Les grammaires attribuées pour la conception et l'assemblage de langages dédiés

Bernard Fotsing

UNIVERSITÉ DE YAOUNDÉ I
UNIVERSITY OF YAOUNDE I
FACULTÉ DES SCIENCES
FACULTY OF SCIENCE
DÉPARTEMENT D'INFORMATIQUE
DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITÉ DE RENNES I
ECOLE DOCTORALE MATISSE

Numéro d'ordre de la thèse : 3971

Les grammaires attribuées pour la conception et l'assemblage de langages dédiés

THÈSE EN COTUTELLE

Présentée et soutenue en vue de l'obtention du
Doctorat/Ph.D de l'Université de Yaoundé I et du
Doctorat de l'Université de Rennes I en Informatique

par

FOTSING TALLA Bernard
Matricule : 99Y304
DEA en Informatique

devant le jury composé de

Président :	TCHUENTÉ Maurice	Professeur, Université de Yaoundé 1
Rapporteurs :	Éric BADOUEL	HDR, C.R. INRIA Rennes Bretagne Atlantique
	TANGHA Claude	Maître de Conférences, Université de Yaoundé 1
Membres :	Jean-Claude DERNIAME	Professeur Émérite, LORIA Nancy - France
	Didier PARIGOT	HDR, C.R. INRIA Sophia Antipolis - France
	NDOUNDAM Réné	Chargé de Cours, Université de Yaoundé I

Année : 2010

Dédicace

À ma tendre épouse Adeline.
À Sylvanie, Michel et Myriam.

Remerciements

Je rends grâce à DIEU Tout Puissant qui m'a donné la vie et qui m'a permis d'être là aujourd'hui pour exprimer ma gratitude à ceux qui, de près ou de loin, ont contribué d'une façon ou d'une autre, à la réalisation de ce travail.

- Je voudrais ainsi remercier du fond de mon cœur Dr. Eric BADOUEL, Chargé de Recherche à l'INRIA de Rennes, et Pr. Claude TANGHA, Maître de Conférences à l'ENSP de Yaoundé I, qui m'ont initié à la recherche et ont accepté de diriger cette thèse. Leur aide et leur amitié pendant ces dernières années ont été déterminantes pour l'aboutissement de ce travail. Ils ont toujours su quand intervenir pour me remettre sur les rails.

- J'exprime ma profonde gratitude au Professeur Maurice TCHUENTÉ pour tout le sacrifice qu'il ne cesse de consentir pour notre formation et pour avoir accepté de présider le jury de cette soutenance.

- Je remercie le Professeur Jean Claude DERNIAME et Dr. Didier PARIGOT, Chargé de Recherche INRIA pour avoir accepté d'évaluer ce travail. Je les remercie également pour le privilège qu'ils m'ont accordé en acceptant d'être membres du jury.

- Je remercie Dr. René NDOUNDAM pour avoir accepté dans la simplicité légendaire qui le caractérise d'examiner ce travail. C'est un honneur pour moi de pouvoir le compter parmi les membres du jury de cette soutenance.

- J'exprime également ma gratitude à tous les enseignants des Départements d'Informatique des Universités de Dschang et de Yaoundé I, ainsi qu'à tout le personnel administratif de ces institutions, qui ont participé énormément à ma formation. Je remercie particulièrement Pr. Clémentin TAYOU pour le sacrifice consenti pendant notre formation initiale et, Dr. Georges E. KOUAMOU pour sa disponibilité sans faille tout au long de mon cursus académique.

- Je remercie également L'AUF et l'INRIA à travers le projet S4 qui ont financé cette thèse. Je remercie de tout cœur les membres du projet S4 pour l'accueil chaleureux qu'ils m'ont toujours réservé pendant mes séjours à Rennes. En particulier, j'exprime ma gratitude à Laurence Dinh, assistante du projet S4, qui a toujours été présente pour répondre aux questions administratives.

J'exprime ma reconnaissance à toute ma famille qui, malgré la distance et le temps, m'ont toujours soutenu et encouragé depuis mon enfance.

- J'exprime tout d'abord ma profonde gratitude à Papa Marcel pour ses encouragements et les conseils précieux qu'il n'a cessé de me prodiguer pendant toute la durée de cette thèse.

- Je remercie Maman Makuh et grand-mère Majougang pour leur soutien et leur amour indescriptible.

- Je remercie tous mes frères et sœurs qui m'ont soutenu de quelque manière que ce soit durant tout ce temps. Je remercie ma belle-sœur Tata Rosine qui a été et qui reste pour moi comme une grande sœur.

- Je dois beaucoup et je leur suis très reconnaisant, à tous les frangins de toujours (Maurice, Zéphyrin, Rodrigue Tchougong, Rodrigue Djeumen, Job Alain, Pierre Lotis, François, Vigny, Anne Marie, Nascisse, Mathurin, Guy, Pascal, Jean-Robert, Silas, Léopold, Salomon, et j'en oublie sûrement) qui ont toujours répondu présents à mes sollicitations. Je n'oublie pas mes amis de Rennes, du LABOMAT et de Polytech (Paulin, Etienne, Anicet, Francine, Norbert, Yanik, Donatien, Élie, Vivient, Adamou, Armel, Jerry, Serge E., ...), de Dschang (Serge G., Vincent, Château, Junior, Molière, Rémon, Placide, Léonard, ...) et de Bamendjou (Madeleine, Odette, Kézétas, Annie, Céline, ...) pour tous les bons moments passés ensemble.

- Je dois beaucoup à ma très chère et tendre épouse Adeline pour son amour, mais surtout sa patience et son soutien moral et spirituel. Je ne sais pas ce que je serai sans toi Adeline. Trouve dans ces lignes l'expression de ma parfaite reconnaissance et de mon plus grand amour.

- Je voudrais enfin rendre hommage à un être qui m'est très cher ; quelqu'un qui aurait été d'une joie incommensurable aujourd'hui ; quelqu'un qui n'a eu durant sa vie pour seules priorités que l'éducation et la formation de son entourage et des siens. Repose en paix Papa !

Table des matières

Table des figures

Liste des tableaux

Résumé

La Programmation Orientée Langage est un paradigme de programmation qui tente, par la technique de méta-programmation, de changer les habitudes des développeurs de systèmes informatiques en leur permettant de « travailler en termes de concepts et notions du problème à résoudre, au lieu d'être toujours obligés de traduire leurs idées aux notions qu'un langage généraliste est capable de comprendre » [1,2]. Le développement de logiciels passe de ce fait par la conception de langages dédiés : on définit un ou plusieurs langages qui capturent les caractéristiques du domaine étudié, puis on écrit les applications visées en utilisant ces langages. Dans cette thèse, nous proposons une démarche méthodologique de développement logiciel reposant sur ce concept. Il s'agit de conduire la même démarche méthodologique au niveau des langages que ce qui est classiquement fait au niveau des composants logiciels. En l'occurrence, nous utilisons le formalisme des grammaires attribuées pour tenter de répondre à la question suivante : *comment peut-on créer de nouveaux langages par composition de langages réutilisables existants ?* Nous tirons profit de la traduction des grammaires attribuées en algèbres de combinateurs fonctionnels pour définir des spécifications exécutables de langages dédiés (vus comme composants logiciels), plongés dans le langage fonctionnel pur Haskell. À partir d'exemples significatifs d'extension et de réutilisation de langages dédiés (par stratification de ceux-ci, ou par changement de la monade considérée), nous proposons un typage de langages dédiés en vue de leur assemblage et leur réutilisation. Pour illustrer cette démarche, nous décrivons un langage dédié (bibliothèque de combinateurs) pour l'édition de documents structurés. Un document y est représenté par un Zipper [3] attribué, une structure arborescente localisable, représentant un arbre et son contexte, et caractérisée par une grammaire attribuée. L'édition consiste alors à la modification interactive de cette structure ; ce qui entraîne une réévaluation totale ou partielle des attributs. L'édition peut aussi être réalisée à travers une vue abstraite obtenue par projection de la structure concrète. Ce qui pose le problème de *mise à jour de vue*, un problème familier de la communauté des bases de données, auquel nous donnons une solution grâce à nos combinateurs d'éditeurs.

Mots clés : Grammaires attribuées, Évaluation paresseuse d'attributs, Langages dédiés, Combinateurs fonctionnels, Système de types, Haskell, Réutilisabilité, Édition structurée interactive.

Abstract

Language Oriented Programming (LOP) is a style of computer programming, via metaprogramming, which allows the programmer "to work in terms of the concepts and notions of the problem, instead of being forced to translate his ideas into the notions that a fixed general-purpose language is able to understand" [1, 2]. That means the programmer creates one or more domain-specific languages (DSL) for the problem first, and solves the problem in those languages. In this thesis, we propose a methodological approach to software development based on this concept. We aim to follow the same methodological approach in terms of languages than what is conventionally done for software components. In this case, we use the formalism of attribute grammars to try to answer the following question : *how can we create new languages by composing reusable existing languages ?* We take advantage of their translation into algebra of functional combinators to define functional executable specifications of domain-specific languages (viewed as software components), embedded in the purly functional language Haskell. From significant examples of extension and reuse of domain-specific languages (by stratification of DSLs or by changing the respected monad), we propose a type system for domain-specific languages for their assembly and reuse. To illustrate this approach, we describe a domain-specific language (a library of combinators) for editing structured documents. A document is represented by an attributed Zipper [3], a localizable tree-structure, representing a tree and its context, and characterized by an attribute grammar. The proccess of editing consists therefore in the interactive modification of this structure, resulting in partial or total reevaluation of attributes. The data can also be edited through an abstract view obtained by projection of the concrete structure. This raises the problem of *view update*, a well-known problem of the database community, to which we give a solution with our editor combinators.

Keywords : Attribute grammars, Lazy attribute evaluation, Domain-specific languages, Functional combinators, Type system, Haskell, Reusability, Interactive structured editing.

Introduction générale

Sommaire

La crise du logiciel

Le génie logiciel a vu le jour vers la fin des années 60, et c'est dans cette même période qu'on a commencé à évoquer la *crise du logiciel* pour qualifier les problèmes liés à la réalisation de projets informatiques qui se traduisaient par des délais de développement imprévisibles, une qualité insuffisante des produits réalisés et une inadéquation avec les besoins des utilisateurs. L'origine de ces problèmes venait de la progression constante et rapide de la taille et de la complexité des applications, et de la difficulté de capturer précisément les exigences des utilisateurs. A ces difficultés initiales s'en rajoutent maintenant de nouvelles liées à l'ubiquité croissante de l'informatique qu'on trouvera aussi bien sur des supports aux ressources limitées (en terme d'énergie et de mémoire), qu'embarquée dans des moyens de transports ou distribuée à travers Internet. Des préoccupations telles que la sûreté de fonctionnement, la confidentialité des informations, la sensibilité aux ressources, le respect des temps de réponse contribuent à complexifier encore davantage le problème. Dans le même temps, la pression d'un secteur concurrentiel conduit à chercher systématiquement à minimiser le temps de mise sur le marché de nouveaux produits. On doit ainsi chercher à concilier des aspects de nature très hétérogène dont les interactions ne sont pas toujours bien maîtrisées. Des méthodes, plus ou moins formalisées, ont été développées pour la spécification, la modélisation, la conception et le déploiement de projets informatiques. La programmation orientée objet a pu sembler un certain temps comme l'aboutissement de cette démarche censée procurer le cadre idéal pour affronter cette crise du logiciel. De fait cette technologie s'est bien développée en milieu industriel ; où elle s'appuie, il est vrai, sur un outillage conséquent. Néanmoins cette méthode ne favorise pas la réutilisation logicielle et des aspects comme la synchronisation, la communication à distance, la gestion de la mémoire, la persistance ne sont pas pris en compte de façon satisfaisante. L'introduction des Design Patterns [4], en permettant la réutilisation d'un

certain nombre de schémas génériques de conception, répond malgré tout en partie à ce besoin d'économiser l'investissement intellectuel par la réutilisation de réalisations antérieures. Du fait de ces limites de l'approche objet, un certain nombre de méthodologies, dites *post-objets*, ont été mises en avant avec l'émergence de notions telles que les contrats, les aspects, les composants, les langages dédiés, les fabriques logicielles, l'ingénierie dirigée par les modèles, la programmation générative, la programmation intentionnelle, etc. Martin Ward et Sergey Dmitriev [1, 2] ont mis en avant le concept général de *programmation orientée langage* pour unifier ces différentes approches, qui sont par ailleurs largement présentées dans le livre de Krzysztof Czarnecki et Ulrich W. Eisenecker [5].

De la programmation par composants à la programmation orientée langage

La programmation par composants [6] est une technique adaptée à la conception de systèmes logiciels complexes du fait qu'elle favorise, en la systématisant, la réutilisation des composants logiciels [7]. L'écriture d'un composant se fait par la réutilisation de composants existants en vue de sa propre réutilisation par d'autres composants. A la méthode de conception descendante, qui consiste à décomposer récursivement un problème en une suite de sous problèmes de plus en plus simples jusqu'à aboutir à des problèmes suffisamment élémentaires pour pouvoir être résolus directement, nous superposons une approche ascendante dans laquelle on conçoit des systèmes de plus en plus complexes par réutilisation systématique de composants jusqu'à ce que le problème initialement posé trouve une solution d'une complexité raisonnable en utilisant les composants disponibles « sur l'étagère ». Plutôt que de construire un système donné et immuable on cherche ainsi à mettre en place une « boîte à outils » permettant de développer toute une gamme de produits logiciels concernant un même domaine applicatif. On obtient ainsi des solutions plus facilement adaptables aux changements et l'investissement intellectuel se trouve mieux préservé.

En contrepartie cette méthode nécessite qu'une analyse du domaine (espace du problème) soit effectuée afin d'identifier les composants élémentaires caractéristiques du domaine d'application qui offrent le minimum de redondances et d'interférences et permettent la plus grande combinatoire possible (espace des solutions). Les éléments de cet espace de solution sont ainsi des abstractions de haut niveau, réutilisables et liées à un domaine spécifique (encapsulation de savoir-faire métiers). Une telle abstraction peut être localisable (un composant) ou non (un aspect).

Pour éviter les redondances, les composants doivent être génériques ; cette généricité peut se présenter, selon le cas, sous la forme de paramétrisation, d'héritage, de polymorphisme, etc. L'espace de solution d'un domaine spécifique d'application peut être capturé par un langage dédié (DSL pour *Domain-Specific Language* [8, 9]) qui encapsule la sémantique du domaine (ni plus ni moins). Ces langages sont à même de capturer les caractéristiques communes à une famille de programmes et leur conception peut donc être le résultat de la phase d'analyse du domaine. Un programme dans un tel langage dédié permet d'identifier un composant particulier de cette famille. Un tel langage peut être utilisé directement par un expert du domaine auquel on n'impose pas les contingences et

la maîtrise d'un langage de programmation généraliste. L'approche est ascendante : on définit d'abord un langage qui capture les caractéristiques du domaine puis on écrit les applications visées en utilisant ce langage. C'est cette approche qui est connue sous le nom de *programmation orientée langage* [1].

Vers une ingénierie des langages dédiés

Néanmoins la conception et l'implémentation d'un langage, même si celui-ci est de portée restreinte, est une entreprise complexe. Il faut en effet concevoir un compilateur ce qui nécessite de prendre en compte des étapes d'analyse, de vérification de types, de génération de code, d'optimisation, de gestion des erreurs sans parler d'éventuels outils annexes comme un éditeur structuré, un générateur de documentation, etc. Par ailleurs un langage dédié peut être amené à évoluer fréquemment en fonction de son domaine d'application. Il est très difficile de faire évoluer un compilateur, faudra-t-il tout recommencer à chaque révision majeure ? Enfin il peut être difficile de faire collaborer différents langages dédiés or cela sera rapidement indispensable dans des applications réalistes qui incorporent différents domaines d'expertises.

Nous souhaitons concevoir un métalangage pour définir, générer, composer et optimiser les composants de la façon la plus automatique possible. On parle alors de programmation générative, ou de *métaprogrammation*, ce qui peut être vu comme un effort de formalisation linguistique (la conception de langages vue comme méthode de programmation). Les différents mécanismes d'abstraction présents dans les langages de programmation (fonctions, procédures, modules, classes, objets, agents, etc.) peuvent déjà être perçus comme des moyens d'étendre le langage hôte sur le plan linguistique : programmer une application passe de cette façon par la création d'un langage interne. En poursuivant dans cette logique on peut être conduit à créer explicitement des langages dédiés plongés dans un langage hôte (DSEL : *Domain-Specific Embedded Languages*). L'utilisation d'un langage hôte répond en grande partie aux objections précédentes : on hérite du langage hôte les parties non spécifiques du domaine, on hérite le compilateur et les outils divers associés au langage hôte, plusieurs DSLs peuvent partager des notations communes (e.g. les monades en Haskell) ce qui peut faciliter leur inter-compréhension (*uniform look and feel*) et enfin les DSLs sont intégrés à un langage généraliste et le sont aussi entre eux via ce langage hôte. Un inconvénient de cette approche est que la syntaxe des langages dédiés se trouve limitée par les contraintes syntaxiques du langage hôte. L'utilisation conjointe en Haskell de la surchage d'opérateurs (via la notion de classe), des fonctions d'ordre supérieur et du polymorphisme offre néanmoins une certaine souplesse dans les notations.

La programmation orientée langage utilise les langages dédiés pour encapsuler des connaissances métier (*programming knowledge*) sous une forme facile à utiliser par des spécialistes du domaine. Nous aurons généralement besoin de faire coexister un grand nombre de langages dédiés dans un même projet. Le système de typage du langage hôte sous-jacent (Haskell dans notre cas) pourrait ne pas être suffisant pour assurer qu'un programme utilisant des notations issues de différents langages dédiés s'exécutera toujours correctement. Nous donnerons une condition pour que non seulement un programme d'un langage dédié puisse être simulé de façon correcte par l'expression Haskell qui lui est

associée mais aussi de telle sorte que toute *macro-expression* de ce langage (c'est à dire une expression avec des paramètres) ne donne aucune erreur à l'exécution lorsqu'on substitue aux paramètres dans l'expression Haskell correspondante des expressions ayant le type associé à ces paramètres (expressions qui peuvent utiliser des macro-instructions provenant d'autres langages dédiés).

Pour plonger un langage dédié dans un langage hôte deux méthodes sont *a priori* possibles : soit on écrit une partie frontale pour traduire un programme (ou une macro instruction) en un programme Haskell, soit on implémente le langage dédié directement à l'aide d'un jeu de combinateurs en Haskell. Dans le second cas une expression du langage dédié est directement un programme Haskell valide et aucune étape de précalcul n'est nécessaire. C'est cette seconde méthode qui est utilisée par exemple dans Parsec (http ://legacy.cs.uu.nl/daan/parsec.html) pour implémenter un langage dédié pour la construction d'analyseurs syntaxiques. La syntaxe de ce langage est similaire à la syntaxe BNF pour décrire des grammaires algébriques étendues (les parties droites des productions sont des expressions régulières). Une expression de ce langage décrivant une grammaire donnée est un combinateur Haskell qui procure un analyseur reconnaissant les entrées conformes à cette grammaire et produisant en résultat un arbre de syntaxe abstraite. Cette approche présente un certain nombre d'avantages en comparaison d'outils tels que Lex et Yacc :

• Pas besoin d'une partie frontale pour traduire l'expression représentant la grammaire en le code de l'analyseur correspondant. Cette expression est directement construite à partir de combinateurs de base qui sont des fonctions représentant des analyseurs élémentaires ou des mécanismes d'assemblage de tels analyseurs.

• Ce langage de combinateurs d'analyse est extensible. Il peut être enrichi par adjonction de nouveaux combinateurs chaque fois que des schémas génériques dans l'écriture d'analyseurs sont identifiés. Par exemple on peut ajouter un combinateur pour permettre d'éviter de transformer une grammaire récursive à gauche : on analyse sous la forme d'une récursion à droite mais on présente le résultat en associant à gauche. De la même façon on peut définir des combinateurs spécifiques à des formats d'entrée particuliers (par exemple pour analyser des documents XML). Ce langage est donc facilement adaptable à un domaine particulier pour permettre à l'utilisateur de disposer à chaque fois du langage le plus adapté à son domaine avec une économie dans les notations. Le code est ainsi plus concis, et plus compréhensible et donc plus facile à maintenir.

• Un analyseur peut être paramétré par une algèbre de telle sorte qu'au lieu de produire un arbre de syntaxe abstraite on obtient l'interprétation de cet arbre dans cette algèbre. On peut ainsi très facilement écrire des traductions dirigées par la syntaxe de façon modulaire. Par ailleurs nous montrerons comment les algèbres permettant de paramétrer un tel analyseur pourront être décrites dans un autre DSL (basé sur des grammaires attribuées). Cet exemple est simple mais illustre bien les avantages de l'approche orientée langage, aussi nous en donnerons une description précise dans ce document.

Approche méthodologique

Dans notre travail nous avons opté tout naturellement pour cette seconde solution fondée sur l'utilisation de combinateurs fonctionnels en suivant l'approche préconisée par S.

Doaitse Swierstra, Pablo R. & al. [10]. Il est question de concevoir un formalisme pour la spécification d'un langage dédié, c'est-à-dire une forme de *métalangage* : un langage dédié permettant de concevoir des langages dédiés. Sans surprise le formalisme des grammaires attribuées, introduit par Knuth [11] dans les années soixante et grandement utilisé depuis pour l'écriture de compilateurs, s'est avéré parfaitement adapté à ce propos. En suivant la même démarche que Kuiper & Swierstra [12], nous tirons partie d'une implémentation fonctionnelle des grammaires attribuées [13–15] pour montrer comment utiliser les grammaires attribuées comme des spécifications exécutables de langages dédiés. Concrètement chaque production de la grammaire est traduite en un combinateur fonctionnel permettant de construire la fonction d'interprétation du symbole en partie gauche de la production en terme de fonctions d'interprétation des symboles en partie droite. Les fonctions d'interprétation associées à un symbole sont les fonctions de ses attributs hérités vers ses attributs synthétisés. Nous observons que cette traduction s'étend aux fonctions monadiques, c'est à dire lorsque les fonctions sémantiques de la grammaire attribuée sont composées dans la catégorie de Kleisli de la monade considérée. Cela permet de définir des langages dédiés avec des effets de bord (non déterminisme, manipulation d'un état local, interactions avec les entrées sorties standards, gestion des erreurs, émissions de traces d'exécutions, etc.). À titre d'illustration nous montrons comment le langage dédié des analyseurs syntaxiques peut être défini à l'aide d'une grammaire attribuée dont les règles sémantiques sont des calculs non déterministes (c'est-à-dire utilisant la monade des listes). Les fonctions sémantiques peuvent aussi être écrites directement dans un langage dédié plongé dans Haskell, plutôt que dans le langage Haskell lui même. Nous en donnons une illustration en utilisant le langage FAL (Functional Animation Language [16]) pour écrire les fonctions sémantiques de la grammaire attribuée : ce qui permet de concevoir un langage dédié pour l'assemblage de composants réactifs. Sur la base de ces différentes façons de définir et réutiliser un langage dédié qui s'appuient essentiellement sur le formalisme des grammaires attribuées, nous proposons un typage d'une certaine classe de langages dédiés (que nous définissons au préalable), en vue de leur réutilisation et assemblage. Nous illustrons tout cela au dernier chapitre par une étude approfondie et détaillée de la conception d'un langage dédié pour l'édition de documents structurés. Nous donnons dans le paragraphe qui suit quelques détails sur le contenu et l'organisation des différents chapitres de ce manuscrit de thèse.

Plan du document

La suite de ce document comporte trois chapitres principaux suivis d'une conclusion dans laquelle nous faisons un bilan du travail réalisé et présentons quelques pistes de réflexion pour les travaux futurs.

Chapitre 1 : Grammaires attribuées comme spécifications exécutables de langages dédiés. Dans ce chapitre introductif nous donnons une présentation succincte des notions de signatures multi-sortes, de grammaires abstraites et attribuées. Nous y relevons chaque fois que cela s'avère nécessaire, les aspects du langage fonctionnel Haskell qui en font un bon candidat pour servir d'hôte pour l'implémentation de ces différentes

notions et des langages dédiés. En particulier, nous donnons une présentation du mécanisme d'évaluation paresseuse, des structures de données inductives avec les notions relatives d'algèbres associées. Le choix d'un langage fonctionnel ne doit pas surprendre puisque la sémantique dénotationnelle d'un langage de programmation n'est rien d'autre qu'une implémentation de ce langage dans le λ-calcul et qu'un langage fonctionnel peut être vu comme le λ-calcul édulcoré par du sucre syntaxique. Nous introduisons ensuite une notation commune pour les algèbres et les grammaires attribuées, basée sur le calcul des séquents. Cette notation permet de décrire plus facilement la sémantique opérationnelle des programmes que nous paramétrons par des algèbres. Puis nous présentons une traduction des règles sémantiques d'une grammaire attribuée en une algèbre pour la signature sous-jacente. Cette présentation reprend dans l'esprit les constructions de K. Backhouse & T. Johnsson [13,14] ; elle s'inspire également des travaux de B. Mayol, Chirica & al. [17,18] dans leur volonté de produire une sémantique par point fixe des grammaires attribuées. On peut la traduire de façon presque littérale dans le langage fonctionnel Haskell car le mécanisme d'évaluation paresseuse permet de contourner l'apparente cyclicité du programme ainsi obtenu [19] tant que la grammaire attribuée elle-même est non circulaire. Nous donnons un exemple d'un langage d'assemblage de boîtes pour illustrer la définition d'un langage dédié à base de combinateurs définis par une grammaire attribuée.

Chapitre 2 : Vers une méthodologie pour le développement de langages dédiés. Nous voulons nous baser sur la technique de construction de langages dédiés à partir de grammaires attribuées présentée au chapitre 1, afin de dégager un certain nombre de méthodes pour l'assemblage et la réutilisation de langages. La volonté est de conduire la même démarche méthodologique au niveau des langages que ce qui est classiquement fait au niveau des composants logiciels : Comment peut-on créer de nouveaux langages par composition de langages réutilisables existants [20] ? Il faut donc préciser les caractéristiques des langages et les informations (typage) qu'il faut y attacher pour qu'un langage dédié soit réutilisable. La question est complexe car la réutilisation d'un langage dédié peut mettre en jeu des aspects très différents. Nous illustrons cette notion de réutilisabilité à travers quelques exemples représentatifs à partir desquels nous essayons de tirer des règles générales. Le problème a plusieurs dimensions, et nous pouvons en identifier trois principales. Les deux premières se fondent sur les travaux de Katsumata [21] qui nous indiquent que notre traduction de grammaires attribuées en algèbres fonctionne non seulement dans la catégorie des CPOs (ordres partiels complets) et des fonctions continues, mais aussi dans des catégories plus générales comme la catégorie de Kleisli d'une monade (ce qui nous permet de prendre en compte dans la section 2.2 les langages dédiés dont les programmes ont des effets), ou la catégorie des calculs associés à un autre langage dédié que nous présentons dans la section 2.3. La troisième dimension que nous illustrons encore dans la dernière partie de la section 2.2 concerne l'extension d'un langage dédié par adjonction de nouveaux aspects à la grammaire attribuée qui décrit ce langage. Nous proposons enfin dans la section 2.4 un typage de langages dédiés pour la conception de langages vue comme méthode de programmation.

Chapitre 3 : Étude de cas : un DSL pour l'édition de documents structurés Ce chapitre traite d'un exemple significatif de conception d'un langage dédié à base de

combinateurs fonctionnels d'éditeurs, utilisant la méthode présentée dans le chapitre précédent. Son étude fait intervenir différents aspects (programmes avec effets, assemblage de composants, etc.) présentés dans les chapitres précédents. Le problème de conception et d'implémentation des éditeurs dirigés par la syntaxe n'est pas nouveau et une vue d'ensemble de la question a été présenté dans [22]. Notre façon d'aborder le sujet est inspirée du langage *Modeless structure editor* de B. Sufrin & O. De Moor [23], un langage simple d'édition structurée.

Une donnée structurée est représentée intentionnellement sous la forme d'une structure arborescente caractérisée par une grammaire algébrique abstraite. On associe des attributs à cette structure pour répondre aux questions sémantiques [24, 25]. La manipulation de telles données consiste essentiellement à modifier de façon dynamique et interactive leur structure ; ce qui entraîne éventuellement une réévaluation totale ou partielle des attributs. On conçoit donc un éditeur de données structurées comme un programme interactif qui réagit aux actions de l'utilisateur (pression d'une touche du clavier, clic de la souris, insertion d'une production, copier-coller, ...) pour modifier les objets édités et produire incrémentalement un résultat. Par ailleurs, la représentation interne d'un document dans un éditeur interactif doit pouvoir être localisable (grâce à un point focal représentant à tout moment un nœud du document sur lequel les actions courantes d'édition s'effectuent) et partiellement définie (puisque le document en cours d'édition reste incomplet). Afin de pouvoir permettre cette localisation, nous représentons le document structuré par un Zipper, une structure de données introduite par Gérard Huet [3] permettant de représenter un arbre et son contexte, afin de faciliter la navigation dans le document, mais aussi pour pouvoir appliquer les actions d'édition directement à l'endroit indiqué dans le document. Nous donnons à la fin de ce chapitre des applications concrètes de ce langage d'éditeurs.

Chapitre 1

Grammaires attribuées comme spécifications exécutables de langages dédiés

Sommaire

Depuis leur introduction en 1968 par D. Knuth [11], les grammaires attribuées ont été largement étudiées et continuent d'être l'objet de nombreux travaux de recherche. Conçues à l'origine comme une méthode de spécification de la sémantique des langages de programmation, elles constituent de nos jours une manière convenable de spécification des traductions dirigées par la syntaxe, en particulier dans l'écriture des compilateurs. Dans leur définition d'origine [11,26], les grammaires attribuées combinaient à la fois la syntaxe concrète du langage, donnée par une grammaire algébrique, et les règles sémantiques permettant de définir la valeur des attributs attachés aux différentes catégories syntaxiques. On peut néanmoins de nos jours séparer l'interprétation (évaluation des attributs) de l'analyse syntaxique en définissant directement les règles sémantiques sur les arbres de syntaxe abstraite.

Dans ce chapitre après avoir défini les notions de signature multi-sortes, de grammaires abstraites et d'algèbres, nous considérons le problème d'inversion d'une liste (*reverse*) utilisant un paramètre d'accumulation et celui du calcul de la largeur arborescente et de la profondeur d'imbrication d'un arbre obtenu par l'analyse d'une expression bien parenthésée. Leurs premières solutions qui utilisent des algèbres appropriées sont assez édifiantes. Elles permettent de montrer l'intérêt de paramétrer un programme (un analyseur syntaxique par exemple) par une algèbre afin d'obtenir une gamme de produits logiciels. En même temps, on note à travers ces exemples la difficulté pour un programmeur de spécifier

concrètement des algèbres (i.e. de donner le code de leurs fonctions d'interprétation) pour répondre à une question particulière.

Nous définissons ensuite le formalisme des grammaires attribuées que nous utilisons pour reformuler des solutions « plus agréables » à nos exemples. Cette approche, déjà préconisée par Swierstra & al. [10], montre l'intérêt d'utiliser des grammaires attribuées pour spécifier aisément des algèbres. En effet penser en termes de grammaires attribuées est très utile pour l'écriture des fonctions assez compliquées.

Nous montrons enfin comment les règles sémantiques d'une grammaire attribuée peuvent être automatiquement converties en une structure d'algèbre pour la signature sous-jacente constituée des constructeurs des arbres de syntaxe abstraite (productions de la grammaire). Cette structure d'algèbre formée d'un ensemble de combinateurs fonctionnels dérivés à partir d'une grammaire attribuée constitue le langage dédié plongé dans Haskell (DSEL) pour la signature sous-jacente. On en déduit que le formalisme des grammaires attribuées peut être utilisé pour définir des langages dédiés et nous illustrons cela sur l'exemple d'un langage pour l'assemblage de boîtes positionnées.

1.1 Schéma de récursion associé à une signature

La structure logique des objets qui nous intéressent peut être représentée par une grammaire abstraite. Cette représentation peut être complétée par l'introduction des attributs attachés aux diverses catégories syntaxiques et on parle de grammaires attribuées abstraites (voir section 1.3). Les structures syntaxiques se décrivent à partir de types algébriques ou d'algèbres multi-sortes. Ces dernières sont 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. Un objet est alors représenté par un terme pour la signature correspondante.

1.1.1 Signature multi-sortes et grammaire abstraite

Définition 1.1. On appelle *signature (multi-sortes)* $\Sigma = (\mathcal{S}, \mathcal{OP})$ la donnée d'un ensemble fini $\mathcal{S}$ de sortes ou types de base et d'un ensemble fini de symboles d'opérateur $\mathcal{OP}$; chacun d'eux ayant un type qui est un élément de $\mathcal{S}^* \times \mathcal{S}$.

Un opérateur (ou symbole d'opérateur) op de type $X_{op(1)}\ X_{op(2)}\ ...\ X_{op(|op|)} \rightarrow X_{op(0)}$ est dit d'arité $\alpha(op) = X_{op(1)}\ X_{op(2)}\ ...\ X_{op(|op|)} \in \mathcal{S}^*$ et de sorte $\sigma(op) = X_{op(0)} \in \mathcal{S}$. Il sera noté $op : X_{op(1)} \times X_{op(2)} \times ... \times X_{op(|op|)} \rightarrow X_{op(0)}$ ou sous sa forme currifiée $op : X_{op(1)} \rightarrow X_{op(2)} \rightarrow ... \rightarrow X_{op(|op|)} \rightarrow X_{op(0)}$ dans laquelle $X_{op(i)}$ pour $i \in \{1, ..., |op|\}$ est la catégorie syntaxique du i^e argument de op et, $X_{op(0)}$ la catégorie syntaxique de son résultat. Le rang de l'opérateur op est la longueur de son arité : $\rho(op) = |\alpha(op)|$. Si l'arité de op est égale à ϵ, c'est-à-dire que si $|op| = 0$, on dit que op est une *constante* de sorte $\sigma(op)$. La signature $\Sigma = (\mathcal{S}, \mathcal{OP})$ constitue ce qu'on appelle une *syntaxe abstraite*. On parle aussi parfois de types algébriques ou d'algèbres multi-sortes. Un élément est alors représenté par un terme pour cette signature, qu'on appelle syntaxe abstraite ou structure logique de cet élément. La (ou une) syntaxe concrète est donnée par le biais d'une grammaire algébrique.

Nous ne nous intéressons pas ici à la syntaxe concrète et considérons par conséquent les grammaires abstraites, i.e des grammaires algébriques sans symboles terminaux (ce qui est équivalent au formalisme des signatures multi-sortes avec un axiome). Les structures de données sont habituellement définies de manière mutuellement récursive comme point fixe d'un système d'équations. Nous nous limitons aux systèmes d'équations polynomiales que nous présentons à l'aide des grammaires abstraites.

Définition 1.2. Une *grammaire 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 mises en jeu, d'un symbole grammatical particulier $A \in \mathcal{S}$ appelé axiome, et d'un ensemble fini $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{S}^*$ de productions.

Une production $P = (X_{P(0)},\ X_{P(1)}\ X_{P(2)}\ ...\ X_{P(|P|)})$ est notée $P : X_{P(0)} \rightarrow X_{P(1)}\ X_{P(2)}\ ...\ X_{P(|P|)}$ dans laquelle $|P|$ désigne la longueur de la partie droite de la production P. À une grammaire abstraite $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$, on associe une signature $\Sigma_{\mathbb{G}} = (\mathcal{S}, \mathcal{OP})$ dont les sortes sont les symboles grammaticaux et les opérateurs les productions de la grammaire. Une grammaire abstraite est alors une signature multi-sortes à laquelle on a ajouté un symbole particulier appelé axiome. Une production $P : X_{P(0)} \rightarrow X_{P(1)}\ X_{P(2)}\ ...\ X_{P(|P|)}$ dans $\mathbb{G}$ est vue comme un opérateur $p : X_{P(1)} \times\ X_{P(2)} \times\ ...\ \times\ X_{P(|P|)} \rightarrow X_{P(0)}$ dans $\Sigma_{\mathbb{G}}$. Un symbole $X \in S$ n'apparaissant pas en partie gauche d'aucune production est appelé un *paramètre* de la grammaire, l'axiome ne doit pas être un paramètre. Ainsi les arbres de syntaxe abstraite de $\mathbb{G}$ (dont les nœuds sont étiquetés par les productions de la grammaire) sont effectivement les termes de la signature sous-jacente $\Sigma_{\mathbb{G}}$.

1.1.2 Représentation en Haskell

À partir d'une grammaire abstraite ou d'une signature multi-sortes, on déduit de manière systématique les types de données Haskell correspondants comme suit :

1. On associe un type de données à chaque catégorie syntaxique X_i qui apparaît comme le type du résultat d'au moins un opérateur (type défini).
2. On associe ensuite un constructeur $Op :: X_{op(1)} \rightarrow X_{op(2)} \rightarrow ... \rightarrow X_{op(|op|)} \rightarrow X_{op(0)}$ à chaque opérateur *op* de $\mathcal{OP}$.
3. Les catégories syntaxiques qui n'apparaissent pas comme le type du résultat d'aucun opérateur sont associées à des variables de types appelées paramètres.

Exemple 1 : Structure de données des listes. Les productions de la grammaire décrivant les éléments d'une liste de type a sont les suivantes :

$$\begin{array}{lcccl} Cons & : & \langle L \rangle & \rightarrow & a\ \ \langle L \rangle \\ Nil & : & \langle L \rangle & \rightarrow & () \end{array}$$

On en déduit la structure de données Haskell ci-dessous, dans laquelle on a associé à l'unique catégorie syntaxique `L` le type de données paramétrique `List a` et aux productions `Cons` et `Nil` respectivement les constructeurs `Cons` et `Nil`. Le type `List a` est

équivalent au type prédéfini `[a]` du prélude standard de Haskell avec [] pour `Nil` et (:) pour `Cons`.

```
data List a = Nil | Cons a (List a)
```

Exemple 2 : Structures de données des arbres. Cet exemple met en jeu plusieurs catégories syntaxiques. On définit une grammaire abstraite $\mathbb{T} = (\mathcal{S}, \mathcal{P}, Tree)$ des arbres et forêts dans laquelle $\mathcal{S} = \{Tree, Forest, Label\}$ est l'ensemble des symboles grammaticaux, $Tree$ est l'axiome, et on dispose de l'ensemble $\mathcal{P} = \{Node, Nil, Cons\}$ des productions définies comme suit avec $Label$ comme unique paramètre de la grammaire :

$$\begin{array}{lcl} Node & : & \langle Tree \rangle \rightarrow \langle Label \rangle \ \langle Forest \rangle \\ Nil & : & \langle Forest \rangle \rightarrow () \\ Cons & : & \langle Forest \rangle \rightarrow \langle Tree \rangle \ \langle Forest \rangle \end{array}$$

La grammaire des arbres se traduit en Haskell par la définition récursive des structures de données ci-dessous dans laquelle le paramètre *Label* de la grammaire devient un paramètre (*label*) pour les constructeurs de types associés aux (autres) symboles grammaticaux[1] :

```
data Tree label = Node label (Forest label)
data Forest label = Nil | Cons (Tree label) (Forest label)
```

1.1.3 Interprétation d'un terme associé à une signature

Les éléments finis (termes) associés à une signature se définissent comme le plus petit ensemble tel que :

- Une constante $c : () \rightarrow s$ est un terme de sorte s.
- Si $op : s_{op(1)}\ s_{op(2)}\ ...\ s_{op(n)} \rightarrow s_{op(0)}$ est un opérateur et $t_i \in T(\Sigma)_{s_{op(i)}}$ alors $op\ (t_1, t_2, ..., t_n) \in T(\Sigma)_{s_{op(0)}}$.

Les termes d'une structure de données sont régulièrement sujets à plusieurs interprétations. Par exemple, le terme *lst = Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Nil))))* appartenant au type de données *List a* peut être interprété pour calculer son plus grand élément, sa taille (nombre d'éléments), la somme ou le produit de ses éléments, son inverse, etc. De tels interpréteurs peuvent se définir directement par récursion comme indiqué dans le tableau 1.1 où *concatList* est la fonction de concaténation des listes de type *List a*, équivalente à la fonction (++) du prélude standard de Haskell.

On remarque que les fonctions *sumList, prodList*, *maxList* et *reverse* suivent le même schéma de récursion dans lequel f fait correspondre la valeur e à la liste vide, la valeur associée à une liste non vide est une certaine fonction $\oplus$ appliquée à sa tête et sa liste résiduelle.

$$\begin{array}{lcl} f\ \ Nil & = & e \\ f\ \ (Cons\ x\ xs) & = & x \oplus f\ xs \end{array}$$

C'est pourquoi on préfère le plus souvent spécifier ces interpréteurs au moyen des algèbres.

[1] En Haskell les noms de types ou de constructeurs de types ou de données doivent commencer par une lettre majuscule et les noms des variables, et en particulier des paramètres, par des lettres minuscules.

Tableau 1.1 Interprétation directe des termes du type *List a*

```
-- somme des éléments d'une liste d'entiers
 sumList :: List Int -> Int
 sumList Nil = 0
 sumList (Cons x xs) = x + (sumList xs)

-- produit des éléments d'une liste d'entiers
 prodList :: List Int -> Int
 prodList Nil = 1
 prodList (Cons x xs) = x * (prodList xs)

-- plus grand élément d'une liste d'entiers
 maxList :: List Int -> Int
 maxList Nil = 0
 maxList (Cons x xs) = x `max` (maxList xs)

  -- inverse d'une liste
 reverse :: List a -> List a
 reverse Nil = Nil
 reverse (Cons x xs) = (reverse xs) `concatList` (Cons x Nil)
```

Ce qui permet d'encapsuler ce motif de récursion dans une fonction d'évaluation[2] avec les fonctions $\oplus$ et e passées en arguments.

1.2 Algèbres et évaluation

Les algèbres représentent une façon très commode de décrire des fonctions qui "consomme" des arbres de syntaxe abstraite : on définit d'abord la structure d'algèbre associée à la grammaire (abstraite) sous-jacente, puis on décrit le schéma de récursion (ensemble des fonctions d'évaluation) pour cette algèbre. Chaque solution est obtenue en spécifiant convenablement les fonctions d'interprétation de la structure d'algèbre.

1.2.1 Algèbre associée à une signature

Définition 1.3. Une *algèbre* $\mathcal{A}$ associée à la signature $\Sigma = (\mathcal{S}, \mathcal{OP})$ est la donnée d'un domaine d'interprétation A_s associé à chaque sorte $s \in \mathcal{S}$ et d'une application $op^{\mathcal{A}} : A_{s_{op(1)}} \times ... \times A_{s_{op(n)}} \rightarrow A_{s_{op(0)}}$ associée à chaque opérateur $op : s_{op(1)} \times ... \times s_{op(n)} \rightarrow s_{op(0)}$ dans $\mathcal{OP}$.

L'interprétation d'un terme $t = op\ (t_1, ..., t_n)$ selon une algèbre $\mathcal{A}$ (notée $t^{\mathcal{A}}$) est définie inductivement par $t^{\mathcal{A}} = (op\ (t_1, ..., t_n))^{\mathcal{A}} = op^{\mathcal{A}}((t_1)^{\mathcal{A}}, ..., (t_n)^{\mathcal{A}})$.

[2] Cette fonction d'évaluation paramétrée par une algèbre est équivalente à la fonction *foldr* du prélude standard pour l'interprétation des listes.

On en déduit un type de données Haskell pour toute algèbre associée à une signature. Les paramètres de ce type sont les différents domaines d'interprétation de l'algèbre et les fonctions d'interprétation de l'algèbre sont les sélecteurs de l'unique constructeur de données de ce type.

Algèbre des listes

Les deux constructeurs du type *List* de types $Nil :: List\ a$ et $Cons :: a \rightarrow List\ a \rightarrow List\ a$ sont remplacés par les fonctions d'interprétation correspondantes $nil :: b$ et $cons :: a \rightarrow b \rightarrow b$ où b et a représentent respectivement les domaines d'interprétation du type $List\ a$ et du paramètre a. On en déduit, comme indiqué dans le tableau 1.2, le type de données des algèbres de listes ($AlgList\ a\ b$)[3] et la fonction d'évaluation des listes (*evalList*) paramétrée par cette algèbre. Les interpréteurs *sumList, prodList* et *maxList* s'obtiennent en spécifiant les bonnes fonctions d'interprétation.

Tableau 1.2 Interprétation des termes du type *List* par une algèbre

```
-- type de données des algèbres de listes
data AlgList a b = AlgList { nil  :: b,
                             cons :: a -> b -> b}

-- fonction d'évaluation d'une liste paramétrée par l'algèbre des listes
evalList :: AlgList a b -> List a -> b
evalList alg = f where f Nil = nil alg
                       f (Cons x xs) = cons alg x (f xs)

-- somme des éléments d'une liste d'entiers
sumList = evalList (AlgList 0 (+))

-- produit des éléments d'une liste d'entiers
prodList = evalList (AlgList 1 (*))

-- plus grand élément d'une liste d'entiers
maxList = evalList (AlgList 0 max))

-- inverse d'une liste
reverse = evalList (AlgList Nil fcons)
                   where fcons = (\x xs -> xs `concatList` (Cons x Nil))
```

Algèbre des arbres

Soient a le domaine d'interprétation du paramètre *label,* b et c ceux respectivement des types *Tree* et *Forest*. Les différents constructeurs (*Node, Nil* et *Cons*) sont remplacés par

[3] Chaque sélecteur d'un type de données Haskell prend ce type comme son premier argument. Ce qui justifie la présence de l'argument *alg* dans l'appel de *nil* et *cons*.

les fonctions d'interprétation correspondantes (*node, nil* et *cons*) et on obtient le listing du tableau 1.3.

Tableau 1.3 Interprétation des termes du type *Tree* par une algèbre

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

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 (Cons t f) = cons alg (evalTree alg t) (evalForest alg f)
```

1.2.2 Une notation pour les algèbres

Nous avons défini à la section 1.1.1 une grammaire abstraite comme une signature multi-sortes avec un axiome précisant le symbole qu'on s'attend à trouver en la racine d'un arbre de syntaxe abstraite associé à un terme de cette signature. Cette définition correspond à ce que nous avons appelé dans [27] grammaire enracinée (*rooted grammar* en anglais). De manière similaire on peut définir la notion d'algèbre enracinée (*rooted algebra*) en ajoutant à l'algèbre associée à une signature multi-sortes une fonction *return* permettant d'extraire le résultat en la racine. Les algèbres opèrent sur les arbres de syntaxe abstraite comme des automates déterministes descendants d'arbres, ou si l'on préfère comme des transducteurs d'arbres descendants [20]. Les états d'un tel automate représentent les différentes variables qui permettent de synthétiser les résultats des calculs effectués par l'automate.

Définition 1.4. Un *automate déterministe descendant d'arbres* est la donnée d'un quadruplet $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ défini sur un alphabet Σ de symboles avec arité, dans lequel Q est un ensemble fini d'états , $\delta \subseteq Q \times \Sigma \rightarrow Q^*$ la relation de transition, et q_0 l'état initial.

Un arbre t est défini par un ensemble (fini) $dom(t) \subseteq \mathbb{N}^*$, clos par préfixe tel que si $u \cdot i \in dom(t)$ et $1 \leq j \leq i$ alors $u \cdot j \in dom(t)$. Le noeud à l'adresse $v \cdot i$ est le i-ème fils du noeud à l'adresse v. Un arbre t étiqueté dans Σ est une fonction $l(t) : dom(t) \rightarrow \Sigma$ telle que si v a i fils, alors $l(t)$ est un symbole de Σ d'arité i. On écrit aussi t à la place de $l(t)$. On note souvent $T(\Sigma)$ l'ensemble des arbres étiquetés dans Σ. Les éléments de Σ sont les étiquettes des nœuds des arbres sur lesquels les calculs de l'automate sont effectués. Un calcul de l'automate $\mathcal{A}$ sur un arbre t est un étiquetage $e : dom(t) \rightarrow Q$ tel que :

- $e(\epsilon) = q_0$: on associe l'état initial à la racine de l'arbre.
- Soit $v \in dom(t)$ un nœud d'étiquette a (d'arité j) $\in \Sigma$, et de fils $v_1, \ldots, v_j$. Alors $e(v) = q$ et $e(v_i) = q_i$, pour $q, q_i \in Q (1 \leq i \leq j)$ tels que $(q, a, <q_1, \ldots, q_j>) \in \delta$

Un calcul e est acceptant (on dit aussi que l'arbre est reconnu) si on a pu associer à chacun des nœuds de l'arbre un état de telle sorte que la racine soit étiquetée par l'état initial q_0.

Nous proposons ici une notation pour les algèbres de ce type. Cette notation s'appuie sur la signature ou la grammaire sous-jacente. L'intérêt principal de cette présentation est d'aboutir à une notation commune avec les grammaires attribuées que nous présenterons dans la section 1.3. On associe donc à chaque symbole grammatical X une liste finie de variables $v_1,\ v_2,\ \ldots,\ v_n$. On ne précise pas de noms pour ces variables et on considère simplement que chaque variable est accessible via sa position dans la liste.

$$\langle X\rangle(v_1,\ v_2,\ \ldots,\ v_n)$$

Les règles de calcul qui permettent de donner une interprétation particulière aux arbres de syntaxe abstraite s'écrivent en associant à chaque symbole apparaissant dans le type d'un opérateur $op : X_1\ X_2\ ...\ X_n \rightarrow X_0$, une liste d'expressions utilisant des variables dont la portée est limitée à cet opérateur. Les positions d'entrée correspondent aux positions des variables des nœuds fils et les positions de sortie aux variables du nœud père. On impose que les expressions en positions d'entrée soient toutes des variables, que toutes ces variables soient distinctes, et qu'elles soient les seules qui apparaissent dans les expressions en positions de sortie. On a alors la notation suivante pour chaque opérateur :

$$op \ : \ \langle X_1\rangle(v_{11},\ \ldots,\ v_{1m_1})\ \ldots\ \langle X_n\rangle(v_{n1},\ \ldots,\ v_{nm_n}) \ \rightarrow \ \langle X_0\rangle(v_{01},\ \ldots,\ v_{0m_0})$$

qui décrit la structure d'algèbre donnant une certaine interprétation de la signature multi-sortes sous-jacente, et qu'on peut aussi écrire sous la forme d'un automate descendant d'arbres ou d'un transducteur. Par exemple l'algèbre enracinée permettant de calculer la somme des éléments d'une liste est donnée comme suit :

$$\begin{array}{lcll} return & : & \langle L\rangle(s) & \rightarrow \ \langle Rl\rangle(s) \\ nil & : & & \rightarrow \ \langle L\rangle(0) \\ cons_a & : & \langle L\rangle(s_1) & \rightarrow \ \langle L\rangle(a+s_1) \end{array}$$

Cette notation représente bien une structure d'algèbre (pour la grammaire sous-jacente dans laquelle on a ajouté la production racine $RootL \ : \ \langle Rl\rangle \ \rightarrow \ \langle \ L\rangle$) qu'on peut aussi écrire sous la forme d'un transducteur par :

$$\begin{array}{lcl} \mathbf{s}\ (Rl \quad xs) & \rightarrow & \mathbf{s}\ xs \\ \mathbf{s}\ Nil & \rightarrow & 0 \\ \mathbf{s}\ (Cons\ x\ xs) & \rightarrow & x + (\mathbf{s}\ xs) \end{array}$$

1.2.3 Programmer avec les algèbres : avantages et inconvénients

Les exemples présentés ci-dessus sont des illustrations assez simples de programmes paramétrés par des algèbres. Nous considérons dans cette section d'autres exemples plus significatifs pour montrer comment, programmer avec des algèbres offre une grande flexibilité dans l'écriture et le paramétrage de programmes. C'est une technique qui permet au programmeur d'obtenir une gamme variée de produits logiciels concernant un même domaine applicatif lié à une signature multi-sortes. On obtient ainsi des solutions (optimisées) qu'on peut facilement adapter aux changements. Toutefois, le problème qu'on rencontre le plus souvent lors de la spécification d'une algèbre est celui de la définition des fonctions d'interprétation décrivant les solutions appropriées aux problèmes posés. Les exemples qui suivent nous permettront de relever les principaux avantages et inconvénients liés à la programmation utilisant le paramétrage par des algèbres.

Interprétation d'un arbre donné par une expression bien parenthésée

Dans ce paragraphe, nous définissons en utilisant la notation présentée ci-dessus des exemples de programmes paramétrés par une algèbre, permettant d'interpréter un arbre donné par une expression bien parenthésée :

- Le premier programme appelé *nesting* permet de calculer sa profondeur d'imbrication de parenthèses. On compte les différentes imbrications de parenthèses dans l'arbre et on retourne la taille de la plus grande. On aura par exemple *nesting* "(()((()))()())()" = 4 dans laquelle la plus grande imbrication est donnée en bleu sur la figure 1.1.
- Et le deuxième (*breath*) donne sa largeur arborescente. C'est la longueur de la plus grande "structure de liste" (dont les éléments sont des blocs non imbriqués de parenthèses ouvrante et fermante) apparaissant dans l'arbre résultat de l'analyse. Ainsi les expressions *breath* "(()((()))()())()" = 3 (voir en rouge sur la figure 1.1) et, *breath* "((((()()))()(()())(()))()" = 4 dans lesquelles nous avons isolé (en rouge) la largeur arborescente i.e le plus grand nombre de blocs non imbriqués de parenthèses ouvrante et fermante.

La grammaire algébrique enracinée de telles expressions (bien parenthésées) est donnée par :

$$\begin{array}{lcll} Rt & : & \langle Rp \rangle & \rightarrow \quad \langle P \rangle \\ Nil & : & \langle P \rangle & \rightarrow \quad \epsilon \\ Bin & : & \langle P \rangle & \rightarrow \quad '(' \ \langle P \rangle \ ')' \ \langle P \rangle \end{array}$$

Nous abordons le problème en construisant dans un premier temps, un analyseur syntaxique qui produit l'arbre binaire *Rtree* correspondant à toute expression bien parenthésée.

```
data Rtree = Rt Btree
data Btree = Nil | Bin Btree Btree
```

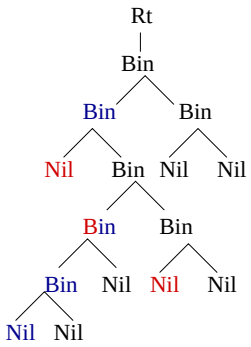

FIGURE 1.1 – Profondeur d'imbrication et largeur arborescente pour l'expression "(()((()))()())()"

Nous montrerons ensuite l'intérêt de paramétrer cet analyseur par une algèbre associée aux arbres binaires, offrant ainsi au programmeur une flexibilité dans l'analyse et

l'interprétation des expressions bien parenthésées. Il lui suffira à chaque fois de spécifier les fonctions d'interprétation appropriées.

Un analyseur (*parser*) se définit comme un programme qui prend en entrée une liste de lexèmes (tokens) et produit un arbre décoré par attributs ou une abstraction de ce dernier. Si s et a sont des variables de types représentant respectivement le type des tokens et celui des résultats produits, le type d'un analyseur devrait être le suivant :

```
type Parser s a = [s] -> a
```

Pour composer séquentiellement deux analyseurs, il faut admettre qu'un tel analyseur ne consomme qu'un préfixe du mot en entrée, qu'il produise un fragment de l'arbre de syntaxe abstraite et qu'il transmette à l'analyseur suivant la partie du mot non encore analysée. Par ailleurs même avec une grammaire non ambiguë il n'y a aucune raison pour que la reconnaissance d'un tel préfixe soit déterministe. On gère le non déterministe de façon paresseuse en utilisant à chaque étape la liste des alternatives. Le backtracking est ainsi géré de façon implicite par le mécanisme d'évaluation paresseuse. Cela nous conduit à considérer plutôt :

```
type Parser s a = [s] -> [(a,[s])]
```

L'interprétation en est la suivante : si $p :: Parser\ s\ a$ est un analyseur et $xs :: [s]$ une chaîne d'entrée alors, $p\ xs$ produit la liste des paires (t, zs) dans lesquelles t est un arbre de dérivation produit par p en lisant un préfixe ys de xs et zs est la chaîne résiduelle telle que $xs = ys\ {}_{++}\ zs$.

On veut reconnaître le plus grand préfixe qui corresponde à une écriture bien parenthésée et lui associer l'arbre binaire *Rtree* correspondant. En utilisant le schéma de compréhension des listes, l'analyseur décrit pour cette grammaire est donné par la fonction *paren* dans laquelle l'analyseur $symbol :: Char \to Parser\ Char\ ()$ se contente de reconnaître si la chaîne d'entrée commence par le caractère donné en argument :

```
-------------------------------------------------------------------------
  data Rtree = Rt Btree
  data Btree = Nil | Bin Btree Btree

  paren :: Parser Char Rtree
  paren xs = [(Rt (Bin t t'), vs) | (_,ys) <- symbol '(' xs,
                                    (t,zs) <- paren ys,
                                    (_,us) <- symbol ')' zs,
                                    (t',vs) <- paren us] ++ [(Rt Nil,xs)]
  symbol c [] = []
  symbol c (x:xs) = if c then [((),xs)] else []
-------------------------------------------------------------------------
```

Si on désigne par a et b les domaines d'interprétation des sortes *Btree* et *Rtree* respectivement, la structure d'algèbre enracinée correspondante est donnée par le type paramétré *AlgBtree* dans lequel *return*, *nil* et *bin* représentent respectivement les fonctions d'interprétation des constructeurs *Rt*, *Nil* et *Bin*. Le schéma de récursion associé à cette algèbre est donné par les fonctions *evalRt* et *evalBt*.

```
-------------------------------------------------------------------------
data AlgBtree a b = AlgBtree {return :: a -> b, nil :: a,
                                                 bin :: a -> a -> a}
evalRt :: AlgRtree a b -> Rtree -> b
evalRt alg@(AlgRtree return _ _) (Rt t) = return (evalBt alg t)

evalBt :: AlgRtree a b -> Btree -> a
evalBt (AlgRtree _ nil bin) = build
                  where build Nil = nil
                        build (Bin lft rgt) = bin (build lft) (build rgt)
-------------------------------------------------------------------------
```

Ainsi l'algèbre permettant de calculer la profondeur d'imbrication de parenthèses dans une expression bien parenthésée est donnée par :

$$\begin{array}{lcl} return & : & \langle P\rangle(nest) \rightarrow \langle Rp\rangle(nest) \\ nil & : & \rightarrow \langle P\rangle(0) \\ bin & : & \langle P\rangle(nest_1)\ \langle P\rangle(nest_2) \rightarrow \langle P\rangle(max\ (nest_1+1)\ nest_2) \end{array}$$

et celle permettant de calculer la largeur arborescente par :

$$\begin{array}{lcl} return & : & \langle P\rangle(l,p) \rightarrow \langle Rp\rangle(max\ l\ p) \\ nil & : & \rightarrow \langle P\rangle(0,0) \\ bin & : & \langle P\rangle(l_1,p_1)\ \langle P\rangle(l_2,p_2) \rightarrow \langle P\rangle(max\ (max\ l_1\ l_2)\ p_1, p_2+1) \end{array}$$

La traduction en Haskell est immédiate et est donnée par :

```
-------------------------------------------------------------------------
-- profondeur d'imbrication de parenthèses
nesting = (evalRt (AlgRtree id 0 g)) . fst . head . paren
        where g n m =  max (1+n) m

-- lageur arborescente
breath = (evalRt (AlgRtree res f g)) . fst . head . paren
        where res (l,p) = max l p
              f = (0,0)
              g (l,p) (l',p') = (max (max l l') p, p'+1)

-- nombre de parenthèses dans une expression
number = (evalRt (AlgRtree id 0 g)) . fst . head . paren
        where g n m = n+m+2
-------------------------------------------------------------------------
```

On note la flexibilité dans l'écriture et le paramétrage des programmes. Cependant la spécification de certaines algèbres nécessite un effort intellectuel non négligeable. C'est le cas des algèbres pour les programmes *nesting* et *breath* qu'il est difficile de dériver directement en suivant leurs définitions récursives.

Remarque 1.5. Une solution plus générale consiste à paramétrer directement l'analyseur par cette algèbre. Ce qui permet de retourner comme résultat de l'analyseur, non plus l'arbre d'analyse correspondant, mais directement son interprétation. L'analyseur syntaxique analyse et interprète le code source sans avoir besoin de produire l'arbre de syntaxe abstraite comme résultat intermédiaire. Cette technique d'optimisation dite de *déforestation*, a été largement traitée dans la thèse de E. Duris [28] qui a contribué à établir les relations entre grammaires attribuées et programmes fonctionnels. Ainsi l'analyseur *parenAlg* paramétré par l'algèbre *AlgRtree* est le programme permettant d'analyser et d'interpréter dans le domaine b toute expression bien parenthésée.

```
-------------------------------------------------------------------------
parenAlg1 :: AlgRtree a b -> Parser Char a
parenAlg1 alg@(AlgRtree _ nil bin) xs =
          [(bin t t', vs) | (c, ys) <- symbol '(' xs ,
                            (t, zs)<- parenAlg1 alg ys ,
                            (c', us)<- symbol ')' zs ,
                            (t', vs)<- parenAlg1 alg us ] ++ [(nil, xs)]

parenAlg :: AlgRtree a b -> Parser Char b
parenAlg alg@(AlgRtree result _ _) = f <$> parenAlg1 alg
                                          where f a = result a
-------------------------------------------------------------------------
```

Inversion d'une liste avec paramètre d'accumulation

La première solution proposée à la section 1.2 pour l'inversion d'une liste est assez coûteuse, puisque la fonction de concaténation parcourt la première liste chaque fois qu'elle est appelée pendant la récursion. Une meilleure solution consiste à utiliser un paramètre d'accumulation dont le schéma de récursion associé est donné par :

```
reverse xs = build xs Nil
       where build Nil ys = ys
             build (Cons x xs) ys = build xs (Cons x ys)
```

On observe que ce schéma de récursion est celui associé à l'algèbre enracinée des listes suivante :

```
-------------------------------------------------------------------------
data AlgListR a b c = AlgListR {return :: b -> c, nil    :: b,
                                                  cons   :: a -> b -> b}

algListRev = AlgListR return nil cons
    where return f = f Nil -- return :: (List a -> List a) -> List a
          nil = id         -- nil :: List a -> List a
          cons :: a -> (List a -> List a) -> (List a -> List a)
          cons x f ys = f (Cons x ys)
-------------------------------------------------------------------------
```

On voit ici que la présence du paramètre d'accumulation rend difficile la spécification directe de cette structure d'algèbre. La sorte *List* est interprétée sur le domaine de type fonction *List a -> List a*. Si l'on peut dériver cette algèbre directement à partir de la définition récursive, cela n'aurait pas été facile de l'écrire de but en blanc. Pour mieux interpréter cette algèbre, il faut remarquer que le paramètre d'accumulation porte de l'information héritée du contexte. Le résultat final qui est synthétisé par la fonction *return* dépend de cette information provenant du contexte. Il est par ailleurs assez difficile de représenter une telle information dans la nouvelle notation pour l'algèbre associée. Fort heureusement, cette dépendance entre informations provenant du contexte et informations synthétisées se décrit facilement à l'aide des règles sémantiques d'une grammaire attribuée.

Nous introduisons dans la section suivante le formalisme des grammaires attribuées qui généralise celui des algèbres et qui permet de spécifier des grammaires dont les domaines sont d'ordre supérieur, i.e. des fonctions des attributs hérités vers les attributs synthétisés.

1.3 Grammaires attribuées abstraites

Comme pour les grammaires algébriques, nous ne nous intéressons pas à la syntaxe concrète et considérons par conséquent les grammaires attribuées sans symboles terminaux.

Définition 1.6. Une *grammaire attribuée abstraite* $\mathbb{GA} = (\mathbb{G}, Attr, \mathcal{R})$ est la donnée d'une grammaire algébrique abstraite $\mathbb{G}$ dans laquelle chaque symbole non terminal X est associé à un ensemble d'attributs *Attr(X)* partitionnés en *attributs hérités Inh(X)* et *attributs synthétisés Syn(X)* tels que $Attr(X) = Inh(X) \uplus Syn(X)$; et d'un ensemble $\mathcal{R}$ de *règles sémantiques* associées aux différentes productions de la grammaire $\mathbb{G}$ qui indiquent les dépendances fonctionnelles entre les valeurs d'attributs des symboles non terminaux qui apparaissent dans la production.

Un nœud u d'un arbre de dérivation associé à un symbole non terminal X (u est dit une occurrence de X) a les mêmes attributs que X. Intuitivement, comme indiqué sur la figure 1.2, la valeur d'un attribut synthétisé d'un nœud ne dépend que du sous arbre issu de ce nœud, tandis qu'un attribut hérité de ce nœud ne dépend que du reste de l'arbre (son contexte). Un attribut synthétisé véhicule alors l'information depuis les feuilles d'un arbre de dérivation jusqu'à la racine tandis qu'un attribut hérité transporte l'information en sens inverse.

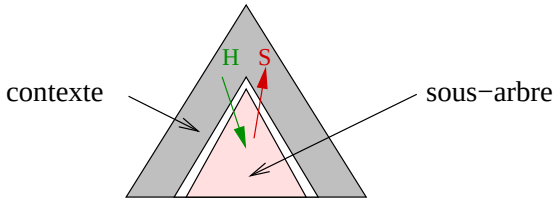

FIGURE 1.2 – Provenance des valeurs d'attributs d'un nœud u

Définition 1.7. Soit $P : X_{P(0)} \rightarrow X_{P(1)}\ X_{P(2)}\ ...\ X_{P(n)}$ une production d'une grammaire abstraite avec $n = |P|$. On appelle *attribut d'entrée* de la production P tout attribut qui est soit un attribut hérité de $X_{P(0)}$ (dont la valeur doit provenir du contexte) soit un attribut synthétisé d'un des $X_{P(i)}$ pour $1 \leq i \leq n$ (dont la valeur doit provenir du sous arbre correspondant). Les autres attributs, c'est-à-dire les attributs synthétisés de $X_{P(0)}$ et les attributs hérités des $X_{P(i)}$ pour $1 \leq i \leq n$ sont appelés *attributs de sortie* ou *attributs définis* de P.

Les règles sémantiques associées à la production P contiennent exactement une définition de la forme $a = f\,(b_1, \ldots, b_k)$ pour chaque attribut de sortie a de P et dans laquelle les b_i sont des attributs d'entrée de P, c'est-à-dire que chaque attribut de sortie dépend fonctionnellement des attributs d'entrée dont les valeurs sont fournies par l'environnement (le contexte) comme indiqué sur la figure 1.3. Certains attributs appelés *attributs locaux* ont une portée locale c'est-à-dire que leurs valeurs ne proviennent ni du contexte ni du sous-arbre. Ils sont parfois nécessaires pour calculer les valeurs d'autres attributs. Par ailleurs lorsqu'on s'intéresse à la syntaxe concrète d'une grammaire attribuée, la valeur associée à un symbole terminal est souvent donnée par un analyseur lexical et on parle alors d'*attributs lexicaux*.

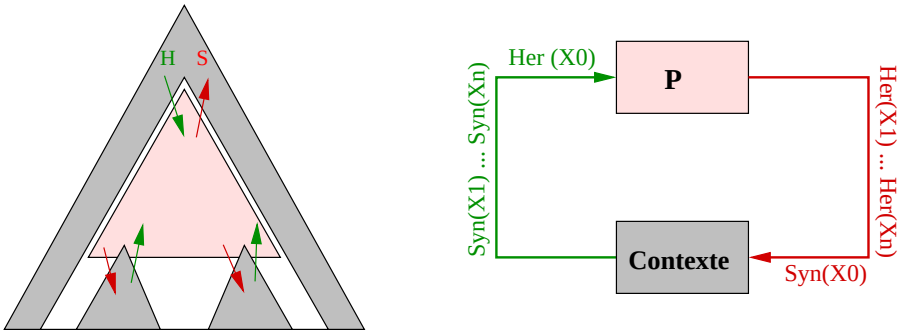

FIGURE 1.3 – Règles sémantiques exprimant les attributs de sortie en fonction des attributs d'entrée d'une production $P : X_0 \rightarrow X_1 \ldots X_n$

1.3.1 Une notation commune pour les algèbres et les grammaires attribuées

Nous avons défini au paragraphe 1.2.2 une notation pour les algèbres. Nous l'étendons ici pour obtenir une notation commune pour les algèbres et les grammaires attribuées, basée sur le calcul des séquents et dans laquelle l'ensemble des règles sémantiques s'interprète comme un système formel formé d'axiomes et de règles d'inférences. Nous partons de l'exemple de la fonction `flatten` ; qui permet de calculer la liste des feuilles (de la gauche vers la droite) d'un arbre binaire. Les règles sémantiques associées à cette fonction peuvent être présentées de la façon suivante :

$$\begin{array}{lllll} leaf_a & : & & \rightarrow & \langle tree\rangle(\downarrow xs, \uparrow (a : xs)) \\ bin & : & \langle tree\rangle(\downarrow zs, \uparrow ys)\quad \langle tree\rangle(\downarrow xs, \uparrow zs) & \rightarrow & \langle tree\rangle(\downarrow xs, \uparrow ys) \end{array}$$

La signature sous-jacente, qui est la signature d'entrée de la grammaire attribuée, est

$$\begin{array}{lcccc} Leaf_a & : & & \rightarrow & \langle tree \rangle \\ Bin & : & \langle tree \rangle \ \langle tree \rangle & \rightarrow & \langle tree \rangle \end{array}$$

Chaque symbole grammatical est associé à une liste d'attributs (hérités $\downarrow h$ ou synthétisés $\uparrow s$). On ne précise pas de noms pour ces attributs. Chaque attribut est simplement accessible via sa position dans la liste. Dans cet exemple il y a juste deux attributs, de type $[\alpha]$ associés à l'unique symbol grammatical $\langle Tree \rangle$:

$$attribute \qquad \langle Tree \rangle \ (\downarrow [\alpha], \uparrow [\alpha])$$

Les règles sémantiques s'écrivent en associant à chaque symbole apparaissant dans le type d'un opérateur $op : X_1 \ X_2 \ ... \ X_n \rightarrow X_0$, une liste d'expressions utilisant des variables. Ces expressions peuvent être des termes sur la signature de sortie si la grammaire attribuée est utilisée comme transformateur d'arbres. La portée des variables est limitée à l'opérateur et l'utilisation d'un même nom de variable pour des opérateurs distincts n'a aucune signification particulière. On distingue les positions d'entrée (positions des attributs d'entrée) et les positions de sortie (positions des attributs de sortie). On impose que les expressions en positions d'entrée soient toutes des variables, que toutes ces variables soient distinctes, et qu'elles soient les seules qui apparaissent dans les expressions en positions de sortie. Une grammaire attribuée est dite SUR (single use requirement) si chaque variable apparaissant dans les règles associées à un opérateur y apparait exactement deux fois : une fois en position d'entrée, et une fois dans une expression en position de sortie. L'exemple précédent est un peu particulier parce que les expressions en position de sortie sont également des variables (sauf dans un cas).

Un autre exemple correspondant à la solution grammaticale de l'inversion d'une liste avec paramètre d'accumulation est le suivant :

$$\begin{array}{lcll} return & : & \langle L \rangle (\downarrow Nil, \uparrow ys) & \rightarrow \ \langle Rl \rangle (\uparrow ys) \\ nil & : & & \rightarrow \ \langle L \rangle (\downarrow xs, \uparrow xs) \\ cons_a & : & \langle L \rangle (\downarrow (Cons \ a \ xs), \uparrow ys) & \rightarrow \ \langle L \rangle (\downarrow xs, \uparrow ys) \end{array}$$

On note une certaine flexibilité dans la représentation des informations provenant du contexte.

Cas particulier : S'il n'y a que des attributs synthétisés dans la grammaire attribuée, on omet à ce moment les annotations $\downarrow$ et $\uparrow$. On ne trouve que des variables en positions d'entrée et les expressions en position de sortie n'utilisent que ces variables là. Ce qui correspond à la notation décrite à la section 1.2 pour les algèbres. Par exemple l'algèbre permettant de calculer la largeur arborescente d'une expression bien parenthésée est donnée comme suit :

$$\begin{array}{lcllll} return & : & \langle P \rangle (l, p) & & \rightarrow & \langle Rp \rangle (max \ l \ p) \\ nil & : & & & \rightarrow & \langle P \rangle (0, 0) \\ bin & : & \langle P \rangle (l_1, p_1) & \langle P \rangle (l_2, p_2) & \rightarrow & \langle P \rangle (max \ (max \ l_1 \ l_2) \ p_1, p_2 + 1) \end{array}$$

Ce qui n'est rien d'autre qu'une structure d'algèbre pour la signature multi-sortes sous-jacente, et qui peut aussi s'écrire sous la forme d'un transducteur :

$$\begin{array}{lcl}
\mathbf{res}\ (Bin\quad t) & \rightarrow & max\ (\mathbf{l}\ t)(\mathbf{p}\ t) \\
\mathbf{l}\ Nil & \rightarrow & 0 \\
\mathbf{l}\ Nil & \rightarrow & 0 \\
\mathbf{l}\ (Bin\ t_1\ t_2) & \rightarrow & max\ (max\ (\mathbf{l}\ t_1)(\mathbf{l}\ t_2))(p\ t_2) \\
\mathbf{p}\ (Bin\ t_1\ t_2) & \rightarrow & (\mathbf{p}\ t_2) + 1
\end{array}$$

Nous définissons de manière générale une notation sous forme de séquent par :

$$inhs \vdash \langle exp \rangle \rhd synths$$

pour signifier que l'évaluation des attributs pour l'expression close $\langle exp \rangle$ pour les attributs hérités *inhs* produit les attributs synthétisés *synths*. L'ensemble des règles sémantiques, telles qu'on se les ait données ci-dessus, s'interprète comme un système formel (formé d'axiomes et de règles d'inférences) décrivant inductivement l'ensemble de ces séquents valides pour l'interprétation qu'on vient d'indiquer. Sur l'exemple de la fonction `flatten` de départ ces règles se présentent ainsi :

$$Leaf_a \quad \frac{}{xs \vdash Leaf_a \rhd (a : xs)} \qquad\qquad Bin \quad \frac{zs \vdash t_1 \rhd ys \qquad xs \vdash t_2 \rhd zs}{xs \vdash Bin\ t_1\ t_2 \rhd ys}$$

Un arbre de preuve pour ce système est un arbre de syntaxe abstraite (pour la signature sous-jacente) décoré par attributs de façon conforme aux règles sémantiques. Cela nous procure une structure d'algèbre dans laquelle l'interprétation d'un terme clos est l'ensemble des séquents valides qui lui sont associés :

$$[\![exp]\!] \ = \ \{inhs \vdash \langle exp' \rangle \rhd synths \mid exp' = exp\}$$

Les règles ci-dessus donnent l'interprétation des opérateurs de la signature pour cette algèbre :

$$\begin{array}{rcl}
[\![leaf_a]\!] & = & \{xs \vdash Leaf_a \rhd (a : xs) \mid xs \in [\alpha]\} \\
[\![bin]\!] & = & \{xs \vdash Bin\ t_1\ t_2 \rhd ys \quad | \quad (zs \vdash t_1 \rhd ys) \in [\![t_1]\!]\,, \\
 & & \qquad\qquad\qquad\quad (xs \vdash t_2 \rhd zs) \in [\![t_2]\!]\}
\end{array}$$

Il est parfois nécessaire pour effectuer certains calculs de connaître le type de grammaire attribuée traitée (circulaire on non), ou de définir un graphe de dépendances d'attributs de tout arbre de dérivation et un ordre d'évaluation de ces attributs. Nous présentons dans les paragraphes qui suivent quelques résultats de la littérature obtenus sur le calcul des attributs pour certains types de grammaires attribuées.

1.3.2 Évaluation d'attributs et graphes de dépendances

L'évaluation d'attributs est un procédé par lequel on calcule à l'aide des règles sémantiques de la grammaire attribuée toutes les instances d'attributs d'un arbre de dérivation [29]. Le caractère non procédural des grammaires attribuées est un avantage particulier qui a été à l'origine de nombreux travaux de recherche sur le calcul d'attributs.

Pour tout arbre de dérivation décoré par attributs, on construit un graphe orienté (GD) de dépendances des attributs en associant un sommet à chaque attribut ; et en ajoutant un arc d'un attribut a_1 à un attribut a_2 si, et seulement si, le calcul de a_2 dépend directement de a_1. Le GD exprime donc le flot de données qui doit être utilisé pour calculer la valeur de toutes les instances d'attributs apparaissant dans l'arbre (ordre d'évaluation). On pourra calculer l'ensemble des valeurs des instances d'attributs de l'arbre si, et seulement si, son graphe de dépendances ne contient pas de circuit (un attribut ne doit pas dépendre de lui-même). Il y a une boucle dans le calcul des attributs si, et seulement si, il existe un circuit dans le graphe. Le choix d'un ordre de calcul des attributs correspond au choix d'un ordre sur les sommets tel que s'il existe un arc de a_i à a_j, alors a_i précède a_j dans l'ordre. Si ce graphe ne contient aucun circuit on peut en effectuant un tri topologique de ce graphe (i.e trouver l'ordre d'évaluation) calculer tous les attributs de l'arbre. C'est exactement ce que réalise le mécanisme d'évaluation paresseuse dans l'algorithme d'évaluation des attributs que nous décrirons plus bas (au paragraphe 1.4.2).

Certains évaluateurs cependant, comme ceux décrits dans [30–33], essayent de déterminer un ordre d'évaluation de manière statique. L'ordre d'évaluation n'est pas déterminé par la forme de l'arbre de dérivation mais plutôt à partir des relations de dépendances entre les différentes instances d'attributs de chaque production. Ces types d'évaluateurs ont l'inconvénient de restreindre le type de grammaires attribuées qu'elles peuvent traiter.

D'autres évaluateurs construisent le graphe de dépendances entre les différentes instances d'attributs dans l'arbre attribué de dérivation et le parcourent ensuite, en suivant une stratégie en "profondeur d'abord" [34]. Malheureusement, on doit recalculer entièrement ce graphe chaque fois que l'on modifie l'arbre ; puisque l'ordre dans lequel les attributs sont évalués dépend de la forme de l'arbre de dérivation.

D'autres encore transforment chaque attribut en une fonction et simulent le parcours du graphe de dépendances en utilisant la pile des appels entre les différentes fonctions. Jourdan [35] a proposé un algorithme pour ce type d'évaluateurs et celui-ci a été implémenté dans le système GEODE [29]. Ces évaluateurs ont l'avantage d'être simple, de ne pas restreindre le type de grammaires attribuées à traiter et surtout, d'effectuer une évaluation par nécessité.

Le système FNC-2 [36,37] est un système de traitement de grammaires attribuées fortement non-circulaires. Les évaluateurs qu'il génère, basés sur le paradigme des séquences de visite, sont complètement déterministes. Ils sont aussi efficaces en temps et en place que des programmes écrits à la main utilisant un arbre comme structure de données interne.

Nous avons récemment introduit une nouvelle technique pour l'évaluation des attributs d'une grammaire attribuée [27, 38]. Notre algorithme consiste à évaluer un attribut par réduction à une forme normale en utilisant un automate d'arbres opérant sur une représentation arborescente des zippers [3] donnée sous la forme d'une structure de donnée cyclique. Nous utilisons une approche algébrique basée sur le calcul de plus petits points fixes qui permet d'évaluer les attributs des grammaires attribuées potentiellement circulaires et de manipuler les structures de données potentiellement infinies.

1.3.3 Grammaires attribuées non circulaires

Graphe de dépendances locales d'une production

Soit $P : X_{P(0)} \rightarrow X_{P(1)} \ldots X_{P(|P|)}$ une production d'une grammaire attribuée abstraite. On construit le *graphe de dépendances locales* $GDL(P)$ associé à la production P par : les sommets sont les instances d'attributs $X_i.a$ (a est un attribut du symbole associé à X_i) de P et on dispose d'un arc de chacun des $X_i.a$ (instance d'un attribut d'entrée de P) vers $X_j.b$ (instance d'un attribut de sortie de P) tel que la valeur de $X_j.b$ dépend fonctionnellement de celle de $X_i.a$. On peut alors définir le graphe de dépendances $GD(t)$ d'un arbre de dérivation t en « recollant » les graphes de dépendances locales le long de t.

Graphe HS (Hérités-Synthétisés) d'un symbole grammatical

On associe à chaque symbole non terminal X un *graphe hérités-synthétisés* $HS(X)$ dont les sommets sont les attributs de X et tel que l'on ait un arc de $X.a$ vers $X.b$ lorsque a est un attribut hérité de X, b un attribut synthétisé de X et il existe un chemin de $X.a$ vers $X.b$ dans le graphe de dépendances d'un arbre de dérivation ayant X en racine ; c'est à dire qu'il existe potentiellement une dépendance entre ces deux attributs (voir figure 1.4).

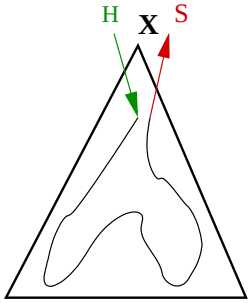

FIGURE 1.4 – Graphe $HS(X)$

Graphe de dépendances étendues d'une production

On peut alors définir le *graphe de dépendances étendues* $GDE(P)$ d'une production $P : X_{P(0)} \rightarrow X_{P(1)} \cdots X_{P(|P|)}$ en ajoutant à son graphe de dépendances locales une copie des arcs de $HS(X_{P(i)})$ pour tous les symboles non terminaux $X_{P(i)}$ apparaissant en partie droite de la production $(1 \leq i \leq |P|)$

$$GDE(P) \;=\; GDL(P) \cup \left(\bigcup_{1 \leq i \leq |P|} HS(X_{P(i)}) \right)$$

Définition 1.8. Une grammaire attribuée est *non circulaire* si le graphe de dépendances des attributs de tout arbre de dérivation selon cette grammaire est sans circuit (cycle), c'est-à-dire qu'il existe un ordre faisable de calcul des attributs.

Un algorithme de décision de la non circularité d'une grammaire attribuée existe mais il est de complexité exponentielle en la taille de la grammaire [39] et donc non utilisable en pratique. Néanmoins la plupart des grammaires attribuées non circulaires sont en fait *fortement non circulaires* (confer Défintion 1.9). Cette dernière condition est une condition suffisante (mais non nécessaire) de non circularité qui peut être testée en temps polynomial.

Définition 1.9. Une grammaire attribuée est dite *fortement non-circulaire (FNC)* lorsque tous ses graphes de dépendances étendues sont sans cycle.

Si une grammaire attribuée est circulaire alors il existe un cycle dans le graphe de dépendances d'un arbre de dérivation. On considère le plus grand sous arbre de l'arbre de dérivation dans lequel ce chemin est circonscrit et P la production utilisée à la racine de ce sous-arbre. Le cycle peut alors se décomposer en une suite de chemins dans le graphe de dépendances locales de P reliés par des chemins circonscrits dans les différents sous-arbres chacun reliant un attribut hérité à un attribut synthétisé du symbole en racine de ce sous-arbre (voir la figure 1.5). Ces chemins induisent des arcs dans les graphes $HS(X_{P(i)})$ associés et donc globalement on trouvera un cycle dans le graphe de dépendances étendues de P. Ainsi une grammaire fortement non-circulaire est non circulaire.

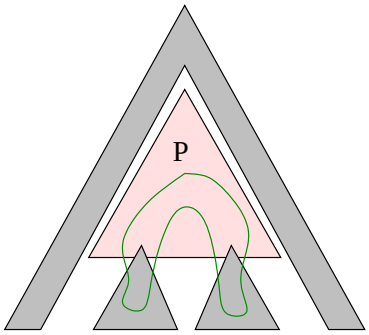

FIGURE 1.5 – Cycle dans un graphe de dépendances

Remarque 1.10. On a une dépendance mutuelle entre graphes de dépendances étendues et graphes HS :

$$\begin{aligned} GDE(P) &= GDL(P) \cup \left(\bigcup_{1 \leq i \leq |P|} HS(X_{P(i)}) \right) \\ HS(X) &= \bigcup \left\{ GDE(P)^* \cap Att(X) \mid X_{P(0)} = X \right\} \end{aligned}$$

La dernière équation stipule que $HS(X)$ est obtenu en fusionnant les graphes obtenus comme restrictions à l'ensemble des attributs de X des fermetures transitives des graphes de dépendances étendues $GDE(P)$ pour toutes les productions P ayant le symbole X en partie gauche. On obtient ainsi ces graphes par un calcul de point fixe :

$$\begin{aligned} GDE_0(P) &= GDL(P) \\ HS_{k+1}(X) &= \bigcup \left\{ GDE_k(P)^* \cap Att(X) \mid X_{P(0)} = X \right\} \\ GDE_{k+1}(P) &= GDL(P) \cup \left(\bigcup_{1 \leq i \leq |P|} HS_{k+1}(X_{P(i)}) \right) \end{aligned}$$

Exemple de grammaire non circulaire qui n'est pas FNC

Nous donnons ici un exemple d'une grammaire attribuée qui est non circulaire, et qui n'est pas fortement non circulaire. Les graphes de dépendances locales de cette grammaire sont indiqués dans la figure 1.6.

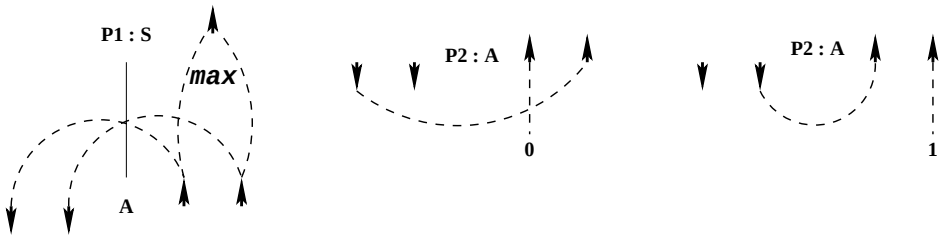

FIGURE 1.6 – GDLs d'une grammaire non circulaire qui n'est pas FNC.

Cette grammaire est notée par

$$\begin{array}{lcll} p_1 & : & \langle A\rangle(\downarrow s_1, \downarrow s_2, \uparrow s_1, \uparrow s_2) & \rightarrow \quad \langle S\rangle(\uparrow (max\ s_1\ s_2)) \\ p_2 & : & & \rightarrow \quad \langle A\rangle(\downarrow h_1, \downarrow h_2, \uparrow 0, \uparrow h1) \\ p_3 & : & & \rightarrow \quad \langle A\rangle(\downarrow h_1, \downarrow h_2, \uparrow h_2, \uparrow 1) \end{array}$$

dans laquelle h_1 et h_2 (respectivement s_1 et s_2) sont les attributs hérités (respectivement synthétisés) de A et s est l'unique attribut (synthétisé) de l'axiome S.

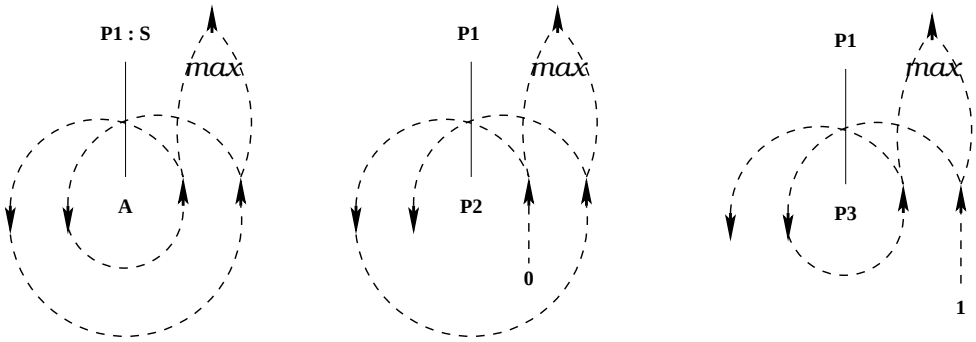

FIGURE 1.7 – Le graphe de dépendances étendues est cyclique mais les graphes de dépendances des deux arbres de dérivation ne le sont pas.

Cette grammaire attribuée n'est pas fortement non circulaire car le graphe de dépendances étendues de la production P_1 (à gauche de la figure 1.7) est cyclique. Néanmoins cette grammaire est non circulaire. En effet, il n'existe que deux arbres de dérivation possibles indiqués avec leurs graphes de dépendances en partie droite de la figure 1.7, qui sont sans cycle.

1.4 Grammaires attribuées, programmes fonctionnels et langages dédiés

La traduction des grammaires attribuées en programmes fonctionnels utilise le mécanisme d'évaluation paresseuse, les fonctions d'ordre supérieur et le polymorphisme qui sont autant de caractéristiques faisant du langage fonctionnel Haskell un hôte idéal. De même l'implémentation des langages dédiés en Haskell repose de façon critique sur ces caractéristiques ainsi que sur les types de données inductifs et co-inductifs, la notion de classe et dans certains cas sur les monades. Nous présentons dans les parties qui suivent les techniques essentielles de traduction des grammaires attribuées en programmes fonctionnels vus comme spécifications directement exécutables de langages dédiés plongés dans Haskell.

1.4.1 Haskell et l'évaluation paresseuse

Comme tout langage fonctionnel, Haskell se fonde sur le λ-calcul [40]. Les termes du λ-calcul sont formés à partir d'un ensemble dénombrable de variables à l'aide de deux opérateurs :

• Le premier opérateur, appelé abstraction, permet de former des expressions de la forme $\lambda x \rightarrow M$ dans laquelle x est une variable et M un terme. Ce qui permet de représenter une fonction f d'une variable x dont la valeur $f(x)$ est donnée par le terme M. L'abstraction λx permet de lier les occurrences de la variable x dans M qui ne sont pas déjà liées par des abstractions associées à la même variable dans M (occurrences libres de x dans M). Ces variables, dites muettes, peuvent être renommées sans que la signification associée à l'expression n'en soit modifiée. Autrement dit, le terme $\lambda x \rightarrow M$ sera considéré comme étant équivalent au terme $\lambda y \rightarrow M[y/x]$ dans lequel y est une variable non libre dans M et $M[y/x]$ représente le terme obtenu à partir de M en substituant y à toutes les occurrences libres de x dans M. Les λ-termes ne sont définis que modulo cette relation d'équivalence, appelée α-conversion.

• Le second opérateur est l'application d'une fonction à son argument notée par une simple juxtaposition $M\ N$ des termes représentant respectivement la fonction M et son argument N. L'évaluation d'un λ-terme est donnée par la β-réduction qui consiste à remplacer un sous terme de la forme $(\lambda x \rightarrow M)\ N$, appelée radical (ou redex pour *reducible expression* en anglais) par sa forme contractée $M[N/x]$ obtenue en substituant N aux occurrences libres de x dans M après une α-conversion éventuelle de M afin d'éviter qu'une variable libre de N ne soit « capturée » par une lambda abstraction dans M. Un terme est en forme normale s'il ne contient aucun radical.

La stratégie d'évaluation précise dans quel ordre on choisit de réduire les radicaux. Dans le cas séquentiel (on effectue une seule réduction à la fois) deux stratégies sont classiquement distinguées :

– La première stratégie, qualifiée d'évaluation fonctionnelle, consiste à réduire à chaque étape le radical le plus interne le plus à gauche. Ainsi pour réduire un radical $(\lambda x \rightarrow M)\ N$, on réduit d'abord M pour obtenir une forme normale M' puis on réduit l'argument N en N' et enfin le radical lui-même. Cela signifie qu'on évalue

l'argument d'une fonction avant de le transmettre à celle-ci.
- La stratégie alternative, qualifiée d'évaluation normale, consiste à réduire à chaque étape le radical le plus externe, le plus à gauche. Ce qui revient, à l'opposé de l'évaluation fonctionnelle, à réduire la fonction avant de réduire ses arguments.

L'évaluation fonctionnelle est plus facile à implémenter (par une machine à pile) et est en général (mais pas toujours) plus efficace en temps de calcul que l'évaluation normale. Cette stratégie d'évaluation est adoptée par la plupart des langages fonctionnels comme Lisp, ML, etc. Haskell par contre s'inscrit dans la lignée des langages fonctionnels dits non stricts ou paresseux qui utilisent l'évaluation normale. Haskell est effectivement du λ-calcul enrichi avec du sucre syntaxique.

Si, dans le cadre d'une réduction normale, un argument est dupliqué (la variable correspondante a plusieurs occurrences libres dans le « corps » de la fonction) on sera amené à réduire indépendamment chacune de ses copies en répétant le même calcul. Pour éviter cette redondance de calcul, ces sous expressions sont partagées pour ne les réduire qu'une seule fois. L'évaluation paresseuse combine à la fois la stratégie de réduction normale et le partage de sous expressions communes. On parle aussi de réduction optimale ou d'évaluation par nécessité. Un autre avantage de l'évaluation paresseuse est de pouvoir manipuler des structures de données potentiellement infinies en utilisant des constructeurs paresseux.

1.4.2 Evaluation paresseuse des attributs

Un ordre partiel complet (CPO pour *Complete Partial Order* en anglais) est un ensemble ordonné dans lequel toute partie dirigée admet une borne supérieure. Un CPO est dit pointé (pCPO) s'il admet un plus petit élément ordinairement appelé *bottom* et noté $\perp$ (valeur « indéfinie ») correspondant à l'absence d'information. La difficulté souvent observée lors de l'évaluation des attributs avec les langages conventionnels vient du fait que les programmes impératifs et fonctionnels stricts ($f \perp = \perp$) qui en découlent doivent spécifier exactement dans quel ordre les valeurs d'attributs sont calculées. La classe des grammaires attribuées étudiées dans ces cas est souvent celle des grammaires attribuées non circulaires.

Par contre, dans un langage fonctionnel paresseux, l'ordre réel dans lequel les expressions sont évaluées est déterminé à la demande, pendant l'exécution par les dépendances de données. Ainsi, du fait de l'évaluation paresseuse, les programmes Haskell manipulent des informations partielles : tout type contient une valeur implicite $\perp$ indiquant l'absence d'information. Les valeurs sont ordonnées avec $\perp$ comme plus petit élément : $a \leq b$ signifie que a est une approximation de b. Un type doit contenir également toutes les limites de ses chaînes croissantes d'approximants. Un type, vu comme domaine de valeurs, est alors représenté comme un ensemble ordonné dans lequel un élément apparaît comme la borne supérieure de ses approximants. Dans un langage fonctionnel paresseux les fonctions ne sont généralement pas strictes car une fonction ne cherche à évaluer son argument que lorsqu'elle a besoin d'information sur celui-ci pour progresser dans son évaluation, et donc en général $f \perp \neq \perp$ car on peut connaître de l'information sur la valeur de l'expression $f\ x$ sans utiliser la moindre information sur x. Ce qui fait que la catégorie $p\mathcal{CPO}$ des CPOs pointés et des applications continues est la catégorie qui permet de modéliser les types et les fonctions Haskell.

Soit $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ une grammaire attribuée abstraite. Les domaines de valeurs des attributs associés à chaque symbole grammatical sont des pCPOs. Si on désigne par

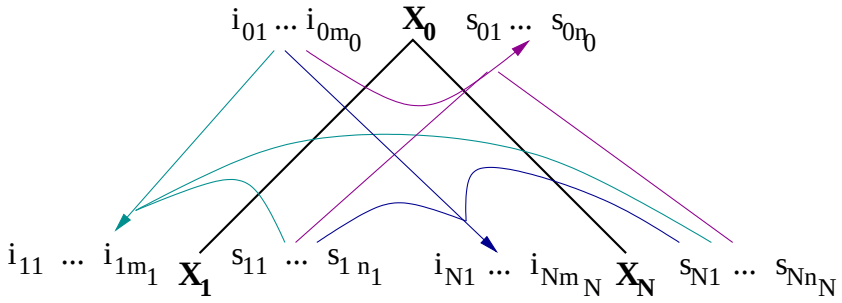

FIGURE 1.8 – Règles sémantiques d'une production $P : X_0 \rightarrow X_1\ X_2\ ...\ X_N$

$inhs_j = (i_{j1}, ..., i_{jm_j})$ et $synths_j = (s_{j1}, ..., s_{jn_j})$ les tuples formés respectivement des attributs hérités et synthétisés d'un symbole grammatical X_j de la production $P : X_0 \rightarrow X_1\ X_2\ ...\ X_N$, pour $0 \leq j \leq N$; alors les règles sémantiques de la forme $a = f_a(b_1, ..., b_k)$ exprimant chaque attribut de sortie a en fonction des attributs d'entrée b_i de P comme indiqué sur la figure 1.8 sont décrites par :

$$\begin{array}{rcll} P & : & \langle X_1\rangle(\downarrow f_{1k}(inhs_0, synths_j)_{k\in\{1,...,m_1\}}, \uparrow s_{11}, ..., \uparrow s_{1n_1}) & ..., \\ & & \langle X_N\rangle(\downarrow f_{Nk}(inhs_0, synths_j)_{k\in\{1,...,m_N\}}, \uparrow s_{N1}, ..., \uparrow s_{Nn_N}) & \rightarrow \\ & & \langle X_0\rangle(\downarrow i_{01}, ..., \downarrow i_{0m_0}, \uparrow f_{0k}(inhs_0, synths_j)_{k\in\{1,...,n_0\}}) & \textit{pour } 1\leq j\leq N \end{array}$$

En utilisant la notation introduite au paragraphe 1.3.1, l'ensemble de ces règles sémantiques s'interprète comme un système formel de la manière suivante, pour $(1 \leq j \leq N)$:

$$P\quad \frac{\begin{array}{l} f_{1k}(inhs_0, synths_j)_{k\in\{1,...,m_1\}} \vdash \langle X_1\rangle \rhd s_{11}, ..., s_{1n_1} \quad ... \\ f_{Nk}(inhs_0, synths_j)_{k\in\{1,...,m_N\}} \vdash \langle X_N\rangle \rhd s_{N1}, ..., s_{Nn_N} \end{array}}{i_{01}, ..., i_{0m_0} \vdash \langle X_0\rangle \rhd f_{0k}(inhs_0, synths_j)_{k\in\{1,...,n_0\}}}$$

Un arbre de preuve pour ce système est un arbre de syntaxe abstraite (pour la signature sous-jacente) décoré par attributs de façon conforme aux règles sémantiques. Cela nous procure une structure d'algèbre telle que l'interprétation des opérateurs de la signature pour cette algèbre est donnée par les règles sémantiques décrites ci-dessus. On a alors :

$$\begin{array}{rcl} [\![p]\!] & = & \{i_{01}, ..., i_{0m_0} \vdash \langle X_0\rangle \rhd f_{0k}(inhs_0, synths_j)_{k\in\{1,...,n_0\}} | \\ & & (f_{1k}(inhs_0, synths_j)_{k\in\{1,...,m_1\}} \vdash \langle X_1\rangle \rhd s_{11}, ..., s_{1n_1}) \in [\![\langle X_1\rangle]\!] \\ & & (f_{Nk}(inhs_0, synths_j)_{k\in\{1,...,m_N\}} \vdash \langle X_N\rangle \rhd s_{N1}, ..., s_{Nn_N}) \in [\![\langle X_N\rangle]\!]\} \end{array}$$

1.4.3 Traduction des grammaires attribuées en Haskell

De part leur structuration, les grammaires attribuées peuvent être vues comme des programmes fonctionnels déclaratifs. Cette technique d'évaluation offre l'avantage de n'imposer aucune contrainte sur les dépendances d'attributs. Et lorsqu'on utilise un langage fonctionnel paresseux comme Haskell, l'aspect déclaratif des grammaires attribuées garantit au programmeur l'existence d'un évaluateur d'attributs [12–14]. La méthode classique

de traduction d'une grammaire attribuée en un programme fonctionnel consiste à transformer la grammaire attribuée en une fonction de tous les attributs hérités d'origine vers un tuple des attributs synthétisés d'origine. L'applicabilité de cette méthode repose essentiellement sur la possibilité dans un langage fonctionnel paresseux d'avoir des définitions récursives des valeurs non fonctionnelles.

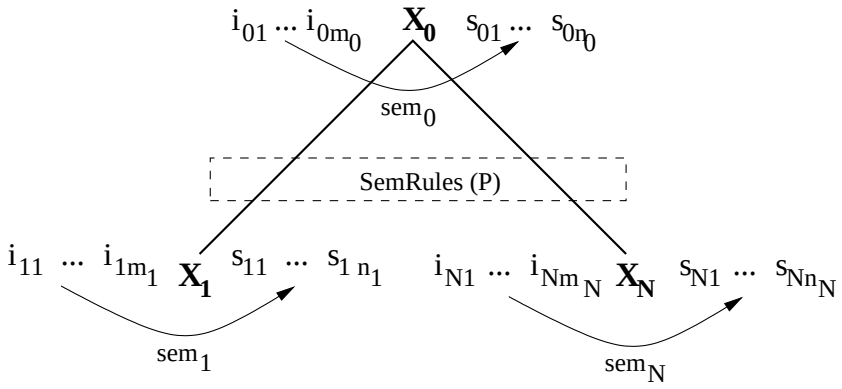

FIGURE 1.9 – Implémentation fonctionnelle des grammaires attribuées

Concrètement, on transforme chaque production de la grammaire en une fonction $trans\_P$ dont les arguments sont les fonctions sémantiques associées aux symboles de la partie droite de la production et le résultat la fonction sémantique associée à sa partie gauche. Ainsi, pour chaque production $P : X_0 \rightarrow X_1\ X_2\ ...\ X_N$, si on dénote par $SemX_j :: (i_{j1}, ..., i_{jm_j}) \rightarrow (s_{j1}, ..., s_{jn_j})$ pour $0 \leq j \leq N$, le type des fonctions sémantiques qui calculent les attributs synthétisés en fonction des attributs hérités associés à chaque symbole X_j (voir figure 1.9), on obtient le programme Haskell $trans\_P$ ci-dessous, correspondant à la traduction directe de la grammaire attribuée pour la production P, dans lequel les règles sémantiques $SemRules\,(P)$ de la forme $a = f(b_1, ..., b_k)$ expriment chaque attribut de sortie a en fonction des attributs d'entrée b_i.

$$
\begin{array}{l}
trans\_P\ ::\ SemX_1 \rightarrow SemX_2 \rightarrow ... \rightarrow SemX_N \rightarrow SemX_0 \\
trans\_P\ \ sem1\ \ ...\ \ semN\ \ (i_{01}, ..., i_{0m}) \ =\ (s_{01}, ..., s_{On}) \\
\qquad where\ (s_{11}, ..., s_{1n_1}) \ =\ sem1\ \ (i_{11}, ..., i_{1m_1}) \\
\qquad\qquad (s_{21}, ..., s_{2n_2}) \ =\ sem2\ \ (i_{21}, ..., i_{2m_2}) \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad (s_{N1}, ..., s_{Nn_N}) \ =\ semN\ \ (i_{N1}, ..., i_{Nm_N}) \\
\qquad\qquad \{ \quad \cdots\ SemRules\,(P)\ \cdots \}
\end{array}
$$

Cette définition est récursive puisque dans la clause `where`, les attributs de sortie $(i_{j1}, ..., i_{jm_j})$ hérités des X_j pour $1 \leq j \leq N$, apparaissent à la fois à gauche et à droite de ce système d'équations. Le constructeur `where` de Haskell permet de donner des définitions locales et l'ordre d'apparition de ces définitions n'a pas d'importance. En cas de circularité (en supposant néanmoins le système bien gardé : pas de définition du genre `x=x`) les variables de la déclaration sont liées aux différentes composantes du plus petit point fixe du système correspondant. Ce qui nous permet, le cas échéant de donner une sémantique par plus petit point fixe pour des grammaires attribuées circulaires. Cette formulation reprend

dans l'esprit les constructions de K. Backhouse & T. Johnsson [13, 14] ; elle s'inspire également des travaux de B. Mayol, Chirica & al. [17, 18] dans leur volonté de procurer une sémantique par point fixe des grammaires attribuées. On peut la traduire de façon littérale dans le language fonctionnel Haskell car le mécanisme d'évaluation paresseuse permet de contourner l'apparente cyclicité du programme ainsi obtenu [19] tant que la grammaire attribuée elle-même est non circulaire.

Nous illustrons cela par la traduction en programmes fonctionnels des exemples de grammaires attribuées données au paragraphe 1.2.3.

```
-------------------------------------------------------------------------
-- Inversion des listes avec paramètre d'accumulation
type SemL a = List a -> List a

trans_Nil :: SemL a
trans_Nil acc = rev where rev = acc

trans_Cons :: a -> SemL a -> SemL a
trans_Cons x sem1 acc0 = rev0
        where rev1 = sem1 acc1
              acc1 = Cons x acc0
              rev0 = rev1

-- Profondeur d'imbrication d'une écriture parenthésée
type SemP = Integer
trans_Nilp :: SemP
trans_Nilp = nest where nest = 0

trans_Binp :: SemP ->  SemP ->  SemP
trans_Binp nest1 nest2 = nest
        where nest = max (nest1 + 1) nest2

-- Largeur arborescente d'une ecriture parenthésée
type SemB = (Integer, Integer)
trans_Nilb :: SemB
trans_Nilb = (larg, taille) where {larg = 0; taille = 0}

trans_Binb :: SemB ->  SemB ->  SemB
trans_Binb sem1 sem2 = (larg, taill)
   where (larg1,tail1) = sem1
         (larg2,tail2) = sem2
         larg = maxThree larg1 larg2 tail1
         taill = tail2 + 1
maxThree a b c = max (max a b) c
-------------------------------------------------------------------------
```

Comme déjà mentionné par Swierstra [10], les fonctions $trans\_P$ associées aux produc-

tions de la grammaire sont effectivement des spécifications des fonctions d'interprétation de l'algèbre associée à la signature sous-jacente.

Cette traduction est systématique et on peut très bien envisager d'implémenter les fonctions $trans\_P$ de façon générique, par induction sur la structure de la grammaire. Elle suit effectivement la structure inductive des types et, lorsqu'on s'intéresse à une grammaire attribuée particulière, l'ensemble des combinateurs $trans\_P$ constitue le langage dédié associé à cette signature ou grammaire attribuée.

Ainsi l'inverse d'une liste paramétrée par l'algèbre des listes est donnée par :

```
reverse xs = evalList algListRevAG xs Nil
    where algListRevAG = AlgList trans_Nil trans_Cons
```

tandis que la profondeur d'imbrication et la largeur arborescente d'une écriture parenthésée sont données respectivement par[4] :

```
nesting = parenAlg (AlgBtree trans_Nilp trans_Binp)
breath  = parenAlg (AlgBtree trans_Nilb trans_Binb)
```

1.5 Un exemple de conception d'un DSL : traduire un arbre de boîtes en liste de blocs positionnés

La construction que nous avons présentée dans la section précédente qui associe aux productions d'une grammaire attribuée des combinateurs (un pour chaque production), permet de définir le langage dédié associé à cette grammaire. Nous nous proposons dans cette partie d'illustrer cette approche sur un exemple précis. Considérons la grammaire attribuée des boîtes positionnées suivante :

$$
\begin{array}{lcll}
elem_{h,w,a} & : & \rightarrow & \langle Box\rangle(\downarrow x, \downarrow y, \uparrow h, \uparrow w, \uparrow res) \\
\\
vert_{vpos} & : & & \langle Box\rangle(\downarrow x1, \downarrow (y0 + h2/2), \uparrow h1, \uparrow w1, \uparrow res1) \\
 & & & \langle Box\rangle(\downarrow x2, \downarrow (y0 - h1/2), \uparrow h2, \uparrow w2, \uparrow res2) \\
 & & \rightarrow & \langle Box\rangle(\downarrow x0, \downarrow y0, \uparrow (h1 + h2), \uparrow (max\, w1\, w2), \uparrow (res1 + +res2)) \\
\\
hor_{hpos} & : & & \langle Box\rangle(\downarrow (x0 - w2/2), \downarrow y1, \uparrow h1, \uparrow w1, \uparrow res1) \\
 & & & \langle Box\rangle(\downarrow (x0 + w1/2), \downarrow y2, \uparrow h2, \uparrow w2, \uparrow res2) \\
 & & \rightarrow & \langle Box\rangle(\downarrow x0, \downarrow y0, \uparrow (max\, h1\, h2), \uparrow (w1 + w2), \uparrow (res1 + +res2))
\end{array}
$$

dans laquelle chacune des expressions *res, res1, res2* décrit le résultat final attendu formé de la liste des blocs positionnés pour la boîte concernée ; et les variables *x1, x2, y1, y2* représentent les expressions permettant de calculer les points d'ancrage des boîtes composites en fonction de la disposition de celles-ci. Elles sont données par :

[4]Les résultats de ces deux programmes sont des listes de différentes alternatives, puisque la structure d'arbre sur laquelle le graphe de dépendances est défini est obtenue par analyse de la chaîne d'entrée par l'analyseur des écritures parenthésées paramétré par l'algèbre associée. Ainsi le résultat final de *breath* qui est la première composante du premier résultat de la liste des alternatives devrait être :
breath = fst . fst . head . parenAlg (AlgBtree trans_Nilb trans_Binb)

$$
\begin{array}{rcll}
x1\,(resp.\,x2) & = & x0-(w-w1)/2\,(resp.\,x0-(w-w2)/2) & si\,vpos = Left\_ \\
 & = & x0 & si\,vpos = Middle \\
 & = & x0+(w-w1)/2\,(resp.\,x0+(w-w2)/2) & si\,vpos = Right\_ \\
y1\,(resp.\,y2) & = & y0+(h-h1)/2\,(resp.\,y0+(h-h2)/2) & si\,hpos = Top \\
 & = & y0 & si\,hpos = Center \\
 & = & y0-(h-h1)/2\,(resp.\,y0-(h-h2)/2) & si\,hpos = Bottom
\end{array}
$$

Ainsi chaque bloc de base a une largeur et une hauteur déterminées, ainsi qu'un contenu de type a. De telles boîtes peuvent être positionnées verticalement l'une au dessus de l'autre en précisant si on les aligne à gauche (`Left_`), à droite (`Right_`) ou si on les centre (`Middle`). Elles peuvent également être positionnées horizontalement en précisant également leur alignement : en haut (`Top`), en bas (`Bottom`) ou au centre (`Center`). Toute boîte composite est associée à un rectangle englobant qui est le plus petit rectangle contenant chacune des boîtes élémentaires se trouvant dans cette boîte. La largeur, hauteur et le centre de ce rectangle sont autant d'attributs associés à une boîte. Une boîte élémentaire est identifiée à une boîte composite qui ne contient que celle-ci, bien sûr elle coïncide avec son rectangle englobant.

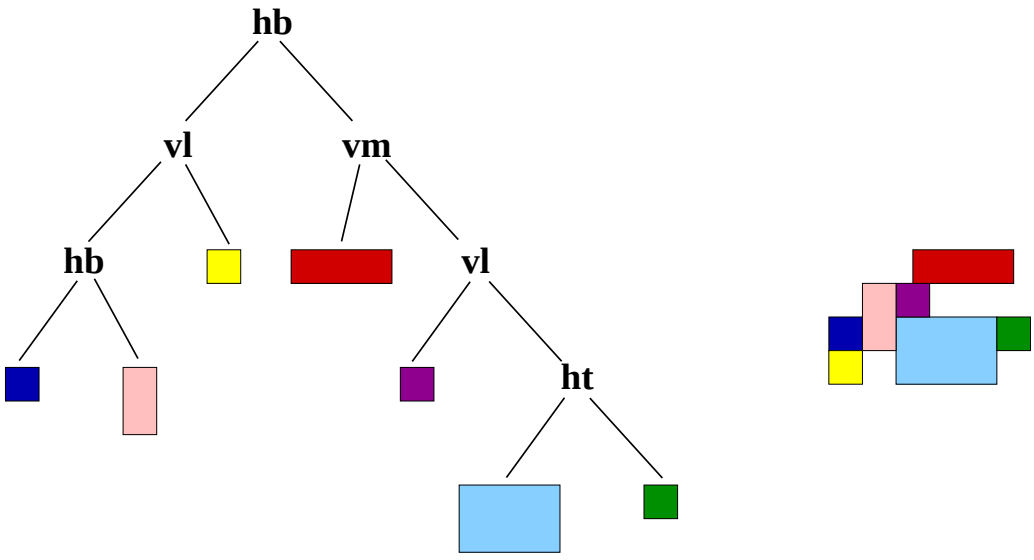

FIGURE 1.10 – Traduction d'un arbre de boîtes en liste de boîtes positionnées

On déduit à partir de la signature associée à cette grammaire abstraite les types de données Haskell suivants :

```
-------------------------------------------------------------------------
data Box a = Elembox {height, width :: Int, value a}
           | Vert {vpos :: VPos, high, low :: Box a}
           | Hor {hpos :: HPos, left, right :: Box a}
data Vpos = Left_ | Middle | Right_
data Hpos = Top | Center | Bottom
-------------------------------------------------------------------------
```

Le résultat recherché est la liste des boîtes élémentaires positionnées (une telle boîte étant caractérisée par son origine, sa taille et son contenu) en supposant que la boîte globale est centrée à l'origine des axes. Sur la figure 1.10, nous avons utilisé pour simplifier les abréviations suivantes : `hb` pour `Hor Bottom`, `ht` pour `Hor Top`, `vl` pour `Vert Left_`, et `vm` pour `Vert Middle`.

Les règles sémantiques de cette grammaire s'interprètent comme le système formel suivant :

$$Elem_{h,w,a} \quad \frac{}{x, y \vdash ElemBox_{h,w,a} \rhd h, w, res}$$

$$Vert_{vpos} \quad \frac{x1,\ y0 + h2/2 \vdash b_1 \rhd h1, w1, res1 \qquad x2,\ y0 - h1/2 \vdash b_2 \rhd h2, w2, res2}{x0, y0 \vdash Vert\ b_1\ b_2 \rhd (h1 + h2),\ (max\, w1\, w2),\ (res1 + +res2)}$$

$$Hor_{hpos} \quad \frac{x0 - w2/2,\ y1 \vdash b_1 \rhd h1, w1, res1 \qquad x0 + w1/2,\ y2 \vdash b_2 \rhd h2, w2, res2}{x0, y0 \vdash Hor\ b_1\ b_2 \rhd (max\, h1\, h2),\ (w1 + w2),\ (res1 + +res2)}$$

Ce qui procure une structure d'algèbre telle que l'interprétation des opérateurs de la signature pour cette algèbre soit donnée par ces règles sémantiques. Nous donnons dans les paragraphes suivants la traduction (directe) correspondante en Haskell. Dans un premier temps on précise la structure d'algèbre et la fonction d'évaluation correspondante pour la structure de données des arbres de boîtes :

```
-------------------------------------------------------------------------
data AlgBox a b = AlgBox{ elem :: Float -> Float -> a -> b
                        , vert :: VPos -> b -> b -> b
                        , hor  :: HPos -> b -> b -> b}

evalBox :: AlgBox a b -> Box a -> b
evalBox alg (Elembox h w v) = elem alg h w v
evalBox alg (Vert vpos high low) = vert alg vpos semh seml
                                        where semh = evalBox alg high
                                              seml = evalBox alg low
evalBox alg (Hor hpos left right) = hor alg hpos seml semr
                                        where seml = evalBox alg left
                                              semr = evalBox alg right
-------------------------------------------------------------------------
```

Le non terminal *Box* possède un attribut hérité de type *Origin* et deux attributs synthétisés de type respectifs *Size* et *Result*. *SemBox* est le type sémantique exprimant chacun de ces attributs synthétisés en fonction de l'unique attribut hérité.

```
data Size = Size{height_ , width_ :: Float}
data Origin = Pt{abs, ord :: Float}
type Result a = [LocBox a]
type LocBox = (Origin, Size, a)
```

```
type SemBox a = Origin -> (Size,Result a)
```

Le calcul du graphe HS associé au symbole Box montre que la taille ne dépend pas de l'origine et qu'il serait sans doute plus judicieux de choisir un type sémantique de la forme

```
data SemBox_ a = SemBox {size::Size, result :: Origin -> Result a}
```

Néanmoins la traduction proposée s'exprime plus simplement lorsqu'on décide, de façon uniforme, de prendre pour domaine sémantique associé à un symbole grammatical l'ensemble des fonctions de ses attributs hérités vers ses attributs synthétisés. Nous reviendrons sur ce point au chapitre suivant pour étudier sous quelle circonstance le typage plus précis obtenu par le calcul du graphe HS peut présenter un intérêt additionnel.

A nouveau la structure d'algèbre est une transcription immédiate de l'ensemble des règles sémantiques pour chacune des trois productions :

```
-------------------------------------------------------------------------
elem :: Float -> Float -> a -> SemBox a
elem height width a point = (size, res)
 where size = Size height width
       res  = [(point, size, a)]

vert :: HPos -> SemBox a -> SemBox a -> SemBox a
vert vpos sem1 sem2 (Pt x0 y0)= (Size h w, res)
 where h = h1 + h2
       w = max w1 w2
       res = res1++res2
       x1 = case vpos of
                 Left_  -> x0 - (w - w1)/2
                 Middle -> x0
                 Right_ -> x0 + (w - w1)/2
       y1 = y0 + h2/2
       x2 = case vpos of
                 Left_  -> x0 - (w - w2)/2
                 Middle -> x0
                 Right_ -> x0 + (w - w2)/2
       y2 = y0 - h1/2
       (Size h1 w1, res1) = sem1 (Origin x1 y1)
       (Size h2 w2, res2) = sem2 (Origin x2 y2)

hor :: HPos -> SemBox a -> SemBox a -> SemBox a
hor hpos sem1 sem2 (Pt x0 y0)= (Size h w, res)
 where h = max h1 h2
       w = w1 + w2
       res = res1++res2
       x1 = x0 - w2/2;
       y1 = case hpos of
```

```
                Top    -> y0 + (h - h1)/2
                Center -> y0
                Bottom -> y0 - (h - h1)/2
        x2 = x0 + w1/2
        y2 = case hpos of
                Top    -> y0 + (h - h2)/2
                Center -> y0
                Bottom -> y0 - (h - h2)/2
        (Size h1 w1, res1) = sem1 (Origin x1 y1)
        (Size h2 w2, res2) = sem2 (Origin x2 y2)
-----------------------------------------------------------------------
```

La traduction consiste à évaluer l'arbre de boîte pour l'algèbre associée aux règles sémantiques et à appliquer la fonction obtenue à l'origine :

```
trad :: Box a -> Result a
trad box = res
 where (_,res) = evalBox (AlgBox elem vert hor) box (Pt 0 0)
```

Les composantes de cette algèbre

```
elem :: Float -> Float -> a -> SemBox a
hor :: HPos -> SemBox a -> SemBox a -> SemBox a
vert :: HPos -> SemBox a -> SemBox a -> SemBox a
```

auxquelles on ajoute les sélecteurs associés au type abstrait `SemBox a`

```
size :: SemBox a -> Size
size box = sze where (sze,_) = box (Pt 0 0)
list :: SemBox a -> [LocBox a]
list box = res where (_,res) = box (Pt 0 0)
```

constituent le langage dédié pour la description, la manipulation et l'assemblage de boîtes. Dans ce langage nous pouvons écrire l'expression suivante :

```
box = hor Bottom (vert Left_ (hor Bottom ebox ebox) ebox)
                 (vert Middle ebox
                              (vert Left_ ebox
                                    (hor Top ebox ebox)))
      where ebox = elem 1 1 ()
```

D'un coté cette expression est clairement une description de la boîte donnée à la figure 1.10 ci-dessus ; d'un autre côté il s'agit d'une expression Haskell de type `SemBox a` qu'on peut utiliser à l'intérieur d'une autre expression Haskell (via les sélecteurs `size` et `list`, car `SemBox a` est un type de données abstrait). Par exemple l'expression `size box` retourne la taille de cette boîte :

```
size box = Size{height = 3, width = 4}
```

et l'expression `map fst (list box)` retourne la liste des points représentant les origines des boîtes élémentaires se trouvant dans cette boîte composite. Puisque `SemBox a` est un type polymorphe on peut aussi concevoir que les expressions de ce petit langage sont des macro-instructions qui peuvent prendre des expressions Haskell en paramètre.

Chapitre 2

Vers une méthodologie pour le développement de langages dédiés

Sommaire

2.1 Introduction

Nous avons décrit dans le chapitre précédent une implémentation fonctionnelle des grammaires attribuées pour montrer comment utiliser les grammaires attribuées comme des spécifications exécutables de langages dédiés. Les fonctions d'interprétation associées à un symbole grammatical sont les fonctions de ses attributs hérités vers ses attributs synthétisés. Tout en retournant la valeur des attributs synthétisés on imagine maintenant que ces fonctions peuvent avoir certains effets, comme par exemple construire l'arbre décoré par attributs correspondant, manipuler un état interne, gérer les exceptions, les entrées-sorties ou le non-déterminisme. On songe alors à écrire cette algèbre sous forme monadique en Haskell.

Dans ce chapitre, nous observons dans un premier temps que la traduction précédente des grammaires attribuées peut s'étendre aux fonctions monadiques, c'est à dire lorsque les fonctions sémantiques de la grammaire attribuée sont composées dans la catégorie de Kleisli de la monade considérée. Nous montrons de ce fait comment la réutilisation d'un langage dédié peut être liée au changement de la monade considérée et se faire au moyen des transformateurs de monades.

Nous observons ensuite qu'on peut réutiliser un langage dédié pour l'écriture des règles sémantiques d'une grammaire attribuée destinée à définir un autre langage dédié. Ce qui permet de montrer que la traduction des grammaires attribuées en algèbres fonctionne également dans une catégorie dont les calculs sont associés à un langage dédié.

Nous proposons dans la dernière partie, sur la base de ces différentes façons de réutiliser un langage qui s'appuient sur le formalisme des grammaires attribuées, un typage des langages dédiés en vue de leur réutilisation. Ce qui constitue un effort de formalisation d'une méthodologie de développement logiciel centré sur la conception et l'assemblage de langages dédiés.

2.2 Interprétation monadique des grammaires attribuées

Nous commençons cette partie par une présentation succincte des monades [41–43], utiles si on veut manipuler des langages dédiés dont les programmes ont des effets autres que de porter une valeur. Par ailleurs les monades ont été déjà utilisées pour définir la sémantique de langages de programmation de façon modulaire [44–46] en utilisant la notion de transformateur de monades qui autorise une stratification des différents types d'effets des calculs. Nous proposons ensuite une extension du formalisme de représentation des grammaires attribuées défini au chapitre 1 à des grammaires dont les fonctions sémantiques peuvent produire des effets. Ce qui permet d'illustrer une autre façon d'étendre un langage dédié, semblable à la programmation par aspects. Mais l'illustration la plus aboutie de cette technique de réutilisation de langages dédiés liée au changement d'une monade, que nous décrivons au paragraphe 2.2.4, est la conception par une approche grammaticale des combinateurs fonctionnels d'analyse.

2.2.1 Introduction aux monades

Définition et propriétés

Une des propriétés fondamentales des langages fonctionnels purs est la *transparence référentielle* liée à l'absence « d'effets de bord » lors de l'évaluation d'une expression fonctionnelle. On dit que l'évaluation d'une telle expression est « pure » et sa valeur ne dépend pas de son contexte et ne le modifie en aucune façon. Cette propriété est très utile lorsqu'il s'agit de réutiliser du code ou de raisonner sur celui-ci. Malheureusement programmer ne se limite pas à évaluer des fonctions : un programme est amené à interagir avec son environnement. Ceci est pris en compte en Haskell par la notion de monade [41–43] ; une structure algébrique empruntée à la théorie mathématique des catégories. Les monades permettent d'implémenter les effets de bord dans le contexte de la programmation fonctionnelle. C'est aussi une notation mise à la disposition d'un utilisateur pour désigner de nouveaux types. Une monade est un constructeur de types avec un jeu de fonctions prédéfinies lui permettant de créer et de manipuler des objets de ces types sans avoir à se soucier de la façon dont cette monade est implémentée. Il est assez usuel de définir des langages dédiés de cette façon (types abstraits + combinateurs associés). L'interface d'une monade est définie en Haskell comme une classe de la manière suivante :

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   fail   :: String -> m a
```

Une définition minimale de cette classe peut se réduire aux méthodes `return` et >>= (prononcée `bind`) en remarquant que

```
p >> q = p >>= \_ -> q
fail s = error s
```

La notion de classe en Haskell est liée à la catégorie de polymorphisme correspondant à la surcharge d'opérateur ; c'est-à-dire lorsqu'on souhaite utiliser la même notation pour des opérateurs différents. Ces opérateurs surchargés correspondent alors à des méthodes pour une classe (de types). Par exemple la somme peut être définie pour tous les types numériques

```
(+) :: Num a => a -> a -> a
```

La définition de la classe des monades signifie qu'un type paramétré m est une monade s'il supporte les deux méthodes `return` et >>= ayant les types suivants :

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

signifiant par exemple que (>>=) est du type $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ pour toute monade m. Il faut ajouter à cela le polymorphisme paramétrique dans lequel les fonctions polymorphes sont définies de manière uniforme. Par exemple la fonction qui permet de définir la longueur d'une liste $length :: [a] -> Integer$ ne dépend pas de la structure des éléments de la liste argument (correspondant à la variable de type a). Ainsi, du fait de la présence de variables de types (a et b en l'occurrence) dans la définition des opérateurs de la classe *Monad*, on utilise à la fois le polymorphisme paramétrique et la surcharge des opérateurs.

L'interprétation des méthodes d'une monade est la suivante. Une expression p de type $m\ a$ représente un calcul (dont les effets sont associés à la monade m) procurant un résultat de type a. La méthode $return :: a \rightarrow m\ a$ convertit une valeur en un calcul qui se contente de retourner cette valeur. La méthode $(>>=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ permet de composer deux calculs : le premier calcul produit une valeur de type a et, le second produit une valeur de type b en utilisant le résultat produit par le premier. Ce dernier résultat est retourné par le calcul composite. La composition simple (sans production de valeurs) $(>>) :: m\ a \rightarrow m\ b \rightarrow m\ b$ est un cas particulier de la méthode précédente qui s'exprime par : $p >> q \ = \ p >>= \backslash\_ \rightarrow q$ (c'est-à-dire que le second calcul ne tient pas compte du résultat produit par le premier). On peut aussi définir la composition de Kleisli, comme une méthode dérivée de la manière suivante :

$$
\begin{array}{l}
(\lozenge) :: \ Monad\ m \ \Rightarrow \ (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c) \\
(p \lozenge q)\ x \ = \ p\ x >>= q
\end{array}
$$

Une application $f :: a \rightarrow m\ b$ représente un calcul (dont les effets sont décrits par m) produisant un résultat de type b à partir d'une donnée de type a. Cette opération est censée vérifier les lois suivantes :

$$
\begin{array}{lcl}
p \lozenge return & = & p \\
return \lozenge p & = & p \\
(p \lozenge q) \lozenge r & = & p \lozenge (q \lozenge r)
\end{array}
$$

Lorsqu'on définit une instance particulière d'une monade, ce qui revient à définir les méthodes *return* et >>= correspondantes, on doit veiller à ce que ces lois soient satisfaites car Haskell n'a aucun moyen de les vérifier par lui-même. Une instance d'une monade est donc définie en Haskell par :

```
instance Monad ma_monade where
   -- return :: a -> ma_monade a
   return a = ........
   -- (>>=) :: ma_monade a -> (b -> ma_monade b) -> ma _monade b
   p >>= f  = ........
```

Exemples de monades

Nous donnons ici des exemples de structures de monades assez significatives et nécessaires pour la compréhension de la suite.

1. La monade des listes est une structure de données fréquemment utilisée en Haskell. Elle permet de modéliser les calculs non déterministes en associant à un calcul la liste de ses résultats possibles :

```
instance Monad [] where
    -- return :: a -> [a]
    return x = [x]
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = [y | x <- xs, y <- f x]
    -- fail :: String -> [a]
    fail s = []
```

On voit dans cette définition que les résultats de la composition de deux calculs non déterministes $f \Diamond g$ sont produits dans l'ordre suivant : on retourne les résultats produits par g à partir du premier résultat produit par f, suivis de ceux produits à partir du second résultat produit par f et ainsi de suite. Par ailleurs du fait du mécanisme d'évaluation paresseuse ces résultats ne seront produits qu'au fur et à mesure qu'on en aura besoin. Ce mécanisme est très pratique pour implémenter des algorithmes de recherche dans un espace de solutions générées de façon dynamique : en cas d'échec (liste vide de résultats) on passe automatiquement aux possibilités suivantes. Le retour arrière (`backtracking`) est ainsi géré automatiquement et de façon paresseuse ; ce qui permet de manipuler des espaces de solutions potentiellement infinis.

2. La monade avec addition correspond aux calculs qui peuvent échouer (c'est-à-dire qui admettent un zéro absorbant pour la composition des calculs) et tels qu'on puisse fusionner deux calculs en faisant le cumul de leurs résultats.

```
class (Monad m) => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Ainsi, une monade additive est d'abord une monade qui possède (en plus des méthodes usuelles d'une monade) :

– Une méthode constante `mzero`, qui est supposée être un élément absorbant pour la composition monadique, i.e. qu'elle doit vérifier les lois suivantes

```
mzero >>= k == mzero
u >>= mzero == mzero
```

– Une méthode additive `mplus` dont le zéro est l'élément neutre, et celui-ci est indécomposable :

```
mzero `mplus` x == x `mplus` mzero == x
x `mplus` y == mzero si, et seulement si, x == y == mzero
```

La monade des listes est par exemple une instance de la monade additive

```
instance MonadPlus [] where
    mzero = []
    xs `mplus` ys = xs++ys
```

2.2.2 Un formalisme pour le calcul des grammaires attribuées avec effets

Nous nous proposons dans ce paragraphe de définir un formalisme pour le calcul des grammaires attribuées avec effets, similaire à celui défini dans le chapitre précédent qui a permis de décrire en même temps le calcul des algèbres et des grammaires attribuées ordinaires. Nous donnerons ensuite dans la section qui suivra quelques extensions simples de ce formalisme.

Considérons l'expression `q` suivante, dans laquelle $f :: a_1 \to a_2 \to \cdots \to a_n \to b$ est une fonction n-aire et telle que, pour tout $1 \leq i \leq n$, on a $p_i :: m\ a_i$ pour une certaine monade m :

```
q = p1 >>= \ x1 ->
    p2 >>= \ x2 ->
    ...
    pn >>= \ xn ->
    return (f x1 x2 ... xn)
```

La flèche est prioritaire sur la composition monadique de sorte que la portée de la première abstraction (λx_1) est toute l'expression qui suit la flèche correspondante : $q = q_1 = p_1 >\!\!>\!\!= \lambda x_1 \to q_2$ avec $q_2 = p_2 >\!\!>\!\!= \lambda x_2 \to q_3$, ... , $q_n = p_n >\!\!>\!\!= \lambda x_n \to q_{n+1}$ et $q_{n+1} = return\ (f\ x_1\ x_2 \cdots x_n)$. On observe que cette expression est bien typée en vérifiant par récurrence sur k que q_{n-k+1} est de type $m\ b$ ($q_{n-k+1} :: m\ b$) sous l'hypothèse que $x_1 :: a_1$, ..., $x_{n-k} :: a_{n-k}$. Une notation a été introduite en Haskell (la notation « `do` ») permettant de récrire l'expression précédente sous la forme suivante :

```
q = do x1 <- p1
       x2 <- p2
       ....
       xn <- pn
       return (f x1 x2 ... xn)
```

ou de façon équivalente sous la forme :

```
q = do { x1 <- p1; x2 <- p2; ....; xn <- pn; return (f x1 x2 ... xn)}
```

Cette nouvelle façon de présenter facilite la lecture et la compréhension de la composition de plusieurs calculs. Elle représente le calcul consistant à effectuer les calculs représentés respectivement par les expressions p_1, ..., p_n et produisant la valeur $(f\ v_1 \cdots v_n)$ dans laquelle v_i est la valeur produite par le calcul représenté par l'expression p_i. Pour la monade des listes on dispose d'une notation spéciale, appelée schéma de compréhension des listes (par analogie au schéma de compréhension des ensembles bien connu en théorie mathématique des ensembles). On la note

```
[f x1 x2 ... xn | x1 <- xs1, x2 <- xs2, ..., xn <- xsn]
```

pour désigner alternativement

```
do x1 <- xs1
   x2 <- xs2
   ...
   return (f x1 x2 ... xn)
```

Dans la notation `do`, on ne peut pas écrire les définitions dans un ordre arbitraire. Par exemple, pour exprimer la fonction `flatten` que nous avons décrite au paragraphe 1.3.1, on ne pourra pas écrire

```
leaf a xs = return (a:xs)
bin t1 t2 xs = do ys <- t1 zs
                  zs <- t2 xs
                  return ys
```

En effet, comme la notation `do{x<-u;v}` est une abréviation de `u>>=\x->v`, la portée de la variable `x` est l'expression `v` ; et en particulier la première occurrence de la variable `zs` dans l'expression ci-dessus est libre. Bien sûr dans le cas présent on s'en tire sans trop de difficultés en changeant l'ordre des deux générateurs dans la boucle `do` :

```
leaf a xs = return (a:xs)
bin t1 t2 xs = do zs <- t2 xs
                  ys <- t1 zs
                  return ys
```

Cette dernière expression pouvant, dans ce cas, être simplifiée en

```
bin t1 t2 xs = do {zs <- t2 xs; return (t1 zs)}
```

ou, de façon encore plus concise, `bin t1 t2 = t2 >>= t1`.

Mais, dans quels cas est-on sûr de pouvoir s'en tirer ainsi ? On doit produire une solution pour chaque opérateur de la signature ; l'ordonnancement choisi par cette solution doit être indépendant des sous expressions qui pourront être données en argument à cet opérateur : la solution doit être polymorphe. Donc, une chose est certaine, la forte non circularité de la grammaire est une condition nécessaire. Si la grammaire attribuée est

fortement non circulaire, chaque graphe de dépendances étendues est acyclique et indique les dépendances fonctionnelles potentielles entre les valeurs des attributs. On peut utiliser un tri topologique de ce graphe pour obtenir un ordonnancement correct. Néanmoins on peut être amené à découper un même séquent en plusieurs expressions qui se trouveront non contigües dans le code correspondant. Le plus simple est de ne considérer que des séquents de la forme

$$inhs \vdash \langle exp \rangle \rhd_q synth$$

dans lesquels l'indice `q` fait référence à un attribut synthétisé associé au type de l'expression `exp`. Ce séquent exprime le fait que l'évaluation de cet attribut pour l'expression `exp` est `synth` dans le contexte associant les valeurs `inhs` aux attributs hérités dont celui-ci dépend selon les graphes HS. En utilisant les graphes de dépendances locales étendues, on peut donc admettre le format suivant pour les règles sémantiques d'une grammaire attribuée FNC. A tout attribut synthétisé $a \in Syn(s)$ associé à la sorte s, on associe un opérateur

$$q : s \times \sigma(h_1) \times \cdots \times \sigma(h_{k_a}) \to \sigma(a)$$

où $\{h_1, \ldots, h_{k_a}\} = \{h \in Inh(s) \mid (h, a) \in HS(s)\}$ est l'ensemble des attributs hérités dont a dépend potentiellement, selon le graphe $HS(s)$. Cet opérateur est appelé *état* attaché à la sorte $s = \iota(q)$. On note $\gamma(q) = \sigma(h_1) \times \cdots \times \sigma(h_{k_a})$ le type de l'*environnement* de q. L'arité de cet opérateur est alors $\alpha(q) = \iota(q) \times \gamma(q)$. A tout opérateur $op : s_1 \times \cdots \times s_n \to s$ de la signature sous-jacente, et à tout état q attaché à la sorte s on dispose d'une règle

$$op_q \quad \frac{\{H_j \vdash x_{i_j} \rhd_{q_j} S_j\}\, 1 \leq j \leq m}{H \vdash op(x_1, \cdots, x_n) \rhd_q S}$$

dans laquelle

- $H = h_1 \ldots h_k$ est une suite de variables[1], disjointe de l'ensemble $X = \{x_1, \ldots, x_n\}$, et de sorte $\sigma(H) = (\sigma(h_1), \ldots, \sigma(h_k)) = \gamma(q)$.
- $Y = \{S_j \mid 1 \leq j \leq m\}$ est une suite de variables disjointe à la fois de H et de X ($Y \cap H = \emptyset$ et $Y \cap X = \emptyset$) dont les sortes sont $\sigma(S_j) = \sigma(q_j)$.
- H_j est une suite d'expressions de sorte $\gamma(q_j)$ dont les variables sont prises dans V_j avec $V_j = H \cup \{S_i \mid i < j\}$[2].
- S est une expression de sorte $\sigma(q)$ dont les variables sont dans V[3] avec $V = H \cup Y$.
- q_j est un état attaché à la sorte s_{i_j}.

La méthode de traduction d'une grammaire attribuée dont le calcul des expressions contenues dans les règles sémantiques associées peuvent produire des effets, est similaire à la méthode classiquement connue de traduction des grammaires attribuées en programmes fonctionnels. Il est nécessaire cependant que la grammaire attribuée traitée soit fortement non circulaire (FNC) pour permettre de définir un ordonnancement correct décrivant les dépendances fonctionnelles entre les différentes valeurs d'attributs. Les fonctions sémantiques sont ensuite définies pour chaque attribut synthétisé associé à un symbole grammatical comme indiqué sur la figure 2.1. Elles sont définies de l'ensemble des attributs

[1] H est l'environnement de l'état q attaché à la sorte s.

[2] V_j est l'ensemble formé de H et des variables liées jusqu'au stade j (i.e des attributs synthétisés S_i générés *avant* S_j) selon l'ordonnancement obtenu par le tri topologique du graphe associé.

[3] V est l'ensemble de toutes les variables (attributs hérités selon le graphe $HS(s)$ et synthétisés S_j) entrant dans le calcul de S.

constituant l'environnement de cet attribut (ce sont les attributs hérités dont cet attribut dépend selon le graphe HS et éventuellement les attributs synthétisés générés avant celui-ci), vers un calcul monadique (avec effets) produisant la valeur de l'attribut synthétisé considéré.

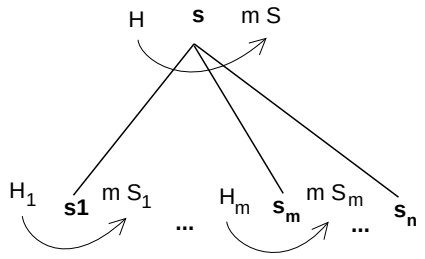

FIGURE 2.1 – Implémentation fonctionnelle des grammaires attribuées FNC avec effets dans une monade m

Ces règles s'implémentent directement en Haskell dans une monade arbitraire `m` de la façon suivante

$$q\ (op\ x_1\ \cdots\ x_n)\ h_1\ \cdots\ h_k\ \ =\ \ do\ \ S_j \leftarrow q_j\ x_{i_j}\ H_j\quad 1 \leq j \leq m \\ return\ S$$

où les fonctions sémantiques de la grammaire attribuée sont composées dans la catégorie de Kleisli de la monade `m` grâce à la notation `do`.

Si les expressions portent des valeurs qui sont des termes pour une signature de sortie Δ, on obtient ainsi un transducteur de Σ (la signature d'entrée) dans Δ et ayant des effets dont la nature dépend de la monade considérée. Cela suppose bien sûr que les expressions S correspondant aux expressions retournées par les calculs ont effectivement des effets. Si par contre, on oublie les effets (calculs purement fonctionnels) et si on suppose que les expressions dans les H_j sont des termes sur la signature Δ dont les variables sont dans H exclusivement (et non dans $V_j = H \cup \{S_i \mid i < j\}$) ; alors la règle peut s'exprimer sous la forme suivante :

$$q\ (op\ x_1\ \cdots\ x_n)\ h_1\ \cdots\ h_k\ \ =\ \ S[(q_j\ x_{i_j}\ H_j)/S_j]$$

et ce qu'on obtient n'est rien d'autre qu'un transducteur d'arbres avec paramètres de contexte (macro tree transducers), ou plus précisément une version multi-sortes de tels transducteurs (mais une telle distinction –entre version mono-sorte ou multi-sortes– n'a aucune incidence d'un point de vue théorique).

2.2.3 Quelques extensions de ce formalisme

Spécifications non déterministes

On peut écrire des définitions *non-déterministes* si on admet que plusieurs règles puissent être associées à une paire (op, q). Dans un tel cas la monade doit appartenir à la classe `MonadPlus` et les différentes règles associées à une même paire (op, q) sont regroupées à l'aide du combinateur `mplus`. On peut également souhaiter que les différentes

règles soient examinées dans leur ordre d'apparition : si la règle courante échoue (valeur `mzero`) on essaye la règle suivante. Pour cela on a besoin d'une méthode supplémentaire `orelse`. Ce qui permet de décrire la monade des alternatives `MonadAlt` suivante

```
class (MonadPlus m) => (MonadAlt m) where
  orelse :: m a -> m a -> m a
```

vérifiant les axiomes suivants

$$\begin{aligned} mzero\ `orelse`\ m &= m \\ m\ `orelse`\ p &= m \qquad \text{si } m \neq mzero \end{aligned}$$

Ce qui est le cas si le constructeur m appartient à la classe `Eq`

```
instance (MonadPLus m, Eq m) => (MonadAlt m) where
  m `orelse` n = if m /= mzero then m else n
```

Spécifications récursives (forwarding et programmation par aspects)

On peut considérer une extension du formalisme de base correspondant au format suivant

$$op_q \quad \frac{\{H_j \vdash t_j \rhd_{q_j} S_j\}\, 1 \leq j \leq m}{H \vdash op(x_1, \cdots, x_n) \rhd_q S}$$

dans lequel t_j est une expression ayant ses variables dans $X \cup V_j$.

L'idée est similaire à la définition des langages extensibles [47–49] par adjonction de nouveaux aspects [50] aux grammaires attribuées qui les décrivent, combinant *forwarding* [51] et attributs d'ordre supérieur [52]. Un aspect est donné ici par un jeu d'attributs complémentaires avec les règles sémantiques qui leur correspondent. Pour une grammaire attribuée avec forwarding, chaque production est de la forme $P : X_0 \rightarrow X_1 \ldots X_n\ X_{f_p}$ dans laquelle X_{f_p} est un non-terminal optionnel appelé *forwards-to* qui, lorsqu'il existe est le même que le non-terminal X_0 en partie gauche de la production. Un aspect se décrit alors par ce nœud supplémentaire qui complète le schéma initial. Et comme pour les grammaires attribuées d'ordre supérieur [52, 53], de tels non-terminaux sont en réalité des attributs non-terminaux. Ainsi les arbres de syntaxe abstraite (ou arbres sémantiques dans le cas où ils représentent des aspects) enracinés à ces non-terminaux sont générés par des règles sémantiques associées à la production. Pour attribuer des valeurs à ces non-terminaux, on utilise des attributs dont les valeurs sont des productions. Ces productions seront appliquées aux arbres appropriés pour produire l'arbre à assigner à l'attribut non-terminal considéré. Ainsi, le type des attributs d'une grammaire attribuée avec forwarding ou d'ordre supérieur contient aussi bien les types de base, les terminaux et non-terminaux, que l'ensemble des signatures de toutes ses productions. La similarité avec notre formalisme provient du fait qu'en ajoutant l'ensemble $X = \{x_1, \ldots, x_n\}$ aux expressions t_j, on autorise aux attributs d'avoir des valeurs dont les types peuvent être des signatures des productions de départ auxquelles on a éventuellement associé des règles sémantiques. Autrement dit, on autorise dans ce formalisme l'écriture des spécifications récursives.

Avec cette extension, on peut par exemple, décrire les combinateurs suivants de manière récursive :

```
many :: (MonadAlt m) => m a -> m [a]
many m = (some m) `orelse` (return [])

some :: (MonadAlt m) => m a -> m [a]
some m = do{a <- m; as <- many m; return (a:as)}

option :: (MonadAlt m, Eq a) => m a -> a -> m  a
option m a = m `orelse` return a
```

Le combinateur *many* est une itération simple qui répète un calcul produisant une valeur de type a autant de fois qu'il est possible (éventuellement aucun calcul) ; le résultat final étant la liste éventuellement vide des résultats de chaque calcul. Le combinateur *some* est une itération stricte, puisqu'il se comporte comme *many* sauf qu'on obtient un échec dans le cas où aucun calcul ne peut s'effectuer. Le combinateur *option* s'occupe des situations optionnelles et retourne simplement son deuxième argument si le calcul du premier échoue. Ce qui correspond à la spécification suivante

$$many \;=\; \frac{\vdash some\ m \rhd a : as}{\vdash many\ m \rhd a : as}\ `orelse`\ \frac{}{\vdash many\ m \rhd [\,]}$$

$$some \;=\; \frac{\vdash m \rhd a \quad \vdash many\ m \rhd as}{\vdash some\ m \rhd a : as}$$

$$option \;=\; \frac{\vdash m \rhd a}{\vdash option\ m\ \_ \rhd a}\ `orelse`\ \frac{}{\vdash option\ m\ a \rhd a}$$

Avec le format de base on pouvait garantir que si l'évaluation de chaque expression utilisée dans la spécification termine en un temps fini il en sera de même pour la spécification elle-même. Ceci n'est évidemment plus le cas maintenant, puisqu'on s'autorise la possibilité d'écrire des spécifications récursives dont la terminaison est non garantie en général.

Bien que les combinateurs précédents peuvent être très utiles si on les utilise à bon escient comme on le verra plus loin, leur utilisation sans précaution peut conduire à des résultats qui peuvent surprendre ; par exemple `some [1]` produit une suite infinie de 1 et l'évaluation de `some (Just 1)` boucle sans produire aucun résultat. On obtient alors dans les deux cas une exception du type « débordement de capacité de la pile ».

Spécifications dynamiques

Une autre généralisation du format de base, analogue au formalisme des grammaires attribuées dynamiques introduit par D. Parigot & al. [54], consiste à conditionner chaque séquent en prémisse par une condition portant sur les variables liées à ce stade. Une condition peut aussi apparaître en fin de prémisse, elle porte alors sur toutes les variables de V :

$$op_q\ \frac{\left\{C_j : H_j \vdash x_{i_j} \rhd_{q_j} S_j\right\} 1 \leq j \leq m \quad C :}{H \vdash op(x_1, \cdots, x_n) \rhd_q S}$$

où C_j est un prédicat portant sur les variables V_j et C un prédicat portant sur V. Dans la pratique on fera paraître en prémisse séquents et conditions alternant dans un ordre arbitraire. Une condition est un prédicat portant sur l'ensemble de variables visibles à l'endroit de la règle où cette condition apparaît, et elle conditionne la suite de l'évaluation de la règle. Cela correspond exactement à l'interprétation de l'opérateur de garde pour une monade additive qu'on définit par

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

tel que $guard\,(x>0) >> p$ permet de garder l'exécution de p par la condition $x>0$. Le code Haskell correspondant à la spécification de départ se présente comme suit :

$$\begin{array}{rcll} q\ (op\ x_1\ \cdots\ x_n)\ h_1\ \cdots\ h_k & = & do & guard\ C_j \\ & & & y_j \leftarrow q_j\ x_{i_j}\ H_j \quad 1 \leq j \leq m \\ & & & guard\ C \\ & & & return\ S \end{array}$$

Ce qui permet par exemple de spécifier la conditionnelle par

$$cond \;=\; \frac{\rho \vdash c \rhd True \quad \rho \vdash x \rhd v}{\rho \vdash cond\ c\ x\ y \rhd v} \quad mplus \quad \frac{\rho \vdash c \rhd False \quad \rho \vdash y \rhd v}{\rho \vdash cond\ c\ x\ y \rhd v}$$

qui se traduit en Haskell par

```
(cond c x y) env = do{guard (eval c env);x}
                     `mplus`
                   do{guard (not (eval c env));y}
```

(en posant `cond c x y = \env -> eval (cond c x y) env`).

En utilisant les différents types précédents d'extensions (non déterminisme, forwarding et caractère dynamique), on peut décrire le combinateur `while`

$$while \;=\; \frac{\rho \vdash cond \rhd True \quad \rho \vdash body \rhd \rho' \quad \rho[\rho'] \vdash while\ cond\ body \rhd \rho''}{\rho \vdash while\ cond\ body \rhd \rho''}$$

$$orelse \quad \frac{\rho \vdash c \rhd False}{\rho \vdash while\ cond\ body \rhd \rho}$$

```
(while cond body) env = do guard (eval cond env)
                           env' <- eval body env
                           while cond body (update env env')}
                        `orelse` (return env)
```

2.2.4 Compilateurs extensibles et combinateurs d'analyse

Engler [55] a décrit un système, *Magik*, qui permet au programmeur de modifier le processus de compilation. Comme cela a été mis en valeur par Reynolds [56], la signification d'un langage est donnée par son interpréteur ou son compilateur. Ce travail précurseur de Reynolds a donné lieu à de nombreux travaux sur la construction modulaire d'interpréteurs (voir par exemple [57]). Plutôt que de chercher à étendre un langage ou à modifier un programme on peut songer à modifier le processus de compilation, c'est ce que le système *Magik* autorise en permettant au programmeur d'introduire des extensions qui ont accès à toutes les informations qui sont calculées pendant la compilation.

Dans le cadre de Haskell les monades et les transformateurs de monades ont été introduits dans un but similaire pour produire des compilateurs de façon modulaire [44–46]. A titre d'illustration nous donnons ici une reconstruction des combinateurs fonctionnels d'analyse [58, 59] à l'aide d'une grammaire attribuée dont les règles sémantiques utilisent des calculs avec effets (grâce aux monades et aux transformateurs de monades).

Combinateurs fonctionnels d'analyse

Le problème d'analyse peut s'énoncer comme suit : étant donnée une chaîne, construire un arbre qui décrit la structure de ladite chaîne. La chaîne peut être décrite par une grammaire algébrique et son analyse effectuée par un analyseur lexical ou syntaxique.

Les combinateurs fonctionnels d'analyse [58, 59] représentent un exemple typique de langage dédié implémenté par un jeu de combinateurs pour l'analyse lexicale ou syntaxique d'une grammaire algébrique. Ils permettent d'assembler de manière compositionnelle des analyseurs équivalents à ceux produits par des générateurs comme `Lex` ou `Yacc`. Pour pouvoir composer séquentiellement des analyseurs, il faut admettre qu'un analyseur ne consomme qu'un préfixe de la chaîne en entrée, produit un fragment de l'arbre résultat et transmet aux analyseurs suivants la partie de la chaîne non encore analysée. Par ailleurs même avec une grammaire non ambiguë il n'y a aucune raison pour que la reconnaissance d'un tel préfixe soit déterministe. On gère le non déterministe de façon paresseuse en utilisant à chaque étape la liste des alternatives. Le retour-arrière (backtracking) est ainsi géré de façon implicite par le mécanisme d'évaluation paresseuse. Ce qui conduit à considérer le type suivant pour un analyseur :

```
type Parser s a = [s] -> [(a,[s])]
```

dont l'interprétation en est la suivante : si $p :: Parser\ s\ a$ est un analyseur et $xs :: [s]$ est une chaîne d'entrée alors $p\ xs$ produit la liste des paires (t, zs) dans lesquelles t est un arbre de dérivation produit par p en lisant un préfixe ys de xs et, zs est la chaîne résiduelle telle que $xs = ys +\!+ zs$.

Pour la composition séquentielle, J. Fokker [58] a utilisé les listes en compréhension, pour transmettre la chaîne résiduelle produite par l'analyseur courant à l'analyseur suivant. La notion de listes en compréhension, similaire à la notation mathématique bien connue d'ensembles en compréhension, s'exprime de façon générale de la manière suivante : une expression `[f x | x <- xs]` se définit intuitivement comme « la liste de toutes les valeurs `f x` telles que `x` parcourt la liste `xs` ». Le principal inconvénient avec cette solution utilisant les listes en compréhension est que la manipulation de la chaîne en

entrée est gérée de manière explicite par le programmeur. Pour palier à cet inconvénient, les analyseurs monadiques de G. Hutton [59] utilisent la monade d'état pour rendre implicite la manipulation de la chaîne d'entrée lorsqu'on les compose séquentiellement. Nous proposons dans la section suivante une solution plus générale basée sur les grammaires attribuées.

Approche grammaticale des combinateurs d'analyse

Notre solution est directement implémentée en terme de grammaires attribuées avec des calculs paramétrés par une monade arbitraire.
Un analyseur est du type `[s] -> m (a,[s])` pour une monade `m` d'une certaine classe. C'est donc une flèche de `[s]` dans `(a,[s])` dans la catégorie de Kleisli de la monade `m` considérée. On voit bien que les combinateurs peuvent se décrire par des règles sémantiques dans le langage de cette monade, avec un attribut hérité `inp` de type `[s]` (chaîne d'entrée) et deux attributs synthétisés `res` de type `a` (le résultat de l'analyse), et `out` de type `[s]` (la chaîne résiduelle). Les graphes de dépendances locales associés aux productions s'en déduisent naturellement et nous en donnons une illustration sur la figure 2.2 pour les productions *Succeed : P -> value* et *SeqComp : P -> P P* décrivant respectivement l'analyseur qui réussit toujours sur son unique argument et, celui qui réalise la composition séquentielle de deux analyseurs.

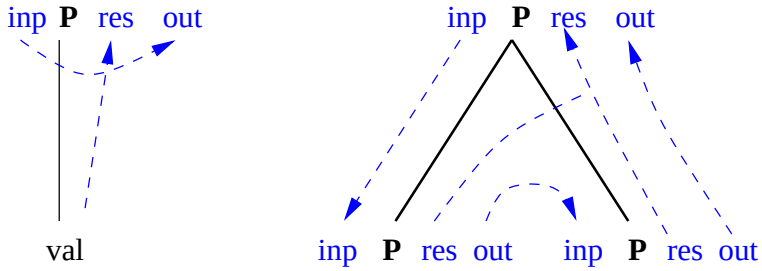

FIGURE 2.2 – GDLs des productions *Succeed* et *SeqComp*

Cette solution directe présente néanmoins le même inconvénient que celle proposée par J. Fokker : la manipulation de la chaîne d'entrée est effectuée de manière explicite par le programmeur. Nous avons opté pour l'approche préconisée par G. Hutton qui utilise la monade d'état pour rendre implicite le traitement de la chaîne d'entrée. Notons cependant que la grammaire attribuée qui décrit les différents combinateurs dans cette dernière approche ne contient aucun attribut hérité. Pour permettre la réutilisation par changement de monades, nous définissons un transformateur de monades pour les analyseurs.

Un transformateur de monades est une fonction qui transforme toute monade m en une monade $t\ m$ telle qu'il existe un morphisme de monades $lift :: m \rightarrow t\ m$ c'est-a-dire une fonction polymorphe $lift :: m\ a \rightarrow t\ m\ a$ encapsulée dans la classe des transformateurs de monades suivante

```
class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a
```

et vérifiant les lois suivantes ci-après :

$$\begin{aligned} lift\ .\ return &= return \\ lift\ m >>= k &= (lift\ m) >>= \lambda a \rightarrow (lift\ (k\ a)) \end{aligned}$$

L'opération `lift` permet de relever toutes les opérations de la monade immergée `m` dans la monade étendue `t m`. La monade étendue herite aussi des fonctionalités de la monade de base meme si cela est moins direct. Un transformateur de monades dans le cas des analyseurs peut ainsi se définir comme suit :

```
data ParserT s m a = MkP {runParserT :: [s] -> m (a,[s])}
```

On observe dans le listing ci-après que si `m` est une monade (classe `Monad`), respectivement une monade additive (`MonadPlus`) ou un élément de la classe `MonadAlt` il en est de même de `ParserT s m`. De même `ParserT s` est un transformateur de toute monade `m`.

```
-----------------------------------------------------------------------
instance Monad m => Monad (ParserT s m) where
  -- return :: a -> ParserT s m a
  return a = MkP $ \xs -> return (a,xs)
  --(>>=) :: ParserT s m a -> (a -> ParserT s m b) -> ParserT s m b
  p >>= f = MkP $ \xs -> do (a,xs')<- runParserT p xs
                            runParserT (f a) xs'

instance (Monad m) => MonadTransformer (ParserT s) m  where
  -- lift :: m a -> ParserT s m a
  lift m = MkP $ \xs-> do {x <- m; return (x,xs)}

instance MonadPlus m => MonadPlus (ParserT s m) where
  -- mzero :: ParserT s m a
  mzero = MkP $ \xs -> mzero
  -- mplus :: ParserT s m a -> ParserT s m a -> ParserT s m a
  p `mplus` q = MkP $ (\xs -> (runParserT p xs)
                              `mplus` (runParserT q xs))

instance (MonadAlt m, Eq s) => MonadAlt (ParserT s m) where
  -- orelse :: (Eq a) => ParserT s m a -> ParserT s m a
  --                                   -> ParserT s m a
  p `orelse` q = MkP $ (\xs -> (runParserT p xs)
                               `orelse` (runParserT q xs))
-----------------------------------------------------------------------
```

Les combinateurs `some`, `many`, `option` définis au paragraphe 2.2.3 se déduisent alors automatiquement :

```
many :: (MonadAlt m, Eq s, Eq a) => ParserT s m a -> ParserT s m [a]
many p = do {x <- p ; xs <- many p ; return (x:xs)} `orelse`  return []
```

```
option ::(MonadAlt m,Eq s,Eq a) =>ParserT s m a -> a -> ParserT s m a
option p a = p `orelse` return a

some :: (MonadAlt m,Eq s,Eq a) => ParserT s m a -> ParserT s m [a]
some p = do {x<- p; xs <- many p; return (x:xs)}
```

On définit aussi la fonction

```
item :: (MonadPlus m) => ParserT s m s
item = MkP $ \xs -> if null xs then mzero else return (head xs,tail xs)
```

Il s'agit de l'unique autre combinateur qui utilise explicitement la représentation concrète des analyseurs. La classe des analyseurs relative à la structure de monade m et le type des lexèmes s est celle qui en plus des méthodes se déduisant de la structure de la classe `MonadAlt` possède une méthode supplémentaire permettant de lire le lexème courant :

```
class (MonadAlt m) => (ParserMonad m s) where
   read :: m s
instance (MonadPlus m, MonadAlt (ParserT s m))
         => ParserMonad (ParserT s m) s where
   read = item
```

On peut alors par exemple définir le combinateur `sat` qui retourne un lexème si celui-ci satisfait un prédicat, de la manière suivante :

$$sat \;=\; \frac{\vdash read \rhd s \qquad p\ s}{\vdash sat\ p \rhd s}$$

```
sat :: ParserMonad m s => (s -> Bool) -> m s
sat p = do{s <- read; if p s then  return s else mzero}
```

Tous les autres combinateurs gardent leur définition monadique d'origine, seul leur type devient plus général. Ainsi les combinateurs `done`, `lower`, `symbol`, `digit`, `succeed`, etc. se décrivent de la manière suivante

$$done \;=\; \frac{}{\vdash done \rhd ()}$$

$$lower \;=\; \frac{\vdash sat\ isLower \rhd c}{\vdash lower \rhd c}$$

$$digit \;=\; \frac{\vdash sat\ isDigit \rhd c}{\vdash digit \rhd (ord\ c \;-\; ord\ '0')}$$

$$succeed \;=\; \frac{}{\vdash succeed\ x \rhd x}$$

$$symbol \;=\; \frac{\vdash sat\ (==x) \rhd c \qquad \vdash done \rhd ()}{\vdash symbol\ x \rhd ()}$$

et se traduisent en Haskell par :

```
-------------------------------------------------------------------------
lower :: ParserMonad m Char => m Char
lower = sat isLower

symbol :: (ParserMonad m s, ParserMonad m (), Eq s) => s -> m ()
symbol x = do {c<- sat (==x); done}

done ::ParserMonad m () =>  m ()
done = return ()

succeed :: ParserMonad m a => a -> m a
succeed x = return x

digit :: ParserMonad m Char => m Int
digit = do {c <- sat isDigit; return (ord c - ord '0')}
-------------------------------------------------------------------------
```

Réutilisation de DSLs grâce aux transformateurs de monades

Le type $ParserT\ m\ s\ a$ du transformateur d'une certaine monade m pour le cas des analyseurs que nous venons de décrire pour les classes de monade additive (`MonadPlus`) et alternative (`MonadAlt`) a été suffisant pour définir de façon grammaticale et modulaire les combinateurs d'analyse. Ce qui nous a permis d'obtenir un langage d'analyseurs syntaxiques paramétrés par les monades de ces classes. La réutilisation d'un langage dédié (pour l'analyse lexicale et/ou syntaxique) liée au changement de la monade se fera en utilisant ce transformateur de monades. Un même analyseur sera de ce fait interprété différemment lorsque nous changeons la monade m. Par exemple :

- En prenant pour m la monade des listes, on retrouve le jeu de combinateurs d'origine décrits par G. Hutton & E. Meijer [59].
- Et si on ne s'intéresse qu'à des analyseurs déterministes (1 ou 0 résultat), on remplace la monade des listes par la monade des calculs partiels (monade `Maybe`).

Mais on peut aussi songer à étendre la monade pour inclure des effets supplémentaires : génération de traces, manipulation d'un état, gestion d'exceptions, etc.

2.2.5 Vers une classe de monades récursives

On a vu que les monades additives et alternatives fournissent une syntaxe parfaite pour décrire les grammaires attribuées dans un contexte monadique, ainsi que pour certaines extensions de celles-ci suffisamment expressives pour décrire la sémantique opérationnelle de certains langages de programmation. En particulier, l'exemple des combinateurs d'analyse a permis d'illustrer cela malgré le fait qu'on n'ait pas eu à utiliser des attributs hérités. On s'est par ailleurs placé dans l'hypothèse où la grammaire attribuée est fortement non circulaire. En effet, la fonction `bind` (>>=) de la classe `Monad` utilisée dans la composition de Kleisli est habituellemnt stricte en son premier argument. C'est pourquoi on doit

spécifier un ordre d'évaluation des variables liées dans une expression `do` pour l'écriture des fonctions sémantiques monadiques. Si on souhaite obtenir une sémantique (monadique) par point fixe pour une grammaire attribuée circulaire, on utilisera la notation μdo [60,61], une extension de la notation *do* usuelle qui autorise le calcul récursif sur les valeurs produites par les monades. La notation μdo considère l'opérateur *mfix* de type $\forall a.$ *(a -> m a) -> m a* dans lequel m est la monade considérée comme opérateur du point fixe. L'implémentation en Haskell de cet opérateur est réalisée dans la classe *MonadFix* du module *Control.Monad.Fix* de la bibliothèque standard :

```
class (Monad m) => MonadFix m where
    mfix :: (a -> m a) -> m a
```

Le point fixe d'un calcul monadique *mfix f* exécute l'action f une seule fois, avec la sortie éventuellement renvoyée en entrée. C'est pourquoi f ne doit pas être stricte. Nous ne disposons malheureusement pas à l'heure actuelle d'un exemple assez significatif où ce calcul récursif est nécessaire.

2.3 Stratification de langages dédiés

Nous avons montré dans la section précédente comment la réutilisation d'un langage dédié peut être liée au changement de la monade et se faire au moyen des transformateurs de monades. Nous donnons dans cette partie une autre façon de réutiliser un langage dédié. On l'utilise pour l'écriture des règles sémantiques d'une grammaire attribuée destinée à définir un autre langage dédié. Nous illustrons cette approche en définissant des extensions (implémentation visuelle, jeu de billard) du langage d'assemblage des boîtes, introduit au chapitre 1, dans lequel les règles sémantiques sont écrites dans le langage FAL (Functional Animation Language) de Paul Hudak [16] (dont une description sommaire est donnée en annexe B), un langage plongé dans Haskell dédié à la programmation d'animations réactives. Ce qui permet de montrer que la traduction des grammaires attribuées en algèbres fonctionne également dans une catégorie dont les calculs sont associés à un langage dédié. Et si ce langage est lui-même décrit à l'aide d'une grammaire attribuée, on peut observer le résultat obtenu comme une forme de composition horizontale dans laquelle on organise les langages en strates horizontales (signature d'entrée à gauche, et signature de sortie à droite) de telle sorte que dans chaque strate on peut utiliser des notations produites dans les strates antérieures.

2.3.1 Extensions du langage d'assemblage de boîtes

Nous avons présenté dans la section 1.5 un exemple de langage dédié décrit à partir d'une grammaire attribuée pour l'assemblage de boîtes positionnées. Une implémentation visuelle de ce langage peut s'effectuer dans le langage FAL où les règles sémantiques sont écrites dans la catégorie des calculs effectués dans ce DSL. Cette implémentation est donnée par le tableau 2.1, où les fonctions `over, paint, translate, constB, ...` du langage FAL sont décrites en annexe B.

Tableau 2.1 Langage d'assemblage de boîtes statiques

```
elemBox :: Float -> Float -> Color -> SemBox Color
elemBox w h c (Pt x0 y0) = SynBox (Size w h) res
        where res  = let rect =  translate ((constB x0) ,(constB y0))
                                           (rec (constB w) (constB h))
                         fixBox = paint (constB c) rect
                     in fixBox

vertBox :: VPos -> SemBox Color -> SemBox Color -> SemBox Color
vertBox vpos sem1 sem2 (Pt x0 y0) = SynBox (Size w h) res
 where SynBox (Size w1 h1) res1 = sem1 origin1
       SynBox (Size w2 h2) res2 = sem2 origin2;
       (h , w) = (h1 + h2 , max w1 w2);
       (origin1 , origin2) = (Pt x1 y1 , Pt x2 y2);
       res = res1 `over` res2;
       x1 = case vpos of
                  Left_ -> x0 + (w1 - w)/2
                  Middle -> x0
                  Right_ -> x0 + (w - w1)/2
       x2 = case vpos of  Left_ -> x0 + (w2 - w)/2
                          Middle -> x0
                          Right_ -> x0 + (w - w2)/2
       (y1 , y2) = (y0 +(h - h1)/2 , y0 + (h2 - h)/2)

horBox :: HPos -> SemBox Color -> SemBox Color -> SemBox Color
horBox hpos sem1 sem2 (Pt x0 y0) = SynBox (Size w h) res
 where SynBox (Size w1 h1) res1 = sem1 origin1
       SynBox (Size w2 h2) res2 = sem2 origin2;
       (h , w)  = (max h1 h2 , w1 + w2);
       (origin1 , origin2) = (Pt x1 y1 , Pt x2 y2);
       res = res1 `over` res2;
       (x1 , x2) = (x0+(w1-w)/2 , x0 + (w - w2)/2);
       y1 = case hpos of Top -> y0 + (h - h1)/2
                         Center -> y0
                         Bottom -> y0 + (h1 - h)/2
       y2 = case hpos of Top -> y0 + (h - h2)/2
                         Center -> y0
                         Bottom -> y0 + (h2 - h)/2
```

Les structures de données, d'algèbre et le schéma de récursion associé restent sans changement. Seuls les domaines sémantiques (exprimant les fonctions des attributs hérités vers les attributs synthétisés) et les spécifications concrètes des fonctions d'interprétation de l'algèbre (expression des règles sémantiques pour chaque production) sont décrits dans FAL. Ainsi on définit le type sémantique `SemBox` par :

```
type SemBox a = Origin -> SynBox a
data SynBox a = SynBox {size :: Size , res :: Behavior Picture}
```

dans lequel l'attribut synthétisé *res* :: *Behavior Picture* représente l'image résultat de l'ensemble des boîtes positionnées. L'interprétation d'un terme de type *Box a* en une

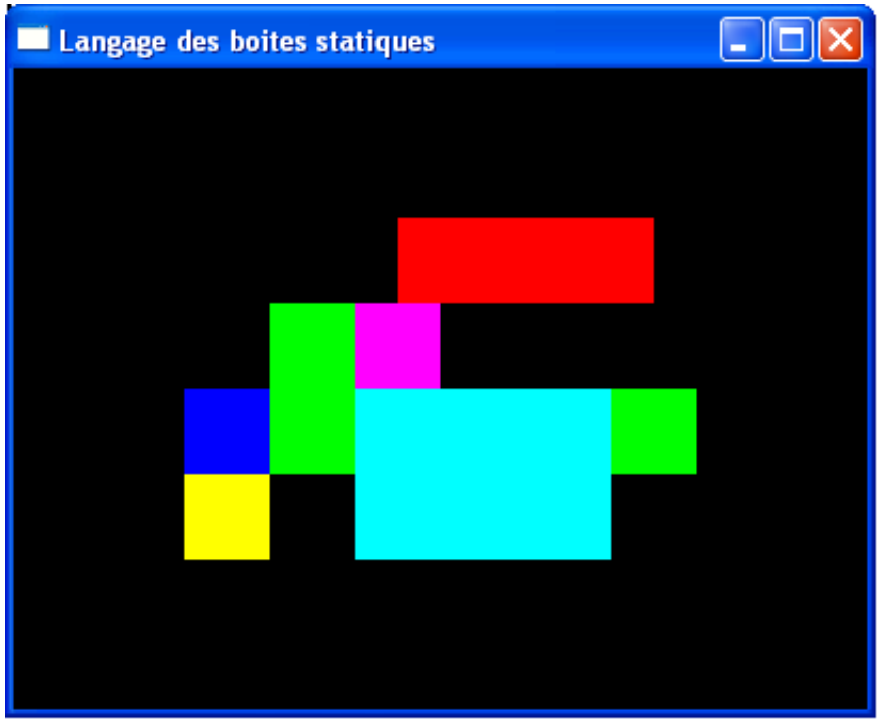

FIGURE 2.3 – Langage d'assemblage de boîtes statiques

image de type *Behavior Picture* est donnée par la fonction `mkBoxes` qui utilise l'algèbre `evalBox`. Lorsque le paramètre `a` désigne la couleur de la boîte, le résultat de l'exécution de la boîte `box` est donné par la figure 2.3.

```
-------------------------------------------------------------------------
mkBoxes :: Box Color -> Behavior Picture
mkBoxes box = res
 where SynBox _ res = evalBox (AlgBox elemBox vertBox horBox) box (Pt 0 0)

box1 = ElemBox 1 1 Blue
box2 = ElemBox 1 2 Green
box3 = ElemBox 1 1 Yellow
box4 = ElemBox 3 2 Cyan
box5 = ElemBox 1 1 Green
box6 = ElemBox 1 1 Magenta
box7 = ElemBox 3 1 Red
box =Hor Bottom (Vert Left_ (Hor Bottom box1 box2) box3)
                (Vert Middle box7 (Vert Left_ box6 (Hor Top box4 box5)))
-------------------------------------------------------------------------
```

On souhaite étendre cet affichage de boîtes en y ajoutant de la réactivité. Par exemple on aimerait que la boîte sur laquelle un utilisateur a cliqué se mette à tourner et s'arrête pour passer la main à une autre boîte si celle-ci a reçu le même évènement. Si l'évènement (clic de souris) se produit en dehors de l'ensemble des boîtes assemblées, l'animation s'arrête ; ainsi de suite. Pour cela, on doit ajouter de nouveaux attributs et définir les règles sémantiques appropriées. On rajoute ainsi l'attribut hérité *lbpt* :: *Event Vertex* correspondant au clic du bouton gauche de la souris et qui porte les coordonnées du point où l'on a cliqué. Le type sémantique `SemBox` devient alors

```
data InhBox  = InhBox {orig :: Origin, lbpt :: Event Vertex}
data SynBox a = SynBox {size :: Size , res :: Behavior Picture}
type SemBox a = InhBox -> SynBox a
```

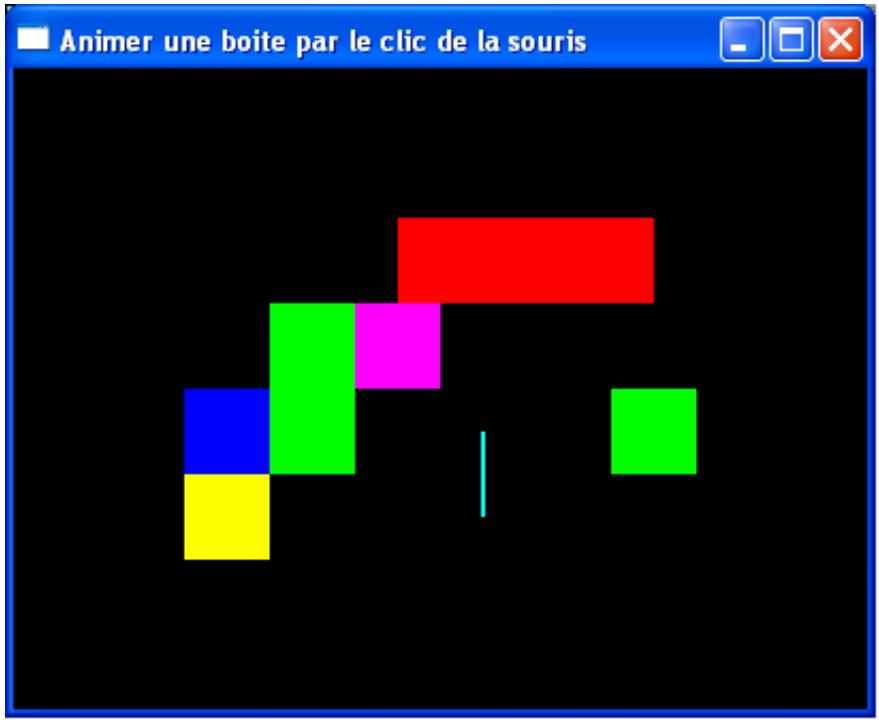

FIGURE 2.4 – Boîtes animées par le bouton gauche de la souris

Le nouveau code correspondant aux combinateurs `elemBox`, `vertBox` et `horBox` est donné par le listing du tableau 2.2.

Tableau 2.2 Langage des boîtes animées par le bouton gauche de la souris

```
elemBox :: Float -> Float -> Color -> SemBox Color
elemBox w h c (InhBox (Pt x y) lbp) = SynBox (Size w h) res
 where res = let rect = translate orig (rec (constB w) (constB h));
                 orig = ((constB x), (constB y))
                 fixBox = paint (constB c) rect;
                 animBox = paint (constB c) (translate orig
                                              (rec (sin time) (cos time)));
                 resultBox = fixBox `switch` (lbp  =>> \p ->
                      condB (rect `containsRB` (lift0 p)) animBox fixBox)
             in resultBox

horBox :: HPos -> SemBox Color -> SemBox Color -> SemBox Color
horBox hpos sem1 sem2 (InhBox (Pt x0 y0) lbp0) = SynBox size res
 where SynBox (Size w1 h1) res1 = sem1 (InhBox origin1 lbp1)
       SynBox (Size w2 h2) res2 = sem2 (InhBox origin2 lbp2);
       size = Size w h;
       lbp1 = lbp0;
       lbp2 = lbp0;
         ...

vertBox :: VPos -> SemBox Color -> SemBox Color -> SemBox Color
vertBox vpos sem1 sem2 (InhBox (Pt x0 y0) lbp0) = SynBox size res
 where SynBox (Size w1 h1) res1 = sem1 (InhBox origin1 lbp1)
       SynBox (Size w2 h2) res2 = sem2 (InhBox origin2 lbp2);
       size = Size w h;
       lbp1 = lbp0;
       lbp2 = lbp0;
         ...
```

L'implémentation du combinateur `elemBox` pour cette version mérite une attention particulière du fait de sa relative complexité. La variable `rect` dans la clause `let` permet de placer la boîte élémentaire à sa position définitive donnée par `orig`. La variable `fixBox` construit la forme statique de la boîte tandis que la variable `animBox` construit sa forme animée. L'animation consiste à faire varier au cours du temps la largeur (respectivement la hauteur) de la boîte suivant la fonction `sin` (respectivement `cos`) du temps `time`, donnant ainsi l'impression que la boîte est en train de tourner. Le résultat final est donné par la variable `resultBox` qui correspond initialement à la forme statique de la boîte, puis bascule au clic du bouton gauche de la souris (`lbp`) vers une suite infinie de comportements correspondant à sa forme animée chaque fois que le clic se produit dans une composante de la boîte englobante (*rect `containsRB` (lift0 p)*) ; ou à sa forme fixe sinon. Les combinateurs `vertBox` et `horBox` se contentent de transmettre l'évènement `lbp` du contexte vers les sous arbres. Le résultat obtenu pour le terme `box` précédent est celui présenté à la figure 2.4 où l'utilisateur a cliqué sur la boîte de couleur cyan.

2.3.2 Un billard d'une balle en un coup

Nous souhaitons aller plus loin dans l'extension du langage des boîtes en fabriquant une sorte de billard dont la table est formée de l'ensemble des boîtes assemblées et sur laquelle une balle se déplace suivant des règles simples mais bien définies. La table du billard est initialement fixe et statique. Lorsqu'un utilisateur clique à l'intérieur d'une boîte, cela crée ou réinitialise la balle du billard qui va s'y déplacer avec une vitesse constante et suivant une direction (donnée par un angle) pour :

- soit rebondir et changer de direction (en restant dans la même boîte) si elle rencontre une frontière non commune à aucune autre boîte (bordure de la boîte englobante),
- soit traverser et conserver la même direction si elle rencontre une frontière commune à deux boîtes assemblées.

Les trous du billard sont ainsi formés des frontières communes de l'ensemble des boîtes assemblées.

L'implémentation de ce langage a priori simple nécessite la prise en compte de plusieurs éléments et contraintes. D'un point de vue programmation, ceci ne constitue pas un problème insurmontable. Cependant, puisque nous souhaitons reduire le temps et l'effort de programmation mais surtout, rendre la solution réutilisable, nous avons tout naturellement opté pour la technique des grammaires attribuées. Nous reprenons donc la grammaire attribuée des boîtes définie dans la section précédente dont les règles sémantiques cette fois-ci, écrites dans le langage FAL, décrivent le comportement du billard tel que décrit ci-dessus. Cette grammaire explicite de ce fait les fonctions d'interprétation de la structure d'algèbre des boîtes correspondante. Cette approche a pour avantage de garder les mêmes structures de données, la même structure d'algèbre et le même schéma de récursion associé. Les modifications ne sont alors effectuées que sur le domaine sémantique et les règles sémantiques associées.

Le domaine sémantique pour cette structure d'algèbre est une fonction qui à partir du point d'ancrage de la boite englobante associe à un évènement en entrée (correspondant au lancement du jeu et portant la position et la direction de départ de la balle), un comportement en sortie traduisant le déplacement de la balle dans la boîte comme indiqué sur la figure 2.5. Plus concrètement, les attributs qui vont entrer en jeu se définissent comme suit :

• Les informations comme le point d'ancrage de la boîte englobante, l'évènement en entrée ainsi que la vitesse de la balle sont extérieures à la boîte c'est-à-dire qu'elles proviennent du contexte. Ce sont des attributs hérités. Nous considérons la vitesse de la balle constante, c'est pourquoi elle n'apparaîtra pas dans le type du domaine sémantique. Le point d'ancrage d'une boîte de type `Position = (Float,Float)` représente son origine. L'évènement d'entrée de type `Input = Event (Position, Direction)` porte le point de départ $p0$ (x_0, y_0) de la boule et la direction `d0(x,y)::(Float,Float)` qu'elle doit prendre.

• La largeur et la hauteur de la boîte sont des attributs synthétisés.

• On ajoute à cela des informations précisant si un point (ce sera la position courante de la balle du billard représentée par son origine) se trouve ou non à l'une des frontières de la boîte englobante. Ce qui permet de définir sous forme de prédicats les attributs synthétisés `nB, sB, eB, wB :: Position -> Bool` indiquant respectivement si le point

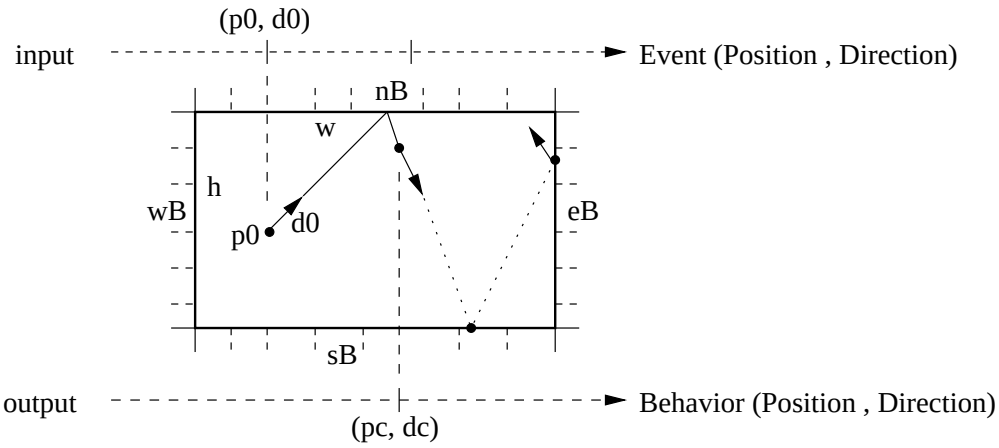

FIGURE 2.5 – Domaine sémantique du billard

passé en paramètre se trouve à la frontière nord, sud, est ou ouest de la boîte.

• L'attribut synthétisé `output` est l'interface de sortie qui est un comportement de type `Output = Behavior (Position, Direction)` contenant à tout moment la position courante `pc` de la balle et la prochaine direction `dc`. Cette dernière dépend de la frontière de la boîte et de la direction actuelle de la balle. En effet, lorsque que la balle rencontre un trou, elle garde la même direction ; et lorsqu'elle rencontre une frontière verticale (resp. horizontale), sa direction horizontale (resp. verticale) s'inverse par la fonction `negate`.

• Le dernier attribut `res` de type `Behavior Picture` est une image qui synthétise le résultat final observable.

Ce qui nous permet de définir le domaine sémantique `CompSem` suivant :

```
-------------------------------------------------------------------------
type Position  = (Float , Float)
type Direction = (Float , Float)
type Input     = Event (Position , Direction)
type Output    = Behavior (Position , Direction)
data Synth = Synth {w, h :: Float, res :: Behavior Picture,
                    nB, sB, eB, wB :: Position -> Bool, output :: Output}
type CompSem = Position -> Input -> Synth
-------------------------------------------------------------------------
```

Comme l'algèbre et le schéma de récursion associé n'ont pas changé, l'interprétation d'une boîte élémentaire est donnée par le combinateur `elemComp` du tableau 2.3. Cette interprétation repose essentiellement sur la fonction `behComp` qui décrit le comportement élémentaire d'une boîte et permet de calculer l'attribut synthétisé `output` qui représente l'interface de sortie du composant élémentaire. L'application de la fonction `behComp` à ses arguments, `behComp w h p0@(x0,y0) p@(abs,ord) d0@(xvel0,yvel0))`, signifie que la position courante `pc@(xpos,ypos)` de la balle se déplace dans une boîte (de largeur `w`, de hauteur `h` et ancrée au point `p0`) à partir de la position `p` suivant une direction initiale `d0`. Le déplacement de ce point est assuré par la fonction `integral` sous l'influence d'une force accélérée par la vitesse `(xvel,yvel)`. Ainsi, lorsque la balle atteint par exemple

une frontière verticale, l'évènement `xbounce` se produit et la vitesse `xvel` en `x` qui décrit également la direction change de signe par la fonction `negate`. Le résultat final est un comportement dont les valeurs sont la position courante `(xpos,ypos)` de la balle et la direction courante `(xvel,yvel)`. Les autres calculs du combinateur `elemComp` indiquent comment un point `(x,y)` appartient à l'une des frontières d'une boîte élémentaire.

Tableau 2.3 Interprétation du billard dans une boîte élémentaire

```
elemComp :: Float -> Float -> Color -> CompSem
elemComp w h c p0@(x0,y0) input = Synth w h res nb sb wb eb output
  where output = (behComp w h p0 (x0,y0) (1,1)) `switch`
                          input =>> \(pos,dir) -> behComp w h p0 pos dir)
        nb (x,y) = (x0-w/2 <= x && x<=x0+w/2) && (y >= y0+ h/2)
        sb (x,y) = (x0-w/2 <= x && x<=x0+w/2) && (y <= y0 -h/2)
        wb (x,y) = (x <= x0-w/2) && (y0-h/2 <= y && y <=y0+h/2)
        eb (x,y) = (x >= x0+w/2) && (y0-h/2 <= y && y <=y0+h/2)
        res = paint (constB c) (translate ((constB x0),(constB y0))
                                          (rec (constB w) (constB h)))

behComp :: Float -> Float -> Position -> Position -> Direction
                                -> Behavior (Position,Direction)
behComp w h (x0,y0) (abs,ord) (xvel0,yvel0) =
   let xvel = xvel0 `stepAccum` xbounce ->> negate
       yvel = yvel0 `stepAccum` ybounce ->> negate
       xpos = (constB abs) + integral xvel
       ypos = (constB ord) + integral yvel
       xbounce = when ((xpos >=* constB (x0+w/2)) ||*
                       (xpos <=* constB (x0-w/2)))
       ybounce = when ((ypos >=* constB (y0+h/2)) ||*
                       (ypos <=* constB (y0-h/2)))
   in  pairB (pairB xpos ypos) (pairB xvel yvel)
```

Lorsqu'on compose (verticalement ou horitalement) deux boîtes, il faut prendre en compte plusieurs contraintes pour que le billard continue de fonctionner comme prévu. En particulier, on doit pouvoir localiser la présence ou non de la balle dans l'une des boîtes. Nous utilisons un attribut local `loc` (pour `localisation`) qui représente un comportement de type booléen (`Behavior Bool`) et prend la valeur `True` (respectivement `False`) pour indiquer la présence de la balle dans la première (respectivement dans la deuxième) boîte. Ainsi, le comportement du billard dans la boîte englobante donné par l'attribut synthétisé `output` est filtré par la valeur de `loc` et se réduit au comportement `output1` de la première boîte si `loc` vaut `True` et `output2` de la seconde boîte sinon. Plusieurs situations permettent à la balle de changer de localisation. En effet :

- L'attribut `loc` prend la valeur `True` lorsque l'évènement `init1` de type `Input` s'est produit dans la première boîte pour y réinitialiser le jeu, ou parce que l'évènement `cross2to1` (figure 2.6 - b) de type `Event()` s'est produit, indiquant le fait que la

balle a traversé la frontière commune partant de la deuxième boîte à la première.
- Il prendra la valeur `False` dans les cas contraires correspondants i.e. lorsque l'évènement `init2` s'est produit dans la deuxième boîte, ou parce que la balle a traversé la frontière de la première à la deuxième boîte ; situation matérialisée par l'évènement `cross1to2` (figure 2.6 - a).

Ce changement de localisation de la balle est calculé par un attribut local `change` de type `Event Bool` qui implémente les quatre alternatives considérées. La traversée de la

FIGURE 2.6 – a). Cross 1 to 2 b). Cross 2 to 1

frontière commune de la boîte 1 à la boîte 2, indiquée sur la figure 2.6 - a) s'effectue lorsqu'à l'instant `t-1`, la balle identifiée par son point d'ancrage `p` se trouve au nord de la deuxième boîte (`nB2 p == True`) et que immédiatement après (à l'instant `t`), elle se trouve au sud de la première boîte (`sB1 p == True`). Pour vérifier ces conditions, il est nécessaire de synchroniser ces deux comportements en décalant le premier d'une unité de temps. Nous avons enrichi le langage FAL avec la fonction `delayB` suivante qui retarde un comportement d'une unité de temps.

```
delayB :: a -> Behavior a -> Behavior a
delayB a (Behavior f) = Behavior (\uts -> a : (f uts))
```

Nous matérialisons sur la figure 2.7 les conditions de traversée d'une frontière commune à deux boîtes assemblées verticalement. Les valeurs portées par le comportement `nB2 p` en rouge sont décalées d'une unité de temps comme représenté sur la courbe `nB2 (delayB p2 p)`. Ce qui permet de garder cette valeur à l'instant suivant immédiat afin de vérifier toutes les conditions de traversée de la frontière commune.

La mise à jour des attributs locaux `init1` et `init2` qui permettent de réinitialiser le jeu au clic de la souris « dans une boîte quelconque » est effectuée par la fonction `filterE_`, que nous avons ajoutée au langage FAL pour la circonstance. En effet, le clic de la souris est capturé par l'attribut hérité `input :: Event (Position, Direction)` et la fonction `filterE_` "regarde" par la fonction `inBox` dans quelle boîte on a cliqué afin d'orienter l'évènement vers le bon endroit (`init1` ou `init2`). L'appel, `inBox w h p0 p`, teste si le point `p` est dans la boîte de largeur `w`, de hauteur `h` et centrée à `p0`. Les évènements d'entrée `input1` (respectivement `input2`) de la première (respectivement de la deuxième) composante (attributs hérités), sont alors formés soit de l'évènement de réinitialisation `init1` (respectivement `init2`), soit de l'évènement `cross2to1` (respectivement `cross1to2`).

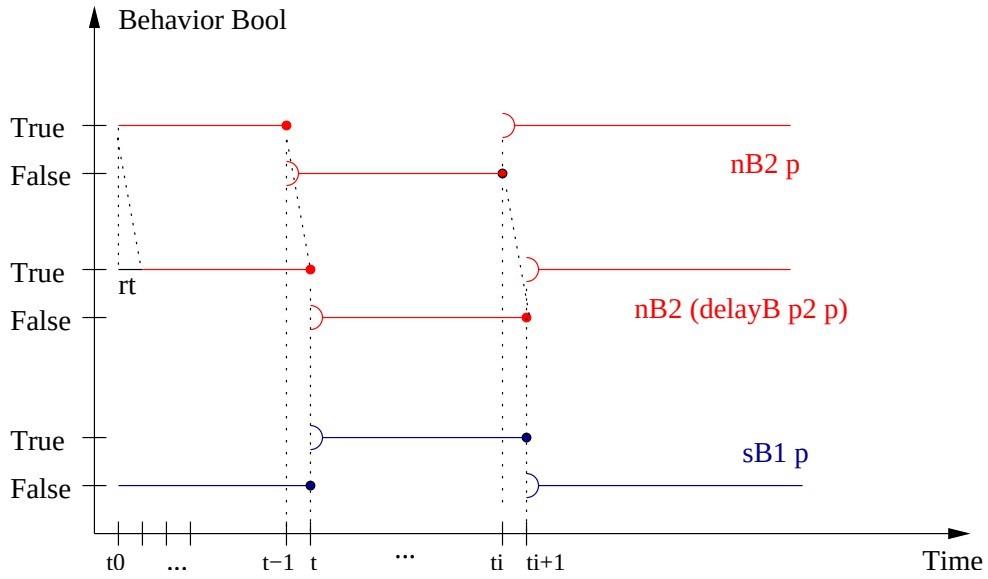

FIGURE 2.7 – Traversée d'une frontière commune à deux boîtes positionnées verticalement : notion de retard

```
-----------------------------------------------------------------------
inBox :: Float -> Float -> Position -> Position -> Bool
inBox w h (x0,y0) (x,y) = (x0-w/2 <= x &&  x <= x0+w/2) &&
                          (y0-h/2 <= y &&  y <= y0+h/2)
filterE :: Event a -> (a -> Bool) -> (a -> b) -> Event b
filterE (Event fe) p f = Event (\uts -> map (appl p f) (fe uts))
   where appl p f Nothing = Nothing
         appl p f (Just a) = if p a then Just (f a) else Nothing
filterE_ e p = filterE e p id
-----------------------------------------------------------------------
```

Au bout du compte, l'interprétation traduisant le jeu de billard pour une composition verticale de deux boîtes est donnée par le combinateur `vertComp` du tableau 2.4. Les frontières ouest `wB` et est `eB` de la boîte composite dépendent de la façon dont les boîtes ont été calées lors de leur assemblage ; tandis que la frontière sud `sB` (respectivement nord `nB`) de la boîte composite est la même que la frontière sud `sB2` de la deuxième (respectivement nord `nB1` de la première) boîte. Les règles sémantiques pour le calcul des autres attributs (`w`, `h`, `x1`, `x2`, `y1` et `y2`) restent sans changement.

Tableau 2.4 Jeu de billard pour deux boîtes assemblées verticalement

```
vertComp ::  VPos ->  CompSem -> CompSem -> CompSem
vertComp vpos sem1 sem2 (x0,y0) input =
                                Synth w h res nB sB eB wB output
 where output = condB loc output1 output2
       change = (init1 ->> (const True)) .|.
                (init2 ->> (const False)) .|.
                (cross2to1 ->> (const True)) .|.
                (cross1to2 ->> (const False))
       loc = True `stepAccum` change
       p = fstB output
       cross1to2 = when (((lift1 nB2) (delayB (x1,y1) p)) &&*
                         (((lift1 sB1) p)))
       cross2to1 = when (((lift1 sB1) (delayB (x2,y2) p)) &&*
                         (((lift1 nB2) p)))
       init1 = filterE_ input ((inBox w1 h1 (x1,y1)) . fst)
       init2 = filterE_ input ((inBox w2 h2 (x2,y2)) . fst)
       input1 =  init1 .|. (snapshot_ cross2to1 output)
       input2 =  init2 .|. (snapshot_ cross1to2 output)
       Synth w1 h1 res1 nB1 sB1 eB1 wB1 output1 = sem1 (x1,y1) input1
       Synth w2 h2 res2 nB2 sB2 eB2 wB2 output2 = sem2 (x2,y2) input2
       res = res1 `over` res2
       nB  = nB1
       sB  = sB2
       wB p = case vpos of
             Left_ -> (wB1 p)||(wB2 p)
             _     -> if w1>w2 then (wB1 p)
                else if w2>w1 then (wB2 p) else (wB1 p)||(wB2 p)
       eB p = case vpos of
             Right_ -> (eB1 p)||(eB2 p)
             _      -> if w1>w2 then (eB1 p)
                else if w2>w1 then (eB2 p) else (eB1 p) || (eB2 p)
       ...
```

On procède par un raisonnement analogue pour dériver le combinateur `horComp` comme indiqué dans le tableau 2.5.

Tableau 2.5 Jeu de billard pour deux boîtes assemblées horizontalement

```
horComp ::  HPos ->  CompSem -> CompSem -> CompSem
horComp hpos sem1 sem2 (x0,y0) input = Synth w h res nB sB eB wB output
 where output = condB loc output1 output2
       change = (init1->>(const True)) .|. (init2 ->> (const False)).|.
                (cross2to1 ->>(const True)).|.(cross1to2->> (const False))
       loc = True `stepAccum` change
       p = fstB output
       cross1to2 = when (((lift1 wB2) (delayB (x1,y1) p)) &&*
                         (((lift1 eB1) p)))
       cross2to1 = when (((lift1 eB1) (delayB (x2,y2) p)) &&*
                         (((lift1 wB2) p)))
       init1 = filterE_ input ((inBox w1 h1 (x1,y1)) . fst)
       init2 = filterE_ input ((inBox w2 h2 (x2,y2)) . fst)
       input1 =  init1 .|. (snapshot_ cross2to1 output)
       input2 =  init2 .|. (snapshot_ cross1to2 output)
       Synth w1 h1 res1 nB1 sB1 eB1 wB1 output1 = sem1 (x1,y1) input1
       Synth w2 h2 res2 nB2 sB2 eB2 wB2 output2 = sem2 (x2,y2) input2
       res = res1 `over` res2;
       wB = wB1
       eB = eB2
       nB p = case hpos of
                  Top -> (nB1 p) ||(nB2 p)
                  _   -> if (h1>h2) then (nB1 p)
                            else if (h2>h1) then (nB2 p)
                                            else (nB1 p)||(nB2 p)
       sB p = case hpos of
                  Bottom -> (sB1 p)||(sB2 p)
                  _   -> if (h1>h2) then (sB1 p)
                           else if (h2>h1) then (sB2 p)
                                           else (sB1 p)||(sB2 p)
       ...
```

La fonction `result` ci-dessous interprète un terme de type `Box Color` comme une image simulant le comportement du jeu de billard. Elle utilise l'algèbre des boîtes pour décrire la fonction qui donne le résultat final. Le point d'ancrage (attribut hérité) est initialisé à `(0,0)` , l'évènement en entrée est le clic `lbP` du bouton gauche de la souris et la direction initiale (à chaque fois que le jeu est réinitialisé) est `(1,1)`. La balle du billard (`ball`) est un rectangle noir qui suit la position courante donnée pas l'attribut synthétisé `output` obtenu en interprétant la boîte englobante par la structure d'algèbre définie par les combinateurs `elemComp`, `vertComp` et `horComp`.

```
-----------------------------------------------------------------------
result :: Box Color -> Behavior Picture
result box = ball `over` res
```

```
  where Synth _ _ res _ _ _ _ output =
                                evalBox (AlgBox elemComp vertComp horComp)
                                     box (0,0) (lbP =>> (\p -> (p,(1,1))))
        ball = paint black (translate (xpos,ypos) (rec 0.15 0.15))
        behPos = fstB output
        (xpos,ypos) = (fstB behPos, sndB behPos)
-------------------------------------------------------------------------
```

La figure 2.8 représente une partie du jeu de billard au moment où la balle est sur le point de traverser la frontière commune entre les boîtes rouge et violette.

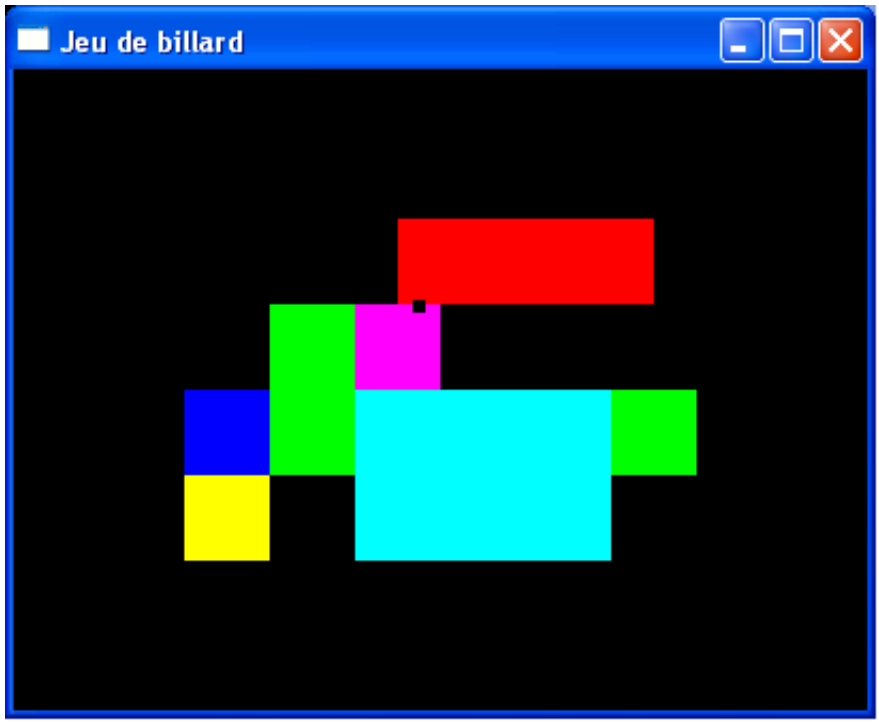

FIGURE 2.8 – Jeu de billard : traversée d'une frontière commune

2.3.3 Similarité avec la programmation par composants

La description précédente permet de mesurer la difficulté à laquelle on aurait été confronté s'il fallait donner une spécification directe de la structure d'algèbre décrivant le comportement du billard. L'utilisation des grammaires attribuées s'est avérée bénéfique et on a pu réduire considérablement le temps de travail et l'investissement intellectuel tout en offrant une solution facilement adaptable et réutilisable. La similarité avec la programmation par composants se situe au niveau de la façon avec laquelle la solution a été construite. En effet, pour concevoir ce jeu de billard assez complexe, nous sommes partis de la description d'un langage simulant le comportement le plus élémentaire d'un billard formé d'une boîte de base. Nous avons ensuite décrit la sémantique comportementale d'un billard formé d'une composition verticale ou horizontale de deux boîtes quelconques.

Ces trois éléments ont été suffisants pour constituer des composants disponibles « sur l'étagère » permettant d'obtenir des billards encore plus complexes par leur forme, mais produisant le même résultat.

Cependant, pour mieux appréhender cette similarité, nous considérons maintenant le domaine sémantique du billard pour la structure d'algèbre correspondante comme un composant. Ce qui s'interprète facilement par la représentation graphique donnée par la figure 2.5 de la page 60. Pour rendre de tels composants permanents et donc disponibles pour de futures utilisations, nous encapsulons leur sémantique dans une structure de données Haskell appelée `Component`[4] et définie comme suit :

```
-------------------------------------------------------------------------
data Component = Comp {width, height :: Float,
                       northBorder :: Position -> Position -> Bool,
                       southBorder :: Position -> Position -> Bool,
                       westBorder  :: Position -> Position -> Bool,
                       eastBorder  :: Position -> Position -> Bool,
                       behavior    :: Position -> Input -> Output}
-------------------------------------------------------------------------
```

Les fonctions `width` et `height` permettent de sélectionner la largeur et la hauteur d'un composant de type `Component`. Le premier argument de type `Position` des sélecteurs de frontières d'un composant représente le point d'ancrage du composant. Sa présence explicite ici est due au fait que le calcul des frontières d'un composant dépend de la position de celui-ci repérée justement par son point d'ancrage. Cet argument a par contre une signification particulière pour le sélecteur `behavior` puisqu'il traduit le résultat de calcul du graphe HS pour l'unique catégorie syntaxique `Box` considérée. C'est ce même calcul du graphe HS qui a permis de l'omettre dans la définition des sélecteurs `width` et `height`.

Finalement le programme Haskell qui permet de traduire la sémantique du jeu de billard décrite par `CompSem` en un composant de type `Component` est donné par :

```
-------------------------------------------------------------------------
-- Evènement nul
nullevent :: Event a
nullevent = Event (\uts -> repeat Nothing)

-- Traduction de la sémantique du billard en un composant
sem2comp :: CompSem -> Component
sem2comp sem = Comp wt ht nb sb wb eb beh
           where wt    = w (sem (0,0) nullevent)
                 ht    = h (sem (0,0) nullevent)
                 nb p0 = nB (sem p0 nullevent)
```

[4] L'avantage de définir les éléments qui construisent un composant comme des sélecteurs permet de rendre implicite la sélection des différentes composantes correspondantes. Par exemple les composantes *width*, *height* sont en réalité de type :: $Component \rightarrow Float$

```
                sb p0 = sB (sem p0 nullevent)
                wb p0 = wB (sem p0 nullevent)
                eb p0 = eB (sem p0 nullevent)
                beh p0 input = output (sem p0 input)
-------------------------------------------------------------------------
```

La fonction `nullevent` qui ne produit aucun évènement au cours du temps est utilisée uniquement pour répondre aux contraintes de typage du langage Haskell. Elle n'a donc aucune signification particulière et n'influe en aucun cas sur le résultat de la traduction. On note aussi l'avantage de faire d'abord un calcul des graphes HS pour décider du domaine sémantique associé au symbole grammatical `Box` ; et l'importance de raisonner en termes de composants.

2.3.4 Composition horizontale de langages dédiés

Dans le cadre du développement modulaire et réutilisable des langages dédiés, nous avons observé que ceux-ci peuvent être décrits de différentes façons à l'aide du formalisme des grammaires attribuées, vu comme métalangage pour la spécification des DSLs. Nous avons montré par ailleurs comment l'écriture de la spécification (par une grammaire attribuée) d'un langage dédié pouvait se faire en utilisant un autre langage dédié (lui aussi décrit probablement par une autre grammaire attribuée). L'exemple typique a été celui de la construction dans le langage FAL[5] d'un jeu de billard a priori simple mais assez édifiant. Si le langage FAL avait été lui-même décrit par une grammaire attribuée (ce que nous estimons être désormais le cas pour la spécification de la plupart des langages dédiés, surtout lorsqu'on souhaite avoir une certaine flexibilité dans leur assemblage et leur réutilisation), on aurait obtenu une sorte de stratification horizontale de langages dédiés (comme indiqué sur la figure 2.9) dans laquelle on organise les langages en strates horizontales (signature d'entrée à gauche, et signature de sortie à droite) de telle sorte que dans chaque strate on peut utiliser des notations produites dans les strates antérieures.

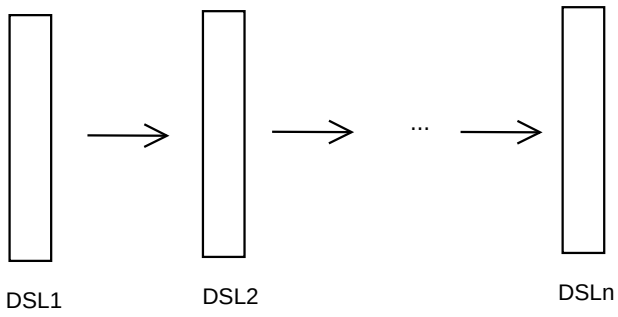

FIGURE 2.9 – Stratification horizontale de langages dédiés

Cette approche est liée à la notion de composition descriptionnelle des grammaires attribuées décrite par Ganzinger & Giegerich [62] dont le lien avec la déforestation de

[5] Malheureusement ici, les auteurs de FAL ne l'ont pas décrit à l'aide d'une grammaire. Notons cependant qu'il est possible de le spécifier par une grammaire attribuée comme nous avons pu le faire dans la première partie de ce chapitre pour les combinateurs fonctionnels d'analyse.

programmes fonctionnels définie par P. Wadler [63] a été établi par Correnson, Parigot & al. [64, 65]. On peut aussi considérer, pour illustrer une autre façon d'étendre un langage dédié, une stratification verticale de langages dédiés liée à une décomposition modulaire des grammaires attribuées portant sur leur signature d'entrée comme mentionnée dans les travaux de Badouel & Djeumen [66].

2.4 Typage de langages dédiés en vue de leur réutilisation

Nous voulons nous baser sur la technique de construction de langages dédiés à partir de grammaires attribuées pour dégager un certain nombre de méthodes pour l'assemblage et la réutilisation de langages. La volonté est de conduire la même démarche méthodologique au niveau des langages que ce qui est classiquement fait au niveau des composants logiciels : Comment peut-on créer de nouveaux langages par composition de langages réutilisables existants [20] ? Il faut donc préciser les caractéristiques des langages et les informations qu'il faut y attacher pour qu'un langage dédié soit réutilisable. Comme on a vu depuis le chapitre 1 jusqu'aux sections précédentes, la réutilisation d'un langage dédié met en jeu des aspects très différents. Ce qui rend la question très complexe. Nous essayons dans cette partie de tirer des règles générales, bien que celles-ci, il faut le noter, restent encore embryonnaires et difficilement généralisables.

2.4.1 Composant et système de types

Un composant est un programme informatique encapsulé dans une interface qui indique son type, ses ports d'entrées-sorties et éventuellement son comportement. Le typage est une technique qui permet d'associer des spécifications (types) à des programmes et de détecter automatiquement une classe d'erreurs dans ces programmes. Un programme peut être mal typé (par exemple on essaie d'additionner deux quantités non homogènes) ou, bien typé mais incorrect (i.e. qu'il ne produit pas le résultat attendu). L'intérêt principal du typage est qu'il garantit par une propriété de sûreté du typage, démontrée par le concepteur de tout système de types, que tout programme bien typé ne peut pas planter.

Un type est équivalent à un ensemble de données ayant la même forme et défini de manière compositionnelle. Ce qui lui donne une simplicité dans l'analyse des programmes. Par ailleurs, de part son caractère compositionnel et l'exploitation des notions duales de polymorphisme paramétrique et d'abstraction de types, le typage favorise la modularité, c'est-à-dire le découpage des programmes en unités indépendantes et complémentaires, activité centrale pour le développement de systèmes logiciels complexes. La construction d'un type se fait en général de façon compositionnelle [67] : on part des types de base (par exemple `int`, `bool`, `string`, etc.) auxquels on ajoute des types composés $A \times B$ (paires), $A + B$ (ou), $A \to B$ (type fonction), $List(A)$ (type liste), etc. Puis on définit les règles de typage avec la notation suivante appelée jugement de typage ou séquent :

$$x_1 : A_1, \ldots, x_n : A_n \vdash expr : B$$

qui se lit : « en supposant que les variables $x_1, \ldots, x_n$ ont les types $A_1, \ldots, A_n$ respectivement, l'expression *expr* est bien typée et son type est B ». Les règles de types sont alors définies par la notation suivante

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}$$

dans laquelle les P_i et C sont des séquents valides. Cette notation se lit : « si les prémisses $P_1, \ldots, P_n$ sont toutes vraies, alors la conclusion C est vraie ». Il existe en général deux sortes de typage :

Le typage dynamique : pendant l'exécution du programme, chaque opération vérifie les types de ses arguments, et arrête le programme si une erreur de type est détectée. L'avantage de ce type de typage est que le programme ne va pas continuer avec des données mal formées. Par contre, les erreurs de type sont détectées trop tard pendant l'exécution.

Le typage statique : les types sont vérifiées lors de la compilation du programme, avant toute exécution de tout ou d'une partie de celui-ci. Toute erreur de type arrête la compilation. L'avantage ici est que les erreurs sont détectées très tôt au cours du cycle de développement, mais ce typage peut rejeter à tort des programmes corrects.

La vérification de types exige que le programmeur déclare les types des variables et des fonctions et que le typeur vérifie la cohérence de ces déclarations. Il est parfois demandé au typeur de "deviner" les types des variables et des fonctions à partir de leurs utilisations dans le programme i.e. de rendre automatique (ou presque) la recherche du (des) type(s) qu'admet un programme : on parle d'inférence de types. La plupart des langages fonctionnels fortement typés comme Haskell infèrent les types de leurs programmes. Étant donnée la nature compositionnelle du typage, le problème d'inférence de types associé à un fragment de programme admet une solution si, et seulement si, les sous problèmes associés aux sous fragments qui le composent admettent également des solutions.

Notre but dans cette partie est de modéliser un système de types basé sur le calcul des grammaires attribuées permettant d'identifier des langages dédiés vus comme composants logiciels, afin de permettre leur réutilisation et leur assemblage.

2.4.2 Typage de langages dédiés

La notion de type d'un composant que nous définissons ici repose essentiellement sur les types de base du langage Haskell. Elle est basée en plus sur une relation bipartie Γ, que nous définissons entre deux ensembles disjoints I et O ($\Gamma \subseteq I \times O$ telle que $I \cap O = \emptyset$) où I et O représentent l'interface du composant et indiquent respectivement ses ports d'entrée et ses ports de sortie.

Dans le cadre des langages dédiés décrits par les grammaires attribuées, on associe à chaque symbole grammatical X un composant C. La définition du type Γ d'un tel composant est guidée par les graphes $HS(X)$ (graphes hérités-synthétisés associés au symbole grammatical X). D'une manière générale, les attributs hérités constituent l'ensemble I de ses ports d'entrée et les attributs synthétisés l'ensemble O de ses ports de sortie. Il est nécessaire que ces ensembles soient disjoints pour éviter les configurations cycliques, sauf

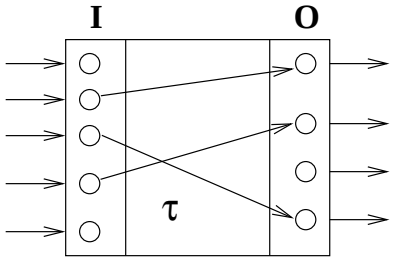

FIGURE 2.10 – Notion de composant et son interface

dans le cas où certains ports sont des attributs locaux. Finalement, le graphe $\Gamma \subseteq I \times O$ qui décrit le type d'un composant C (on note $C :: \Gamma$) associé à un symbole grammatical X est défini par : $(x, y) \in \Gamma$ si, et seulement si :

- x est un attribut hérité de X
- y est un attribut synthétisé de X
- Il existe un chemin de x vers y dans le graphe de dépendance d'un arbre de dérivation de racine X.

2.4.3 Composition (assemblage) de langages dédiés

Soient deux composants $C_1 :: \Gamma_1$ et $C_2 :: \Gamma_2$ associés aux symboles X_1 et X_2 de la production $P : X_0 \rightarrow X_1\, X_2$ d'une grammaire attribuée. La question que l'on se propose de répondre ici est celle de savoir dans quelles conditions ces deux composants sont composables ?

On remarque de prime abord que les règles sémantiques associées à la production constituent les règles opérationnelles qui permettent de coller les composants. Ainsi, en plus des graphes $HS(X)$ associés aux différents symboles grammaticaux mis en jeu, la composition de composants est liée au graphe de dépendances locales $GDL(P)$ associé à la production P.

Les composants $C_1 :: \Gamma_1 \subseteq I_1 \times O_1$ et $C_2 :: \Gamma_2 \subseteq I_2 \times O_2$ (vus comme langages dédiés) sont composables (comme indiqué sur la figure 2.11) avec pour interface de composition $\rho = (I_1 \cap O_2) \cup (O_1 \cap I_2)$ si les deux conditions suivantes sont vérifiées :

1. Leurs ports d'entrée et leurs ports de sortie, pris séparément, sont disjoints. Autrement dit $I_1 \cap I_2 = \emptyset = O_1 \cap O_2$.
2. La fermeture de leurs relations biparties $(\Gamma_1 \cup \Gamma_2)^*$ est acyclique.

et on note $C = C_1 \otimes C_2 :: \Gamma \subseteq I \times O$ le langage résultat de la composition associé au symbole X_0 de la production P, ou si l'on préfère

$$\otimes \;=\; \frac{C_1 :: \Gamma_1 \qquad C_2 :: \Gamma_2}{C = C_1 \otimes C_2 :: \Gamma}$$

avec

- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ i.e. $I_1 \cup I_2 \setminus \rho$
- $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$ i.e. $O_1 \cup O_2 \setminus \rho$ et
- $\Gamma = (\Gamma_1 \cup \Gamma_2)^* \cap (I \times O)$.

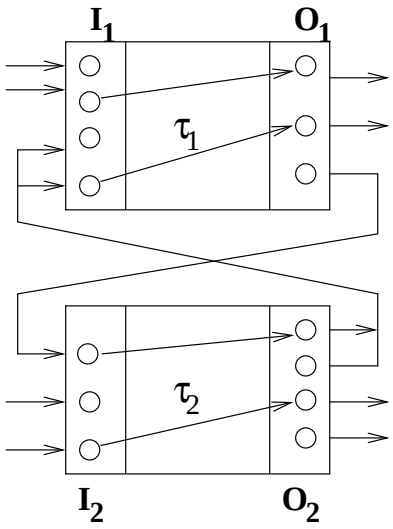

FIGURE 2.11 – Composition de deux composants (langages dédiés)

Somme de deux composants

La somme de deux composants est un cas particulier de la composition de langages dédiés. Elle décrit une union disjointe de deux langages sans transfert de valeurs des ports de sortie de l'un vers les ports d'entrée de l'autre, i.e. lorsque l'interface de composition ρ est vide. Elle est notée $C_1 \oplus C_2 :: \Gamma_1 \oplus \Gamma_2$ avec $\Gamma_1 \oplus \Gamma_2 = \Gamma_1 \uplus \Gamma_2 \subseteq (I_1 \uplus I_2) \times (O_1 \uplus O_2)$.

2.4.4 Contraintes de typage et propriétés

On définit d'une manière générale la composition de langages dédiés de la manière suivante. On associe à chaque port (d'entrée ou de sortie) un type s_i qui est en général un CPO, ou un objet de la catégorie de Kleisli d'une monade ou de la catégorie des calculs associés à un autre langage dédié. Pour chaque production $P : X_0 \rightarrow X_1\, X_2, \ldots, X_n$, on associe à chaque symbole X_i un composant $C_i :: \Gamma_i \subseteq I_i \times O_i$ avec $i \in \{1, \ldots, n\}$. Ce qui implique qu'on ne considère plus les catégories syntaxiques X_i mais plutôt les relations biparties Γ_i de telle sorte que l'opérateur *op* est vu comme une fonction op_C : $\Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_n \rightarrow \Gamma$.

Les composants C_1, C_2, ..., C_n sont alors composables si

- $I_i \cap I_j = \emptyset = O_i \cap O_j$ et $(\Gamma_i \cup \Gamma_j)^*$ est acyclique, avec $i, j \in \{1, \ldots, n\}$ et $i \neq j$.
- Ou mieux si $(GDL(op) \cup \cup_i \Gamma_i)^* \cap Syn(X_0) \subseteq \Gamma$.

Cette condition signifie que la projection, sur l'ensemble des attributs synthétisés de la racine (X_0), de la fermeture transitive du graphe de dépendances locales de l'opérateur *op* réuni avec les graphes $HS(X_i)$, $i \in \{1, \ldots, n\}$ doit être une partie de la relation bipartie Γ. On a alors :

$$op_C \;=\; \frac{C_1 :: \Gamma_1 \qquad C_2 :: \Gamma_2 \qquad \ldots \qquad C_n :: \Gamma_n}{op_C\,(C_1, C_2, \ldots, C_n) :: \Gamma}$$

$$si \quad (GDL(op) \cup \cup_i \Gamma_i)^* \cap Syn(X_0) \subseteq \Gamma$$

avec
- $I = \cup_j (I_j \setminus \cup_i O_i) \quad i \neq j$
- $O = \cup_j (O_j \setminus \cup_i I_i) \quad i \neq j$ et
- $\Gamma = (\cup_i \Gamma_i)^* \cap (I \times O)$ avec $i \in \{1, \ldots, n\}$.

Proposition 2.1. *$C :: \Gamma$ et $\Gamma \subseteq \Gamma' \Rightarrow C :: \Gamma'$*

Preuve. Supposons $C :: \Gamma$ et $\Gamma \subseteq \Gamma'$, montrons que $C :: \Gamma'$. Soit $C :: \Gamma$ un composant dont les éléments sont des couples (x, y) du graphe Γ. Pour tout $(x, y) \in \Gamma$, on a $(x, y) \in \Gamma'$ puisque $\Gamma \subseteq \Gamma'$. Ce qui conclut que $C :: \Gamma'$ ■

Relation d'ordre

On définit une relation d'ordre $\preceq$ sur le typage décrit précédemment par $\Gamma \preceq \Gamma'$, qui signifie que Γ est moins contraint (peu de règles opérationnelles) que Γ'.

Propriétés du typage

1.
$$\frac{op :: \Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_n \to \Gamma \qquad \Gamma'_i \preceq \Gamma_i \quad et \quad \Gamma \preceq \Gamma'}{op :: \Gamma'_1 \times \Gamma'_2 \times \ldots \times \Gamma'_n \to \Gamma'}$$

2.
$$\frac{c_1 :: \Gamma'_1 \qquad op\ c_1 :: \Gamma_2 \to \Gamma \qquad c_2 :: \Gamma'_2 \preceq \Gamma_2}{op\ c_1\ c_2 :: \Gamma'_1 \times \Gamma'_2 \to \Gamma}$$

2.5 Conclusion

L'idée était de concevoir un métalangage basé sur les grammaires attribuées pour la conception et l'assemblage de langages dédiés de manière la plus automatique possible. Afin de faire coexister un certain nombres de langages, le système de typage du langage Haskell s'est avéré insuffisant et nous l'avons enrichi en proposant un typage de langages dédiés assez simple et extrêmement lié à la technique de traduction des grammaires attribuées en programmes fonctionnels. Nous ne prétendons pas avoir répondu totalement à la question déjà abordée dans [20] à savoir : Comment créer de nouveaux langages par composition de langages réutilisables existants, compte tenu du caractère hétérogène des techniques d'assemblage étudiées. Mais nous pensons avoir posé les jalons d'une future généralisation qui prendra en compte toutes ces formes d'extensions de langages. L'accent sera mis dans les travaux futurs sur une définition des étapes de développement de Génie Logiciel par stratification de modèles de langages dédiés.

Chapitre 3

Étude de cas : un DSL pour l'édition de documents structurés

Sommaire

3.1 Introduction et principe de l'édition

Nous nous proposons dans ce chapitre de donner un exemple assez significatif de conception d'un langage dédié utilisant la méthode présentée dans le chapitre précédent. Le choix porté sur l'édition de documents structurés n'est pas anodin. Il provient des travaux que nous effectuons en équipe depuis un certain temps dans le cadre de la conception d'un atelier collaboratif d'édition de documents électroniques. Par ailleurs, le problème d'édition des documents structurés demande de prendre en compte plusieurs éléments dont la plupart, comme nous le verrons dans la suite, font intervenir les différents aspects (programmes avec effets, assemblage de composants, etc.) présentés dans les chapitres précédents.

Le problème de conception et d'implémentation des éditeurs dirigés par la syntaxe n'est pas nouveau et une vue d'ensemble de la question a été présentée dans [22]. Notre façon d'aborder le sujet est inspirée du langage *Modeless structure editor* de B. Sufrin & O. De Moor [23], un langage simple d'édition structurée.

Une donnée structurée est représentée intentionnellement sous la forme d'une structure arborescente caractérisée par une grammaire algébrique abstraite. On associe à cette structure des attributs (pour répondre aux questions sémantiques) qui peuvent être traitées indépendamment des aspects purement structuraux [24,25]. La manipulation de telles

données consiste essentiellement à modifier de façon dynamique et interactive leur structure ; ce qui entraîne éventuellement une réévaluation totale ou partielle des attributs. On conçoit donc un éditeur de données structurées comme un programme interactif qui réagit aux actions d'un utilisateur (pression d'une touche du clavier, clic de la souris, insertion d'une production, copier-coller, ...) pour modifier les objets édités et produire incrémentalement un résultat.

La représentation interne d'un document dans un éditeur interactif doit pouvoir être localisable (grâce à un point focal représentant à tout moment un nœud du document sur lequel les actions courantes d'édition s'effectuent) et partiellement définie (puisque le document en cours d'édition reste incomplet). Afin de pouvoir permettre cette localisation, nous représentons le document structuré par un Zipper, une structure de données introduite par Gérard Huet [3] permettant de représenter un arbre et son contexte, i.e. un arbre avec un point focal sur un nœud de celui-ci. Le but principal du Zipper est de faciliter la navigation dans le document, mais il permet aussi d'appliquer les actions d'édition directement à l'endroit indiqué dans le document (la position courante du curseur d'édition). Nous souhaitons manipuler de manière générique les opérations classiques d'édition comme les opérations de navigation, d'insertion ou de suppression de productions, etc. La généricité ici signifie que ces opérations sont canoniquement associées aux productions de la grammaire abstraite associée. Nous devons pouvoir aussi annuler et refaire chacune des opérations considérées.

En principe, un langage généraliste comme `C++` ou `Java` est suffisant pour aborder ce problème. Cependant, nous avons opté pour l'approche des combinateurs fonctionnels décrits par une grammaire attribuée. Nous ne sommes pas les premiers à aborder ce problème en suivant une approche par combinateurs. Oliver Braun & al. [68, 69] ont proposé le système `EdComb`, une bibliothèque de combinateurs d'éditeurs implémentée en Haskell qui permet d'assembler des éditeurs de manière compositionnelle. Nous en présentons les aspects fondamentaux dans la section qui suit tout en relevant à chaque fois les similitudes et les avantages qu'offre notre approche.

3.2 Le système EdComb de Braun & al.

Le système `EdComb` (Editor Combinators) [68] est le premier prototype de bibliothèque de combinateurs d'éditeurs implémenté en Haskell qui permet d'assembler de manière compositionnelle des éditeurs structurés plutôt que de les générer par un compilateur à partir d'une description. Leur structure est très analogue à celle des combinateurs d'analyse. `EdComb` est constitué d'éditeurs primitifs, de combinateurs pour la composition séquentielle et alternative d'éditeurs, et d'un mécanisme d'instanciation pour le rendu d'un éditeur. La description de ce système que nous abordons dans cette section ne se veut pas exhaustive ; elle donne simplement un aperçu diagonal de ce premier prototype de combinateurs d'éditeurs. Le lecteur intéressé pourra se référer au papier originel [68] pour les détails d'implémentation.

3.2.1 Typage d'un éditeur dans EdComb

Concrètement, « un éditeur pour un type `a` est un mécanisme qui permet de manipuler des valeurs de type `a` en maintenant un état interne définissant une vue sur la valeur courante » [69]. Le type de données d'un éditeur donné par `Editor v s a` est un enregistrement dans lequel environ vingt-neuf sélecteurs sont définis et divisés en groupes :

- Un groupe dit de navigation qui contient toutes les fonctions permettant de changer la position dans l'état de l'éditeur (`eLeft, eUp, ...`), et des prédicats permettant d'obtenir la position courante (`eLeftmost, eBottommost, etc.`).
- Le groupe de valeurs qui contient toutes les fonctions nécessaires pour construire un état de sortie d'une valeur et de calculer la valeur que l'état représente : `eBuild, eExtract, etc.`
- Le groupe `emplacement` est constitué des emplacements interne et externe : `eIsGroup, eFocus, etc.` L'emplacement externe est une liste d'entiers faisant référence à une partie d'un sous-éditeur logique d'un groupe ou d'une séquence d'éditeurs. Un emplacement interne consiste en une information d'emplacement passée de l'extérieur vers des composants d'éditeurs.
- Le groupe d'analyse contient comme son nom l'indique la fonction d'analyse `eParser` et beaucoup d'autres fonctions utiles, comme par exemple `eTokNames` qui calcule la liste de toutes les classes de lexèmes acceptables.
- La fonction `eReact` fournit la fonctionnalité centrale pour l'interaction avec l'éditeur.
- Le groupe d'affichage offre la fonctionnalité nécessaire au mécanisme d'instanciation.

Ainsi, la définition du type de données d'un éditeur est donnée en Haskell par le listing suivant :

```
-------------------------------------------------------------------------
 data Editor v s a = Editor {
       eRoot, eLeft, ..., :: s -> s
       ,eLeftmost, eTopmost, ..., :: s -> Bool
       ,eBuil :: a -> s, eExtract :: s -> a, eEmpty :: s, ...
       ,eIsGroup :: s -> Bool, eFocus :: Loc -> s -> s, ...
       ,eParser :: Parser s, eTokNames :: s -> [String], ...
       ,eReact :: Key -> React s, eCursorPos :: InPos -> s -> Pos
       ,eIdent :: InId -> s -> Id}
-------------------------------------------------------------------------
```

3.2.2 Composants primitifs et combinateurs de base

La signature des composants les plus primitifs est donnée par le listing suivant :

```
-------------------------------------------------------------------------
tagEditor   :: String -> Editor v () ()
trivEditor  = tagEditor ""

constEditor :: a -> String -> Editor v () a
tokenEditor :: String -> String -> Scanlet -> Editor v SPair String
```

```
readEditor :: (Read a,Show a)=> String -> String -> a -> Editor v SPair a
type SPair = (String, String)
-------------------------------------------------------------------------
```

Le composant `tagEditor` est un éditeur n'ayant aucun état et aucune valeur observables. Il est utilisé pour les mots clés et les séparateurs. Le composant `trivEditor` représente simplement une chaîne vide, tandis que l'éditeur `constEditor` permet d'éditer des valeurs constantes. Le `tokenEditor` est utilisé pour les lexèmes lexicaux non-constants (variables). Le type de son état interne `SPair` est formé de la valeur courante et de l'information sur la position courante dans cette valeur. Son premier argument de type `String` représente le nom de la classe de lexème concernée, et le second argument de même type est la chaîne initiale par défaut. Le *scanlet* (utilisé pour la complétion) permet de reconnaître les lexèmes de la classe pour laquelle le composant `tokenEditor` a été construit. L'éditeur `readEditor` se dérive de `tokenEditor` pour tous les types de données appartenant aux classes `Read` et `Show` définies dans le prélude standard de Haskell.

3.2.3 Composition des éditeurs

Les opérateurs <*> and </> représentent les combinateurs les plus basiques pour la composition séquentielle et alternative de deux éditeurs. Leur signature est donnée par :

```
-------------------------------------------------------------------------
(<*>) :: Editor v x a -> Editor v y b -> Editor v (PES x y) (a,b)
(<|>) :: Editor v x a -> Editor v y b -> Editor v (SES x y) (Either a b)

type PES x y = ((x, y), Bool)
type SES x y = Either SPair (Either x y)

type GES s = (s, Bool)
eGroup :: EditView v => String -> Editor v s a -> Editor v (GES s) a
-------------------------------------------------------------------------
```

L'état de l'éditeur résultat de la composition séquentielle de type `PES x y` contient un drapeau indiquant à chaque instant lequel des éditeurs possède le focus. Par contre, l'état `SES x y` pour la somme de deux éditeurs consiste en trois variantes possibles : la première alternative appelée `Left SPair`, indique que l'éditeur est dans un état non engagé ; tandis que les autres alternatives représentent l'éditeur final avec l'une de ses composantes étant effectivement valide. Le combinateur `eGroup` est utilisé pour combiner un nombre arbitraire d'éditeurs. L'état de l'éditeur résultat du regroupement contient un drapeau pour indiquer si l'ensemble du groupe est focalisé ou non.

Plusieurs autres combinateurs utiles sont dérivés des précédents : certains sont des juxtapositions d'éditeurs ignorant la valeur d'un composant ; et d'autres des compositions d'éditeurs permettant d'éviter la possibilité de focaliser un composant ignoré.

```
-------------------------------------------------------------------------
(*>) :: EditView v => Editor v x a -> Editor v y b -> Editor v (PES x y) b
(<*) :: EditView v => Editor v x a -> Editor v y b -> Editor v (PES x y) a
```

```
(>*>) :: EditView v => Editor v () a -> Editor v y b -> Editor v y b
(<*<) :: EditView v => Editor v x a -> Editor v () b -> Editor v x a
-----------------------------------------------------------------------
```

Illustration

Ces combinateurs de base sont assez suffisants pour présenter un exemple significatif. Considérons un éditeur pour une liste d'expressions arithmétiques définies par la grammaire algébrique suivante :

```
data Expr = Var String | Num Integer | BinOp Op Expr Expr
data Op = Plus | Minus | Mult | Div
```

L'éditeur résultat s'écrit directement en suivant la structure de la grammaire par :

```
mkExprEditor e = identEditor <|> intEditor
                               <|> eParen (e <*> opEditor <*> e)
opEditor = (tagEditor "+" <|> tagEditor "-")
            <|> (tagEditor "*" <|> tagEditor "/")
```

L'éditeur `identEditor` est un `tokenEditor` utilisant un scanlet pour les identificateurs. L'éditeur `intEditor` est une instance du composant `readEditor`, `opEditor` est une alternative qui permet d'identifier l'opérateur compris entre deux expressions, et enfin `eParen` est construit de la manière suivante :

```
eParen e = eGroup "BinOp" (tagEditor "(" >*> e <*< tagEditor ")")
```

3.2.4 Analyse critique du système EdComb

L'un des problèmes majeurs relevés par les concepteurs de `EdComb` a été le rendu (l'interprétation) de l'état de l'éditeur par le mécanisme d'instanciation i.e. le développement des interfaces utilisateurs pour les éditeurs, dont une tentative d'amélioration a d'ailleurs fait l'objet de la thèse en Master de Oliver Braun [69]. Par ailleurs Jan Scheffczyk [70] a aussi développé pour `EdComb` une première interface utilisateur orientée texte appelée `XTermInstance` et une deuxième interface reposant sur Tk appelée `FranTkInstance` qui utilise le système d'interface utilisateur graphique `FranTk` [71]. La vue sur le contenu de l'éditeur est définie dans ces instances, et toutes les entrées et sorties sont obtenues à travers des fonctions définies dans ces instances. Un inconvénient dans le choix du type de l'éditeur dans `EdComb` est lié au fait que toutes les caractéristiques d'un éditeur (navigation, rendu de son état, vue, etc.) ont été groupées ensemble comme sélecteurs dans la structure de données de l'éditeur elle-même. Ce qui implique par exemple une manipulation explicite des états de l'éditeur. La vue du contenu sera calculée entièrement après chaque interaction de l'utilisateur, alors qu'une modification incrémentale aurait été une meilleure solution.

D'autre part, les auteurs de `EdComb` ont sacrifié l'efficacité du Zipper, en se concentrant sur l'interface nécessaire aux combinateurs d'éditeurs. Nous proposons une approche qui met l'accent sur la structure de donnée Zipper rendant transparent (dans une monade d'état) la manipulation de l'état de l'éditeur. Notre approche est décrite par une grammaire attribuée et bénéficie de ce fait de son pouvoir d'expression.

3.3 Approche grammaticale des combinateurs d'éditeurs

Un de nos objectifs, contrairement au type de données d'éditeurs proposé dans `EdComb` est de séparer autant que possible les caractéristiques d'édition du type de données de l'éditeur. Notre but est de donner une sémantique « naturelle » au concept d'un éditeur, vu comme système réactif. L'état d'un éditeur sera alors un type de données polymorphe et pourra être représenté, comme par exemple dans le cas de l'édition interactive de données structurées, par une structure de données de type Zipper. Ce qui permet au programmeur de bénéficier directement des avantages offerts par le langage défini pour la manipulation du type de données Zipper. Des attributs sont par ailleurs associés à la structure du document pour répondre aux questions sémantiques comme par exemple l'obtention d'une vue du document en cours d'édition.

Nous utilisons une monade d'état pour rendre transparents à l'utilisateur les changements effectués sur la structure du document. Plus concrètement, nous définissons le type d'un éditeur par un transformateur de monades qui permet de paramétrer les effets produits au cours de l'édition, offrant ainsi une sorte de réutilisation des éditeurs liée au changement de la monade. L'éditeur est complètement décrit par une grammaire attribuée dont les attributs hérités et synthétisés correspondent aux commandes d'édition associées à l'éditeur. Afin de tester et valider notre DSL, nous avons implémenté pour le rendu des éditeurs une interface utilisateur en ligne de commande (voir annexe C) en prenant comme paramètre effectif à la monade définie dans le type de l'éditeur la monade des entrées-sorties (*IO monad*).

3.3.1 Typage d'un éditeur de base

Le principe de construction de tels systèmes assez complexes comme mentionné depuis le chapitre introductif de cette thèse repose sur la réutilisation systématique de composants jusqu'à obtention d'une solution ayant une complexité raisonnable. Il s'agit dans le cas présent de construire des éditeurs de base (pour un caractère, un entier, ...) ; puis de construire des combinateurs d'éditeurs qui vont permettre à partir des éditeurs de base, de construire en suivant la structure des documents à éditer des éditeurs pour ceux-ci.

Intuitivement, un éditeur est un programme interactif qui réagit aux actions utilisateur pour modifier son état interne, et qui produit un résultat correspondant à une vue de sa représentation interne [72]. Cet état interne est en général un Zipper représentant la

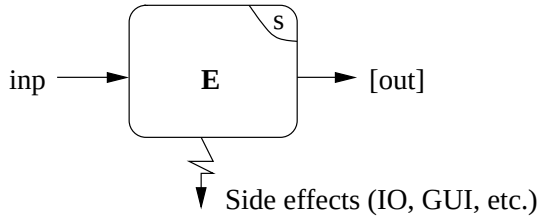

FIGURE 3.1 – Architecture d'un éditeur

structure arborescente du document en cours d'édition. Les attributs associés à chaque

nœud de cette structure permettent de décrire la sémantique du document. L'état de l'éditeur est donc un Zipper attribué.

Ainsi, un éditeur est un calcul produisant des effets secondaires qui prend des actions d'édition en entrée, et retourne en sortie une liste éventuellement vide de commandes. La figure 3.1 en est une représentation graphique. Les actions utilisateur sont transformées en opérations d'édition (définies dans un format propriétaire) et placées dans un tampon d'entrée `inp`. La représentation interne de l'éditeur est donnée par le type polymorphe `s`. Ce sera dans notre cas un Zipper attribué. Les commandes en sortie de l'éditeur qui seront associées à une certaine monade que nous préciserons, représentent des actions d'édition portées par le tampon de sortie de type `out`. Ces valeurs de sortie (lorsqu'elles existent) seront transmises au même éditeur ou à un autre éditeur (notion d'assemblage ou de composition) pour effectuer des tâches spécifiques.

En conformité avec le système de types défini au paragraphe 2.4.2 pour les langages dédiés, on observe dans l'analyse du domaine effectuée précédemment que l'interface d'un éditeur de base (vu comme un composant) est formée :

- Des actions d'édition de type `inp` qui constituent les ports d'entrée du composant, i.e. les attributs hérités de la grammaire associée.
- De la liste éventuellement vide de commandes d'édition (`[out]`) qui constituent ses ports de sortie.
- De son comportement élémentaire, décrit par les changements que son état interne (de type `s`) subit pendant son exécution. L'éditeur comporte aussi des effets secondaires produits au cours de son exécution et capturés par une monade définie dans son type.

Nous voulons rendre ce comportement transparent à l'utilisateur en utilisant la monade d'état qui offre une certaine flexibilité dans la consultation et la modification de l'état d'un système. Le concept d'état en langages de programmation s'identifie à toute abstraction de l'historique de l'exécution d'un programme. Il peut désigner par exemple un compteur incrémenté à chaque évaluation d'une variable, une chaîne de caractères contenant la trace d'exécution d'un programme, un tampon d'entrée/sortie, etc. Haskell définit pour cela un transformateur d'états par :

```
newtype ST s a = ST (s -> (s,a))
```

dans lequel `s` indique le type des objets représentant l'état et `a` le type de la valeur de retour de l'expression à évaluer. Cette évaluation dépend du contexte contrôlé par l'état `s`. Le résultat de l'exécution de la fonction `s -> (s,a)` est alors une paire formée de l'état suivant (éventuellement modifié) et de la valeur de l'expression associée. Ce transformateur d'états implémenté comme une instance de la classe des monades est alors très utile pour passer les informations d'états pendant les calculs en utilisant la méthode `bind` (>>=) de la classe `Monad`.

En plus de ce transformateur d'états, les programmes interactifs supposent des interactions avec le monde extérieur c'est-à-dire avec un utilisateur final. Nous combinons alors la monade d'états avec une certaine monade paramétrique `m` qui permettra de capturer les interactions avec le monde extérieur : le plus simple étant de l'instancier par la monade des entrées/sorties (`IO monad`) ; mais on pourra bien envisager de l'instancier par des interfaces utilisateur graphiques (GUI) comme FranTk [71] ou Fruit [73], ou le cas

échéant implémenter une interface graphique visuelle appropriée. Ce qui conduit à la définition du type suivant :

```
newtype StateT s m a = StateT {runstate :: s -> m (s,a)}
```

qui appartient à la classe des monades (`Monad`) et à celle des transformateurs de monades (`MonadTransformer`) :

```
-------------------------------------------------------------------------
instance Monad m => Monad (StateT s m) where
  -- return :: a -> StateT s m a
  return a = StateT $ \s -> return (s,a)
  -- (>>=) :: StateT s m a -> (a -> StateT s m b) -> StateT s m b
  m >>= f = StateT $ \s ->
        do{(s',a) <- runstate m s; runstate (f a) s'}

instance (Monad m) => MonadTransformer (StateT s) m  where
  -- lift :: m a -> StateT s m a
  lift m = StateT $ \s-> do{x <- m; return (s,x)}

-- avec
class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a
-------------------------------------------------------------------------
```

En tant que système réactif, le fonctionnement d'un éditeur ne sera déclenché que par une commande d'édition à travers ses ports d'entrée de type `a`. Un éditeur est formellement défini comme une fonction de ses ports d'entrée (attributs hérités) vers ses ports de sortie (attributs synthétisés) ; ces derniers ayant été, comme décrit précédemment, intégrés dans une monade d'état (voir figure 3.2). On obtient alors (en Haskell) le typage suivant pour un éditeur :

```
type Editor s m a b = a -> StateT s m [b]
```

Ce qui n'est rien d'autre que la traduction de la primitive suivante

$$editor \;=\; \frac{}{a \vdash editor \rhd [b]}$$

Un éditeur se définit donc comme une fonction qui réagit aux opérations d'édition de type `a`, pour modifier son état interne de type `s` (un Zipper attribué dans notre cas) et produire éventuellement des commandes de type `[b]` en effectuant des effets de type `m`.

Avec ce typage, on est déjà à mesure de décrire certains combinateurs nécessaires à la composition des éditeurs, sans avoir une idée précise de leur fonctionnement interne ; i.e. des combinateurs (généraux) agissant convenablement sur des structures polymorphes. Nous les complèterons plus loin, lorsque nous aurons décrit dans la section 3.4 le fonctionnement élémentaire d'un éditeur de base pour un document structuré, par d'autres combinateurs plus spécifiques agissant uniquement sur des structures de Zipper.

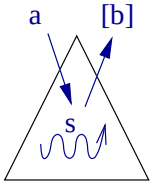

FIGURE 3.2 – GDL d'un éditeur élémentaire.

3.3.2 Problème de composition des éditeurs

Tel que décrit dans la section précédente, on voit un éditeur comme un processus réactif qui répond aux commandes d'édition pour modifier son état interne et produire éventuellement d'autres commandes d'édition, nécessaires à la composition avec d'autres éditeurs. Considérons deux éditeurs `F` et `G` comme indiqué sur la figure 3.3.

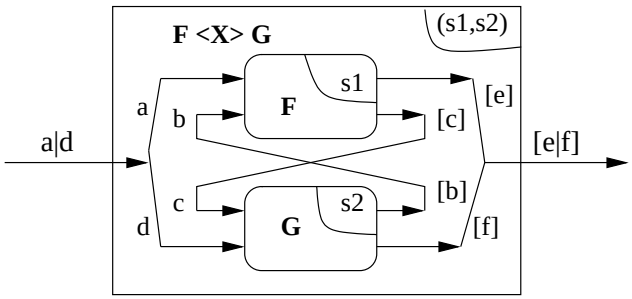

FIGURE 3.3 – Composition de deux éditeurs F et G

Nous souhaitons intérioriser (rendre transparentes à l'utilisateur) certaines commandes d'édition émises par l'un des éditeurs et reçues par l'autre. Nous regroupons alors l'ensemble des ports d'entrée et de sortie de chaque éditeur en conséquence. Ainsi, l'éditeur `F` a un état interne de type `s1`, ses actions d'édition en entrée sont soit de type `a` soit de type `b` (représenté en Haskell par `Either a b`), et ses commandes de sortie sont de type `e` ou `c` (`Either e c`). Similairement, l'état de l'éditeur `G` est de type `s2`, ses actions d'édition en entrée sont de type `c` ou `d` (`Either c d`) et ses commandes de sortie sont de type `b` ou `f` (`Either b f`). Comme le type de l'état des éditeurs est polymorphe, le plus simple est de considérer l'état résultat de la composition comme une paire `(s1,s2)` formée des états correspondant à chacun des sous-éditeurs. Ceci est aussi dû au fait que la composition est synchrone et nous ne souhaitons pas représenter des tampons de communication interne entre eux. Les ports d'entrée de l'éditeur résultat sont de type `a`, ou `d` et ses ports de sortie sont de type `e`, ou `f`. Ainsi les commandes d'édition de `F` vers `G` (de type `c`) ou de `G` vers `F` (de type `b`) ne sont pas accessibles de l'extérieur : ils constituent l'interface de composition des deux éditeurs. Un tel combinateur est utile si on veut connecter un éditeur structuré (réagissant aux actions d'un utilisateur externe) avec son presse-papiers vu aussi comme un éditeur. Le presse-papiers reçoit par exemple des commandes externes `Undo` et `Redo` et interagit avec l'éditeur principal (par des commandes internes transparentes à l'utilisateur) pour annuler ou refaire des actions déjà exécutées. En termes

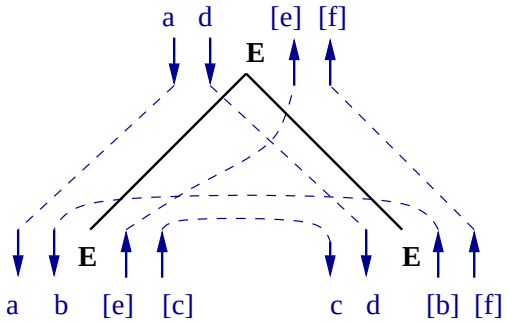

FIGURE 3.4 – GDL pour la composition de deux éditeurs

grammaticaux, le combinateur `<%>` est dérivé de la production `TimesEd : E -> E E` dont les règles sémantiques, représentées graphiquement par le graphe de dépendances locales associé à la production `TimesEd` (voir figure 3.4), implémentent l'assemblage des deux composants selon la sémantique décrite précédemment. Il est cependant difficile d'écrire

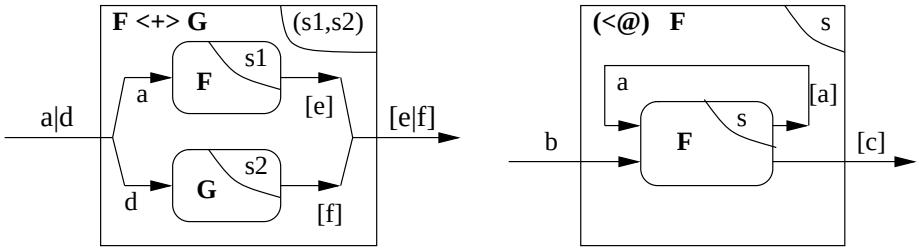

FIGURE 3.5 – **a)** Combinateur <+> **b)** Combinateur <@

directement ces règles sémantiques pour qu'elles puissent traduire la sémantique décrite pour cette composition sans passer par des combinateurs intermédiaires. Le constat est que le combinateur `<%>` repose essentiellement sur deux combinateurs supplémentaires implémentant respectivement l'union disjointe de deux éditeurs et la boucle, comme représenté graphiquement sur la figure 3.5. Les combinateurs `<+>` et `<@` sont dérivés des productions `PlusEd : E -> E E` et `LoopEd : E -> E` de la grammaire sous-jacente et les graphes de dépendances locales correspondants sont donnés par la figure 3.6.

3.3.3 Union disjointe de deux éditeurs : <+>

Concrètement, le type du combinateur `<+>` pour la réunion disjointe de deux éditeurs (voir figure 3.5.a) se dérive de la manière suivante. On considère les types des éditeurs `F` et `G` suivants, pour toute monade `m` :

```
F :: Editor s1 m a e  = a -> StateT s1 m [e]
G :: Editor s2 m d f  = d -> StateT s2 m [f]
```

et on souhaite obtenir le type suivant pour l'éditeur résultat de leur union :

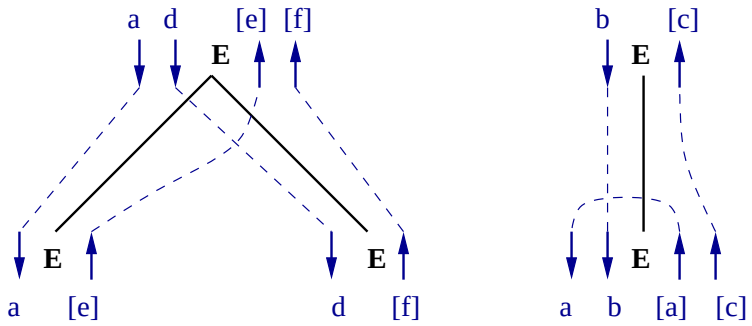

FIGURE 3.6 – GDL pour les productions respectives PlusEd et LoopEd

```
F <+> G :: Editor (s1,s2) m (Either a d) (Either e f) c'est-à-dire
F <+> G :: (Either a d) ->  StateT (s1,s2) m [Either e f]
```

Si on réunit les types des éditeurs `F` et `G` membre à membre, on obtient un objet de type :

```
(Either a d) -> StateT (s1,s2) m (Either [e] [f])
```

Ce qui ne correspond pas au type du résultat attendu. On a alors besoin d'une fonction de conversion (`fusion`) qui permet de fusionner les deux listes alternatives de valeurs de retour des éditeurs assemblés (`Either [e] [f]`) en une seule liste de valeurs alternatives (`[Either e f]`), résultat de l'assemblage. Elle est définie par :

```
-------------------------------------------------------------------------
fusion :: Monad m => StateT s m (Either [e] [f]) -> StateT s m [Either e f]
fusion p = mapMonad fct p  where fct (Left xs)  = map Left xs
                                 fct (Right xs) = map Right xs
-------------------------------------------------------------------------
```

Le combinateur <+> est finalement donné par le code suivant :

```
-------------------------------------------------------------------------
(<+>):: Monad m => Editor s1 m a e  -> Editor s2 m d f  ->
                                 Editor (s1,s2) m (Either a d)(Either e f)
f <+> g  =   h where h (Left a)  = fusion (lift_left (f a))
                     h (Right d) = fusion (lift_right (g d)))
-------------------------------------------------------------------------
```

Les fonctions `lift_left` et `lift_right` permettent respectivement de transformer (relever) un éditeur ayant un état et une seule valeur de retour en un éditeur ayant un couple d'états et deux valeurs alternatives de retour dans lesquels seules les composantes à gauche (respectivement à droite) de la valeur et de l'état sont concernées par le processus d'édition. Ceci permet d'éviter les erreurs de typage du langage hôte.

```
-------------------------------------------------------------------------
lift_left :: Monad m => StateT s m a -> StateT (s,s') m (Either a a')
```

```
lift_left (StateT f) = StateT g
               where g (s,s') = mapMonad (\(s1,x) -> ((s1,s'),Left x)) (f s)

lift_right :: Monad m => StateT s m a -> StateT (s',s) m (Either a' a)
lift_right (StateT g) = StateT h
              where h (s',s) = mapMonad (\(s1,x) -> ((s',s1),Right x)) (g s)
-------------------------------------------------------------------------
```

La fonction `mapMonad` est une variante de la fonction standard `map`. Elle permet d'appliquer une fonction aux valeurs de retour d'une monade de type `m`. Elle est définie par :

```
mapMonad :: (Monad m) => (a -> b) -> m a -> m b
mapMonad f p = do {x <- p ; return (f x)}
```

3.3.4 Combinateur de boucle : <@

L'implémentation du combinateur `<@` (voir figure 3.5.b) utilisé pour faire boucler (indépendamment de l'utilisateur) une commande d'un éditeur sur lui-même est donnée par le code suivant, pour toute monade `m` :

```
-------------------------------------------------------------------------
(<@) :: Monad m => Editor s m (Either a b) (Either a c) -> Editor s m b c
(<@) f = g  where g b = h (Right b)
        where h x =
          do {cmds_ac <- f x;
              cmdss_c <- mapM h [Left a | Left a <- cmds_ac];
              return ([c | Right c <- cmds_ac]++ (concat cmdss_c))}
-------------------------------------------------------------------------
```

Dans cette définition, la fonction locale `h` est implémentée de telle sorte que la commande d'édition de type `a` qui a été intériorisée, boucle seulement une fois. Ce choix permet au processus de terminer. Par ailleurs, la fonction `mapM` du prélude standard de type `mapM :: Monad m => (a -> m b) -> [a] -> m [b]` permet d'appliquer une fonction monadique sur une liste.

3.3.5 Intériorisation de commandes d'édition : <%>

L'assemblage de deux éditeurs comme décrit plus haut par la figure 3.3 (page 82) est obtenu par composition des combinateurs `<+>` et `<@`, en y ajoutant des fonctions adéquates (`phi`, `psi` et `bimap`) permettant de rediriger certaines sorties vers des entrées appropriées afin d'éviter des erreurs de typage pendant la composition, comme indiqué sur la figure 3.7.
Finalement, le code implémentant les règles sémantiques pour le combinateur `<%>` est donné par le listing ci-après :

```
-------------------------------------------------------------------------
(<%>) :: Monad m => Editor s1 m (Either a b) (Either e c) ->
```

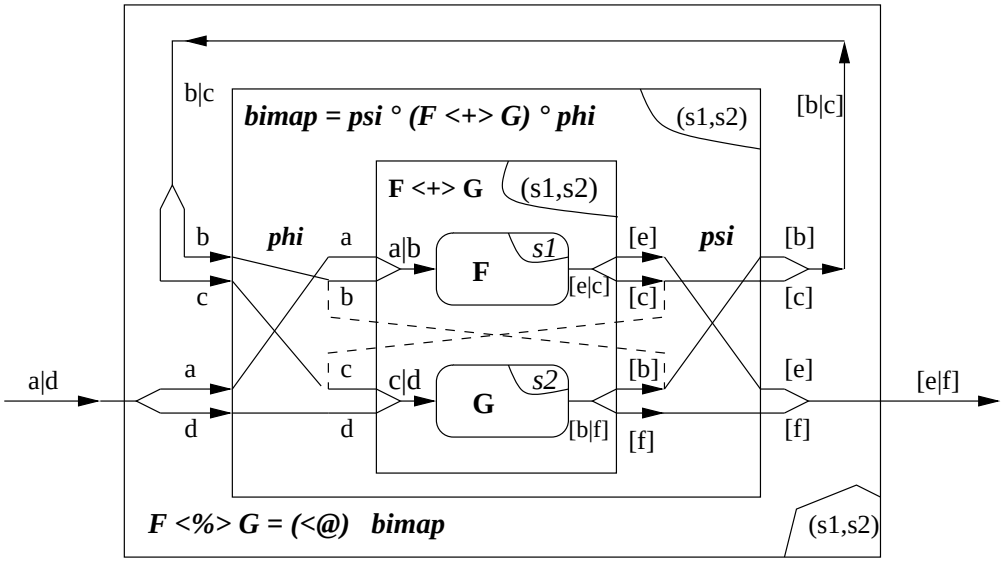

FIGURE 3.7 – Représentation graphique de la solution (<%>)

```
                  Editor s2 m (Either c d) (Either b f) ->
                  Editor (s1,s2) m (Either a d) (Either e f)
f <%> g = (<@) (bimap phi (f <+> g) psi)
-----------------------------------------------------------------------
```

dans lequel les fonctions `phi`, `psi` et `bimap` sont définies par

```
-----------------------------------------------------------------------
phi :: Either(Either b c)(Either a d) -> Either(Either a b)(Either c d)
phi (Right (Left a))  = Left (Left a)
phi (Left (Left b))   = Left (Right b)
phi (Left (Right c))  = Right (Left c)
phi (Right (Right d)) = Right (Right d)

psi :: Either(Either e c)(Either b f) -> Either(Either b c)(Either e f)
psi (Left (Left e))   = Right (Left e)
psi (Left (Right c))  = Left (Right c)
psi (Right (Left b))  = Left (Left b)
psi (Right (Right f)) = Right (Right f)

bimap :: Monad m => (in'->inp)-> Editor s m inp out -> (out -> out') ->
                                                    Editor s m in' out'
bimap phi e  psi = (mapMonad (map psi)) . e . phi
-----------------------------------------------------------------------
```

3.3.6 Composition "naturelle" d'éditeurs : <.>

Un autre combinateur utile est la composition "naturelle" de deux éditeurs notée <.> et représentée par la figure 3.8.a. La liste des commandes d'édition de type [b] correspondant au résultat de l'exécution de l'éditeur F est séquentiellement transmise à l'éditeur G par la fonction Haskell mapM. Le combinateur <.> se comporte alors comme l'opérateur bind de la classe Monad.

```
-------------------------------------------------------------------------
(<.>) :: Monad m => Editor s1 m a b  -> Editor s2 m b c ->
                                                    Editor (s1,s2) m a c
(f <.> g) a  = (StateT h)
        where h (s0,s0_) =
                  do {(s1,bs) <- runstate  (f a) s0;
                      (s2,css) <- runstate  (mapM g bs) s0_;
                      return ((s1,s2),concat css)}

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
-------------------------------------------------------------------------
```

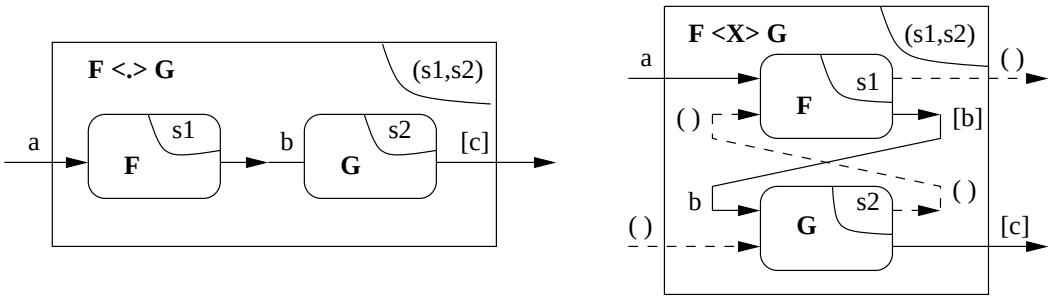

FIGURE 3.8 – **a)** Composition naturelle (<.>) **b)** Solution équivalente avec <%>

On remarque que le combinateur <.> est dérivé de <%> et la figure 3.8.b montre qu'il est obtenu en "enlevant" de <%> les commandes d'édition (en entrée et en sortie) appropriées et représentées sur la figure par des traits pointillés. Autrement dit, on remplace les types de données correspondantes par le type unitaire () pour signifier que les valeurs retournées sont sans intérêt.

3.4 Fonctionnement d'un éditeur de base

Les combinateurs que nous avons définis à la section précédente sont assez généraux et présentent un intérêt particulier. En effet le type de leurs états internes est polymorphe et peut de ce fait s'appliquer à des structures de données différentes. C'est un avantage significatif et non négligeable en pratique. Néanmoins, lorsqu'on s'intéresse à l'édition

proprement dite de documents structurés, il devient incontournable de restreindre ce polymorphisme à une structure de données adéquate, sans laquelle la définition du comportement élémentaire d'un éditeur serait impossible. Un tel comportement est décrit par les changements que l'état interne `s` d'un éditeur de type `Editor s m a b` subit pendant son exécution. Nous avons noté à l'introduction de ce chapitre que la représentation interne d'un document dans un éditeur (son état interne) doit pouvoir être localisable et partiellement définie. La structure adéquate connue comme la plus indiquée dans un environnement d'édition dans le cadre de la programmation fonctionnelle est le Zipper [3].

Nous donnons dans la section qui suit une présentation succincte de la structure de données Zipper, puis nous définissons quelques opérations d'édition admissibles sur celle-ci. Ceci constitue effectivement le fonctionnement interne d'un éditeur élémentaire. Tout cela nous permet par la suite d'enrichir notre bibliothèque par de nouveaux combinateurs dédiés à la manipulation des éditeurs dont l'état interne est une donnée structurée représentée par un Zipper attribué.

3.4.1 Structure de données Zipper de Huet

Dans le contexte de la programmation applicative et dans le but d'améliorer leur efficacité, la plupart des algorithmes cherchent à utiliser des opérations non destructives dans les structures de données. Pour le cas des arbres par exemple, certaines solutions comme le `functionnal arrays` de Paulson [74] permettent de modifier une occurrence dans un arbre de manière non destructive en copiant son chemin à partir de la racine de l'arbre, contrairement à la solution naïve qui copierait tout l'arbre. Cette technique s'avère cependant prohibitive lorsque la structure de données représente un contexte global, comme le tampon d'un éditeur de texte. Gérard Huet [3] a donc proposé une solution simple en offrant la structure de données Zipper qui permet de représenter un arbre et son contexte, i.e. un arbre avec un point focal sur un de ses nœuds, indiquant la position du curseur d'édition. La figure 3.9 est une illustration d'un tel arbre dans lequel le curseur d'édition se trouve sur le nœud `n10`. L'édition d'un arbre est de ce fait complètement locale ; et la manipulation des données ne se fait pas au niveau de la racine de l'arbre mais plutôt à la position courante dans l'arbre.

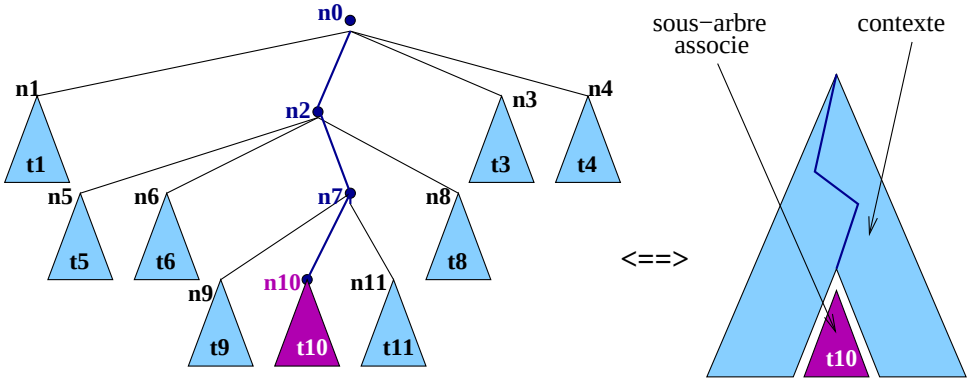

FIGURE 3.9 – Exemple d'un arbre localisé indiquant son contexte et le sous-arbre associé.

Pour construire le Zipper, l'idée consiste à retourner l'arbre de l'intérieur vers l'extérieur comme un gant retourné ; les pointeurs de la racine jusqu'à la position courante ayant été renversés dans une structure de données appelée `Path` représentant le contexte de l'arbre i.e. le chemin emprunté depuis la racine jusqu'au nœud courant. Le sous-arbre associé est l'arbre dont la racine est la position actuelle du curseur (point focal). A titre d'illustration, le premier graphique de la figure 3.10 représente la structure de données `Path` pour l'exemple précédent. Le Zipper est donc formé de la structure `Path` (le contexte) et du sous-arbre associé.

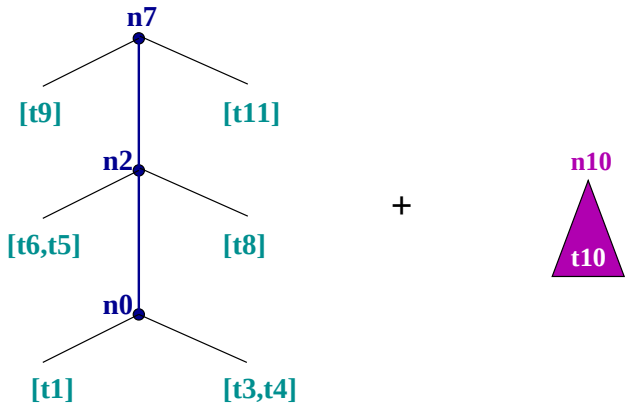

FIGURE 3.10 – Représentation d'un Zipper : contexte et sous-arbre associé.

En s'inpirant de la représentation d'origine d'un Zipper, plusieurs types Haskell ont été proposés pour le représenter, chaque proposition étant guidée par le problème à résoudre. Uustalu & Vene [75] ont ainsi traduit en Haskell la représentation d'origine sur un arbre d'arité 2, et proposé un algorithme d'évaluation d'attributs basé sur une structure de comonades, qui est la notion duale des monades en catégorie. Nous avons étendu cet algorithme sur un arbre d'arité quelconque (*rose trees* en anglais) comme décrit dans l'annexe A. Nous avons aussi dans la même veine proposé une représentation arborescente des Zippers sous forme d'une structure de donnée cyclique. Ce qui nous a permis d'introduire une nouvelle technique pour l'évaluation des attributs d'une grammaire attribuée potentiellement circulaire et de manipuler les structures de données potentiellement infinies [27, 38]. Notre algorithme consiste à évaluer un attribut par réduction à une forme normale en utilisant un automate d'arbres opérant sur cette représentation.

Toutefois, la représentation la plus adaptée au problème posé dans cette partie est celle proposée par B. Sufrin & Oege De Moor dans *Modeless structure editor* [23]. Elle est basée sur la notion de couches regroupant à tout moment dans une liste les différentes strates du contexte d'un Zipper. Les définitions correspondantes en Haskell sont les suivantes :

```
data Tree a      = MkT {label :: a, children :: [Tree a]}
type Path  a     = [Layer a (Tree a)]
type Layer a x   = (a,[x],[x])
data Zipper a    = (:>) { context :: Path a, subtree :: Tree a}
```

dans lesquelles, un arbre de type `Tree` consiste en un nœud étiqueté de type `a` et une liste éventuellement vide de ses sous-arbres (ses descendants). Le type polymorphe `a` décrit

les différentes constructions de la structure éditée. Ce sera par exemple, des attributs hérités ou synthétisés permettant de répondre aux questions sémantiques. Le Zipper est représenté par le type `Zipper a`, formé du sous-arbre associé (de type `Tree a`), et de la liste des directions partant de la racine jusqu'au sous-arbre sélectionné (le chemin de type `Path a`). Un Zipper `(p :> t) :: Zipper a` représente alors la sélection de l'arbre `t` à la position indiquée par le chemin `p`. Un chemin de type `Path a` est une liste de couches (strates) du contexte. Il représente le contexte exactement dans l'ordre rencontré lorsqu'on remonte du sous-arbre sélectionné vers la racine. Une donnée de type `Layer a (Tree a)` représente comme son nom l'indique une couche du contexte.

Avec cette définition le terme `t0` correspondant à l'arbre de départ (d'arité quelconque) de la figure 3.9 est donné par le listing suivant :

```
-------------------------------------------------------------------------
t0  = MkT "n0" [t1, t2, t3, t4]
t1  = MkT "n1" []
t2  = MkT "n2" [t5, t6, t7, t8]
t3  = MkT "n3" []
t4  = MkT "n4" []
t5  = MkT "n5" []
t6  = MkT "n6" []
t7  = MkT "n7" [t9, t10, t11]
t8  = MkT "n8" []
t9  = MkT "n9" []
t10 = MkT "n10" []
t11 = MkT "n11" []
-------------------------------------------------------------------------
```

dans lequel les nœuds sont simplement étiquetés par une chaîne de caractères indiquant simplement leurs noms comme représentés sur ladite figure. Le Zipper `z10` correspondant à la situation où le curseur d'édition se trouve au nœud `n10` ayant suivi le chemin indiqué sur la figure 3.10 est donné par

```
z10 = [("n7",[t9],[t11]), ("n2",[t6,t5],[t8]), ("n0",[t1],[t3,t4])]
     :> MkT "n10" []
```

Pour arriver à cette position, des opérations de navigation dans la structure de Zipper sont nécessaires en partant de la racine dont le terme correspondant est `z0 = [] :> t0`. Nous implémentons ces opérations dans la section qui suit.

3.4.2 Opérations classiques d'édition sur un Zipper

Comme déjà mentionné dans ce document, le but principal de la structure de donnée Zipper est de faciliter la navigation dans le document, et d'appliquer les opérations d'édition directement à la position courante du curseur d'édition. La représentation du Zipper telle que décrit précédemment offre ces avantages. Les opérations de navigation permettant de se déplacer dans le Zipper à partir de la position courante sont directement

décrites en utilisant les modèles de couches. Nous introduirons plus loin d'autres opérations d'édition (insertion et suppression d'une production) qui dépendent de la grammaire attribuée sous-jacente. Les opérations `left` et `right` permettent de se déplacer entre les frères d'un nœud donné et les opérations `up` et `down` permettent de se déplacer entre les fils et les parents lorsque cela est possible comme implémentées dans le tableau 3.1. Ainsi, `left` permet de se déplacer à gauche quand c'est possible et reste sur place lorsqu'on atteint la position la plus à gauche i.e. lorsque le chemin est vide, ou lorqu'il n'y a aucun nœud à gauche du sous-arbre sélectionné dans son parent. Un raisonnement similaire est valable pour l'opération `right`.

Tableau 3.1 Opérations classiques d'édition sur la structure de Zipper

```
left, right, up, down :: Zipper a  -> Zipper a

left zp@([]:>_)                =  zp
left zp@(((_,[],_):c):>_)      =  zp
left (((a, u:left, right):c) :> t) = ((a, left, t:right):c) :> u

right zp@([]:>_)                  =  zp
right zp@(((_,_,[]):c):>_)        =  zp
right (((a, left, u:right):c) :> t) = ((a, t:left, right):c) :> u

down zp@(c:>(MkT _ [])) = zp
down (c:>(MkT a (t:ts))) = ((a,[],ts):c):>t

up  zp@([]:>_)           =  zp
up (((a,left,right):c):>t) =  c:>(MkT a (flat left (t:right)))

flat :: [a] -> [a] -> [a]
flat [] right = right
flat (t:left) right = flat left (t:right)

replace :: Tree a -> Zipper a -> Zipper a
replace s (p :> t) = p :> s
```

De la même manière, l'opération `down` permet de descendre au premier fils du nœud sélectionné. Bien évidemment, on n'autorise la descente que lorsqu'il y a au moins un fils ; et dans le cas contraire, on reste sur place. L'opération `up` quant à elle permet de remonter dans l'arbre à partir du nœud courant (i.e. vers son père s'il existe) en concaténant ses frères gauches (pris dans l'ordre inverse) avec lui-même et ses frères droits. Cette concaténation dans le bon ordre est réalisée par la fonction `flat`. L'inversion des sous-arbres à gauche de la position courante est due au fait qu'ils sont représentés dans chaque couche du contexte à partir du frère le plus proche de la position courante. On définit aussi une fonction `replace` qui remplace le sous-arbre associé à un Zipper par l'arbre passé en paramètre.

Comme illustration, considérons l'arbre localisé de la figure 3.9. Pour atteindre le nœud `n10` à partir du Zipper initial `(z0 = [] :> t0)`, on doit appliquer successivement sept opérations de navigation comme l'indique le code suivant :

```
-----------------------------------------------------------------------
z0 = [] :> t0
z10 = right (down (right (right (down (right (down z0))))))
      ou bien
z10 = (right . down . right . right . down . right . down ) z0
-----------------------------------------------------------------------
```

D'autres opérations classiques d'édition ont été définies dans le papier originel [23] et permettent par exemple d'aller au prochain nœud éditable lorsqu'on se trouve à une feuille (`next :: Zipper a -> Zipper a`) ; ou de parcourir le Zipper dans l'ordre préfixé et de s'arrêter au prochain nœud satisfaisant un prédicat passé en paramètre (`nextSuchThat :: (Zipper a -> Bool) -> Zipper a -> Zipper a`), etc. Il n'est pas nécessaire de nous attarder sur ces autres opérations, puisque nous avons garder leurs définitions d'origine. Par contre, nous présentons dans la section suivante des opérations spécifiques, directement liées à la grammaire attribuée sous-jacente.

3.4.3 Représentation d'une grammaire attribuée par un Zipper

Nous avons déjà mentionné qu'une donnée structurée peut être représentée intentionnellement sous la forme d'une structure arborescente caractérisée par une grammaire algébrique abstraite. Cette grammaire peut être enrichie par des attributs et des règles sémantiques appropriées pour répondre aux questions sémantiques. Une grammaire attribuée est alors décrite par un ensemble de productions, qui associe à chaque production un ensemble de règles sémantiques (`rules`) et une règle indiquant, en partie gauche (`lhs`) le nœud sur lequel la production est appliquée et, en partie droite (`rhs`) la liste éventuellement vide des nœuds dans lesquels le symbole en partie gauche se réécrit. Dans la mesure où nous aurons à manipuler des arbres clos et partiellement developpés (arbres avec bourgeons), on distinguera trois types de nœuds :

- Des nœuds ouverts (`Hole`) associés aux symboles grammaticaux non développés.
- Des nœuds internes (`Internal p`) associés à des productions i.e. à des symboles déjà développés.
- Des nœuds fermés (`Leaf`) associés aux symboles grammaticaux non réductibles (qu'on ne peut pas développer). Ce sont les feuilles de l'arbre.

En plus de son type (`Hole`, `Internal p` ou `Leaf`), un nœud (`NodeType p`) indique le symbole grammatical `s` et l'ensemble des attributs `[Attribute]` qui lui sont associés. Un attribut de type `Attribute` est défini par son nom (une chaîne de caractères), son type (hérité ou synthétisé), sa signature (ou domaine de ses valeurs) décrite aussi par une chaîne de caractères, et une chaîne représentant sa valeur par défaut ou lorsqu'elle est modifiée.

Le Zipper approprié permettant de représenter de telles grammaires attribuées est donné par le type `ZipperA`, étiqueté par un label de type `Node s p`. Le code Haskell équivalent est donné par le listing du tableau 3.2.

Tableau 3.2 Représentation d'une grammaire attribuée par un Zipper

```
type Grammar symb prod rules =
             prod -> (Node symb prod, [Node symb prod], [rules])

data Node s p = MkN {symbol     :: s,
                     nodeType   :: NodeType p,
                     attributes :: [Attribute]}

data NodeType p = Hole | Internal p | Leaf

data Attribute = Att {attName :: String,  -- nom de l'attribut
                      attType :: AttType, -- type de l'attribut
                      attSign :: String,  -- domaine de valeurs
                      attValue:: String}  -- valeur de l'attribut
data AttType = Inh | Syn

type ZipperA s p = Zipper (Node s p)
```

Les fonctions de sélection des différentes composantes associées à cette signature sont définies par

```
-------------------------------------------------------------------------
lhs :: Grammar s p r -> p -> Node s p
lhs gram p = a where (a,_, _) = gram p

rhs :: Grammar s p r -> p -> [Node s p]
rhs gram p = as where (_,as,_) = gram p

rules :: Grammar s p r -> p -> [r]
rules gram p = rs where (_,_,rs) = gram p

-- Symbole en partie gauche d'une production
lhsSymb :: Grammar s p r -> p -> s
lhsSymb gram p = symbol (lhs gram p)

-- liste des symboles en partie droite d'une production
rhsSymb :: Grammar s p r -> p -> [s]
rhsSymb gram p = map symbol (rhs gram p)
-------------------------------------------------------------------------
```

Cette représentation nous permet de définir les opérations d'édition ci-après. Nous définirons chaque fois que cela s'avèrera nécessaire les opérations duales permettant de les annuler.

3.4.4 Suppression d'un sous-arbre (*kill*) ou d'une couche (*delete*)

L'opération `kill` (ou `drop`) permet de remplacer un sous-arbre par un nœud ouvert. Elle correspond à la destruction d'une branche de l'arbre ; comme par exemple le reste de la chaîne (suite de caractères) dans un éditeur de texte. Elle est utile lorsqu'on souhaite par exemple "couper" une partie d'un document. L'opération `delete` implémente la suppression de la couche directement au dessus du curseur d'édition. Elle correspond par exemple dans un éditeur de texte à la suppression du caractère situé avant le curseur. Cette opération n'est autorisée que si le symbole associé au sous-arbre courant est le même que le symbole associé au nœud père (indiqué dans la couche à supprimer). On les définit par :

```
-------------------------------------------------------------------------
kill :: Grammar s p r -> ZipperA s p -> ZipperA s p
kill gram zp@(pat:>t) = case nodeType (label (subtree zp)) of
               Internal p -> replace (MkT lab []) zp
                 where lab = MkN (symbol nd) Hole (attributes nd)
                       nd  = lhs gram p
               _ -> zp
drop = kill

undo_kill :: Tree (Node s p) -> ZipperA s p -> ZipperA s p
undo_kill t st  = replace t st

delete :: Eq s => Grammar s p r -> ZipperA s p -> ZipperA s p
delete gram zp@((((MkN s (Internal p) _),_,_) : ls) :> t) =
                 if (symbol (label t)) == s then ls :> t else zp
delete gram zp@(_ :> _) = zp

undo_delete :: Layer (Node s p) (Tree (Node s p)) -> ZipperA s p
                                                  -> ZipperA s p
undo_delete l (ls :> t) = ((l:ls) :> t)
-------------------------------------------------------------------------
```

Lorsqu'on supprime une partie du document par les fonctions `kill` ou `delete`, on doit garder la trace de la partie supprimée dans un tampon d'édition pour pouvoir reconstituer le document initial si on annule la suppression par une action `Undo`. C'est ce que réalisent les opérations `undo_kill` et `undo_delete`.

3.4.5 Insertion d'une production à la position courante : *insert*

Une action d'édition élémentaire consiste à remplacer un nœud ouvert associé à un symbole grammatical par un modèle d'arbre associé à une production s'appliquant à ce symbole. Un modèle d'arbre (*template*) est donc un arbre de profondeur 1 dont la racine est un nœud interne associé à la production considérée et ayant autant de successeurs (nœuds ouverts) que de symboles grammaticaux en partie droite de ladite production. Il

est implémenté par la fonction `template` dans laquelle `makeHole` permet de fabriquer un nœud ouvert associé à un symbole grammatical.

```
-------------------------------------------------------------------------
template :: Grammar s p r -> p -> Tree (Node s p)
template gram p =
     MkT (MkN (symbol a) (Internal p) (attributes a)) (map makeHole as)
                          where (a,as, _) = gram p
makeHole :: Node s p -> Tree (Node s p)
makeHole s = MkT (MkN (symbol s) Hole (attributes s)) []

-- permet de savoir si le curseur se trouve sur un noeud ouvert
atHole :: ZipperA s p -> Bool
atHole (_ :> (MkT (MkN _ Hole _) _)) = True
atHole  _ = False

-- permet de savoir si le curseur se trouve sur un noeud développé
atInternal :: ZipperA s p -> Bool
atInternal (_:> (MkT (MkN _ (Internal _) _)_)) = True
atInternal  _ = False

-- Variante de replace avec déplacement vers le prochain trou
replace_ :: Tree (Node s p) -> ZipperA s p -> ZipperA s p
replace_ t = (nextSuchThat atHole) . (replace t)

insert ::(Eq s)=> Grammar s p r -> p -> ZipperA s p ->ZipperA s p
insert gram p zp@(_:> (MkT (MkN a Hole _) _)) =
                            if a == (symbol (lhs gram p)) then
                                (replace (template gram p) zp)
                            else zp
insert gram p zp@(_:>(MkT (MkN a (Internal q) _)_)) =
               if (symbol (lhs gram q)) == (symbol (lhs gram p))
                    then (replace (template gram p) zp) else zp
insert gram p zp@(_:>_) = zp

undo_insert :: Tree (Node s p) -> ZipperA s p -> ZipperA s p
undo_insert t zp  = replace t zp
-------------------------------------------------------------------------
```

L'opération `insert` permet d'insérer une production (c'est en réalité le modèle d'arbre associé à ladite production) à la position courante du curseur d'édition. L'insertion n'est admissible que si les deux productions ont le même symbole en partie gauche. Pour annuler une telle opération, la solution la moins coûteuse est de garder dans le tampon d'édition le sous-arbre coupé. Par ailleurs, après insertion, on aurait souhaité que le curseur se positionne soit sur le premier nœud ouvert (s'il y en a) du template, sinon sur le prochain premier nœud ouvert dans le Zipper. Il faudrait pour cela utiliser la variante `replace_`

pour le remplacement de sous-arbres. Seulement, l'annulation ne devrait être possible dans ce cas que si la partie droite du template inséré contenait effectivement avant son insertion des trous. Ce qui aurait permis à coup sûr de remonter d'abord à la bonne position (par une opération `up`) avant d'annuler l'insertion. Et dans le cas où le template inséré ne possédait aucun trou en sa partie droite, revenir à la position initiale s'est avéré être un casse-tête difficile à résoudre et impossible à automatiser. Le plus simple a donc été d'utiliser dans le code de la fonction `insert` la fonction `replace` qui permet de rester sur place après insertion du template ; et d'annuler simplement l'opération par la fonction `undo_insert`. Cette solution présente l'inconvénient selon lequel il faut explicitement se deplacer vers le prochain trou après insertion du template.

3.4.6 Insertion d'une production dans un Zipper : *insertinto*

Le mode d'insertion décrit précédemment et s'appliquant à des nœuds ouverts ou internes implique une perte d'informations. En effet, l'opération `insert` fait un remplacement de sous-arbres avec suppression du reste de l'arbre (i.e. de tous ses descendants). On aimerait bien insérer une production à l'intérieur d'un Zipper sans détruire le sous-arbre associé. Il s'agit par exemple de pouvoir insérer un caractère à la position du curseur dans un éditeur de texte sans perdre le reste du texte comme le permettrait l'opération `insert` ; ou bien de remplacer dans la grammaire des boîtes une disposition verticale de deux boîtes par une disposition horizontale de ces mêmes boîtes sans détruire lesdites composantes. C'est ce que décrit l'opération `insertinto` ci-dessous. Elle n'est admissible que si les parties droites des deux productions concernées sont identiques.

```
-------------------------------------------------------------------------
insertinto :: Eq s => Grammar s p r -> p -> ZipperA s p -> ZipperA s p
insertinto gram p zp@(c :> (MkT (MkN _ (Internal q) _) as)) = if ps==qs
      then (c:>(MkT (MkN (symbol nd) (Internal p) (attributes nd)) as))
         else zp
             where ps = map symbol (rhs gram p)
                   qs = map symbol (rhs gram q)
                   nd = lhs gram p
insertinto gram p zp@(_ :> _) = zp

undo_insertinto :: Grammar s p r -> p -> ZipperA s p -> ZipperA s p
undo_insertinto gram q zp@(c :> (MkT _ as)) =
         (c:> (MkT (MkN (symbol nd) (Internal q) (attributes nd)) as))
             where nd = lhs gram q
-------------------------------------------------------------------------
```

Pour annuler cette opération, il est nécessaire de garder dans le tampon d'édition la production remplacée. L'annulation consiste alors à la remettre à sa place initiale comme défini par la fonction `undo_insertinto`.

3.4.7 Modèles de couches et insertion d'une production à une position donnée : *inserto_i*

Un template (modèle d'arbre) permet d'insérer une production par remplacement de sous-arbres. Il agit sur le sous-arbre sélectionné à partir de la position du curseur. Celui-ci étant placé sur le symbole grammatical à gauche de la production concernée. Il nous a permis d'implémenter l'opération `insert`. Un *template layer* (modèle de couche) va permettre d'insérer une production par ajout d'une couche au contexte. Il agit sur le contexte à partir de la position du curseur placé sur un des symboles grammaticaux en partie droite de la production concernée. Un modèle de couche est donc une couche dont :

- le premier élément représente le nœud interne parent (correspondant au symbole grammatical en partie gauche de la production associée) ;
- la première liste est l'ensemble des nœuds ouverts correspondant aux frères gauches du symbole courant (celui sur lequel est placé le curseur d'édition) ; et
- la deuxième liste de nœuds ouverts représente ses frères droits.

On constate avec cette définition qu'il y a autant de modèles de couches associés à une production que de symboles en partie droite de cette production.

```
-------------------------------------------------------------------------
template_layer_gen ::Eq s => Grammar s p r -> p ->Path (Node s p)
template_layer_gen gram p = case (rhs gram p) of
  [] -> []      -- production sans symbole en partie droite
  as -> map f (decompose as)
    where decompose xs = [(ys,x,zs) | (ys,x:zs) <-
              [(take n xs, drop n xs) | n <- [0..(length xs)-1]]]
          f (ys,_,zs)=(MkN (symbol nd) (Internal p) (attributes nd)
                      ,map makeHole ys, map makeHole zs)
          nd = lhs gram p

template_layer :: Eq s => Grammar s p r  -> p -> Path (Node s p)
template_layer gram p = [pt | pt@(MkN x _ _,_,_) <-
                              template_layer_gen gram p, a == x]
          where a = symbol (lhs gram p)
-------------------------------------------------------------------------
```

La fonction `template_layer_gen` effectue un calcul automatique des modèles de couches pour toute production. La version la plus pratique est celle donnée par `template_layer` qui ne retient que les modèles dont les symboles en partie droite sont identiques au symbole en partie gauche de la production concernée. Ceci permet l'insertion d'une production à l'intérieur du Zipper sans détruire le sous arbre associé.

Pour effectuer une insertion d'un modèle de couche, on doit pouvoir l'identifier dans la liste des modèles de couches associés à la production considérée. La solution la plus simple est de passer en paramètre de la fonction `insert_i` un entier, indiquant l'index du modèle qu'on doit insérer. L'opération d'insertion est donnée par le listing suivant :

```
-------------------------------------------------------------------------
insert_i ::  (Eq s) => Grammar s p r -> p -> Int -> ZipperA s p
```

```
                                                    -> ZipperA s p
insert_i gram p i zp@(cxt :> t) = case template_layer_ gram p of
                [] -> zp
                tls -> case elem_i i (zip tls [1..length tls]) of
                                Nothing -> zp
                                Just l -> (l:cxt) :> t

elem_i :: Int -> [(a,Int)] -> Maybe a
elem_i i [] = Nothing
elem_i i ((x,j):xs) = if i == j then Just x else elem_i i xs

undo_insert_i :: ZipperA s p -> ZipperA s p
undo_insert_i ((l:ls) :> t) = (ls :> t)
-------------------------------------------------------------------------
```

Comme l'insertion se fait directement à partir de la position du curseur i.e. comme premier élément du contexte, l'annulation de cette opération, décrite par la fonction `undo_insert_i`, consiste simplement à retirer le premier élément de la liste des couches formant le contexte.

Ces opérations d'édition traduisent le comportement d'un éditeur de base. Ainsi définies, elles resteront transparentes au programmeur qui se contentera d'utiliser simplement le langage des combinateurs d'éditeurs pour composer ses éditeurs afin d'obtenir des éditeurs plus complexes, manipulant des structures de données générales, ou des données représentées sous forme de Zippers.

3.5 Combinateurs d'éditeurs liés à la structure de Zipper

Le résultat d'analyse d'une chaîne, obtenu en utilisant des combinateurs d'analyse, est en général représenté par une structure de données définie par l'utilisateur. L'objectif est d'offrir comme abordé dans EdComb de O. Braun & al. [68, 69], la possibilité de combiner des éditeurs en suivant la structure du document à éditer, représentée sous forme arborescente par un Zipper attribué. Il s'agit principalement de construire deux combinateurs `<*>` et `<|>` pour la composition séquentielle et alternative de deux éditeurs, de telle sorte que l'éditeur permettant de manipuler toute donnée structurée s'obtient avec ces combinateurs en suivant la syntaxe BNF de la production qui la décrit. Par exemple, si on a deux productions `P1 : A -> B C` et `P2 : A -> D`, ou de manière plus compacte `P : A -> B C | D`, alors l'éditeur structuré `editorA` pour le type `A` sera obtenu en composant les éditeurs `editorB`, `editorC` et `editorD` pour les types respectifs `B`, `C` et `D` de la manière suivante :

```
editorA = (editorB <*> editorC) <|> editorD
```

Nous nous proposons dans cette partie de définir ces combinateurs, et de généraliser le combinateur `<*>` par un combinateur `<*` qui implémente l'étoile de Kleene pour une production d'arité quelconque. Nous avons mentionné depuis l'introduction de ce document

qu'on ne considérait que les grammaires abstraites i.e. des grammaires algébriques ou attribuées sans symboles terminaux. Nous souhaitons revenir sur cette condition et considérons maintenant les grammaires concrètes. Ce qui nous permet de proposer d'abord des éditeurs pour les types Haskell de base.

3.5.1 Éditeurs pour les types Haskell de base

Il s'agit des éditeurs les plus primitifs pour les valeurs constantes appartenant aux types Haskell de base. Nous n'en donnons que quelques uns (comme l'éditeur pour un entier, pour un caractère ou pour un booléen) et en déduisons un éditeur pour toute constante d'un type de base quelconque.

Un éditeur pour un entier est défini par la primitive `intEditor` qui prend en paramètre l'entier qu'on édite et le symbole grammatical auquel il est associé et retourne en sortie un éditeur dont l'état interne est un Zipper contenant cet entier comme valeur (attribut synthétisé). Le nœud associé est un nœud fermé i.e. une feuille de type `Leaf`. Un raisonnement similaire permet de construire un éditeur `charEditor` pour un caractère et de façon générale un éditeur `constEditor` pour une constante. Il est nécessaire pour ce dernier éditeur que la constante `c` qu'on édite soit un type qu'on peut afficher d'où la présence de la contrainte `Show c`.

```
-------------------------------------------------------------------------
intEditor :: Monad m => Int -> s -> Editor (ZipperA s p) m a b
intEditor n s _ =  (StateT f)
                where f _ = return (([] :> MkT (MkN s Leaf atts) []),[])
                           where atts = [Att "value" Syn "Int" (show n)]

charEditor :: Monad m => Char -> s -> Editor (ZipperA s p) m a b
charEditor c s _ = (StateT f)
                where f _ = return (([] :> MkT (MkN s Leaf atts) []),[])
                          where atts = [Att "value" Syn "Char" (show c)]

constEditor :: (Monad m, Show c) => c -> s -> Editor (ZipperA s p) m a b
constEditor c s _ = (StateT f)
                where f _ = return (([] :> MkT (MkN s Leaf atts) []),[])
                           where atts = [Att "value" Syn "c" (show c)]
-------------------------------------------------------------------------
```

3.5.2 Composition séquentielle d'éditeurs : <*> et <*

Soient `n` éditeurs `e1`, `e2`, ..., `en` de type `Editor (ZipperA s p) m a b` chacun, associés aux symboles $X_1, X_2, \ldots, X_n$ de la production $P : X_0 \rightarrow X_1\, X_2, \ldots, X_n$ d'une grammaire attribuée. La question que l'on se pose ici est de savoir comment composer ces éditeurs pour obtenir un éditeur pour le symbole X_0 ?

Supposons sans nuire à la généralité que `n = 2`, il s'agit alors de la composition séquentielle de deux éditeurs `e1` et `e2` notée `<*>`. Le comportement de l'éditeur résultat est décrit par les règles sémantiques de la grammaire attribuée sous-jacente comme indiqué

sur la figure 3.11. Intuitivement, les commandes d'édition en entrée (héritées de l'éditeur

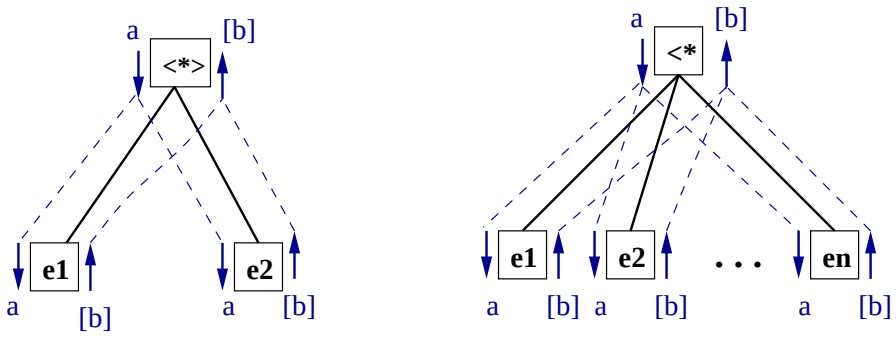

FIGURE 3.11 – Composition séquentielle de deux éditeurs.

résultat) sont transmises aux différentes composantes et modifient potentiellement leurs états internes. Les commandes en sortie (synthétisées des composantes) sont rassemblées (par concaténation) pour former les sorties de l'éditeur résultat. L'état interne de ce dernier est construit à partir des états modifiés des différentes composantes : il conserve son contexte et son étiquette, mais a pour descendants les sous-arbres associés aux états de ses composantes. La traduction en Haskell est donné par le listing suivant :

```
-------------------------------------------------------------------------
(<*>) :: Monad m => Editor (ZipperA s p) m a b  ->
                 Editor (ZipperA s p) m a b -> Editor (ZipperA s p) m a b
(e1 <*> e2) a = (StateT h) where h zp@(ct:> (MkT nd _)) =
                             do {(c1:>t1, bs1) <- runstate (e1 a) zp;
                            (c2:>t2, bs2) <- runstate (e2 a) zp;
                            return ((ct:>MkT nd [t1,t2]),bs1++bs2)}
-------------------------------------------------------------------------
```

Pour généraliser, on procède par un raisonnement analogue pour implémenter les règles sémantiques associées à la composition séquentielle de `n` éditeurs notée `<*` (étoile de Kleene) et représentée par le deuxième graphe de dépendances locales de la figure 3.11. Il est nécessaire que $n \geq 2$ pour que la composition soit possible. Le code Haskell correspondant est donné par

```
-------------------------------------------------------------------------
(<*) :: (Monad m, Comonad m) => [Editor (ZipperA s p) m a b] ->
                                                Editor (ZipperA s p) m a b
(<*) es a = (StateT h) where h zp@(ct:> (MkT nd _)) =
                 return ((ct:> (MkT nd [ti | (_:>ti,_) <- sts])),
                         concat [bsi | (_,bsi) <- sts])
         where sts = map counit [(runstate (e a) zp) | e <- es]

class Comonad m where
   counit :: m a -> a
-------------------------------------------------------------------------
```

On remarque aussi dans la définition de `<*` qu'il est nécessaire que la monade associée `m` soit en plus une comonade restreinte à la méthode unité `counit`. Cette méthode permet de se débarrasser du contexte (des effets) de la valeur calculée. Ceci est un handicap dans l'utilisation concrète de ce combinateur, puisqu'il sera difficile en pratique de trouver une structure de données qui instancierait à la fois une monade et une comonade (fusse-t-elle restreinte à la méthode unité). Cette remarque est aussi valable dans notre cas, étant donné que nous envisageons, pour permettre l'interaction de l'éditeur avec le monde extérieur (utilisateur final), d'instancier `m` par une interface utilisateur graphique ou textuelle (IO) qui se rapproche essentiellement des calculs avec effets (monade). Dans ces conditions, le programmeur devra alors pour chaque production donner une variante du combinateur `<*` comportant le nombre exact des éditeurs à passer en paramètres. Le code Haskell correspondant qui devra être complété est de la forme :

```
-------------------------------------------------------------------------
(<*) :: Monad m => Editor (ZipperA s p) m a b  -> ... ->
                                                 Editor (ZipperA s p) m a b
(<*) e1 e2 ... en a = (StateT h) where h zp@(ct:> (MkT nd _)) =
                                  do {(c1:>t1, bs1) <- runstate (e1 a) zp;
                                      (c2:>t2, bs2) <- runstate (e2 a) zp;
                                        ...
                                      (cn:>tn, bsn) <- runstate (en a) zp;
                                      return ((ct:>MkT nd [t1,t2, ...,tn])
                                               ,bs1++bs2++ ... ++bsn)}
-------------------------------------------------------------------------
```

3.5.3 Composition alternative d'éditeurs : $<|>$

Pour la composition alternative, on constate qu'elle peut se réduire à la composition de deux éditeurs puisque l'éditeur résultat se comporte intuitivement comme l'un ou (exclusif) l'autre de ses composantes. Ainsi si l'on souhaite composer alternativement `n` éditeurs `e1`, `e2`, ..., `en`, on utilisera le même combinateur `<|>` de la manière suivante : `(((e1 <|> e2) <|> e3) <|> ....) <|> en`. Il est nécessaire de passer en paramètre de ce combinateur un drapeau qui permet de filtrer à chaque envoi de commandes d'entrée, la composante concernée par la composition. Ce drapeau est un booléen qui vaut `True` lorsque c'est le premier éditeur qui est concerné par l'édition et `False` s'il s'agit du deuxième éditeur. Le code Haskell équivalent est donné par le listing suivant :

```
-------------------------------------------------------------------------
(<|>) :: Monad m => Editor (ZipperA s p) m a b  ->
         Editor (ZipperA s p) m a b -> Bool -> Editor (ZipperA s p) m a b
(e1 <|> e2) flag  = if flag then e1 else e2
-------------------------------------------------------------------------
```

3.6 Quelques applications de ces combinateurs d'éditeurs

3.6.1 Édition structurée avec presse-papiers

Considérons à titre illustratif les productions suivantes d'une grammaire algébrique concrète :

```
-------------------------------------------------------------------------
  P1 : A --> B C  | D
  P2 : B --> 1
  P3 : C --> 'a'
  P4 : D --> 'e'
-------------------------------------------------------------------------
```

Alors, l'éditeur `editorA` pour la catégorie syntaxique `A` (axiome de la grammaire) qui permet d'éditer un document conforme à cette grammaire est obtenu par composition d'éditeurs de base `editorB`, `editorC` et `editorD` pour les symboles respectifs `B`, `C` et `D`. Il est défini de la manière suivante :

```
-------------------------------------------------------------------------
editorA flag =  ((editorB <*> editorC) <|> editorD) flag

editorB :: Monad m => Editor (ZipperA String p) m a b
editorB = intEditor 1 "B"

editorC, editorD :: Monad m => Editor (ZipperA String p) m a b
editorC = charEditor 'a' "C"
editorD = charEditor 'e' "D"
-------------------------------------------------------------------------
```

De manière générale, un éditeur de ce type réagit aux actions d'édition d'un utilisateur pour modifier son état interne (un Zipper de type `ZipperA s p`). Pour pouvoir agir de manière uniforme sur la structure de Zipper, les actions d'édition sont définies dans un format propriétaire (par exemple de type `Cmd`) et chaque action utilisateur est transformée en action d'édition correspondante dans ce format. Pour que chaque action d'édition puisse être annulée ou refaite, tout éditeur de donnée structurée doit transmettre à son presse-papiers des informations concernant chaque action d'édition. Nous considérons donc le presse-papiers comme un éditeur qui reçoit ces informations et met à jour son propre état ; représenté par un ensemble de deux piles (ou listes) d'actions effectuées (que l'on peut annuler avec la commande `Undo`) et d'actions annulées (que l'on peut refaire avec la commande `Redo`). Les actions utilisateur de type `Undo` et `Redo` sont directement reçues par l'éditeur de presse-papiers pour respectivement annuler et refaire les actions précédentes.

Concrètement comme indiqué sur la figure 3.12, lorsqu'un utilisateur demande l'annulation par la commande `Undo` de la dernière action effectuée, celle-ci est retirée de la pile des actions effectuées et des informations permettant de l'annuler sur l'état interne

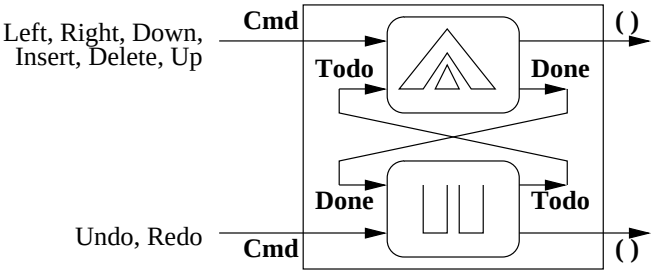

FIGURE 3.12 – Édition structurée avec presse-papiers.

de l'éditeur principal sont placées sur son port de sortie `Todo`. En même temps d'autres informations sont empilées dans la pile des actions annulées pour donner la possibilité de refaire ultérieurement toutes les actions annulées par la commande `Redo`. De la même manière quand une commande d'édition comme `Left`, `Up`, `Insert`, etc. est reçue par l'éditeur principal, celui-ci met à jour son état interne et renseigne le presse-papiers par son port de sortie `Done` sur la nature de l'action qui a été effectuée. Le presse-papiers empile cette information dans sa pile d'actions effectuées. Le code Haskell correspondant (utilisant le combinateur `<%>`) est donné par le listing suivant

```
-------------------------------------------------------------------------
grammarEditor :: EditorZp s p -> EditorCb s p -> EditorRt s p
grammarEditor e1 e2 = e1 <%> e2

runEditGram :: Grammar s p r -> EditorRt s p -> IO ()

type EditorZp s p = Editor (ZipperA s p) IO (Either Cmd Todo)
                                                (Either () Done)
type EditorCb s p = Editor (Buffer s p) IO (Either Done Cmd)
                                                (Either Todo ())
type EditorRt s p = Editor (ZipperA s p, Buffer s p)  IO
                                  (Either Cmd Cmd) (Either () ())

-- Etat de l'éditeur de presse-papiers
type Buffer s p = (RedoBuffer s p , UndoBuffer s p)
type RedoBuffer s p = [ActionE s p]
type UndoBuffer s p = [ActionE s p]
-------------------------------------------------------------------------
```

dans lequel l'état de l'éditeur principal, `EditorZp`, est un Zipper attribué de type `ZipperA s p` ; celui de l'éditeur de presse-papiers, `EditorCb`, un couple de type `Buffer s p` et l'état de l'éditeur résultat de la composition, `EditorRt`, est le couple formé des états de ses composantes. Les informations sur les actions effectuées ou annulées sont décrites par le type `ActionE s p`. Pour exécuter un éditeur, nous instancions la monade `m` par la monade des entrées-sorties `IO`. Et la fonction `runEditGram` permet finalement de tester dans un environnement ligne de commande l'éditeur résultat `grammarEditor e1 e2` pour toute

donnée représentée par une grammaire attribuée. Les détails d'implémentation sont donnés en annexe C.

3.6.2 Mise à jour de vue et relèvement de transformations

Problème de mise à jour de vue

Pendant une session réelle d'édition interactive, l'utilisateur final n'agit pas directement sur la structure interne de l'éditeur. Il la manipule à travers une vue abstraite proche du WYSIWYG[1]. A titre illustratif, considérons l'expression mathématique $\frac{1}{x} = (x + 2)$ conforme à la grammaire suivante :

```
Exp --> Id | Nb | Fr | ( Exp ) | Exp Op Exp | Fr Op Exp
Fr  --> Exp Exp
Nb  --> 0 | .. | 9
Id  --> a | .. |z | A | ..| Z
Op  --> + | - | = | * | /
```

L'arbre de dérivation pour cette expression est indiqué par la figure 3.13.a et représente la structure concrète manipulée pendant l'édition. Si on suppose que les symboles visibles sont ceux dont les nœuds y sont étiquetés par de petits points noirs, alors l'algorithme de projection (sur cet ensemble de symboles visibles) décrit par E. Badouel & M. Tchoupé [76, 77] permet d'obtenir la vue abstraite représentée par la figure 3.13.b. Toutes

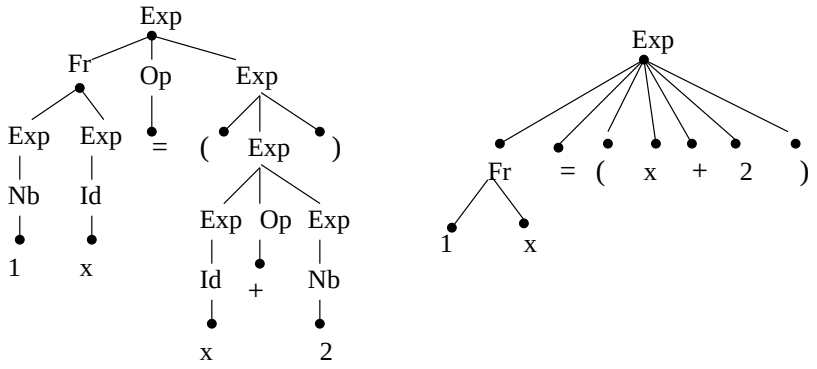

FIGURE 3.13 – **a)** Vue concrète **b)** Vue abstraite

les modifications effectuées sur la vue abstraite doivent être propagées sur la vue concrète afin de garder l'uniformité. Cette propagation des changements effectués sur la vue abstraite vers la vue concrète constitue le problème de mise à jour de vue ; un problème familier de la communauté des bases de données.

Greenwald & al. [78] ont proposé pour ce problème une solution utilisant une approche linguistique. Ils définissent un ensemble de combinateurs appelés lentilles (*lenses* en anglais), qui constituent un DSL pour les transformations bi-directionnelles d'arbres. Soient `C` un ensemble de structures concrètes et `A` un ensemble de vues abstraites associées.

[1] What You See Is What You Get

Définition 3.1 (Well-behaved lenses [78]). Une *lentille* `l` consiste en deux fonctions partielles : une fonction *get* définie de C vers A, notée $l\uparrow$, et une fonction *putback* de $A \times C$ vers C, notée $l\downarrow$. Une lentille a un *bon comportement (well-behaved lens* en anglais) si, et seulement si, ses fonctions *get* et *putback* obéissent aux lois *GETPUT* et *PUTGET* suivantes *:*

$$\textit{(GETPUT)} \qquad c \in \mathrm{dom}\ (l\uparrow) \Rightarrow l\downarrow\ (l\uparrow\ c,\ c) = c$$
$$\textit{(PUTGET)} \qquad (a,c) \in \mathrm{dom}\ (l\downarrow) \Rightarrow l\uparrow\ (l\downarrow\ (a,\ c)) = a$$

La fonction `get` associe à une vue concrète une vue abstraite, tandis que la fonction `putback` reconstitue la nouvelle structure concrète à partir de la vue abstraite modifiée et de la structure concrète initiale. Pour accomplir convenablement ces transformations bi-directionnelles d'arbres, les fonctions `get` et `putback` d'une lentille `l` doivent s'exécuter toutes d'un coup en aller et retour. Pour cela, elles doivent satisfaire aux lois *GETPUT* et *PUTGET*.

La loi *GETPUT* stipule que si une vue abstraite obtenue d'une vue concrète `c` n'est pas modifiée, en la remettant dans `c`, on obtient la même vue concrète de départ. La loi *PUTGET* stipule que si on met une vue abstraite `a` dans une vue concrète `c` et qu'on obtient une vue concrète `c'`, alors la vue abstraite obtenue de `c'` sera exactement la vue abstraite `a` de départ.

Dans la section qui suit, nous proposons une solution à ce même problème en décrivant le concept de relèvement (uniforme) de transformations ; qui associe à toute transformation admissible d'une vue abstraite, une transformation équivalente de la structure concrète. En particulier, nous utilisons le combinateur d'éditeur `<%>` pour exécuter convenablement ce comportement. Et nous établissons une correspondance bijective entre les relèvements uniformes de transformations et les lentilles ayant un bon comportement de Greenwald and al. [78].

Relèvement de transformations avec le combinateur <%>

Soient `C` un ensemble de vues concrètes, `A` l'ensemble de vues abstraites associées, une fonction de transformation `t` de vues abstraites correspond concrètement à une action d'édition comme l'insertion, la suppression de productions, ou simplement une opération de navigation. Nous notons par $T \subseteq A \rightarrow A$ l'ensemble des transformations admissibles, et par $\pi : C \rightarrow A$ la fonction de projection sur les symboles visibles. Comme mentionné dans [76], une même vue abstraite peut correspondre à un nombre infini de vues concrètes comme le montre la figure 3.14.

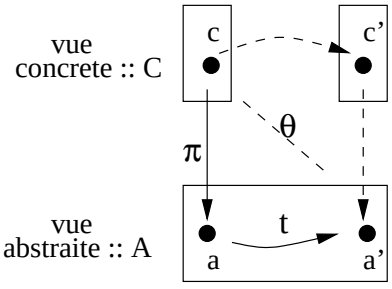

FIGURE 3.14 – Relèvement uniforme de transformations

Définition 3.2 (Relèvement uniforme de transformations). $\theta :: T \to (C \to C)$ est un *relèvement de l'ensemble des transformations admissibles* $T \subseteq A \to A$ *vers* C, *si* $\pi\ ((\theta\ t)\ c) = t\ (\pi\ c)$ pour tout $t \in T$ et $c \in C$. Le relèvement est dit *uniforme* si en plus, $t_1\ (\pi\ c) = t_2\ (\pi\ c) \Rightarrow \theta\ t_1\ c = \theta\ t_2\ c$.

Nous garantissons dans cette définition que la vue concrète $c' \in C$ doit être obtenue de manière non ambiguë en appliquant le relèvement θ sur la transformation t et sur la vue concrète c (i.e. $\theta\ t\ c$), tel que $\pi\ c' = a'$ avec $a' = t\ a$ et $\pi\ c = a$. Lorsqu'une action t est appliquée sur la vue abstraite courante, la transformation correspondante $t' :: C \to C$ obtenue par $\theta\ t$ est instantanément envoyée à l'éditeur de la vue concrète qui modifie la structure de l'objet édité. La projection π est aussi appliquée à la nouvelle vue concrète pour rafraichir l'état de l'éditeur de la vue abstraite avec la commande a'.

Cette situation correspond effectivement au problème de mise à jour de vue où nous avons explicitement pris en compte la transformation de la vue abstraite, un aspect encapsulé dans le langage des lentilles de Greenwald & al. [78]. La solution est obtenue en composant deux éditeurs associés respectivement à la représentation concrète et abstraite de l'objet édité. On utilise pour cela le combinateur <% comme le montre la figure 3.15.

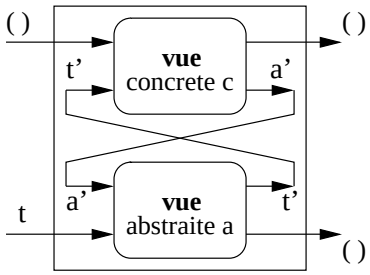

FIGURE 3.15 – Solution au problème de mise à jour de vue avec <%>

Proposition 3.3. *:* Il existe une correspondance bijective entre les *relèvements uniformes de transformations admissibles* $\theta :: T \to (C \to C)$ et les *lentilles* $l = (l\uparrow\ ,\ l\downarrow)$ ayant un *bon comportement,* vérifiant *:*

(i) $dom\ (l\uparrow) = C$ *et*

(ii) $\forall\ c \in C$, and $t \in T$, $(t\ (l\uparrow\ c),\ c) \in dom\ (l\downarrow)$

Preuve.

De θ vers l

- $l\uparrow = \pi$, $dom\ (l\uparrow) = C$.
- $dom\ (l\downarrow) = \{(a', c) \mid \exists\ t \in T,\ a' = t\ (l\uparrow\ c)\}$. Maintenant, $l\downarrow\ (a', c) = \theta\ t\ c$ où la transformation t est telle que $a' = t\ (l\uparrow\ c)$ et par uniformité de θ, ceci ne dépend pas du choix de t. La propriété *(ii)* est alors vérifiée par définition de $l\downarrow$

De l vers θ

- θ t c = $l\downarrow$ (t ($l\uparrow$ c) , c)

  $\pi\ (\theta$ t c) = $l\uparrow\ (l\downarrow$ (t ($l\uparrow$ c) , c))

  = t ($l\uparrow$ c) par (PUTGET)

  = t (π c)

– si $t_1\ (l{\uparrow}\ \mathrm{c}) = t_2\ (l{\uparrow}\ \mathrm{c})$ (i.e. si $t_1\ (\pi\ \mathrm{c}) = t_2\ (\pi\ \mathrm{c})$)
$\theta\ t_1\ \mathrm{c} \quad = \quad l{\downarrow}\ (t_1\ (l{\uparrow}\ \mathrm{c}), \mathrm{c}) \quad = \quad l{\downarrow}\ (t_2\ (l{\uparrow}\ \mathrm{c}), \mathrm{c}) \quad = \quad \theta\ t_2\ \mathrm{c}$

■

3.7 Conclusion

Nous avons défini dans ce chapitre un DSL plongé dans Haskell pour l'édition interactive de documents structurés. Ce langage est une illustration assez significative de la méthodologie de conception et d'assemblage de langages dédiés développée dans cette thèse qui utilise le formalisme des grammaires attribuées. On a noté la réutilisation de langages par changement de monades. Ce qui nous a permis de paramétrer l'interface utilisateur par la monade des `IO` et d'obtenir un environnement textuel d'édition de données structurées représentées par un Zipper attribué. Les systèmes FranTk [71] ou Fruit [73], a priori adaptés pour offrir un environnement graphique, se sont avérés inutilisables parce qu'ils n'ont pas été, entre autres, régulièrement maintenus par leurs auteurs. Nous envisageons, de ce fait, implémenter une GUI adaptée à nos éditeurs.

D'autre part, la représentation d'un Zipper basée sur la notion de couches préconisée par Sufrin et De Moor [23] a permis d'implémenter facilement les opérations d'édition. Cependant, en pratique, le calcul des attributs dans une telle représentation s'avère assez complexe. Une solution immédiate a consisté à ajouter à l'architecture de l'éditeur une strate supplémentaire (un préprocesseur) qui permet de traduire cette représentation en une structure correspondante, représentant les grammaires attribuées comme des transformateurs de Zippers [27, 38] ; la valeur d'un attribut étant alors calculée par réduction à une forme normale utilisant un automate d'arbres opérant sur cette représentation. Toutefois, un travail plus conséquent dans le futur serait d'offrir un DSL plongé dans Haskell pour le codage des évaluateurs d'attributs. Le programmeur pourra alors utiliser ces combinateurs pour spécifier sa grammaire attribuée (en écrivant surtout les règles sémantiques associées), et le faisant, il sera effectivement en train de construire un programme Haskell pour l'évaluateur correspondant. La structure de flèches (*arrows* en anglais) de J. Hughes [79], une généralisation des monades ; ou plus précisément la nouvelle notation des *arrows* introduite par R. Paterson [80] (semblable à la notation `do` pour les monades) pourront être utiles pour cette spécification.

Conclusion générale

Sommaire

Nous présentons dans ce chapitre conclusif un bilan général du travail réalisé dans cette thèse. Nous commençons par rappeler la problématique abordée, les objectifs fixés et les choix méthodologiques adoptés. Nous présentons ensuite les principaux résultats obtenus et en faisons une analyse critique. Ce qui nous permet de préciser quelques pistes de réflexion pour les travaux futurs à court et à long terme.

Rappel des objectifs et des choix méthodologiques

Nous nous sommes intéressés aux méthodes et techniques de spécification, de conception et d'assemblage de langages dédiés, avec des applications à la manipulation de documents structurés. Un objet structuré complexe est représenté intentionnellement sous la forme d'une structure arborescente décorée par des attributs. Les structures licites sont caractérisées par exemple par une DTD au format XML ou alternativement, ce que nous avons préféré, par une grammaire algébrique abstraite.

Martin Ward et Sergey Dmitriev [1, 2] ont introduit le paradigme de la *programmation orientée langage* pour unifier diverses approches post-objets. Selon eux, la programmation orientée langage permet au programmeur « d'être capable de travailler en termes de concepts et notions du problème qu'il est en train de résoudre, au lieu d'être obligé de traduire ses idées aux notions qu'un langage généraliste est capable de comprendre (classes, méthodes, boucles, conditionnels) ». La notion d'abstraction devient alors centrale dans l'écriture des logiciels efficaces. Paul Hudak [8, 9] indique que l'abstraction idéale et ultime pour une application particulière, est effectivement un langage conçu uniquement pour cette application : on parle de langage dédié. Ainsi, la programmation orientée langage utilise la technique de la métaprogrammation dans laquelle, le programmeur définit d'abord un ou plusieurs langages (dédiés) qui capturent les caractéristiques du domaine étudié, puis écrit les applications visées en utilisant ces langages.

Dans le cadre de cette thèse nous nous sommes donnés comme objectif principal l'illustration d'une démarche méthodologique de développement de logiciels basée sur la concep-

tion et la réutilisation de langages dédiés. Il s'est agit pour nous de proposer un formalisme pour la conception et l'assemblage de langages dédiés vus comme méthode de programmation. Notre volonté étant de conduire la même démarche méthodologique au niveau des langages que ce qui est classiquement fait au niveau des composants logiciels : comment peut-on créer de nouveaux langages par composition de langages réutilisables existants pour résoudre un problème ?

Cette question qui soulève de façon générale le problème d'extension de langages [20] est assez complexe, puisque la réutilisation d'un langage dédié met en jeu des aspects très différents. Pour pouvoir identifier des règles générales de spécification et de composition, nous avons illustré cette notion de réutilisabilité à travers quelques exemples représentatifs. Le formalisme des grammaires attribuées s'est avéré très adapté au propos. Nous avons tiré profit de leur traduction en programmes fonctionnels pour définir des spécifications exécutables de langages dédiés (vus comme composants logiciels) en utilisant une approche par combinateurs. Afin de faire coexister un certain nombre de langages, le système de typage du langage hôte (Haskell dans le cas d'espèce) peut s'est avéré insuffisant. Nous avons donc proposé un système simple de typage de langages dédiés, lié à la technique classique de traduction des grammaires attribuées en programmes fonctionnels.

La section qui suit fait une analyse des principaux résultats obtenus.

Principaux résultats obtenus

Les résultats que nous présentons ci-dessous font suite à un travail préliminaire [24, 25] dans lequel nous avons étudié et validé sur un fragment significatif du domaine (traitant de la manipulation des documents structurés mathématiques) l'approche préconisée dans cette thèse, qui utilise la technique des grammaires attribuées comme méthode de conception de systèmes logiciels. Nous y avions montré comment utiliser des grammaires attribuées pour définir l'affichage bidimensionnel et bidirectionnel de certains objets mathématiques.

Approche grammaticale de conception des langages dédiés. Nous avons décrit au premier chapitre comment les règles sémantiques d'une grammaire attribuée peuvent être automatiquement converties en une structure d'algèbre formée d'un ensemble de combinateurs fonctionnels pour la signature sous-jacente. Ces combinateurs constituent un langage dédié associé à cette signature. Parallèlement, nous avons montré l'utilité des algèbres pour obtenir des gammes de produits logiciels variés. Nous avons introduit à cette occasion une notation commune pour les algèbres et les grammaires attribuées basée sur le calcul des séquents et définie par : $inhs \vdash \langle exp \rangle \rhd synths$ pour signifier que l'évaluation des attributs pour l'expression close `<exp>` pour les attributs hérités `inhs` produit les attributs synthétisés `synths`. L'ensemble des règles sémantiques s'interprète alors comme un système formel formé d'axiomes et de règles d'inférences décrivant inductivement l'ensemble de ces séquents valides. L'intérêt principal de cette notation est qu'elle nous a permis de décrire plus facilement la sémantique opérationnelle des programmes paramétrés par des algèbres décrites par des grammaires attribuées.

Réutilisation de langages dédiés par changement de monades. Nous avons ensuite

montré dans la première partie du chapitre 2 que la traduction classique des grammaires attribuées en programmes fonctionnels peut s'étendre aux fonctions monadiques, c'est-à-dire lorsque les fonctions sémantiques de la grammaire attribuée sont composées dans la catégorie de Kleisli d'une monade. Ce qui nous a conduit tout naturellement à étendre le formalisme de représentation des grammaires attribuées ordinaires à des grammaires dont les fonctions sémantiques peuvent produire des effets. Nous avons été contraint cependant de nous restreindre aux grammaires attribuées fortement non circulaires. En effet, les monades qui permettent d'effectuer et de composer en Haskell des calculs imposent de connaître de façon statique l'ordre d'évaluation des attributs, information qui peut être obtenue lorsque la grammaire attribuée est fortement non circulaire. Nous avons illustré cette technique sur l'exemple bien connu des combinateurs fonctionnels d'analyse [59].

Stratification horizontale de langages dédiés. Un langage dédié peut être conçu par une grammaire attribuée dont les règles sémantiques sont décrites par un autre langage dédié. Ce qui en d'autres termes montre que la traduction des grammaires attribuées en algèbres de combinateurs fonctionne également dans une catégorie dont les calculs sont associés à un langage dédié. Pour l'illustrer, nous avons considéré le langage FAL [16] dédié à l'animation réactive, comme langage hôte pour l'écriture des règles sémantiques. Ce langage nous a permis de décrire des extensions du langage de boîtes positionnées introduit au chapitre 1. Nous avons ainsi pu concevoir et implémenter :

- Un langage dédié à l'affichage graphique et visuel des boîtes positionnées. Et en y adjoignant des attributs supplémentaires et des règles sémantiques appropriées (utilisant les fonctions de FAL), nous avons permis que ces boîtes puissent s'animer en réaction aux actions d'un utilisateur externe sur celles-ci.
- Un jeu de billard "d'une balle en un coup" dont la table est formée de l'ensemble des boîtes assemblées et les trous des frontières communes à ces boîtes. La balle rebondit aux frontières non communes et traverse les trous jusqu'à arrêt ou réinitialisation par l'utilisateur.

Ces extensions nous ont permis de montrer que la méthode préconisée dans cette thèse offre une sorte de stratification horizontale de langages dédiés liée à la notion de composition descriptionnelle des grammaires attribuées développée par Ganzinger et Giegerich [62].

Typage de langages dédiés. À partir de ces différentes extensions nous avons pu définir un système de types pour les langages dédiés proche du typage des composants pour leur assemblage. Ce typage est basé sur la transformation de grammaires attribuées en programmes fonctionnels : un langage dédié, associé à un symbole grammatical, est un composant dont le type est défini par une relation bipartie entre attributs hérités (ports d'entrée) et attributs synthétisés (ports de sortie) guidée par les graphes hérités-synthétisés HS. Nous avons donné des conditions permettant l'assemblage de tels composants ; cet assemblage étant guidé par le graphe de dépendances locales GDL associé à la production sous-jacente.

Bibliothèque de combinateurs d'éditeurs. Une partie importante du travail que nous avons réalisé concerne une étude de cas d'un langage de combinateurs dédié à l'édition interactive de documents structurés. Un éditeur de données structurées est un

programme interactif qui réagit aux actions de l'utilisateur (navigation dans le document, insertion ou suppression d'une production, copier-coller, ...) pour modifier les objets édités et produire incrémentalement un résultat. La représentation de l'objet édité doit pouvoir être localisable (par un point focal où les actions d'édition courantes s'effectuent) et partiellement définie (puisque le document en cours d'édition reste incomplet). Nous avons, à ce propos, utilisé le type de donnée Zipper de Huet [3]. Nous avons défini un éditeur comme une fonction de ses ports d'entrée vers une monade d'état produisant comme valeurs ses ports de sortie, et paramétrée par une monade arbitraire `m` permettant de capturer d'autres effets. Sur cette base nous avons constitué une bibliothèque de combinateurs qui peuvent se répartir en deux catégories.

- Combinateurs généraux : La généralité signifie ici que le type de leur état est polymorphe. Ce sont les combinateurs `<+>`, `<@`, `<%>` et `<.>` implémentant respectivement l'union disjointe de deux éditeurs, l'éditeur de boucle, la composition de deux éditeurs dont certains échanges de commandes restent transparents à l'utilisateur, et la composition naturelle de deux éditeurs. Ils représentent une solution, utilisant une approche grammaticale, aux problèmes que nous avons traités dans [72].
- Combinateurs liés au type de donnée Zipper : Ce sont principalement les combinateurs `<*>`, `<|>` et `<*` permettant de combiner des éditeurs en suivant la structure grammaticale du document à éditer. Les deux premiers implémentent respectivement la composition séquentielle et alternative de deux éditeurs, tandis que le dernier implémente la composition séquentielle d'un nombre arbitraire d'éditeurs (étoile de Kleene). Le fonctionnement d'un éditeur de base est essentiellement lié aux opérations classiques d'édition dont l'implémentation est canoniquement liée à la structure de donnée Zipper et aux productions associées.

Nous avons illustré ces combinateurs en assemblant dans un premier temps (grâce aux combinateurs `<*>` et `<|>`) un éditeur simple d'objets structurés. Et afin de permettre d'annuler ou de refaire les actions précédentes, nous avons combiné cet éditeur (grâce au combinateur `<%>`) avec son presse-papiers (vu aussi comme un éditeur). Nous avons testé et validé son fonctionnement grâce à une interface utilisateur textuelle (ligne de commande) obtenue en prenant pour `m` la monade des `IO` comme on peut le voir dans l'annexe C.

Critique des résultats et perspectives

Nous donnons ici les limitations que nous entrevoyons au travail réalisé et des pistes de travail pour des extensions futures.

Généralisation aux arrows et sémantique monadique par point fixe.

L'implémentation fonctionnelle des grammaires attribuées classiques interprète une production comme une fonction ayant pour arguments les types sémantiques de ses symboles en partie droite et pour résultat le type sémantique de son symbole en partie gauche (voir figure 1.9 page 31). Les types sémantiques sont des fonctions des attributs hérités vers les attributs synthétisés du symbole associé, et on a :

```
trans_p :: (inhs1 -> syns1) -> ... -> (inhsN -> synsN)
                                          -> (inhs0 -> syns0)
```

On a aussi observé que cette traduction peut s'étendre sous certaines conditions (grammaires FNC, monade additive et alternative) aux fonctions monadiques i.e. lorsqu'en retournant la valeur d'un attribut synthétisé, ces fonctions peuvent produire des effets (voir figure 2.1 de la page 45).

```
trans_p :: (Monad m) => (inhs1 -> m syns1) -> ... ->
                              (inhsN -> m synsN) -> (inhs0 -> m syns0)
```

A court terme, nous envisageons de généraliser cette traduction en utilisant la structure des *arrows* (flèches) de J. Hughes [79], une généralisation des monades, introduite pour répondre à certains problèmes ne trouvant pas de solutions avec la structure des monades. Ce qui se traduirait en Haskell par le type suivant pour une certaine flèche `arr` de la classe `Arrow`.

```
trans_p :: (Arrow arr) => (arr inhs1 syns1) -> ... ->
                                (arr inhsN synsN) -> (arr inhs0 syns0)
```

Cependant, même avec cette généralisation, l'hypothèse que la grammaire attribuée soit fortement non circulaire n'a aucune chance d'être levée. On devra toujours indiquer un ordre d'évaluation des variables liées dans une expression utilisant la notation des flèches pour l'écriture des fonctions sémantiques monadiques. Ainsi, pour ne pas restreindre le type de grammaires traitées (FNC), nous envisageons de proposer une sémantique monadique par point fixe. La notation μdo [60, 61], une extension de la notation *do* usuelle qui autorise le calcul récursif sur les valeurs produites par les monades, pourra être utile à cet effet.

Stratification de la méthodologie de développement logiciel. Le système de typage des langages dédiés que nous avons proposé n'offre pas un pouvoir d'expression suffisant pour s'appliquer à toutes les techniques d'assemblage connues.

- Un premier travail consistera à définir une classe générale de grammaires attribuées sur laquelle ce typage devra se reposer et permettre ainsi la coexistence d'un grand nombre de langages dédiés.
- Un projet plus ambitieux consisterait à définir chaque étape de développement du génie logiciel par stratifications de modèles de langages dédiés, de telle sorte que chaque étape (d'analyse, de conception, d'implémentation, ou de déploiement) du développement d'une application s'effectue par stratification de modèles de langages appropriés. Il s'agira principalement de définir les différents langages appropriés, dédiés à l'analyse d'un projet, à sa conception, son implémentation, son déploiement, etc.

Évaluation incrémentale d'attributs dans un Zipper. Nous avons donné dans ce manuscrit une représentation des grammaires attribuées par une structure de Zipper attribué, nécessaire pour décrire la structure des documents édités. Cette représentation a facilité de manière convenable l'implémentation des opérations d'édition. Cependant certaines opérations d'édition, par exemple, l'insertion ou la suppression d'une production, impliquent une réévaluation des attributs afin de garder la

conformité du document. Nous avons introduit une autre façon de représenter les grammaires attribuées comme des transformateurs de zippers ; et implémenté une nouvelle technique d'évaluation des attributs d'une grammaire attribuée potentiellement circulaire [27,38]. Une solution immédiate a consisté à ajouter à l'architecture de l'éditeur une strate supplémentaire (un préprocesseur) permettant de traduire la représentation d'origine en une structure correspondante de la nouvelle représentation, d'évaluer les attributs sur cette structure en utilisant notre algorithme et de revenir à la structure de départ. Bien sûr ce calcul doit se faire d'un seul coup sans interruption et le combinateur `<%>` a été utile à ce effet.

Cependant cette solution naïve n'est pas optimale et est sans doute très coûteuse de part même sa description, puisqu'un préprocesseur est nécessaire là où on aurait souhaité effectuer un calcul direct sur la représentation de départ. Bien plus, on aimerait ne recalculer que les valeurs d'attributs touchés par la modification. Il s'agira probablement là d'une nième façon de calculer de manière incrémentale les attributs d'une grammaire attribuée représentée par un zipper (attribué) qui, cette fois-ci, va transformer un zipper attribué en un autre zipper attribué.

GUI pour l'édition et la génération d'éditeurs structurés. L'interface utilisateur est un aspect important dans un environnement d'édition. Elle permet l'interaction avec le monde extérieur et en particulier avec l'utilisateur final. En paramétrant le type d'un éditeur par une monade `m`, on s'est donné une certaine flexibilité dans la définition des interfaces utilisateur appropriées. L'environnement textuel obtenu en prenant pour `m` la monade des entrées-sorties (`IO`) n'a été utile que pour tester et valider notre approche. On espérait par la suite tirer profit des systèmes FranTk [71] et surtout Fruit [73], a priori adaptés pour obtenir un environnement graphique approprié. Mais ces systèmes se sont avérés inutilisables parce qu'une maintenance rigoureuse n'a pas suivi leur déploiement. Ainsi :

- Nous envisageons dans un premier temps, de concevoir et implémenter une GUI adaptée à nos éditeurs en utilisant une approche grammaticale. Ce qui constituera une deuxième étude de cas de conception d'un DSL par la méthodologie développée dans cette thèse.
- On pourra ensuite envisager une génération d'éditeurs à partir d'une signature (grammaires attribuées et/ou XML) spécifiée dans un DSL visuel décrit à cet effet. On tendrait ainsi, à coup sûr, vers une utilisation de la méthode en industrie où l'ingénieur génère rapidement les éditeurs dont il a besoin pour résoudre un problème particulier.

Bibliographie

[1] Martin Ward. Language oriented programming. *Software - Concept and Tools*, 15(4) :147–161, 1994.

[2] Sergey Dmitriev. Language oriented programming : The next programming paradigm. *http ://www.onboard.jetbrians.com/articles/04/10/lop/index.html*, Novembre 2004.

[3] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :pp. 549–554, September 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

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

[6] *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[7] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24 :131–183, 1992.

[8] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[9] Paul Hudak. Modular domain specific languages and tools. *ACM Computing Surveys*, 28A(4) :196, december 1996.

[10] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

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

[12] M. F. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. Technical Report RUU-CS-86-16, Department of Information and Computing Sciences, Utrecht University, 1986.

[13] 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.

[14] 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.

[15] 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.

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

[17] B. Mayol. Attribute grammars and mathematical semantics. *SIAM Journal of Computing*, 3(3) :503–518, 1981.

[18] L.M. Chirica and D.F. Martin. An order-algebraic definition of knuthian semantics. *Mathematical System Theory*, 13(1) :1–27, 1979.

[19] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21 :239–250, 1984.

[20] Eric Badouel and Marcel Tonga. Growing a domain specific language with split extensions. Research Report 6314, INRIA, October 2007.

[21] Shin-Ya Katsumata. Attribute grammars and categorical semantics. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2008.

[22] Gerd Szwillus and Lisa Neal, editors. *Structure-based editors and environments.* Academic Press, Inc., Orlando, FL, USA, 1996.

[23] B. Sufrin and O. de Moor. Modeless structure editing. In *J. Davies, A. W. Roscoe and J.C.P. Woodcock (editors)*, Proceedings of the OxfordMicrosoft symposium in Celebration of the work of Tony Hoare, September 13-15 1999.

[24] Bernard T. Fotsing and Georges Edouard Kouamou. Une approche formelle de description et de manipulation des objets structurés mathématiques. In Yahya Slimani Emmanuel Kamgnia, Bernard Philippe, editor, *Proceedings of the 7th African Conference on Research in Computer Science and Applied Mathematics (CARI'04)*, pages 263–271, October 2004.

[25] Bernard T. Fotsing and Georges Edouard Kouamou. Une approche formelle de description et de manipulation d'objets structurés mathématiques. *Electronic Journal of ARIMA (African Revue in Informatics and Mathematics Applied)*, 3 :71–86, November 2005.

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

[27] Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Yet another implementation of attribute evaluation. Research Report 6315, INRIA, October 2007.

[28] Etienne Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle.* PhD thesis, Université d'Orléans, oct 1998. 200 pages.

[29] CRUZ LARA SILVA S. *GEODE : un système pour la génération d'environnements de programmation intégrés.* PhD thesis, Institut National Polytechnique de Lorraine, novembre 1988.

[30] P. M. Lewis, D. J. Rosenkratz, and R. E. Stearms. Attribute translations. *J. Comput. Syst. Sci.*, 9(3) :pp. 279–307, 1974.

[31] G. V. Bochman. Semantics evaluation from left to right. *Communications of the ACM.*, 19(2), February 1976.

[32] K. Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *In Conference Record of the 3rd ACM Symposium on Principles of programming Languages*, pages 32–49, Atlanta, GA, 1976. ACM.

[33] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3) :pp. 229–256, 1980.

[34] B. Lorho. Semantics processing in the system delta, methods of algorithmic language implementation. In *A. Ershov, C.H.A. Koster, ed.* LNCS 47. Springer-Verlag, 1977., 1977.

[35] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In *INRIA Rocquencourt*, Décembre 1984.

[36] Didier Parigot. *Transformations, Evaluation Incrémentale et Optimisation des Grammaires Attribuées : Le système FNC-2.* PhD thesis, Université de Paris-Sud, Orsay, 1988.

[37] Didier Parigot. Le système de grammaires attribuées FNC-2, http ://www-rocq.inria.fr/oscar/www/fnc2/fnc2-fra.html/, 1998.

[38] Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Attribute grammars as recursion schemes over cyclic representations of zippers. In Elsevier, editor, *Proceedings of the Workshop on Mathematically Structured Functional Programming (MSFP 2008), Reykjavik, Iceland*, July 2008.

[39] Mehdi Jazayeri, William F. Ogden, and William C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM*, 18 :667–706, 1975.

[40] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics.* North Holland Publishing Company, Amsterdam, 1984.

[41] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[42] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[43] Simon Peyton Jones. Tackling the awkward squad : monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2001.

[44] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058, pages 219–234. Springer-Verlag, 1996.

[45] Guy L. Steele. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages : Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[46] Jose Emilio Labra Gayo, Juan Manuel Cueva Lovelle, Maria Candida Luengo Diez, and Bernardo M. Gonzalez Rodriguez. A language prototyping tool based on semantic building blocks. In *EUROCAST*, pages 597–611, 2001.

[47] Eric Van Wyk. Aspects as modular language extentions. *Electronic Notes in Theorical Computer Science*, 82 No. 3 :20, 2003.

[48] Jimin Gao, Mats Heimdahl, and Eric Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering, FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 102–116. Springer Verlag, March 2007.

[49] E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver : an extensible attribute grammar system. In *Proc. of LDTA 2007, 7^{th} Workshop on Language Descriptions, Tools, and Analysis*, 2007. To appear in ENTCS.

[50] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[51] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International conf. on Compiler Construction Volume 2304 of LNCS pages 128-142. Sprigger-Verlag*, 2002.

[52] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145, ACM, July 1989.

[53] Harald Vogt. *Higher order Attribute Grammars.* Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands, February 1993.

[54] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic attribute grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of Lect. Notes in Comp. Sci., pages 122–136, Aachen, Springer-Verlag, September 1996.

[55] Dawson R. Engler. Incorporating application semantics and control into compilation. In *In Proceedings of the Conference on Domain-Specific Languages (DSL-97*, pages 103–118. USENIX Association, 1997.

[56] John C. Reynolds. Definitional interpreters for higher-order programming languages. *ACM '72 : Proceedings of the ACM annual conference*, pages 717–740, 1972.

[57] M. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects, 2003.

[58] Jeroen 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.

[59] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4) :437 – 444, 1998.

[60] Levent Erkök and John Launchbury. Recursive monadic bindings : Technical development and details. Technical report, Oregon Graduate Institute of Science and Technology, june 2000.

[61] Levent Erkôk. *Value Recursion in Monadic Computations*. PhD thesis, OGI School of Science and Engineering, OHSU, october 2002.

[62] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. *ACM SIGPLAN Notices*, 1986. ACM SIGPLAN'84 Symposium on Computer Construction Montréal, pp 157–170, June 1984.

[63] P. Wadler. Deforestation : Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Berlin : Springer-Verlag, 1988.

[64] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute grammars and functional programming deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, Paris, France, 1997.

[65] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Composition symbolique. In *Journées Francophones des Langages Applicatifs*, Come, Italie, 1998.

[66] Eric Badouel and Rodrigue Djeumen. Modular grammars and splitting of catamorphisms. Research Report 6313, INRIA, October 2007.

[67] François Pottier. *Types et contraintes*. Mémoire d'habilitation à diriger des recherches, Université de Paris 7, December 2004.

[68] Wolfram Kahl, Oliver Braun, and Jan Scheffczyk. Editor combinators - a first account. Technical Report Nr. 2000-01, Fakulät für Informatik, Universität des Bundeswehr München, 36 pages, juni 2000.

[69] Olivier Braun. Editors combinators : Improving the user interface. Master's thesis, Université de Munich, 2000.

[70] Jan Scheffczyk. Declarative user interfaces for editor combinators, Studienarbeit, UniBwM IS-10/2000, Universität des Bundeswehr München, 2000.

[71] Meuring Sage. FranTk - a declarative gui system for haskell, 1999, http ://haskell.cs.yale.edu/FranTk/.

[72] Bernard T. Fotsing. Interactive editing of tree-structured data. In Marc Kokou Assogba Eric Badouel, Abderrahmane Sbihi, editor, *Proceedings of the 9th African Conference on Research in Computer Science and Applied Mathematics (CARI'08)*, pages 711–718, October 2008.

[73] Antony Alexander Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, Faculty of the Graduate School - Yale University, May 2004.

[74] L. C. Paulson. ML for the working programmer, Cambridge University Press, 1991.

[75] T. Uustalu and V. Vene. Comonadic functional attribute evaluation. In *M. van Eekelen, ed.*, Proc. of 6th Symposium on Trends in Functional Programming,TFP'05 (Tallinn, Sept. 2005), pages 33–43. Institute of Cybernetics, September 2005.

[76] 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.

[77] Maurice T. Tchoupé. *Une approche grammaticale pour la fusion des réplicats partiels d'un document structuré : application à l'édition collaborative asynchrone.* PhD thesis, En cotutelle entre l'Université de Yaoundé 1 et l'Université de Rennes 1, Août 2009.

[78] Michael B. GreenWald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. A language for bi-directional tree transformations. Technical report, MS-CIS-03-08 -University of Pennsylvania, August 5, 2003.

[79] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3) :67–111, 2000.

[80] Ross Paterson. A new notation for arrows. In *In International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[81] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

Annexe A

Évaluation comonadique des attributs dans un arbre d'arité variable

Nous présentons dans cette annexe une extension sur un Zipper d'arité variable (*rose trees*) de l'algorithme d'évaluation comonadique des attributs introduit par Uustalu & Vene [75] sur un Zipper d'arité 2. Cet algorithme s'appuie sur la notion de comonade i.e. d'un calcul dépendant du contexte.

A.1 Comonades, structure de Zipper et fonctions de navigation

Les comonades sont une approche duale des monades. Les catégories de coKleisli rendent les comonades utiles pour l'analyse des notions de fonctions dépendant du contexte. Si l'objet $D\ A$ est vu comme le type des valeurs de A situées contextuellement, une fonction dépendant du contexte de A vers B est un diagramme $D\ A \to B$ dans la catégorie de base, i.e. un diagramme de A vers B dans la catégorie de coKleisli.

Le *counit* $\epsilon_A : D\ A \to A$ débarasse le contexte de ses entrées pendant que la coextension $k^+ : D\ A \to D\ B$ d'une fonction $k : D\ A \to B$ le duplique essentiellement.

En Haskell, on définit les comonades comme un constructeur de type par

```
class Comonad w where
    counit :: w a -> a
    cobind :: (w a -> b) -> w a -> w b
```

Pour des besoins de commodité avec les structures de données définies par Uustalu et al., nous représentons un arbre d'arité variable comme suit :

```
data RTree a = a :< Trunk (RTree a)
data Trunk x = Node [x]
```

Le contexte détermine tous les noeuds au-dessus et aux alentours du point focal de l'arbre. On le définit en Haskell par

```
data Cxt a = Nil | Cxt a :> Local a (RTree a)
data Local a x = Loc {attr :: a, leftL, rightL :: [x]}
```

Un Zipper est un arbre localisé par un point focal i.e. un contexte et un sous-arbre :

```
data CxtRTree a = Cxt a :=| RTree a
```

Avec la structure de Zipper définie précédemment, les fonctions de navigation s'implémentent "aisément" comme dans le listing ci-après :

```
-----------------------------------------------------------------------
-- Remonter simplement au parent
goUp :: CxtRTree a -> Maybe (CxtRTree a)
goUp (Nil :=| as) = Nothing
goUp ((az :> Loc a ls rs) :=| t) =
                          Just (az :=| (a :< Node (ls++[t]++rs)))

-- Aller directement à gauche
goLeft :: CxtRTree a -> Maybe (CxtRTree a)
goLeft (Nil :=| as) = Nothing}
goLeft ((az :> Loc a ls rs) :=| t) = case ls of
     [] -> Nothing
     _  -> Just (az :> Loc a (init ls) ([t] ++ rs) :=| (last ls))

-- Aller directement à droite
goRight :: CxtRTree a -> Maybe (CxtRTree a)
goRight (Nil :=| as) = Nothing
goRight ((az :> Loc a ls rs) :=| t) = case rs of
     [] -> Nothing
     _  -> Just (az :> Loc a (ls++ [t]) (tail rs) :=| (head rs))

-- Descendre au i-ème fils
goI :: Int -> CxtRTree a -> Maybe (CxtRTree a)
goI i (az :=| (a :< Node xs)) =
   if (elem i [0..(length xs -1)]) then
    Just (az :> Loc a (take i xs) (drop (i+1) xs)  :=| (xs !! i))
   else Nothing

-- Descendre dans le Zipper à un noeud précis (le détruire)
goChildren :: CxtRTree a -> Trunk (CxtRTree a)
goChildren (az :=| (a :< Node xs)) = Node cxtls
     where cxtls = [(az :> Loc a ls rs) :=| t |
                    (ls,t,rs) <- prefXsuff xs]

-- Remonter dans le contexte (reconstruire le Zipper)
goParSibl :: CxtRTree a -> Maybe (Local (CxtRTree a) (CxtRTree a))
goParSibl (Nil :=| as) = Nothing
goParSibl ((az :> Loc a ls rs) :=| t) = Just (Loc par leftSibls rightSibls)
           -- le contexte tree parent
```

```
     where par = (az :=| (a :< Node (ls++[t]++rs)))
           -- la liste des contextes tree des freres gauches
           leftSibls = [(az :> Loc a ls1 (rs1++[t]++rs)) :=| t1 |
                          (ls1,t1,rs1) <- prefXsuff ls]
           -- la liste des contextes tree des freres droites
           rightSibls = [(az :> Loc a (ls++[t]++ls2) rs2) :=| t2 |
                           (ls2,t2,rs2) <- prefXsuff rs]

-- prefXsuff :: Show a => [a] -> [([a],a,[a])]
prefXsuff :: [a] -> [([a],a,[a])]
prefXsuff [] = []
prefXsuff (x:xs) = ([],x,xs) : [(x:x1, y1, z1) |
                                (x1,y1,z1) <- prefXsuff xs]
-------------------------------------------------------------------------
```

A.2 Évaluation comonadique d'attributs dans un Zipper

La suite consiste à définir la structure de Zipper comme une comonade i.e. comme une instance de la classe `Comonad`. On obtient donc

```
-------------------------------------------------------------------------
instance Comonad CxtRTree where
 counit (_ :=| (a :< _)) = a
 cobind k d = cobindC k d :=| cobindT k d
     where cobindC k d = case goParSibl d of
        Nothing -> Nil
        Just (Loc par lSibls rSibls) -> cobindC k par :>
                                  Loc (k par) lSiblsCob rSiblsCob
         where lSiblsCob = (map (cobindT k) lSibls)
               rSiblsCob = (map (cobindT k) rSibls)
           cobindT k d = k d :< case goChildren d of
             Node as -> Node (map (cobindT k) as)
-------------------------------------------------------------------------
```

Basée sur la structure de comonade, l'évaluation est une extension des travaux effectués par Uustalu et Vene [75]. La valeur locale d'un attribut dépend des valeurs d'attributs de son contexte environnant. Le Zipper enregistre aussi bien la valeur locale, que les chemins inférieur, supérieur et environnant d'un attribut. L'évaluation est donnée par le code suivant :

```
-------------------------------------------------------------------------
-- Classe pour la recherche des valeurs d'attributs des fils
class Synth d where
 children :: d a -> Trunk a
```

```
-- Classe pour la recherche des valeurs d'attributs du parent et
-- des frères d'un noeud
class Inh d where
 parSibl :: d a -> Maybe (Local a a)

-- Morphismes de coKleisli représentant les fonctions d'arbres
class TF d where
 run :: (d a -> b) -> RTree a -> RTree b

-- Morphismes de CoKleisli interprétés comme les fonctions d'arbre
-- pour les contextes des rose trees
instance TF CxtRTree where
 run k as = bs where Nil :=| bs = cobind k (Nil :=| as)

-- Valeurs des fils pour un Rose trees et son contexte
instance Synth CxtRTree where
 children (_ :=| (_ :< Node xs)) = Node (map counit xs)

-- Calcul des valeurs d'attributs du parent et des frères d'un
-- noeud (si un contexte local au moins existe)
instance Inh CxtRTree where
 parSibl (Nil :=| _)  = Nothing
 parSibl ((_ :> (Loc val lSibls rSibls)) :=| _) =
                                    Just (Loc val lvals rvals)
                where lvals = (map counit lSibls)
                      rvals = (map counit rSibls)

-- Structure de comonade pour les rose trees --
instance Comonad RTree where
 counit (a :< _) = a
 cobind k d@(_ :< (Node xs)) = k d :< Node (map (cobind k) xs)

instance TF RTree where
 run = cobind

-- Valeurs des fils d'un noeud un Rose Trees
instance Synth RTree where
 children (_ :< (Node xs)) = Node (map counit xs)
-----------------------------------------------------------------------
```

Annexe B

Le langage FAL de Paul Hudak

B.1 Description

FAL (Functional Animation Language) [16] est une extension du langage Fran (Functional Reactive Animation) [81], un DSL plongé dans Haskell pour l'animation réactive 2D. Ses éléments de base sont constitués d'une collection de types de données et de fonctions définis pour la construction de formes géométriques comme des cercles, des rectangles, des triangles, des polygones, etc.

```
data Shape  = Rectangle Side Side  | Ellipse Radius Radius
            | RtTriangle Side Side | Polygon [Vertex]
            | Polyline [Vertex]

type Vertex = (Float, Float)
```

Afin d'accroître les fonctionnalités des formes géométriques, on les combine pour obtenir des régions plus larges, ou des régions qui se chevauchent.

```
data Region = Shape Shape                -- primitive shape
            | Translate Vector Region    -- translated region
            | Scale Vector Region        -- scaled region
            | Complement Region          -- inverse of region
            | Region `Union` Region      -- union of regions
            | Region `Intersect` Region  -- intersection of regions
            | Empty                      -- empty region

type Vector = (Float,Float)
```

Ces définitions (`Shape` et `Region`) ne sont rien d'autre que des types de données Haskell. Leur sémantique dénotationnelle (signification) est liée à ce qu'ils représentent, et est équivalente aux fonctions mathématiques `containsS` et `containsR` qui déterminent si un point se trouve à l'intérieur ou à l'extérieur d'une forme géométrique ou d'une région.

```
containsS :: Shape -> Vertex -> Bool
containsR :: Region -> Vertex -> Bool
```

La troisième couche construite sur cette structure est le type de données `Picture` qui donne la possibilité de colorier des régions, de les composer en les plaçant les unes « sur » les autres, et de pouvoir les afficher sur un bureau virtuel dans une fenêtre graphique.

```
data Picture = Region Color Region | Picture `Over` Picture | EmptyPic
```

La fonction $drawInWindow :: Window \rightarrow Graphic \rightarrow IO()$ est utile pour dessiner une valeur graphique dans une fenêtre. Des opérateurs adéquats sont alors nécessaires pour traduire les structures de données précédentes en valeurs graphiques (*Graphic*) :

```
shapeToGraphic  :: Shape -> Graphic
regionToGraphic :: Region -> Graphic
picToGraphic    :: Picture -> Graphic
```

B.2 Évènements et comportements

Les notions de comportements et d'évènements polymorphes sont au centre du langage FAL. Un comportement (*Behavior*) est une valeur (polymorphe) continue variant dans le temps. Une représentation « raisonnable » pour les comportements est donnée par :

```
newtype Behavior a = Behavior ([Maybe UserAction], [Time]) -> [a])
```

Dans cette définition, l'horodatage des actions utilisateur (*Maybe UserAction*) permet d'indiquer le moment que celles-ci apparaissent (*Just ua*) ou pas (*Nothing*). Les valeurs des comportements de type *a* sont échantillonnées par le temps et éventuellement l'action d'un utilisateur externe.

Un évènement est une sorte de comportement qui indique à chaque instant si l'évènement se produit ou non. Ainsi la signature en Haskell des évènements devrait être un synonyme du type de données *Behavior* i.e *type Event a = Behavior (Maybe a)*. Toutefois, pour des raisons de sécurité du type, l'auteur les représente comme un nouveau type de données par :

```
newtype Event a = Event ([Maybe UserAction], [Time]) -> [Maybe a])
```

Ces deux notions de comportements et d'évènements sont au centre de la définition et de la signification des différents opérateurs de FAL. Ainsi, le comportement *time* qui représente le temps à l'infini, ne dépend pas des actions utilisateur et est défini par :

```
time :: Behavior Time
time = Behavior (\_,ts) -> ts)
```

La fonction *constB* ci-dessous, permet de relever toute valeur constante au niveau des comportements. Par exemple, le comportement constant *red* est un flux infini de valeurs de type *Red* représentant la couleur rouge.

```
constB :: a -> Behavior a
constB x = Behavior (\_ -> repeat x)

red = constB Red
```

La stratégie de relèvement des fonctions standards de Haskell consiste à définir un opérateur de relèvement généralisé *($*)* et d'en déduire les opérateurs concrets de relèvement (*lift1, lift2, lift3*, etc.) des fonctions Haskell, de sorte que par exemple, les opérateurs *(<*)* et *(>*)* représentent respectivement les opérateurs de comparaison d'infériorité et de supériorité strictes pour les comportements.

```
-------------------------------------------------------------------------
($*) :: Behavior (a -> b) -> Behavior a -> Behavior b
(Behavior ff) $* (Behavior fb) =
               Behavior (\uts -> zipWith ($) (ff uts) (fb uts))
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift1 f b1 = lift0 f $* b1

lift2 :: (a -> b -> c) -> (Behavior a -> Behavior b -> Behavior c)
lift2 f b1 b2  = lift1 f b1 $* b2
...
lift6 f b1 b2 b3 b4 b5 b6  = lift5 f b1 b2 b3 b4 b5 $* b6
...

(>*), (<*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
(>*)       = lift2 (>)
(<*)       = lift2 (<)

pairB :: Behavior a -> Behavior b -> Behavior (a,b)
pairB = lift2 (,)
fstB :: Behavior (a,b) -> Behavior a
fstB  = lift1 fst
sndB :: Behavior (a,b) -> Behavior b
sndB  = lift1 snd
-------------------------------------------------------------------------
```

On peut aussi coupler à chaque instant les valeurs de deux comportements par la fonction *pairB*, ou filtrer la première valeur (*fstB*) respectivement la deuxième valeur (*sndB*) d'un comportement de valeurs couplées, etc.

B.3 Réactivité

Les combinateurs *untilB* and *switch* représentent la jointure des comportements et des évènements dans FAL. Ils sont cruciaux pour la sémantique de FAL en terme de réactivité.

```
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
switch :: Behavior a -> Event (Behavior a) -> Behavior a
```

b `untilB` es se comporte initialement comme *b*, puis bascule au premier comportement dans le flux d'évènements *es* quand celui-ci se produit.
b `switch` es se comporte initialement comme *b*, puis bascule au premier comportement

dans le flux d'évènements *es* quand celui-ci se produit, ensuite au suivant, et ainsi de suite.
Les évènements de base (primitives) sont filtrés par les actions utilisateur. Par exemple, l'évènement *lbp : : Event ()* est filtré par le constructeur *Button p True True* du type de données *Event* au moment où le clic du bouton droit de la souris se produit. On peut marquer une valeur de type *b* lorsqu'un évènement de type *a* se produit avec l'opérateur $(- >>)$. L'opérateur $(=>>)$ est similaire à la fonction *fmap* et sa signification est immédiate.

```
(->>) :: Event a -> b -> Event b
(=>>) :: Event a -> (a -> b) -> Event b
```

Les prédicats *while* et *when* transforment un comportement booléen en un évènement.

```
while :: Behavior Bool ->  Event ()
when  :: Behavior Bool ->  Event ()
```

L'exemple suivant permet de cerner la différence entre *while* et *when*. L'expression *while (time >* 5)* ne génère aucun événement (*Nothing*) jusqu'à ce que le temps soit supérieur à 5, puis génère souvent des événements *Just ()* ou *Nothing* de manière infinie, tandis que l'expression *when (time >* 5)* génère exactement un événement lorsque le temps dépasse 5.
D'autres extensions de ces fonctions sont utiles dans certaines situations. Ainsi l'opérateur *withElem* agit comme la fonction *zip* standard de Haskell, en couplant la valeur correspondante de type *b* avec la valeur de l'évènement de type *a* quand ce dernier se produit. L'opérateur *withElem_* est une variante de *withElem* qui ignore simplement la première composante dans le résultat.

```
withElem  :: Event a -> [b] -> Event (a,b)
withElem_ :: Event a -> [b] -> Event b
(.|.)     :: Event a -> Event a -> Event a
snapshot  :: Event a -> Behavior b -> Event (a,b)
snapshot_ :: Event a -> Behavior b -> Event b
```

Le combinateur (.|.) agit comme l'opérateur logique OU standard. Il « favorise » le premier évènement (à gauche) lorsque les deux se produisent simultanément. La fonction *snapshot* permet de coupler les résultats de ses deux arguments à chaque occurrence d'évènement de son premier argument. L'opérateur *snapshot_* en est une variante qui ignore la première composante du résulat.
Les opérateurs *step* et *stepAccum* sont deux variantes de *switch*.

```
step :: a -> Event a -> Behavior a
a `step` e = constB a `switch` e =>> constB

stepAccum :: a -> Event (a -> a) -> Behavior a
a `stepAccum` e = b
         where b = a `step` (e `snapshot` b =>> uncurry (\$))
```

L'idée générale dans la signification de *step* et *stepAccum* est que *a ‘step‘ e* commence comme *a* et passe aux valeurs successives associées aux évènements dans *e* ; tandis que dans *stepAccum*, la fonction associée à chaque évènement est appliquée à la dernière valeur du comportement, pour produire un nouveau comportement.

L'ensemble de ces opérateurs constitue le langage FAL pour l'animation réactive des objets 2D. L'utilisateur n'a pas besoin de savoir comment les combinateurs ont été implémentés. Il se contente juste de comprendre comment chacun de ces opérateurs fonctionne, afin de les utiliser à sa convenance. Par exemple, le comportement :

```
counter = 0 ‘stepAccum‘ lbp ->> (+1)
```

est, comme son nom l'indique, un compteur qui fonctionne comme suit : chaque fois qu'on clique sur le bouton gauche de la souris, la valeur du compteur *counter* s'incrémente de un.

Finalement, pour exécuter et obtenir un rendu appréciable (une vue graphique) de certains comportements, des notions avancées de concurrence et d'entrées/sorties ont été utilisées pour développer un interpréteur (*reactimate*) de données graphiques du langage FAL, dont le premier argument de type *String* est le titre de la fenêtre englobante. La fonction *test* ci-dessous permet par exemple d'exécuter une image de type *Picture*, dans une fenêtre de titre "FAL test".

```
reactimate :: String -> Behavior Graphic -> IO ()

test :: Behavior Picture -> IO ()
test beh = reactimate "FAL Test" (lift1 picToGraphic beh)
```

Annexe C

Un environnement d'édition structurée en ligne de commande

Nous présentons dans cette annexe un environnement d'édition en ligne de commande (batch) pour les données structurées représentées par une structure de Zipper. Nous avons implémenté cet environnement dans le but de tester et valider notre DSL, formé de la bibliothèque de combinateurs d'éditeurs décrits dans le chapitre 3. Cette interface utilisateur repose sur la monade des entrées-sorties (`IO monad`) passée comme paramètre effectif du type de base d'un éditeur.

C.1 Édition simple d'un texte (chaîne de caractères)

Un texte est simplement une chaîne de caractères que l'on peut représenter par la grammaire concrète suivante

```
End : Text -> ()
Add : Text -> a Text
```

dans laquelle `Text` représente l'unique catégorie syntaxique (symbole non terminal) et `a` une variable représentant l'ensemble des caractères éditées : `a  -> 'a'|'b'|....`
La traduction en Haskell est donnée par :

```
data Symb = Text deriving Show
data Prod = End | Add Char deriving Show
textNode = MkN Text Hole []

textGram :: Grammar Symb Prod a
textGram  End      = (textNode,[],[])
textGram (Add x) = (textNode,[textNode],[])
```

Les modèles d'arbres (templates) associés à ces productions et qui permettent d'insérer un caractère `c` (`add c`) ou de terminer l'édition (`end`) sont donnés par :

```
-- End : Text -> ()
fin :: Tree (Node Symb Prod)
fin = MkT (MkN Text Leaf []) []
-- Add : Text -> a Text
add :: Char -> Tree (Node Symb Prod)
add x = MkT (MkN Text (Internal (Add x)) [])
            [MkT (MkN Text Hole []) []]
```

Cette insertion comme décrite dans le chapitre 3 est destructive puisqu'elle détruit le sous-arbre associé (i.e. le reste de la chaîne). Par ailleurs, la production `Add` admet le symbole grammatical `Text` en partie droite ; on peut alors lui associer un modèle de couche (template layer) pour permettre l'insertion au milieu du texte sans détruire le reste de la chaîne.

```
 -- template layer associé a la production Add
layer_add :: Char -> Layer (Node Symb Prod) (Tree (Node Symb Prod))
layer_add x = (MkN Text (Internal (Add x)) [],[],[])
```

Nous avons mentionné dans le même chapitre 3 que les commandes d'édition sont définies dans un format propriétaire de telle sorte que les actions utilisateur soient traduites (par la fonction `str2cmd`) dans ce format pour effectuer de manière uniforme les opérations d'édition. Nous définissons ce format par le type de données `Command`. Les actions utilisateur dans un environnement ligne de commande sont représentées simplement par des chaînes spécifiques que l'utilisateur saisit au prompt de l'éditeur et qui sont capturées grâce à la monade des entrées-sorties par la fonction `getCmd`.

```
-------------------------------------------------------------------------
-- Les commandes d'édition
data Command = Enter Char | InsertE Char | Insert Char |
               InsertA Char | Drop | Delete | Error | Help | Quit|
               Undo | None | Redo | Lft | Rgt deriving (Eq,Show)

-- transforme une chaîne de caractères en une commande
str2cmd :: String -> Command
str2cmd ('E':'n':'t':'e':'r':' ':'\'':c:'\'':[]) = Enter c
str2cmd ('I':'n':'s':'e':'r':'t':'E':' ':'\'':c:'\'':[]) = InsertE c
str2cmd ('I':'n':'s':'e':'r':'t':' ':'\'':c:'\'':[]) = Insert c
str2cmd ('I':'n':'s':'e':'r':'t':'A':' ':'\'':c:'\'':[]) = InsertA c
str2cmd "Undo" = Undo
str2cmd "Redo" = Redo
str2cmd "Drop" = Drop
str2cmd "Delete" = Delete
str2cmd "Quit" = Quit
str2cmd "Help" = Help
str2cmd "Left" = Lft
str2cmd "Right" = Rgt
```

```
str2cmd _ = Error -- toute autre chaine est une commande inconnue

-- convertit la chaîne saisie en une commande
getCmd :: IO Command
getCmd = do x <- getLine
            return (str2cmd x)
-------------------------------------------------------------------------
```

Pour autoriser l'annulation ou la reconstitution des actions effectuées ou annulées, il est nécessaire d'implémenter un presse-papiers. On va considérer dans un premier temps que le presse-papiers est simplement un couple formé de la commande saisie, et d'une structure de données de type `Path` (ou `Tree`) contenant les changements apportés en vue d'une éventuelle application des commandes `Undo` ou `Redo`. Nous décrivons le presse-papiers par le type synonyme `Clipboard`. Cependant, un *clipboard* sera en général implémenté comme un éditeur comme on va le voir dans la deuxième section de cette annexe.

```
type Clipboard =  (Command, Path (Node Symb Prod))
```

Une fonction `clipboard :: Command -> ZipperA Symb Prod -> Clipboard` permet de construire le presse-papiers pour chaque commande de telle sorte que si `ab|cd... :: ZipperA Symb Prod` est la chaine courante, et si une action (destructive) `Enter 'x'` est appliquée sur celle-ci, le presse-papiers se construit de la manière suivante :

```
clipboard (Enter 'x') ab|cd... =
           (Enter 'x',[(MkN Text (Internal (Add 'd')) [],[],[]),
                       (MkN Text (Internal (Add 'c')) [],[],[])])
```

Le résultat de l'exécution de cette commande est la chaîne `abx|...` et, on garde en même temps dans le presse-papiers les caractères `c` et `d` supprimés (le sous-arbre associé au Zipper). Ce qui permettra de revenir à la chaîne initiale si on applique une commande `Undo`. De la même manière, le presse-papiers se construit pour les autres commandes de telle sorte qu'on a par exemple les configurations suivantes :

```
-------------------------------------------------------------------------
clipboard (Insert 'x') ab|cd... = (Insert 'x',[])
clipboard Drop ab|cd...  =
               (Drop,[(MkN Text (Internal (Add 'c')) [],[],[])])
clipboard Drop abcd|...= (None,[])
clipboard Delete ab|cd... =
                  (Delete,[(MkN (Internal (Add 'b')) [],[],[])])
clipboard Delete |abcd... = (None,[])
  ...

clipboard_ :: Command -> Clipboard -> Clipboard
clipboard_ Undo (c,_) = (c,[])
-------------------------------------------------------------------------
```

La fonction `clipboard_` est spécialement définie pour la commande `Undo` puisque cette dernière s'applique directement sur le presse-papiers. Cette fonction est nécessaire pour exécuter les commandes `Redo` afin de reconstituer les actions annulées.

Les fonctions `undo` et `redo` invoquées respectivement à chaque exécution de la commande `Undo` ou `Redo` sont définies pour chaque commande. Elles utilisent le presse-papiers et le texte courant (celui qui vient d'être modifié) passés en arguments, et retournent le texte précédent. Leurs signatures sont données par :

```
undo, redo :: Clipboard -> ZipperA Symb Prod -> ZipperA Symb Prod
```

Les fonctions `clipboard` et `clipboard_` ne permettent d'annuler ou de refaire qu'une seule fois l'opération précédente. Pour annuler ou refaire de manière exhaustive toutes les opérations effectuées, nous créons un couple de deux piles de presse-papiers ; l'une (à droite) pour les opérations annulables avec `Undo` et l'autre (à gauche) pour les opérations qu'on peut reconstituer avec `Redo`. Ce que nous représentons par le type `Buffer = ([Clipboard],[Clipboard])`. La nouvelle fonction généralisant la construction du presse-papiers est donnée par :

```
-------------------------------------------------------------------------
-- Construction du presse-papiers pour plusieurs undo et redo
clipboard_n :: Command -> ZipperA Symb Prod -> Buffer -> Buffer
clipboard_n c y (rs,us) = case c of
        Undo -> if null us then (rs,[])
            else ((cmd,[]):rs,tail us) where cmd = fst (head us)
        Redo -> if null rs then ([], us)
            else (tail rs, (clipboard cmd y):us)
                                    where cmd = fst (head rs)
        _ -> (rs,(clipboard c y):us)
-------------------------------------------------------------------------
```

Finalement, on définit la fonction `textEditor` ci-après pour un éditeur de texte. Son état `StateEd` est formé d'un presse-papiers et d'un Zipper. On remarque en particulier que l'interface utilisateur est obtenue en utilisant la monade `IO`.

Tableau C.1 Un simple éditeur simple de texte

```
textEditor :: Editor StateEd IO Command ()
textEditor _  = do
  (cur_cb,cur_z) <- readST
  lift (afficher cur_z)
  cmd <- lift getCmd
  case cmd of
   Enter c -> do new_cb <- lift (getBuffer
                                (clipboard_n cmd cur_z cur_cb))
                 updateST' (update_e3 new_cb (car c))
                 textEditor None
   Insert c-> do new_cb <- lift (getBuffer
                                (clipboard_n cmd cur_z cur_cb))
                 updateST' (update_e3 new_cb (insert c))
                 textEditor None
   Undo -> do
       ((_,us),_) <- readST
       if null us then textEditor None
                  else do ((_,(cur_cmd,cur_path):_),_) <- readST
       new_cb <- lift (getBuffer (clipboard_n cmd cur_z cur_cb))
       updateST' (update_e3 new_cb (undo (cur_cmd,cur_path)))
              textEditor None
   Redo -> do
        ((rs,_),_) <- readST
        if null rs then textEditor None
                else do (((cur_cmd,cur_path):_,_),_) <- readST
        new_cb <- lift (getBuffer (clipboard_n cmd cur_z cur_cb))
        updateST' (update_e3 new_cb (redo (cur_cmd,cur_path)))  -
        textEditor None
   ...
   Error     -> do lift (putStrLn "Commande inconnue ! ")
                   lift (putStrLn "Tapez  Help")
                   textEditor None
   Help      -> do
      lift (putStrLn "Enter 'c': Insertion de c avec destruction")
      lift (putStrLn "InsertE 'c'  : Insertion de c,
                              puis déplacement du curseur en fin")
      ...
   Quit      -> do lift (putStr (show (fini (end cur_z))))

type StateEd = (Buffer, ZipperA Symb Prod)

runTextEditor = runST (textEditor None)
                      (([],[]), ([] :> MkT (MkN Text Hole []) []))
```

Une session d'édition utilise l'interpréteur Haskell Hugs ou GHC sous les plate-formes Windows ou Linux. Le prompt de l'éditeur est représenté par >> et celui de l'interpréteur Haskell par > comme indiqué sur la figure C.1. On exécute l'éditeur par la fonction `runTextEditor`. La commande d'édition `Help` permet d'obtenir les informations nécessaires pour l'utilisation des autres commandes.

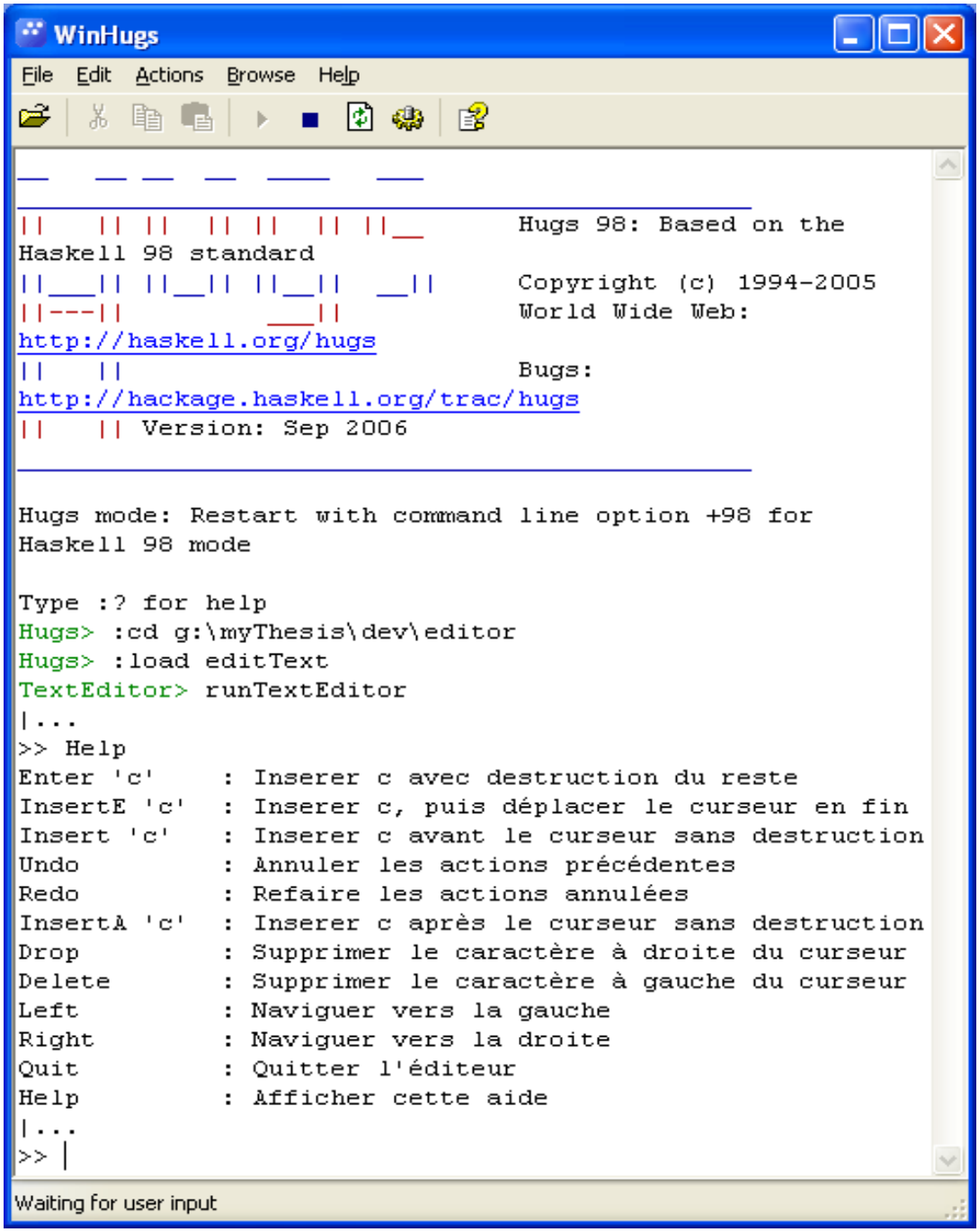

Figure C.1 – Environnement d'édition d'un texte

La figure C.2 représente une session d'édition concrète d'un texte.

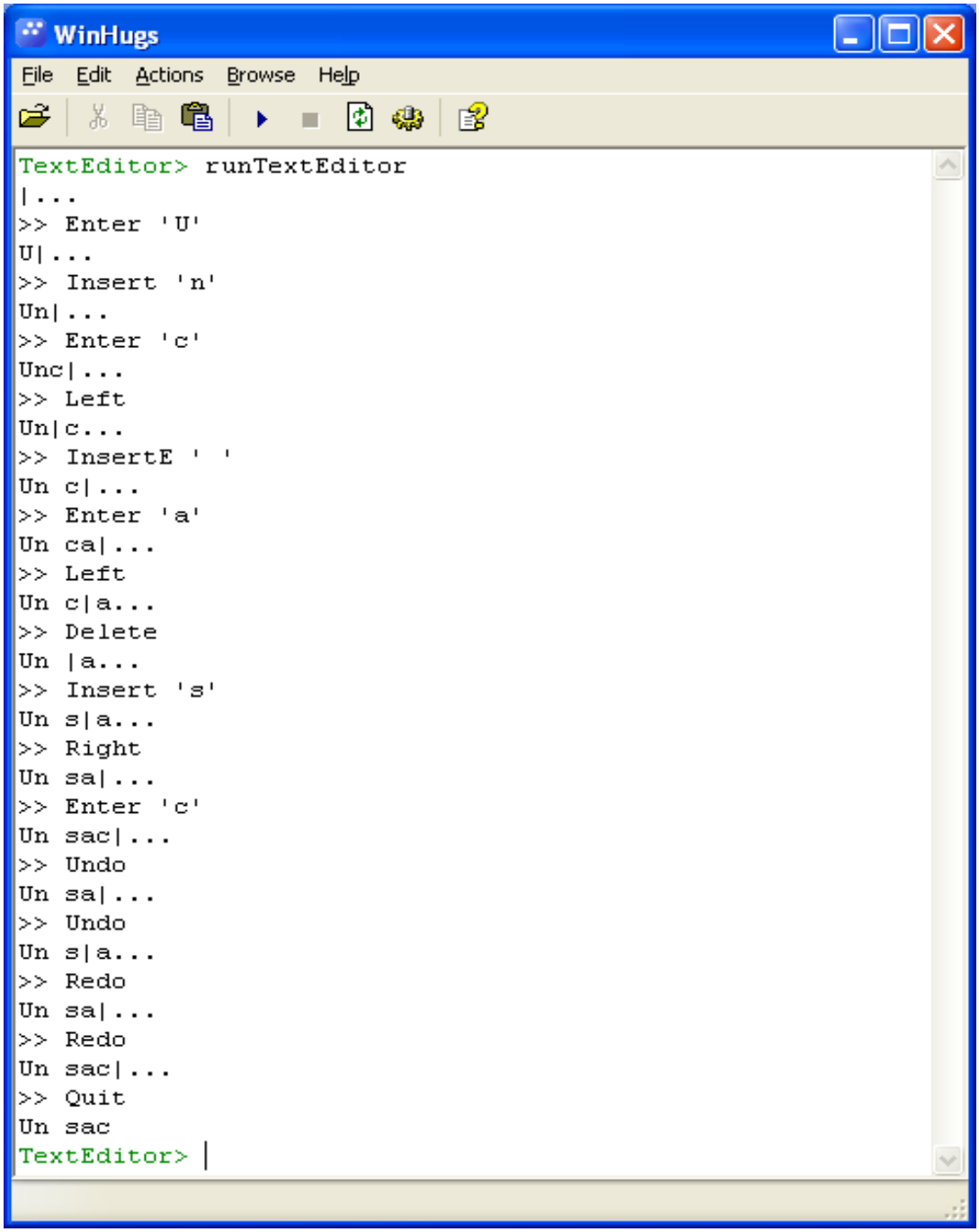

FIGURE C.2 – Une session d'édition d'une chaîne de caractères

C.2 Édition d'une donnée structurée

De manière générale, on voit le tampon d'un éditeur structuré comme un autre éditeur et l'éditeur résultat est obtenu en combinant les deux éditeurs de base comme décrit au paragraphe 3.6.1 page 102. L'état de l'éditeur de presse-papiers est un couple de type `Buffer s p` et celui de l'éditeur de la donnée structurée un Zipper de type `ZipperA s p`. Les commandes internes (`Todo` et `Done`) transparentes à l'utilisateur sont rassemblées dans

la structure de données `ActionE s p` et, afin de lever toute ambiguïté, nous avons scinder le format des commandes externes en deux catégories : `Command` pour les commandes `Redo` et `Undo` agissant directement sur le presse-papiers et, `Cmd` pour les autres commandes (`Insert`, `Delete`, etc.) envoyées sur l'éditeur de base.

```
data Cmd =  Enter Prod | Insert Prod Int | Drop | Delete | Help
          | Error | Lft | Rgt | Up | Down | None | Quit
data Command = Undo | Redo
```

L'éditeur de presse-papiers est alors défini par

```
-------------------------------------------------------------------------
type EditorCb s p = Editor (Buffer s p) IO
                 (Either Command (ActionE s p)) (Either () (ActionE s p))

editorCb :: EditorCb s p
editorCb act = case act of
        (Left Undo) -> do (_,us) <- readST
                          updateST' (updateBuf Udo)
                            if null us then return []
                               else return [Right (UndoA (head us))]
        (Left Redo) -> do (rs,_) <- readST
                          updateST' (updateBuf Rdo)
                            if null rs then return []
                              else return [Right (RedoA (head rs))]
        (Right act) -> do updateST' (updateBuf act)
                          return []

-- updating a buffer
updateBuf :: ActionE s p -> Buffer s p -> Buffer s p
updateBuf action (rs,us) = case action of
        Udo         -> if null us then (rs,[])
                       else (a:rs, tail us) where a = head us
        Rdo         -> if null rs then ([],us)
                      else (tail rs, a:us) where a = head rs
        _          -> (rs, action:us)
-------------------------------------------------------------------------
```

Nous définissons de manière similaire l'éditeur pour une donnée structurée représentée par un Zipper par le listing suivant :

```
-------------------------------------------------------------------------
type EditorZp s p = Editor (ZipperA s p) IO
                     (Either (ActionE s p) Cmd) (Either (ActionE s p) ())

gram_editor :: (Show s, Eq s) => Grammar s Prod () -> EditorZp s Prod
gram_editor g action  = case action of
```

```
        (Left (UndoA act)) -> do updateST' (runUndoA (UndoA act))
                               return [] --[Left act]
        (Left (RedoA act)) -> do updateST' (runRedoA g (RedoA act))
                               return []
        (Right Help)       -> do return []
        (Right Quit)       -> do return []
        (Right Error)      -> do return []
        (Right (Insert p i)) -> do updateST' (insert_i g p i)
                                   return [Left (Ins p i)]
        (Right Delete)     -> do (cxt :> _) <- readST
                                 if null cxt then return []
                                 else do updateST' (delete g)
                                         return [Left (Del (head cxt))]
        (Right Drop)       -> do (_ :> t) <- readST
                                 updateST' (kill g)
                                 return [Left (Kill t)]
        (Right Lft)        -> do updateST' left
                                 return [Left LftA]
        (Right Rgt)        -> do updateST' right
                                 return [Left RgtA]
        (Right Up)         -> do updateST' up
                                 return [Left UpA]
        (Right Down)       -> do updateST' down
                                  return [Left DownA]
-----------------------------------------------------------------------
```

L'éditeur résultat est donné par le code du tableau C.2 dans lequel on remarque qu'à l'avant-dernière ligne de la fonction `grammarEditor`, nous avons utilisé le combinateur `<%>` pour combiner (dans l'ordre) l'éditeur du presse-papiers et l'éditeur d'une donnée structurée quelconque.

Tableau C.2 Un éditeur structuré avec presse-papiers

```
type EditorRt s p = Editor (Buffer s p, ZipperA s p) IO
                           (Either Command Cmd) (Either () ())

grammarEditor :: (Show s, Eq s) => Grammar s Prod () ->
                                                   EditorRt s Prod
grammarEditor  g  _ =
  do (b,z) <- readST
     lift (putStrLn "")
     lift (putStrLn "------Etats courants---------")
     lift (putStrLn "Représentation interne d'un éditeur : Zipper")
     lift (display g z)
     lift (putStrLn "-------------")
     lift (putStrLn ("Buffer Redo = " ++ show (fst b)))
     lift (putStrLn ("Buffer Undo = " ++ show (snd b)))
     lift (putStrLn "--------------------------------------")
     lift (putStr ">>")
     cmd <- lift getCommand
     case cmd of
        Right Quit -> do lift (display g z)
                         return []
        Right Help ->
          do lift (putStrLn "---Signification des commandes-----")
             lift (putStrLn "Enter p : insert le modèle associé à
                                                   la production p")
             ...
             lift (putStrLn "Help       : affiche cette aide")
             lift (putStrLn "---------------------------------")
             grammarEditor g cmd
        Right Error -> do lift (putStrLn "Erreur. Tapez Help")
                          grammarEditor g cmd
        _ -> do (cbEditor <%> (gram_editor g)) cmd
                grammarEditor g cmd

runEditGram :: (Show s, Eq s) => Grammar s Prod () -> Prod -> IO ()
runEditGram g p = mapMonad nothing (runST (e_gram g (Right Help))
                                                        (bf0, s0))
              where s0 = ([] :> MkT (MkN (lhsSymb g p) Hole []) [])
                    bf0 = ([],[]) -- l'état initial du tampon
                    nothing _ = ()
```

À titre illustratif, si on considère une grammaire abstraite dont les productions sont définies par

```
P1 : A -> B C D
P2 : A -> A D
P3 : B -> B A
P4 : C -> C D C
P5 : C -> ()
P6 : D -> ()
P7 : A -> C
```

alors, la traduction en Haskell est donnée par :

```
-------------------------------------------------------------------------
data SymbG = A|B|C|D deriving (Show, Eq)

data Prod = AX | P1 | P2 | P3 | P4 | P5 | P6 | P7 deriving (Show,Eq)

nodeA = MkN A Hole []
nodeB = MkN B Hole []
nodeC = MkN C Hole []
nodeD = MkN D Hole []

gram :: Grammar SymbG Prod ()
gram AX = (nodeA,[nodeA],[])
gram P1 = (nodeA,[nodeB,nodeC,nodeD],[])
gram P2 = (nodeA,[nodeA,nodeD],[])
gram P3 = (nodeB,[nodeB,nodeA],[])
gram P4 = (nodeC,[nodeC,nodeD,nodeC],[])
gram P5 = (nodeC,[],[])
gram P6 = (nodeD,[],[])
gram P7 = (nodeA,[nodeC],[]))

display :: (Show s,Show p) => Grammar s p () -> ZipperA s p -> IO ()
-------------------------------------------------------------------------
```

La production `AX` permet de définir l'axiome de la grammaire. Nous avons défini en plus une fonction `display`, nécessaire pour afficher l'état interne de l'éditeur résultat. Cet état est normalement formé de la représentation sous forme de Zipper attribué de la structure éditée, et des différents tampons d'édition. Mais pour plus de lisibilité, nous avons choisi de n'afficher que la structure éditée comme indiqué sur la figure C.3.

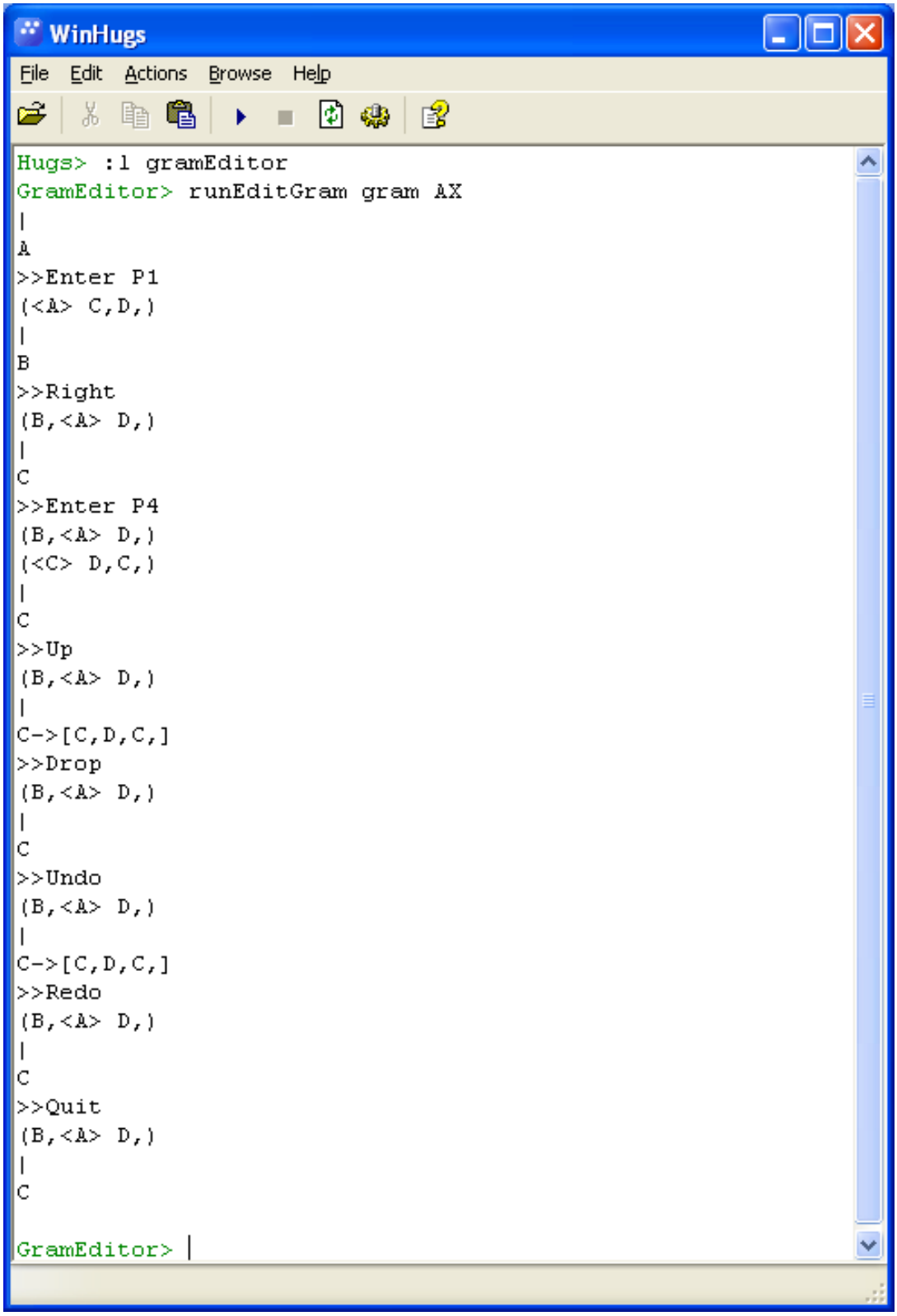

FIGURE C.3 – Une session d'édition structurée