatre sections suivies d'une conclusion. Ces différentes sections présentent successivement les documents structurés et leurs sérialisations (section 2), la notion de vue associée à un sous-ensemble de symboles grammaticaux (section 3), l'algorithme d'expansion permettant de produire l'ensemble des documents ayant une vue partielle donnée (section 4), et enfin notre solution au problème de la cohérence de vues (section 5). Nous nous sommes attachés à donner une implémentation détaillée de tous les algorithmes proposés dans le langage Haskell [38]. Ce choix est motivé par le fait que notre solution repose fortement sur les structures de données co-inductives et l'évaluation paresseuse.

Note préliminaire : *Nous avons conscience que bien des lecteurs potentiellement intéressés par le sujet abordé dans cet article ne sont pas nécessairement familiers avec le langage Haskell. Aussi nous avons ajouté une courte annexe pour introduire quelques notations qui devraient suffire à la bonne compréhension de ce code. Il s'agit de quelques combinateurs classiques de Haskell et d'une présentation du schéma de compréhension des listes que nous utilisons de façon systématique. Le lecteur plus aventureux, ou plus courageux, qui voudrait non seulement comprendre ce code mais l'expérimenter aura besoin de quelques fonctions complémentaires présentées dans une seconde annexe.*

2. Documents structurés et leurs sérialisations

Nous nous intéressons uniquement à la structure d'un document indépendamment de son contenu ainsi que de divers autres attributs pouvant y être attaché. Les documents licites sont ceux dont la structure est conforme à une grammaire algébrique abstraite.

Définition 1 *Une grammaire algébrique abstraite est la donnée* $\mathbb{G} = (S, P, A)$ *d'un ensemble fini* S *de* ***symboles grammaticaux*** *qui correspondent aux différentes* ***catégories syntaxiques*** *en jeu, d'un symbole grammatical* $A \in S$ *particulier, appelé* ***axiome****, et d'un ensemble fini* $P \subseteq S \times S^*$ *de* ***productions****. Une production* $P = \big(X_{P(0)}, X_{P(1)} \cdots X_{P(n)}\big)$ *est notée* $P : X_{P(0)} \to X_{P(1)} \cdots X_{P(n)}$ *et* $|P|$ *désigne la longueur de la partie droite de* P*.*

Nous représentons en Haskell une grammaire par le type suivant :

$$\begin{aligned} data\ Gram\ prod\ symb = Gram\ \{ & prods :: [prod], \\ & symbols :: [symb], \\ & lhs :: prod \to symb, \\ & rhs :: prod \to [symb]\} \end{aligned}$$

dans lequel les variables de type *prod* et *symb* donnent respectivement le type des productions et le type des symboles grammaticaux de la grammaire. Elle est ainsi présentée comme un quadruplet dont le premier élément

$$prods :: Gram\ prod\ symb \to [prod]$$

fournit la liste des productions, le second

$$symbols :: Gram\ prod\ symb \to [symb]$$

la liste des symboles grammaticaux. La fonction

$$lhs :: Gram\ prod\ symb \to prod \to symb$$

retourne le symbole grammatical se trouvant en partie gauche d'une production, tandis que la fonction

$$rhs :: Gram\ prod\ symb \to prod \to [symb]$$

retourne la liste des symboles grammaticaux en partie droite d'une production. Par exemple la grammaire constituée des productions

$$\begin{array}{lll} P_1 : A \to C\,B & P_3 : B \to C\,A & P_5 : C \to A\,C \\ P_2 : A \to \varepsilon & P_4 : B \to B\,B & P_6 : C \to C\,C \\ & & P_7 : C \to \varepsilon \end{array}$$

est définie comme suit :

$$
\begin{array}{l}
data\ Prod\ =\ P1 \mid P2 \mid P3 \mid P4 \mid P5 \mid P6 \mid P7\ deriving\ (Eq,\ Show) \\
data\ Symb\ =\ A \mid B \mid C\ \ deriving\ (Eq, Show) \\
gram\ ::\ Gram\ Prod\ Symb \\
gram = Gram\ lprod\ lsymb\ lhs\_\ rhs\_ \\
where \quad lprod = [P1,\ P2,\ P3,\ P4,\ P5,\ P6,\ P7] \\
\qquad\quad lsymb = [A,\ B,\ C] \\
\qquad\quad lhs\_\ p = case\ p\ of \\
\qquad\qquad\qquad P_1 \to A;\ \ P_3 \to B;\ \ P_5 \to C; P_7 \to C \\
\qquad\qquad\qquad P_2 \to A;\ \ P_4 \to B;\ \ P_6 \to C \\
\qquad\quad rhs\_\ p = case\ p\ of \\
\qquad\qquad\qquad P_1 \to [C,\ B];\ \ P_3 \to [C,\ A];\ \ P_5 \to [A,\ C];\ \ P_7 \to [\,] \\
\qquad\qquad\qquad P_2 \to [\,];\ \ P_4 \to [B,\ B];\ \ P_6 \to [C,\ C]
\end{array}
$$

2.1. Arbres de dérivation vs arbres de syntaxes abstraites

Un document structuré se présente sous la forme d'un arbre dont les noeuds sont associés à des catégories syntaxiques (symboles de la grammaire). Cette structure arborescente reflète la structure hiérarchique du document

Définition 2 *Un **arbre** dont les noeuds sont étiquetés dans un alphabet **A** est une fonction $t : \mathbb{N}^* \to \mathbf{A}$ dont le domaine $Dom(t) \subseteq \mathbb{N}^*$ est un ensemble clos par préfixe tel que pour tout $u \in Dom(t)$ l'ensemble $\{i \in \mathbb{N} \mid u \cdot i \in Dom(t)\}$ est un intervalle d'entiers $[1, \cdots, n] \cap \mathbb{N}$; l'entier n est l'**arité** du noeud d'adresse u. Si $t_1, \cdots, t_n$ sont des arbres et $a \in \mathbf{A}$ on note $t = a(t_1, \ldots, t_n)$ l'arbre t de domaine $Dom(t) = \{\varepsilon\} \cup \{i \cdot u \mid 1 \leq i \leq n,\ u \in Dom(t_i)\}$ avec $t(\varepsilon) = a$ et $t(i \cdot u) = t_i(u)$.*

Nous représentons de tels arbres en Haskell par la structure de données

$$data\ Tree\ a = Node\{top :: a,\ succ\_ :: [Tree\ a]\} \qquad deriving\ (Eq, Show)$$

Un document structuré est conforme à la grammaire si chacune des étapes de décomposition hiérarchique du document obéit à une des productions de la grammaire, ce qui revient à dire qu'il s'agit d'un *arbre de dérivation* au sens de la définition suivante :

Définition 3 *L'ensemble $Der(\mathbb{G}, X)$ des **arbres de dérivation** selon la grammaire $\mathbb{G}$ associés au symbole grammatical X est constitué des arbres de la forme $X(t_1, \ldots, t_n)$ pour lesquels il existe une production P telle que $X = X_{P(0)}$, $n = |P|$ et $t_i \in Der(\mathbb{G}, X_i)$ pour tout $1 \leq i \leq n$.*

Un arbre de dérivation est donc un arbre dont les noeuds sont étiquetés par les symboles grammaticaux (document structuré) de façon conforme à la grammaire. De la même manière on peut définir les arbres de syntaxe abstraite qui sont des arbres dont les noeuds sont, cette fois-ci, étiquetés par des productions de la grammaire.

Définition 4 *L'ensemble $AST(\mathbb{G}, X)$ des **arbres de syntaxe abstraite** selon la grammaire $\mathbb{G}$ associés au symbole grammatical X est constitué des arbres de la forme $P(t_1, \ldots, t_n)$ où P est une production telle que $X = X_{P(0)}$, $n = |P|$ et $t_i \in AST(\mathbb{G}, X_i)$ pour tout $1 \leq i \leq n$. Les arbres de syntaxe abstraite sont donc les termes pour la signature dont les sortes sont les symboles grammaticaux et dont les opérateurs sont les productions de la*

grammaire où la production $P : X_{P(0)} \to X_{P(1)} \cdots X_{P(n)}$ *est vue comme un opérateur d'arité* $X_{P(1)} \times \cdots \times X_{P(n)} \to X_{P(0)}$.

On peut interpréter les arbres de syntaxe abstraite comme des preuves de la conformité des documents structurés par rapport à la grammaire (voir fig. 1). La relation $t \vdash u$ dans laquelle t est un arbre de syntaxe abstraite et u un document structuré (arbre dont les noeuds sont étiquetés par des symboles grammaticaux) est la plus petite relation pour laquelle on a $P(t_1,\ldots,t_n) \vdash X(u_1,\ldots,u_m)$ si $X = X_{P(0)}$, $m = |P|$ (c'est-à-dire m) et $t_i \vdash u_i$ pour tout $1 \leq i \leq n$. Un document u est ainsi conforme (c'est-à-dire est un arbre de dérivation) si, et seulement si, il existe un arbre de syntaxe abstraite t tel que $t \vdash u$. Dans cette relation t détermine u (il suffit de remplacer chaque production par son symbole en partie gauche) ; et la réciproque est vraie si *chaque production de la grammaire est caractérisée par la donnée conjointe de sa partie gauche et de sa partie droite*, hypothèse que nous ferons par la suite.

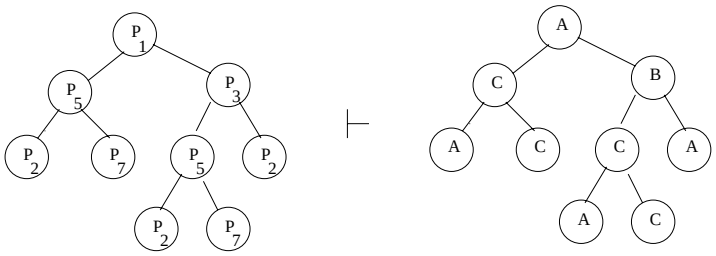

Figure 1. *Représentation intentionnelle (arbre de syntaxe abstraite) et arbre de dérivation associé*

2.2. Sérialisation d'un document structuré

Pour des besoins de sérialisation un arbre peut être linéarisé sous la forme d'un mot de Dyck à n lettres où n est la taille de l'alphabet étiquetant les noeuds de l'arbre. Rappelons que le *langage de Dyck* à n lettres $\Sigma_n = \{(_i,\ )_i \mid 1 \leq i \leq n\}$ c'est-à-dire le langage des parenthèses sur n lettres, est donné par la grammaire

$$\langle Dyck \rangle \ \to\ (_i \ \langle Dyck \rangle \ )_i \ \langle Dyck \rangle \mid \varepsilon \qquad 1 \leq i \leq n$$

ou de façon équivalente par

$$\begin{array}{lcll} \langle Dyck \rangle & \to & \langle primeDyck \rangle^* & \\ \langle primeDyck \rangle & \to & (_i \ \langle Dyck \rangle \ )_i & 1 \leq i \leq n \end{array}$$

Un mot premier de Dyck code un arbre et un mot de Dyck une suite d'arbres, c'est-à-dire une forêt (figure 2). Nous représentons une lettre du langage de Dyck par le type :

$$data\ Dyck\ symb = Open\ symb \mid Close\ symb \qquad deriving\ (Eq, Show)$$

La linéarisation d'un arbre est alors donnée par la fonction suivante :

$$\begin{array}{l} linearisation :: Tree\ symb \to [Dyck\ symb] \\ linearisation\ (Node\ symb\ ts) = \\ \quad [Open\ symb]\ ++\ (concat\ (map\ linearisation\ ts))\ ++\ [Close\ symb] \end{array}$$

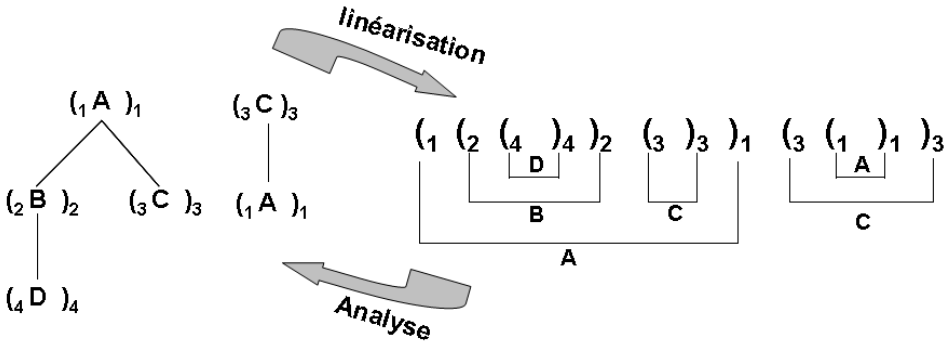

Figure 2. *linéarisation d'une forêt et analyse d'un mot de Dyck*

La fonction *analyse* qui reconstitue un arbre ou une forêt à partir de sa sérialisation est décrite dans l'annexe 2.

À une grammaire abstraite $\mathbb{G} = (S, P, A)$ on associe une grammaire algébrique $\mathbb{G}_S = (S, T, P_S, A)$ obtenue en ajoutant comme symboles terminaux une parenthèse ouvrante et une parenthèse fermante associées à chaque catégorie syntaxique, c'est-à-dire $T = Dyck\, S$. Les productions de P_S sont obtenues à partir de celles de P en insérant leur partie droite entre une paire de parenthèses : les productions de P_S sont de la forme $p : A_0 \rightarrow (_{A_0} A_1 \cdots A_n)_{A_0}$ pour $p : A_0 \rightarrow A_1 \cdots A_n$ une production de P. [1] Les deux grammaires ont les mêmes arbres de syntaxe abstraite et la linéarisation d'un arbre de dérivation t pour $\mathbb{G}$ n'est rien d'autre que le mot reconnu par $\mathbb{G}_S$ dont l'arbre de dérivation correspond au même arbre de syntaxe abstraite que t. Une grammaire abstraite ne reconnaît aucun mot, sinon le mot vide, ce qui nous intéresse dans son cas est l'ensemble de ses arbres de dérivation et le langage reconnu par la grammaire $\mathbb{G}_S$ est justement l'ensemble des codages des arbres de dérivation de $\mathbb{G}$. À ce stade on s'aperçoit que trois représentations équivalentes peuvent être données d'un document structuré : une représentation "interne" (son arbre de syntaxe abstraite), une représentation "externe" (son arbre de dérivation) et la linéarisation de cette dernière sous la forme d'un mot du langage de Dyck (une représentation à la XML).

1. La grammaire $\mathbb{G}_S$ est une grammaire équilibrée au sens de [7]. Des langages algébriques à structures de parenthèses (avec un seul jeu de parenthèses néanmoins) ont été introduits et étudiés il y a assez longtemps par McNaughton [31] et Knuth [25]. Ces langages sont engendrés par des grammaires dites de parenthèses. L'essor de la technologie XML a permis de raviver ce domaine [6] en particulier Berstel et Boasson ont introduit les grammaires équilibrées (*Balanced Grammars*[7]) qui généralisent les grammaires de parenthèses à plusieurs types de parenthèses (et qui autorisent également des langages réguliers en partie droite des productions). Tous ces langages sont des cas particuliers de langages engendrés par des automates à pile rendant "visibles" les actions sur la pile (chaque symbole d'entrée détermine si l'automate effectue une action push, une action pop ou une action ne touchant pas à la pile). Ces langages, appelés *Visibly Pushdown Languages* [3], ont de très bonnes propriétés de stabilité (clôture par union, intersection, complémentation, renommage, concaténation et itération de Kleene), une caractérisation logique en terme de formules du second ordre monadique et un lien étroit avec les langages réguliers d'arbres. À ce titre cette famille de langages est un cadre idéal pour étudier de manière formelle les objets et les transformations mis en jeu par la technologie XML.

3. Vues et projections inverses

L'arbre de dérivation, donnant la vue "externe" d'un document structuré, rend visible l'ensemble des symboles grammaticaux de la grammaire. Un programme manipulant un tel document, tel un éditeur structuré, n'aura cependant pas accès à l'ensemble de tous ces symboles grammaticaux, seul un sous-ensemble d'entre eux correspondent à des catégories syntaxiques perceptibles comme telles par cet outil. Les autres symboles grammaticaux apparaîtront plutôt comme des artefacts permettant de structurer la grammaire. Contrairement au point de vue adopté dans la section précédente, il est donc légitime de faire une distinction entre les symboles grammaticaux de la grammaire et les catégories syntaxiques (qui en sont un sous-ensemble) associées à un outil de manipulation de ces documents structurés. Par ailleurs ce qui constitue une catégorie syntaxique pour un outil ne le sera pas nécessairement pour un autre : chacun disposera d'une vue partielle de l'arbre de dérivation.

3.1. Vue et projection associée

Une vue est un sous-ensemble de symboles grammaticaux $V \subseteq S$. Intuitivement il s'agit des symboles associés aux catégories syntaxiques visibles dans la représentation considérée (arbre de dérivation). On implémentera une vue comme un prédicat sur les symboles. Par exemple les vues $V_1 = \{A, B\}$ et $V_2 = \{A, C\}$ sur l'alphabet $\{A, B, C\}$ sont définies comme suit :

$$
\begin{array}{l}
viewAB :: Symb \to Bool \\
viewAB\ symb = \begin{array}{l} case\ symb\ of \\ A \to True \\ B \to True \\ C \to False \end{array}
\end{array}
\qquad
\begin{array}{l}
viewAC :: Symb \to Bool \\
viewAC\ symb = \begin{array}{l} case\ symb\ of \\ A \to True \\ B \to False \\ C \to True \end{array}
\end{array}
$$

À chaque vue est associée une projection sur les arbres de dérivation qui efface les noeuds étiquetés par des symboles invisibles tout en conservant la structure de sous-arbre. Le résultat est une liste d'arbres qui pourra effectivement contenir plusieurs éléments dans le cas où l'axiome est invisible.

$$
\begin{array}{l}
projection :: (symb \to Bool) \to Tree\ symb \to [Tree\ symb] \\
projection\ view\ der = if\ view\ (top\ der)\ then\ [Node\ (top\ der)\ sons]\ else\ sons \\
\quad where\ sons = concat\ (map\ (projection\ view)\ (succ\_der))
\end{array}
$$

La figure 3 montre un arbre de dérivation *der* et ses deux projections *derAB=projAB der* et *derAC=projAC der* sur $\{A, B\}$ et $\{A, C\}$ respectivement.

$$
projAB = projection\ viewAB \qquad projAC = projection\ viewAC
$$

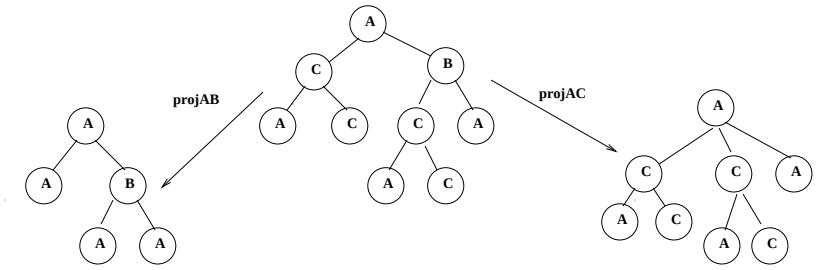

Figure 3. *un arbre de dérivation et ses projections sur {A,B} et {A,C}*

3.2. Lien avec la sérialisation

On peut définir une projection sur les mots de Dyck consistant à effacer toutes les parenthèses associées à des symboles invisibles, le résultat est un mot de Dyck sur l'alphabet réduit aux parenthèses visibles :

$$\begin{array}{l} \mathit{projection\_} :: (\mathit{symb} \to \mathit{Bool}) \to [\mathit{Dyck\ symb}] \to [\mathit{Dyck\ symb}] \\ \mathit{projection\_\ view} = \mathit{filter}\ g \\ \quad \mathit{where} \quad g\ (\mathit{Open\ symb}) = \mathit{view\ symb} \\ \qquad\qquad g\ (\mathit{Close\ symb}) = \mathit{view\ symb} \end{array}$$

On peut facilement établir que

$$\mathit{linearisation} \circ (\mathit{projection\ view}) = (\mathit{projection\_\ view}) \circ \mathit{linearisation}$$

Cette fonction produit la *trace visible* d'un arbre de dérivation

$$\mathit{trace\ view\ der} = (\mathit{projection\_\ view})\ (\mathit{linearisation\ der})$$

et ainsi la projection d'un arbre de dérivation peut alternativement être obtenue par analyse de sa trace visible (voir la figure 4) :

$$t = \mathit{projection\ view\ der} \quad \mathit{ssi} \quad \mathit{Just}\ t = \mathit{analyse}\ (\mathit{trace\ view\ der})$$

La trace visible d'un arbre de dérivation *der* correspond également au mot reconnu par la

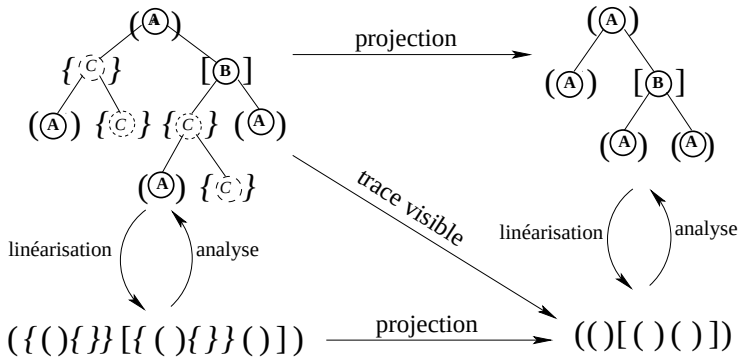

Figure 4. *relations entre projections et sérialisation*

grammaire algébrique $\mathbb{G}_V$ associée à la vue $V \subseteq S$ pour l'arbre de dérivation relatif à cette

grammaire et ayant le même arbre de syntaxe abstraite que *der*. La grammaire algébrique $\mathbb{G}_V = (S, T, P_V, A)$ est obtenue à partir de $\mathbb{G}$ en ajoutant comme symboles terminaux une parenthèse ouvrante et une parenthèse fermante associées à chaque catégorie syntaxique visible, c'est-à-dire $T = Dyck\ V$. Les productions de P_V sont obtenues à partir de celles de P en insérant leur partie droite entre une paire de parenthèses lorsque le symbole en partie gauche de la production est visible : les productions de P_V sont les productions $p : A_0 \to A_1 \cdots A_n$ de P pour lesquelles le symbole A_0 est invisible et les productions de la forme $p : A_0 \to (_{A_0} A_1 \cdots A_n)_{A_0}$ pour $p : A_0 \to A_1 \cdots A_n$ une production de P et A_0 un symbole visible. La grammaire $\mathbb{G}_V$ reconnait l'ensemble des traces visibles des arbres de dérivation de $\mathbb{G}$ pour la vue $V \subseteq S$.

3.3. Projection inverse

Le problème de la projection inverse consiste à déterminer tous les arbres de syntaxe abstraite donnant lieu à une projection visible donnée.

Il est facile de voir qu'il existe un nombre infini d'arbres de dérivation *der* correspondant à l'une ou l'autre des vues obtenues par projections de la figure 3. En effet, les ensembles $\{der \mid projAB\ der = derAB\}$ et $\{der \mid projAC\ der = derAC\}$ sont tous les deux infinis comme nous l'illustre la figure 5 où il suffit d'itérer sur la production $p_6 : C \to C\,C$ pour produire un ensemble infini de solutions.

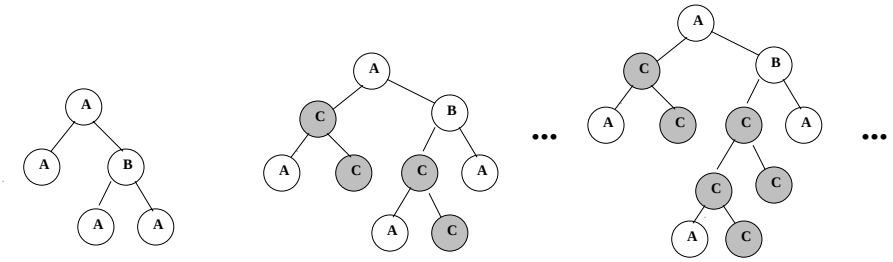

Figure 5. *les expansions possibles d'une vue partielle*

Par contre un arbre de dérivation pour cette grammaire est caractérisé par l'ensemble de ses deux projections. Nous dirons qu'un ensemble de vues *couvre* une grammaire si tout arbre de dérivation pour cette grammaire est caractérisé par l'ensemble de ses projections. Nous n'avons pour l'instant pas de réponse à la question naturelle suivante :

Problème 5 *Peut on décider si un ensemble donné de vues couvre une grammaire donnée ?*

4. Algorithme d'expansion

L'algorithme d'expansion vise à construire l'ensemble des arbres de syntaxe abstraite ayant une projection visible donnée. Comme cet ensemble peut être infini nous introduisons une structure de données co-inductive permettant de représenter des ensembles potentiellement infinis d'arbres. Le lecteur désireux d'en savoir plus sur les foncteurs, et sur les notions d'algèbres et de co-algèbres qui leur sont associées aussi bien dans un cadre catégorique que sur la façon dont ces notions sont implémentées dans un langage fonc-

tionnel paresseux, comme Haskell, pourra se référer par exemple aux documents [36, 5]. Pour l'usage que nous en ferons ici il nous suffit de rappeler les éléments suivants. Un foncteur en Haskell est un constructeur de types pour lequel on dispose d'une méthode *fmap* du type suivant

$$classe\ Functor\ f\ where\ fmap\ ::\ (a \to b) \to f\ a \to f\ b$$

intuitivement *fmap g* applique la fonction *g* à toute occurrence d'éléments de type *a* dans un élément de type $f\ a$; la fonction *fmap* est supposée compatible avec l'identité et la composition fonctionnelle, en ce sens que $fmap\ id\ =\ id$ et $fmap\ (g \cdot h)\ =\ (fmap\ g) \cdot (fmap\ h)$. Les structures de données récursives sont définies comme point fixes de certains foncteurs ; on écrira

$$newtype\ Fix\_f\ = In\{out\ ::\ f\ Fix\_f\}$$

pour définir la structure de données *Fix_f* comme point fixe du foncteur *f*, c'est-à-dire telle que $Fix\_f \cong f\ Fix\_f$. Cet isomorphisme est donnée par la paire de fonctions associées respectivement au constructeur $In\ ::\ f\ Fix\_f \to Fix\_f$ et au sélecteur $out\ ::\ Fix\_f \to f\ Fix\_f$. Une algèbre pour le foncteur f de domaine *a* est une fonction de type $f\ a \to a$:

$$type\ Alg\_f\ a\ =\ f\ a \to a$$

On peut alors évaluer de façon inductive un élément de la structure de données *Fix_f* dans le domaine *a* d'une algèbre en utilisant le combinateur *fold* :

$$\begin{array}{l} fold\_f\ ::\ Alg\_f\ a\ \to Fix\_f \to a \\ fold\_f\ alg\ =\ alg\ \cdot (fmap\ (fold\_f\ alg)) \cdot out \end{array}$$

Une fonction d'évaluation associée à une algèbre, c'est à dire de la forme *fold_f alg*, est appelée un *catamorphisme*. De façon duale, une co-algèbre pour le foncteur f de domaine *a* est une fonction de type $a \to f\ a$:

$$type\ Coalg\_f\ a\ =\ a \to f\ a$$

Et on peut générer, de façon co-inductive, un élément de la structure de données *Fix_f* à partir d'une structure de co-algèbre et d'un élément du domaine de cette co-algèbre (un *germe*) en utilisant le combinateur *unfold* :

$$\begin{array}{l} unfold\_f\ ::\ Coalg\_f\ a\ \to a \to Fix\_f \\ unfold\_f\ coalg\ =\ In \cdot (fmap\ (unfold\_f\ coalg)\ ) \cdot coalg \end{array}$$

Une fonction de génération associée à une co-algèbre, c'est à dire de la forme *unfold_f coalg*, est appelée un *anamorphisme*.

4.1. Une structure de données paresseuse pour des ensembles d'arbres de syntaxe abstraite

Le point fixe du foncteur paramétrique $f\ a$ où f est donné par

$$\begin{array}{l} type\ f\ a\ x = [(a,\ [x])] \\ instance\ Functor\ (f\ a)\ where\ fmap\ g\ xs\ =\ [(a, map\ g\ ys)\ |\ (a, ys) \leftarrow xs] \end{array}$$

est la structure polymorphe *Trees a* définie en Haskell par :

$$newtype\ Trees\ a\ =\ Branch\ \{unbranch :: [(a, [Trees\ a])]\} \qquad deriving\ (Eq,\ Show)$$

L'isomorphisme $Trees\ a \cong [(a, [Trees\ a])]$ est donnée par la paire de fonctions associées respectivement au constructeur *Branch* et au sélecteur *unbranch* :

$$\begin{aligned} &Branch\ ::\ [(a, [Trees\ a])]\ \rightarrow\ Trees\ a \\ &unbranch\ ::\ Trees\ a\ \rightarrow\ [(a, [Trees\ a])] \\ &unbranch\ (Branch\ list)\ =\ list \end{aligned}$$

Nous représentons graphiquement un *trees* par un arbre ayant deux types de noeuds (voir figure 6 dans laquelle les annotations (*) et (**) signifient que les sous arbres issus des noeuds avec la même annotation sont identiques). Les noeuds "ou" représentent des *trees* et les noeuds "et" représentent des listes de trees. Un noeud "ou" représentant un trees t aura autant de successeurs que d'éléments (a, ts) dans $unbranch\ t$; l'arc associé à cet élément est étiqueté par la lettre a et le successeur correspondant est le noeud "et" associé à la liste ts. Si l'ensemble $unbranch\ t$ est vide ce noeud n'a donc aucun successeur ; un noeud "ou" sans successeur représente un ensemble vide d'arbres et il sera représenté par le symbole $\bot$. Une liste de trees ts est associé à un noeud "et" dont le nombre de successeurs est donné par la longueur de cette liste, le $i^{\text{éme}}$ successeur étant le noeud "ou" associé au $i^{\text{éme}}$ élément (trees) de cette liste. Lorsque cette liste est vide ce noeud "et" n'a donc aucun successeur et il sera noté $\top$. L'interprétation d'un trees comme représentation

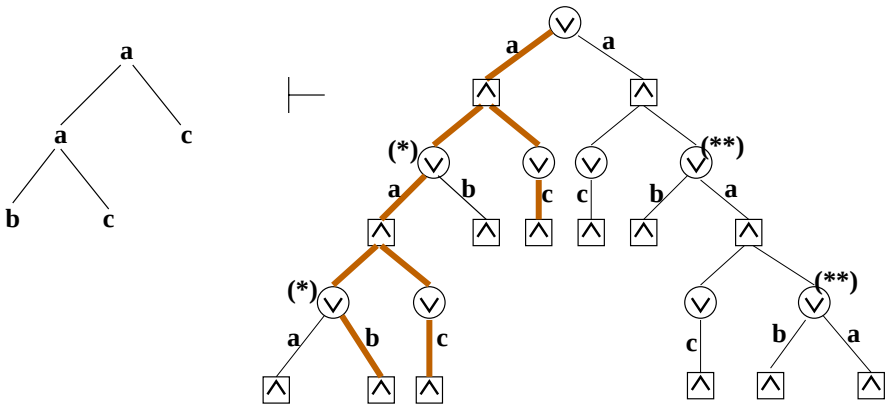

Figure 6. *structure de données paresseuse pour les ensembles d'arbres de dérivation*

d'un ensemble d'arbres est la suivante : un arbre $t = Node\ a\ ts$ est membre de *trees* s'il existe, en partant de la racine du trees, une branche étiquetée a menant à un noeud "et" ayant autant de successeurs que la longueur de ts et tel que chacun des arbres de ts est membre du successeur de ce noeud ayant le même rang. Ce qui correspond à la fonction suivante :

$$\begin{aligned} &isMember\ ::\ (Eq\ a)\ \Rightarrow\ Tree\ a\ \rightarrow\ Trees\ a\ \rightarrow\ Bool \\ &isMember\ (Node\ a\ ts)\ trees\ =\ or\ [and\ (zipWith\ isMember\ ts\ list\_trees)\mid \\ &\qquad\qquad (elet, list\_trees)\ \leftarrow\ unbranch\ trees, \\ &\qquad\qquad elet == a, \\ &\qquad\qquad (length\ ts) == (length\ list\_trees)] \end{aligned}$$

Par exemple les arcs représentés en gras dans la figure 6 établissent l'appartenance de l'arbre

$$Node\ a\ [Node\ a\ [Node\ b\ [], Node\ c\ []], Node\ c\ []]$$

à l'ensemble d'arbres représenté par le trees indiqué dans cette figure. Il est facile de voir que cet ensemble est constitué des arbres ayant une des deux formes suivantes :

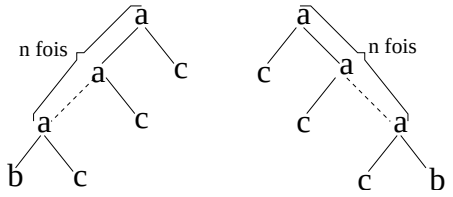

Figure 7. *arbres membres de l'ensemble représenté par le trees de la figure 6*

On observe qu'un arbre qui appartient à (l'ensemble d'arbres défini par) un trees peut être identifié au sous arbre du trees dont l'ensemble des noeuds vérifie la condition suivante : il s'agit d'un sous-ensemble fini (et bien sûr connexe) de noeuds contenant la racine du trees et tel que (i) tout noeud "ou" de cet ensemble admet un, et un seul de ses successeurs dans cet ensemble tandis que (ii) tout successeur d'un noeud "et" de cet ensemble s'y trouve également. Un arbre ainsi associé à un tel sous-ensemble de noeuds sera dit "inscrit" dans le trees (et on l'identifie à son "image" dans le trees). De même on parlera d'un arbre inscrit en un noeud "ou" d'un trees pour désigner un arbre inscrit dans le trees issu de ce noeud "ou". Intuitivement pour tracer un arbre inscrit dans un trees on doit partir de sa racine, en chaque noeud "ou" on fait le choix d'un arc (qui donne l'étiquette du noeud courant de l'arbre), et dans un noeud "et" on emprunte en parallèle chacun de ses successeurs (chacun d'entre eux servant à définir un des sous arbres issus de ce noeud). On ne s'intéresse ici uniquement qu'à des arbres finis, cette construction doit donc se terminer. Tous les chemins ainsi construits doivent donc aboutir à des noeuds sans successeurs. Ces noeuds ne peuvent pas être des noeuds "ou" car s'il n'a pas de successeur, on a aucun moyen d'en choisir un comme c'est requis par la condition (i) ci-dessus. Le fait qu'un noeud "et" n'ait pas de successeur ne contredit par contre pas la condition (ii). Les feuilles de l'arbre vont donc nécessairement correspondre à des arcs conduisant à des noeuds "et" sans successeur (c'est-à-dire des noeuds $\top$).

L'interprétation d'un noeud "ou" (ou son extension) est donnée par l'ensemble des arbres inscrits à partir de ce noeud. L'interprétation d'un noeud "et" d'arité *n* est donnée par l'ensemble des *n*-uplets d'arbres constitués d'un élément pris dans les extensions de chacun de ses successeurs. Par conséquent, si un noeud "et" possède un successeur $\bot$ (noeud "ou" sans successeur) il représentera un ensemble vide de n-uplets et n'apportera aucune contribution à l'extension des noeuds se trouvant au dessus de lui, et en particulier à la racine. On ne changera donc pas l'extension du trees en supprimant un tel noeud, ce qui est réalisé en coupant l'arc qui de son père (un noeud "ou") mène à ce noeud. En supprimant de tels arcs on peut être amené à créer de nouveaux noeuds $\bot$, et on réitère cette opération, dite de nettoyage, tant qu'elle s'applique jusqu'à aboutir à un trees équivalent sans noeud $\bot$, hormis éventuellement la racine qui en est alors l'unique noeud.

On peut énumérer les éléments d'un trees s'il est *fini* au moyen de la fonction *enumerate* ci-dessous.

$$
\begin{array}{l}
enumerate :: Trees\ a \rightarrow [Tree\ a] \\
enumerate\ trees = [Node\ elet\ list\_tree \mid \\
\qquad\qquad\qquad\qquad (elet, list\_trees) \leftarrow unbranch\ trees, \\
\qquad\qquad\qquad\qquad list\_tree \leftarrow dist\ (map\ enumerate\ list\_trees)] \\
dist\ [\,] = [[\,]] \\
dist\ (xs : xss) = [y : ys \mid y \leftarrow xs,\ ys \leftarrow dist\ xss]
\end{array}
$$

et tester la vacuité de cet ensemble par la fonction

$$
\begin{array}{lcl}
isEmpty & :: & Trees\ a \rightarrow Bool \\
isEmpty\ t & = & and\ [or\ (map\ isEmpty\ list\_trees) \mid (\_, list\_trees) \leftarrow unbranch\ t]
\end{array}
$$

Nous allons utiliser la structure de *Trees* pour représenter des arbres de syntaxe abstraite. Les étiquettes des noeuds sont donc les productions de la grammaire et forment ainsi un alphabet gradué. L'arité d'une production est donnée par le nombre de symboles grammaticaux en sa partie droite. Nous supposerons donc que le noeud "et" issu d'une branche étiquetée *a* possédera un nombre de successeur égal à l'arité de ce symbole. Et donc en particulier un trees est un arbre à branchement fini (et même borné aux noeuds "ou" par la taille de l'alphabet et aux noeuds "et" par la plus grande arité). Par le lemme de König un trees aura nécessairement au moins une branche infinie dès qu'il représente un ensemble infini d'arbres, c'est justement pour cette raison que nous avons introduit cette structure de données.

4.2. Automates d'arbres comme générateurs

Une co-algèbre pour le foncteur F est une application :

$$coalg : x \rightarrow [(a,\ [x])]$$

Une telle co-algèbre peut s'interpréter comme un automate d'arbres (descendant) où :

- a est le type des étiquettes des noeuds de l'arbre à reconnaitre
- x est le type des états (tous considérés comme finals) et,
- $q \rightarrow (A, \langle q_1, \cdots, q_n \rangle)$ est une transition de l'automate lorsque la paire $(A, [q_1, \cdots, q_n])$ apparait dans la liste $coalg\ \ q$.

Pour reconnaitre un arbre à l'aide de cet automate, à partir d'un état initial, on procède comme suit :

- On associe l'état initial à la racine de l'arbre.
- Si un noeud étiqueté A, associé à l'état q, possède n successeurs non encore associés à des états, et que la transition $q \rightarrow (A, \langle q_1, \cdots, q_n \rangle)$ est une transition de l'automate, alors on associe les états de q_1 jusqu'à q_n à chacun des n successeurs de ce noeud.
- L'arbre est reconnu si on a ainsi réussi à associer un état à chacun des noeuds de l'arbre.

La fonction de génération (anamorphisme) associée à une co-algèbre est définie comme suit :

$$
\begin{array}{l}
type\ Coalg\ a\ b = b \rightarrow [(a, [b])] \\
ana :: Coalg\ a\ b \rightarrow b \rightarrow Trees\ a \\
ana\ coalg\ gen = Branch\ [(a,\ map\ (ana\ coalg)\ gens) \mid (a, gens) \leftarrow coalg\ gen]
\end{array}
$$

Un élément du domaine de la co-algèbre est un germe à partir duquel l'anamorphisme engendre un trees. Lorsqu'on interprète une co-algèbre comme un automate d'arbres *auto* un germe est un état et *(ana auto init)* fournit une représentation de l'ensemble des arbres reconnus par l'automate à partir d'un état (initial) *init*. Dans la suite nous identifions les notions de co-algèbres et d'automates d'arbres. Par exemple le trees de la figure 6 représente l'ensemble des arbres reconnus par l'automate d'arbres, d'état initial q_0, dont les règles sont les suivantes :

$$\begin{array}{llll} q_0 \rightarrow (a,[q_1,q_2]) & q_1 \rightarrow (a,[q_1,q_2]) & q_2 \rightarrow (c,[]) & q_3 \rightarrow (b,[]) \\ q_0 \rightarrow (a,[q_2,q_3]) & q_1 \rightarrow (b,[]) & & q_3 \rightarrow (a,[q_2,q_3]) \end{array}$$

On peut décider si le langage reconnu par un automate est non vide lorsque l'ensemble des états accessibles à partir de l'état initial est fini. Si Q est cet ensemble d'états on appelle marquage sur Q tout sous-ensemble d'états $m \subseteq Q$ tel que pour toute règle $q \rightarrow (A,[q_1,\ldots,q_n])$, l'état q est marqué si les états q_1 à q_n le sont. Le langage reconnu par l'automate est non vide si, et seulement si, l'état initial appartient au plus petit marquage m_{min}. Plus précisément $m_{min} = \cup_{n\in\mathbb{N}} m_n$ où la suite m_n est définie par $m_0 = \emptyset$ et m_{n+1} contient les états q pour lesquels il existe une règle $q \rightarrow (A,[q_1,\ldots,q_n])$ de l'automate dont tous les états q_i en partie droite sont dans m_n. Il vient immédiatement par récurrence que $q \in m_n$ si, et seulement si, il existe un arbre de profondeur au plus n reconnu à partir de q. Évidemment on peut à chaque étape de calcul n'essayer de marquer que des états non encore marqués, ce qui revient à ne considérer pour chaque q que des arbres de profondeur minimale reconnus à partir de q. On peut présenter une variante de cet algorithme de marquage basé sur la structure de trees. Pour cela on étiquette chaque noeud par l'état de l'automate qui a servi à l'engendrer, et on élague cet arbre de la manière suivante. On considère pour chaque chemin à partir de la racine le premier noeud étiqueté par un état qui apparaît précédemment sur ce même chemin (comme l'ensemble des états accessibles à partir de l'état initial est fini une telle répétition apparaîtra au plus tard après avoir rencontré $n+1$ états où n est la cardinalité de cet ensemble d'états). On coupe alors tous les arcs issus du noeud qui le long de ce chemin porte la seconde occurrence de l'état répété. Ce noeud devient un noeud $\perp$. La figure 8 donne le résultat de cette procédure appliquée au trees de la figure 6. Dans cette figure on représente également le résultat de l'opération de nettoyage (suppression des noeuds $\perp$ internes) après élagage.

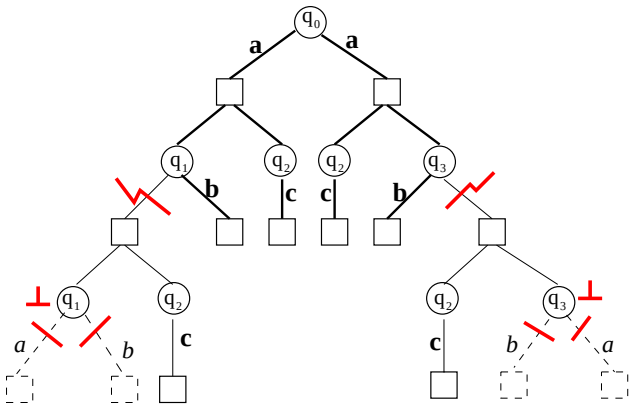

Figure 8. *en traits pleins : élagage du trees de la figure 6*

Après élagage, on obtient un arbre à branchement fini et qui ne contient que des chemins de longueur finie, par le lemme de König il ne possède donc qu'un nombre fini de noeuds. Puisqu'il est fini les fonctions *enumerate* et *isEmpty* présentées à la section précédente s'appliquent. Ainsi le trees de la figure 8 est réduit aux deux arbres suivants :

$$Node\ a\ [Node\ b\ [\,],\ Node\ c\ [\,]] \quad \text{et} \quad Node\ a\ [Node\ c\ [\,],\ Node\ b\ [\,]]$$

Proposition 6 *Un trees est vide si, et seulement si, son élagage est vide.*

Preuve. Dans un sens il est clair qu'en élaguant un arbre on ne peut que perdre des solutions. Il faut donc vérifier que l'élagage d'un trees non vide est lui même non vide. Pour cela nous introduisons une opération de réduction qui transforme tout arbre inscrit dans un trees en un arbre inscrit dans son élagage. On considère donc un arbre inscrit dans le trees. On coupe le sous arbre issu d'un noeud qui a été marqué $\perp$ dans le processus d'élagage et on greffe ce sous arbre en lieu et place de celui qui est issu du noeud qui plus haut sur le même chemin portait le même état. Comme les trees issus de ces deux noeuds sont les mêmes, puisque engendrés à partir du même état, le résultat obtenu est encore un arbre inscrit dans le trees de départ. Ce faisant la taille de l'arbre a diminué strictement. Nous itérons cette transformation tant qu'elle peut s'appliquer. Comme l'arbre de départ est fini cette procédure termine et fournit un arbre inscrit dans l'arbre élagué. □

On en déduit l'algorithme suivant qui permet de décider si le langage d'un trees engendré à partir de l'état initial d'un automate d'arbres est vide. Il s'agit d'une adaptation de la fonction *isEmpty* présentée à la section précédente consistant à appliquer cette dernière sur l'élagage du trees. Pour cela nous ajoutons un paramètre d'accumulation qui permet de mémoriser les états rencontrés le long du chemin courant afin de procéder à l'élagage lors de la première rencontre d'un état répété.

```
void :: (Eq b) ⇒ Coalg a b → b → Bool
void auto init      =  isVoid [] init      where
isVoid path state   |  elem state path = True
                    |  otherwise = and [or (map (isVoid (state : path)) states)
                                       | (_, states) ← auto state]
```

Sur le même principe la fonction suivante énumère les arbres se trouvant dans l'élagage du trees engendré à partir de l'état initial d'un automate d'arbres. C'est-à-dire que non seulement nous décidons ainsi de la vacuité du trees mais lorsque cet ensemble est non vide nous produisons la liste finie des éléments les plus "simples" de cet ensemble (c'est-à-dire ceux qui appartiennent à son élagage).

```
enum :: (Eq b) ⇒ Coalg a b → b → [Tree a]
enum auto init      =  enum_ [] init      where
enum_ path state    |  elem state path = []
                    |  otherwise = [Node elet list_tree |
                       (elet, states) ← auto state,
                       list_tree ← dist (map (enum_ (state : path)) states)]
```

4.3. Algorithme d'expansion

L'algorithme d'expansion va consister à associer un automate d'arbres à une vue dont l'état initial est la projection selon cette vue du document global, qui est l'inconnue du problème. Cet automate doit être conçu de sorte que les documents ayant comme projection la vue partielle, donnée par cet état initial, sont ceux appartenant au trees engendré par cet automate à partir de cet état initial. Les algorithmes présentés dans la section précédente permettent alors de décider si un tel document existe et dans l'affirmative de produire les documents les plus "simples" (les plus représentatifs) de l'ensemble éventuellement infini des solutions.

Dans un premier temps nous introduisons la fonction *gram*2*coalg* qui permet d'associer à une grammaire abstraite un automate d'arbres (une co-algèbre) qui reconnait les arbres de syntaxe abstraite de la grammaire.

$$\begin{array}{l} gram2coalg :: (Eq\ symb) \Rightarrow Gram\ prod\ symb \rightarrow Coalg\ prod\ symb \\ gram2coalg\ gram\ symb\ = \\ \quad [(p,\ rhs\ gram\ p) \mid p \leftarrow prods\ gram,\ symb == lhs\ gram\ p] \end{array}$$

La fonction d'expansion que nous cherchons à réaliser :

$$\begin{array}{r} expansion\ :: (Eq\ symb) \Rightarrow Gram\ prod\ symb \rightarrow (symb \rightarrow Bool) \rightarrow symb \\ \rightarrow [Tree\ symb] \rightarrow Trees\ prod \end{array}$$

doit être telle que *expansion gram view symb forest* retourne une représentation de l'ensemble des arbres de syntaxe abstraite (issus du symbole précisé) dont la forêt donnée en argument est la partie visible. Cette fonction est définie comme un anamorphisme :

$$\begin{array}{c} expansion\ gram\ view\ axiom\ ts = ana\ gramview\ (axiom,\ ts) \\ where\ gramview = gram2coalgview\ gram\ view \end{array}$$

Les états de cet automate sont des couples $\langle X, ts \rangle$ constitués d'un symbole grammatical X et d'une liste d'arbres. Les arbres devant être reconnus à partir de cet état sont tous les arbres de syntaxe abstraite dont la racine est une production ayant X en partie gauche et tels que la projection selon la vue de l'arbre de dérivation correspondant est soit ts si X est un symbole invisible ou bien la liste réduite à l'arbre (*Node X ts*) si le symbole X est visible. Si *forest* est la forêt dont on cherche à calculer l'expansion, alors de deux choses l'une : ou bien l'axiome est visible auquel cas la forêt doit être réduite à un arbre *forest*=[*Node axiom ts0*] et on prend $q_0 = \langle axiom, ts0 \rangle$ comme état initial, ou bien l'axiome n'est pas visible et l'état initial est $q_0 = \langle axiom, forest \rangle$. Supposons qu'un noeud associé à la production $p : A_0 \rightarrow A_1 \cdots A_n$ soit étiqueté par la paire $(symb, ts)$, il faut d'une part que $symb = A_0$ et qu'on puisse trouver des listes d'arbres $ts_1, \ldots, ts_n$ tels que ts puisse se décomposer sous la forme $ts = ts'_1 +\!\!+ \cdots +\!\!+ ts'_n$ de telle sorte que $ts'_i = ts_i$ si le symbole A_i est invisible et $ts_i = [Node\ A_i\ ts'_i]$ si le symbole A_i est visible. Les transitions de cet automate d'arbres sont de la forme

$$(A_0, ts) \rightarrow (p, [(A_1, ts_1), \cdots, (A_n, ts_n)])$$

dans laquelle $p : A_0 \to A_1 \cdots A_n$ est une production de la grammaire et le symbole A_0 et les suites d'arbres ts et ts_i vérifient les conditions précédentes. La fonction réalisant le calcul de cet automate d'arbres est donc la suivante :

$$
\begin{array}{l}
gram2coalgview :: (Eq\ symb) \Rightarrow Gram\ prod\ symb \to (symb \to Bool) \\
\qquad\qquad \to Coalg\ prod\ (symb,\ [Tree\ symb]) \\
gram2coalgview\ gram\ view\ (symb,\ ts) = \\
\quad [(p,\ zip\ (rhs\ gram\ p)\ tss)\ | \\
\qquad p \leftarrow prods\ gram, \\
\qquad symb == lhs\ gram\ p, \\
\qquad tss \leftarrow match\ view\ (rhs\ gram\ p)\ ts]
\end{array}
$$

Noter que la fonction *zip* effectue l'appariement de chaque symbole en partie droite de la production avec la liste correspondante d'arbres afin de constituer l'état devant être associé à l'argument correspondant dans la transition associée de l'automate d'arbres. La fonction *match* associée à une vue prend comme premier argument une liste de symboles grammaticaux $A_1 \cdots A_n$, comme second argument une suite d'arbres ts et elle génère toutes les listes $[ts_1, \ldots, ts_n]$ associées aux décompositions $ts = ts'_1 +\!+ \cdots +\!+ ts'_n$ de ts telles que $ts'_i = ts_i$ si le symbole A_i est invisible et $ts'_i = [Node\ A_i\ ts_i]$ sinon, en vue de les apparier aux symboles A_i.

$$
\begin{array}{l}
match :: (Eq\ symb) \Rightarrow (symb \to Bool) \to [symb] \to [Tree\ symb] \\
\qquad\qquad \to [[[Tree\ symb]]] \\
match\ view\ []\ ts\ =\ if\ null\ ts\ then\ [[[]]]\ else\ [] \\
match\ view\ (symb : symbs)\ ts\ = \\
\quad [(ts1 : tss)\ |\ \ (ts1, ts2) \leftarrow matchone\ view\ symb\ ts, \\
\qquad\qquad tss \leftarrow match\ view\ symbs\ ts2] \\
matchone :: (Eq\ symb) \Rightarrow (symb \to Bool) \to symb \to [Tree\ symb] \\
\qquad\qquad \to [([Tree\ symb], [Tree\ symb])] \\
matchone\ view\ symb\ \ ts\ = \\
\quad if\ view\ symb\ then \\
\qquad if (null\ ts \| ((top\ (head\ ts)) /= symb))\ then\ []\ else\ [(succ\_\ (head\ ts), tail\ ts)] \\
\quad else\ split\ ts \\
split\ []\ =\ [([],[])] \\
split\ (x : xs)\ =\ [([], x : xs)] +\!+ [(x : xs1, xs2)\ |\ (xs1, xs2) \leftarrow split\ xs]
\end{array}
$$

En dépliant le membre droit de la définition de la fonction *expansion* on obtient la reformulation suivante :

$$
\begin{array}{l}
expansion :: (Eq\ symb) \Rightarrow Gram\ prod\ symb \to (symb \to Bool) \\
\qquad\qquad \to symb \to [Tree\ symb] \to Trees\ prod \\
expansion\ gram\ view\ axiom\ ts\ =\ g\ (axiom,\ ts) \\
\quad where\ g\ (symb,\ ts)\ =\ Branch\ [(p,\ map\ g\ (zip\ (rhs\ gram\ p)\ tss))\ | \\
\qquad\qquad p \leftarrow prods\ gram, \\
\qquad\qquad symb == lhs\ gram\ p, \\
\qquad\qquad tss \leftarrow match\ view\ (rhs\ gram\ p)\ ts]
\end{array}
$$

La fonction réalisant l'appariement d'une liste d'arbres à une liste de symboles grammaticaux procède par essai-erreur en utilisant la fonction *split* qui génère toutes les façons de scinder la liste résiduelle d'arbres. Cela génère de nombreux retour-arrières pouvant affecter la performance de cet algorithme. Notons néanmoins que du fait de l'évaluation paresseuse cet espace de solutions potentielles va être exploré par nécessité. Lorsque

nous travaillons avec plusieurs vues partielles d'un même document les différents algorithmes d'expansion associés vont être synchronisés, comme nous le verrons plus bas. Elles vont donc agir comme un ensemble de coroutines et chacune d'entre elles va, au fur et à mesure que son exécution progresse, restreindre l'espace de recherche des autres. Le non déterminisme va alors être rapidement réduit et ce d'autant plus lorsque la majorité des symboles grammaticaux sont présents dans au moins une vue et lorsqu' il y a suffisamment de symboles partagés par plusieurs vues (ce qui améliore la synchronisation). Néanmoins nous pouvons améliorer les performances de l'algorithme d'expansion présenté ci-dessus. L'idée est la suivante : plutôt que d'utiliser une opération binaire de scindage (la fonction split) qui n'exploite pas l'information sur les symboles visibles se trouvant à la suite du symbole courant, nous pouvons utiliser l'ensemble des symboles visibles de la partie droite d'une production comme des éléments pivots nous permettant de découper globalement la liste d'arbres suivant un motif défini par cette partie droite. L'écriture de cette variante de la fonction *match* ne pose pas de difficultés particulières, nous ne la développerons pas ici.

4.4. Un exemple

Nous illustrons l'algorithme d'expansion en considérant à nouveau la grammaire donnée par les productions suivantes.

$$\begin{array}{lll} P_1 : A \to C\,B & P_3 : B \to C\,A & P_5 : C \to A\,C \\ P_2 : A \to \varepsilon & P_4 : B \to B\,B & P_6 : C \to C\,C \\ & & P_7 : C \to \varepsilon \end{array}$$

et la vue constituée des symboles A et B. Nous représentons dans cet exemple des listes d'arbres par leur linéarisations. Dans l'écriture de ces linéarisations, nous utilisons la parenthèse ouvrante et fermante, '(' et ')', pour représenter respectivement les symboles de Dyck ($OpenA$) et ($CloseA$) associés au symbole visible A et le crochet ouvrant et fermant, '[' et ']', pour représenter respectivement les symboles de Dyck ($OpenB$) et ($CloseB$) associés au symbole visible B. Chacune des productions de la grammaire peut alors être convertie en une famille de règles (schéma de règles) pour l'automate d'arbres de la manière suivante.

$$\begin{array}{lll} \langle A,w\rangle \longrightarrow (P_1,[\langle C,u\rangle,\langle B,v\rangle]) & \text{si } w = u[v] \\ \langle A,w\rangle \longrightarrow (P_2,[]) & \text{si } w = \varepsilon \\ \langle B,w\rangle \longrightarrow (P_3,[\langle C,u\rangle,\langle A,v\rangle]) & \text{si } w = u(v) \\ \langle B,w\rangle \longrightarrow (P_4,[\langle B,u\rangle,\langle B,v\rangle]) & \text{si } w = [u][v] \\ \langle C,w\rangle \longrightarrow (P_5,[\langle A,u\rangle,\langle C,v\rangle]) & \text{si } w = (u)v \\ \langle C,w\rangle \longrightarrow (P_6,[\langle C,u\rangle,\langle C,v\rangle]) & \text{si } w = uv \\ \langle C,w\rangle \longrightarrow (P_7,[]) & \text{si } w = \varepsilon \end{array}$$

Le premier schéma de règles , par exemple, exprime le fait qu'un arbre de syntaxe abstraite reconnu à partir de l'état $\langle A,w\rangle$ peut être obtenu en utilisant la production P_1 avec des arguments qui sont respectivement des arbres associés aux états $\langle C,u\rangle$, et $\langle B,v\rangle$ pour des mots de Dyck u et v tels que w puisse se décomposer sous la forme $w = u[v]$. Ce qui signifie dans ce cas que w doit se terminer par un crochet fermant. Comme w est une expression bien parenthésée, on sait déterminer sans ambiguïté le crochet ouvrant correspondant et les mots u et v sont ainsi déterminés. On dira dans ce cas que le motif associé à ce schéma de règles est déterministe, ce qui est le cas pour tous les schémas sauf pour celui associé à la production P_6.

La trace visible de l'arbre de la figure 1 est $(()[()()])$. Appliquons notre algorithme d'expansion à cette trace visible. Comme l'axiome A est un symbole visible, associé aux parenthèses '(' et ')', l'état initial de l'automate est $q_0 = \langle A, ()[()()]\rangle$. En se restreignant aux états accessibles à partir de q_0 nous obtenons l'automate d'arbres fini suivant :

$$
\begin{array}{lll}
q_0 \longrightarrow (P_1, [q_1, q_2]) & & \text{avec } q_1 = \langle C, ()\rangle \text{ et } q_2 = \langle B, ()()\rangle \\
q_1 \longrightarrow (P_5, [q_3, q_4]) & & \text{avec } q_3 = \langle A, \varepsilon\rangle \text{ et } q_4 = \langle C, \varepsilon\rangle \\
q_1 \longrightarrow (P_6, [q_4, q_1]) & | \quad (P_6, [q_1, q_4]) & \\
q_2 \longrightarrow (P_3, [q_1, q_3]) & & \\
q_3 \longrightarrow (P_2, [\,]) & & \\
q_4 \longrightarrow (P_6, [q_4, q_4]) & | \quad (P_7, [\,]) &
\end{array}
$$

La figure 9 représente l'élagage (et son nettoyage) du trees engendré par l'automate d'arbres à partir de son état initial. Certaines parties non développées dans ce schéma, indiquées par des triangles d'une certaine couleur, doivent être remplacées par le sous arbre issu du noeud portant la même couleur (il s'agit des noeuds étiquetés q_4 et q_1 respectivement).

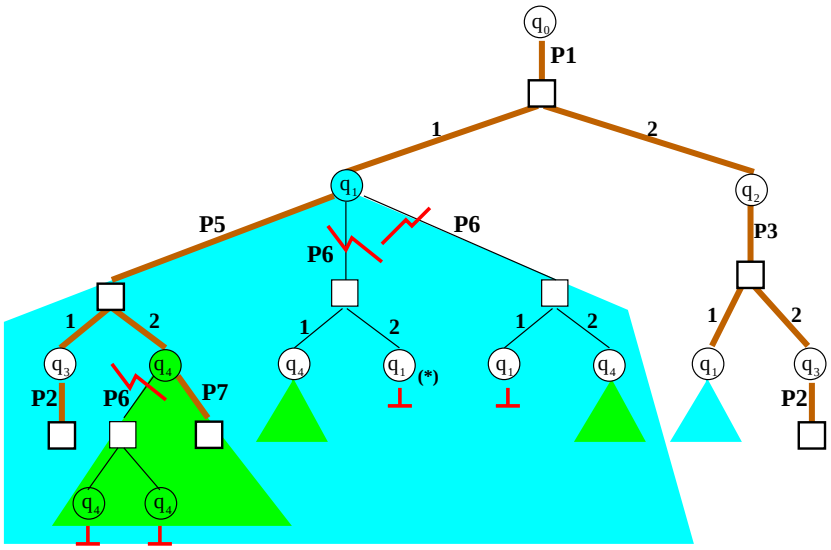

Figure 9. *élagage (et son nettoyage) du trees engendré par l'automate d'arbre associé à la vue* $(()[()()])$

Il n'y a qu'un arbre inscrit dans ce trees (indiqué en traits forts) qui n'est rien d'autre que l'arbre à partir duquel on est parti (arbre de la figure 1). Néanmoins il est facile de voir comment à partir de cette solution minimale on peut reconstruire les autres solutions en effectuant "à l'envers" l'opération de réduction opérant sur les arbres inscrits dans un trees. Par exemple considérons le chemin conduisant au noeud marqué (*). On peut couper le sous arbre issu du noeud portant la première occurrence de l'état répété (à savoir q_1) pour le greffer au noeud portant la seconde occurrence de cet état (c'est-à-dire le noeud (*)) et compléter les autres branches par des arbres inscrits aux noeuds correspondant de l'élagage. On obtient ainsi une nouvelle solution présentée à la figure 10. On pourrait reconstituer toute solution (arbre inscrit dans le trees de départ) par cette opération d"'auto-greffe" et celle similaire de "greffe croisée" à partir des solutions minimales

(c'est-à-dire inscrites dans l'élagage). Sur cette base on pourrait concevoir un algorithme énumérant de façon incrémentale les arbres engendrés par un automate à partir d'un état donné. Nous ne développerons pas cet aspect ici car nous serons davantage intéressés par la représentation de cet ensemble d'arbres par une structure coinductive de trees plutôt qu'à une énumération des éléments de cet ensemble.

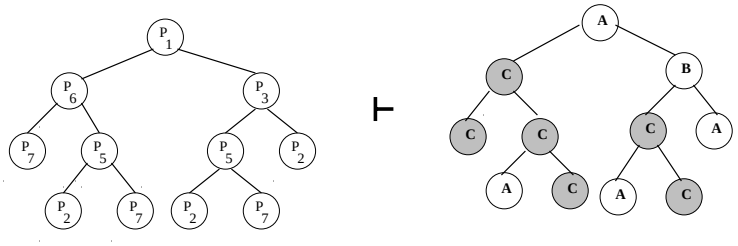

Figure 10. *une solution (arbre associé à la vue* $(()[()()])$*) obtenue par "auto-greffe" à partir de l'arbre inscrit dans l'élagage de la figure 9*

5. Cohérence d'un ensemble de vues

5.1. Le problème de la cohérence de vues

Différents outils externes peuvent manipuler des vues partielles distinctes d'un même document structuré. L'édition collaborative est une bonne illustration de cette situation : différents intervenants opèrent sur des vues distinctes ; néanmoins comme ces vues peuvent être interdépendantes, une modification opérée par un intervenant peut avoir des répercussions observables par d'autres. On voit ainsi qu'un document structuré peut agir comme une interface venant en support à la collaboration entre différents outils basés sur des vues distinctes du système. Chacune de ces vues peut d'ailleurs reposer sur un langage dédié encapsulant le savoir-métier propre à une catégorie particulière d'intervenants. Si on admet que des modifications puissent être effectuées de façon asynchrone sur les différentes vues il sera nécessaire de résoudre le problème de la *cohérence des vues* au moment de leur re-synchronisation. Le problème de la cohérence de vues consiste à décider s'il existe un document correspondant à un ensemble de vues données et dans l'affirmative à le construire (synchronisation des vues, *c.f.* fig.11).

5.2. Synchronisation de deux vues

Nous introduisons deux combinateurs : le combinateur $\langle \# \rangle$ permet de synchroniser deux *trees* ie. de calculer une représentation de l'intersection de leurs ensembles d'arbres de syntaxe abstraite ; le combinateur $\langle \S \rangle$ permet de synchroniser deux automates d'arbres (co-algèbres) afin de construire un automate reconnaissant les arbres de l'intersection des

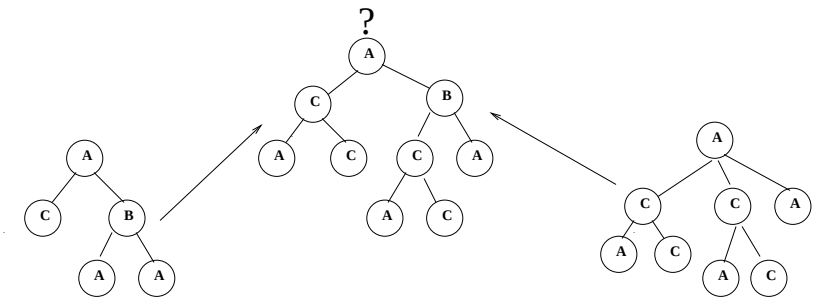

Figure 11. *Cohérence de deux vues*

ensembles d'arbres reconnus par chacun des deux automates. Le premier de ces combinateurs est donné comme suit :

$$
\begin{array}{l}
infix\ 4\ \langle\#\rangle \\
(\langle\#\rangle) :: (Eq\ a) \Rightarrow Trees\ a \rightarrow Trees\ a \rightarrow Trees\ a \\
t1\ \langle\#\rangle\ t2\ =\ Branch \\
\qquad [(a1, zipWith\ (\langle\#\rangle)\ ts1\ ts2) | \\
\qquad\qquad (a1, ts1) \leftarrow unbranch\ t1, \\
\qquad\qquad (a2, ts2) \leftarrow unbranch\ t2, \\
\qquad\qquad a1 == a2, \\
\qquad\qquad (length\ ts1) == (length\ ts2)]
\end{array}
$$

Le combinateur $\langle\#\rangle$ calcule l'intersection de deux *Trees*. C'est-à-dire, si on note $t \models u$ pour signifier que t est un arbre apparaissant dans la liste $enumerate\ u$:

Proposition 7 $\quad (t \models u \wedge t \models v) \quad \Leftrightarrow \quad t \models u \langle\#\rangle v$

Preuve. Notons que $t \models u$ revient à dire que $unode\ t\ =\ (a, [t_1, \cdots, t_n])$ et qu'il existe un élément $(a, [u_1, \cdots, u_n])$ de $unbranch\ u$ tel que $t_i \models u_i$ pour tout $1 \leq i \leq n$. Ainsi la conjonction de $t \models u$ et de $t \models v$ équivaut à l'existence de $(a, [u_1, \cdots, u_n])$ dans $unbranch\ u$ et $(a, [v_1, \cdots, v_n])$ dans $unbranch\ v$ tels que $t_i \models u_i$ et $t_i \models v_i$. Par hypothèse de récurrence ces deux dernières conditions équivalent à $t_i \models u_i \langle\#\rangle v_i$; Par définition du combinateur $\langle\#\rangle$, cela équivaut à l'existence d'un $(a, [w_1, \cdots, w_n])$ dans $unbranch\ u \langle\#\rangle v$ tel que $t \models w$, soit $t \models u \langle\#\rangle v$. □

On peut de façon similaire définir une opération de synchronisation $\langle\S\rangle$ sur les co-algèbres (automates d'arbres). Il s'agit de l'automate ayant pour états les paires d'états de chacun des automates et dont les règles sont obtenues en synchronisant les règles correspondantes de ces deux automates.

$$
\begin{array}{l}
(\langle\S\rangle) :: (Eq\ a) \Rightarrow Coalg\ a\ b \rightarrow Coalg\ a\ c \rightarrow Coalg\ a\ (b,\ c) \\
(coalg1\ \langle\S\rangle\ coalg2)\ (etat1,\ etat2)\ =\ [(a1,\ zip\ etats1\ etats2)\ | \\
\qquad\qquad (a1,\ etats1)\ \leftarrow\ coalg1\ etat1, \\
\qquad\qquad (a2,\ etats2)\ \leftarrow\ coalg2\ etat2, \\
\qquad\qquad a1\ ==\ a2, \\
\qquad\qquad length\ etats1\ ==\ length\ etats2]
\end{array}
$$

Ainsi cet automate composite reconnaît, à partir de la paire constituée des états initiaux des deux automates, l'intersection des ensembles d'arbres reconnus par chacun des deux automates. C'est-à-dire :

$$ana\ (coalg_1 \langle\S\rangle coalg_2)\ (init1,\ init2) = (ana\ coalg_1\ init1) \langle\#\rangle (ana\ coalg_2\ init2)$$

5.3. Algorithme de cohérence

L'algorithme de cohérence de deux vues est obtenu en combinant les expansions selon chacune des vues avec le combinateur $\langle\#\rangle$ procurant la représentation co-inductive de l'intersection des ensembles d'arbres correspondants.

$$\begin{aligned}
&coherence :: (Eq\ prod,\ Eq\ symb) \Rightarrow Gram\ prod\ symb \rightarrow (symb \rightarrow Bool) \\
&\qquad\qquad \rightarrow (symb \rightarrow Bool) \rightarrow symb \\
&\qquad\qquad \rightarrow [Tree\ symb] \rightarrow [Tree\ symb] \rightarrow Trees\ prod \\
&coherence\ gram\ view_1\ view_2\ axiom\ ts_1\ ts_2 = trees_1 \langle\#\rangle trees_2 \\
&\quad where \quad trees_1 = expansion\ \ gram\ \ view_1\ \ axiom\ \ ts_1 \\
&\qquad\quad\ \ trees_2 = expansion\ \ gram\ \ view_2\ \ axiom\ \ ts_2
\end{aligned}$$

De façon équivalente on peut utiliser les constructions correspondantes sur les automates d'arbres :

$$\begin{aligned}
&coherence :: (Eq\ prod,\ Eq\ symb) \Rightarrow Gram\ prod\ symb \rightarrow (symb \rightarrow Bool) \\
&\qquad\qquad \rightarrow (symb \rightarrow Bool) \rightarrow symb \\
&\qquad\qquad \rightarrow [Tree\ symb] \rightarrow [Tree\ symb] \rightarrow Trees\ prod \\
&coherence\ gram\ view_1\ view_2\ axiom\ ts_1\ ts_2 = \\
&\qquad ana\ (gview_1\ \langle\S\rangle\ gview_2)\ ((axiom,\ ts_1),\ (axiom,\ ts_2)) \\
&\qquad\quad where \quad gview_1 = gram2coalgview\ \ gram\ \ view_1 \\
&\qquad\qquad\quad\ \ gview_2 = gram2coalgview\ \ gram\ \ view_2
\end{aligned}$$

À partir d'une grammaire abstraite donnée, de deux vues, de deux approximations associées à ces vues, et de l'axiome de la grammaire cette fonction produit une représentation de l'ensemble des expansions cohérentes de ces deux approximations. Elle procède en calculant les automates d'arbres associés à chaque vue ($gview_1$, et $gview_2$), puis les synchronisent ($gview_1\ \langle\S\rangle\ gview_2$) pour obtenir l'automate d'arbres qui génère via l'anamorphisme correspondant une représentation de l'intersection des deux expansions.

6. Conclusion

Nous avons apporté une solution au problème de la cohérence des vues dans les grammaires algébriques abstraites dans le cas où une vue est assimilée à un sous-ensemble de catégories syntaxiques. Cette solution repose sur le codage d'ensembles (infinis) d'arbres de dérivation par une structure de données paresseuse. Il s'agit donc d'un exemple d'utilisation de structures co-inductives et de co-algèbres ce qui est encore assez peu répandu en programmation surtout lorsque l'on songe que les notions duales de structures inductives et algèbres sont omniprésentes en programmation. Or il se trouve que c'est dans la manipulation des structures co-inductives que le mécanisme d'évaluation paresseuse présente tout son intérêt et son originalité (c'est d'ailleurs pourquoi on utilise l'expression de

structures de données paresseuses pour qualifier les structures de données co-inductives). Notre algorithme présente ainsi un exemple original d'utilisation du mécanisme d'évaluation paresseuse.

La partie centrale de notre algorithme, donnant l'expansion d'une vue partielle, est par ailleurs une utilisation d'un analyseur syntaxique fonctionnel [15, 23]. Néanmoins un lexème est ici un arbre alors que tous les analyseurs fonctionnels que nous avons rencontrés dans la littérature ont pour lexèmes de simples symboles n'ayant aucune structure interne exploitée par l'algorithme. Nous pensons qu'il y a là matière à réflexion sur cette algorithmique des analyseurs fonctionnels. Notons à ce propos qu'on peut toujours considérer une grammaire algébrique comme une grammaire abstraite dont les catégories syntaxiques sont tous les symboles grammaticaux, terminaux ou non-terminaux en ajoutant une production $A \to \varepsilon$ pour tout symbole terminal A. Si on considère la vue associée aux symboles terminaux, la projection associée à cette vue produit la suite des symboles terminaux (lexèmes) ; c'est-à-dire le feuillage d'un arbre de syntaxe abstraite. L'algorithme d'expansion n'est alors rien d'autre qu'un algorithme d'analyse produisant les arbres de syntaxe abstraite associés à une suite donnée de lexèmes. C'est en ce sens que notre algorithme d'expansion est un algorithme d'analyse généralisé opérant sur des données partiellement structurées. À ce propos il y a deux aspects que nous voudrions regarder plus en détail. D'une part nous voudrions concevoir un jeu de combinateurs fonctionnels, similaires aux combinateurs fonctionnels d'analyse [15, 23], afin d'obtenir un langage dédié à la description des vues : lorsqu'un utilisateur décrit une notion de vue en utilisant de tels combinateurs il produira un code Haskell pour l'algorithme d'expansion correspondant. Le second point vient du fait que les algorithmes d'analyse sont des exemples typiques d'*inversion de programmes* et nous voudrions voir dans quelle mesure les techniques présentées en [2, 28, 29] pourraient s'appliquer à notre algorithme d'expansion.

La solution que nous avons proposée est particulièrement simple : son code Haskell a été intégralement donné dans le corps de cet article. Il a été testé sur quelques exemples dont celui indiqué dans cet article. La réponse a toujours été instantanée mais il est vrai que nous ne l'avons pas utilisé avec des grammaires de tailles importantes.

Dans la poursuite de ce travail, en dehors de la résolution du problème de couverture d'une grammaire algébrique par des vues, nous nous intéresserons à la prise en compte d'attributs et de dépendances fonctionnelles entre attributs ; c'est-à-dire à l'extension de ce travail au cadre des grammaires attribuées. Il n'y a là *a priori* pas d'obstacles majeurs si on se repose en particulier sur la traduction des grammaires attribuées en programmes fonctionnels [24, 16, 4]. Cette extension peut être très intéressante d'un point de vue pratique car un attribut d'un symbole grammatical visible peut dépendre de la valeur d'attributs de symboles invisibles. Les attributs apparaissent ainsi comme des moyens de communication entre les différentes vues partielles. Les grammaires attribuées pourraient ainsi fournir des mécanismes de coordination. En préalable à cette étude il nous parait utile d'essayer de définir le quotient d'une grammaire par une vue en vue de caractériser les vues partielles licites. L'idée est d'identifier dans la grammaire de départ les symboles grammaticaux équivalents en ce sens que les ensembles des vues partielles des documents licites engendrés à partir de deux tels symboles dans le même contexte sont les mêmes. Pour l'instant la validité d'une vue partielle par rapport à la grammaire de départ est vérifiée par l'algorithme d'expansion ; c'est-à-dire en s'assurant qu'il existe au moins un document licite vis-à-vis de cette grammaire se projetant sur cette vue partielle. On pourrait alors étendre notre travail dans le cadre des grammaires attribuées en réalisant un

couplage entre les ensembles d'attributs de la grammaire globale d'une part et ceux de la grammaire projetée d'autre part.

Même dans le cas où la grammaire projetée ne serait en mesure de caractériser qu'un sur-ensemble des vues partielles licites elle constituerait un outil utile en vue d'associer un langage dédié et des outils spécifiques à une vue (pour l'édition et la manipulation d'un document à partir d'une vue donnée). En particulier il n'est pas trop difficile à partir d'une grammaire abstraite de concevoir un éditeur pour les structures correspondantes en utilisant la structure de zipper associée [22]. Les opérations de navigation s'implémentent aisément avec la structure de zipper, on y adjoint des opérations de copier-coller et des fonctions d'édition associées à chaque production de la grammaire. Il serait intéressant de pouvoir associer de façon générique un éditeur à la donnée conjointe d'une grammaire et d'une vue partielle pour cette grammaire. De façon plus générale il serait intéressant d'étudier les grammaires abstraites avec vues : problème d'équivalence (même famille de forêts visibles), minimisation ...

7. Bibliographie

[1] Serge Abiteboul. On Views and XML. Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems, 1–9, 1999.

[2] Sergei M. Abramov, Robert Glück. The universal resolving algorithm : inverse computation in a functional language. Proceedings of Mathematics of Program Construction 2000. R. Backhouse and J. N. Oliveira, Eds. Springer-Verlag Lecture Notes in Computer Science vol. 1837, pp. 187–212. 2000.

[3] Rajeev Alur, P. Madhusudan. Visibly Pushdown Languages. Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. Chicago, IL, USA, 202-211, 2004.

[4] Kevin S. Backhouse. A Functional Semantics of Attribute Grammars. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science, Springer-Verlag, 2002.

[5] Roland Carl Backhouse, Roy L. Crole, Jeremy Gibbons, Eds. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, International Summer School and Workshop, Oxford, UK, April 10-14, 2000. Lecture Notes in Computer Science vol. 2297, Springer-Verlag, 2002.

[6] Jean Berstel, Luc Boasson. XML Grammars. In Proc. *Mathematical Foundations of Computer Science* (MFCS2000) (M. Nielsen and B. Rovan, Eds.), Springer Verlag Lecture Notes in Computer Science vol. 1893, pp. 182-191, 2000.

[7] Jean Berstel, Luc Boasson. Balanced Grammars and Their Languages. In *Formal and Natural Computing : Essays Dedicated to Grzegorz Rozenberg*, Springer Lecture Notes in Computer Science vol 1243, pp. 135-150, 1997.

[8] R.S. Bird. *Introduction to Functional Programming using Haskell*, 2nd edition, Prentice Hall, 1998.

[9] V. Braganholo. Updating relational databases through xml views. Thesis proposal - in preparation, PPGC-UFRGS, Porto Alegre, 2002.

[10] Y. Chen and T. Ling and M. Lee. Designing Valid XML Views. ER Conference, 2002

[11] Krzysztof Czarnecki, Ulrich W. Eisenecker. *Generative Programming : Methods, Tools, and Applications*. Addison Wesley, 2000.

[12] Antony Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.

[13] Sergey Dimitriev. Language Oriented Programming : The Next Paradigm. http ://www.onboard.jetbrains.com/articles/04/lop/index.html

[14] Kees Doets, and Jan van Eijck. *The Haskell Road to Logic, Maths, and Programming*. King's College Publication, London, 2004.

[15] J. Fokker. Functional Parsers. In J. Jeuring and E. Meijer, editors, *First International School on Advanced Functional Programming*, volume 925 of Lecture Notes in Computer Science, 1-23, Springer-Verlag, 1995.

[16] M. Fokkinga, J. Jeuring, L. Meertens, E. Meijer. A Translation from Attribute Grammars to Catamorphisms. *The Squiggolist*, 2(1) :20-26, 1991.

[17] Geremy Gibbons, and Oege de Moor. *The Fun of Programming*. Palgrave, 2002.

[18] Cordelia Hall, and John O'Donnell. *Discrete Mathematics using a Computer*. Springer, 2000.

[19] T. Lindholm. A Three-way Merge for XML Documents. ACM Symposium on Document Engineering, October 2004.

[20] Paul Hudak, John Peterson, Joseph H. Fasel. *A Gentle Introduction to Haskell 98.*, 1999.

[21] Paul Hudak. *The Haskell School of Expression : Learning Functional Programming through Multimedia*. Cambridge University Press, New-York, 2000.

[22] G. Huet. The Zipper. *Journal of Functional Programming* 7(5), Sept 1997, pp. 549-554.

[23] G. Hutton, E. Meijer. Monadic Parsing in Haskell. J. Functional Programming 8(4), pp. 437-444, 1998.

[24] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In G. Kahn, ed, *Proc. of 3rd Int. Conf. on Functional Programming and Computer Architecture*, FPCA'87, vol. 274 of Lecture Notes in Computer Science, 154-173, Springer-Verlag, 1987.

[25] Donald E. Knuth. A Characterization of Parenthesis Languages. *Information and Control*, 11(3) :269-289, 1967.

[26] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical System Theory*, 2(2) : 127-145, June 1968.

[27] Gerald W. Manger. Generic Algorithm for Merging SGML/XML-Instances. Proceedings of XML Europe, Berlin, Germany, 2001.

[28] Shin-Cheng Mu. A Calculational Approach to Program Inversion. PhD Thesis, Oxford University Computing Laboratory, 2003.

[29] Shin-Cheng Mu, Richard Bird. Theory and applications of inverting functions as folds. Science of Computer Programming (Special Issue for Mathematics of Program Construction, vol. 51, pp. 87–116, 2003.

[30] S.C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In Seventh International Conference on Mathematics of Program Construction (MPC 2004), Stirling, Scotland, July 2004. Springer Verlag, LNCS.

[31] R. McNaughton. Parenthesis Grammars. *Journal of the ACM*, 14(3) :490-500, 1967.

[32] J. Paakki. Attribute Grammar Paradigms–A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2) : 196-255, June 1995.

[33] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries - The Revised Report*. Cambridge University Press, 2003.

[34] Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In B. Randell (Ed.) The future of Software, Proceeding of the joint International Computer Limited. University of Newcastle seminar (Also : Technical Report MSR-TR-95-52, Microsoft Research, redmond), 1995.

[35] Simon Thompson. *Haskell, the Craft of Functional Programming*, 2nd edition. Addison Wesley, 1999.

[36] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types*, Faculty of Mathematics, University of Tartu, Estonia, 2000.

[37] Eric Van Wyk, Oege de Moor, Ganesh Sittampalam, Ivan Sanabria Piretti, Kevin Backhouse, Paul Kwiatkowski. Intentional Programming : A Host of Language Features. Oxford University Computing Laboratory, PRG-RR-01-21, 2001.

[38] Haskell, A Purely Functional Language. http ://www.haskell.org

[39] XQuery : http ://www.w3.org/TR/xquery

[40] XSL : http ://www.w3.org/TR/xsl/

Annexe 1 : quelques notations en Haskell

Cette annexe n'est évidemment pas une introduction au langage Haskell pour lequel nous renvoyons le lecteur intéressé au document de référence [33], et à l'introduction [20] tous les deux accessibles sur le site officiel du langage Haskell (www.haskell.org) ainsi qu'aux divers livres d'introduction à la programmation fonctionnelle utilisant Haskell comme support (les plus classiques sont [8, 12, 35] mais le lecteur pourra aussi consulter [21, 14, 17, 18] avec intérêt). Les éléments qui suivent devraient néanmoins suffire pour une bonne compréhension du code Haskell présenté dans notre article.

Quelques combinateurs Haskell

length donne la longueur d'une liste : *length* $[1,3,5,7]$ vaut 4.

head, tail donnent respectivement l'élément de tête et la liste résiduelle d'une liste non vide : *head* $[1,2,3]$ vaut 1 et *tail* $[1,2,3]$ vaut $[2,3]$.

elem teste si un élément se trouve dans une liste : *elem* 2 $[1,3,5,7]$ vaut *False*.

concat, $++$ permettent de concaténer des listes : $xs++ys$ est la concaténation des listes *xs* et *ys* ; si *xss* est une liste de listes, *concat xss* est la concaténation des listes se trouvant dans *xss*.

and, or retourne la conjonction (respectivement la disjonction) des éléments d'une liste de booléens ; cas particulier de la liste vide : *or* $[]$ vaut *False* et *and* $[]$ vaut *True* (éléments neutres respectifs).

map retourne la liste obtenue en appliquant une fonction à chaque élément d'une liste : $map :: (a \to b) \to [a] \to [b]$, par exemple $map\ (*2)\ [0,1,2,3,4]$ vaut $[0,2,4,6,8]$.

zip apparie les éléments de même rang des deux listes arguments. Si les deux listes n'ont pas la même longueur la génération de la liste résultante s'arrête lorsque la plus courte des deux listes est épuisée : $zip :: [a] \to [b] \to [(a,b)]$, par exemple $zip\ [1,2,3,4]\ ['a','b','c']$ vaut $[(1,'a'),(2,'b'),(3,'c')]$.

zipWith est similaire à zip sauf qu'au lieu de retourner des paires on retourne les résultats obtenus en appliquant une fonction à deux arguments. Ainsi *zipWith f xs ys* équivaut à

$$map\ (\backslash(x,y) \to f\ x\ y)\ (zip\ xs\ ys)$$

par exemple $and\ (zipWith\ (<=)\ xs\ (tail\ xs))$ teste si une liste est ordonnée.

Schéma de compréhension des listes

Cette notation très intuitive s'inspire du schéma de compréhension des ensembles. Nous la présentons à l'aide de quelques exemples qui devraient suffire à en saisir les usages. Sous sa forme de base, on écrit

$$[f\ x \mid x \leftarrow xs]$$

pour représenter la liste des éléments $f(x)$ lorsque la variable x parcourt la liste donnée par l'expression xs (appelée *générateur de liste*). Cette expression est donc équivalente à l'expression $map\ f\ xs$. A partir de cette notation de base plusieurs extensions sont possibles :

avec plusieurs générateurs On peut utiliser plusieurs générateurs. Chaque générateur peut alors dépendre des variables précédemment introduites. L'ordre des générateurs est important et on peut interpréter le processus de génération de la liste résultante comme une exécution de boucle imbriquées. Par exemple

- $[(x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$ vaut $[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]$,
- $[(x,y) \mid x \leftarrow [1,2], y \leftarrow [\,'a','b']]$ vaut $[(1,'a'),(1,'b'),(2,'a'),(2,'b')]$ et
- $[(x,y) \mid y \leftarrow [\,'a','b'], x \leftarrow [1,2]]$ vaut $[(1,'a'),(2,'a'),(1,'b'),(2,'b')]$.

avec des gardes On peut utiliser des gardes (prédicats) pour filtrer certaines solutions. Par exemple $[x\ 'mod'\ 3 \mid x \leftarrow [12,11,9,4,13,20,7], even\ x]$ vaut $[0,1,2]$ (les restes dans la division par 3 des nombres pairs de la liste argument).

avec des motifs On peut aussi remplacer une variable par un motif (*pattern*) chaque élément de la liste est alors filtré selon le motif, seuls les éléments pour lequel le filtrage (*pattern matching*) réussit sont pris en compte et les variables du motif sont alors instanciées en conséquence. Par exemple $[a \mid Node\ a\ [] \leftarrow ts]$ retourne la liste les étiquettes des arbres pris dans une liste ts et qui sont réduits à une feuille ; cette expression est équivalente à $[a \mid Node\ a\ ts' \leftarrow ts, ts' == []]$.

Et pour conclure quelques exemples combinant ces différents éléments :

concat : une façon d'écrire le combinateur *concat* :

$$concat\ xss\ =\ [x \mid xs \leftarrow xss,\ x \leftarrow xs]$$

zipWith : une autre façon d'écrire *zipWith* en terme de *zip* :

$$zipWith\ f\ xs\ ys\ =\ [f\ x\ y \mid (x,y) \leftarrow zip\ xs\ ys]$$

et donc d'écrire la fonction qui teste si une liste est triée :

$$sorted\ xs\ =\ and\ [x <= y \mid (x,y) \leftarrow zip\ xs\ (tail\ xs)]$$

positions : une fonction $positions :: (Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow [Int]$ qui retourne la liste des positions où un élément apparaît dans une liste (e.g. $positions\ 0\ [1,0,0,1,0,1,1,0]$ vaut $[2,3,5,8]$) :

$$positions\ x\ xs\ =\ [i \mid (x',i) \leftarrow zip\ xs\ [1..],\ x' == x]$$

crible d'Eratostène : fonction qui génère la liste (infinie) des nombres premiers en utilisant le crible d'Eratostène :

$$\begin{array}{lcl} premiers & = & crible\ [2..] \\ crible\ (n:ns) & = & n : (crible\ [m \mid m \leftarrow ns, m\ 'mod'\ n\ \neq 0]) \end{array}$$

Annexe 2 : quelques fonctions de base sur les données structurées

Dans cette annexe nous décrivons quelques fonctions de base pour la manipulation des données dont la structure est conforme à une grammaire algébrique abstraite. Sous l'hypothèse que *chaque production est caractérisée par la donnée conjointe de sa partie gauche et de sa partie droite* on peut représenter une structure conforme à une telle grammaire de trois façons équivalentes : une représentation "interne" (son arbre de syntaxe abstraite), une représentation "externe" (son arbre de dérivation), et la linéarisation de cette dernière sous la forme d'un mot d'un langage de Dyck (une représentation à la XML). Les fonctions présentées dans cette annexe sont pour l'essentiel des traductions entre ces différentes représentations. L'algorithmique sous-jacente est parfaitement standard mais nous souhaitions fournir ce code par soucis de complétude afin qu'un lecteur souhaitant expérimenter notre solution au problème de la cohérence de vues puisse le faire sans avoir à coder de fonctions supplémentaires. En particulier l'algorithme d'expansion prend en entrée une liste d'arbres de syntaxe abstraite et non leur sérialisation ; pour développer des exemples pratiques il est cependant plus naturel de partir de versions sérialisées (à la XML) d'où l'intérêt de disposer d'un algorithme d'analyse produisant un arbre (ou une forêt) à partir de sa sérialisation. Les lecteurs non familiers de Haskell peuvent utiliser les fonctions ci-dessous sans se préoccuper de la façon dont elles sont codées ; cela ne leur créera aucune gêne dans la compréhension de l'article.

La fonction suivante recherche une production donnée par ses parties droite et gauche sous l'hypothèse, que nous faisons dans l'ensemble de l'article, selon laquelle *chaque production est caractérisée par la donnée conjointe de sa partie gauche et de sa partie droite*.

```
isProd :: (Eq symb) ⇒ Gram prod symb → symb → [symb] → Maybe prod
isProd gram symb symbs =
  case [p | p ← prods gram, (symb == lhs gram p), (symbs == rhs gram p )] of
      [] → Nothing
      [p] → Just p
      otherwise → error "two productions with identical left and right hand sides"
```

Sous l'hypothèse précédente, un document structuré conforme à une grammaire peut être représenté de façon équivalente par un arbre de dérivation ou par un arbre de syntaxe abstraite. Le premier est un arbre dont les noeuds sont étiquetés par les symboles grammaticaux et le second est un arbre dont les noeuds sont étiquetés par les productions de la grammaire. Un arbre de syntaxe abstraite est ainsi un arbre étiqueté par les productions de la grammaire dont la conformité est donnée par la fonction suivante

```
isAST :: (Eq symb) ⇒ Gram prod symb → Tree prod → symb → Bool
isAST gram (Node p ts) symb =
    if (symb == left) && (length ts) == (length right)
        then and (zipWith (isAST gram) ts right)
        else False
        where  left = lhs gram p
               right = rhs gram p
```

La représentation intentionnelle d'un document, utilisée en interne par le système, est un tel arbre de syntaxe abstraite (décoré par des attributs). Néanmoins les outils externes,

comme les outils d'édition, utilisent plutôt la représentation sous forme d'arbre de dérivation dans la mesure où ils sont définis en terme des différentes catégories syntaxiques en jeu et non en terme des productions de la grammaire qui n'ont pas de signification directe pour eux. Le changement de représentation est défini par la fonction :

$$\begin{array}{l} ast2der :: (Eq\ symb) \Rightarrow Gram\ prod\ symb \rightarrow Tree\ prod\ \rightarrow Tree\ symb \\ ast2der\ gram\ (Node\ p\ ts) = Node\ (lhs\ gram\ p)\ (map\ (ast2der\ gram)\ ts) \end{array}$$

Un arbre de syntaxe abstraite peut être reconstitué à partir de l'arbre de dérivation qui lui est associé. La fonction suivante teste si un arbre dont les noeuds sont étiquetés par des symboles grammaticaux est conforme à la grammaire (c'est-à-dire est un arbre de dérivation) et retourne alors l'arbre de syntaxe abstraite qui lui est associé. [2]

$$\begin{array}{l} der2ast :: (Eq\ symb)\ \Rightarrow\ Gram\ prod\ symb \rightarrow Tree\ symb \rightarrow Maybe\ (Tree\ prod) \\ der2ast\ gram\ (Node\ symb\ ts) = \\ \quad do \quad p\ \leftarrow isProd\ gram\ symb\ (map\ top\ ts) \\ \qquad\quad ts'\ \leftarrow mapM\ (der2ast\ gram)\ ts \\ \qquad\quad return\ (Node\ p\ ts') \end{array}$$

Décrivons l'algorithme d'analyse qui doit permettre de reconstituer un arbre ou une forêt à partir de son codage par un mot de Dyck. On s'aperçoit aisément que cette analyse est déterministe et on introduit par conséquent le type suivant pour les analyseurs :

$$newtype \quad Parser\ s\ a\ =\ MkP\ ([s] \rightarrow Maybe(a,[s]))$$

La fonction appliquant un analyseur $p :: Parser\ s\ a$ à une chaîne d'entrée $xs :: [s]$ est donnée par

$$\begin{array}{l} apply\ ::\ Parser\ s\ a \rightarrow [s] \rightarrow Maybe(a,[s]) \\ apply\ (MkP\ f)\ xs\ =\ f\ xs \end{array}$$

$apply\ p\ xs\ =\ Nothing$ lorsque l'analyse échoue et $apply\ p\ xs\ =\ Just\ (a,xs')$ si l'analyse produit le résultat a à partir d'un préfixe du mot xs ; le mot xs' est la partie du mot d'entrée

2. On utilise ici la monade *Maybe* pour la composition des fonctions (applications partielles) :

$$\begin{array}{l} data\ Maybe\ a\ =\ Just\ a\ |\ Nothing \\ instance\ Monad\ Maybe\ where \\ --\ return\ ::\ a \rightarrow Maybe\ a \\ return\ x\ =\ Just\ x \\ --\ (>>=)\ ::\ Maybe\ a\ \rightarrow (a \rightarrow Maybe\ b) \rightarrow (Maybe\ b) \\ (Just\ x)\ >>=\ k\ =\ k\ x \\ Nothing\ >>=\ k\ =\ Nothing \end{array}$$

Le combinateur *mapM*, tel qu'utilisé ici, applique une fonction à une liste de valeurs, le résultat est défini lorsque cette fonction est définie en chacun des arguments apparaissant dans la liste et retourne alors la liste des images par f de ces éléments. Ce combinateur est néanmoins, de façon plus générale, défini pour toute structure de monade :

$$\begin{array}{l} mapM :: (Monad\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b] \\ mapM\ f\ [\,]\ =\ return\ [\,] \\ mapM\ f\ (x : xs)\ =\ do \quad a \leftarrow f\ x \\ \qquad\qquad\qquad\qquad\quad bs \leftarrow mapM\ f\ xs \\ \qquad\qquad\qquad\qquad\quad return\ (a : bs) \end{array}$$

(suffixe) non consommée par le processus d'analyse. Pour composer des fonctions on utilise la monade *Maybe* des résultats partiels. Les analyseurs déterministes ont également une structure de monade donnée par

```
instance Monad (Parser s) where
-- return :: a → Parser s a
return x = MkP f   where   f xs = Just (x,xs)
-- (>>=) :: Parser s a → (a → Parser s b) → (Parser s b)
p >>= q = MkP f
 where   f xs =   do   (x,ys) ← apply p xs
                       (y,zs) ← apply (q x) ys
                       return (y,zs)
```

La fonction suivante reconnaît un mot de Dyck premier et retourne l'arbre correspondant

```
primeDyck   ::   (Eq symb) ⇒ Parser (Dyck symb) (Tree symb)
primeDyck   =    do   symb ← open
                      ts ← dyck
                      close symb
                      return (Node symb ts)
```

Cette définition utilise la fonction reconnaissant une parenthèse ouvrante

```
open :: Parser (Dyck symb) symb
open = MkP f      where
 f ((Open symb) : xs)   =   Just (symb,xs)
 f _                    =   Nothing
```

celle reconnaissant une parenthèse fermante associée à un symbole donné

```
close :: (Eq symb) ⇒ symb → Parser (Dyck symb) ()
close symb = MkP f      where
 f ((Close symb') : xs)   =   if (symb' == symb) then Just ((),xs) else Nothing
 f _                      =   Nothing
```

et l'analyseur reconnaissant des mots de Dyck et retournant la forêt correspondante

```
dyck :: (Eq symb) ⇒ Parser (Dyck symb) [Tree symb]
dyck = many primeDyck
many :: (Eq a,Eq s) ⇒ Parser s a → Parser s [a]
many p = do{t ← p ; ts ← many p ; return (t : ts)} `orelse` return []
orelse :: Parser s a → Parser s a → Parser s a
(MkP f) `orelse` (MkP g) = (MkP h)   where
    h xs = if (isNothing fxs) then g xs else fxs   where
        fxs = f xs
        isNothing Nothing = True
        isNothing (Just x) = False
```

Nous demandons de surcroît que l'analyse consomme entièrement la chaîne d'entrée :

```
analyse      ::   (Eq symb) ⇒ [Dyck symb] → Maybe [Tree symb]
analyse xs   =    do   (ts,[]) ← apply dyck xs
                       return ts
```