Une approche grammaticale pour la fusion des réplicats partiels d'un document structuré : application à l'édition coopérative asynchrone

Maurice Tchoupe Tchendji

Université de Yaoundé 1
Département d'Informatique

Université de Rennes 1
Ecole doctorale Matisse

Numéro d'ordre de la thèse : 3899

Une approche grammaticale pour la fusion des réplicats partiels d'un document structuré : application à l'édition coopérative asynchrone

THÈSE

présentée et soutenue publiquement le 31 août 2009

pour l'obtention du

Doctorat de l'université de Rennes 1

Doctorat/PhD de l'université de Yaoundé 1

(spécialité informatique)

par

TCHOUPÉ TCHENDJI Maurice

Matricule : 99Z300

DEA en Informatique

Composition du jury :

Président :	Maurice TCHUENTÉ,	Professeur, Université de Yaoundé 1
Rapporteurs :	Jean Claude DERNIAME	Professeur, LORIA, Nancy
	Didier PARIGOT	Chargé de recherche, INRIA, Sophia A.
	Mokhtar SELLAMI	Professeur, Université d'Annaba
Examinateurs :	Georges-E. KOUAMOU	Chargé de Cours, Univ. de Yaoundé 1
Directeurs :	Éric BADOUEL	Chargé de Recherche, INRIA, Rennes
	Claude TANGHA	Maître de Conférences, Univ. de Yaoundé 1

Remerciements

Le travail présenté ici est la partie émergée d'un iceberg dont la base repose sur les efforts, la patience, la disponibilité, la sollicitude ... de bien des personnes physiques ou morales. Sachant ne pas pouvoir trouver des mots assez justes pour leur exprimer toute ma gratitude, je voudrais ici juste mentionner ces personnes et le rôle qu'elles ont joué dans le cadre de ce travail. Je remercie donc très chaleureusement :

- Dr. Eric Badouel chargé de Recherche INRIA/Rennes et Pr. Claude Tangha Maître de Conférences ENSP/UYI, pour avoir été plus que des directeurs de thèse.
- Pr. Maurice TCHUENTÉ, pour avoir accepté de présider le jury.
- Dr. Didier PARIGOT, Pr. Jean Claude DERNIAME et Pr. Mokhtar SELLAMI pour avoir accepté de rapporter ce travail.
- Dr. Georges-E. KOUAMOU pour avoir accepté de participer au jury de soutenance en tant qu'examinateur.
- L'INRIA (à travers le projet SARIMA et l'équipe S4/IRISA) et le SCAC qui ont financé cette thèse.
- Les membres de l'équipe S4 de l'IRISA ainsi que l'assistante de ce projet (Laurence Dinh) pour l'accueil qu'ils m'ont toujours réservé au cours de mes différents séjours à Rennes.
- Dr. Didier Parigot pour l'accueil qu'il m'a réservé lors du séjour effectué à Sophia Antipolis au sein de l'équipe SmartTools.
- Les enseignants des universités de Dschang et de Yaoundé I qui ont énormément contribué à ma formation.
- Les responsables du LABOMAT de l'université de Yaoundé I pour m'avoir accepté comme membre de leur laboratoire.
- Mes parents Raymond et Jeanne pour toute la peine qu'ils se sont donné depuis mes tous premiers jours pour assurer mon éducation.
- Mes frères et soeurs (Gaétan, Rachel, Mayeul, Thomas, Vianney, Céline) pour leur soutien inconditionnel.
- Les amis de Rennes et aussi du Cameroun pour les moments fraternels passés ensemble (Bernard, Rodrigue, Guy, Vigny, Anne Marie, Francine, Anicet, ...). Les camarades de la chorale Inter-Peule de Rennes (Félix, Aurélie, Régis, ...) pour les bons moments passés ensemble.
- Steve et Chris dont la venue au monde pendant la période de préparation de cette thèse a été le levain qui m'a aidé à faire lever la pâte.
- Last but surely the first, ma très tendre épouse Rosy Pascale qui a su être là en prodiguant le conseil et le soutien moral qu'il fallait au moment voulu.

Dédicaces

Aux membres de ma jeune famille : Chris Anderson, Steve Daryl, Rosy P.

Table des matières

Remerciements i
Dédicaces ii
Résumé vii
Abstract viii

Introduction **1**
Des documents et des hommes 1
Fusion des réplicats partiels 3
Réconciliation des mises à jour 6
Le travail réalisé 7
Plan du document 8

Chapitre 1 Documents dans un système workflow : modélisation et état de l'art **10**
1.1 Document dans un workflow : le modèle des statescharts . . . 11
1.2 Documents structurés et leurs sérialisations 13
1.2.1 Arbres de dérivation vs arbres de syntaxe abstraite . . 15
1.2.2 Sérialisation d'un document structuré 17
1.3 Vues, projections et projections inverses 19
1.3.1 Vue et projection associée 20
1.3.2 Lien avec la sérialisation 21
1.3.3 Grammaire avec vues 21
1.3.4 Problème de la projection inverse 23
1.3.5 Problèmes de cohérence et de fusion 25
1.4 DTD et grammaire XML 27
1.4.1 Grammaire XML dérivée d'une DTD 28
1.4.2 DTD dérivée d'une grammaire XML 29
1.5 État de l'art : vues XML et fusion des documents structurés . 30
1.5.1 Notion de vue XML 30
1.5.2 Fusion de vues XML 33
1.5.3 Technique de fusion two-way versus three-way. 34
1.6 Synthèse 37

Chapitre 2 Arbres et arènes **38**
2.1 Structures de données paresseuses et leurs manipulations . . . 39
2.1.1 Grammaires, signature et structures de données 39
2.1.2 Algèbres et évaluations 40
2.1.3 Co-algèbres et générations 42
2.1.4 Généralisation : les foncteurs 44
2.1.5 Grammaire, co-algèbre et automates d'arbres 48
2.2 La structure d'arène 51
2.2.1 Extension d'une arène 54
2.2.2 Elagage et nettoyage d'une arène 56
2.3 Synthèse 58

Chapitre 3 Fusion de vues partielles d'un document structuré 59
3.1 Projection inverse : algorithme d'expansion 60
3.1.1 Un exemple 64
3.2 Cohérence d'un ensemble de vues partielles 66
3.2.1 Synchronisation de deux vues 67
3.2.2 Algorithme de cohérence 68
3.3 Fusion de vues partielles d'un document structuré 70
3.4 Synthèse 84

Chapitre 4 TinyCE : un prototype d'éditeur coopératif désynchronisé **87**
4.1 Architecture de TinyCE 88
4.1.1 Interconnexion des modules de TinyCE 88
4.2 Exemple d'utilisation de TinyCE 92
4.3 Synthèse 95

Conclusion **99**
Problématique abordée et choix méthodologiques 99
Analyse critique des résultats obtenus 101
Perspectives 105

Bibliographie **108**

Annexe A Quelques notations en Haskell **113**
A.1 Quelques combinateurs Haskell 113

Annexe B Quelques fonctions sur les données structurées **116**

Table des figures

1.1 A statechart 12
1.2 Exemple d'édition collaborative asynchrone à l'aide des statescharts 14
1.3 Représentation intentionnelle (arbre de syntaxe abstraite) et arbre de dérivation associé 17
1.4 linéarisation d'une forêt et analyse d'un mot de Dyck 18
1.5 Diverses représentations d'un document structuré : arbre de syntaxe abstraite, arbre de dérivation, représentation à la XML 19
1.6 Un arbre de dérivation et ses projections suivant {A,B} et {A,C} 21
1.7 Relations entre projections et sérialisation 22
1.8 Relations entre grammaires et diverses représentations d'un document structuré 24
1.9 Les expansions possibles d'une vue partielle 25
1.10 Fusion et répercution des mises à jour locales 26
1.11 Cohérence de deux vues 26
1.12 Cohérence au moyen des anamorphismes 27
1.13 Fusion suivant la technique "two-way" de documents [Fon02] . 35
1.14 Fusion suivant la technique "three-way" de documents [Fon02] 35
1.15 Dilemme insert/delete 36

2.1 Diagramme commutatif pour les algèbres 46
2.2 Diagramme commutatif pour les co-algèbres 46
2.3 Structure de données paresseuse pour les ensembles d'arbres de dérivation 53
2.4 Arbres membres de l'ensemble représenté par l'arène de la figure 2.3 54
2.5 Génération des arènes et énumération de leurs éléments 55
2.6 en traits pleins : élagage de l'arène de la figure 2.3 57

3.1 Dérivation d'un automate d'arbre associé à une vue 62
3.2 Élagage (et son nettoyage) de l'arène engendré par l'automate d'arbre associé à la vue partielle (()[()()]) 66

3.3 Une solution (arbre associé à la vue partielle (()[()()])) obtenue par "auto-greffe" à partir de l'arbre inscrit dans l'élagage de la figure 3.2 66
3.4 Exemple d'arbre de syntaxe abstraite ouvert (a), d'arbre de dérivation correspondante (b) et une projection de ces arbres suivant *{A,B}* (c). 75
3.5 *two-way fusion* des mises à jour effectuées sur les projections sur *{A}* et *{A,B}* respectivement d'un document 85
3.6 *three- way fusion* d'un document et des mises à jour effectuées sur ses projections sur *{A,B}* et *{A,C}* respectivement 86

4.1 Architecture de *TinyCE* 94
4.2 Exemple d'édition coopérative asynchrone à l'aide des statescharts : apllication à la grammaire G_{run} 95
4.3 Interface serveur de *TinyCE* présentant un début d'édition ainsi que les vues partielles à envoyer aux différents clients. . . 95
4.4 Interface du premier client de *TinyCE* présentant la vue partielle reçu du serveur. 96
4.5 Interface du second client de *TinyCE* présentant la vue partielle reçu du serveur. 96
4.6 Interface du premier client de *TinyCE* présentant la mise à jour locale de la vue partielle à envoyer au serveur. 97
4.7 Interface du second client de *TinyCE* présentant la mise à jour locale de la vue partielle à envoyer au serveur. 97
4.8 Interface serveur de *TinyCE* présentant le document de base, les mises à jour de ses réplicats partiels (provenant des différents clients) ainsi que leur fusion. 98

Résumé

Un document structuré complexe est représenté intentionnellement sous la forme d'une structure arborescente décorée par des attributs. Si on ne s'intéresse qu'aux aspects purement structurels, les documents licites peuvent être caractérisés par une grammaire algébrique abstraite. Dans cette thèse, après avoir montré comment l'édition coopérative de tels documents peut être étudiée au moyen d'un modèle inspiré des modèles workflow, nous posons et donnons une solution au problème de la fusion en un document global (cohérent) de diverses vues partielles d'un document éditées de façon asynchrone. A cette fin, nous représentons l'ensemble (potentiellement infini) de documents compatibles avec une vue partielle donnée par une structure de données co-inductive appelée *arène*. Cette structure encapsule un ensemble régulier d'arbres et peut être considérée comme l'image d'une vue partielle du document par le morphisme canonique (anamorphisme) associé à une coalgèbre (un automate d'arbres). Ainsi présenté, fusionner les diverses vues partielles revient à construire l'intersection des ensembles réguliers d'arbres correspondants à chacune des vues ; cette intersection peut être obtenue en utilisant une opération de synchronisation définie sur les automates d'arbres. Nous présentons un outil sommaire permettant de faire la démonstration de l'algorithmique issue de ce travail.

Mots clés : DTD, Grammaire algébrique, Représentation intentionnelle, Cohérence de vues, Évaluation paresseuse, Automates d'arbres, Anamorphisme, Réplicat partiel.

Abstract

A complex structured document is intentionnally represented as a tree decorated with attributes. If we focus our attention to purely structural aspects, the set of legal documents can be fully caracterized by an abstract context-free grammar. In this thesis we address the problem of the cooperative edition of structured documents in a distributed workflow system. We present and give a solution to the problem of how to merge a set of partial views of a document (edited asynchronously) into one global coherent document. For that purpose, we represent the potentially-infinite set of documents compatible with a given partial view as a coinductive data structure. This set is a regular set of trees that can be obtained as the image of the partial view of the document by the canonical morphism (anamorphism) associated with a coalgebra (some kind of tree automaton). Merging partial views then amounts to computing the intersection of the corresponding regular sets of trees which can be obtained using a synchronization operation on tree automata. We present a tool for demonstrating the various algorithms resulting from our study.

Keywords : DTD, Contex-free grammar, Intentional representation, Coherence of views, Lazy evaluation, Tree automata, Anamorphism, Partial replication.

Introduction

Sommaire

Des documents et des hommes **1**
Fusion des réplicats partiels **3**
Réconciliation des mises à jour **6**
Le travail réalisé **7**
Plan du document **8**

Des documents et des hommes

L'édition et la publication de documents jouent un rôle croissant dans la mise en oeuvre de systèmes informatisés. Selon les cas, on peut distinguer [Bon98] les documents linéaires, modélisés par un flot de données qui mêle le contenu du document aux informations de présentation ; les documents semi-structurés ou auto-descriptifs qui sont libres de tout modèle de description de données mais dont la structure des éléments présente une certaine régularité et les documents structurés dont la forme est contrainte par un modèle générique appelé *modèle de document*. Dans cette thèse nous ne nous intéresserons qu'à ces derniers.

Un document structuré peut être interprété comme un arbre décoré par des attributs. Ces attributs peuvent décrire l'apparence du document dans un contexte particulier d'utilisation, ainsi que des propriétés du document qui peuvent être exploitées par l'application qui le manipule. L'ensemble des structures licites d'un document peut être spécifié par une grammaire algébrique avec un ensemble de contraintes sur les valeurs possibles des attributs (ce qui peut être donné, par exemple, par les règles sémantiques d'une grammaire attribuée [Knu68, Paa95]). Néanmoins, nous ne tiendrons pas compte ici des attributs ; ces derniers sont liés à des aspects sémantiques qui peuvent être traités de façon indépendante des aspects purement structurels qui vont nous intéresser. En particulier nous ne nous intéresserons ni à la forme syn-

taxique de ces documents ni à la façon dont ils peuvent être visuellement perçus par un utilisateur particulier ; puisque ces différents aspects peuvent être associés à des attributs du document. Il sera toujours possible de retrouver ces informations en évaluant l'attribut correspondant. On peut aussi relier le document à une de ses représentations concrètes en lui appliquant des transformations bidirectionnelles d'arbres [MHT, MHT04, MHT06].

Depuis l'essor des technologies XML [W3C00] et des services web les documents structurés sont devenus des outils incontournables pour la publication et l'échange d'informations entre applications le plus souvent hétérogènes et distantes. La puissance toujours croissante des réseaux de communication en terme de débit et de sûreté a aussi révolutionné la façon d'éditer de tels documents : au modèle classique d'un auteur éditant en local et de façon autonome son document, s'est substituée l'édition coopérative dans laquelle plusieurs auteurs situés sur des sites géographiquement éloignés se coordonnent pour éditer de façon asynchrone un même document. Dans un contexte de travail coopératif il est par ailleurs raisonnable de voir les documents échangés autant comme des *supports de communication et une aide à la coordination* que comme le *produit de cette coopération*. Ces documents véhiculent les informations nécessaires au travail coopératif. On imagine une équipe dont chacun des acteurs, avec son expertise propre, opère sur le document par le biais d'une interface en utilisant des outils spécifiques liés à son rôle particulier et à son domaine d'expertise. Les actions d'édition qu'il est susceptible d'effectuer pourront être capturées dans un langage dédié qui encapsule le savoir-faire métier correspondant. Ces outils spécifiques et ce langage dédié auront en général été conçus préalablement et indépendamment de l'application de travail coopératif qui les utilise. Il n'y a donc aucune raison pour que ces outils aient conscience de la structure globale des documents qu'ils manipulent. Ce sera donc au contraire la grammaire globale qui sera conçue en fusionnant les structures manipulées par les différents acteurs qui interviennent dans le système. Les productions de cette grammaire imposent par conséquent des contraintes de cohérence entre les versions manipulées localement par chaque intervenant et peuvent par la même occasion véhiculer de l'information entre ces acteurs au moment de la resynchronisation des mises à jour locales. Ces flots d'information et ces contraintes peuvent aussi être de nature non-contextuelle s'ils sont portés par des attributs de la grammaire (aspect que nous ne développerons pas ici). C'est en ce sens que le document sert de support à la coordination des différents acteurs. Ainsi chacun des acteurs opère sur une vue partielle distincte du document global. Une telle vue partielle est obtenue en ne gardant dans le document, représenté par un arbre de syntaxe abstraite, que les noeuds correspondant à des catégories syntaxiques perceptibles par l'utilisateur.

Fusion des réplicats partiels

Nous admettons qu'à un moment donné plusieurs vues partielles du document, que nous appelerons réplicats partiels, peuvent coexister dans le système et qu'ils peuvent être modifiés de façon asynchrone par différents utilisateurs : comme dans [SS05, IOM+07, Sha02], nous optons donc pour une approche optimiste de la réplication (partielle) des documents. Nous devons lors de la synchronisation de ces réplicats être capable de décider si ces différentes versions sont cohérentes et dans l'affirmative de fusionner les mises à jour locales, effectuées sur les réplicats partiels, afin d'obtenir une version actualisée cohérente du document global.

La fusion de documents est une problématique assez ancienne ; elle a déjà fait l'objet de nombreuses études dans des contextes assez variés. On trouve des études sur la fusion de documents structurés XML [Lin04, CSGM, CSR+, Fon02, Ask94, Man01], ainsi que des études sur la re-synchronisation de fichiers manipulés par des utilisateurs mobiles lorsque ces derniers se connectent au réseau [BP98] (auquel ils n'ont accès que par intermittence). La fusion de programmes [HSR89, Men02] dans le contexte de développement coopératif de logiciels rentre aussi dans cette catégorie. Enfin, les notions de cohérence de vues partielles et de re-synchronisation sont également familières dans la communauté des bases de données. Dans toutes ces études, la structure grammaticale des différents réplicats est la même et il s'agit de fusionner différentes modifications locales d'un document ; ce qui peut nécessiter, lorsque des conflits apparaissent, de "réconcilier" les différents points de vues en remettant en cause certaines actions d'édition. La réconciliation de vues partielles met en jeu diverses heuristiques car ce genre de problème n'admet pas de solution générale et systématique. Notre point de vue est différent pour plusieurs raisons. D'une part, nous insistons sur l'hétérogénéité des réplicats partiels qui sont dans notre cas associés à des structures grammaticales distinctes ; la partie centrale de notre solution algorithmique sera de fait constituée d'un algorithme, dit d'expansion, qui va construire une représentation de l'ensemble (en général infini) des documents ayant une vue partielle donnée. D'autre part, les conflits qui apparaissent lors de la réconciliation de vues partielles proviennent essentiellement des opérations d'effacement ou de restructuration (déplacement) de l'information dans le document. Ces deux difficultés seront évacuées dans notre cas du fait qu'on ne s'intéresse qu'à une édition incrémentale (non destructive) et qu'on préserve une séparation nette entre la structure logique du document et sa présentation à l'utilisateur.

Dans un système à flot de tâches, une nouvelle activité est initiée par l'introduction d'un document dans le système, par exemple un formulaire.

Pendant sa durée de vie dans le système, ce document ne fera que croître en incorporant de façon incrémentale de nouvelles informations jusqu'à ce que le traitement de l'activité soit arrivé à son terme et que le document, support de cette activité, sorte du système (pour être éventuellement archivé). Normalement, un tel document a une durée de vie limitée dans le système et, quand on le met à jour, on a pas besoin d'y effacer de l'information (en général, comme dans tout bon système administratif, on considérera plutôt que ce serait une mauvaise pratique que de détruire de l'information). C'est cette situation que nous qualifions d'*édition incrémentale*. Cela ne signifie pas, bien entendu, que l'utilisateur en local ne puisse revenir en arrière sur une de ses actions d'édition. Ce qui importe pour le système c'est que la suite des actions d'édition effectuées en local entre le moment où l'utilisateur reçoit le document et celui où il le restitue au système se traduise globalement par un accroissement du document. C'est-à-dire que l'utilisateur peut remettre en cause une de ses actions d'édition tant qu'il n'a pas retransmis le document au système et s'il "efface" des parties du document reçu ou en déplace certaines parties cela n'aura en réalité aucun effet sur le document lui-même mais seulement sur la perception qu'il en aura. Evidemment on devra maintenir une cohérence entre le document manipulé et sa représentation à l'utilisateur ; mais cela est réalisé de façon purement locale, sans qu'il y ait besoin d'aucune forme de synchronisation, et donc cela ne pose aucune difficulté.

C'est cette parfaite séparation entre la structure logique du document et sa présentation à l'utilisateur qui fait que toute action d'édition, tant qu'elle reste locale, peut être rendue parfaitement réversible et n'être effectuée que de façon purement virtuelle. Ce sont ces deux aspects, incrémentalité et virtualisation des actions d'édition locales, qui nous conduisent à une simplification drastique de la notion de vue. On trouve dans la communauté XML une notion de *vue XML (XML view)* qui désigne, suivant les auteurs, soit un document XML contenant des données provenant d'une base de données [Abi99] et sujette aux opérations d'édition permettant la mise à jour de la base de données correspondante [Bra02], soit un fragment de document XML obtenu à partir d'un document XML (de base) par combinaison de certaines opérations de base [CLL02] : 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 dans ce contexte peut se faire aisément en utilisant l'un des langages proposés par le consortium World Wide Web pour la transformation des documents XML comme *XSL* [XSL] ou pour l'interrogation des documents XML comme *XQuery* [XQu] (voir les exemples de la section 1.5.1). Avec nos hypothèses,

une vue se réduit à projeter le document en ne conservant que les noeuds perceptibles par l'utilisateur concerné, tout en conservant la structure hiérarchique du document.

Puisque la structure réelle du document n'est pas directement perceptible par les utilisateurs, on peut se demander pourquoi on ne considère pas une structure grammaticale unique pour le document sur laquelle les différents intervenants opèrent à travers une vue partielle. Il y a plusieurs raisons à cela.

L'approche de conception est ascendante Comme nous l'avons déjà signalé, un utilisateur travaille sur un réplicat partiel du document en utilisant des outils et des interfaces dont la conception est antérieure à celle du système de travail coopératif auquel il participe. Leurs concepteurs ne pouvaient pas prévoir tous les contextes dans lesquels ces outils pourraient être amenés à être utilisés. En local, on peut disposer d'une représentation abstraite de la structure logique du document et d'une représentation concrète plus proche de la perception de l'utilisateur. Ces deux représentations sont néanmoins fortement connectées, leur couplage peut-être implémenté par des transformations bi-directionnels d'arbres [MHT, MHT04, MHT06]. Dans le cas le plus simple, elles pourront être en correspondance bi-univoque. Mais il est plus probable que plusieurs structures abstraites donneront lieu à une même structure concrète. En effet, il est en général nécessaire de préserver de l'information dans la représentation abstraite si on souhaite pouvoir rendre les actions d'édition locale réversibles. Ce qui importe c'est que toute suite d'actions d'édition sur la vue concrète se relève de manière unique en une transformation sur la représentation abstraite. Cette propriété de relèvement et la réversibilité sont les deux seules hypothèses que nous ferons sur ces outils. Ces hypothèses nous permettent de faire abstraction de la représentation concrète et de ne considérer que les formes abstraites dont la structure grammaticale constitue l'interface de l'outil d'édition locale. Lorsqu'on construit de façon ascendante l'application de travail coopératif en assemblant ces différents outils, on est amené à concevoir (quand c'est possible et notamment quand il n'y a pas d'ambigüité entre les grammaires) une grammaire globale qui intègre les interfaces de ces outils. Un document conforme à cette grammaire se projette alors en des documents conformes aux différentes interfaces en effaçant les symboles grammaticaux non concernés par cette vue ; ce qui revient à dire qu'on extrait pour chaque outil d'édition locale les parties du document qui le concerne. Les productions de la grammaire globale servent alors à la coordination des différents outils d'édition

locale.

Les systèmes workflow sont hiérarchiques Ce que nous venons d'indiquer pour un niveau de décomposition doit pouvoir se répéter de manière récursive afin d'obtenir un modèle hiérarchique du système global.

Les mises à jour locales se font de façon asynchrone Si toute action d'édition locale doit pouvoir se relever de façon unique en un document abstrait conforme à l'interface de cet outil, ce n'est plus le cas au niveau global. Il y a en général potentiellement une infinité de documents globaux pour une vue partielle donnée. La reconstruction du document global actualisé nécessite de resynchroniser les différents réplicats partiels qui ont été mis à jour de façon asynchrone.

Réconciliation des mises à jour

À partir de plusieurs vues partielles, on souhaite obtenir par fusion un document global qui intègre toutes les mises à jour effectuées sur chacune de ces vues. Deux problèmes peuvent se présenter à ce niveau :

Conflit Un tel document ne peut pas être obtenu parce que des modifications locales non compatibles ont été effectuées.

Ambiguïté Plusieurs solutions sont au contraire possibles.

L'ambiguïté est considérée comme un défaut dans la conception du système. La structure grammaticale obtenue par fusion des interfaces des outils locaux d'édition qui constituent l'application à un niveau donné doit être "couverte" par l'ensemble de ces interfaces ; ce qui signifie que si le système de travail coopératif est bien conçu, le document à un niveau donné doit pouvoir être caractérisé par l'ensemble des différentes mises à jour de ses réplicats partiels lorsque ces derniers ne présentent pas de conflit. Cette situation est analogue à ce qui se passe en analyse syntaxique où une grammaire ambiguë est généralement perçue comme un défaut de conception : un document s'il est conforme à la grammaire doit pouvoir être interprété de manière non ambiguë par un arbre de syntaxe abstraite. Néanmoins si l'analyseur est écrit au moyen de combinateurs d'analyse, ces derniers sont en général non déterministes. Mais ce non-déterminisme est local et doit être levé au niveau global. Nous avons ici une situation similaire. Nous introduisons un opérateur binaire (commutatif et associatif) de fusion tel que, la fusion d'un ensemble fini de vues partielles s'obtient par itération de cet opérateur. Même si nous supposons qu'il n'y a pas d'ambiguïté au niveau global, cet opérateur binaire de fusion n'a aucune raison de produire un document unique. En général, il produira un ensemble infini de documents.

Le conflit, quant à lui, peut être abordé de deux façons différentes :

La réconciliation Réconcilier des vues partielles non compatibles consiste à remettre en cause certaines actions d'édition locale afin de lever les conflits et aboutir à une version globale cohérente. Les études portant sur la réconciliation de versions d'un document reposent sur des heuristiques dans la mesure où il n'y a pas de solution générale à ce problème. Dans notre cas, dans la mesure où toutes les actions d'édition sont réversibles et incrémentales (monotones) et qu'il est facile de localiser les conflits lors de la tentative de fusion des vues partielles, on dispose d'une méthode canonique pour éliminer les conflits. Il suffit en effet de prendre le préfixe maximal du document ne contenant pas de conflit (on coupe au niveau des noeuds où un conflit apparaît, en remplaçant le sous arbre correspondant par un noeud ouvert indiquant que cette partie du document n'est pas encore éditée), appelons ce document le consensus. La suite des actions d'édition qui ont été opérées sur un des réplicats partiels peut se représenter comme un ordre partiel ; il suffit alors, en partant des éléments maximaux de cet ordre, d'inverser les actions d'édition qui contribuent à rendre le document plus petit que la projection correspondante du consensus. On fusionne enfin les nouvelles versions locales ainsi obtenues.

Le contrôle Une technique inspirée du contrôle des systèmes à événements discrets consiste à synthétiser un contrôleur pour chacun des sites de telle sorte que les réplicats partiels restent compatibles tant que les modifications locales sont exécutées sous la supervision du contrôleur. Il est légitime de supposer que toute action d'édition est à la fois observable et contrôlable. Le contrôle doit être distribué pour être intéressant ; néanmoins il est envisageable de synthétiser dans un premier temps un contrôleur centralisé, puis de rechercher le minimum de points de synchronisation à rajouter entre les sites pour le rendre distribuable.

On peut aussi mélanger les deux approches dans le cas où par contrôle on restreint trop les actions d'édition ou si cela rajoute trop de points de synchronisation. Dans ce cas on ne fait qu'un contrôle partiel et on applique la technique de réconciliation dans le cas où un conflit apparaît.

Le travail réalisé

Nous avons modélisé les documents en cours d'édition sous la forme d'arbres contenant des *noeuds ouverts*, appelés aussi *bourgeons*. Ce sont des feuilles à partir desquelles la structure peut se développer. Nous déduisons

une relation d'ordre naturelle notée $\leq$ sur les documents telle que $t_1 \leq t_2$ si t_2 peut être obtenu de t_1 en remplaçant des bourgeons de t_1 par des arbres ; nous disons alors que t_2 est une *mise à jour* de t_1. Un document est complet, ou clos, s'il ne contient aucun bourgeon, c'est donc un élément maximal pour cet ordre.

Dans un premier temps nous nous sommes restreint aux documents clos pour aborder le *problème de cohérence de vues* qui consiste à décider s'il existe un document global correspondant à un ensemble de vues partielles et dans l'affirmative à produire un tel document. Nous donnons une solution à ce problème à la section 3.2, basée sur un algorithme dit *d'expansion* qui produit une représentation de l'ensemble (régulier) de documents associés à une vue donnée. Cette représentation est une *arène*, une structure de données co-inductive que nous présentons au chapitre 2. Une arène permet de représenter les ensembles potentiellement-infinis d'arbres. Nous adaptons alors dans la section 3.3 cet algorithme d'expansion pour le cas des arbres ayant des noeuds ouverts pour qu'il produise une représentation de toutes les mises à jour possibles de documents associés à une vue partielle donnée. La fusion des mises à jour effectuées localement sur les différentes vues partielles peut alors être obtenue de façon similaire à ce qu'on a réalisé dans le cas du problème de cohérence de vues.

Plan du document

La suite de ce document comporte cinq chapitres et deux annexes organisés comme suit :

Chapitre 1 : Documents dans un système workflow. Après une présentation succincte des outils formels issus de la littérature (systèmes workflow, statecharts, grammaires algébriques, automates d'arbre) que nous utilisons pour la spécification et la manipulation des documents structurés, nous introduisons de façon plus formelle les définitions relatives aux notions de vues, de projection et de projection inverse. Ce chapitre se termine par un état de l'art sur la fusion de documents XML.

Chapitre 2 : Arbres et arènes. Après une présentation sommaire de la notion de structure de données paresseuses en Haskell ainsi que leur manipulation à l'aide des algèbres (interprétation) et des co-algèbres (génération), nous présentons la structure d'arène (*arena*) [BT08, BT07] : une structure de donnée paresseuse permettant de représenter un ensemble potentiellement infini d'arbres.

Chapitre 3 : Fusion des vues partielles d'un document structuré. Nous donnons successivement une solution au problème de cohérence de vues et à celui de la fusion des mises à jour des réplicats partiels d'un document.

Chapitre 4 : TinyCE : un prototype d'éditeur coopératif désynchronisé. Dans ce chapitre, nous présentons *TinyCE* (a **Tiny C**ollaborative **E**ditor, prononcer "*Tennesse*"), un prototype expérimental d'éditeur coopératif désynchronisé WYSIWYG [1] qui nous permet de mettre en oeuvre les algorithmes présentés dans ce travail et d'en faire la démonstration.

Conclusion générale. Donne un bilan de notre travail et présente quelques pistes d'extension de ce travail.

Annexe A : Quelques notations en Haskell Tous les algorithmes présentés dans ce manuscrit sont écrits dans le langage fonctionnel *Haskell*. Conscient que nos potentiels lecteurs ne seront pas nécessairement familiers de ce type de programmation, nous donnons dans cette courte annexe quelques notations qui suffisent à la compréhension du code donné dans le manuscrit.

Annexe B : 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 certaines hypothèses 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. Nous avons souhaité fournir ce code par souci de complétude afin qu'un lecteur souhaitant expérimenter nos solutions puisse le faire sans avoir à coder de fonctions supplémentaires.

1. What You See Is What You Get

Chapitre 1

Manipulation de documents dans un système à flots de tâches (système workflow)

Sommaire

1.1 Document dans un workflow : le modèle des statescharts **11**
1.2 Documents structurés et leurs sérialisations **13**
1.3 Vues, projections et projections inverses **19**
1.4 DTD et grammaire XML **27**
1.5 État de l'art : vues XML et fusion des documents structurés **30**
1.6 Synthèse **37**

Notre travail s'inscrit dans la conception de systèmes à flots de tâches (worflow systems) dont les acteurs géographiquement distants coordonnent leurs activités par échange de documents structurés. Un tel système doit présenter une structure hiérarchique dans laquelle on doit pouvoir à la fois décrire la circulation du document d'une activité à une autre (décomposition de type "ou" : le document est dans telle ou telle phase de traitement) que la distribution de divers réplicats partiels du document à des intervenants qui seront amenés à opérer sur ces copies de façon asynchrone (décomposition de type "et" : le document est simultanément présent en plusieurs endroits du système). Un modèle particulièrement adapté à ce type de situation est celui des *statescharts* que nous présentons dans la section 1.1.

Par la suite (sec. 1.2) nous indiquons la façon dont nous représentons les documents par des arbres de syntaxe abstraite conformes à une grammaire algébrique (abstraite). Nous introduisons ensuite (sec. 1.3) nos définitions relatives aux notions de vues, de projection d'un document suivant une vue

et enfin de fusion de vues partielles d'un document. Nous mettons aussi en exergue sous forme d'une revue de la littérature (sec. 1.4) les relations qui existent entre les grammaires XML et les DTD d'une part, les automates d'arbres et les grammaires algébriques (abstraites) d'autre part.

Nous achevons ce chapitre (sec. 1.5) par un état de l'art sur la fusion des vues d'un document XML.

1.1 Document dans un workflow : le modèle des statescharts

La modélisation d'un processus d'édition collaborative repose sur un graphe dont les noeuds représentent les sites, distribués géographiquement, où s'effectuent les actions d'édition sur les réplicats partiels du document. Les arcs entre ces sites permettent de représenter les façons dont le document peut circuler à travers le système. De tels systèmes sont appelés des systèmes à flot de tâches (*workflow systems*) que *Dirk Wodtke et al.* dans [WW97] définissent de façon informelle comme un ensemble d'activités avec des contrôles explicites et un flot de données entre ces activités. En accord avec [WW97], nous représentons de tels systèmes workflow par des *statecharts* [Har87]. Nous aurions pu également choisir des diagrammes d'états d'UML (une standardisation des statecharts) ou une classe spécifique de réseaux de Petri à cette fin. Cependant les statecharts semblent directement adaptés au problème qui nous intéresse dans la mesure où ils offrent un formalisme visuel pour la description modulaire de diagrammes états/transitions en autorisant le regroupement d'états, l'orthogonalité entre états - pour permettre des traitements concurrents - et le raffinement d'états. Ce sont donc des extensions des diagrammes états/transitions classiques dans lesquels on a ajouté des décompositions *et/ou* sur les états, des transitions entre *macro-états*[1] ainsi qu'un mécanisme de diffusion entre composants concurrents du statecharts. D. Harel dans [Har87] résume la définition des statescharts par l'équation :
statecharts = state-diagrams + depth + orthogonality + broadcast-communication.
Nous allons retenir les grandes lignes de ce modèle à l'exception du méchanisme de diffusion entre composants concurrents que nous n'utiliserons pas.

Un état d'un statechart peut être décomposé hiérarchiquement de deux

1. Nous entendons par *macro-états* un état obtenu par regroupements d'états - dits états constitutifs - à l'aide de l'opérateur "*et*" ou à l'aide de l'opérateur "*ou*". Les états constitutifs regroupés à l'aide de l'opérateur *ou* sont concurrents ie. qu'on ne peut se retrouver dans deux tels états en même temps tandis que les états constitutifs regroupés à l'aide de l'opérateur *et* sont dits parallèles. Dans la suite, tant que la compréhension sera implicite, nous parlerons d'état pour désigner indifféremment un macro-état ou un état.

façons : la décomposition “ou” (représentée dans la figure 1.1 par le fait qu'une boîte se décompose en un diagramme de transition entre sous-boîtes représentant ses constituents “ou”) et la décomposition “et” (représentée dans la figure 1.1 par la subdivision d'une boîte en sous-boîtes séparées par des traits interrompus). Lorsque nous utilisons les statescharts comme modèle conceptuel des systèmes pour l'édition collaborative, nous considérons qu'un document placé dans un état “ou” quelconque, se trouve dans exactement un de ses (sous-)états constitutifs ; s'il est placé dans un état “et” une vue partielle (réplicat partiel) de ce document doit être présent dans chaque partie (sous-état) constitutive de cet état.

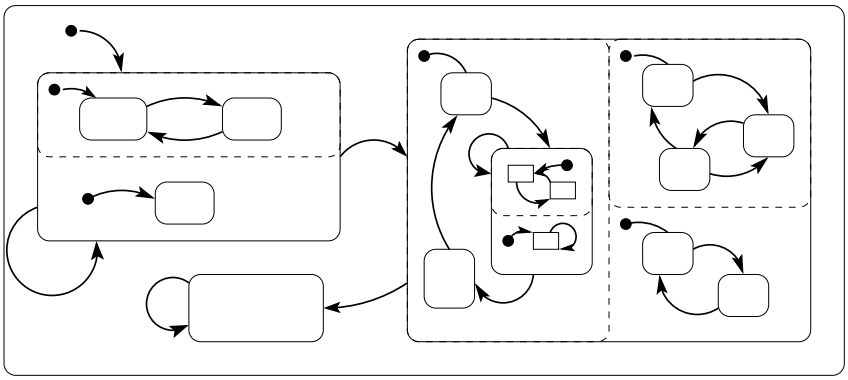

FIGURE 1.1 – A statechart

Une transition entre deux états peut être étiquetée par un événement ou par une garde exprimant une condition qui, quand elle est satisfaite provoque le franchissement de la transition. Dans chaque état aussi bien que dans chacune des parties constitutives d'un état “et”, un état initial est spécifié que nous activons initialement en entrant dans l'état père. En entrant dans un état “et”, le document est projeté sur chacun des composants (états constitutifs) de cet état et chaque réplicat partiel est dirigé vers l'état initial du composant correspondant. Le résultat de la projection doit être conforme au modèle de document associé à cet état. Après projection, chaque vue partielle du document transite dans son sous-système où il peut être manipulé. En quittant l'état “et”, les vues partielles provenant des diverses composantes de l'état sont fusionnées et le document résultant est acheminé vers l'état suivant du système. Nous ferons l'hypothèse qu'il n'y a aucune synchronisation entre différents composants “et” ce qui signifie que les modifications sont opérées sur les différents réplicats partiels de façon totalement asynchrone. Si une telle synchronisation doit intervenir, elle sera représentée explicitement par une transition : une boucle sur un état de type “et”, dont le franchissement est conditionnée rappelons-le par un événement ou une garde, revient à fu-

sionner les actions d'édition opérées sur les différents réplicats et à projeter le document ainsi actualisé sur les différents constituants. Afin de compléter le modèle nous devons attacher à chaque état une structure grammaticale décrivant la classe des documents qui peuvent se trouver en cet état. Un état de type "ou" et chacun de ces constituants immédiats doivent être associés à la même structure grammaticale. Et nous devons disposer pour chaque état de type "et" de fonctions permettant d'une part de projeter un document conforme à sa structure grammaticale sur des documents conformes aux structures grammaticales de ses constituants et inversement d'une fonction de fusion prenant en entrée des documents conformes aux structures grammaticales des constituants pour former en retour un document conforme à la structure grammaticale de cet état global.

Illustration. La figure 1.2 donne un exemple d'édition collaborative modélisée à l'aide d'un statecharts. Ce statecharts comporte les états A, A_1, A_2, B, B_1, B_{11}, B_{12}, B_2, B_{21}, B_{22}, et C tels que : l'état A a pour états constitutifs A_1 et A_2 (décomposition *ou*), l'état B a pour états constitutifs B_1 et B_2 (décomposition *et*) .... Dans l'état A_1 (état initial) on a un document vide. Dans l'état A_2 le document est édité et est représenté par l'arbre de dérivation `Node A [Node A [ ], Node B[ ], Node C[ ]]`. Lors du franchissement de la transition (2), le document est projeté suivant les sous-ensembles de catégories syntaxiques $\mathcal{V}_1 = \{A, B\}$ et $\mathcal{V}_2 = \{A, C\}$ et on obtient les réplicats partiels `Node A [Node A [ ], Node B[ ]]` dans l'état B_{11} et `Node A [Node A [ ], Node C[ ]]` dans l'état B_{21}. Ces réplicats partiels sont édités en parallèle (transitions (5) et (6)) et on obtient les documents indiqués sur les états respectifs B_{12} et B_{22}. Quand on quitte l'état B par le franchissement de la transition (7), les réplicats partiels localisés sur les états B_{12} et B_{22} sont fusionnés et on obtient le document de l'état C (état final).

1.2 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 attachés. Les documents licites sont ceux dont la structure est conforme à une grammaire algébrique abstraite.

Définition 1.2.1. *Une* ***grammaire algébrique abstraite*** *est la donnée* $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ *d'un ensemble fini* $\mathcal{S}$ *de* ***symboles grammaticaux*** *qui correspondent aux différentes* ***catégories syntaxiques*** *en jeu, d'un symbole grammatical* $A \in \mathcal{S}$ *particulier, appelé* ***axiome****, et d'un ensemble fini* $\mathcal{P} \subseteq$

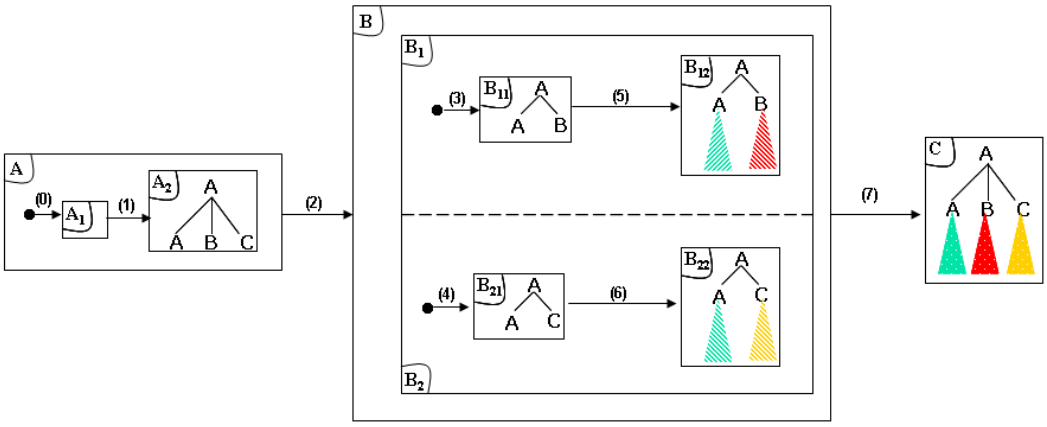

FIGURE 1.2 – Exemple d'édition collaborative asynchrone à l'aide des statescharts

$\mathcal{S} \times \mathcal{S}^*$ *de* ***productions***. *Une production* $P = \left(X_{P(0)}, X_{P(1)} \cdots X_{P(n)}\right)$ *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 :

```
1 data Gram prod symb = Gram{prods :: [prod],
2                            symbols :: [symb],
3                            lhs :: prod -> symb,
4                            rhs :: prod -> [symb]}
```

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 $G_{run} = (\{A, B, C\}, P, A)$ [2] 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 :

```
1  data Prod = P1 | P2 | P3 | P4 | P5 | P6 | P7  deriving (Eq,Show)
2  data Symb = A | B | C  deriving (Eq,Show)
3
4  gram :: Gram Prod Symb
5  gram = Gram lprod lsymb lhs_ rhs_
6     where  lprod = [P1, P2, P3, P4, P5, P6, P7]
7            lsymb = [A, B, C]
8            lhs_ p = case p of
9                   P1 -> A; P3 -> B; P5 -> C; P7 -> C
10                  P2 -> A; P4 -> B; P6 -> C
11           rhs_ p = case p of
12                  P1 -> [C, B]; P3 -> [C, A]; P5 -> [A, C]; P7 -> []
13                  P2 -> [];     P4 -> [B, B]; P6 -> [C, C]
```

1.2.1 Arbres de dérivation vs arbres de syntaxe abstraite

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 1.2.2. *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 $t(i \cdot u) = t_i(u)$.*

2. Cette grammaire sera largement utilisée tout au long de ce manuscrit pour des besoins d'illustrations.

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

```
data Tree a = Node {top :: a, succ_ :: [Tree a]} 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 1.2.3. *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* $t_i \in Der(\mathbb{G}, X_i)$, $X_i = X_{P(i)}$ *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 1.2.4. *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* $t_i \in AST(\mathbb{G}, X_i)$, $X_i = X_{P(i)}$ *pour tout* $1 \leq i \leq n$. *Les arbres de syntaxe abstraite sont donc les termes pour la signature dont*[3] *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. 1.3). 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 $m = n$) 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

3. Une définition formelle de *signature* est donnée page 39.

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.

Les fonctions Haskell respectives *isAST*, *ast2der* et *der2ast* dont les codes sont donnés dans l'annexe B permettent respectivement de tester la conformité d'un arbre de syntaxe abstraite vis-à-vis d'une grammaire, de donner l'arbre de dérivation équivalente à un arbre de syntaxe abstraite et réciproquement.

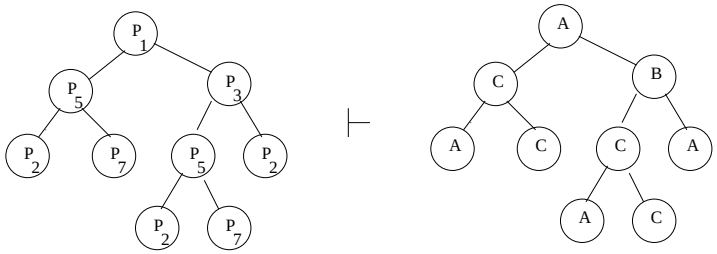

FIGURE 1.3 – Représentation intentionnelle (arbre de syntaxe abstraite) et arbre de dérivation associé

1.2.2 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

$$\langle Dyck \rangle \ \rightarrow\ (_i\ \langle Dyck \rangle\ )_i\ \langle Dyck \rangle \mid \epsilon \qquad 1 \leq i \leq n$$

ou de façon équivalente par

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

Un mot premier de Dyck code un arbre et un mot de Dyck une suite d'arbres, c'est-à-dire une forêt (figure 1.4). Nous représentons une lettre du langage de Dyck par le type :

```
data Dyck symb = Open symb | Close symb deriving (Eq,Show)
```

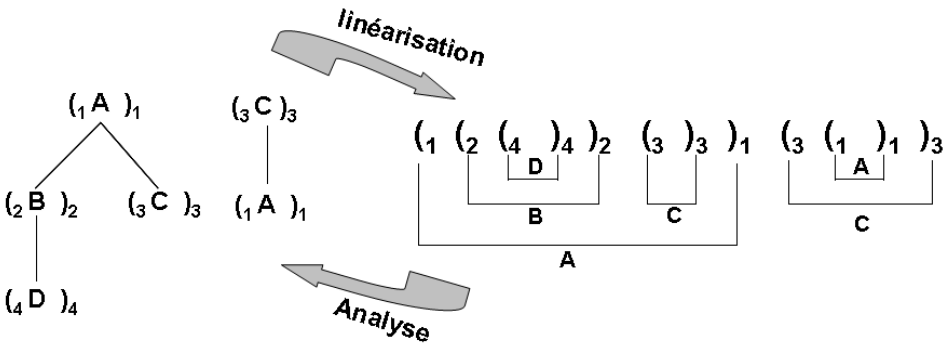

FIGURE 1.4 – linéarisation d'une forêt et analyse d'un mot de Dyck

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

```
1 linearisation :: Tree symb -> [Dyck symb]
2 linearisation (Node symb ts) =
3    [Open symb] ++ (concat (map linearisation ts)) ++ [Close symb]
```

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

À une grammaire algébrique abstraite $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ on associe une grammaire algébrique concrète $\mathbb{G}_\mathcal{S} = (\mathcal{S}, \mathcal{T}, \mathcal{P}_\mathcal{S}, A)$ obtenue en ajoutant comme symboles terminaux une parenthèse ouvrante et une parenthèse fermante associées à chaque catégorie syntaxique, c'est-à-dire $\mathcal{T} = Dyck\ \mathcal{S}$. Les productions de $\mathcal{P}_\mathcal{S}$ sont obtenues à partir de celles de $\mathcal{P}$ en insérant leur partie droite entre une paire de parenthèses : les productions de $\mathcal{P}_\mathcal{S}$ sont 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 $\mathcal{P}$.[4] 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 $\mathbb{G}_\mathcal{S}$ 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

4. La grammaire $\mathbb{G}_\mathcal{S}$ est une grammaire équilibrée au sens de [BB97]. 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 [McN67] et Knuth [Knu67]. Ces langages sont engendrés par des grammaires dites de parenthèses. L'essor de la technologie XML a permis de raviver ce domaine [BB00]. En particulier, Berstel et Boasson ont introduit les grammaires équilibrées (*Balanced Grammars*[BB97]) qui généralisent les grammaires de parenthèses à plusieurs types de parenthèses (et qui autorisent également des langages réguliers en partie droite des productions).

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) (voir fig. 1.5 et l'annexe B qui fournit des fonctions de traduction entre les différentes représentations d'un documents structurés).

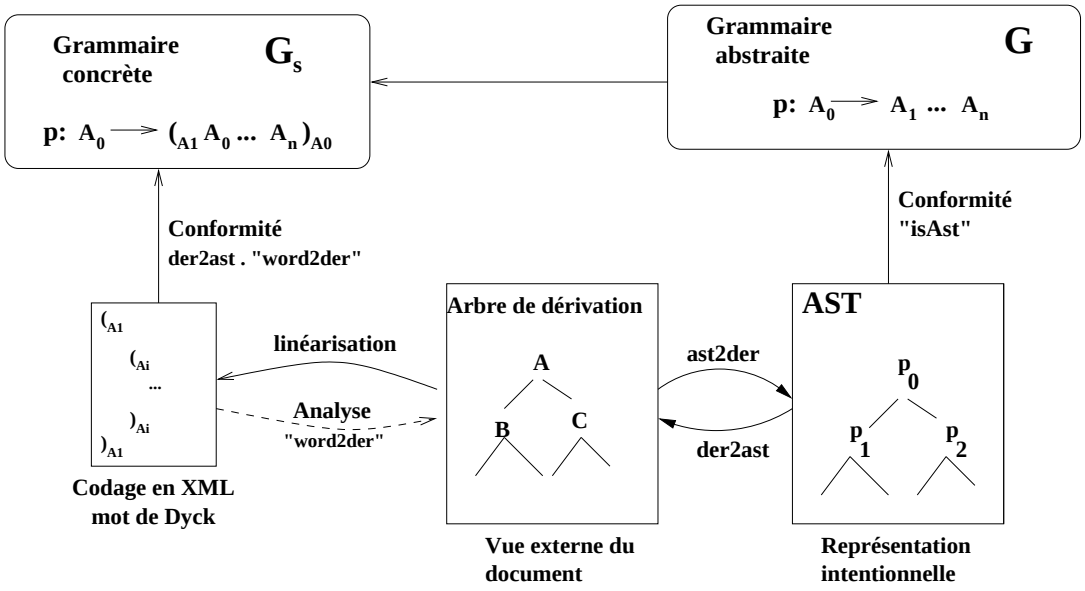

FIGURE 1.5 – Diverses représentations d'un document structuré : arbre de syntaxe abstraite, arbre de dérivation, représentation à la XML

1.3 Vues, projections 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 les sections précédentes, 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 liée à ses catégories syntaxiques.

1.3.1 Vue et projection associée

Une vue $\mathcal{V}$ est un sous-ensemble de symboles grammaticaux ($\mathcal{V} \subseteq \mathcal{S}$). 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 $\mathcal{V}_1 = \{A, B\}$ et $\mathcal{V}_2 = \{A, C\}$ sur l'alphabet $\{A, B, C\}$ sont définies comme suit :

```
viewAB :: Symb -> Bool
viewAB symb = case symb of
       A -> True
       B -> True
       C -> False
```

```
viewAC :: Symb -> Bool
viewAC symb = case symb of
       A -> True
       B -> False
       C -> True
```

À chaque vue est associée une opération de 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. Le code suivant donne une implémentation de cette opération de projection :

```
1 projection :: (symb -> Bool) -> Tree symb -> [Tree symb]
2 projection view der = if view (top der) then [Node (top der) sons]
3                       else sons
4         where sons = concat (map (projection view) (succ_ der))
```

La figure 1.6 montre un arbre de dérivation *der* et ses deux projections *derAB=projAB der* et *derAC=projAC der* suivant les vues {*A*, *B*} et {*A*, *C*} respectivement.

```
projAB = projection viewAB        projAC = projection viewAC
```

Dans la suite, le fait que t' est la projection de t suivant la vue $\mathcal{V}$ sera noté $\pi_{\mathcal{V}}(t) = t'$.

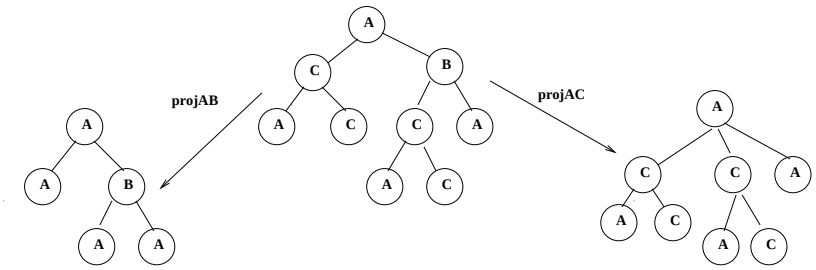

FIGURE 1.6 – Un arbre de dérivation et ses projections suivant {A,B} et {A,C}

1.3.2 Lien avec la sérialisation

On peut définir la même opération de 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. Le code de cette opération est le suivant :

```
1 projection_ :: (symb -> Bool) -> [Dyck symb] -> [Dyck symb]
2 projection_ view = filter g
3          where  g (Open symb) = view symb
4                 g (Close symb) = view symb
```

On peut facilement établir que

$$linearisation \circ (projection\ view) \;=\; (projection\_\ view) \circ linearisation$$

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

$$trace\ view\ der \;=\; (projection\_\ \ view)\ (linearisation\ der)$$

et ainsi la projection d'un arbre de dérivation peut alternativement être obtenue par analyse de sa trace visible (voir la figure 1.7) :

$$t \;=\; projection\ view\ der \quad ssi \quad Just\ t \;=\; analyse\ (trace\ view\ der)$$

1.3.3 Grammaire avec vues

De la notion de vue définie précédemment, on peut définir (tout comme pour les arbres de dérivation) une opération de projection sur les grammaires

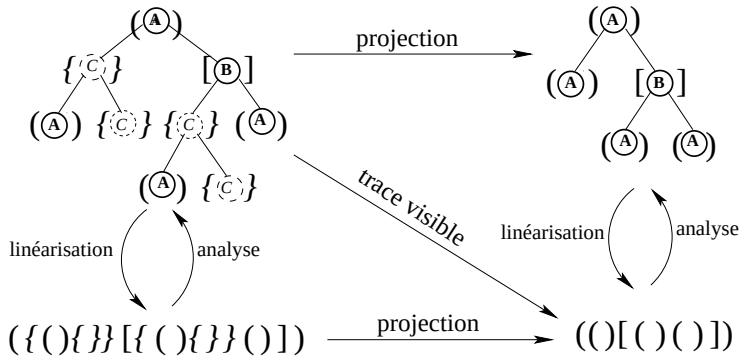

FIGURE 1.7 – Relations entre projections et sérialisation

algébriques abstraites telle que, si $G^{'}$ est la grammaire obtenue par projection de la grammaire G suivant la vue $\mathcal{V}$ et on note $\pi_{\mathcal{V}}(G) = G^{'}$, alors pour tout AST t conforme à G, la projection de t suivant $\mathcal{V}$ est conforme à $G^{'}$.

Si on note $\therefore$ la relation de conformité, on a : $(\forall t \therefore G,\ \pi_{\mathcal{V}}(t) \therefore G^{'}, \textit{avec}\ G^{'} = \pi_{\mathcal{V}}(G))$. En considérant à nouveau l'édition collaborative, si la vue $\mathcal{V}_i$ représente les catégories syntaxiques pertinentes pour les auteurs résidants sur le site i, la grammaire obtenue par projection de la grammaire globale suivant $\mathcal{V}_i$ sera le modèle des documents (partiels) édités sur ce site.

Définition 1.3.1. *Une* ***grammaire avec vue*** $G_{\mathcal{V}} = (\mathcal{S}_{\mathcal{V}}, T_{\mathcal{V}}, \mathcal{P}_{\mathcal{V}}, \mathcal{A}_{\mathcal{V}})$ *est une grammaire algébrique (concrète) obtenue par projection d'une grammaire algébrique abstraite* $G = (\mathcal{S}, \mathcal{P}, \mathcal{A})$ *suivant la vue* $\mathcal{V} \subseteq \mathcal{S}$ *comme suit :*

- $\mathcal{S}_{\mathcal{V}} = \mathcal{S}$*, les symboles non terminaux de* $G_{\mathcal{V}}$ *sont les symboles grammaticaux de* G*.*
- $T_{\mathcal{V}} = Dyck\ \mathcal{V}$*, les symboles terminaux de* $G_{\mathcal{V}}$ *sont obtenues à partir des symboles de la vue en associant à chacun d'eux une parenthèse ouvrante et une parenthèse fermante.*
- $\mathcal{A}_{\mathcal{V}} = \mathcal{A}$*, les deux grammaires ont le même axiome,*
- *Les productions de* $\mathcal{P}_{\mathcal{V}}$ *sont obtenues à partir de celles de* $\mathcal{P}$ *en insérant leur partie droite entre une paire de parenthèses lorsque le symbole en partie gauche de la production est visible. Les productions de* $\mathcal{P}_{\mathcal{V}}$ *sont donc les productions* $p : A_0 \to A_1 \cdots A_n$ *de* $\mathcal{P}$ *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* $\mathcal{P}$ *et* A_0 *un symbole visible.*

Exemple : Si on considère à nouveau la grammaire G_{run} de la page 15, la grammaire $G_{\mathcal{V}}$ obtenue par projection de G_{run} suivant $\mathcal{V} = \{A, B\}$ est

donnée par $G_{\mathcal{V}} = (\{A, B, C\}, \{A, B\}, P_{\mathcal{V}}, A)$ et $P_{\mathcal{V}}$ contient les productions :

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

Une grammaire avec vue a même ensemble de symboles non terminaux que la grammaire abstraite à laquelle elle est associée ainsi que le même nombre de productions. Bien plus, la trace visible d'un arbre de dérivation *der* pour la grammaire abstraite G correspond également au mot reconnu par la grammaire algébrique $G_{\mathcal{V}}$ associé à la vue $\mathcal{V} \subseteq \mathcal{S}$ pour l'arbre de dérivation relatif à cette grammaire et ayant le même arbre de syntaxe abstraite que *der*. Il en découle qu'on peut décider (en local) de la validité d'un réplicat partiel en cours d'édition par rapport à la grammaire globale. En fait, un réplicat partiel $t_{\mathcal{V}}$ sera dit valide par rapport à la grammaire globale G s'il existe un ast t tel que $t_{\mathcal{V}} = \pi_{\mathcal{V}}(t)$. Pour décider de la validité d'un réplicat patiel $t_{\mathcal{V}}$, on peut soit écrire un analyseur sur les mots comme dans [HM98, Fok95] pour la grammaire $G_{\mathcal{V}}$ et l'appliquer à la linéarisation de $t_{\mathcal{V}}$ puis, de tester si le résultat est défini, soit utiliser l'algorithme d'expansion qui sera présenté plus loin (sec. 3.1).

La figure 1.8 donne un aperçu général des différentes notions présentées dans cette section.

1.3.4 Problème de la projection inverse

Le problème de la projection inverse ou expansion consiste à déterminer tous les arbres de syntaxe abstraite donnant lieu à une projection visible donnée. Ces arbres peuvent être en nombre infini. En effet, si on considère à nouveau la grammaire G_{run} de la page 15, 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 partielles obtenues par projections de l'arbre de dérivation de la figure 1.6 ; les ensembles $\{der \mid projAB\ der = derAB\}$ et $\{der \mid projAC\ der = derAC\}$ sont tous les deux infinis comme nous l'illustre la figure 1.9 où il suffit d'itérer sur la production $p_6 : C \to C\ C$ pour produire un ensemble infini de solutions.

Par contre un arbre de dérivation pour la grammaire G_{run} 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.

Remarque 1.3.2. *Le problème de la couverture d'une grammaire abstraite par un ensemble de vues est indécidable.*

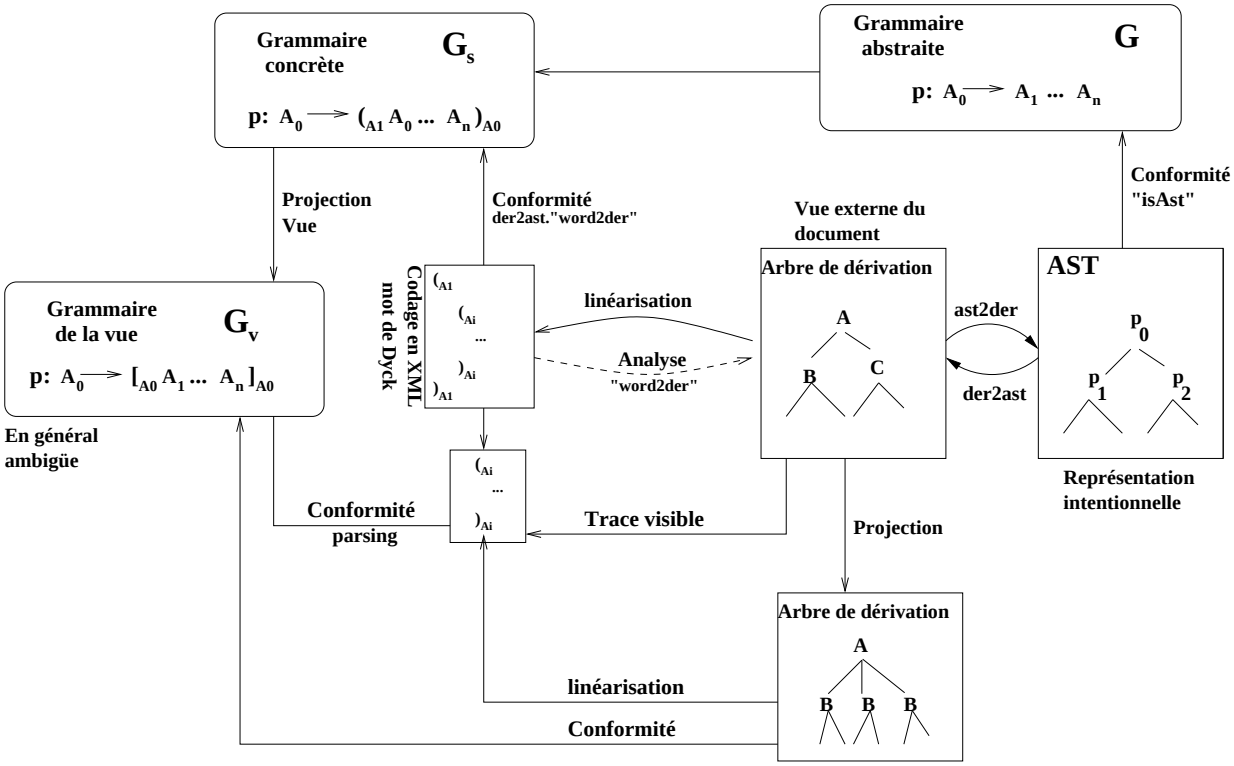

FIGURE 1.8 – Relations entre grammaires et diverses représentations d'un document structuré

Le problème de l'ambiguïté des grammaires algébriques, qui est indécidable, peut en effet se réduire au problème de couverture d'une grammaire abstraite par un ensemble de vues. Cette réduction provient de l'observation suivante. 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 des productions explicites $A \rightarrow \epsilon$ pour tout symbole terminal. Si on considère la vue constituée des 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. Décider de l'ambiguïté de la grammaire algébrique équivaut à décider si la grammaire abstraite associée est couverte par le singleton constitué par la vue associée aux symboles terminaux.

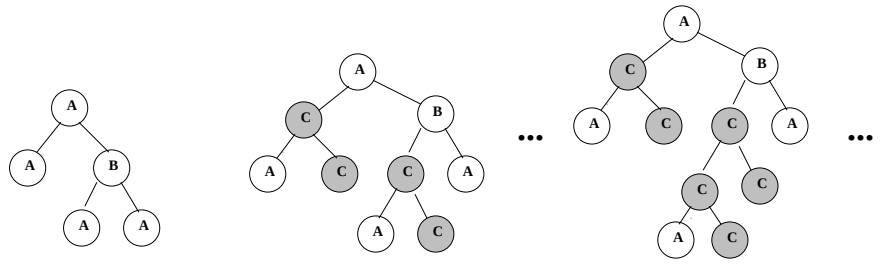

FIGURE 1.9 – Les expansions possibles d'une vue partielle

1.3.5 Problèmes de cohérence de vues et de fusion des mises à jour locales

Le problème de la fusion des mises à jour locales consiste à déterminer s'il existe un document (global) intégrant toutes les informations contenues dans celles-ci. La fusion vise aussi à terme à répercuter les diverses modifications, lorsqu'elles ne rentrent pas en conflit, d'une vue à une autre (fig. 1.10). Plus formellement, nous modélisons un document en cours d'édition sous la forme d'un arbre contenant des *noeuds ouverts*, appelés aussi *bourgeons*. Ce sont des feuilles à partir desquelles la structure peut se développer. Nous déduisons une relation d'ordre naturelle notée $\leq$ sur les documents telle que $t_1 \leq t_2$ si t_2 peut être obtenu de t_1 en remplaçant des bourgeons de t_1 par des arbres ; nous disons alors que t_2 est une *mise à jour* de t_1. Un document est complet, ou clos, s'il ne contient aucun bourgeon, c'est donc un élément maximal pour cet ordre. Si on a n vues $\mathcal{V}_1, \ldots, \mathcal{V}_n$ et n vues partielles $u_1, \ldots u_n$ correspondantes, fusionner $u_1, \ldots u_n$ consiste à rechercher un document t satisfaisant

$$\forall i \in 1, \cdots, n \quad (\exists t_i \leq t \quad \pi_{\mathcal{V}_i}(t_i) = u_i)\,,$$

où $\pi_{\mathcal{V}}(t) = u$ signifie que l'arbre u est la projection du document t suivant la vue $\mathcal{V}$.

Dans un premier temps on se restreint aux documents complets, le problème se réduit à celui de la cohérence de vues. 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 (fig. 1.11). En d'autre termes, cela revient à construire un arbre de syntaxe abstrait t tel que, étant données n vues partielles $t_1, \ldots t_n$ et n vues $\mathcal{V}_1, \ldots, \mathcal{V}_n$, la projection de l'AST t suivant la vue $\mathcal{V}_i$ est $t_i : \pi_{\mathcal{V}_i}(t) = t_i,\ i \in 1 \ldots n$. La solution au problème de cohérence de vues utilise un algorithme d'expansion qui construit une représentation de l'ensemble des documents ayant une vue donnée. Le problème de la cohérence de vues consiste à déterminer si l'intersection des expansions des différentes vues est non vide, et à fournir un élément de cette

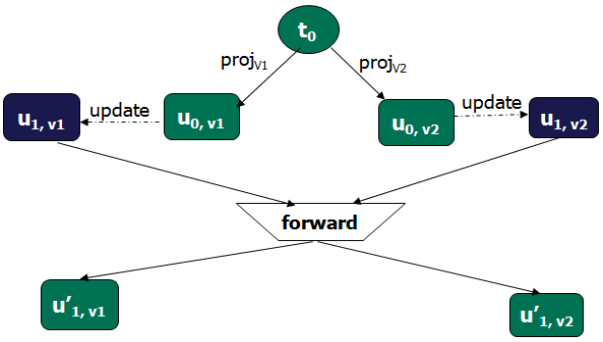

FIGURE 1.10 – Fusion et répercution des mises à jour locales

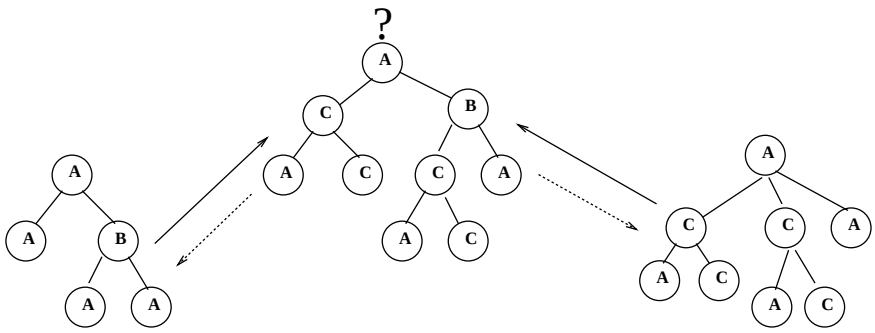

FIGURE 1.11 – Cohérence de deux vues

intersection si tel est le cas (fig. 1.12(a)). Comme déjà observé à la section 1.3.4, l'expansion d'une vue partielle est généralement infinie. il n'est donc pas possible de générer chacun de ces ensembles avant d'en calculer l'intersection. Par ailleurs, même si ces ensembles étaient finis cette procédure serait trop coûteuse puisqu'elle serait amenée à produire beaucoup de solutions qui devraient être écartées par la suite. L'idée est donc d'implémenter les algorithmes d'expansion associés à chaque vue sous la forme de coroutines qui construisent de façon paresseuse une représentation de leur espace de solution respectif de telle sorte que chacune d'entre elles, au fur et à mesure de son exécution, contribue à restreindre l'espace de recherche des autres. Ces co-routines collaborent ainsi pour engendrer de façon paresseuse l'ensemble des arbres de syntaxe abstraite appartenant à l'intersection recherchée (fig. 1.12(b)).

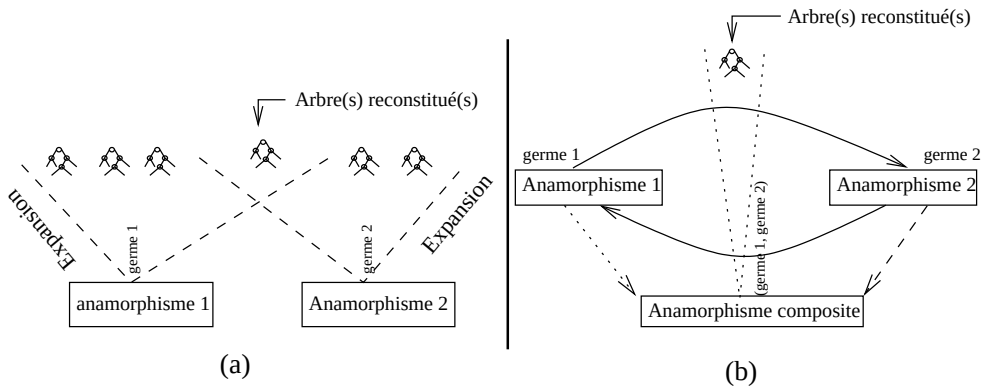

FIGURE 1.12 – Cohérence au moyen des anamorphismes

1.4 DTD et grammaire XML

Dans la communauté XML, le modèle des documents est généralement spécifié à l'aide d'une définition de type de document (DTD) ou par un schéma XML [W3C00]. On montre que ces DTDs sont équivalentes à des grammaires (régulières) ayant des caractéristiques particulières appelées *grammaires XML* [BB00]. Les grammaires (algébriques) sont donc une généralisation des DTDs et de part les études dont elles ont fait l'objet, principalement comme modèle formel pour la spécification des langages de programmation, elles offrent un cadre idéal pour l'étude formelle des transformations mises en jeux dans les technologies XML. C'est pourquoi nous les utilisons dans nos travaux comme outils de spécification de la structure des documents.

En ne s'intéressant qu'à la structure (abstraite) des documents XML, c'est-à-dire en ignorant le texte qu'ils contiennent, si on note δ la dtd associée à un document donnée et A (resp. $\bar{A}$) l'ensemble des balises ouvrantes (resp. fermantes), un document XML conforme à δ sera un *mot premier de Dyck* sur l'alphabet $A \cup \bar{A}$ [5]. Ceci suggère que la structure de tels documents (XML) peut être spécifiée au moyen de grammaires algébriques abstraites. En fait, si le texte de la dtd δ est rédigé dans le fichier δ.dtd, lors de la rédaction du contenu de ce fichier, l'élément racine des documents conformes à δ est introduit par le mot clef *DOCTYPE* et l'ensemble des règles associées à une balise (un tag) est introduit par le mot clé *ELEMENT* . De la syntaxe de l'ensemble des règles associés à un élément, on peut déduire une correspondance *un-pour-un* entre une paire de balises (ouvrante et fermante) et les

5. Ici on a fait l'hypothèse suivant laquelle à chaque balise ouvrante correspond une balise fermante, ce qui n'est pas vrai en général.

productions d'une grammaire algébrique appelées *grammaire XML*. En fait, lors de la création d'un élément, on précise son nom et son modèle de contenu au moyen d'une *expression régulière* spécifiant l'ensemble des parties droites des productions associées à l'élément. On peut alors définir une procédure de dérivation d'une grammaire XML à partir d'une DTD et vice-versa [BB00]. Avant de présenter ces procédures, nous introduisons la définition suivante relative aux grammaires XML.

Définition 1.4.1. *Soit $G = (\mathcal{S}, \mathcal{T}, \mathcal{P}, \mathcal{A}xiom)$ une grammaire algébrique concrète. G est une* ***grammaire XML*** *si :*

1. *$\mathcal{T} = A \cup \bar{A} \cup B$ tel que : A, $\bar{A}$, B forment une partition de T.*
   B permet de prendre en compte le contenu textuel des éléments. Si on ne s'intéresse qu'à la structure du document et non à son contenu, on a $B = \phi$ et on dit que ***la grammaire est pure****.*
2. *A chaque élément X_a de $\mathcal{S}$ correspond (de façon unique) un élément a de A et un élément $\bar{a}$ de $\bar{A}$.*
3. *Pour tout élément a de A, on associe un ensemble régulier $R_a \subset (S \cup B)^*$ permettant de définir un ensemble (potentiellement infini) de productions $X_a \rightarrow aR_a\bar{a}$.*

Un langage XML est un langage dénoté par une grammaire XML.

1.4.1 Grammaire XML dérivée d'une DTD

Soit δ une DTD. L'algorithme suivant permet de construire une grammaire XML (pure) $G_\delta = (\mathcal{S}, \mathcal{A} \cup \bar{\mathcal{A}}, \mathcal{P}, \mathcal{A}xiom)$ engendrant le même langage que δ.

1. A chaque élément a de la DTD δ, créer les symboles grammaticaux suivants pour la grammaire G_δ : un non terminal X_a dans $\mathcal{S}$, un terminal a dans $\mathcal{A}$ ainsi que son dual $\bar{a}$ dans $\bar{\mathcal{A}}$.
2. L'axiome de la grammaire G_δ est le non terminal X_a associé au type de document (*DOCTYPE a*).
3. A chaque règle de la DTD δ définissant un élément, ie. les lignes de la forme : *<!ELEMENT a Rhs>*, créer la famille de productions $X_a \rightarrow a\ (Rhs')\ \bar{a}$; dans G_δ où Rhs' est obtenu en remplaçant dans Rhs chaque occurrence d'un élément b de la DTD δ par X_b, le non terminal correspondant dans G_δ.

Illustration. Exemple tiré de [BB00].
Considérons la DTD δ_{ex} suivante :

```
<!DOCTYPE a [
        <!ELEMENT a ((a|b), (a|b))>
        <!ELEMENT b (b)*>
        ]>
```

Appliquons la démarche décrite ci-dessus pour dériver une grammaire XML $G_{\delta_{ex}}$ pour cette DTD. δ_{ex} contient deux éléments : l'élément a et l'élément b. Associons à l'élément a le non terminal X_a et à l'élément b le non terminal X_b. L'élément a étant le type de document de la DTD δ_{ex}, le non terminal X_a sera l'axiome de la grammaire XML $G_{\delta_{ex}}$ recherché. La dernière étape de la démarche permet d'obtenir l'ensemble P des productions suivantes :

$$X_a \rightarrow a(X_a \mid X_b)(X_a \mid X_b)\bar{a}$$
$$X_b \rightarrow b(X_b)^*\bar{b}$$

Et en définitive, $G_{\delta_{ex}} = (\{X_a, X_b\}, \{a, b, \bar{a}, \bar{b}\}, P, X_a)$

1.4.2 DTD dérivée d'une grammaire XML

Soit $G = (\mathcal{S}, \mathcal{A} \cup \bar{\mathcal{A}}, \mathcal{P}, \mathcal{A}xiom)$ une grammaire XML pure (ie. $B = \phi$). On construit une DTD δ_G engendrant le même langage que G en procédant comme suit :

1. Pour chaque non terminal X_a de $\mathcal{S}$, créer un élément a pour la DTD δ_G. L'élément a_S associé à l'axiome X_{a_S} de G est le type de document de δ_G : $<!DOCTYPE\ a_S\ [\ \ldots]$
2. A chaque famille de productions de la forme $X_a \rightarrow a\ R_a\ \bar{a}$ de $\mathcal{P}$, insérer dans la DTD δ_G une clause de création d'un élément $<!ELEMENT\ a\quad R'_a>$, où a est l'élément de la DTD associé à X_a et R'_a est l'expression obtenue à partir de R_a en remplaçant chaque occurrence d'un non terminal X_b appartenant à $\mathcal{S}$ par l'élément b qui lui est associé.

Illustration : Exemple tiré de [BB00].
Soit la grammaire XML $G = (\{S, T\}, \{a, b, \bar{a}, \bar{b}\}, P, S)$ où P est donné par :

$$S \rightarrow a(S \mid T)(S \mid T)\bar{a}$$
$$T \rightarrow bT^*\bar{b}$$

En procédant comme décrit ci-dessus, on dérive la DTD δ_G suivante à partir de G :

```
<!DOCTYPE a [
        <!ELEMENT a ((a|b), (a|b))>
        <!ELEMENT b (b)*>
        ]>
```

1.5 État de l'art sur les vues XML et sur la fusion des documents structurés

1.5.1 Notion de vue XML

La notion de vue que nous utilisons dans cette thèse a aussi émergé dans la communauté XML sous l'appellation *XML view* pour désigner comme dans [CLL02] un fragment de document XML par combinaison à partir de certaines opérations de base que sont la *projection*, la *sélection* et l'*échange*. Les vues obtenues par ces opérations peuvent s'avérer invalides (par exemple si elles ne préservent pas certaines propriétés d'ordre sémantique du document de base comme les dépendances fonctionnelles entre éléments ...). Ya Bing Chen et al. proposent dans [CLL02] une méthode permettant d'obtenir des vues valides.

Les exemples suivants inspirés de ceux contenus dans [CLL02] illustrent cette notion de *vue XML* ainsi que les opérations de projection, sélection, swap permettant d'obtenir une vue XML à l'aide du langage XQuery [6][XQu].

6. Les scripts XQuery présentés dans ces exemples ont été interprété en utilisant SAXON [Sax] un processeur de requêtes XQuery écrit en java. Après avoir téléchargé et installé Saxon par exemple dans le répertoire *saxon* , l'expression XQuery est saisie dans un fichier, disons "myquery.xql". On lance l'interprétation de ce fichier au moyen de la commande suivante saisie en ligne de commande : *java -cp ./saxon/saxon9.jar net.sf.saxon.Query myquery.xql*

```
<db>
<project jno="j001">
    <supplier sno="s001">
       <part pno="p001">
         <price> 100</price>
       </part>
    </supplier>
    <supplier sno="s002">
       <part pno="p001">
        <price> 100</price>
       </part>
    </supplier>
</project>
</db>
```

Xdoc1.xml : un document XML

```
<?xml version="1.0"
          encoding="UTF-8"?>
<db>
   <project jno="j001">
      <supplier sno="s001">
         <price> 100</price>
      </supplier>
      <supplier sno="s002">
         <price> 100</price>
      </supplier>
   </project>
</db>
```

Xdoc2.xml : une vue XML (projection) du document *Xdoc1.xml* obtenue suite à l'interprétation du script *"XQuery5.xql"*.

```
<?xml version="1.0"
           encoding="UTF-8"?>
<db>
   <project jno="j001">
     <supplier sno="s002">
       <part pno="p001">
         <price> 100</price>
       </part>
     </supplier>
   </project>
</db>
```

Xdoc3.xml : une vue XML (sélection-projection) du document *Xdoc1.xml* obtenue suite à l'interprétation du script *"XQuery6.xql"*

```
<?xml version="1.0"
          encoding="UTF-8"?>
<db>
   <project jno="j001">
     <part pno="p001">
       <supplier sno="s001">
          <price>100</price>
       </supplier>
       <supplier sno="s002">
          <price>100</price>
       </supplier>
      </part>
   </project>
</db>
```

Xdoc4.xml : une vue XML (swap) du document *Xdoc1.xml* obtenue suite à l'interprétation du script *"XQuery7.xql"*

```
<db>
{
for $proj in doc('Xdoc1.xml')//project
    return
       <project jno= "{$proj/@jno}" >
         {for $supp in ($proj/supplier)
           return
             <supplier sno = "{$supp/@sno}" >
                 {$supp/part/price}
             </supplier>
         }
       </project>
}
</db>
```

XQuery5.xql : Expression XQuery (projection) pour la vue XML *"Xdoc2.xml"*

```
<db>
{
for $proj in doc("Xdoc1.xml")//project
    return
       <project jno="{$proj/@jno}">
         {for $supp in ($proj/supplier[@sno="s002"])
            return
               <supplier sno = "{$proj/$supp/@sno}">
                   {$proj/$supp/part}
               </supplier>
        }
       </project>
}
</db>
```

XQuery6.xql : Expression XQuery (sélection) pour la vue XML *"Xdoc3.xml"*

```
<db>
{
for $j in doc('Xdoc1.xml')//project
      return
         <project jno= "{$j/@jno}">
              {for $pn in distinct-values($j//part/@pno)
                 return
                   <part pno="{$pn}">
                      {for $s in $j/supplier[part/@pno=$pn]
                         return
                            <supplier sno="{$s/@sno}">
                                {$s/part[@pno=$pn]/price}
                            </supplier>
                       }
                   </part>
              }
        </project>
}
</db>
```

XQuery7.xql : Expression XQuery (swap) pour la vue XML *"Xdoc4.xml"*

1.5.2 Fusion de vues XML

Dans un environnement distribué on peut admettre la possible existence de copies d'un même document sur plusieurs sites. On appelle *réplicats* les différentes copies du *document de base*. Si les différentes copies d'un document peuvent être sujettes à des modifications au niveau des sites qui les hébergent, la synchronisation ou fusion des réplicats d'un document consiste à fusionner les mises à jours effectuées de façon désynchronisée sur ces réplicats par rapport au document de base afin d'obtenir une nouvelle copie du document de base qui intègre éventuellement toutes les modifications des différents réplicats dans le cas où elles sont non conflictuelles, ou éventuellement des indications sur les sources de conflits. L'objectif général de la synchronisation des réplicats d'un document consiste donc à détecter les mises à jour conflictuelles et de propager celles non conflictuelles.

On distingue généralement deux phases lors de la synchronisation [BP98] : la phase de *Détection des mises à jour* qui consiste à reconnaître dans les différents réplicats les noeuds (endroits) où les mises à jour ont été effectuées depuis la dernière synchronisation, et la phase de *réconciliation* qui consiste à combiner les mises à jour effectuées sur les différents réplicats pour produire

un nouvel état (document) synchronisé pour chaque réplicat. Par exemple, étant donné deux réplicats A et B sujets à des mises à jour asynchrones, leur synchronisation produira A' respectivement B' obtenu en intégrant dans A (resp. dans B) les mises à jour non conflictuelles effectuées sur B (resp. sur A). Si toutes les mises à jour effectuées sur chacun des réplicats sont non conflictuelles alors, on aura $A' = B'$.

On trouve dans la littérature plusieurs techniques de fusion qu'on peut classifier en un certains nombre de catégories non mutuellement exclusives : fusion *two-way* ou *three-way*, fusion textuelle, syntaxique, sémantique ou structurelle, fusion des réplicats ou des opérations effectuées sur les réplicats .... Nous présentons ci-dessous les techniques de fusion *two-way* et *three-way* des documents XML. Le lecteur intéressé pourra consulter [Men02] pour en savoir plus sur les autres techniques de fusions.

1.5.3 Technique de fusion two-way versus three-way.

Ce sont des techniques utilisées pour la fusion de documents. La principale différence entre elles réside dans le fait que, dans la technique two-way (fig. 1.13) l'algorithme de fusion essaye de fusionner les réplicats sans tenir compte du replicat de base (l'ancêtre commun des réplicats à fusionner) ce qui n'est pas le cas de la technique three-way (fig. 1.14) dans laquelle le réplicat de base est utilisé. On montre que le fait d'utiliser le réplicat de base rend la technique three-way plus puissante que la technique two-way car, dans plusieurs cas, la technique two-way signalera des conflits de fusion qui n'en sont pas véritablement si on utilise la technique three-way. Par exemple dans la figure 1.15 un algorithme de fusion utilisant la technique two-way signalera une situation de conflit car ne sachant pas décider si le résultat de la fusion des réplicats t_1 et t_2 doit-être l'arbre de la figure 1.15*(a)*, ou celle de la figure 1.15*(b)*, ou alors celle de la figure 1.15*(c)*. En fait, il n'y a aucun moyen de savoir *a priori* dans la technique two-way si les noeuds *f*, *h*, et *i* sont issus des opérations d'édition ou de suppressions par rapport au document de base. Un algorithme utilisant la technique three-way pourra utiliser le document de base pour lever ce dilemme.

Un certain nombre d'options peuvent être fixée pour permettre un traitement automatique de certains conflits par l'algorithme de fusion. Par exemple : Que doit-on faire si un noeud est déplacé dans le réplicat T_1 et mis à jour dans le réplicat T_2 ?, si un noeud est déplacé (move) dans le réplicat T_1 et supprimé dans le réplicat T_2 ? Comment fusionner les listes *ab*, *abi*, *abj* qui sont des listes d'un même noeud de T_0, T_1, T_2 respectivement (T_0 est le réplicat de base)...

Selon le type d'application qu'on manipule on peut se donner un réplicat

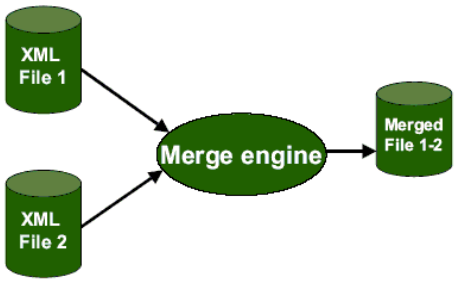

FIGURE 1.13 – Fusion suivant la technique "two-way" de documents [Fon02]

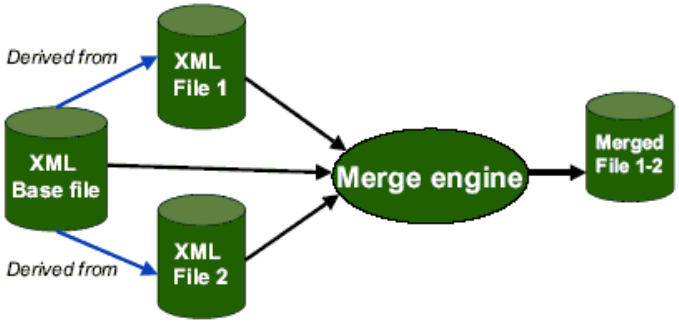

FIGURE 1.14 – Fusion suivant la technique "three-way" de documents [Fon02]

(disons T_1) ayant la priorité, tel que en cas d'un conflit *move-update* ou *move-delete* par exemple, ce sont toujours les modifications effectuées sur T_1 qui l'emportent. On peut aussi requérir l'intervention manuelle d'un opérateur humain, car après détection d'un conflit, sa résolution peut-être manuelle ou entièrement automatisée.

Certains conflits peuvent être résolus en permutant l'ordre d'application des opérations. C'est le cas par exemple d'une opération consistant à renommer une variable pouvant apparaître à différents endroits dans un réplicats. Si cette opération est parallèle à une autre opération mettant à jour un autre réplicat, lors de la fusion, la seconde opération doit d'abord être appliquée avant la première renommant la variable afin de garantir que toutes les occurrences de la variable seront bel et bien renommées.

Certains conflits aussi peuvent être résolus en accordant une priorité aux opérations effectuées sur un réplicat donné. C'est le cas par exemple si lors de la fusion d'un fichier texte la même ligne a été modifiée en parallèle sur deux réplicats. On peut juste considérer la modification effectuée par le réplicat le plus prioritaire et ignorer celle effectuées par l'autre. Un exemple d'algorithme de fusion "two-way" de document XML conforme à une DTD

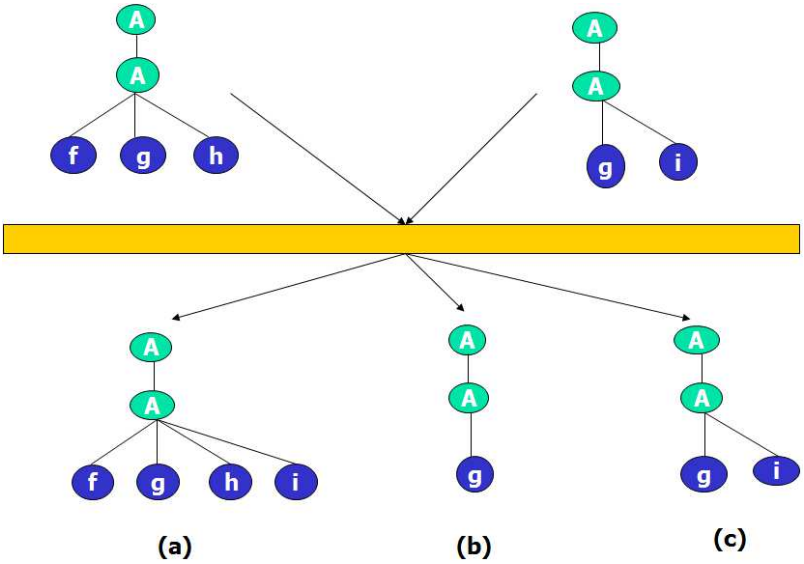

FIGURE 1.15 – Dilemme insert/delete

est présenté dans [Man01] et un autre de fusion "three-way" dans [Lin04].

Dans l'édition désynchronisée, on distingue généralement deux formes principales d'éditions suivant que les modifications opérées sur un réplicat sont répercutées immédiatement sur les autres (mode synchrone) ou ultérieurement (mode asynchrone). Des exemples d'études portant sur le mode d'édition synchrone sont effectués dans [ZHT04, MHT] où la cohérence entre les réplicats et le document de base est garantie par le fait que chaque opération d'édition sur un réplicat (resp. sur le document de base) est directement répercutée sur le document de base (resp. les réplicats) en utilisant des transformations bi-directionnelles d'arbres. Des études portant sur le mode asynchrone qui nous intéresse particulièrement sont effectué dans [Man01, Lin04]. Dans ces études, la problématique principale est de construire un document XML (de base) conforme à une DTD δ donnée en fusionnant des réplicats (des vues XML) conforment à la même DTD δ. Le document de base ainsi que ses vues doivent être conformes à la même DTD, ce qui n'est pas le cas dans nos travaux car, "nos réplicats" sont plutôt des vues partielles (réplicats partiels) du document de base et ne sont généralement pas conforme à la grammaire de base mais, plutôt à une de ses projections.

1.6 Synthèse

Dans ce chapitre, nous avons présenté des outils pouvant intervenir dans la mise en oeuvre d'un environnement d'édition collaboratif. Nous avons aussi présenté les problèmes majeurs qui peuvent avoir cours dans ce type d'édition. Ces problèmes surgissent principalement au moment de la re-synchronisation des différents réplicats et leur nombre peut être considérablement réduit si des dispositions sont prises localement pour s'assurer au préalable de ce que le réplicat à envoyer pour la fusion satisfait déjà à un certains nombre de pré-conditions. Dans le modèle que nous proposons, on peut, après édition locale d'un réplicat partiel s'assurer qu'elle reste fusionnable en examinant sont expansibilité ie., tester la vacuité de son expansion vis-à-vis de la grammaire globale. Ce test n'est possible que si on sait déjà construire l'ensemble (potentiellement infini) représentant l'expansion d'un réplicat partiel. Il est donc nécessaire de trouver une technique permettant de représenter de façon finie un ensemble infini d'objets (d'arbres) afin de pouvoir les manipuler : c'est l'objet du chapitre suivant.

Chapitre 2

Une structure de données paresseuse pour des ensembles d'arbres : les arènes

Sommaire

2.1 Structures de données paresseuses et leurs manipulations **39**
2.2 La structure d'arène **51**
2.3 Synthèse **58**

La projection inverse ou *expansion* d'une vue partielle $t_{\mathcal{V}}$ (dont l'algorithme est donné à la section 3.1) vise à construire l'ensemble des arbres de syntaxe abstraite (AST) ayant $t_{\mathcal{V}}$ comme projection visible. Cette opération (projection inverse) peut produire comme résultat un ensemble d'AST infini (fig. 1.9) rendant ainsi difficile voir impossible certaines opérations (recherche d'un élément particulier, ...) qu'on pourrait chercher à appliquer *a posteriori* sur ce résultat. Il convient alors de pouvoir représenter de façon finie cet ensemble de solutions. Nous présentons donc essentiellement dans ce chapitre une structure de données co-inductive permettant de représenter des ensembles potentiellement infinis d'arbres. Les éléments de ce type de données, définit comme point fixe d'un foncteur sont appelés *arènes*, par analogie avec les arènes de la théorie des jeux : un arbre est un élément d'une arène s'il peut être considéré comme une stratégie gagnante pour le jeu qu'elle définie.

Nous montrons qu'on peut décider si une arène est vide c'est-à-dire si elle ne contient pas de stratégie gagnante. Nous montrons aussi qu'une arène peut être produite par génération paresseuse à partir d'une co-algèbre (un générateur) et d'un élément du support de celle-ci (le germe) : l'arène est

l'image du germe par l'anamorphisme associé au générateur (c'est-à-dire, l'unique morphisme de co-algèbre du générateur vers la co-algèbre terminale).

Une grammaire algébrique abstraite peut être interprétée comme un automate d'arbre (descendant) ; de même, un automate d'arbre peut être interprété comme une co-algèbre. Ce faisant, l'arène représentée par un automate d'arbre et un état initial (le germe) est une représentation de l'ensemble (régulier) d'arbres reconnu par l'automate d'arbre à partir de cet état initial.

Dans ce qui suit, avant de présenter (sec. 2.2) la structure d'arène ainsi que certaines opérations définies sur elle, nous rappelons tout d'abord (sec. 2.1) quelques notions sur les structures de données paresseuses ainsi que les techniques permettant d'interpréter (resp. de générer) les éléments de ces structures à l'aide des algèbres (resp. des co-algèbres).

2.1 Structures de données paresseuses (Haskell) et leurs manipulations

2.1.1 Grammaires algébriques, signature multi-sortes et structures de données (Haskell)

Les structures de données sont définies habituellement de manière mutuellement récursive comme point fixe d'un système d'équations. Sachant que les données qui nous intéressent sont les arbres de syntaxe abstraite, nous nous limiterons ici aux systèmes d'équations polynomiales (sommes de produits) que nous présentons à l'aide des grammaires abstraites. Les structures syntaxiques (AST) sont donc associées à un ensemble de *types* qui décrivent les différentes *catégories syntaxiques* utilisées et à un ensemble d'opérateurs typés qui permettent de construire les objets de ces catégories syntaxiques.

Définition 2.1.1. *On appelle* ***signature*** $\Sigma_G = (\mathcal{S}, \mathcal{O}P)$*, la donnée d'un ensemble fini* $\mathcal{S}$ *de sortes ou type de base et d'un ensemble fini de symboles d'opérateur* $\mathcal{O}P$ *chacun d'entre-eux ayant une arité qui est un élément de* $\mathcal{S}^* \times \mathcal{S}$*. Un tel opérateur* $op \in \mathcal{O}P$ *d'arité* $(X_{op(1)}, X_{op(2)}, \ldots, X_{op(|op|)}, X_{op(0)})$ *sera noté* $op : X_{op(1)} \times X_{op(2)} \times \ldots X_{op(|op|)} \to X_{op(0)}$ *ou sous forme currifiée* $op : X_{op(1)} \to X_{op(2)} \to \ldots X_{op(|op|)} \to X_{op(0)}$*.* $X_{op(i)},\ i \in 1 \ldots |op|$ *est la catégorie syntaxique du* i^{ieme} *arguments de op et* $X_{op(0)}$ *est la catégorie syntaxique du résultat de op.*

À une grammaire abstraite $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ on associe une signature $\Sigma_G = (\mathcal{S}, \mathcal{O}P)$ dont les sortes sont les symboles grammaticaux et dont les opérateurs sont les productions de la grammaire : la production $p : X_{p(0)} \to$

$X_{p(1)} \cdots X_{p(n)}$ est vue comme un opérateur d'arité $X_{p(1)} \times \cdots \times X_{p(n)} \to X_{p(0)}$. Ainsi, les arbres de syntaxe abstraite de $\mathbb{G}$ sont les termes pour la signature Σ_G.

À partir d'une grammaire ou d'une signature, on déduit facilement les types de données (Haskell) correspondant en procédant comme suit :

- associer un type de données à chaque catégorie syntaxique X_i qui apparaît comme le type du résultat d'au moins un opérateur (type défini) ;
- associer un constructeur $Op : X_{op(1)} \to X_{op(2)} \to \ldots X_{op(|op|)} \to X_{op(0)}$ à chaque opérateur $op \in \mathcal{O}P$;
- les catégories syntaxiques qui n'apparaissent pas comme type du résultat d'aucun opérateur sont associées à des variables de types (paramètres).

Exemple : dérivation d'une structure de données pour les listes. Les productions d'une grammaire dénotant les listes d'éléments de type a sont les suivantes :

$$\begin{aligned} cons : &\quad L \to a\ L \\ nil : &\quad L \to \varepsilon \end{aligned}$$

Le type de données Haskell déduit de cette grammaire par application de la procédure précédente est :

$$data\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

ici, on a associé au type de données paramétrique *List a* à la catégorie syntaxique L, au constructeur *Nil* la production *nil*, et au constructeur *Cons* la production *cons*. Ce type est équivalent au type prédéfini *[a]* avec *Nil* pour *[]* et *Cons* pour *(:)*.

2.1.2 Evaluation d'un élément d'une structure de données paresseuse : les algèbres

Les éléments d'une structure de données peuvent être sujets à plusieurs interprétations aussi différentes les unes que les autres. Par exemple, pour le terme *liste = Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Nil))))* appartenant au type de donnés *List a* défini ci-dessus, on peut avoir une interprétation produisant son plus grand élément, sa longueur, ...De tels interpréteurs sont spécifiés le plus souvent au moyen des algèbres.

Définition 2.1.2. ***Une algèbre*** A_Σ *associée à une signature* $\Sigma_G = (\mathcal{S}, \mathcal{O}P)$ *est la donnée d'un ensemble* $A_\mathcal{S}$ *pour chaque* $s \in \mathcal{S}$*, et d'une application* $op_A : \quad A_{s_1} \to A_{s_2} \to \ldots A_{s_{|op|}} \to A_s$ *pour tout* $op : \quad s_1 \to s_2 \to \ldots s_{|op|} \to s$

dans $\mathcal{OP}$. On suppose que le produit vide est un singleton, c'est-à-dire que si $|op| = 0$, op_A est un élément de $A_{\mathcal{S}}$. L'opérateur op est dans ce cas appelé une constante. On suppose aussi qu'il existe au moins une constante associée à chaque type $s \in \mathcal{S}$.

Il est facile de déduire à partir d'une structure de données Haskell un type (une structure) pour les algèbres qui lui sont associées permettant d'implémenter les interpréteurs pour les objets appartenant à ce type. Il suffit pour cela de procéder comme suit :

- associer un *domaine sémantique* aussi appelé *domaine d'interprétation* ou encore *domaine dévaluation* à tous les types (définis) de la structure ;
- associer une fonction d'interprétation à chaque constructeur de données de la structure. Cette fonction spécifie comment interpréter les données construites à partir du constructeur. Le type de cette fonction se déduit du type du constructeur. En effet, si le constructeur a pour type $C : \ A_{s_1} \rightarrow A_{s_2} \rightarrow \ldots \rightarrow A_{s_n} \rightarrow A_s$ alors la fonction d'interprétation f_C associée à ce constructeur aura pour type $f_C : \ sem_1 \rightarrow sem_2 \rightarrow \ldots \rightarrow sem_n \rightarrow sem$ où sem_i est le domaine sémantique associé au type A_{s_i}, et sem celui associé à A_s.

Exemple : dérivation d'une structure d'algèbre pour les listes.
Pour interpréter les listes d'objets de type a

$$data\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

on utilise la structure d'algèbre suivante :

```
1 data AlgList a b = AlgList{nil :: b,
2                            cons :: a -> b -> b}
```

où la variable de type b désigne le domaine sémantique associé au type *List a*. On peut alors définir une fonction *fold* permettant d'interpréter un objet appartenant à une structure de données au moyen d'une algèbre quelconque. Pour la structure *List a* on a[1] :

```
1 fold :: AlgList a b -> List a -> b
2 fold alg = f
```

1. les sélecteurs *nil* et *cons* apparaissant dans la définition du type *AlgList a b* ont les types suivants : *nil : : AlgList a b -> b* et *cons : : AlgList a b -> a -> b -> b*. C'est ce qui justifie les notations *nil alg* et *cons alg x (f xs)* apparaissant dans les lignes 3 et 4 du code de la fonction *fold* ci-dessus.

```
3     where f Nil = nil alg
4           f (Cons x xs) = cons alg x (f xs)
```

Illustration : Avec les définitions suivantes :

```
-- Somme des éléments d'une liste au moyen d'une algèbres
sumList :: List a -> b
sumList = fold algSum
         where algSum = AlgList 0 (+)

-- Produit des éléments d'une liste au moyen d'une algèbres
multList :: List a -> b
multList = fold algMult
         where algMult = AlgList 1 (*)

--Recherche de l'élément maximum d'une liste
maxList :: List a -> b
maxList = fold algMax
        where algMax = AlgList 0 max

liste = Cons 1 Cons 2 Cons 3 Cons 4 Cons 5 Nil
```

on a :

```
sumList liste = 15
multList liste = 120
maxList liste = 5
```

2.1.3 Génération d'un élément d'une structure de données paresseuse : les co-algèbres

On peut également s'intéresser à la façon dont les objets d'une structure de données sont produits en utilisant la notion duale des algèbres : les co-algèbres.

Il est facile de déduire d'une structure de données Haskell un type (une structure) pour les co-algèbres qui lui sont associées et permettant de spécifier comment générer des objets appartenant à cette structure de données. Il suffit pour cela de procéder comme suit :

- associer un *domaine de génération* à chaque type de données de la structure : c'est à ce domaine qu'appartiendront les germes ;

– associer une fonction de génération à chaque type de données de la structure. Cette fonction spécifie comment seront engendrés les objets du type à partir d'un germe. Si le type A de l'objet à engendrer est une somme de produit, c'est-à-dire A est de la forme

$$\begin{array}{rcl} data\ A & = & A_1\ a_{(A_1,1)} \ldots a_{(A_1,|A_1|)} \\ & | & A_2\ a_{(A_2,1)} \ldots a_{(A_2,|A_2|)} \\ & | & \ldots \\ & | & A_n\ a_{(A_n,1)} \ldots a_{(A_n,|A_n|)} \end{array}$$

si on désigne par a le domaine de génération de A, alors il faut :
– disposer pour chaque constructeur A_i du type A d'une fonction (un prédicat) $isA_i :: a \rightarrow Bool$ [2]. L'ensemble des prédicats isA_i jouent un rôle d'aiguillage en renseignant sur l'alternative de A qui doit être générée.
– disposer pour chaque donnée $a_{(A_i,j)}$ [3], $i \in 1 \cdot \cdot n,\ j \in 1 \cdot \cdot |A_i|$ apparaissant dans le constructeur A_i d'une fonction $germ_{A_i,j} :: a \rightarrow b$ permettant de produire le germe de type b : b est le *domaine de génération* de $a_{(A_i,j)}$ et a est celui de A.

Exemple : dérivation d'une structure de co-algèbre pour les listes. Dans le cas des listes d'objets de type a

$$data\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

en désignant par b le domaine de génération associé au type défini $List\ a$ et par a celui associé aux éléments de la liste, et en suivant la démarche décrite ci-dessus on obtient la structure de co-algèbres suivante :

```
1 data CoAlgList a b = CoAlgList{stop :: b -> Bool,
2                                out :: b -> a,
3                                next :: b -> b}
```

Remarquons que la fonction *stop : : b -> Bool* donne une indication sur l'alternative de la structure liste à générer. Si *stop germ* est vrai, alors ont doit arrêter la génération ie. produire une liste vide : *Nil* ; sinon, ont doit continuer à générer la suite des éléments de la liste. Remarquons aussi qu'en

2. Les prédicats isA_i sont mutuellement exclusifs c'est-à-dire qu'il ne doit pas exister deux tels prédicats isA_i qui soient vrais pour une même valeur donnée x du domaine de génération a.

3. Les $a_{(A_i,j)}$ sont soit des types de données comme A, soit des variables de type.

suivant rigoureusement la démarche décrite ci-dessus [4], on devrait avoir deux telles fonctions (*isNil* et *isCons*). Mais comme la structure *List a* n'a que deux constructeurs (deux alternatives possibles) et compte tenu du fait que ces deux fonctions doivent être mutuellement exclusives comme précisé précédemment, on a remplacé ces deux fonctions par l'unique fonction *stop* qui assume pleinement leurs rôles.

Avec la structure de co-algèbre *CoAlgList a b*, on peut alors définir aisément une fonction de génération *gen* permettant d'appliquer une co-algèbre sur un germe pour produire une liste ; cette fonction est définie comme suit : [5]

```
1 gen :: CoAlgList a b ->  b -> List a
2 gen coalg = build
3     where build germe =
4           if (stop coalg germe) then Nil
5           else Cons (out coalg germe) (build (next coalg germe))
```

Illustration : on peut par exemple définir la fonction *zip* (donnée dans l'annexe A) qui apparie les éléments de même rang de deux listes pris en arguments au moyen d'une co-algèbre comme suit :

```
zip_ :: List a -> List b -> List (a,b)
zip_ = gen (CoAlgList stop_ out_ next_)
     where stop_ xs ys = (null xs) || (null ys)
           out (x:xs) (y:ys) = (x,y)
           next (x:xs) (y:ys) = (xs, ys)
```

2.1.4 Généralisation : les foncteurs

Les notions d'algèbres et de co-algèbres présentées dans les deux sections précédentes se généralisent dans le cadre catégoriel à des classes générales

4. Si on appliquait rigoureusement la démarche de dérivation d'une co-algèbre sur la structure *List a*, on devrait avoir :
data CoAlgList a b = CoAlgList { isNil : : b -> Bool,
isCons : : b -> Bool,
germ_ a : : b -> a,
germ_ List : : b -> b }

5. Comme précédemment les sélecteurs apparaissant dans la définition du type (*CoAlgList a b*) ont les types suivants : *stop : : CoAlgList a b -> b -> Bool*, *out : : CoAlgList a b -> b -> a*, et *next : : CoAlgList a b -> b -> b* ; ce qui justifie les notations *(stop coalg germe)*, *(out coalg germe)* et *(next coalg germe)* apparaissant dans les lignes 4 et 5 du code de la fonction *gen*.

de *foncteurs*. Dans ce qui suit nous ne faisons qu'une présentation sommaire de ces notions en convergeant rapidement vers les utilisations qui nous intéressent ; toutefois, le lecteur désireux d'en savoir plus sur les foncteurs, et sur les notions d'algèbres et de co-algèbres qui leurs sont associées aussi bien dans un cadre catégorique que sur la façon dont elles sont implémentées dans un langage fonctionnel paresseux, comme Haskell, pourra se référer par exemple aux documents [Ven00, BCG02]. 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 :

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

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* (voir fig. 2.1) :

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

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 $a \to f\ a$:

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

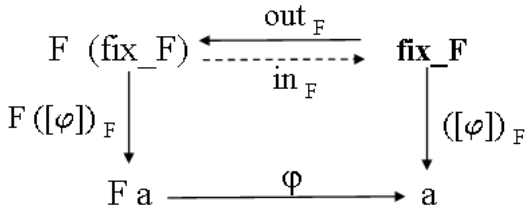

FIGURE 2.1 – Diagramme commutatif pour les algèbres

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* (voir fig. 2.2) :

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

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

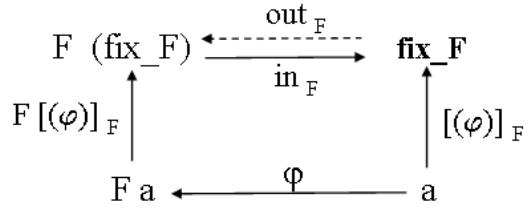

FIGURE 2.2 – Diagramme commutatif pour les co-algèbres

Exemple : foncteur, algèbre et co-algèbre pour les arbres.
Les arbres peuvent être définies par la grammaire ayant les productions suivantes :

$$\begin{aligned} node: &\quad Tree \to Label\ Forest \\ nil: &\quad Forest \to \varepsilon \\ cons: &\quad Forest \to Tree\ Forest \end{aligned}$$

Cette grammaire se traduit en Haskell par la définition récursive de structures de données suivante dans laquelle le paramètre *Label* de la grammaire devient une variable de type pour les constructeurs de types associés aux autres symboles grammaticaux :

```
data  Tree label = Node {val :: label,
                         forest :: Forest label}
data  Forest label = Nil
                   | Cons {head :: Tree label,
                           tail :: Forest label}
```

À cette grammaire est associé comme dans [BD07] le foncteur $F_{\mathbb{T}} =< F^{Tree},\ F^{Forest} >$ où $F^{Tree}(D_{Label},\ D_{Tree},\ D_{Forets}) = (D_{Label} \times D_{Forets})_{\perp}$ est le foncteur polynomial associé au symbole grammatical *Tree* et $F^{Forest}(D_{Label},\ D_{Tree},\ D_{Forets}) = 1+(D_{Tree} \times D_{Forets})_{\perp}$ le foncteur polynomial associé au symbole grammatical *Forest*.

Une algèbre $\varphi :\ F_{\mathbb{T}}a \rightarrow a$ pour le foncteur $F_{\mathbb{T}}$ se décompose alors en $\varphi = node_{\varphi} \times [nil_{\varphi},\ cons_{\varphi}]$ dans laquelle les composantes $node_{\varphi} :: x \times b \rightarrow a,\ nil_{\varphi} :: () \rightarrow b,\ cons_{\varphi} :: a \times b \rightarrow b$ sont en correspondance avec les productions de la grammaire. Les variables de type $x,\ a$, et b désignent les domaines d'interprétation des différentes catégories syntaxiques. Ceci se traduit en Haskell par :

```
data AlgTree x a b = AlgTree{node :: x -> b -> a,
                             nil :: b,
                             cons :: a -> b -> b}
```

La définition des fonctions d'évaluation est une traduction directe de l'équation

$$fold\_F_{\mathbb{T}}\ \varphi\ =\ \varphi\ \cdot \big(fmap_{F_{\mathbb{T}}}\ (fold\_F_{\mathbb{T}}\ \varphi)\big) \cdot out$$

qui caractérise le catamorphisme $([\varphi])_{F_{\mathbb{T}}}$. Ce dernier se décompose en fait en : $([\varphi])_{F_{\mathbb{T}}} =< eval^{\varphi}_{Tree},\ eval^{\varphi}_{Forest} >$ avec :

$$eval^{\varphi}_{Tree}\, o\, Node = node_{\varphi}\, o\, (id_x \times eval^{\varphi}_{Forest})$$
$$eval^{\varphi}_{Forest}\, o\, [Nil, Cons] = [nil_{\varphi}, con_{\varphi}]\, o\, (eval^{\varphi}_{Tree} \times eval^{\varphi}_{Forest})$$

ce qui se traduit en Haskell par :

```
evalTree :: AlgTree a b c -> Tree a -> b
evalTree alg (Node x f) = node alg x (evalForest alg f)

evalForest :: AlgTree a b c -> Forest a -> c
evalForest alg Nil = nil alg
evalForest alg (Cons t f) =
               cons alg (evalTree alg t) (evalForest alg f)
```

Une co-algèbre pour le foncteur $F_{\mathbb{T}}$ est une application $\varphi : \ a \rightarrow F_{\mathbb{T}}a$. Des structures de co-algèbre pour la structure de données des arbres peuvent être présentés sous la forme :

```
data CoalgTree a b c = CoalgTree{val_ :: b -> a,
                                 forest_ :: b -> c,
                                 isnil_ :: c -> Bool,
                                 head_ :: c -> b,
                                 tail_ :: c -> c}
```

où les variables de type a, b, c représentent respectivement les domaines de génération de *Label*, *Tree* et de *Forest*. Les générateurs correspondant qui sont une transcription directe de l'équation

$$unfold\_F_{\mathbb{T}}\ \varphi \ = \ In \cdot \big(fmap_{F_{\mathbb{T}}}\ (unfold\_F_{\mathbb{T}}\ \varphi)\ \big) \cdot \varphi$$

qui caractérisent l'anamorphisme $[(\varphi)]_{F_{\mathbb{T}}}$ sont alors définis en Haskell comme suit :

```
genTree :: CoalgTree a b c -> b -> Tree a
genTree coalg gen = Node (val_ coalg gen)
                         (genForest coalg (forest_ coalg gen))
genForest :: CoalgTree a b c -> c -> Forest a
genForest coalg gen
    | isnil_ coalg gen = Nil
    | otherwise = Cons (genTree coalg (head_ coalg gen))
                       (genForest coalg (tail_ coalg gen))
```

2.1.5 Grammaire, co-algèbre et automates d'arbres

Une grammaire algébrique abstraite telle que définie à la section 1.2 peut être considérée comme une application :

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

où les variables de type x et a représentent respectivement le type des symboles grammaticaux et les noms des productions. Cette application associe à un symbole grammatical *symb* la liste des couples $(prod, symbs)$ construits à partir des productions (nom de la production et liste des symboles en sa partie droite) pour lesquelles *symb* apparaît en partie gauche. Une telle application est une co-algèbre pour le foncteur (paramétrique) $F_{\mathbb{G}}\, a$ définie par

$F_{\mathbb{G}}\ a\ x = [(a,\ [x])]$ et peut s'interpréter comme un automate d'arbre (descendant) $A = (\Sigma,\ Q,\ R, q_0)$ où :

- $a = \Sigma$ est le type des étiquettes des noeuds de l'arbre à reconnaître
- $x = Q$ est le type des états (tous considérés comme finals) et,
- $q \to (A, \langle q_1, \cdots, q_n\rangle) \in R$ est une transition de l'automate lorsque la paire $(A, [q_1, \cdots, q_n])$ apparaît dans la liste $gram\ \ q$.

Un automate d'arbre descendant est défini formellement comme suit :

Définition 2.1.3. *Un **automate d'arbre** (descendant) défini sur Σ est un quadruplet $A = (\Sigma, Q, R, q_0)$ où :*

- *Σ est un ensemble de symboles avec arité (signature) ; ses éléments sont les étiquettes des noeuds des arbres à reconnaître ;*
- *Q est un ensemble d'états,*
- *$q_0 \in Q$ est l'état initial,*
- *$R \subseteq Q \times \Sigma \times Q^*$ est la relation de transition. Un élément de R est noté $q \to (\sigma, \langle q_1, \cdots, q_n\rangle)$*

Pour reconnaître un arbre à l'aide d'un automate d'arbre 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 $q \to (A, \langle q_1, \cdots, q_n\rangle)$ 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 Haskell *accept* suivante est une implémentation de cet algorithme.

```
1  type Coalg a b = b -> [(a,[b])]
2  type TreeAuto a b = Coalg a b
3
4  accept :: (Eq a, Eq b) => TreeAuto a b -> Tree a -> b -> Bool
5  accept auto  (Node p ts) symb =
6    if (length ts) == (length right)
7       then and (zipWith (accept auto) ts right)
8       else False
9     where  (left, right) = g (auto symb) symb
10           g xs s = head [(s, r)| (l, r) <- xs, l==p]
```

La fonction de génération (anamorphisme) associée à un automate d'arbre (vu comme co-algèbre) permet d'engendrer tous les arbres (ou une représen-

tation de tous les arbres) reconnus par cet automate d'arbre à partir d'un état initial donné (fig. 2.5).

La fonction de génération[6] qui suit est une transcription directe de la composition des fonctions $fold$ eu $unfold$ vues précédemment et rappelées ci-dessous :

$$fold\_F_{\mathbb{T}}\ \varphi\ =\ \varphi\ \cdot \big(fmap_{F_{\mathbb{T}}}\ (fold\_F_{\mathbb{T}}\ \varphi)\big) \cdot out$$

$$unfold\_F_{\mathbb{G}}\ auto\ =\ In \cdot \big(fmap_{F_{\mathbb{G}}}\ (unfold\_F_{\mathbb{G}}\ auto)\ \big) \cdot auto$$

```
1 gen_ :: TreeAuto a b -> b -> [Tree a]
2 gen_ auto germ = [Node x ts |(x,etats) <- auto germ
3                             , ts <- dist (map (gern_ auto) etats)]
4
5 dist :: [[a]] -> [[a]]
6 dist [ ] = [[ ]]
7 dist (xs:xss) = [y:ys | y <- xs, ys <- dist xss]
```

On peut interpréter une grammaire abstraite comme un automate d'arbre (une co-algèbre). En effet, soit $G = (S, P, Axiom)$ une grammaire abstraite. Il suffit d'associer à cette grammaire l'automate d'arbres $A_G = (\Sigma, Q, R, q_0)$ défini comme suit :

- $\Sigma = P$ {une production $p : A_0 \to A_1 \dots A_n$ est d'arité n},
- $Q = S$,
- $q_0 = Axiom$,
- $R = \{A_0 \to (p, \langle A_1, \cdots, A_n \rangle)\ \mid p : A_0 \to A_1 \dots A_n\}$

Ainsi, les symboles de l'automate sont obtenus à partir des noms des productions de la grammaire, ses états sont obtenus à partir de ceux des catégories syntaxiques de la grammaire, et chaque production de la grammaire permet de définir une transition de l'automate. La fonction *gram2treeauto* suivante donne le code Haskell correspondant à cette transformation :

```
1 gram2treeauto :: (Eq symb) => Gram prod symb -> TreeAuto prod symb
2 gram2treeauto gram symb = [(p,rhs gram p) | p <- prods gram,
3                                             symb == lhs gram p]
```

6. La fonction polymorphe `dist` utilisée dans le code de la fonction *gen_*, à la ligne 3 et définie à partir de la ligne 5 est une fonction qui permet de construire une liste d'éléments à partir d'une liste de liste d'éléments en prenant un élément de chacune des listes d'entrée et ce, de toutes les façons possibles. Par exemple : *dist [[1,2], [3,4], [5..7]]= [[1,3,5],[1,3,6],[1,3,7],[1,4,5],[1,4,6],[1,4,7],[2,3,5],[2,3,6],[2,3,7],[2,4,5],[2,4,6],[2,4,7]]*

Avec cette transformation, un arbre t sur la signature $\Sigma = P$ est un AST pour la grammaire G si et seulement si t est accepté par A_G. De même, lorsqu'on interprète une grammaire comme un automate d'arbres (une co-algèbre) *auto*, un germe est un état et *(ana auto init)* fournit une énumération (représentation) de l'ensemble des arbres reconnus par l'automate à partir d'un état (initial) *init*. Par exemple les arbres de syntaxe abstraite de la grammaire G_{run} donnée à la section 1.2 peuvent être générés par l'automate d'arbre, d'état initial q_0, dont les transitions sont les suivantes :

```
q0 -> (P1,[q2,q1]),  q1 -> (P3,[q2,q0]),   q2 -> (P5,[q0,q2])
q0 -> (P2,[])]),     q1 -> (P4,[q1,q1]),   q2 -> (P6,[q2,q2])
                                           q2 -> (P7,[])])
```

2.2 La structure d'arène

Nous avons vu à la section 1.3.4 que le résultat de la projection inverse d'une vue partielle peut générer un ensemble infini d'AST. Dans cette section nous présentons une structure de données co-inductive appelée *Arène* permettant de représenter un ensemble éventuellement infini d'arbres. Comme présenté à la section précédente, une grammaire G vue comme une application qui à un symbole *symb* associe la liste des couples (*prod*, *symbs*) est une co-algèbre pour le foncteur paramétrique $F\ a\ x = [(a, [x])]$. Ce foncteur est codé en *Haskell* par :

```
1 type F a x = [(a,[x])]
2 instance Functor (F a) where
3         -- fmap :: (x -> y) -> [(a,[x])] -> [(a,[y])]
4         fmap g xs = [(a, map g ys) | (a, ys) <- xs]
```

Le point fixe de ce foncteur est la structure polymorphe *Arena* définie en Haskell par :

```
1 newtype Arena  a = Or  {unOr  :: [(a,Arenas a)]}
2 newtype Arenas a = And {unAnd :: [Arena a]}
```

De la première définition (ligne 1) on déduit l'existence de l'isomorphisme

$$Arena\ a \ \cong\ [(a, Arenas\ a)]$$

donnée par la paire de fonctions associées respectivement au constructeur *Or* et au sélecteur *unOr*.

```
Or :: [(a,Arenas a)] ->  Arena a
unOr :: Arena a -> [(a,Arenas a)]
```

Une arène `arena` représente un ensemble d'arbres dont la structure est conforme aux motifs donnés dans la liste `unOr arena` (une disjonction). Chaque tel motif est une paire `(a, arenas)` constituée d'un élément `a` et d'une liste d'arènes (une conjonction). Un arbre est conforme à ce modèle si sa racine est étiquetée `a` et ses sous-arbres sont conformes aux modèles respectifs contenus dans la liste `unAnd arenas`. L'isomorphisme dérivé

$$Arena\ a \cong [(a, [Arena\ a])]$$

montre bien que cette structure de données est le point fixe du foncteur `F a` où `F a x = [(a, [x])]`. La fonction de génération (anamorphisme) associée à la grammaire G vue comme co-algèbre qui permet de construire l'arène donnant une représentation de l'ensemble des ASTs de la grammaire G est définie comme suit (voir aussi fig. 2.5) :

```
1 type Coalg a b = b -> [(a,[b])]
2 ana :: Coalg a b -> b -> Arena a
3 ana coalg germ = Or[(a, And (map (ana coalg) germs)) |
4                                  (a, germs) <- coalg germ]
```

Nous représentons graphiquement une arène `arena` par un arbre ayant deux types de noeuds. Un exemple d'arène est donné dans la figure 2.3 dans laquelle les annotations (*) et (**) signifient que les sous arbres issus des noeuds avec la même annotation sont identiques. Les noeuds "ou" représentent des arènes et les noeuds "et" représentent des listes d'arènes. Un noeud "ou" représentant une arène `t = Or atss` aura autant de successeurs que d'éléments `(a,ts)` dans la liste `atss` ; 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`. Ainsi si l'ensemble `atss` 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 d'arènes `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 (arena) de cette liste. Lorsque cette liste est vide ce noeud "et" n'a donc aucun successeur et il sera noté $\top$.

L'interprétation d'une arène comme représentation d'un ensemble d'arbres est la suivante : un arbre `t=Node a ts` est membre de l'arène `arena` s'il existe, en partant de la racine de l'arène `arena`, une branche étiquetée `a` menant à

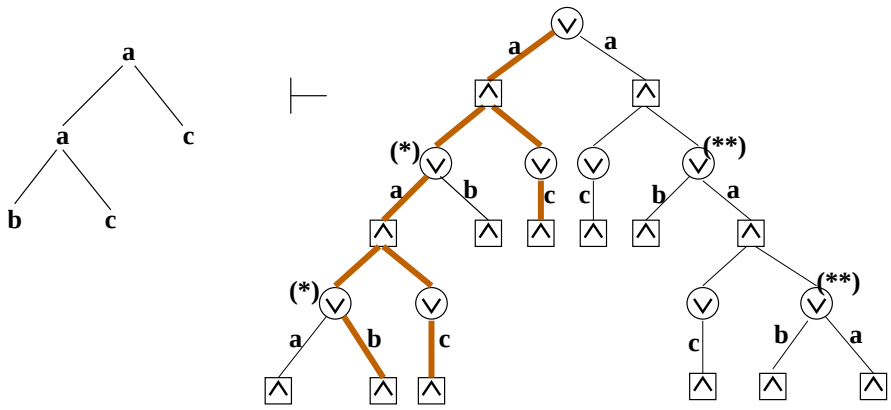

FIGURE 2.3 – Structure de données paresseuse pour les ensembles d'arbres de dérivation

un noeud "et" ayant autant de successeurs que la longueur de `ts` et tel que de façon récursive, chacun des arbres de `ts` est membre du successeur de ce noeud ayant le même rang. Ce qui correspond à la fonction suivante :

```
1 isMember :: (Eq a) => Tree a -> Arena a -> Bool
2 isMember (Node a ts) arena =
3   or [and (zipWith isMember ts (unAnd arenas))|
4                     (elet,arenas) <- unOr arena,
5                      elet  == a,
6                     (length ts)==(length (unAnd arenas))]
```

Par exemple les arcs représentés en gras dans la figure 2.3 établissent l'appartenance de l'arbre

```
Node a [Node a [Node b [], Node c []], Node c []]
```

à l'ensemble d'arbres représenté par l'arène indiqué dans cette figure. Il est facile de voir que cet ensemble est constitué des arbres ayant une des deux formes de la figure 2.4.

On observe qu'un arbre (fini) qui appartient à (l'ensemble d'arbres défini par) une arène peut être identifié au sous arbre de l'arène 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 de l'arène 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 l'arène (et on l'identifie à son "image" dans l'arène).

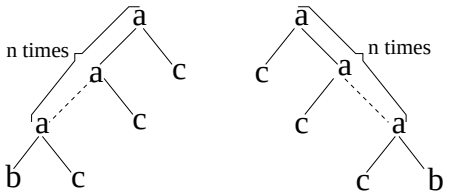

FIGURE 2.4 – Arbres membres de l'ensemble représenté par l'arène de la figure 2.3

De même on parlera d'un arbre inscrit en un noeud "ou" d'une arène pour désigner un arbre inscrit dans l'arène issu de ce noeud "ou". Intuitivement pour tracer un arbre inscrit dans une arène 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 $\top$).

2.2.1 Extension d'une arène

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 d'une arène 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 une arène équivalente sans noeud $\bot$, hormis éventuellement la racine qui en est alors l'unique noeud.

On peut énumérer les éléments d'une arène si elle est *finie* au moyen de la fonction `enumerate` ci-dessous :

```
enumerate :: Arena -> [Tree a]
enumerate arena = [Node elet ts |
                        (elet,arenas) <- unOr arena,
                         ts <- dist (map enumerate (unAnd arenas))]
```

et tester sa vacuité par la fonction

```
isEmpty :: Arena a -> Bool
isempty arena = and[or (map isEmpty (unAnd arenas)) |
                     (_,arenas) <- unOr arena]
```

Nous allons utiliser la structure d'arène *Arena* pour représenter des arbres de syntaxe abstraite (fig. 2.5). Les étiquettes des noeuds sont donc les noms des 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 (un nom de production). Et donc en particulier une arène 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, une arène aura nécessairement au moins une branche infinie dès qu'elle représente un ensemble infini d'arbres ; c'est justement pour cette raison que nous avons introduit cette structure de données.

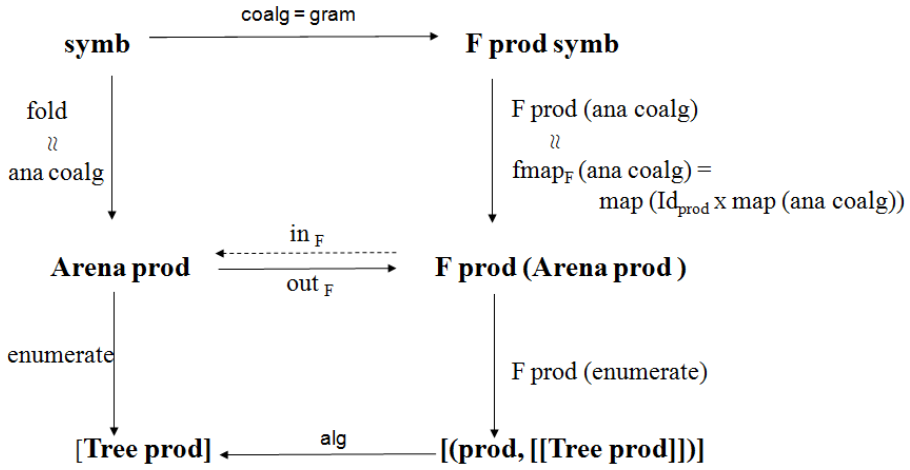

FIGURE 2.5 – Génération des arènes et énumération de leurs éléments

2.2.2 Elagage et nettoyage d'une arène

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 $q \to (A, \langle q_1, \cdots, q_n \rangle)$, 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 $q \to (A, \langle q_1, \cdots, q_n \rangle)$ 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 d'arène. 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 :

1. 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).
2. 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 2.6 donne le résultat de cette procédure appliquée à l'arène de la figure 2.3. Dans cette figure on représente également le résultat de l'opération de nettoyage (suppression des noeuds $\perp$ internes) après élagage.

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 l'arène de la figure 2.6 est réduit aux deux arbres suivants :

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

Proposition 2.2.1. *Une arène est vide si, et seulement si, son élagage est vide.*

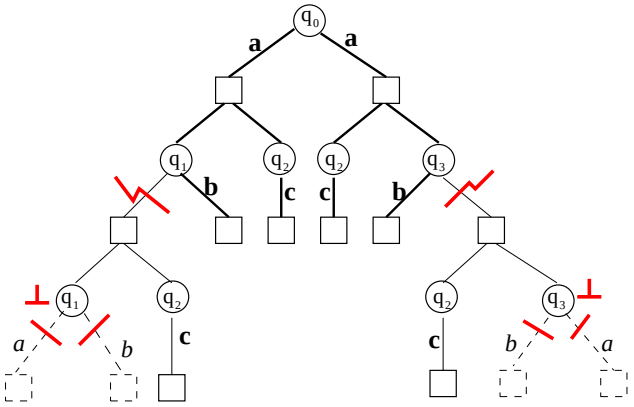

FIGURE 2.6 – en traits pleins : élagage de l'arène de la figure 2.3

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'une arène non vide est lui même non vide. Pour cela nous introduisons une opération de réduction qui transforme tout arbre inscrit dans une arène en un arbre inscrit dans son élagage. On considère donc un arbre inscrit dans l'arène. 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 arènes issus de ces deux noeuds sont les mêmes, puisque engendrés à partir du même état, le résultat obtenu est encore un arbre inscrit dans l'arène 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 (arène) élagué. □

On en déduit l'algorithme suivant qui permet de décider si le langage d'une arène 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 de l'arène. 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.

```
1 void :: (Eq b) => Coalg a b -> b -> Bool
2 void auto init = isVoid [] init where
3 isVoid path state | elem state path = True
4                   | otherwise =
5                         and [or(map (isVoid (state:path)) states)
```

```
6                                   | (_,states) <- auto state]
```

Sur le même principe la fonction suivante énumère les arbres se trouvant dans l'élagage de l'arène 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 de l'arène 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).

```
1 enum :: (Eq b) => Coalg a b -> b -> [Tree a]
2 enum auto init = enum_ [] init where
3 enum_ path state | elem state path = []
4                  | otherwise =
5                     [Node elet trees |
6                      (elet,states) <- auto state,
7                      trees <- dist (map (enum_ (state:path)) states)]
```

2.3 Synthèse

Dans ce chapitre, nous avons introduit la structure d'arène : une structure de données paresseuse permettant de représenter un ensemble (potentiellement) infini d'arbres. Nous avons également défini un certains nombre d'opérations applicables aux éléments de cette structure. Cette structure de données est utilisée dans le chapitre suivant pour capturer le résultat des opérations d'expansion d'une vue partielle et de fusion d'un ensemble de vues partielles.

Chapitre 3

Fusion de vues partielles d'un document structuré

Sommaire

3.1 Projection inverse : algorithme d'expansion **60**
3.2 Cohérence d'un ensemble de vues partielles **66**
3.3 Fusion de vues partielles d'un document structuré . . . **70**
3.4 Synthèse **84**

Rappelons que fusionner deux réplicats partiels u_1 et u_2, associés à des vues $\mathcal{V}_1$ et $\mathcal{V}_2$, revient à trouver un document t tel que :

$$(\exists t_1 \leq t \quad \pi_{\mathcal{V}_1}(t_1) = u_1) \ \wedge \ (\exists t_2 \leq t \quad \pi_{\mathcal{V}_1}(t_2) = u_2)\,,$$

où l'écriture $\pi_{\mathcal{V}}(t) = u$ signifie que l'arbre u est la projection du document t suivant la vue $\mathcal{V}$. Si $\mathcal{A}_1$ et $\mathcal{A}_2$ sont les automates associés aux vues $\mathcal{V}_1$ et $\mathcal{V}_2$ respectivement, alors un document satisfait la condition précédente si, et seulement si, il est une mise à jour d'un document reconnu par la synchronisation de $\mathcal{A}_1$ et $\mathcal{A}_2$ à partir de l'état initial (u_1, u_2).

Ce chapitre décrit la solution que nous apportons au problème de fusion des mises à jour des vues partielles d'un document structuré. Cette solution est basée sur une opération de synchronisation d'automates d'arbre correspondant aux vues partielles. C'est une adaptation de la solution au problème plus spécifique de la cohérence de vues partielles que nous présentons dans la section 3.2. Cette dernière utilise les automates d'arbre et la structure d'arène présentée au chapitre 2. Nous montrons par la suite (sec. 3.3) comment adapter cette solution pour la résolution du problème de la fusion des vues partielles.

3.1 Projection inverse d'une vue partielle : algorithme d'expansion

L'algorithme d'expansion va consister à associer un automate d'arbre à une vue dont l'état initial est la projection, selon cette vue, du document global. Cet automate doit être conçu de sorte que les documents ayant comme projection la vue partielle, donnée par cet état initial, sont ceux appartenant à l'arène engendrée par cet automate à partir de cet état initial. Les algorithmes présentés au chapitre précédant sur les arènes (chap. 2) permettent alors de décider si de tels documents existent et dans l'affirmative de produire les documents les plus "simples" (les plus représentatifs) de l'ensemble éventuellement infini des solutions.

Nous rappelons que la fonction *gram2treeauto* qui permet d'associer à une grammaire abstraite un automate d'arbre (une co-algèbre) qui reconnaît les arbres de syntaxe abstraite de la grammaire est donnée comme suit :

```
1 gram2treeauto :: (Eq symb) => Gram prod symb -> TreeAuto prod symb
2 gram2treeauto gram symb = [(p,rhs gram p) | p <- prods gram,
3                                             symb == lhs gram p]
```

La fonction d'expansion que nous cherchons à réaliser aura le profil suivant :

```
expansion :: (Eq symb)-> Gram prod symb -> (symb -> Bool) -> symb
                                        ->[Tree symb] -> Arena prod
```

Elle doit être telle que `expansion gram view symb forest` retourne une représentation (sous forme d'une arène) de l'ensemble des arbres de syntaxe abstraite issus du symbole précisé et dont la forêt donnée en argument est la partie visible. Nous souhaitons définir cette fonction sous la forme d'un anamorphisme [1] associé à un automate d'arbre correspondant à la vue donnée :

```
1 expansion gram view axiom ts = ana gramview ...
2                where gramview = gram2treeautoview gram view
3
```

Les états de l'automate réalisant l'expansion sont des couples $\langle X, ts \rangle$ constitués d'un symbole grammatical `X` et d'une liste d'arbres (fig. 3.1). 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

1. La fonction *ana* utilisée dans la fonction *expansion* est celle définie à la page 52.

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 $q_0 = \langle axiom, forest\rangle$.

L'expansion s'écrit donc plus précisémment sous la forme :

```
1  expansion :: (Eq symb) => Gram prod symb -> (symb -> Bool) -> symb
2                            -> [Tree symb] -> Arena prod
3  expansion gram view axiom forest =
4    if view axiom
5       then case forest of
6            [Node a ts0] -> if a==axiom
7                               then ana gramview (axiom, ts0)
8                               else Or[]
9            otherwise    -> Or[]
10      else ana gramview (axiom,forest)
11      where gramview = gram2treeautoview gram view
```

Il nous reste à préciser la définition de la fonction `gram2treeautoview` qui associe un automate d'arbre à la donnée d'une grammaire et d'une vue. Supposons qu'un noeud associé à la production $p : A_0 \to A_1 \cdots A_n$ 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'arbre sont de la forme

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

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'arbre est donc la suivante :

```
1  gram2treeautoview :: (Eq symb) => Gram prod symb -> (symb -> Bool)
2                                 -> TreeAuto prod (symb, [Tree symb])
```

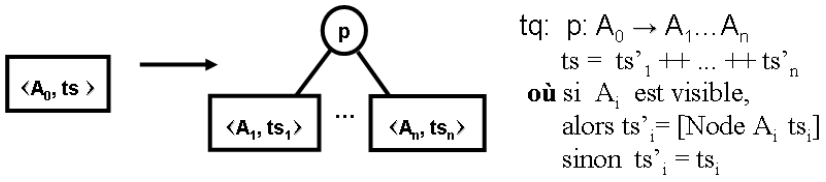

FIGURE 3.1 – Dérivation d'un automate d'arbre associé à une vue

```
gram2treeautoview gram view (symb, ts) =
                  [(p, zip (rhs gram p) tss) | p <- prods gram,
                                  symb == lhs gram p,
                                  tss <- match view (rhs gram p) ts]
```

Notons que la fonction `zip` [2] utilisée dans le code ci-dessus 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'arbre. 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. La fonction `matchone` (utilisée dans la fonction `match`) associée à une vue prend comme premier argument un symbole grammatical `symb` et comme second une liste d'arbres `ts` et, en procédant comme les parseurs définis dans [Fok95], fournit en résultat une liste de couple `(ts1, ts2)` telle que `ts1` est un préfixe de `ts` qui est compatible avec `symb` relativement à la vue prise en argument ; `ts2` est la liste résiduelle.

```
match :: (Eq symb) => (symb -> Bool) -> [symb] -> [Tree symb]
                                                -> [[[Tree symb]]]
match view [] ts = if null ts then [[]] else []
match view (symb:symbs) ts = [(ts1:tss) |
                               (ts1,ts2) <- matchone view symb ts,
                               tss <- match view symbs ts2]
matchone :: (Eq symb) => (symb -> Bool) -> [symb] -> [Tree symb]
                                    -> [([Tree symb],[Tree symb])]
matchone view symb ts = if view symb then
                          if (null ts || ((top (head ts)/=symb))
```

2. Pour plus d'information sur la fonction `zip`, voir annexe A.

```
                                then []
                                else [(succ_ (head ts), tail ts)]
                           else split ts
split :: [a] -> [([a],[a])]
split [] = [([],[])]
split (x:xs) = [([],x:xs)] ++ [(x:xs1,xs2) | (xs1,xs2) <- split xs]
```

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

```
expansion :: (Eq symb) => Gram prod symb -> (symb -> Bool) -> symb
                          -> symb -> [Tree symb] -> Arena prod
expansion gram view axiom forest =
  if view axiom
     then case forest of
          [Node a ts0] -> if a==axiom then g (axiom, ts0) else Or[]
          otherwise    -> Or[]
     else g (axiom,ts)
  where g (symb, ts) = Or [(p, And(map g (zip (rhs gram p) tss)))|
                               p <- prods gram,
                               symb == lhs gram p,
                               tss <- match view (rhs gram p) ts]
```

La fonction réalisant l'appariement d'une liste d'arbres à une liste de symboles grammaticaux procède par essai-erreur en utilisant le cas échéant la fonction `split` [3] qui génère toutes les façons de scinder en deux sous-listes la liste d'arbres qu'elle prend en argument. 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

3. La fonction `split` donne toutes les façons de scinder en deux sous listes une liste qu'elle prend en argument. Ce qui suit est un exemple d'utilisation de cette fonction :
`split [1..3] = [([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])]`

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.

3.1.1 Un exemple

Nous illustrons l'algorithme d'expansion en considérant à nouveau la grammaire G_{run} 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 $\mathcal{V}$ constituée des symboles A et B ($\mathcal{V} = \{A, B\}$). Nous représentons dans cet exemple des listes d'arbres par leurs 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'arbre de la manière suivante.

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

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 la vue partielle obtenue par projection suivant la vue $\mathcal{V} = \{A, B\}$ de l'arbre de dérivation de la figure 1.3 page 17 est $(()[()()])$ (voir fig. 1.6 page 21). 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'arbre fini suivant :

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

La figure 3.2 représente l'élagage (et son nettoyage) de l'arène engendré par l'automate d'arbre 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).

Il n'y a qu'un arbre inscrit dans cette arène (indiqué en traits forts) qui n'est rien d'autre que l'arbre à partir duquel on est parti (arbre de la figure 1.3). 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 une arène. 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 3.3. On pourrait reconstituer toute solution (arbre inscrit dans l'arène de départ) par cette opération d"'autogreffe" 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

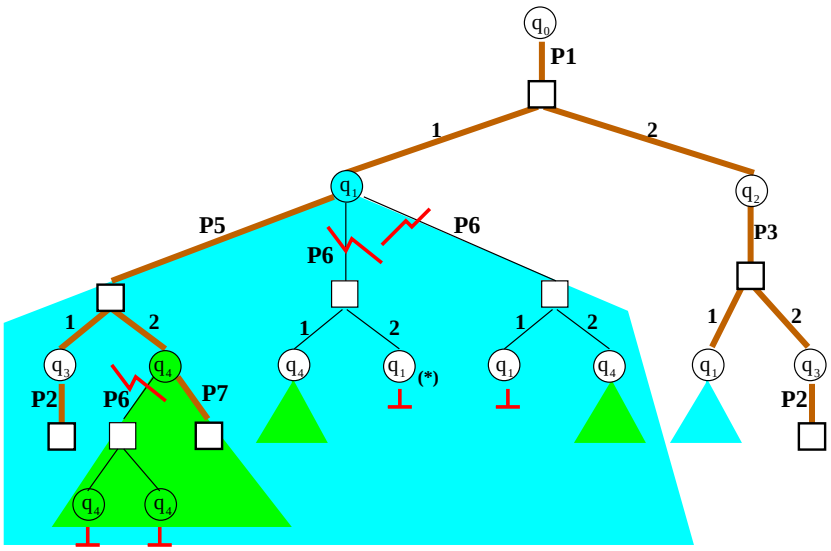

FIGURE 3.2 – Élagage (et son nettoyage) de l'arène engendré par l'automate d'arbre associé à la vue partielle (()[()()])

car nous serons davantage intéressés par la représentation de cet ensemble d'arbres par la structure coinductive d'arène plutôt qu'à une énumération des éléments de cet ensemble.

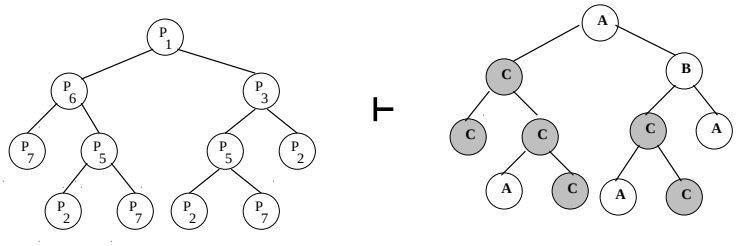

FIGURE 3.3 – Une solution (arbre associé à la vue partielle (()[()()])) obtenue par "auto-greffe" à partir de l'arbre inscrit dans l'élagage de la figure 3.2

3.2 Cohérence d'un ensemble de vues partielles

Le problème de la cohérence de vues consiste à décider s'il existe un document correspondant à un ensemble de vues partielles données et dans l'affirmative à le construire (synchronisation des vues, *c.f.* fig.1.11).

3.2.1 Synchronisation de deux vues

La solution que nous apportons au problème de cohérence de vues est basée sur la synchronisation des modèles utilisés pour réaliser l'expansion : les arènes et les automates d'arbre. Nous introduisons donc deux combinateurs pour le faire : le premier combinateur $\langle\#\rangle$ permet de synchroniser deux arènes, c'est-à-dire de calculer une représentation de l'intersection de leurs ensembles d'arbres de syntaxe abstraite ; le second combinateur $\langle\$\rangle$ permet de synchroniser deux automates d'arbre (co-algèbres) afin de construire un automate reconnaissant les arbres de l'intersection des ensembles d'arbres reconnus par chacun d'eux. Le premier de ces combinateurs est donné par le combinateur suivant :

```
1 infix 4 <#>
2 (<#>) :: (Eq a) => Arena a -> Arena a -> Arena a
3 t1 <#> t2 = Or[(a1, zipWith (<#>) ts1 ts2)|
4                                    (a1, And ts1) <- unOr t1,
5                                    (a2, And ts2) <- unOr t2,
6                                    a1 == a2,
7                                    (length ts1) == (length ts2)]
8
```

Ce combinateur réalise l'intersection de deux arènes ; c'est-à-dire, si on note $t \models u$ pour signifier que t est un arbre apparaissant dans la liste *enumerate* u :

Proposition 3.2.1. $(t \models u \ \wedge \ t \models v) \ \Leftrightarrow \ t \models u \langle\#\rangle v$

Preuve. Rappelons la forme des structures de données que nous avons utilisé pour les arbres et les arènes :

```
data Tree a = Node{top :: a, succ_ :: [Tree a]}
newtype Arena  a = Or  {unOr  :: [(a,Arenas a)]}
newtype Arenas a = And {unAnd :: [Arena a]}
```

Définissons une fonction *unNode* permettant de décharger les termes de *Tree* a du constructeur *Node* :

```
unNode :: Tree a -> (a, Tree a)}
unNode Node a ts = (a, ts)}
```

La notation $t \models u$ revient à dire que $unNode\ t \ = \ (a, [t_1, \cdots, t_n])$ et qu'il existe un élément $(a, \ And[u_1, \cdots, u_n])$ de $unOr\ u$ 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 $(a,\ And[u_1, \cdots, u_n])$ dans $unOr\ u$ et $(a,\ And[v_1, \cdots, v_n])$ dans $unOr\ v$ 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 $(a,\ And[w_1, \cdots, w_n])$ dans $unOr\ u \langle\#\rangle v$ tel que $t \models w$, soit $t \models u \langle\#\rangle v$. □

Le second combinateur $\langle\$\rangle$ de synchronisation sur les automates d'arbre (co-algèbres) est défini de façon similaire. À partir de deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ il produit l'automate $\mathcal{A}_{1,2}$ dont les états sont les paires formées d'un état de $\mathcal{A}_1$ et d'un état de $\mathcal{A}_2$ et dont les règles sont obtenues en synchronisant les règles correspondantes de ces deux automates :

```
(<$>) :: (Eq a) => Coalg a b -> Coalg a c -> Coalg a (b,c)
(coalg1 <$> coalg2)(state1,state2) =
   Or [(a1, And (zip states1 states2)) |
                              (a1,states1) <- coalg1 state1,
                              (a2,states2) <- coalg2 state2,
                              a1 == a2,
                              length states1 == length states2]
```

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 :

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

3.2.2 Algorithme de cohérence

L'algorithme de cohérence de deux vues partielles est obtenu en synchronisant les automates d'arbre associées à chacune des vues partielles à l'aide du combinateur $\langle\$\rangle$.

```
coherence :: (Eq prod,Eq symb) =>
             Gram prod symb -> (symb -> Bool) -> (symb -> Bool)
             -> symb -> [Tree symb] -> [Tree symb] -> Arena prod
coherence gram view1 view2 axiom ts1 ts2 =
  ana (gview1 <$> gview2)((axiom,ts1),(axiom,ts2))
                         where gview1 = gram2coalgview gram view1
                               gview2 = gram2coalgview gram view2
```

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

L'algorithme de cohérence peut aussi être obtenu en utilisant les constructions équivalentes sur les arènes ie. en combinant les expansions selon chacune des vues avec le combinateur $\langle\#\rangle$ pour obtenir la représentation co-inductive de l'intersection des ensembles d'arbres correspondants.

```
1 coherence :: (Eq prod, Eq symb) ->  Gram prod symb -> (symb -> Bool)
2                              -> (symb -> Bool) -> symb -> [ Tree symb]
3                              -> [Tree symb] -> Arena prod
4 coherence gram view1 view2  axiom ts1 ts2 = arena1 <#> arena2
5         where arena1 = expansion gram view1 axiom ts1
6               arena2 = expansion gram view2 axiom ts2
```

Exemple : Si on reprend l'exemple de de la grammaire `gram` définie section 1.2 (page 15) , et on considère les deux vues partielles de la figure 1.6 (page 21) données par :
`derAB = Node A [Node A [], Node B [Node A [], Node A []]]` ,
`derAC = Node A [Node C [Node A [], Node C []], Node C [Node A [], Node C []], Node A []]`
ayant pour linéarisations respectives *(()[()()])* et *({(){}}{(){}}())* dans lesquelles on a utilisé "(" et ")" pour coder les symboles de Dyck *Open A* et *Close A*, "[" et "]" pour représenter les symboles de Dyck *Open B* et *Close B*, "{" et "}" pour représenter les symboles de Dyck *Open C* et *Close C*. En se restreignant aux états productifs et accessibles à partir de l'état initial, on a les transitions suivantes pour l'automate issu de la synchronisation des

différents automates associés aux différentes vues :

$$
\begin{array}{lcll}
q''_0 & \longrightarrow & (P_1, [\,q''_1, q''_2]) & \text{avec} \quad q''_0 = \langle A, ()[()()]\rangle\langle A, \{()\{\}\}\{()\{\}\}()\rangle \\
 & & & \qquad\quad q''_1 = \langle C, ()\rangle\langle C, ()\{\}\rangle \\
 & & & \qquad\quad q''_2 = \langle B, ()()\rangle\langle B, \{()\{\}\}()\rangle \\
q''_1 & \longrightarrow & (P_5, [\,q''_3, q''_4]) & \text{avec} \quad q''_3 = \langle A, \rangle\langle A, \rangle \\
 & & & \qquad\quad q''_4 = \langle C, \rangle\langle C, \rangle \\
q''_2 & \longrightarrow & (P_3, [\,q''_1, q''_3]) & \\
q''_3 & \longrightarrow & (P_2, [\,]) & \\
q''_4 & \longrightarrow & (P_7, [\,]) &
\end{array}
$$

Remarque : Dans la section 3.1, nous avons indiqué que les états de l'automate $\Delta_{\mathcal{V}}$ associé à la vue $\mathcal{V}$ sont des couples $\langle X, ts\rangle$ tels que X est un symbole grammatical et ts est une liste d'arbres. On peut ainsi considérer que $\Delta_{\mathcal{V}}$ est issu du produit synchrone de deux automates : $\Delta_{\mathcal{V}} = \Delta_G \langle\$\rangle \Delta'_{\mathcal{V}}$ définis comme suit :

- Δ_G est l'automate (la co-algèbre) dérivée de la grammaire G suivant le processus de dérivation décrit à la page 61 (fonction *gram2treeauto*) ;
- $\Delta'_{\mathcal{V}}$ est l'automate d'arbre ayant les productions de la forme : $ts \rightarrow (p,\ [ts_1, \ldots, ts_n])$ avec $p : A_0 \rightarrow A_1 \ldots A_n$ une production de G telle que si A_i est visible, alors $ts'_i = [Node\ A_i\ ts_i]$ sinon $ts'_i = ts_i$ avec $ts = ts'_1 + + \ldots + + ts'_n$.

Il en découle que l'automate $\Delta_{\mathcal{V}_{12}} = \Delta_{\mathcal{V}_1} \langle\$\rangle \Delta_{\mathcal{V}_2}$ réalisant l'expansion cohérente des vues partielles peut être présenté comme un produit synchrone de trois automates :

$$
\begin{array}{rcl}
\Delta_{\mathcal{V}_{12}} & = & \Delta_{\mathcal{V}_1} \langle\$\rangle \Delta_{\mathcal{V}_2} \\
 & = & (\Delta_G \langle\$\rangle \Delta'_{\mathcal{V}_1}) \langle\$\rangle (\Delta_G \langle\$\rangle \Delta'_{\mathcal{V}_2}) \\
 & \approx & \Delta_G \langle\$\rangle \Delta'_{\mathcal{V}_1} \langle\$\rangle \Delta'_{\mathcal{V}_2}
\end{array}
$$

3.3 Fusion de vues partielles d'un document structuré

Nous avons vu que le problème de cohérence de vues était une instance du problème plus général de fusion de mises à jour locales sur des réplicats partiels d'un document. Nous allons maintenant présenter une solution à ce problème plus général en adaptant l'algorithme d'expansion et l'opérateur de synchronisation des automates d'arbre qui nous ont servi à décrire la solution du problème de la cohérence de vues. La généralisation consiste à considérer maintenant des documents comme des arbres pouvant contenir

des bourgeons, c'est à dire des feuilles représentant les endroits où le document pourra être developpé suite à des actions d'édition. Notre première tâche consiste donc à préciser cette notion de document contenant des noeuds ouverts :

Définition 3.3.1. ***Un document*** *associé à la grammaire* $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ *est un arbre de syntaxe abstraite pour la grammaire étendue* $\mathbb{G}_\Omega = (\mathcal{S}, \mathcal{P} \cup \mathcal{S}_\Omega, A)$ *obtenue de* $\mathbb{G}$ *en ajoutant une nouvelle* ε*-production* $X_\Omega : X \to \varepsilon$*, pour chaque symbole grammatical* $X \in \mathcal{S}$*. Un noeud étiqueté* X_Ω *est un* ***noeud ouvert*** *(ou* ***bourgeon****) de sorte* X*. On pose* $t_1 \leq t_2$ *si* t_2 *peut être obtenu de* t_1 *en remplaçant certains bourgeons de* t_1 *par des arbres ayant des sortes correspondantes. Nous disons alors que* t_2 *est une* ***mise à jour*** *de* t_1*.*

Les productions de la grammaire étendue sont donc soit des productions (étiquettes des noeuds clos du document) soit des symboles associés aux symboles grammaticaux (étiquettes des noeuds ouverts du document) de la grammaire de départ. Cette dernière information est redondante car l'étiquette d'un noeud ouvert (d'un arbre de syntaxe abstraite) est l'axiome si ce noeud est la racine, sinon elle est caractérisée par la production qui étiquette son père. Nous pouvons ainsi représenter un ensemble d'arbres de syntaxe abstraite par une arène (avec noeuds ouverts) dont la structure est donnée comme suit :

```
1 data ARENA a  = OR{open:: Bool, unOR::[ARENAS a]}
2 data ARENAS a = AND{label ::a, unAND :: [ARENA a]}
```

Le booléen `open arena` indique si le noeud correspondant (noeud "ou") est ouvert. Les éléments d'une arène sont des arbres de la forme suivante :

```
1 data TREE a = BUD | NODE{node_label::a, unNODE::[TREE a]}
```

L'appartenance d'un arbre à une arène est alors définie par la fonction suivante :

```
1 isMember :: (Eq a) => TREE a -> ARENA a -> Bool
2 isMember BUD arena = open arena
3 isMember (NODE a ts) arena =
4      or [and (zipWith isMember ts arenas)|
5                AND elet arenas <- unOR arena,
6                elet == a]
```

Si une arène a une extension finie on pourra énumérer ses éléments

```
enumerate :: ARENA a -> [TREE a]
enumerate arena = [NODE elet ts |
                           AND elet arenas <- unOR arena,
                           ts <- dist (map enumerate arenas)]
                  ++ [BUD | open arena]
```

et tester sa vacuité :

```
isEmpty :: ARENA a -> Bool
isEmpty arena = and [or (map isEmpty arenas)
                       | AND label arenas <- unOR arena]
                && not (open arena)
```

L'ensemble des arènes est défini comme point fixe d'un foncteur dont les coalgèbres nous fournissent une notion d'automates d'arbre :

```
data Automata a b = Auto{exit:: b-> Bool, next :: b -> [(a,[b])]}
```

L'anamorphisme associé à ce foncteur permet d'engendrer à partir d'un automate et d'un état initial l'arène qui représente l'ensemble des arbres reconnus par cet automate à partir de l'état initial indiqué :

```
ana :: Automata a b -> b -> ARENA a
ana auto gen = OR (exit auto gen)
                  [AND a (map (ana auto) gens)
                    | (a, gens) <- next auto gen]
```

Comme précédemment nous pouvons utiliser un algorithme de réduction sur les arènes engendrées par des automates d'arbre (s'ils ont un nombre fini d'états) pour décider de leur vacuité et engendrer la liste de leurs éléments les plus représentatifs (éléments de l'arène réduite) :

```
void :: (Eq b) => Automata 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)
```

```
6                                  | (_, states) <- next auto state]
7                           && not (exit auto state)
8
9  enum :: (Eq b) => Automata a b -> b -> [TREE a]
10 enum auto init = enum_ [ ] init
11   where enum_ path state =
12           if elem state path then [BUD | exit auto state]
13           else [BUD | exit auto state]++
14                [NODE elet list_tree |
15                 (elet, states) <- next auto state,
16                 list_tree <- dist (map (enum_ (state : path)) states)]
```

Dans ce qui précède nous avons reconsidéré la notion d'arène, celles, qui en sont relatives, d'arbres et d'automates d'arbre ainsi que les fonctions de bases associées, que nous avions introduites (et largement commentées) lors de la résolution du problème de cohérence de vues, afin de les adapter pour pouvoir prendre en compte des arbres contenant des bourgeons. Comme on le constate, ces adaptations sont mineures. Il y a maintenant une petite difficulté qui surgit lorsqu'on doit considérer les arbres de dérivation correspondant. Cette difficulté provient de l'observation suivante. Les noeuds des arbres de syntaxe abstraite pour la grammaire étendue sont étiquetés soit par des productions soit par des symboles associés aux symboles grammaticaux de la grammaire de départ. Observons que, suite à l'adjonction des productions supplémentaires $X_\Omega : X \to \varepsilon$ (associées aux symboles grammaticaux $X \in \mathcal{S}$), la grammaire étendue ne vérifie plus l'hypothèse selon laquelle une production est caractérisée par ses parties gauche et droite. On a donc plus coïncidence entre les arbres de syntaxe abstraite et les arbres de dérivation puisque dans ces derniers nous ne saurons pas distinguer, pour une feuille étiquetée par un symbole grammatical A, s'il s'agit d'un bourgeon ou d'un noeud fermé associé à une production constante $p : A \to \varepsilon$ de la grammaire de départ. Pour l'utilisateur cependant la distinction entre un élément constant et un bourgeon (où il peut opérer des actions d'édition) est flagrante. Nous allons par conséquent considérer une classe étendue d'arbres de dérivation dans lesquels nous préservons l'information suivant laquelle certaines feuilles sont des bourgeons. Il s'agit d'arbres sur l'alphabet étendu $\mathcal{S} \cup \overline{\mathcal{S}}$ où un noeud étiqueté par un symbole $\overline{X} \in \overline{\mathcal{S}}$ représente un noeud ouvert de sorte $X \in \mathcal{S}$ (les symboles appartenant à l'ensemble $\overline{\mathcal{S}}$ apparaissent seulement au niveau des feuilles). De cette façon, on rétablit une correspondance bijective entre les documents (ie. des arbres de syntaxe abstraite pour $\mathbb{G}_\Omega$) et les arbres de dérivation (sur l'alphabet étendu $\mathcal{S} \cup \overline{\mathcal{S}}$). Nous représentons en Haskell de tels arbres de la manière suivante (`BTree` pour *budding trees*, arbres avec

bourgeons) :

```
data BTree a = BTree{value::a, sons::Maybe [BTree a]}
```

Un tel arbre est soit un bougeon étiqueté par un élément a, lorsqu'il est de la forme

```
bud :: a -> BTree a
bud a = BTree a Nothing
```

sinon il est de la forme `BTree a (Just ts)`, expression qui représente un arbre dont la racine est étiquetée a et dont la liste des fils est donnée par ts. La projection d'un arbre de syntaxe abstraite sur un sous-ensemble $\mathcal{V} \subset \mathcal{S}$ de symboles grammaticaux est définie comme précédemment en considérant qu'un symbole $\overline{X}$ est visible si, et seulement si, X est visible.

```
projection :: Gram prod symb -> (symb -> Bool) -> symb ->
              TREE prod -> [BTree symb]
projection gram  view   = proj where
  proj symb BUD = [bud symb]
  proj symb (NODE prod ts) =
       if view left then [BTree left  (Just sons)] else sons
              where left  = lhs gram prod
                    right = rhs gram prod
                    sons = concat (zipWith ($)(map proj right) ts)
```

Si tous les symboles sont visibles, la projection permet de convertir un arbre de syntaxe abstraite en son arbre de dérivation (avec bourgeons) équivalent. La figure 3.4 donne un exemple d'arbre de syntaxe abstraite ouvert (fig. 3.4`(a)`), un exemple d'arbre de dérivation (fig. 3.4`(b)`) et un exemple de projection effectuée sur ces arbres (fig. 3.4`(c)`).

L'automate d'arbre (la co-algèbre) qui définit l'expansion d'une vue partielle est défini comme précédemment (sec. 3.1) sauf que nous ajoutons une clause correspondant aux productions additionnelles $X_\Omega : X \to \varepsilon$

```
data Tag x = Close x | Open x

gram2autoview :: (Eq symb) => Gram prod symb -> (symb -> Bool)->
```

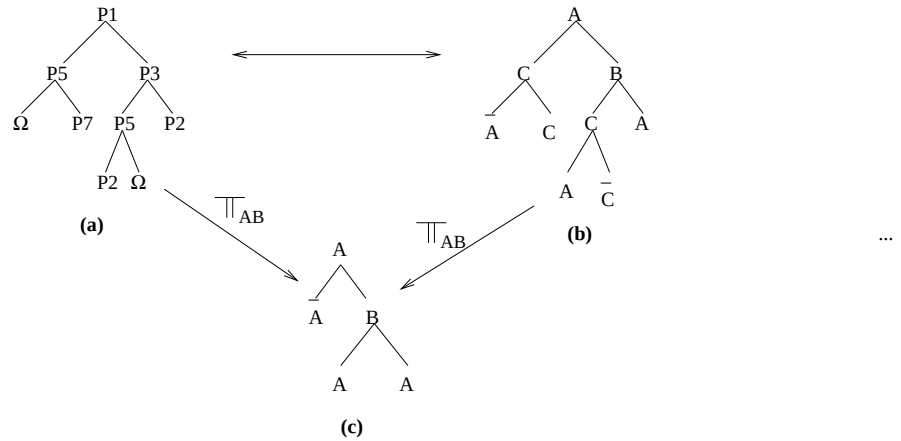

FIGURE 3.4 – Exemple d'arbre de syntaxe abstraite ouvert (a), d'arbre de dérivation correspondante (b) et une projection de ces arbres suivant $\{A,B\}$ (c).

```
4                                   Automata prod (Tag symb, [BTree symb])
5  gram2autoview gram view  = Auto exit next
6    where -- exit :: (Tag symb,[BTree symb]) -> Bool
7          exit (Open symb,[]) = True
8          exit _ = False
9          -- next :: (Tag symb,[BTree symb])
10         --           -> [(prod,[(Tag symb,[BTree symb])])]
11         next (Open symb,_) = []
12         next (Close symb,ts) =
13               [(p, zipWith trans (rhs gram p) tss) |
14                         p <- prods gram,
15                         symb == lhs gram p,
16                         tss <- match view (rhs gram p) ts]
17
18 trans symb (Close ts) = (Close symb, ts)
19 trans symb (Open  ts) = (Open  symb, ts)
20
21 match :: (Eq symb)=> (symb -> Bool) -> [symb] -> [BTree symb]
22                      -> [[Tag [BTree symb]]]
23 match view [] ts = if null ts then [[]] else []
24 match view (symb:symbs) ts =
25    [(ts1:tss)| (ts1,ts2) <- matchone view symb ts,
26                tss <- match view symbs ts2]
27
```

```
matchone :: (Eq symb) => (symb -> Bool) -> symb -> [BTree symb] ->
                         [(Tag [BTree symb],[BTree symb])]
matchone view symb ts =
  if view symb
  then if null ts then []
       else case (head ts) of
            (BTree symb' Nothing) -> if symb==symb'
                                     then [(Open [], tail ts)]
                                     else []
            (BTree symb' (Just ts')) -> if symb==symb'
                                        then [(Close ts', tail ts)]
                                        else []
  else [(Open [], ts)]++
       [(Close (take n ts),drop n ts) | n <- [1..(length ts)]]

```

Nous devons adapter en conséquence l'algorithme d'expansion pour qu'il utilise cette nouvelle traduction d'une grammaire avec vue en automate d'arbre. En utilisant le schéma de récursion associé à l'anamorphisme on obtient la définition récursive donnée ci-dessous. Pour améliorer la lisibilité du code nous introduisons les deux abréviations suivantes

```
bottom = OR False []
omega  = OR True []
```

qui représentent respectivement une arène dont l'extension est vide (`bottom`) et une arène dont l'unique élément est un arbre réduit à un bourgeon (`omega`).

```
expansion :: (Eq symb) => Gram prod symb -> (symb -> Bool) ->
                          symb -> [BTree symb] -> ARENA prod

expansion gram view axiom ts =
  if view axiom
     then case ts of
          [BTree a (Just ts0)] -> if a==axiom
                                  then g (Close axiom, ts0)
                                  else bottom
          [BTree a Nothing]    -> if a==axiom
                                  then omega
                                  else bottom
```

```
13          otherwise            -> bottom
14      else if null ts then omega
15                       else g (Close axiom, ts)
16      -- g = ana (gram2autoview gram view)
17     where g (Close symb,ts) =
18             OR False [AND p (map g (zipWith trans (rhs gram p) tss))|
19                         p <- prods gram,
20                         symb == lhs gram p,
21                         tss <- match view (rhs gram p) ts ]
22          g (Open symb,_) = omega
```

Les justifications de cet algorithme sont les mêmes que celles que nous avions données dans la section 3.1 à propos de l'algorithme d'expansion dont celui-ci est l'adaptation. Nous ne reprenons pas ici ces arguments. Une remarque est néanmoins importante. Dans la procédure `match` servant à filtrer la trace visible selon la partie droite d'une production, nous avons fait le choix (voir `matchone`, ligne 40) de produire un bourgeon pour un symbole invisible associé à une trace vide. Néanmoins, si on remplace ce bourgeon par un arbre de syntaxe abstraite, issu de ce symbole, ayant une trace vide (c'est-à-dire ne contenant aucun symbole visible) nous obtiendrons également un arbre de syntaxe abstraite ayant la même trace visible. On en déduit que notre algorithme produit exactement les arbres de syntaxe abstraite *minimum* ayant la trace donnée. En effet, l'algorithme d'expansion n'exclut que des arbres t qui sont obtenus en remplaçant certains bourgeons par des arbres de trace visible vide à partir d'un arbre t' effectivement produit, on a donc $t' \leq t$ et nous sommes ainsi assuré que notre algorithme produit au moins tous les arbres minimum (pour la relation $\leq$). Peut il en produire plus ? supposons donc que t soit un arbre non minimal ayant la trace visible considérée. Puisque notre algorithme produit tous les arbres minimum ayant la trace en question, il produit un t' minimum dont t est une mise à jour. Ainsi t est obtenu à partir de t' en remplaçant certains de ses bourgeons par des arbres, mais comme t et t' ont la même trace visible tous ces arbres ont nécessairement une trace vide. Or par construction les seuls sous arbres d'un arbre produit par l'algorithme d'expansion qui ont une trace vide sont des arbres réduits à des bourgeons ; donc t doit être égal à t' ce qui contredit le fait qu'il soit non minimal. On a ainsi établi que

Proposition 3.3.2. *Les éléments de l'arène produit par l'algorithme d'expansion à partir d'une trace visible sont exactement les éléments minimum (pour la relation de mise à jour) de l'ensemble des arbres de syntaxe abstraite ayant cette trace visible.*

Pour résoudre le problème de cohérence de vues (section 3.2) nous devions calculer l'intersection des images inverses des différentes projections ; l'algorithme d'expansion fournissait le calcul de l'image inverse de chaque projection et l'opérateur de synchronisation en réalisait l'intersection. Ici nous recherchons les plus petits arbres de syntaxe abstraite dont les projections respectives sont des mises à jour des différents réplicats. De cette manière nous réalisons la fusion des différentes mises-à-jour, à partir des versions locales d'un même document initial, qui ont conduit sur les sites respectifs à ces différents réplicats.

On observe, par ailleurs, qu'une arène produite par cet algorithme d'expansion a une forme particulière : un noeud ouvert n'a aucun successeur. De façon similaire, tous les états de sortie (c'est-à-dire pour lequel le booléen `exit` est à vrai) d'un automate produit par la fonction `gram2autoview` sont terminaux, en ce sens qu'aucune transition n'y est applicable (la valeur de la fonction `next` en un tel état est la liste vide). Une arène ou un automate d'arbre vérifiant cette propriété seront dits *à bourgeons terminaux*.

Proposition 3.3.3. *L'automate d'arbre produit à partir d'une grammaire et d'une vue par la fonction* `gram2autoview` *est déterministe.*

Preuve. Cela signifie que si t est un arbre de syntaxe abstraite reconnu par cet automate, il existe une seule exécution dans cet automate qui conduise à son acceptation. L'état initial d'une telle exécution est donnée par la trace visible de t, donc à ce stade on a pas de choix. La source potentielle de non déterministe provient de la fonction `match` qui choisit un appariement d'une trace visible et d'un tag pour chaque symbole en partie droite de la production. Il faut donc vérifier que deux appariements distincts ne peuvent jamais conduire à la reconnaissance d'un même arbre. Soit t' un sous-arbre de t. Quelles sont les transitions franchissables pour t' en un état q qui étiquette la racine de ce sous arbre pour une exécution de l'automate reconnaissant t ? Soit q est de la forme $(Open\ A_0, [])$ auquel cas aucune transition n'est franchissable et t' est nécessairement un bourgeon. Sinon q est de la forme $q = (Close\ A_0, w)$ et t' doit alors être de la forme $t' = p(t_1, \ldots, t_n)$ où p est une production $p : A_0 \to A_1 \ldots A_n$ de la grammaire. Les transitions franchissables sont de la forme $(p, ((s_1, w_1), \ldots, (s_n, w_n)))$ où les s_i sont des symboles avec tags et les w_i des traces visibles. On doit avoir $w = w'_1 +\!+ \ldots +\!+ w'_n$ tel que, si A_i est visible alors soit $w'_i = [BTree\ A_i(Just\ w_i)]$ et $s_i =$`Close` A_i ou bien $w'_i = [BTree\ A_i\ Nothing]$ et $s_i =$`Open` A_i. Si A_i est invisible $w'_i = w_i$ et si cette trace est vide $s_i =$`Open` A_i, sinon $s_i =$`Close` A_i. Mais les w'_i doivent être les traces visibles des sous termes t_i ; ce qui les caractérise . Il en est donc aussi de même des w_i et des s_i ce qui montre qu'une seule transition de l'automate peut s'appliquer. ■

Evidemment, l'automate d'arbre n'est pas utilisé comme un *reconnaisseur* mais comme un *générateur*. Dans l'état courant du processus de génération on fait un certain nombre de choix correspondant aux différents appariements possibles d'un tag et d'une trace visible pour chaque symbole grammatical de la partie droite de la production courante. La propriété de déterminisme exprime le fait que deux choix différents ne peuvent conduire à la génération d'un même arbre de syntaxe abstraite. Notons cependant que certains choix peuvent conduire à des impasses qui ne seront détectées que plus tardivement ; plutôt que de chercher à faire des retours arrières (*backtracking*) on effectuera *a posteriori* l'élagage de l'arène produite.

Nous définissons maintenant une opération de synchronisation sur des automates. L'objectif est le suivant : si les différents automates ainsi synchronisés sont des automates d'arbre déterministes et à bourgeons terminaux, l'automate résultant de leur synchronisation le sera également et il reconnaîtra les arbres qui sont des mises à jour d'arbres reconnus par chacun de ces automates

$$t \models \otimes_i \mathcal{A}^{(i)} \Rightarrow (\forall i)(\exists t_i) \quad t_i \models \mathcal{A}^{(i)} \text{ et } t_i \leq t$$

et qui sont minimaux vis-à-vis de cette propriété (c'est-à-dire que si t' est tel que $t' \leq t$ et $t' \neq t$ alors t' ne vérifie pas la propriété ci-dessus).

Les automates étant à bourgeons terminaux, on sait que si `exit q` alors `next q =[]`. Si q n'est pas un état de sortie nous noterons $q \xrightarrow{a} (q_1, \ldots, q_n)$ pour chaque élément $(a, [q_1, \ldots, q_n])$ apparaissant dans la liste `next q`. Nous ferons l'hypothèse que les automates ainsi composés vérifient la propriété suivante : si $q^{(i)} \xrightarrow{a} (q_1^{(i)}, \ldots, q_n^{(i)})$ dans $\mathcal{A}^{(i)}$ et $q^{(j)} \xrightarrow{a} (q_1^{(j)}, \ldots, q_m^{(j)})$ dans $\mathcal{A}^{(j)}$ alors $n = m$. Ceci sera vérifié dans notre cas puisque les étiquettes a des transitions correspondent aux productions de la grammaire et le nombre d'états en partie droite de ces transitions coïncide avec le nombre de symboles en partie droite de la production correspondante. L'automate synchronisé $\mathcal{A} = \otimes_{i=1}^{k} \mathcal{A}^{(i)}$ est alors défini comme suit. Ses états sont les vecteurs d'états : $Q = Q^{(1)} \times \cdots \times Q^{(k)}$; $exit\ (q^{(1)}, \ldots, q^{(k)})$ si, et seulement si, $exit\ q^{(i)}$ pour tout $1 \leq i \leq k$; et ses transitions sont données comme suit :

- si `exit q` alors `next q =[]`, sinon
- $(q^{(1)}, \ldots, q^{(k)}) \xrightarrow{a} \left((q_1^{(1)}, \ldots, q_1^{(k)}), \ldots, (q_n^{(1)}, \ldots, q_n^{(k)})\right)$ si, et seulement si pour tout $1 \leq i \leq k$ ou bien
  - $exit\ q^{(i)}$ et $q_j^{(i)} = q^{(i)}$ pour tout $1 \leq j \leq n$, sinon
  - $q^{(i)} \xrightarrow{a} (q_1^{(i)}, \ldots, q_n^{(i)})$

Il s'agit donc d'une relaxation de la synchronisation d'automates que nous avions introduite dans la section 3.1 dans laquelle un automate qui atteint un état de sortie ne contribue plus au comportement mais ne s'oppose pas à la synchronisation des autres automates (il devient dormant). L'automate

synchronisé atteint un état de sortie lorsque toutes ses composantes sont endormies. Par définition cet automate est à bourgeons terminaux et il est déterministe si toutes ses composantes le sont.

Proposition 3.3.4. *Avec les hypothèses posées sur les automates $\mathcal{A}^{(i)}$, leur synchronisation $\mathcal{A} = \otimes_{i=1}^{k} \mathcal{A}^{(i)}$ reconnaît les éléments minimaux de l'ensemble des arbres qui sont des raffinement d'arbres reconnus par chacun des automates $\mathcal{A}^{(i)}$:*

$$t \models \otimes_{i=1}^{k} \mathcal{A}^{(i)} \quad \Leftrightarrow \quad \left\{ \begin{array}{lll} i) & \forall i \quad \exists t_i & t_i \models \mathcal{A}^{(i)} \text{ et } t_i \leq t \\ ii) & \big(\forall i \quad \exists t'_i & t'_i \models \mathcal{A}^{(i)} \text{ et } t'_i \leq t' \leq t\big) \Rightarrow t' = t \end{array} \right.$$

Preuve. Un arbre t est reconnu par l'automate synchronisé si, et seulement si, on peut étiqueter chacun des noeuds de t par un état de l'automate conformément à ce qui est spécifié par les transitions de l'automate, c'est-à-dire que si un noeud associé à un symbole a est étiqueté par l'état q et s'il admet n successeurs étiquetés respectivement par $q_1, \ldots, q_n$, alors $q \xrightarrow{a} (q_1, \ldots, q_n)$ doit être une transition de l'automate. Comme l'automate est déterministe cet étiquetage est par ailleurs unique (y compris l'état initial attaché à la racine de l'arbre). On regarde la $i^{\text{ème}}$ composante de cet étiquetage. Sur chacune des branches, ainsi étiquetée par des états de $\mathcal{A}^i$, on coupe dès qu'on atteint un état de sortie. On obtient ainsi un étiquetage d'un arbre $t_i \leq t$ selon $\mathcal{A}^{(i)}$. Ainsi,

$$t \models \otimes_{i=1}^{k} \mathcal{A}^{(i)} \quad \Rightarrow \quad \forall i \; \exists t_i \quad t_i \models \mathcal{A}^{(i)} \text{ et } t_i \leq t$$

Puisqu'un état de $\mathcal{A}$ est de sortie ssi chacune de ses composantes l'est (dans les $\mathcal{A}^i$), on en déduit que dans un tel cas $t = \bigvee_{i=1}^{k} t_i$. Inversement supposons $t_i \models \mathcal{A}^{(i)}$, par définition de l'automate synchronisé, on a $\bigvee_{i=1}^{k} t_i \models \otimes_{i=1}^{k} \mathcal{A}^{(i)}$. Et donc globalement

$$L\left(\otimes_{i=1}^{k} \mathcal{A}^{(i)}\right) = \left\{ \bigvee_{i=1}^{k} t_i | \quad t_i \models \mathcal{A}^{(i)} \quad \{t_i\}_{i=1}^{k} \text{ est une famille cohérente d'arbres} \right\}$$

où une famille d'arbres $\{t_i\}_{i \in I}$ est cohérente si elle admet un majorant

$$\exists t \quad \forall i \qquad t_i \leq t$$

une telle famille admet alors un plus petit majorant ($\bigvee_{i=1}^{k} t_i$). Supposons $t \models \otimes_{i=1}^{k} \mathcal{A}^{(i)}$ et $t' \leq t$ tel que

$$\forall i \quad \exists t'_i \qquad t'_i \models \mathcal{A}^{(i)} \text{ et } t'_i \leq t'$$

Puisque l'automate est déterministe on en déduit que pour tout arbre t, reconnu par $\mathcal{A} = \otimes_{i=1}^{k}\mathcal{A}^{(i)}$, un arbre t_i vérifiant simultanément $t_i \models \mathcal{A}^{(i)}$ et $t_i \leq t$ est unique. Ainsi $t = \bigvee_{i=1}^{k} t'_i$ et qui entraîne $t \leq t'$ et par voie de conséquence $t = t'$. A ce stade nous avons prouvé que la condition $i) \wedge ii)$ de la proposition est *nécessaire* pour établir que $t \models \otimes_{i=1}^{k}\mathcal{A}^{(i)}$. Montrons maintenant que cette condition est *suffisante*. Nous savons déjà que si $\forall i \quad \exists t_i \qquad t_i \models \mathcal{A}^{(i)}$ et $t_i \leq t$ alors $t' = \bigvee_{i=1}^{k} t_i \models \otimes_{i=1}^{k}\mathcal{A}^{(i)}$ et par la condition ii) on en déduit que $t = \bigvee_{i=1}^{k} t_i$ et donc $t \models \otimes_{i=1}^{k}\mathcal{A}^{(i)}$. ■

Au cours de la preuve précédente, nous avons rencontré une autre caractérisation du langage du produit synchronisé :

Remarque 3.3.5. *$\mathcal{A} = \otimes_{i=1}^{k}\mathcal{A}^{(i)}$ reconnaît les bornes supérieures des familles cohérentes obtenues en choisissant un arbre reconnu par chacun des $\mathcal{A}^{(i)}$:*

$$L\left(\otimes_{i=1}^{k}\mathcal{A}^{(i)}\right) = \left\{ \bigvee_{i=1}^{k} t_i \mid \quad t_i \models \mathcal{A}^{(i)} \quad \{t_i\}_{i=1}^{k} \textit{ est une famille cohérente d'arbres} \right\}$$

La synchronisation de deux automates peut être codée comme suit en Haskell.

```
1  infix 4 <**>
2  (<**>) :: (Eq a) => Automata a b1 -> Automata a b2 ->
3                      Automata a (b1,b2)
4  auto1 <**> auto2 = Auto exit_ next_ where
5    exit_ (state1,state2) = (exit auto1 state1)&&(exit auto2 state2)
6    next_ (state1,state2) =
7       case (exit1,exit2) of
8        (False,False) ->  [(a1,zip states1 states2)
9                                | (a1,states1) <- next auto1 state1,
10                               (a2,states2) <- next auto2 state2,
11                               a1==a2,
12                               (length states1)==(length states2)]
13       (False,True)  -> [(a,zip states1 (sleep state2))
14                               | (a,states1) <- next auto1 state1]
15       (True,False)  -> [(a,zip (sleep state1) states2)
16                               | (a,states2) <- next auto2 state2]
17       (True,True)   -> []
18     where exit1 = exit auto1 state1
19           exit2 = exit auto2 state2
20           sleep state = state:(sleep state)
```

L'opération correspondante sur les arènes s'écrit de façon similaire :

```
1  infix 4 <##>
2  (<##>) :: (Eq a) => ARENA a -> ARENA a -> ARENA a
3  arena1 <##> arena2 =
4    case (open1,open2) of
5     (False,False) -> OR False
6                        [AND a1 (zipWith (<##>) arenas1 arenas2)
7                        | AND a1 arenas1 <- unOR arena1,
8                          AND a2 arenas2 <- unOR arena2,
9                          a1==a2,
10                         (length arenas1)==(length arenas2)]
11    (False,True)  -> arena1
12    (True,False)  -> arena2
13    (True,True)   -> omega
14   where open1 = open arena1
15         open2 = open arena2
```

Contrairement au combinateur précédent, qui décrit la synchronisation de deux automates d'arbre, celui-ci est récursif (voir ligne 6 du code) puisqu'il intègre le schéma de récursion associé à l'anamorphisme. Néanmoins, la notation d'état y est abstraite et on peut facilement l'itérer pour obtenir une fonction réalisant la fusion d'une liste non vide d'arènes : [4]

```
1  fusion :: (Eq a) => [ARENA a] -> ARENA a
2  fusion = foldl1 (<##>)
```

Supposons un document, représenté par un arbre de syntaxe abstraite t avec bourgeons, que l'on projette suivant n vues pour obtenir des réplicats partiels $t_1, \ldots, t_n$. Ces différents réplicats sont modifiés localement, ce qui nous

4. *foldl1* est un combinateur Haskell permettant d'itérer un opérateur binaire sur une liste non vide d'arguments en l'associant à gauche, sa définition est donnée comme suit :

$$
\begin{aligned}
&foldl :: (a \to b \to a) \to a \to [b] \to a \\
&foldl\ f\ z\ []\ =\ z \\
&foldl\ f\ z\ (x : xs)\ =\ foldl\ f\ (f\ z\ x)\ xs \\
&foldl1 :: (a \to a \to a) \to [a] \to a \\
&foldl1\ f\ [x]\ =\ x \\
&foldl1\ f\ (x : xs)\ =\ foldl\ f\ x\ xs \\
&foldl1\ f\ []\ =\ \text{error "Prelude.foldl1 : empty list"}
\end{aligned}
$$

conduit à des versions actualisées $t'_1, \ldots, t'_n$. En fusionnant ces nouvelles versions on obtient une représentation, sous forme d'arène, des arbres t' tels que $\forall i \quad t_i \leq \pi_i(t')$. Néanmoins rien ne prouve que $t \leq t'$, c'est-à-dire que t' soit une mise-à-jour du document initial. En effet t peut contenir des sous arbres qui ne seront visibles dans aucune des vues partielles et qui seront par conséquent remplacés par des bourgeons lors de la construction de t' (voir la figure 3.5 pour une illustration). Pour éviter ce problème il faut utiliser le document initial lors de la fusion des mises à jour (analogue de la technique *three-way fusion*). Pour cela il suffit d'imaginer un site fictif pour lequel tous les symboles grammaticaux sont visibles et sur lequel on n'effectue aucune action d'édition. De façon équivalente il suffit de définir une arène dont l'extension se réduit à un arbre donné en argument :

```
1 unit :: AST a -> ARENA a
2 unit tree = ana automatAST tree
3
4 automatAST ::   Automata a (AST a)
5 automatAST = Auto exit next
6  where exit BUD = True
7        exit (NODE _ _) = False
8        next BUD = []
9        next (NODE a ts) = [(a,ts)]
```

et de fusionner cette arène avec celles associées aux différents réplicats partiels (voir la figure 3.6 pour une illustration) :

```
1 merge :: (Eq prod, Eq symb) =>
2          Gram prod symb -> symb -> AST prod -> [(symb -> Bool)]
3          -> [[BTree symb]] -> ARENA prod
4 merge gram axiom t0 views ts  = fusion ((unit t0):arenas) where
5   arenas = zipWith ($) expansions ts
6   expansions = map (\view -> expansion gram view axiom) views
```

Bien que l'opérateur `<##>` soit commutatif, d'un point de vue algorithmique l'ordre de ses arguments n'est pas indifférent. En effet les deux opérandes sont composées à la manière de coroutines, chacune explorant son propre espace de solution. La coroutine à gauche génère un préfixe de sa solution, puis celle de droite utilise ce préfixe pour restreindre son espace de solution et proposer un préfixe de solution compatible ; le contrôle retourne alors à la coroutine de gauche. Nous avons donc tout intérêt à positionner à gauche celle des deux

coroutines qui contraint le plus l'espace de recherche. C'est pour cela que nous avons mis l'arène `unit t0` la plus à gauche (et avons choisi d'associer l'opérateur `<##>` à gauche dans la définition de la fonction `fusion`) ; de cette façon la fonction `merge` commence par reproduire l'arbre de syntaxe abstraite `t0` avant de commencer la fusion des mises à jour proprement dite à partir des bourgeons de cet arbre. Pour cette raison, même si nous avions l'assurance, dans un cas particulier, que la fusion des mises à jour locales est une mise à jour du document de départ nous aurons toujours intérêt à procéder comme indiqué plus haut par une technique de *three-way fusion*.

3.4 Synthèse

Dans ce chapitre, nous avons présenté et donné une solution au problème de fusion des réplicats (partiels) d'un document. Pour ce faire, nous avons associé un automate d'arbre à une vue partielle. Celui-ci engendre l'ensemble des arbres de syntaxe abstraite, minimums dans l'ordre de mise à jour, ayant une projection donnée. Nous avons ensuite défini un opérateur de synchronisation qui permet de fusionner de façon minimale les mises à jours effectuées de façon asynchrone sur différents réplicats partiels d'un même document. En préalable nous avons résolu le cas particulier, lorsque les documents sont complets et donc non éditables, qui revient à résoudre le problème de cohérence de vues : peut on décider si différents documents sont des vues partielles d'un même document global, et dans l'affirmative en donner une construction ? Ces algorithmes (de fusion et d'expansion) sont utilisés dans le chapitre suivant où nous présentons un prototype d'éditeur coopératif.

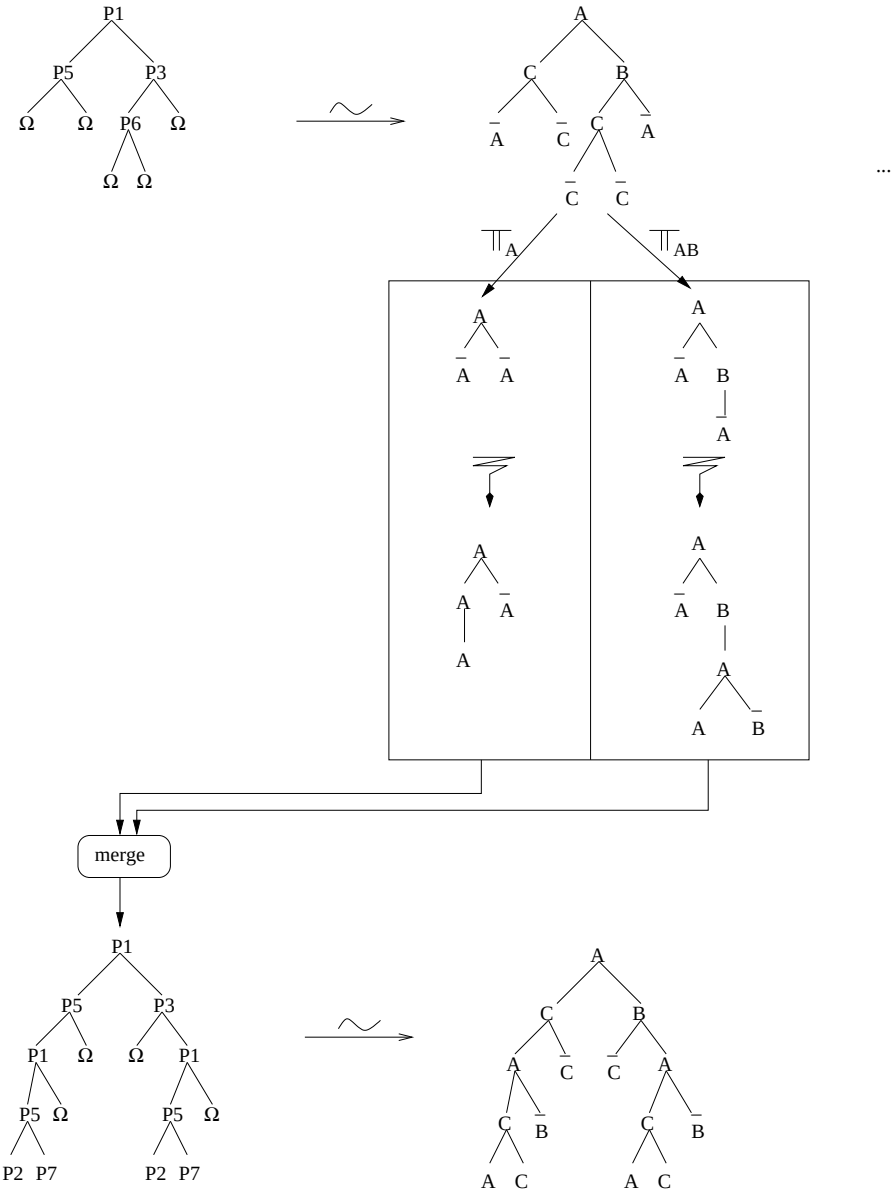

FIGURE 3.5 – *two-way fusion* des mises à jour effectuées sur les projections sur *{A}* et *{A,B}* respectivement d'un document

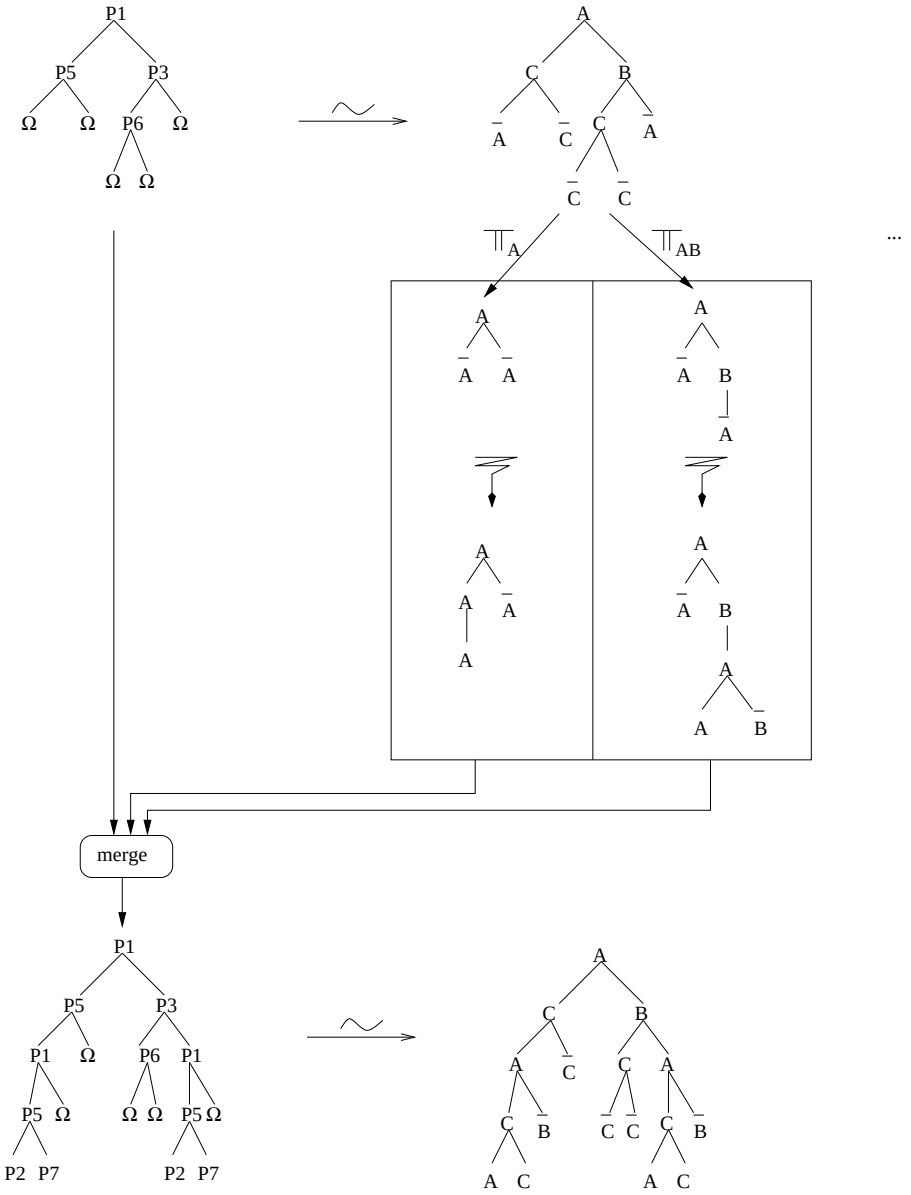

FIGURE 3.6 – *three- way fusion* d'un document et des mises à jour effectuées sur ses projections sur *{A,B}* et *{A,C}* respectivement

Chapitre 4

TinyCE [1] : un prototype d'éditeur coopératif désynchronisé

Sommaire

4.1 Architecture de TinyCE **88**
4.2 Exemple d'utilisation de TinyCE **92**
4.3 Synthèse **95**

Dans ce chapitre, nous présentons *TinyCE*, un prototype d'éditeur coopératif désynchronisé qui implémente non seulement les concepts abordés dans cette thèse mais aussi, offre un cadre d'expérimentation.

TinyCE est un éditeur WYSIWYG [2] permettant l'édition conviviale (édition graphique par simple utilisation de la souris) et coopérative de la structure abstraite (arbre de dérivation) des documents structurés. Il s'utilise en réseau suivant un modèle client-serveur. Son interface utilisateur offre à l'usager des facilitées pour l'édition de la grammaire et des vues qui lui sont associées, l'édition et la validation d'un document global ou d'un réplicat partiel de celui-ci suivant que l'utilisateur est sur le poste serveur ou client. Bien plus, cette interface lui offre aussi des fonctionnalités lui permettant d'expérimenter les concepts de *projection*, d'*expansion* et de *fusion* étudiés dans cette thèse.

Dans la suite de ce chapitre, nous présentons tour à tour l'architecture de *TinyCE*, son interface utilisateur ainsi qu'un exemple d'utilisation.

1. *TinyCE* pour "a Tiny Coopérative Editor" (prononcer "*Tennesse*").
2. What You See Is What You Get

4.1 Architecture de TinyCE

TinyCE est constitué de trois composants principaux : l'interface utilisateur (front-end), le module fonctionnel qui implémente la partie métier de l'application (le back-end) et le module de sauvegarde qui constitue la mémoire de l'éditeur ; c'est elle qui sauvegarde et restaure les documents et leurs modèles (grammaires) manipulés dans l'outil. La figure 4.1 (page 94) donne un aperçu de l'architecture générale de *TinyCE*.

Les différents composants de cette architecture sont les suivants :

- *Les interfaces utilisateur de TinyCE :* on dispose d'une interface utilisateur pour les clients et une autre pour le serveur. L'interface associée au mode serveur (voir figures 4.3, 4.8) permet l'édition de la grammaire (symboles, productions, axiome), des différentes vues, du document (global). Elle permet aussi de préciser les adresses des différents postes clients et de leur expédier une vue de la grammaire et éventuellement un réplicat partiel du document (global) courant.
  L'interface associé au mode client (voir figures 4.4 et 4.5) permet la connexion au site serveur suivie du rapatriement de la grammaire (projetée) ainsi que du réplicat partiel. Elle permet aussi l'édition et la validation de ce réplicat partiel ainsi que sa re-expédition vers le poste serveur après avoir mentionné l'identité du poste serveur dans le champ approprié de l'interface.
- *Le module fonctionnel de TinyCE :* il contient le code (Haskell) des routines permettant de réaliser la projection, l'expansion, la validation, la fusion, ainsi qu'un certain nombre de *codes de services*, essentiellement des *parseurs* utilisés pour le reformatage des données provenant de l'interface utilisateur sous forme de chaîne de caractères. Ce module contient aussi une routine très sollicitée par l'interface graphique pour le calcul des positions relatives des noeuds d'un arbre dans le plan pour réaliser son affichage.
- *Le module de sauvegarde de TinyCE :* il permet la sérialisation/restauration à la XML des grammaires et des documents .

4.1.1 Interconnexion des modules de TinyCE

La communication entre les différents modules de *TinyCE* se fait par appel de procédure. En fait, *TinyCE* est programmé en utilisant deux langages de programmation : Haskell (pour le module fonctionnel) et Java (pour l'interface graphique). L'utilisation de ce couple de langages permet de tirer le meilleur de chacun d'eux. Nous exploitons Java pour la facilité qu'offre ce

langage pour la mise en oeuvre des interfaces graphiques ainsi que pour tirer profit des avantages offerts par les langages à objets (robustesse, modularité, ...). Du langage Haskell nous tirons profit de son grand pouvoir d'abstraction, de son mode d'évaluation (paresseuse), de la modularité, ... Afin de faire coopérer en parfaite synergie ce couple de langages, nous exploitons la possibilité d'appeler un code exécutable à l'intérieur de programmes Java (la même chose est possibble à partir du langage Haskell). Le module fonctionnel de *TinyCE* est donc implémenté comme un exécutable appelé dans des routines appropriées du module interface utilisateur. Nous présentons sommairement ci-dessous une démarche permettant de réaliser une implémentation à partir du couple de langages *Haskell-Java*. C'est cette démarche que nous avons suivie pour la mise en oeuvre de *TinyCE*.

Production d'un exécutable Haskell

L'implémentation GHC [GHC] de Haskell est un compilateur. On peut donc y créer des exécutables et les faire exécuter indépendamment du compilateur qui a servi à leurs générations. Un programme Haskell sous GHC est un fichier d'extension *.hs* semblable à celui ci-dessous enregistré par exemple dans un fichier dénommé *echoGhc.hs*

```
1 module Main where
2 import System.Environment
3 main = do
4            args <- getArgs
5            if (length args) /= 1 then do
6                     putStr("Usage: NomProgramme Argument")
7                     else do putStr("Bonjour "++(head args))
```

Pour compiler un tel programme et créer l'exécutable *helloGhc.exe*, il suffit de saisir la commande

ghc –make -o helloGhc HelloGhc.hs

et pour l'exécution,

helloGhc xxx

où *xxx* est l'argument de l'appel (supposé ici être un nom). En fait, avec GHC, on peut créer des programmes qui utilisent des arguments de la ligne de commandes. Ces arguments sont utilisables dans le programme source par le truchement de la variable *args* (voir ligne *04* dans le listing ci-dessus). La fonction *getArgs* est définie dans le module *System.Environment* et elle a pour type : *getArgs : : IO [Strings]*.

Exécution d'un programme Haskell dans Java

Java permet de lancer un code exécutable (quelconque) à partir d'un code java, et donc, en particulier, du code obtenu à partir d'un programme Haskell. On peut procéder comme suit :

1. Créer l'exécutable Haskell comme présenté ci-dessus,
2. Rassembler les arguments d'appel de l'exécutable Haskell dans une variable (disons *cmd*) de type (Java) *[] String* telle que :
   - *cmd [0]* contient la référence (chemin d'accès) du fichier exécutable Haskell
   - $\forall i > 0$, *cmd [i]* est un paramètre d'appel de cet exécutable.
3. Créer un *process* java pour lancer l'exécutable Haskell,
4. Rediriger convenablement la sortie de l'exécutable Haskell de façon à la récupérer pour traitement dans le programme (Java) courant. Remarquons que, par opposition à ce qui se fait dans les programmes *"C"*, en Java, il y a retour à l'appelant lors de l'exécution d'un *exec.* .

Le listing suivant donne un exemple d'appel du programme Haskell conçu précédemment dans un programme java.

```
1  import java.io.*;
2
3  class HaskellDansJava {
4     //passage par argument de la commande à lancer
5     static String StartCommand(String [] command) {
6       try {
7          //creation du processus
8          Process p = Runtime.getRuntime().exec(command);
9          InputStream in = p.getInputStream();
10           //on récupère le flux de sortie du programme
11         StringBuffer build = new StringBuffer();
12         char c = (char) in.read();
13         while (c != (char) -1) {
14             build.append(c);
15             c = (char) in.read();
16         }
17         String resultat = build.toString();
18         return resultat;
19      }
20      catch (Exception e) {
21         System.out.println("\n" + command + ": commande inconnu ");
```

```
22          return "";
23          }
24      }
25      public static void main(String [] args){
26          String[] cmd = new String[2];
27          cmd[0] = "helloGhc.exe" ;
28          cmd[1] = "Steve Daryl";
29          System.out.println("Appel du programme Haskell");
30          String resultat = StartCommand(cmd);
31          System.out.print("Resultat de l'execution du programme Haskell: ");
32          System.out.println(resultat);
33          }
34  }
```

L'exécution du programme précédent produit la sortie suivante :

```
1  Appel du programme Haskell
2  Resultat de l'execution du programme Haskell: Bonjour Steve Daryl
```

Si on remplace l'instruction de la ligne 28 du listing précédent par celui-ci *cmd[1] = "";* et qu'on le compile et l'exécute à nouveau, on obtient la sortie suivante :

```
1  Appel du programme Haskell
2  Resultat de l'execution du programme Haskell:
3                                          Usage: NomProgramme Argument
```

Nous avons exploité cette faculté d'exécution de programmes Haskell à partir d'un programme Java pour mettre en oeuvre un protocole simple de communication bidirectionnel entre les programmes Haskell et Java basé sur du texte (chaînes de caractères bien formées suivant un codage à la XML). C'est ce protocole que nous utilisons pour faire communiquer le module interface utilisateur programmé en Java avec le module fonctionnel de *TinyCE* programmé en Haskell, moyennant toutefois l'écriture d'un certain nombre de convertisseurs (parseurs) présents dans chaque module. Par exemple, un document (arbre de dérivation) saisi à partir de l'interface utilisateur est converti en une chaîne de caractères avant d'être acheminé vers le module fonctionnel. A la réception de cette chaîne dans le module fonctionnel, elle est convertie en un objet de type (Haskell) correspondant. Le même principe est utilisé pour l'acheminement de la grammaire et des vues.

4.2 Exemple d'utilisation de TinyCE

Pour l'exploitation de *TinyCE* sur un réseau, les actions à effectuer sur le poste serveur et sur les postes clients (deux clients) sont les suivantes :

Sur le poste serveur :

1. *Saisie d'une nouvelle grammaire* ou *sélection d'une grammaire déjà existante* : dans le cas de la saisie [3], on précise le nom de la grammaire, les symboles grammaticaux, les productions. On termine la saisie par une demande d'enregistrement en cliquant sur le bouton *Enregistrer*.
2. *Saisie des vues :* On précise pour chaque poste client quelles sont ses symboles pertinents (sa vue).
3. *Identité des machines :* on précise l'identité du poste serveur (nom et adresse IP) ainsi que celles des différents postes clients.
4. *Début édition et validation du document global :* on peut commencer l'édition du document global sur le poste serveur. À la suite de cette édition on peut procéder à sa validation, c'est à dire vérifier que le document est bien formé vis-à-vis de la grammaire en cliquant sur le bouton "*VALIDATION*".
5. *Projection suivant les différentes vues :* suite à l'édition, on peut obtenir la vue partielle correspondante à chacune des vues en cliquant sur les différents boutons "*PROJECTION suivant vue1*" et "*PROJECTION suivant vue2*". On peut aussi obtenir l'arbre de syntaxe abstraite correspondant à la fusion des vues partielles courantes en cliquant sur le bouton "Get Fusion". Si le résultat de la fusion contient plus d'un

3. Notons que les conventions suivantes doivent être respectées lors de la saisie d'une grammaire :
   - *Symboles grammaticaux* : nous savons déjà que pour les arbres potentiellement ouverts, les symboles grammaticaux appartiennent à l'alphabet étendu $\mathcal{S} \cup \bar{\mathcal{S}}$ (voir sec. 3.3). Dans *TinyCE*, les symboles appartenant à $\mathcal{S}$ sont saisis en lettres capitales et leurs correspondants saisis en lettres minuscules appartiennent à $\bar{\mathcal{S}}$. Ainsi, si $A \in \mathcal{S}$ alors, $a \in \bar{\mathcal{S}}$ et dans la représentation du document sous forme d'arbre de dérivation, les feuilles étiquetées par les lettres minuscules sont des noeuds ouverts.
   - *Productions* : nous savons aussi que les productions de la grammaire appartiennent à l'ensemble $\mathcal{P} \cup \mathcal{S}_\Omega$ où $\mathcal{S}_\Omega$ contient les productions devant étiqueter les noeuds ouverts dans les arbres de syntaxe abstraite représentant les documents. Dans *TinyCE*, de telles productions sont de la forme : $p_i : \ A \longrightarrow \#$ avec $A \in \mathcal{S}$. Les productions appartenant à $\mathcal{P}$ sont sous la forme classique des productions des grammaires algébriques.

arbre, on peut naviguer entre eux au moyen des boutons "NEXT" et "PREV".

6. *Envoi de la grammaire et des documents partiels aux différents sites clients :* pour poursuivre l'édition asynchrone du document commencée sur le poste serveur au niveau des postes clients, on leur envoie la grammaire et les vues partielles par clic sur le bouton *Send* situé dans le panneau de projection correspondant à chacune des vues.
7. *Fusion des réplicats partiels :* À la réception des mises à jour des réplicats partiels envoyés par les différents clients, on peut les fusionner en cliquant sur le bouton "GET FUSION" et naviguer entre les différents résultats possibles (s'il y en a plusieurs) par actions sur les boutons "NEXT" et "PREV".

Sur le poste client :

1. *Identité des machines :* on précise l'identité du poste client ainsi que celui du serveur.
2. *Connexion au poste serveur :* on clique sur le bouton "CONNECTER".
3. *Importation d'une vue partielle de la grammaire :* sur le poste client, on importe une vue partielle de la grammaire définie sur le poste serveur par clic sur le bouton "GET GRAMMAIRE".
4. *Importation d'un réplicat partiel du document (global) :* on importe aussi un réplicat partiel du document associé à la vue dans le cas où une édition avait déjà commencée sur le poste serveur en cliquant sur le bouton "GET ARBRE PARTIEL". Si aucune édition n'avait commencé sur le poste serveur, on commence une nouvelle édition en cliquant sur le bouton "CONSTRUCTION ARBRE".
5. *Édition locale du réplicat partiel :* on peut poursuivre l'édition du document partiel obtenu du serveur en supprimant les noeuds étiquetés par des lettres minuscules (ie. des éléments de $\bar{\mathcal{S}}$) suivie de leurs remplacements par des noeuds étiquetés par des éléments de même sortes dans $\mathcal{S}$.
6. *Validation locale du réplicat partiel mise à jour :* on effectue une validation locale (du document partiel courant) en cliquant sur le bouton "VALIDATION". Un document partiel sera valide s'il existe un document global dont le document partiel courant est la projection.
7. *Visualisation des expansions possibles du document partiel courant :* dans le cas où le document partiel courant est valide, l'utilisateur peut souhaiter voir quelles sont ses expansions possibles (les plus simples)

en actionnant sur le bouton "*Expansions*". Il peut par la suite naviguer entre les différentes expansions en cliquant sur les boutons "*Next*" et "*Prev*".

Les figures 4.3, 4.4, 4.5, 4.6, 4.7 et 4.8 montrent des captures d'écran présentant une application de la démarche présentée plus bas sur le statechart de la figure 4.2, adaptation du statechart de la figure 1.2 (page 14) dans laquelle on utilise la grammaire G_{run} de la page 15.

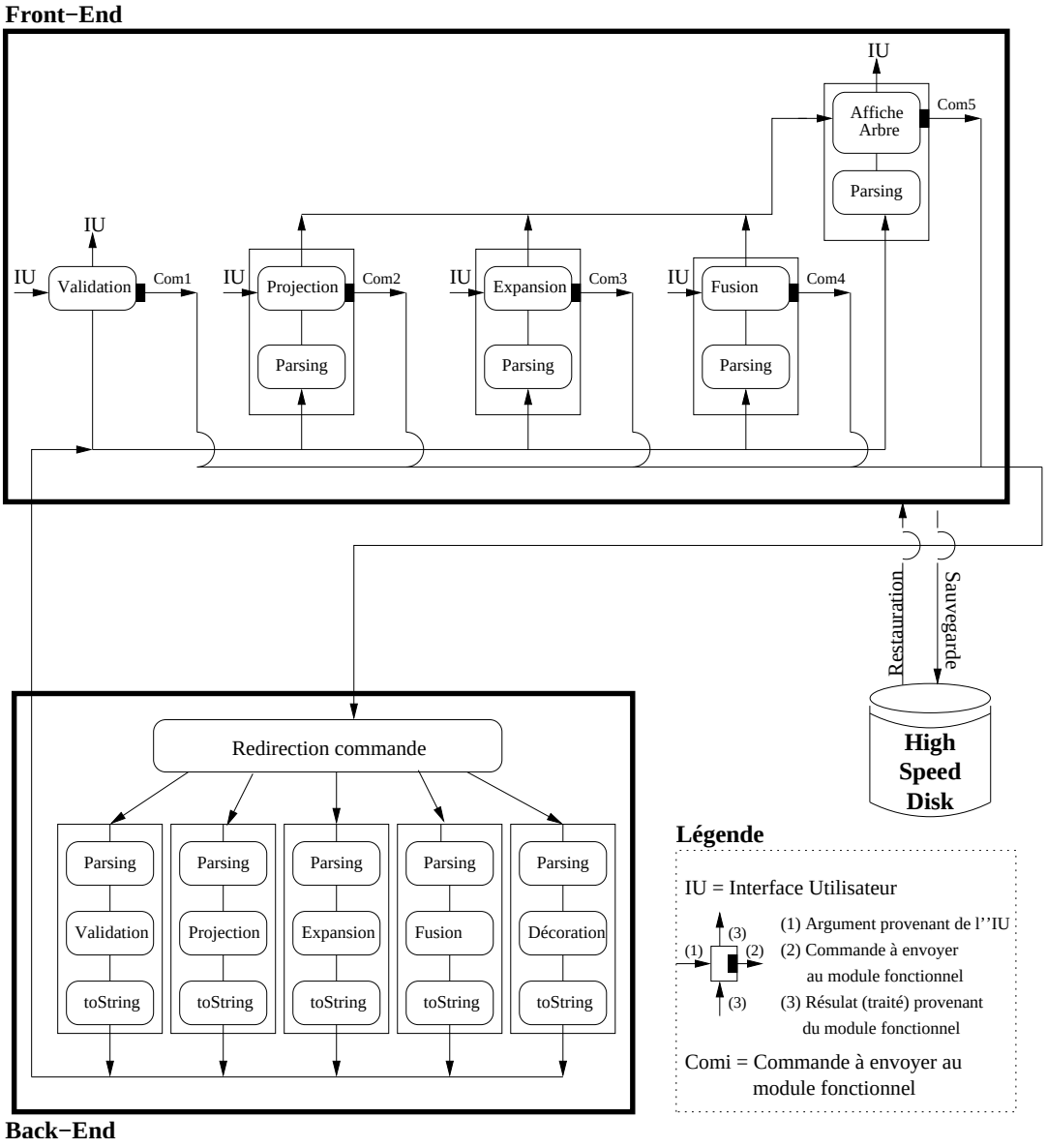

FIGURE 4.1 – Architecture de *TinyCE*

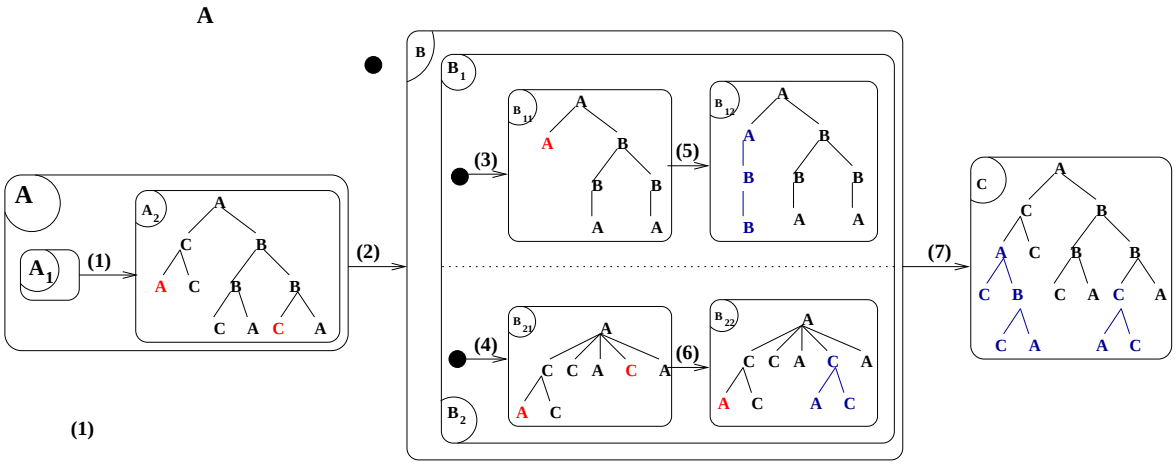

FIGURE 4.2 – Exemple d'édition coopérative asynchrone à l'aide des statescharts : apllication à la grammaire G_{run}

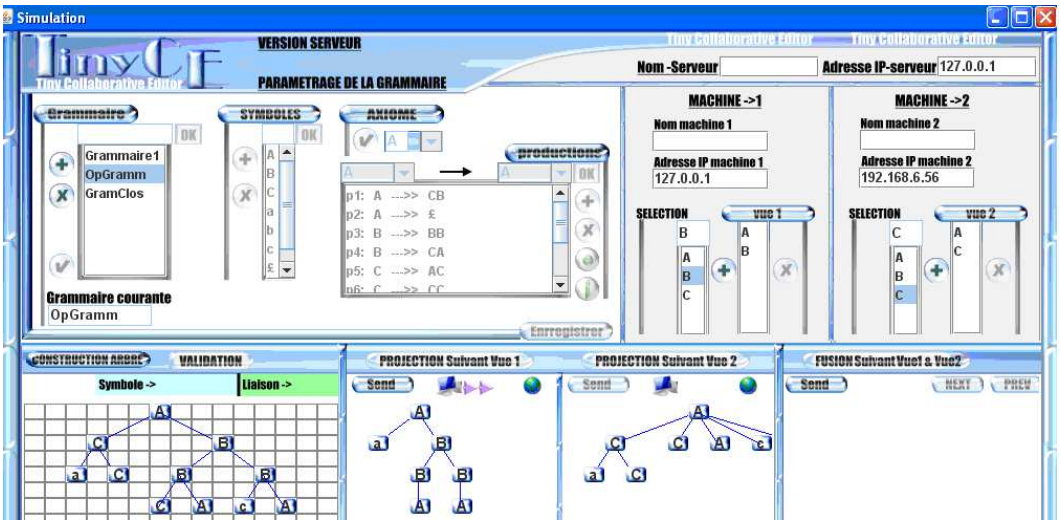

FIGURE 4.3 – Interface serveur de *TinyCE* présentant un début d'édition ainsi que les vues partielles à envoyer aux différents clients.

4.3 Synthèse

Dans ce chapitre, nous avons présenté un prototype d'éditeur coopératif appelé *TinyCE*. Ce prototype est non seulement un exemple d'implémentation des différents concepts étudiés dans cette thèse mais aussi, un outil offrant un cadre d'expérimentation.

La démarche adoptée pour l'implémentation de *TinyCE* au moyen de

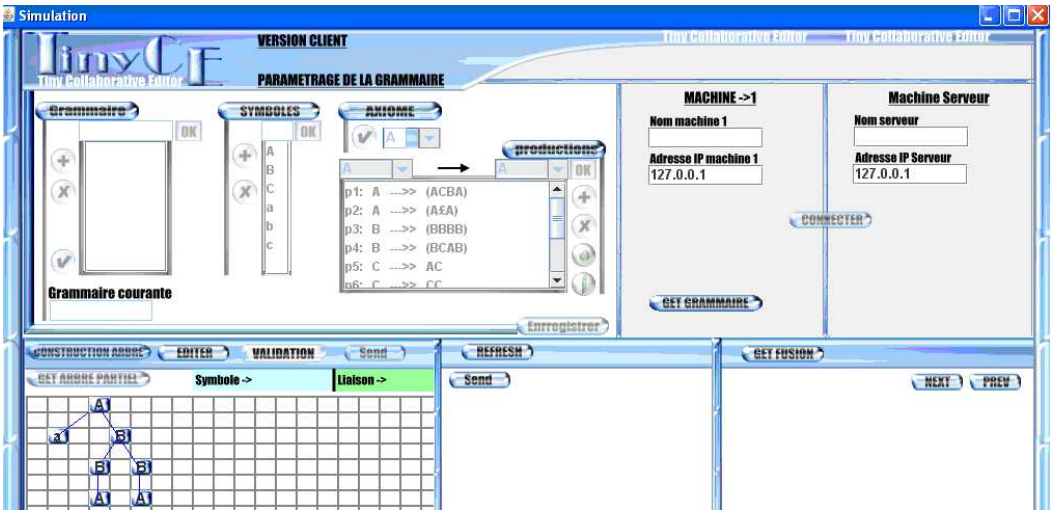

FIGURE 4.4 – Interface du premier client de *TinyCE* présentant la vue partielle reçu du serveur.

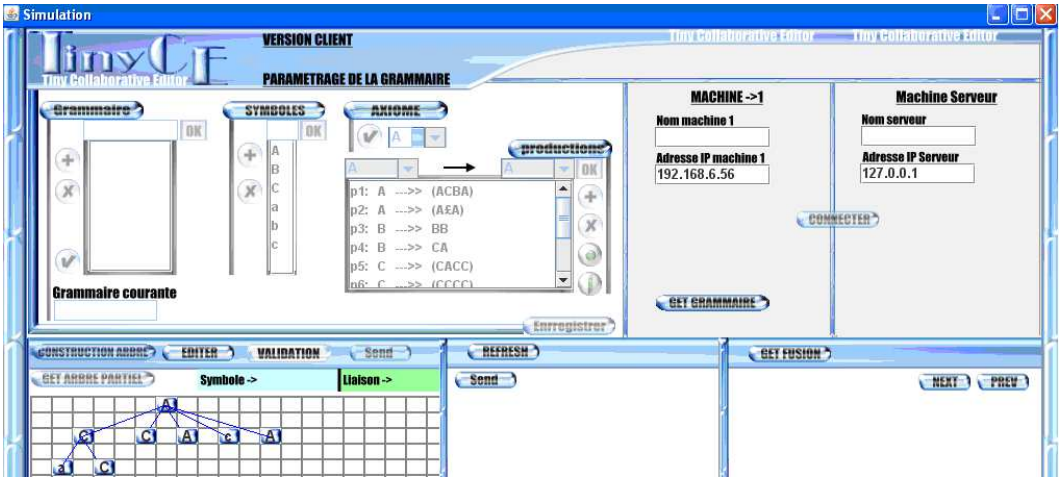

FIGURE 4.5 – Interface du second client de *TinyCE* présentant la vue partielle reçu du serveur.

deux langages dont l'un fonctionnel peut être recommandé en vue de la promotion de l'utilisation des langages fonctionnels purs. En effet, pour la production des applications, on peut avantageusement développer leurs modules métiers en utilisant un langage fonctionnel pur et un autre langage (utilisant des effets de bords et ayant des bibliothèques appropriées) pour développer les autres modules (comme le module interface utilisateur) à l'exemple de ce que nous avons fait avec *TinyCE*.

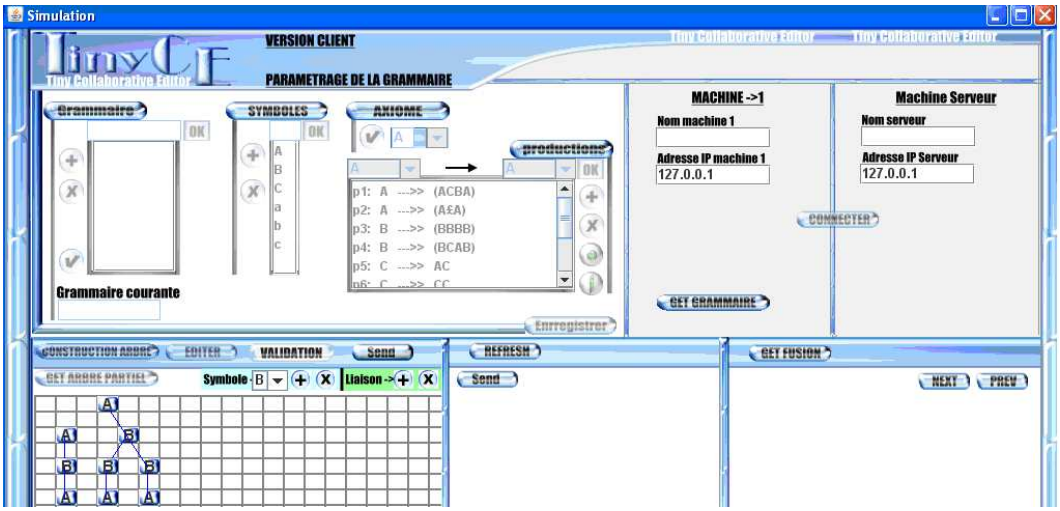

FIGURE 4.6 – Interface du premier client de *TinyCE* présentant la mise à jour locale de la vue partielle à envoyer au serveur.

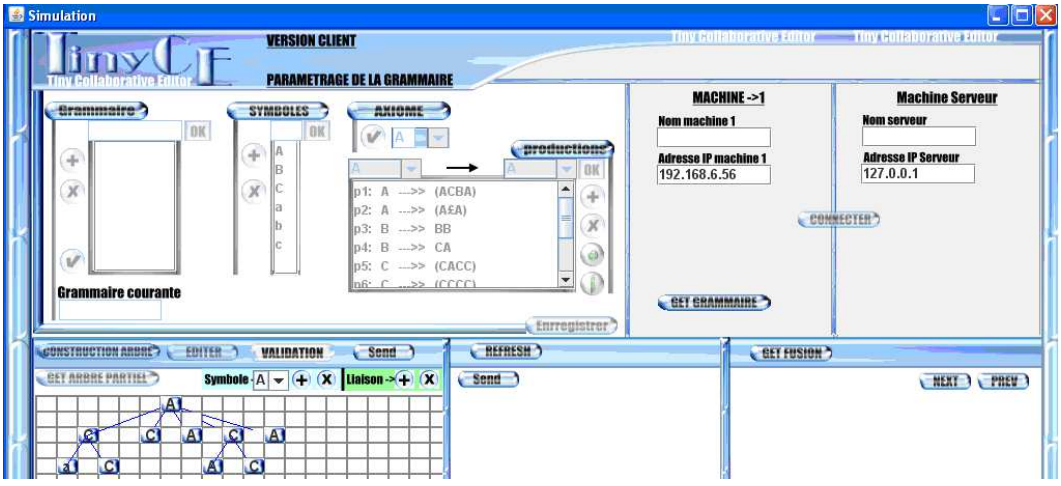

FIGURE 4.7 – Interface du second client de *TinyCE* présentant la mise à jour locale de la vue partielle à envoyer au serveur.

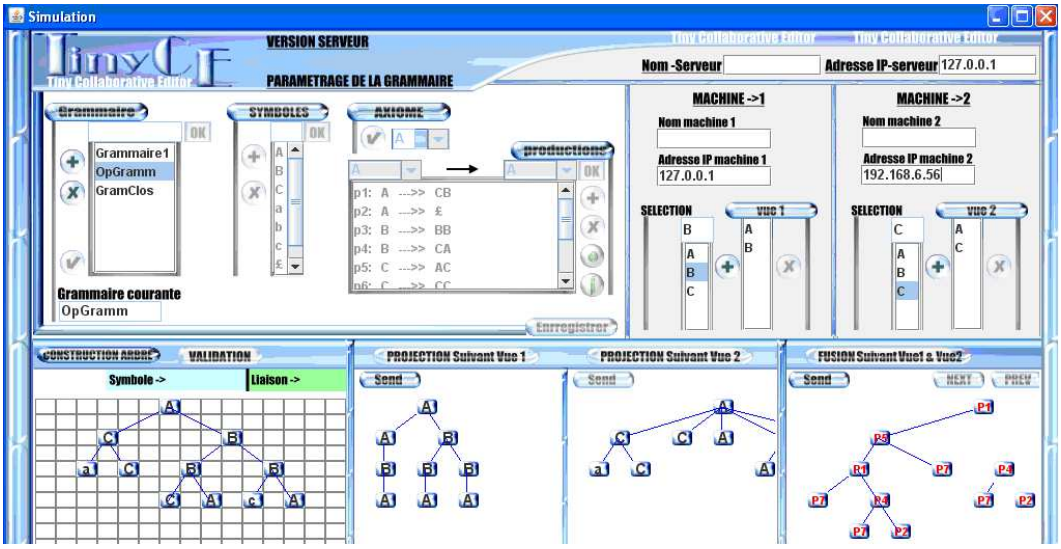

FIGURE 4.8 – Interface serveur de *TinyCE* présentant le document de base, les mises à jour de ses réplicats partiels (provenant des différents clients) ainsi que leur fusion.

Conclusion

Sommaire

Problématique abordée et choix méthodologiques **99**
Analyse critique des résultats obtenus **101**
Perspectives **105**

Nous rappelons dans un premier temps la problématique abordée dans ce travail de thèse et les choix méthodologiques que nous avons opérés pour développer nos solutions. Nous indiquons ensuite les résultats obtenus en mettant l'accent sur les limitations de ce travail et les critiques qu'il convient d'en faire. Nous terminons par une partie prospective indiquant la liste des problèmes auxquels nous nous attacherons dans des travaux futurs.

Problématique abordée et choix méthodologiques

Nous nous sommes intéressé à des systèmes à flot de tâches, ou plus généralement à des applications de travail coopératif, dans lesquelles les documents structurés sont utilisés comme interfaces entre les différents acteurs du système. Lorsqu'un intervenant réceptionne un document il doit savoir, d'après son contenu, ce qu'il *peut* et/ou ce qu'il *doit* faire à propos de ce document. L'information portée par un document ne peut que croître au cours de sa durée de vie dans le système, jusqu'au moment où l'activité, dont ce document est le support, est arrivée à son terme et que le document quitte le système (pour être éventuellement archivé). Un intervenant peut réceptionner un même document à plusieurs moments au cours de son existence dans le système ; néanmoins, il est raisonnable de supposer qu'il ne garde aucune mémoire des interactions précédentes. Il ne fera ainsi aucune différence entre un document qu'il reçoit pour la première fois et un autre qu'il a déjà rencontré. C'est le document qui porte la mémoire de ce qu'il lui est arrivé jusque là dans le système. Comme les acteurs utilisent un protocole sans mémoire il faudra également que le document soit accompagné d'une information (sous

forme d'une *continuation*) indiquant à la personne ou au programme qui le détient ce qu'il devra y faire avant de le restituer au système. C'est cette absence de mémoire qui assure une parfaite orthogonalité entre le comportement d'un intervenant dans le système et l'ensemble des tâches qui traversent ce système. On trouve le même genre de situation en programmation web : un intervenant dans un système à flot de tâches a un comportement similaire à celui d'un serveur web. Les systèmes de travail coopératif sont d'ailleurs très souvent, au moins partiellement, déployés sur le web dans la mesure où Internet est le point d'accès privilégié de certains intervenants (typiquement les "clients" du service modélisé).

Nous nous sommes intéressés à des systèmes de travail coopératif complexes consistant en un certain nombre d'activités distribuées géographiquement. Nous faisons l'hypothèse que plusieurs intervenants peuvent être concernés de façon concurrente par une même tâche sur laquelle ils interviennent sur des parties, liées à leur rôle et compétences spécifiques, qui seront probablement relativement disjointes mais néanmoins pas totalement indépendantes. Il convient donc de synchroniser ces différentes interventions, en introduisant si possible le minimum de contraintes. Comme nous l'avons vu une tâche est associée à un artefact (un document) qui décrit l'état courant de la tâche et se trouve localisée là où l'intervention est effectuée. Nous devons donc admettre qu'il puisse exister des réplicats partiels de ce document qui peuvent être manipulés de façon asynchrone par plusieurs intervenants sur des sites géographiquement distants et dont nous devrons maintenir la cohérence.

Le problème de fusion de mises à jour de différentes versions d'un même document a été étudié dans de multiples contextes (Bases de données, technologie XML, synchronisation de fichiers pour des utilisateurs nomades, développement coopératif de logiciels ...). Néanmoins les hypothèses fortes que nous avons pu poser (édition incrémentale, séparation complète entre la structure logique du document et ses présentations concrètes) nous ont permis d'obtenir des solutions algorithmiques, formellement établies, là où on ne pouvait espérer obtenir de solution dans le cas général. Nos choix méthodologiques ont essentiellement été les suivants :

1. modéliser les documents comme des arbres de syntaxe abstraite pour une grammaire algébrique,
2. assimiler une vue avec un sous-ensemble de symboles grammaticaux,
3. décrire le cycle de vie d'un document dans un système workflow par un statechart dont chaque macro-état est associé à une structure grammaticale décrivant le type des documents licites en cet état.

Les deux premières hypothèses peuvent sembler très restrictives. Nous

avons expliqué, en particulier dans l'introduction de ce manuscrit, pourquoi ces choix étaient raisonnables compte tenu de nos hypothèses de travail. Le fait que cela soit simple et raisonnable constitue une bonne base de travail. Nous sommes par conséquent partis de cette base pour développer une solution algorithmique au problème de fusion de mises à jour de réplicats partiels d'un document. Cela ne signifie pas que cela soit le dernier mot, nous revenons sur ce point un peu plus bas.

La décision de modéliser des systèmes workflow par des statecharts, si elle n'est pas totalement innovante, s'est néanmoins avérée fructueuse dans la suite de ce travail. Ce qui est nouveau, autant qu'on le sache, c'est d'avoir attaché une structure grammaticale à chaque macro-état du statechart. Cette modélisation a permis de mettre l'accent sur les deux opérations fondamentales que sont la projection d'un document sur une vue et la fusion de documents relatifs à un ensemble de vues. Opérations auxquelles nous avons par la suite consacré notre travail.

Analyse critique des résultats obtenus

Nous avons modélisé les documents en cours d'édition sous la forme d'arbres contenant des *noeuds ouverts*, appelés aussi *bourgeons*. Ce sont des feuilles à partir desquelles la structure peut se développer. Nous avons noté $t_1 \leq t_2$ lorsque t_2 peut être obtenu de t_1 en remplaçant des bourgeons de t_1 par des arbres ; nous disons alors que t_2 est une *mise à jour* ou un *raffinement* de t_1. Un document est complet, ou clos, s'il ne contient aucun bourgeon, c'est donc un élément maximal pour cet ordre. Si t est un arbre dont les noeuds sont étiquetés par des symboles grammaticaux $\mathcal{S}$ et si une vue est un sous ensemble $\mathcal{V} \subseteq \mathcal{S}$, on note $\pi_\mathcal{V}$ la projection pour laquelle $\pi_\mathcal{V}(t)$ est la forêt (suite d'arbres) obtenue à partir de t en effaçant les symboles qui ne sont pas dans $\mathcal{V}$, tout en conservant la structure hiérarchique de l'arbre de départ.

Les résultats dont on peut se prévaloir sont les suivants :

Notion d'arène Nous avons introduit une structure coinductive d'arène pour représenter des ensembles potentiellement infinis d'arbres ; tout d'abord dans le cadre des arbres clos, puis nous avons généralisée cette notion aux arbres avec bourgeons. L'ensemble des arènes constitue le point fixe d'un foncteur dont les coalgèbres nous donnent une notion d'automates d'arbre. L'anamorphisme correspondant permet d'associer à un automate d'arbre et à un état initial l'arène qui représente l'ensemble des arbres reconnus par cet automate à partir de cet état initial. Cela nous donne une notion d'arène *régulière* : une arène engendrée par

un automate fini. On sait décider de la vacuité d'une arène régulière ainsi qu'énumérer ses éléments les plus "typiques".

Cohérence de vues Nous avons donné une solution au problème suivant : *étant donnés n documents $d_1, \ldots, d_n$ représentés par des forêts (closes) sur les alphabets $\mathcal{V}_i \subseteq \mathcal{S}$, décider s'il existe un document (arbre clos sur l'alphabet $\mathcal{S}$) conforme à une grammaire donnée et dont les projections sont les d_i ($\pi_{\mathcal{V}_i}(t) = d_i$).* Pour cela on associe un automate d'arbre à une vue $\mathcal{V}_i$, de telle sorte que cet automate engendre à partir d'un état associé à d_i l'ensemble des arbres de syntaxe abstraite conformes à la grammaire donnée et ayant d_i comme projection (algorithme d'expansion) ; il suffit alors d'associer ces automates par un combinateur de synchronisation qui réalise l'intersection des ensemble d'arbres reconnus.

Fusion de mises à jour de réplicats partiels Le problème de la fusion de mises à jour de réplicats partiels s'énonce comme suit : *étant donnés un document t (représenté comme un arbre de syntaxe abstraite avec bourgeons), ses projections $d_i = \pi_{\mathcal{V}_i}(t)$ relatives à des vues $\mathcal{V}_i \subseteq \mathcal{S}$, appelés réplicats partiels de t, et des mises à jours $d_i \leq d'_i$ de ces réplicats, caractériser l'ensemble des arbres de syntaxe abstraite t' (i.e. conformes à la même grammaire que t) minimums, qui sont des raffinements de t et dont les projections sont des raffinements des mises à jours des divers réplicats :*

$$t \leq t' \qquad et \qquad \forall i \quad d'_i \leq \pi_{\mathcal{V}_i}(t')$$

Et en particulier peut-on décider si cet ensemble est non vide ?. Nous avons apporté une solution à ce problème en adaptant l'algorithme d'expansion et l'opérateur de synchronisation des automates dans le cadre des arbres avec bourgeons.

TinyCE Enfin nous avons développé un prototype permettant d'expérimenter nos algorithmes. Cet outil peut constituer un embryon pour le développement d'une plateforme expérimentale de conception et de simulation d'applications de travail coopératif basées sur l'échange de documents structurés.

Les premières critiques que l'on peut émettre à l'égard de notre travail sont les suivantes :

Haskell vs les mathématiques Nous avons fait le choix de présenter notre approche sous la forme de code Haskell commenté ; le lecteur aurait pu préférer une approche plus classique dans laquelle on pose des définitions mathématiques précises. Le choix du langage Haskell était jus-

tifié par l'importance donnée aux structures coinductives pour la représentation et la manipulation d'un espace de solution contenant un nombre potentiellement infini d'éléments. Par ailleurs les portions de code que nous avons présentées n'excèdent généralement pas une dizaine de lignes ; aucun pseudo code n'aurait été de toute façon plus compact et plus lisible. Par ailleurs un programme Haskell est aussi précis et rigoureux qu'une définition formelle et présente l'avantage d'être exécutable ; le fait d'avoir pu expérimenter notre approche au cours du développement de notre travail est quelque chose que l'on a grandement apprécié. Le langage mathématique est néanmoins plus directement compréhensible et il est plus adapté lorsqu'on doit prouver certaines propriétés formellement. Maintenant que nous avons une meilleure compréhension du sujet il est sans doute temps de procéder à une étude plus formelle au cours de laquelle nous étudierons plus précisément les propriétés mathématiques des objets que nous utilisons (arènes, automates d'arbres ...).

Arènes vs automates d'arbres On peut arguer que le travail aurait pu être présenté uniquement en terme d'automates d'arbre sans qu'il soit utile d'introduire la notion d'arène. D'autant plus que les problèmes de décision telle que la vacuité d'une arène et l'extraction d'un élément de cet ensemble n'est *a priori* possible que pour des arènes régulières, c'est-à-dire associées à des automates d'arbres finis. Néanmoins, du fait que l'on y fasse abstraction de l'ensemble des états, la synchronisation des arènes est une opération plus simple que celle des automates ; et si nous supposons qu'il n'y ait pas ambiguïté, c'est-à-dire que le document résultant de son ancienne version et des différentes mises à jour de ses réplicats partiels est caractérisé de manière unique, alors, on pourra directement extraire ce document de l'arène correspondante. Par ailleurs nous pensons que l'étude des arènes peut présenter un intérêt en soi, mais il est vrai que nous n'avons pas pour l'instant d'arguments pour défendre ce point de vue dans la mesure où cette étude formelle de la structure d'arène n'a pas encore été conduite.

Grammaires généralisées Dans des documents XML il est usuel de manipuler des noeuds ayant un nombre non borné de successeurs, par exemple pour représenter une liste, en générale non ordonnée, d'éléments d'une même catégorie : employés d'une entreprise, livres d'une bibliothèque, hôtels d'un système de réservation ... Il faudrait de ce point de vue utiliser des grammaires algébriques généralisées dans lesquelles les parties droites des productions sont des expressions régulières, ou du moins prendre en compte des grammaires algébriques généralisées

sous la forme particulière suivante : on dispose de deux types de productions, celles de la forme classique $p : A_0 \to A_1 \dots A_n$ pour lesquelles chaque symbole en partie droite est accessible en utilisant le sélecteur correspondant et celles de la forme $q : A \to B^*$ pour lesquelles chaque membre en partie droite possède un identifiant unique. Ces identifiants sont indispensables pour l'opération de fusion car sur chacun des réplicats on peut être amené aussi bien à compléter la structure d'un élément existant d'une liste qu'à introduire de nouveaux éléments à cette liste ; il est donc indispensable de savoir distinguer ces différents éléments de manière non ambiguë. Cela signifie que le successeur d'un noeud "et" d'une arène devra être une liste associative dans laquelle la clé d'un élément sera soit son sélecteur, dans le premier cas, ou son identifiant, dans le second cas. Cette extension ne devrait pas poser de difficulté majeure. Nous aurons ainsi à manipuler des arbres de largeur possiblement non bornée (en certains noeuds). Par contre il est tout à fait raisonnable de supposer que les arbres manipulés soient de profondeur bornée, ce qui revient à supposer que la grammaire (généralisée) ne contienne pas de symboles grammaticaux X pour lesquels on puisse trouver de dérivation de la forme $X \Rightarrow^+ \alpha X \beta$. Nous ne connaissons pas de DTD dans lesquelles une telle situation se présente. On pourrait sans doute imaginer une telle situation mais cela doit être exceptionnelle dans la pratique.

Le modèle n'est pas hiérarchique ! Sous l'hypothèse précédente il devrait être raisonnablement facile de trouver une grammaire dont les arbres de syntaxe abstraite sont en correspondance bijective avec les projections des arbres de syntaxe abstraite de la grammaire de départ selon une vue donnée. Ceci est indispensable pour rendre notre modèle hiérarchique. Nous avons, de fait, utilisé un modèle hiérarchique (les *statecharts*) pour décrire la circulation d'un document dans le *workflow* ; mais pour l'instant nous n'avons pas exploité cette propriété car nous n'avons décrit qu'à un niveau donné, les opérations de décomposition d'un document et de fusion des mises à jour de ses réplicats partiels. Ce qui nous manque pour l'instant est un moyen de relier la grammaire d'un noeud "et" d'un *statechart* à celles de ses différents constituants. Nous pensons que l'hypothèse selon laquelle les documents ont une profondeur bornée pourrait être à la base de la solution.

Gestion des conflits Nous n'avons rien dit sur ce qu'il convenait de faire dans le cas où un conflit apparaissait entre les mises à jour des réplicats partiels rendant leur fusion impossible. Dans l'introduction de ce travail nous avons signalé qu'il y avait en gros deux attitudes possibles.

La première, la réconciliation, consiste à remettre en cause un nombre, si possible minimal, d'action d'édition locale afin de restaurer un état dans lequel la fusion est rendue possible. Il s'agit donc d'une solution *a posteriori* où on laisse les conflits apparaître et on les traite au moment de la fusion. La seconde solution consiste à prévenir *a priori* les conflits en contrôlant les actions d'édition locale. La seconde solution qui paraît plus satisfaisante peut néanmoins avoir l'inconvénient de restreindre un peu trop le comportement ainsi que de limiter la concurrence du système par ajout de points de synchronisation (modification du *statechart*).

Méthodologie de conception On aimerait à terme que nos travaux puissent contribuer à mettre en place une méthodologie de conception d'application de travail coopératif qui reposerait sur des outils pour assister le concepteur dans l'élaboration de son modèle (statechart) de façon ascendante par la réutilisation de sous systèmes préalablement élaborés ainsi que d'outils externes. Nous en sommes encore bien loin. Notre prototype est plus un outil de démonstration des algorithmes introduits dans ce travail de thèse, que l'ébauche d'une telle plateforme. Néanmoins, l'étude théorique n'en est encore qu'à ses débuts, et il serait sans doute trop tôt de se lancer dans de tels développements.

Perspectives

Au cours de l'analyse critique de notre travail donnée dans la section précédente, nous avons identifié un certain nombre de pistes de travail pour étendre et améliorer ce qui a été présenté dans ce manuscrit. Elles se résument comme suit :

Formalisation Donner un traitement plus formel et établir des résultats mathématiques sur les objets manipulés dans le cadre de ce travail de thèse (arènes et automates d'arbre).

Extension aux grammaires généralisées Etendre le travail aux grammaires algébriques généralisées pour prendre en compte les DTD qu'on rencontre usuellement dans la pratique.

Rendre le modèle hiérarchique Faire l'hypothèse que les documents se présentent comme des arbres de syntaxe abstraite de profondeur bornée (mais de largeur non bornée) pour rendre le modèle hiérarchique et justifier ainsi pleinement la modélisation par *statecharts*.

Réconciliation et contrôle Donner une solution aux conflits apparaissant lors de la fusion des réplicats partiels d'un document en développant

des techniques de réconciliation et de contrôle adaptées à notre modélisation.

Mais un certain nombre de travaux complémentaires peuvent également être très intéressants. Décrivons sommairement ceux qui nous viennent à l'esprit.

Algorithme d'expansion Rappelons 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 des productions explicites $A \to \epsilon$ pour tout symbole terminal. 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 dérivation. 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 généralisée opérant sur des données partiellement structurées. Nous avons donc une situation similaire au développement d'analyseurs syntaxique fonctionnels [Fok95, HM98]. 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 nouvelle algorithmique des analyseurs fonctionnels. 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, similaire aux combinateurs fonctionnels d'analyse [Fok95, HM98], 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 [AG00, Mu03, MB03] pourraient s'appliquer à notre algorithme d'expansion.

Ambiguïté et fusion de structures grammaticales Le problème de l'ambiguïté des grammaires algébriques est indécidable. Comme nous venons de le rappeler ce problème peut se réduire au problème de savoir si un ensemble de vues partielles d'un document caractérisent celui-ci. Ce dernier problème est donc lui-même également indécidable. Ceci signifie qu'on ne peut espérer avoir une méthode générale pour valider un *statechart* en rassurant le concepteur sur le fait qu'un document pourra toujours être déterminé de façon non ambiguë à la sortie d'un

noeud “et” (lorsqu'on fusionne les vues partielles). Néanmoins, dans la mesure où on procède de façon ascendante, on disposera d'un ensemble de grammaires décrivant les structures syntaxiques manipulées par les différents constituants d'un noeud “et” et il s'agit de concevoir une nouvelle grammaire réalisant de façon non ambiguë la fusion de ces grammaires. Ce problème est beaucoup plus simple. C'est à dire qu'on ne cherchera pas à valider un *statechart* donné *a priori* mais à le construire de telle sorte que la non ambiguïté soit assurée.

Documents actifs Les services web mettent l'accent sur l'utilisation de documents actifs dans lesquels des requêtes sont attachées à certains noeuds dans le but de collecter dynamiquement de l'information à partir d'autres documents [ABM04]. Il pourrait être intéressant d'exécuter de telles requêtes sur des réplicats partiels au moment de leur fusion ce qui autoriserait des flots d'information complémentaires entre les composants au moment de leur synchronisation. On pourrait aborder ce problème dans le contexte des grammaires attribuées en utilisant les techniques d'évaluations fonctionnelles des attributs présentées dans [Joh87, FJMM91, Bac02]. De fait, la valeur d'un attribut d'un symbole grammatical visible pourrait dépendre de la valeur d'attributs de symboles invisibles (mais visibles pour d'autres composantes), et ainsi le calcul d'attribut pourrait procurer un moyen supplémentaire pour coordonner les activités qui ont lieu sur des vues partielles différentes du document.

Bibliographie

[Abi99] Serge Abiteboul. On views and xml. In *Proceedings of the eighteenth ACM Symposium on Principles of Database Systems*, pages 1–9, 1999.

[ABM04] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active xml. In *PODS '04 : Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–45, New York, NY, USA, 2004. ACM.

[AG00] Sergei M. Abramov and Robert Glück. The universal resolving algorithm : inverse computation in a functional language. In *Proceedings of Mathematics of Program Construction 2000. R. Backhouse and J. N. Oliveira, Eds. Springer-Verlag Lecture Notes in Computer Science vol. 1837*, pages 187–212, 2000.

[Ask94] U. Asklund. Identifying conflicts during structural merge. In *Proceedings of Nordic Workshop on Programming Environment Research (NWPER'94), Nordic Workshop on Programming Environment Research, Lund, Sweden .*, pages 231–242, June 1994.

[Bac02] 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.

[BB97] Jean Berstel and Luc Boasson. Balanced grammars and their languages. *In Formal and Natural Computing : Essays Dedicated to Grzegorz Rozenberg Springer Lecture Notes in Computer Science*, 1243 :135–150, 1997.

[BB00] J. Berstel and L. Boasson. Xml-grammars. *(MFCS 2000) Mathematical Foundations of computer Science (M. Nielsen and B. Rovan, Eds.) Springer-Verlag, Lect. Notes Comput Sci.*, 1893 :182–191, 2000.

[BCG02] Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of*

Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, volume 2297 of *Lecture Notes in Computer Science*. Springer, 2002.

[BD07] Eric Badouel and Rodrigue Djeumen. Modular grammars and splitting of catamorphisms. Technical report, INRIA/IRISA, INRIA/IRISA, Rennes, Campus de Beaulieu, October 2007.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2 edition, May 1998.

[Bon98] Stéphane Bonhomme. *Transformation de documents structurés, une combinaison des approches explicite et automatique*. PhD thesis, Université JOSEPH FOURIER, 1998.

[BP98] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer ? In *proceeding of 4th Int. Conf. on Mobile Computing and Networking (MOBICOM). ACM/IEEE, Dallas, TX, USA*, pages 98–108, 1998.

[Bra02] V. Braganholo. *Updating relational databases through xml views*. PhD thesis, PPGC-UFRGS, Porto Alegre, Thesis proposal - in preparation,, 2002.

[BT07] Eric Badouel and Maurice T. Tchoupé. Projection et cohérence de vues dans les grammaires algébriques. *Electronic Journal of ARIMA (African Revue in Informatics and Mathematics Applied)*, 8 :18–48, 2007.

[BT08] Eric Badouel and Maurice Tchoupé. Merging hierarchically-structured documents in workflow systems. *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008), Budapest. Electronic Notes in Theoretical Computer Science*, 203(5) :3–24, 2008.

[CLL02] Ya Bing Chen, Tok Wang Ling, and Mong Li Lee. Designing valid xml views. In *In Proceedings of the 21st International Conference on Conceptual Modeling (ERŠ02*, pages 463–477, 2002.

[CSGM] Chawathe, Sudarshan, and Hector Garcia-Molina. Meaningful change detection in structured data. In *ACM SIGMOD International Conference on Management of Data, ACM Press*, pages 26–37.

[CSR+] Chawathe, Sudarshan, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchical structured information. In *ACM SIGMOD International Conference on Management of Data, ACM Press*, pages 493–504.

[Dav92] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, New York, NY, USA, 1992.

[DV04] Kees Doets and Jan Eijck Van. *The Haskell Road To Logic, Maths And Programming*. King's College Publications, May 2004.

[FJMM91] M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammars to catamorphisms. In *The Squiggolist, 2(1)*, pages 20–26, 1991.

[Fok95] J. Fokker. Functional parsers. In *First International School on Advanced Functional Programming, J. Jeuring and E. Meijer, editors, Lecture Notes in Computer Science*, volume 925, pages 1–23, 1995.

[Fon02] Robin La Fontaine. Merging xml files : A new approach providing intelligent merge of xml data sets. In *proceedings of XML Europe, Berlin, Germany*, 2002.

[GdM05] Jeremy Gibbons and Oege de Moor. *The Fun of Programming (Cornerstones of Computing S.)*. Palgrave Macmillan, June 2005.

[GHC] The Glasgow Haskell Compiler : GHC. http ://haskell.org/ghc/.

[Har87] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[HF99] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell 98. Technical report, 1999.

[HM98] Graham Hutton and E. Meijer. Monadic parsing in haskell. *J. Functional Programming*, 8(4) :437 – 444, 1998.

[HO00] Cordelia Hall and John O'Donnell. *Discrete Mathematics using a Computer*. Springer, 2000.

[HSR89] Horwitz, J. Prins S., and T. Reps. Integrating noninterfering versions of programs. In *ACM Transactions on Programming Languages and Systems*, pages 345–387, 1989.

[Hud00] Paul Hudak. *The Haskell School of Expression : Learning Functional Programming through Multimedia*. Cambridge University Press, February 2000.

[IOM+07] Claudia Ignat, Gérald Oster, Pascal Molli, Michelle Cart, Jean Ferrié, Anne-Marie Kermarrec, Pierre Sutra, Marc Shapiro, Lamia Benmouffok, Jean-Michel Busca, and Rachid Guerraoui. A comparison of optimistic approaches to collaborative editing of

wiki pages. In *International Conference on Collaborative Computing (CollaborateCom), White Plains, NY, USA.*, pages 474–483, nov 2007.

[Joh87] 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, Springer-Verlag*, pages 154–173, 1987.

[Jon03] Simon P. Jones. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, May 2003.

[Knu67] Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11(3) :269–289, 1967.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical System Theory*, 2(2) :127–145, june 1968.

[Lin04] Tancred Lindon. A three-way merge for xml documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 1–10, October 2004.

[Man01] Gerald W. Manger. Generic algorithm for merging sgml/xml-instances. In *proceedings of XML Europe, Berlin, Germany*, 2001.

[MB03] Shin-Cheng Mu and Richard Bird. Theory and applications of inverting functions as folds. In *Science of Computer Programming (Special Issue for Mathematics of Program Construction, vol.51*, pages 87–116, 2003.

[McN67] R. McNaughton. Parenthesis grammars. *Journal of the ACM*, 14(3) :490–500, 1967.

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5) :449–462, 2002.

[MHT] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. A language for bidirectional updating based on injective mapping (unpublished).

[MHT04] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bidirectional updating. In *Second Asian Symposium on Programming Languages and Systems*, pages 2–18, 2004.

[MHT06] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. A programmable editor for developping structured documents based on bidirectional transformations. In *Higher-Order and Symbolic Computation (1)*, pages 1–31, 2006.

[Mu03] Shin-Cheng Mu. *A Calculational Approach to Program Inversion.* PhD thesis, Oxford University Computing Laboratory, 2003.

[Paa95] J. Paakki. Attribute grammar paradigms–a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2) :196–255, june 1995.

[Sax] Saxon. http ://saxon.sourceforge.net/.

[Sha02] Marc Shapiro. *La gestion des objets dans les systèmes répartis de grande échelle.* Habilitation à diriger les recherches (hdr), Université Paris VI — Pierre et Marie Curie, Paris, France, November 2002.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1) :42–81, 2005.

[Tho99] Simon Thompson. *Haskell : The Craft of Functional Programming (2nd Edition).* Addison Wesley, March 1999.

[Ven00] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types.* PhD thesis, Faculty of Mathematics, University of Tartu, Estonia, 2000.

[W3C00] W3C. extensible markup language (xml). W3C Recommendation(1.0 (second edition), octobre 2000.

[WW97] Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *in Database Theory – ICDT'97, Lecture Notes in Computer Science*, volume 1186, pages 230–246, 1997.

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

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

[ZHT04] Shin-Cheng Mu Zhenjiang Hu, Kento Emoto and Masato Takeichi. Bidirectionalizing tree transformations. In *Workshop on New Approaches to Software Construction (WNASC 2004), Komaba, Tokyo, Japan*, September 2004.

Annexe A

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 [Jon03], et à l'introduction [HF99] 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 [Bir98, Dav92, Tho99] mais le lecteur pourra aussi consulter [Hud00, DV04, GdM05, HO00] avec intérêt). Les éléments qui suivent devraient néanmoins suffire pour une bonne compréhension du code Haskell présenté dans notre document.

A.1 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
- $[(x, y) | x \leftarrow [1..3], y \leftarrow [x..3]]$ vaut $[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]$,
- $[(x, y) | x \leftarrow [1, 2], y \leftarrow [{}'a', {}'b']]$ vaut $[(1, {}'a'), (1, {}'b'), (2, {}'a'), (2, {}'b')]$ et
- $[(x, y) | y \leftarrow [{}'a', {}'b'], x \leftarrow [1, 2]]$ 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 B

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 du document.

La fonction suivante recherche une production donnée par ses parties droite et gauche sous l'hypothèse, que nous faisons dans l'ensemble de notre document, 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 :

```
ast2der :: (Eq symb) ⇒ Gram prod symb → Tree prod → Tree symb
ast2der gram (Node p ts) = Node (lhs gram p) (map (ast2der gram) ts)
```

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. [1]

1. On utilise ici la monade *Maybe* pour la composition des fonctions (applications par-

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

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 (suffixe) non consommée par le processus d'analyse. Pour composer des fonctions on utilise la monade *Maybe* des résultats

---

tielles) :

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

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 :

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

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 corres-

pondante

```
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 = case fxs of
                Nothing   → g xs
                otherwise → fxs
           where fxs = f xs
```

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
```