UNIVERSITE D'ORLEANS

THÈSE

PRÉSENTÉE À L'UNIVERSITÉ D'ORLÉANS

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ D'ORLÉANS

dans la discipline

INFORMATIQUE

par

Etienne DURIS

Sujet de la thèse :

CONTRIBUTION AUX RELATIONS ENTRE LES GRAMMAIRES ATTRIBUÉES ET LA PROGRAMMATION FONCTIONNELLE

Présentée et soutenue publiquement à Orléans le 5 Octobre 1998

devant le jury composé de :

Mr	**Martin**	**JOURDAN**	Président
Mme	**Françoise**	**BELLEGARDE**	Rapporteurs
Mr	**Bernard**	**LORHO**	
MM	**Charles**	**CONSEL**	Examinateurs
	Gaétan	**HAINS**	
	Didier	**PARIGOT**	

UNIVERSITE D'ORLEANS

THÈSE

PRÉSENTÉE À L'UNIVERSITÉ D'ORLÉANS

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ D'ORLÉANS

dans la discipline

INFORMATIQUE

par

Etienne DURIS

Sujet de la thèse :

CONTRIBUTION AUX RELATIONS ENTRE LES GRAMMAIRES ATTRIBUÉES ET LA PROGRAMMATION FONCTIONNELLE

Présentée et soutenue publiquement à Orléans le 5 Octobre 1998

devant le jury composé de :

Mr	**Martin**	**JOURDAN**	Président
Mme	**Françoise**	**BELLEGARDE**	Rapporteurs
Mr	**Bernard**	**LORHO**	
MM	**Charles**	**CONSEL**	Examinateurs
	Gaétan	**HAINS**	
	Didier	**PARIGOT**	

Remerciements

C'est Martin Jourdan qui m'a ouvert les portes de la recherche. Je le remercie de la précision de ses jugements scientifiques et de son intérêt constant pour mon travail malgré ses différentes activités. Je le remercie également d'avoir accepté d'être mon directeur de thèse et de présider ce jury.

En me faisant l'honneur d'être rapporteurs de cette thèse, Françoise Bellegarde et Bernard Lorho ont accepté de juger un travail qui implique leurs deux domaines d'excellence respectifs et leur collaboration a permis de recouvrir l'ensemble de mes recherches. Je leur en suis profondément reconnaissant. Je tiens à remercier Françoise Bellegarde pour l'intérêt qu'elle a porté très tôt à mes travaux, pour ses remarques pertinentes et ses critiques constructives. Je remercie également Bernard Lorho pour avoir trouvé le temps de me faire profiter de son expertise des grammaires attribuées et de ses précieux conseils.

Je veux aussi remercier Charles Consel et Gaétan Hains pour leurs remarques, leurs conseils et les suggestions dont ils m'ont fait part. Je suis heureux qu'ils aient accepté de faire partie de ce jury.

Que Didier Parigot trouve ici mes plus chaleureux remerciements. Il a su orienter mes travaux, avec son énergie farouche et son sens intuitif de la recherche. Cette thèse lui doit beaucoup, tant du point de vue scientifique que du point de vue humain. Les grammaires attribuées sont "en lui" !

Je tiens aussi à dire un grand merci à Gilles Roussel. Son point de vue critique mais enthousiaste sur mes travaux et ses conseils éclairés ont largement contribué à cette thèse. Sa confiance, sa sympathie et son aide m'ont été réconfortants à de nombreuses occasions.

Je remercie également Loïc Correnson qui n'a pas hésité à m'expliquer ses excellentes idées, dont certaines ont inspiré mes travaux. Ses qualités scientifiques me laissent rêveur ...

Ces remerciements s'adressent également à ceux qui ont contribué à un cadre de travail fertile et à un équilibre moral indispensable ... Aux chercheurs, enseignants, thésards d'ici ou d'ailleurs, Gérard Ferrand, John Boyland, Alberto Pardo, Guy Tremblay, Marc Zipstein, Rémi Forax, Laurent Granvilliers, Sylvain Lelait, Bruno Marmol et Chan Ung pour leurs conseils et leurs encouragements. À mes amis de toujours, Manu et Denis, irremplaçables et toujours prêts à parler d'autre chose ! À Cécile qui a dû supporter mes doutes métaphysiques et mes apnées intellectuelles ... Et à ma famille bien sûr, pour m'avoir soutenu sans faillir depuis si longtemps et, en particulier, tout au long de cette thèse.

Table des matières

Chapitre 1

Introduction

1.1 Le contexte

Le développement fulgurant de l'utilisation de moyens informatiques, dans pratiquement tous les domaines d'activités, nécessite des méthodes de développement de logiciel de plus en plus sures et de moins en moins coûteuses. Par ailleurs, l'évolution des infrastructures mises à la disposition des programmeurs de logiciels suscite également de nouvelles techniques ou de nouveaux paradigmes de programmation. L'amélioration des performances des matériels ou l'essor des moyens de télécommunications constituent des catalyseurs qui dopent la recherche de solutions satisfaisantes aux nombreux problèmes que ces nouveaux contextes soulèvent.

En à peine plus de 30 ans, les balbutiements de la programmation en assembleur ont été supplantés par des méthodologies de développement de logiciel de haut niveau, permettant d'instaurer un suivi allant de la spécification des applications jusqu'à leur test ou leur mise en œuvre. Cependant, certaines idées, pouvant faire figure de *préhistoriques* dans le contexte d'une science jeune comme l'informatique, restent très actuelles par leur pertinence. Ainsi, l'idée de Burstall et Darlington en 1977 était d'écrire un programme clair et simple, quitte à ce qu'il soit inefficace, et de le transformer automatiquement en un programme efficace, quitte à ce qu'il soit moins clair [BD77]. Cette idée, qui peut paraître naïve, a en fait suscité un nombre important de travaux de recherches dans différents domaines de la programmation.

Des qualités recherchées

Les langages de programmation doivent permettre de décrire des spécifications claires. Une propriété fondamentale, recherchée par la plupart des langages de programmation, est la modularité. La possibilité de découper une application en petits modules, traitant chacun d'une tâche bien précise, facilite à la fois la lisibilité et la vérification de ces modules mais aussi celles des programmes qui les utilisent. Par ailleurs, si un de ces modules est utilisé plusieurs fois dans une même application, cela autorise un gain précieux, en terme de développement, de test, de maintenance et d'évolution de l'ensemble du logiciel : c'est la *réutilisabilité*.

Une autre qualité d'importance est l'aspect déclaratif d'un langage de programmation. Il permet de décrire dans un programme les calculs qui seront à effectuer, sans se lier à la façon dont ils seront mis en œuvre lors de l'exécution du programme. L'aspect déclaratif d'un langage de programmation autorise une dissociation entre la spécification d'un problème qui doit caractériser naturellement le résultat recherché et sa résolution qui est dépendante de l'implantation et qui peut être indifférente au programmeur. Ceci autorise la portabilité du programme, un atout indéniable dans le cadre de l'évolution rapide des matériels et dans un contexte où les applications transitent sur le réseau entre des machines différentes.

Enfin, et pour ne citer que ces trois qualités, les logiciels développés doivent être efficaces. En effet, malgré les progrès des machines, en terme de de rapidité ou de capacité de stockage, la quantité d'information à traiter est telle que l'occupation mémoire et le temps d'exécution d'un programme sont des problèmes bien réels, qu'il convient de ne pas négliger. Malheureusement, les propriétés agréables des logiciels en matière de spécification, c'est-à-dire leur modularité et leur aspect déclaratif, sont souvent antinomiques avec l'efficacité de leur mise en œuvre. La distance entre la spécification et l'implantation d'un programme, instaurée par le niveau d'abstraction élevé des langages de programmation, nécessite des phases d'analyse et de transformation lors de la traduction du langage de spécification en langage machine exécutable.

Différents paradigmes de programmation

Le cadre général dans lequel s'inscrit cette thèse est l'étude de différents paradigmes de programmation, tant sur le plan des concepts que sur celui des implantations, en vue de faciliter leur comparaison pour en faire émerger des fertilisations croisées. En effet, les méthodes développées pour atteindre les objectifs de qualité précédemment mentionnés peuvent apparaître comme très différentes selon qu'elles sont étudiées dans les langages de programmation fonctionnels, logiques, à objets, etc. Cependant, les différences de formalismes et le caractère souvent hermétique des différentes communautés cachent parfois des ressemblances, des relations, des complémentarités dans les approches adoptées pour traiter des problèmes similaires. Ces complémentarités peuvent pourtant être exploitées, d'une part, pour susciter des améliorations des méthodes utilisées dans chaque domaine et, d'autre part, pour généraliser ces méthodes et les abstraire de leur attachement à un paradigme spécifique. L'objet de cette thèse, dans le contexte particulier des transformations de programmes, est précisément de comparer des techniques et des formalismes *a priori* différents et d'utiliser les résultats de cette comparaison afin de développer une méthode surpassant la puissance des différentes approches précédemment utilisées. Au delà des résultats spécifiques au cadre de mon étude, qui seront présentés dans ce mémoire, il est donc important de retenir que la comparaison de formalismes et de méthodes issus de différentes communautés est une tâche délicate et ardue, mais pouvant aboutir à des améliorations qui dépassent la simple fertilisation croisée en faisant émerger de nouvelles solutions généralisant les approches initiales.

1.2 Le cadre de comparaison

Comme je l'ai déjà mentionné, les transformations de programmes permettent de combler en partie le fossé entre le haut niveau d'abstraction nécessaire d'un langage de programmation et le bas niveau d'implantation indispensable à son efficacité. Beaucoup de ces transformations sont tout simplement intégrées dans un compilateur et sont donc dédiées à une implantation donnée d'un langage de programmation donné. D'autres transformations restent au niveau du langage de programmation et sont appelées transformations source à source. Elles tendent à transformer, comme suggéré par Burstall et Darlington, un programme clair et très modulaire en un programme plus efficace, mais toujours décrit dans le même langage : elles produisent un programme ayant la même sémantique, c'est-à-dire produisant les mêmes résultats, mais qui est syntaxiquement plus compliqué et moins lisible, tout en étant plus "rusé" et plus efficace, tel qu'un programmeur expérimenté et averti aurait pu l'écrire directement. L'avantage de telles transformations est de libérer le programmeur de cette programmation périlleuse, en lui autorisant l'écriture de programmes simples, modulaires, faciles à comprendre et à valider et dont l'efficacité ne pâtit pas.

Transformations de programmes fonctionnels

Des programmes modulaires peuvent être écrits en programmation fonctionnelle. La modularité autorisée par les langages de programmation fonctionnels est basée sur la notion de composition de fonctions. Plutôt que de décrire un problème comme un unique programme monolithique ou comme une suite procédurale de modifications d'un ensemble de valeurs, l'idée est de le décomposer en une succession de petites fonctions applicatives s'appelant les unes sur le résultat des autres. De plus, les langages fonctionnels incorporent un aspect déclaratif lié à la récursion. Une définition de fonction par des équations récursives est assez intuitive et elle ressemble au programme fonctionnel qui lui correspond. Par exemple, la définition récursive du calcul de la somme des éléments d'une liste d'entiers peut être décrite dans une syntaxe de langage fonctionnel classique :

$$\begin{array}{l} \texttt{let}\ sum\ x = \texttt{case}\ x\ \texttt{of} \\ \qquad\qquad cons\ head\ tail \rightarrow head + (sum\ tail) \\ \qquad\qquad nil \rightarrow 0 \end{array}$$

Un programme général peut alors faire successivement appel à différentes fonctions de ce genre, en les composant les unes aux autres. Par exemple, si *map_square* est la fonction qui, pour une liste d'entiers donnée, construit la liste des carrés de ces entiers ; si *reverse* est la fonction qui inverse la liste qui lui est passée en paramètre ; alors un programme P, prenant une liste d'entiers en entrée, peut calculer la "somme des carrés des éléments de la liste initiale inversée" simplement en composant les fonctions *sum*, *map_square* et *reverse*. Ce programme sera alors spécifié par $P = (sum \circ map\_square \circ reverse)$, ou bien, pour une liste x donnée, $(P\ x) = (sum\ (map\_square\ (reverse\ x)))$. La figure 1.1 illustre l'effet de ces compositions dans un programme appliqué à une liste particulière.

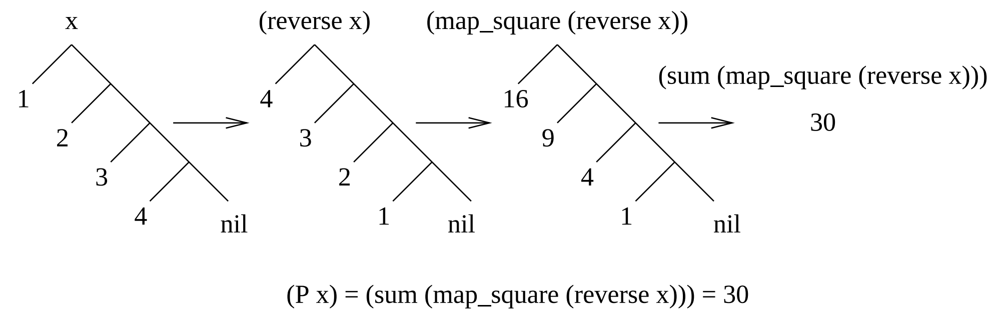

FIG. 1.1: *Un programme composé de plusieurs fonctions*

Même si, dans ce cas précis, un programme calculant le même résultat que P peut être exprimé de manière plus simple, sans avoir besoin de calculer l'inversion de la liste, la spécification du programme P est très claire et très modulaire. Pour la majorité des programmes, la composition reste la seule façon raisonnable de développer correctement une application importante, à l'aide de fonctions simples. Cependant, le prix de cette clarté, obtenue par la composition, se paye lors de l'exécution du programme. En effet, comme l'illustre la figure 1.1, le coût en place mémoire occupée du programme sur cet exemple est dramatique par rapport au coût espéré d'un programme optimal ayant la même sémantique. L'application successive des deux premières fonctions provoque l'allocation de la place nécessaire à la construction de deux listes, de la même taille que la première. Celles-ci ne font pas partie du résultat final. Elles devront donc être détruites et leur emplacement désalloué. Ces structures, qui servent de *colle* entre les différentes fonctions d'une composition [Hug89], sont appelées des structures intermédiaires et sont néfastes du point de vue de l'efficacité d'un programme [Wad88].

Déforestation

Les transformations de programme qui tendent à éliminer les structures intermédiaires apparaissant dans les compositions de fonctions, ou plus exactement à éviter de les construire, sont connues sous le nom de méthodes de déforestation, puisqu'elles éliminent des arbres, ou plus généralement des structures de données (par analogie sylvestre). Ce terme est dû à Wadler, qui s'est intéressé à ce problème à la fin des années 1980 [Wad84, Wad85, Wad88, Wad90] : il faisait alors observer que ces structures intermédiaires sont à la fois la base des programmes fonctionnels — autorisant la composition — et leur fléau — rendant leur exécution inefficace.

Depuis ses travaux et son algorithme de déforestation, de nombreux travaux ont été menés avec le même objectif. Certaines recherches ont étendu ou modifié l'algorithme initial de Wadler [GJ94, Sør94, SGJ94, CK95, Ham96, Sei96, SS97b, SS97a]. D'autres méthodes ont également été étudiées et développées en s'appuyant sur des formalismes algébriques et sur des résultats de la théorie des catégories : ces méthodes plus abstraites et plus formelles, dites *calculationnelles*, utilisent à la fois les schémas de récursion des types de données algébriques et ceux des fonctions pour guider le processus de déforestation [MFP91, GLJ93, SF93, FSZ94, LS95, TM95, HIT96b, OHIT97].

Le chapitre 5 de cette thèse est consacré à une présentation des principales méthodes et formalismes qui sont utilisés dans le cadre de la déforestation des programmes fonctionnels. Ce problème, largement étudié au cours de la dernière décennie et soumis à différentes méthodes, reste un problème difficile à formaliser et à mettre en œuvre, qui n'est pas encore complètement résolu ; outre le recueil et la présentation qui est faite des différentes approches proposées, le travail présenté dans cette thèse apporte des éléments nouveaux à la déforestation des programmes fonctionnels et des solutions à des problèmes qui n'étaient pas résolus jusqu'à présent.

Les grammaires attribuées

Dans l'exemple du programme fonctionnel P utilisé en illustration (figure 1.1), les définitions des fonctions sont dirigées par la structure de la liste. Le schéma de récursion des fonctions est guidé par le constructeur courant de l'argument qui lui est passé en paramètre. Par exemple, le calcul de la somme des éléments d'une liste dans la fonction *sum* est spécifié séparément sur les constructeurs *nil* et *cons*. Ce style de spécification de fonction, *dirigé par la structure des données*, est également caractéristique d'un autre paradigme de programmation : les grammaires attribuées. Comme leur nom l'indique, les grammaires attribuées s'appuient sur la grammaire d'un type de données pour spécifier des calculs à effectuer au niveau de chaque production de cette grammaire (i.e, au niveau des constructeurs du type).

Les grammaires attribuées sont des spécifications déclaratives, dirigées par la structure des données, qui bénéficient d'un solide contexte d'études, d'analyses, d'implantations et de mises en œuvre [Paa95].

Le chapitre 2 présente un rappel des notations, du formalisme et des techniques d'évaluation des grammaires attribuées, mais pour illustrer rapidement mon propos, je donne ici la définition de la grammaire attribuée *sum* calculant la somme des éléments d'une liste d'entiers. Les puristes des grammaires attribuées seront probablement choqués d'une telle définition, mais je rappelle que le sujet de cette thèse est de faire le lien entre deux communautés et la notation adoptée pour les exemples de grammaires attribuées est volontairement orientée vers un style fonctionnel ; les explications et les justifications de cette notation seront présentées au chapitre 2.

$$
\begin{array}{l}
\texttt{aglet}\ \mathit{sum} = \\
\quad \mathit{sum}\ x \rightarrow \\
\quad\quad \mathit{sum.result} = x.s \\
\quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \\
\quad\quad \mathit{cons.s} = \mathit{head} + \mathit{tail.s} \\
\quad \mathit{nil} \rightarrow \\
\quad\quad \mathit{nil.s} = \mathit{0}
\end{array}
$$

Je rappelle ici très brièvement le principe des grammaires attribuées, introduites par Knuth il y a 30 ans [Knu68]. Pour une grammaire indépendante du contexte, permettant de dériver les arbres syntaxiques d'un certain type de données, un ensemble de symboles pouvant contenir des valeurs et appelés *attributs* est associé à chacun des non-terminaux de la grammaire. Certains attributs peuvent véhiculer des informations des feuilles de l'arbre vers sa racine : ils sont appelés attributs synthétisés. Les autres, permettant de transporter l'information en sens inverse, sont appelés attributs hérités. Par ailleurs, à chacune des productions de la grammaire, le programmeur associe un ensemble de règles sémantiques, qui sont des équations orientées permettant de définir les attributs. Pour ce qui est de l'évaluation d'une grammaire attribuée pour un paramètre donné (un "texte" au sens de la grammaire), l'analyse lexico-syntaxique du paramètre produit un arbre syntaxique ; aux différents nœuds de cet arbre, qui correspondent à des instances de non-terminaux, sont attachés les attributs correspondants. Ensuite, un *évaluateur* d'attributs est chargé de calculer, à l'aide des règles sémantiques, l'ordre dans lequel les valeurs des attributs de cet arbre peuvent être calculées.

De manière intuitive, une grammaire attribuée est donc la donnée d'un ensemble de productions, chacune accompagnée d'une ou de plusieurs équations orientées. Les règles sémantiques définissent la façon de calculer, en chaque point de l'arbre, la valeur $var.s$ de l'attribut s porté par une variable var représentant un nœud de l'arbre qui est une instance d'un non-terminal de la grammaire. Les productions sont vues comme des constructeurs d'arbre syntaxique et dénotées comme des *patterns* de type algébrique. La première construction, un peu particulière, représente en fait l'appel de la grammaire attribuée et fait apparaître son argument : elle est appelée le *profil* de la grammaire attribuée. Les autres sont des *patterns* classiques de constructeurs de liste. L'attribut *result* représente le résultat de la grammaire et la règle sémantique qui le définit indique que sa valeur est celle de l'attribut s sur la liste argument x. La figure 1.2 représente chacun de ces constructeurs, y compris le premier, comme des

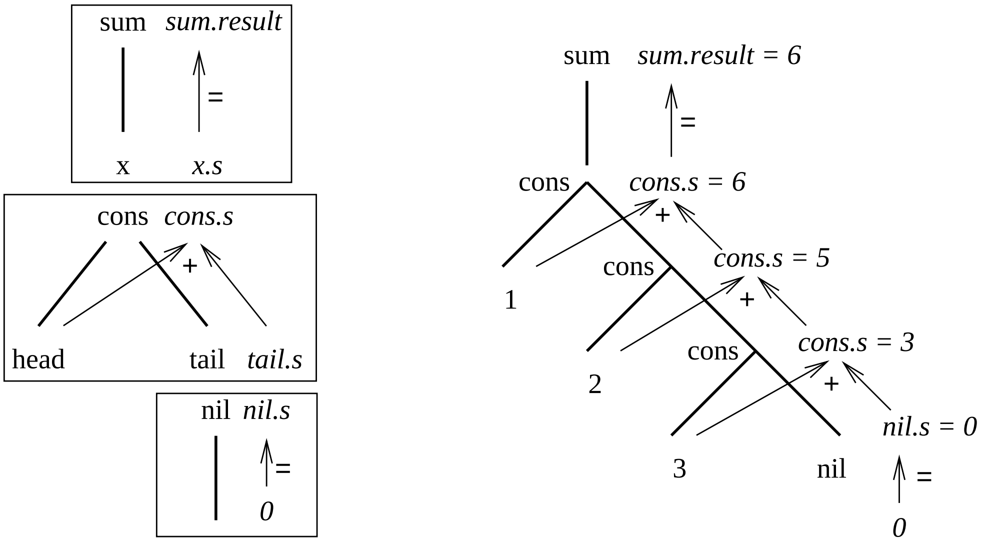

FIG. 1.2: *Illustration de la grammaire attribuée sum, sur ses productions et sur un exemple de liste*

productions d'une grammaire indépendante du contexte ; elle visualise les règles sémantiques définissant les attributs par des flèches. À côté, un arbre syntaxique correspondant à une liste particulière est présenté, décoré par les valeurs des attributs correspondant à chaque nœud.

Les grammaires attribuées sont des spécifications très déclaratives, au sens ou le programmeur ne se soucie pas de l'ordre dans lequel les attributs doivent être calculés : cette tâche est déléguée à un *générateur d'évaluateur*, qui effectue des analyses statiques pour déterminer l'ordre d'évaluation à partir de la forme des règles sémantiques. De nombreux travaux ont été menés dans ce sens [DJL88], mais intuitivement, sur l'ensemble des productions d'une grammaire indépendante du contexte, les règles sémantiques constituent un système d'équations orientées qu'il faut résoudre. C'est la résolution de ce système d'équations qui permet de fournir un évaluateur d'une grammaire attribuée donnée, quel que soit l'argument qui lui est fourni. Le programmeur ne se préoccupe pas de cet ordre d'évaluation lorsqu'il spécifie la grammaire attribuée. Il spécifie *ce* qui doit être calculé et non *comment* cela doit l'être. De plus, les règles sémantiques sont locales aux productions de la grammaire. Cela confère aux grammaires attribuées un caractère de localité, dirigé par la structure des données, qui est particulièrement intéressant du point de vue de la modularité structurelle de la spécification et du point de vue des transformations de programmes que cela autorise. Contrairement aux programmes fonctionnels, les grammaires attribuées n'explicitent pas les appels récursifs mais sont plutôt basées sur la notion de dépendances entre attributs.

Finalement, il est important de signaler que l'ensemble des analyses et des transformations qui ont été développées pour les grammaires attribuées sont indépendantes de quelque modèle d'exécution que ce soit. En fait, un évaluateur d'une grammaire attribuée précise simplement dans quel ordre les choses devront être évaluées, mais il ne dit rien sur ce qui mettra en œuvre cette exécution. Cette propriété confère aux grammaires attribuées un niveau d'abstraction particulièrement intéressant pour l'étude que je présente dans cette thèse.

La composition descriptionnelle

Afin que les grammaires attribuées répondent aux exigences de qualité des langages de programmation, des recherches ont tenté de leur apporter de la modularité. En effet, en dehors de la programmation structurée (par les productions de la grammaire sous-jacente) qu'autorisent les grammaires attribuées classiques, les applications écrites dans ce formalisme étaient plutôt monolithiques. L'exemple souvent exhibé est celui de la grammaire attribuée qui décrit le *front-end* du compilateur Ada, qui correspond à un livre de 500 pages ! Pour cette raison, les grammaires attribuées dites *grammaires couplées par attributs* ont été introduites par Ganzinger et Giegerich [GG84]. Sans remettre en question les bases du formalisme, cette façon de voir les grammaires attribuées permet de les considérer comme des fonctions qui prennent un arbre en entrée et en retournent un autre en sortie. Ceci permet alors d'envisager un programme comme une composition successive de plusieurs grammaires attribuées.

Malheureusement, tout comme en programmation fonctionnelle, ce caractère modulaire introduit des problèmes d'efficacité. Les arbres intermédiaires, permettant de relier les grammaires attribuées entre elles, ne sont rien d'autre que des structures intermédiaires qui ne font pas partie du résultat final et qui nécessitent une forme de déforestation tout comme celle précédemment évoquée dans le cadre de la programmation fonctionnelle. Pour répondre à ce problème d'efficacité, Ganzinger et Giegerich ont introduit une construction qui permet de transformer une suite de grammaires attribuées en une unique grammaire attribuée, qui a la même sémantique que la suite de grammaires attribuées de départ et qui ne construit pas les structures intermédaires [GG84]. Cette construction, appelée la *composition descriptionnelle*, a été plus récemment étudiée et développée par Roussel [Rou94] sous le nom de *méta-composition*.

La comparaison entre la composition descriptionnelle et les différentes méthodes de déforestations fonctionnelles constitue le cadre dans lequel je me propose d'apporter une contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle. À cette fin, différentes études, comparaisons, extensions et développements me seront nécessaires.

1.3 Présentation et organisation des travaux

L'objectif des travaux présentés dans ce mémoire est donc d'apporter un élément de réponse au problème du dilemme entre la nécessité de modularité requise pour le développement des logiciels — génériques, adaptifs, réutilisables — et l'obligation de performance dans leur mise en œuvre — occupation mémoire, temps d'exécution.

Dans le contexte des transformations de programmes, les grammaires attribuées présentent des atouts indéniables par leur caractère déclaratif, dirigé par la structure des données. Cependant, leur utilisation principale pour la production de compilateurs les a longtemps confiné dans un domaine ne leur permettant pas de montrer la généralité de leur puissance. Par ailleurs, leur comparaison avec la programmation fonctionnelle et les opportunités de fertilisations croisées entre les méthodes développées dans chaque communauté sont obscurcies et pénalisées par les différences de notation, de formalisme, de pouvoir d'expression et de champs d'application.

Aussi, cette thèse se décompose en trois grands thèmes. Tout d'abord, je présente le contexte des grammaires attribuées et leur déforestation spécifique avant d'étendre leur pouvoir d'expression pour faciliter la comparaison avec les programmes fonctionnels. Ensuite, j'introduis le contexte fonctionnel, par la présentation et l'étude comparative de différentes méthodes de déforestation ; je compare alors ces méthodes avec celle utilisée pour les grammaires attribuées. Finalement, je m'intéresse à la fertilisation croisée que suggèrent les résultats de ces comparaisons et j'introduis une nouvelle technique de déforestation, plus puissante et générale que celles développées jusqu'à présent et valable tant pour les grammaires attribuées que pour les programmes fonctionnels.

Le contexte des grammaires attribuées

Le chapitre 2 présente un rappel des définitions classiques des grammaires attribuées et des principales méthodes d'évaluation qui leur sont dédiées. Dès ce premier chapitre, je propose une nouvelle formulation des grammaires attribuées, qui ne remet pas en cause les résultats déjà établis, mais qui est destinée à faciliter la compréhension intuitive du formalisme et des résultats présentés. Le but de cette section est principalement de faire le lien entre les travaux existants dans les grammaires attribuées classiques et le formalisme utilisé dans la suite de ce mémoire qui est, pour des raisons évidentes, délibérément orienté vers un style fonctionnel. Ma contribution, dans ce chapitre qui rappelle des notions et des résultats classiques, concerne le formalisme qui est introduit pour la spécification des grammaires attribuées.

Le chapitre 3 présente la composition descriptionnelle. Cette transformation, définie par Ganzinger et Giegerich puis développée par Roussel, confère aux grammaires attribuées un caractère modulaire sans pénaliser leur performance. Ce chapitre pose le contexte de cette transformation de programmes, qui sera ensuite comparée aux méthodes de déforestation fonctionnelles, puis étendue. L'idée intuitive de la composition descriptionnelle qui est la projection de schéma de calcul d'une structure sur une autre est le principe de base de la nouvelle méthode de déforestation générale qui sera présentée au chapitre 8.

Le chapitre 4 propose une extension naturelle et originale des grammaires attribuées : les grammaires attribuées dynamiques. En permettant aux grammaires attribuées classiques de se défaire de la nécessité d'un arbre syntaxique concret, cette extension facilite leur comparaison avec les programmes fonctionnels. Par exemple, la fonction factorielle qui ne s'appuie pas sur une structure de donnée concrète est impossible à spécifier directement avec les grammaires attribuées classiques. L'idée intuitive des grammaires attribuées dynamiques que j'introduis ici est de considérer le schéma de récursion d'une fonction comme structure sous-jacente, en lieu et place de l'arbre syntaxique usuel. L'extension proposée permet par exemple d'exprimer la fonction factorielle dans ce formalisme et une transformation simple permet de se ramener à une grammaire attribuée classique équivalente. Ce principe permet aux grammaires attribuées dynamiques d'atteindre, avec les mêmes méthodes d'évaluation que les grammaires attribués classiques, la puissance d'expression d'un langage fonctionnel du premier ordre.

Ces travaux permettent une représentation dans le formalisme des grammaires attribuées d'un ensemble plus large de programmes fonctionnels. Cela facilite la comparaison des formalismes et des techniques, plus particulièrement entre la composition descriptionnelle et les différentes méthodes de déforestation des programmes fonctionnels, auxquelles je m'intéresse dans les chapitres 5 et 6.

Étude comparative des méthodes de déforestation

Le chapitre 5 présente et compare différentes méthodes fonctionnelles qui ont été développées à des fins de déforestation depuis 10 ans. Les techniques et les formalismes qui sont présentés dans ce chapitre sont issus de différentes équipes de recherches. Dans ce *survey*, chaque méthode est présentée de manière suffisamment formelle, tout en donnant des indications intuitives sur les approches adoptées. En effet, la diversité des formalismes utilisés et leur ancrage dans des théories qui m'étaient peu familières m'ont conduit à étudier précisément les différentes méthodes et à faire abstraction de leurs formalismes afin d'établir leurs effets, leurs principes et les similitudes ou différences qu'elles présentent.

Ce chapitre détaille les trois principales approches qui ont été développées. Tout d'abord, la section 5.2 présente l'algorithme de Wadler, pionnier de la déforestation des programmes fonctionnels ; une série de travaux ont été développés dans la continuité de cet algorithme. Ensuite, la section 5.3 introduit la normalisation des *folds*, ces opérateurs de contrôle génériques qui utilisent à la fois les schémas de récursion des fonctions et ceux des types algébriques sur lesquels elles sont définies. Enfin, la section 5.5 présente la fusion des hylomorphismes, une méthode s'appuyant sur les propriétés algébriques des types et sur des résultats fondamentaux de la théorie des catégories. Cette dernière approche nécessite d'ailleurs quelques rappels sur les formalismes algébriques, qui font l'objet de la section 5.4.

Ma contribution, dans ce chapitre qui est relatif aux travaux d'autres auteurs, tient à la présentation comparative et détaillée qui en est faite, dans le souci de donner au lecteur une vue claire du principe de chaque méthode, illustrée sur des exemples. En outre, pour chaque nouvelle méthode introduite, je présente les similitudes ou les améliorations qu'elle comporte par rapport à la précédente, ainsi que ses limitations ou les problèmes qu'elle soulève.

Le chapitre 6 présente ensuite une comparaison entre les différentes méthodes de déforestations fonctionnelles précédemment étudiées et la composition descriptionnelle des grammaires attribuées. Cette comparaison de leurs formalismes, de leurs principes et de leurs effets, met en évidence des similitudes remarquables concernant le traitement de fonctions assez simples, exprimables par des grammaires attribuées S^1 — ne possédant qu'un seul attribut synthétisé. Par contre, pour des fonctions plus complexes et en particulier sur des fonctions spécifiant des calculs "remontant" dans la structure des données, des différences notables apparaissent. Pour ces programmes, l'existence dans les grammaires attribuées de la notion d'attribut hérité permet de spécifier très simplement les calculs, quel que soit leur sens de propagation dans la structure. Par contre, les formalismes fonctionnels ne peuvent spécifier ce genre de calculs qu'en utilisant des fonctions permettant de modéliser l'accumulation d'un résultat et leur déforestation est bien plus délicate sur ces calculs *encapsulés*. Il apparaît que les schémas de récursion utilisés dans les *folds* ou les hylomorphismes, qui sont déterminés statiquement à partir de la définition d'un type algébrique, sont trop rigides pour permettre la déforestation de ces fonctions, dont le résultat n'est pas construit en suivant exactement le même schéma de récursion que celui de la fonction à déforester. Les évaluateurs d'attributs sont également déterminés statiquement mais prennent en compte, en plus du type des données, la forme des règles sémantiques. Cette différence autorise la déforestation par la composition descriptionnelle de programmes pour lesquels les méthodes fonctionnelles restent impuissantes.

```
let rev x h = case x of
   cons head tail →
      rev tail (cons head h)
   nil → h
```

```
let flat t l = case t of
   node left right →
      flat left (flat right l)
   leaf n → cons n l
```

FIG. 1.3: *Définitions des fonctions rev et flat*

Pour donner tout de suite un exemple intuitif, je présente maintenant un programme Q qui inverse la liste des feuilles d'un arbre binaire. La fonction *rev* inverse une liste grâce à un paramètre d'accumulation. La fonction *flat* construit la liste des feuilles d'un arbre. Ces fonctions sont présentées à la figure 1.3 et le programme Q est donc défini, pour un arbre binaire donné t, par :

$$(Q\ t) = (rev\ (flat\ t\ nil)\ nil)$$

Intuitivement, la liste des feuilles de l'arbre, produite par (*flat t nil*), est une structure intermédiaire qui sera consommée par la fonction *rev*, comme illustré par la figure 1.4.

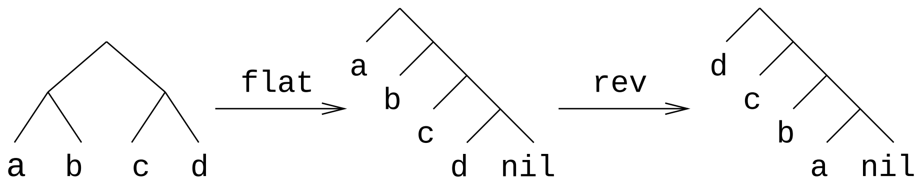

FIG. 1.4: *La structure intermédiaire créée par flat et consommée par rev doit pouvoir être supprimée par déforestation*

Les méthodes fonctionnelles classiques de déforestation sont insuffisantes sur un tel programme. Elles ne permettent pas de transformer Q en un programme construisant uniquement la liste finale, sans aucune autre structure intermédiaire.

Ce résultat optimal est pourtant obtenu par la composition descriptionnelle, appliquée sur les grammaires attribuées équivalentes à *rev* et à *flat*, présentées à la figure 1.5. Une des raisons de ce succès est la plus grande *visibilité* des constructeurs de la structure intermédiaire, grâce à l'utilisation d'attributs hérités dans les spécifications.

Sur de tels programmes, la composition descriptionnelle apparaît donc intrinsèquement plus puissante que les méthodes fonctionnelles existantes. Cependant, elle ne peut pour l'instant être appliquée que sur deux grammaires attribuées distinctes. Les travaux qui sont présentés ensuite permettent de généraliser cette technique et autoriseent son application sur des programmes fonctionnels.

Mise en œuvre d'une nouvelle méthode de déforestation

Il devient donc évident que la composition descriptionnelle doit pouvoir être appliquée à des programmes fonctionnels, quitte à l'enrichir au passage en fonction de ce contexte particulier. Pour cela, il est nécessaire de disposer d'une transformation de programmes fonctionnels en grammaires attribuées permettant de les déforester efficacement. Une fois ces programmes

```
aglet rev =
  rev x h →
    rev.result = x.r
    x.h = h
  cons head tail →
    cons.r = tail.r
    tail.h = (cons head cons.l)
  nil →
    nil.r = nil.h
```

```
aglet flat =
  flat t l →
    flat.result = t.f
    t.l = l
  node left right →
    node.f = left.f
    left.l = right.f
    right.f = node.l
  leaf n →
    leaf.f = (cons n leaf.l)
```

FIG. 1.5: *Définitions des grammaires attribuées pour rev et flat*

déforestés sous leur forme grammaire attribuée, ils pourront être retranscrits en programmes fonctionnels. En effet, la transformation d'une grammaire attribuée en un programme fonctionnel équivalent a déjà été étudiée et ne pose pas de problème majeur [Joh87, PDRJ95a].

Le chapitre 7 présente donc une transformation d'un programme fonctionnel en une grammaire attribuée équivalente. Il s'agit de la traduction FP-to-AG, qui exploite le contexte particulier des programmes fonctionnels. Cette transformation permet de transformer les schémas de récursion explicites des fonctions en des systèmes d'équations orientées, associés aux productions de la grammaire attribuée correspondante.

Le chapitre 8 présente la nouvelle méthode de déforestation générale attendue. Il s'agit de la *composition symbolique*, qui est fondée sur deux transformations fondamentales. La première, l'évaluation symbolique, permet d'effectuer des pas d'évaluation partielle sur des termes finis d'un programme. La seconde, la projection des schémas de calculs, est fondée sur le principe de la composition descriptionnelle et permet de composer des spécifications de grammaire attribuée en produisant des termes qui sont ensuite simplifiés par l'évaluation symbolique. La mise en commun de ces deux transformations permet à la composition symbolique d'éliminer les constructions des structures intermédiaires.

D'une part, la composition symbolique constitue une extension de la composition descriptionnelle. La composition descriptionnelle ne peut être appliquée que sur deux grammaires attribuées distinctes, de manière globale. En revanche, la composition symbolique agit directement sur les termes qui composent les règles sémantiques, à l'intérieur des grammaires attribuées. Par ailleurs, le processus d'évaluation symbolique qu'elle contient permet d'effectuer des pas d'évaluation partielle au cours de la transformation, augmentant ainsi son efficacité.

D'autre part, cette nouvelle technique augmente la puissance de déforestation des programmes fonctionnels. En effet, à l'aide de la transformation FP-to-AG développée au chapitre 7, les programmes fonctionnels traduits en grammaires attribuées peuvent être déforestés par la composition symbolique puis retranscrits en des programmes fonctionnels par la traduction inverse. Ce processus permet de déforester une classe de programmes fonctionnels pour lesquels les méthodes classiques sont impuissantes.

```
let revflat t = revflat2 t (revflat1 t nil)

let revflat1 t l = case t of
  node left right →
    revflat1 right (revflat1 left l)
  leaf n →
    cons n l

let revflat2 t l = case t of
  node left right →
    revflat2 left (revflat2 right l)
  leaf n →
    l
```

FIG. 1.6: *Les fonctions du programme Q' déforesté*

Par exemple, les fonctions *rev* et *flat* (figure 1.3) du programme Q peuvent être traduites sous leurs formes grammaires attribuées (figure 1.5). La composition symbolique appliquée à ces grammaires attribuées produit un résultat qui, une fois retranscrit en programme fonctionnel, donne le programme Q' :

$$(Q' \ t) = (\mathit{revflat} \ t) = (\mathit{revflat2} \ t \ (\mathit{revflat1} \ t \ \mathit{nil}))$$

où les fonctions *revflat*, *revflat2* et *revflat1* sont présentées à la figure 1.6. Le nombre de constructions effectuées par le programme Q' est optimal : c'est exactement le nombre nécessaire à la construction de la liste résultat.

Le chapitre 9 montre que des techniques spécifiques des grammaires attribuées, qui permettent d'effectuer des optimisations statiques source à source, peuvent être utilisées dans le cadre de la composition symbolique. Ainsi, les optimisations d'élimination de règles de copie développées par Roussel [Rou94] accroissent encore les performances obtenues par cette approche en valorisant au maximum les spécificités du formalisme des grammaires attribuées.

Lorsque les conditions d'ordre d'évaluation le permettent, l'application de telles optimisations produit des simplifications notables. Dans le cas du programme Q présenté en exemple ci-dessus, en supposant que l'évaluation sur l'arbre binaire d'entrée t est bien définie, l'élimination des règles de copies de la grammaire attribuée obtenue par la composition symbolique permet d'obtenir une grammaire attribuée correspondant au programme Q'', où la fonction *Rflat* est présentée à la figure 1.7 :

$$(Q'' \ t) = (\mathit{Rflat} \ t \ \mathit{nil})$$

Le formalisme des grammaires attribuées, particulièrement adapté aux transformations de programmes dirigées par la structure des données, produit pour cet exemple un résultat très satisfaisant. En effet, la composition de la fonction d'inversion de liste avec une fonction qui construit la liste des feuilles d'un arbre de gauche à droite produit après déforestation et optimisation une fonction qui construit la liste des feuilles d'un arbre sans constructions intermédiaires superflues et de droite à gauche.

$$
\begin{array}{l}
\texttt{let}\ \mathit{Rflat}\ t\ l = \texttt{case}\ t\ \texttt{of} \\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\quad\quad \mathit{Rflat}\ \mathit{right}\ (\mathit{Rflat}\ \mathit{left}\ l) \\
\quad \mathit{leaf}\ \ n \rightarrow \mathit{cons}\ \ n\ \ l
\end{array}
$$

FIG. 1.7: *La fonction du programme Q'', après déforestation et optimisation*

Pour terminer, le chapitre 10 conclut la présentation de ces travaux, en dressant le bilan des contributions qu'ils apportent. Je propose également dans cette conclusion des ouvertures vers différents travaux à mener, pour poursuivre l'étude et la comparaison entre différents paradigmes de programmation, mais également pour utiliser et valoriser les résultats déjà obtenus.

Quelques conseils de lecture

L'ordre dans lequel sont présentés les travaux dans cette thèse est bien évidement prévu pour qu'elle soit lue d'un bout à l'autre. Cependant, les chapitres sont assez indépendants les uns des autres et, selon ses connaissances, le lecteur peut désirer aller rapidement à l'essentiel. Je donne ici brièvement les indications pour les pré-requis à la lecture de chacun des chapitres.

Le lecteur connaissant bien le domaine des grammaires attribuées peut survoler le chapitre 2, qui a pour but d'en donner les notions de bases ainsi qu'une impression générale intuitive de leurs principes, illustrés à l'aide d'exemples.

Le chapitre 3 rappelle à la fois le principe et l'algorithme de composition descriptionnelle. L'idée intuitive et les exemples sont facilement compréhensibles mais l'algorithme, assez technique, n'est pas indispensable pour le reste de la présentation.

Le chapitre 4 sur les grammaires attribuées dynamiques est assez théorique et relativement indépendant du reste de la présentation. Le lecteur surtout intéressé par la déforestation peut en faire abstraction en première lecture.

Le chapitre 5 est une sorte de *survey* de différentes méthodes de déforestation. Il doit permettre au lecteur de se familiariser avec les différentes approches fonctionnelles et donne des indications pour les comparer entre elles. Il peut être lu seul, en aparté du reste de ce mémoire.

La lecture du chapitre 6 de comparaison entre la composition descriptionnelle des grammaires attribuées et les déforestations fonctionnelles nécessite, sinon leur connaissance, au moins le parcours préalable des chapitres 3 et 5.

La traduction d'un programme fonctionnel en grammaire attribuée, présentée au chapitre 7, peut être admise pour la suite de la présentation, ou étudiée seule : elle est pratiquement indépendante du reste et ne nécessite que quelques notions sur les principes des grammaires attribuées.

Il en va de même pour la composition symbolique, présentée au chapitre 8. Elle est évidemment inspirée des conclusions des travaux précédents, mais peut être abordée indépendamment du reste, à partir des notions générales des grammaires attribuées en notation fonctionnelle et de l'idée intuitive de la déforestation.

Finalement, l'élimination des règles de copies présentée au chapitre 9 est relativement autonome, mais ne trouve tout son sens qu'après la lecture des deux chapitres précédents.

Chapitre 2

Grammaires attribuées

2.1 Notations et formalismes

Cette section présente tout d'abord la notion de grammaire sous-jacente à la définition classique des grammaires attribuées. Afin de donner rapidement une idée intuitive au lecteur, les notions abordées sont illustrées par des exemples donnés dans le formalisme classique, puis une notation plus fonctionnelle pour les grammaires attribuées est introduite. Cette nouvelle notation, qui sera utilisée tout au long de ce mémoire, est équivalente à la notation classique mais permet de se libérer de certaines des contraintes de l'écriture des spécifications par grammaires attribuées. Elle facilitera ensuite leur comparaison avec les programmes fonctionnels.

2.1.1 Grammaires attribuées classiques

Je commence par rappeler la définition d'une grammaire indépendante du contexte, qui est à la base de la notion de grammaire attribuée. L'acronyme anglais CFG (*Context-Free Grammar*) sera souvent utilisé pour dénoter une grammaire indépendante du contexte.

Définition 2.1.1 (Grammaire Indépendante du Contexte) *On appelle grammaire indépendante du contexte (CFG) le quadruplet* (N, T, Z, P) *avec :*

- N *un ensemble fini de non-terminaux ;*
- T *un ensemble fini de terminaux,* $N \cap T = \emptyset$ *;*
- P *un ensemble fini de productions,* $P \subset N \times [N \cup T]^*$ *de la forme* $p : X_0 \to X_1 \ldots X_n$ *où* $(X_0 \in N$ *et* $X_i \in (N \cup T))$[1] *;*
- Z *l'axiome,* $Z \in N$ *; il n'apparaît jamais en partie droite d'une production.*

□

Dès maintenant, je suppose qu'il existe un moyen d'analyser le langage engendré par la grammaire G et de construire les arbres syntaxiques correspondants. Il faut cependant préciser que, dans notre contexte, la grammaire abstraite représentant le type récursif sous-jacent est

1. Je ne détaille pas ici le cas des productions n'ayant aucun non-terminal en partie droite ($p : X_0 \to$), qui est classiquement pris en compte par un terminal particulier, noté ϵ [Rou94]. L'exemple typique est celui de la production $nil : L \to$ du type des listes, présenté à la page suivante.

CFG des listes (*list* e)
$(\{Z, L\}, \{e\}, \{root, cons, nil\}, Z)$
avec

$$\begin{array}{lcl} root & : & Z \rightarrow L \\ cons & : & L \rightarrow e\ L \\ nil & : & L \rightarrow \end{array}$$

CFG des arbres binaires (*tree* e)
$(\{Z, T\}, \{e\}, \{root, node, leaf\}, Z)$
avec

$$\begin{array}{lcl} root & : & Z \rightarrow T \\ node & : & T \rightarrow T\ T \\ leaf & : & T \rightarrow e \end{array}$$

FIG. 2.1: *Grammaires indépendantes du contexte des types list et tree*

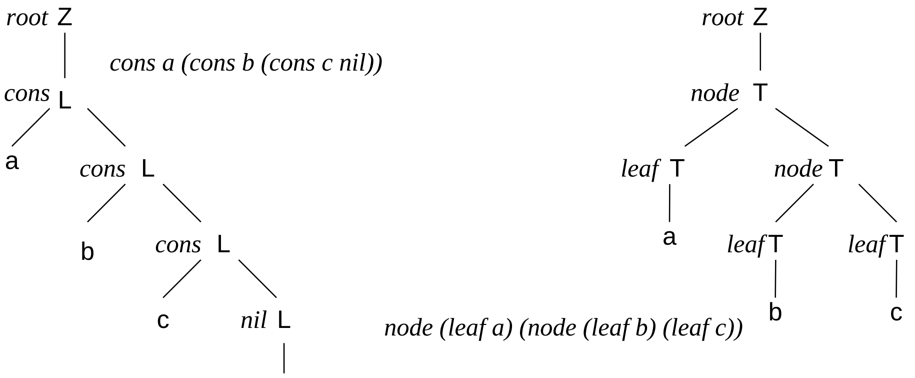

FIG. 2.2: *Arbres syntaxiques abstraits dérivés à partir d'exemples de "phrases" par les CFGs de list et de tree*

plus intéressante que la grammaire textuelle à proprement parler. Par exemple, la figure 2.1 donne les définitions de deux grammaires : celle des listes et celle des arbres binaires. Les mots clés de construction tels que *cons* ou *node* n'apparaissent pas dans cette grammaire abstraite. Les terminaux qui y apparaissent, comme e par exemple, sont en fait ceux qui représentent les feuilles sémantiquement intéressantes. Ils devraient en réalité être des non-terminaux possédant une valeur lexicale variable en fonction du texte analysé mais je préfère, pour simplifier la présentation, les considérer comme des terminaux ayant une valeur.

En ce sens, les non-terminaux seront vus comme des types syntaxiques et les terminaux comme des types sémantiques. Les terminaux peuvent être génériques et, lors de l'analyse d'une phrase, il est possible de récupérer la valeur des terminaux, conformément à leur type sémantique associé.

Plus généralement, les travaux décrits dans ce mémoire ne s'intéressent pas aux problèmes d'analyse syntaxique des terminaux (*parsing*) et considèrent une grammaire comme la définition algébrique d'une famille d'arbres, de termes ou de structures. Dans leur notation, les terminaux (resp. non-terminaux) qui représentent les types seront confondus avec les occurrences de terminaux (resp. occurrences de non-terminaux) qui interviennent effectivement dans les productions.

En guise d'exemple, la figure 2.2 présente les arbres syntaxiques abstraits correspondant à des "phrases" dérivées selon les CFGs des listes et des arbres binaires de la figure 2.1. Intuitivement, chaque nœud d'un arbre correspond à la partie gauche d'une production de la grammaire et chacun des nœuds fils correspond à un non-terminal en partie droite de la même production. Les feuilles de l'arbre correspondent à des terminaux de la grammaire.

Une grammaire attribuée est la spécification, pour chaque production d'une grammaire, d'un ensemble d'équations orientées, appelées règles sémantiques, permettant de définir localement les valeurs de variables, appelées attributs, qui sont attachées aux non-terminaux de la grammaire. Voici la définition formelle d'une grammaire attribuée :

Définition 2.1.2 (Grammaire Attribuée) *Une **grammaire attribuée** est un triplet $AG = (G, A, F)$ où :*

- *$G = (N, T, Z, P)$ est une grammaire indépendante du contexte ;*
- *$A = \bigcup_{X \in N} H(X) \uplus S(X)$ est un ensemble d'attributs[2], avec*
  *$H(X)$ les attributs **hérités** de $X \in N$*
  *$S(X)$ les attributs **synthétisés** de $X \in N$;*
- *$F = \bigcup_{p \in P} F(p)$ est un ensemble de **règles sémantiques**, où dans chaque $F(p)$, r_{p,a,X_i} désigne la règle sémantique définissant l'occurrence de l'attribut a sur le non-terminal X_i dans la production $p : X_0 \to X_1 \dots X_n$ quand $a \in A(X_i)$.*
  De façon générique, une règle sémantique est de la forme[3] :

$$r_{p,a_0,X_{\rho(0)}} : X_{\rho(0)}.a_0 = (f_{p,a_0,X_{\rho(0)}}\ X_{\rho(1)}.a_1 \dots X_{\rho(q)}.a_q)$$

  où $\forall k \in \{0, \dots, q\}$, $\rho(k) \in \{0, \dots, n\}$ et $a_k \in A(X_{\rho(k)})$.
  et où $f_{p,a_0,X_{\rho(0)}}$ est une fonction q-aire.

□

Intuitivement, les attributs synthétisés calculent ou propagent de l'information du bas vers le haut dans un arbre syntaxique abstrait de la grammaire, c'est-à-dire des feuilles de l'arbre vers sa racine. Les attributs hérités propagent ou calculent de l'information dans l'autre sens, du haut vers le bas. On appelle **classe** d'un attribut le fait qu'il soit hérité ou synthétisé. La **classe opposée** d'hérité (resp. de synthétisé) est synthétisé (resp. hérité). Par ailleurs, l'attachement d'un attribut a à un non-terminal X est noté $X.a$ et appelé **occurrence** de l'attribut.

L'ensemble des occurrences d'attributs **de sortie**, ou **définis**, d'une production $p : X_0 \to X_1 \dots X_n$ et noté $W_d(p)$ regroupe les attributs synthétisés portés par X_0 et les hérités portés par les $X_1 \dots X_n$.

$$W_d(p) = S(X_0) \cup \left(\bigcup_{1 \leq i \leq n} H(X_i) \right)$$

De même, l'ensemble des occurrences d'attributs **d'entrée**, ou **utilisées** d'une production $p : X_0 \to X_1 \dots X_n$ et noté $W_u(p)$ regroupe les occurrences attributs hérités de X_0 et les synthétisés des $X_1 \dots X_n$.

$$W_u(p) = H(X_0) \cup \left(\bigcup_{1 \leq i \leq n} S(X_i) \right)$$

2. Ici, $\uplus$ représente la réunion disjointe. Par commodité, on impose en général que $S(X) \cap H(X) = \emptyset$ pour tout X.
3. D'une manière générale, dans ce mémoire, les applications de fonctions seront notées *à la Lisp*, i.e., $(f\ x_1 \dots x_n)$ plutôt que par la notation classique pour les règles sémantiques $f(x_1, \dots, x_n)$.

L'ensemble des occurrences d'attributs d'une production p est alors naturellement défini par

$$W(p) = W_d(p) \cup W_u(p)$$

Je ne traiterai dans cette thèse que des grammaires attribuées **bien formées**, de telle sorte que $F(p)$ contienne exactement une règle sémantique définissant chaque occurrence d'attribut de sortie, soit

$$\forall X_i.a \in W_d(p),\ \exists!\ r_{p,a,X_i} \in F(p)$$

De plus, je ne traiterai que des grammaires attribuées en **forme normale**, c'est-à-dire dont les occurrences d'attributs en partie droite des règles sémantiques sont toutes des occurrences d'attributs d'entrée (dans $W_u(p)$). Cette contrainte ne diminue pas le pouvoir d'expression puisqu'il est possible, à partir d'une grammaire attribuée *non-circulaire* [4] qui n'est pas en forme normale, d'obtenir une grammaire attribuée équivalente en forme normale [Boc76].

Je prends maintenant les exemples des grammaires attribuées *length* et *reverse* qui spécifient respectivement le calcul de la longueur d'une liste et la construction de l'inversion d'une liste. La spécification d'une grammaire attribuée est la donnée des règles sémantiques associées à chaque production de la CFG sous-jacente, où l'on peut numéroter les non-terminaux pour les distinguer dans une production.

La grammaire attribuée *length* présentée à gauche de la figure 2.3 n'utilise qu'un seul attribut synthétisé s. Pour chaque production [5] p de la CFG des listes (cf. figure 2.1), une règle sémantique définit l'occurrence de l'attribut s sur le non-terminal en partie gauche de p. L'arbre syntaxique de l'exemple de liste de la figure 2.2 est présenté à la figure 2.4, *décoré* par les occurrences des attributs de la grammaire attribuée *length*. Les flèches représentent les *dépendances* entre les occurrences d'attributs, c'est-à-dire l'ordre dans lequel ils doivent être calculés. Cet ordre est induit par la forme des règles sémantiques (je reviendrai sur les notions de dépendances à la section 2.2.2). Classiquement, le résultat de la grammaire attribuée est contenu dans l'unique attribut synthétisé z de la racine Z.

4. J'introduirai à la section 2.2.1 la notion de *non-circularité* d'une grammaire attribuée.
5. À l'exception de *root* où l'occurrence d'attribut définie est $Z.z$ qui représente le résultat de la grammaire attribuée : dans ce cas, l'attribut synthétisé défini par la règle sémantique est z.

règles sémantiques de la grammaire attribuée *length* sur les productions de la CFG *list*	règles sémantiques de *reverse*
$root : Z \rightarrow L$	$root : Z \rightarrow L$
$\quad Z.z = L.s$	$\quad Z.z = L.r$
	$\quad L.h = nil$
$cons : L_0 \rightarrow e\ L_1$	$cons : L_0 \rightarrow e\ L_1$
$\quad L_0.s = L_1.s + 1$	$\quad L_0.r = L_1.r$
	$\quad L_1.h = cons\ e\ L_0.h$
$nil : L \rightarrow$	$nil : L \rightarrow$
$\quad L.s = 0$	$\quad L.r = L.h$

FIG. 2.3: *Spécification des règles sémantiques des grammaires attribuées length et reverse*

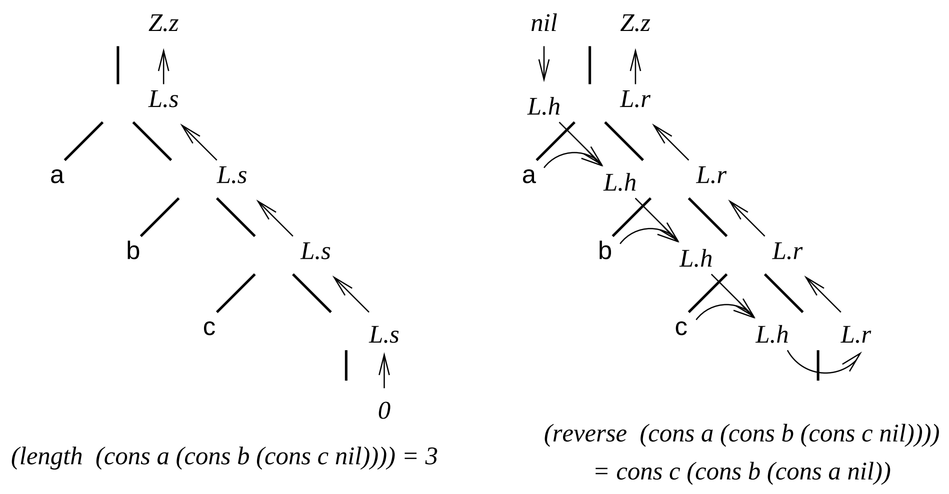

FIG. 2.4: *Arbres syntaxiques abstraits décorés par les attributs des grammaires attribuées length (à gauche) et reverse (à droite)*

La grammaire attribuée *reverse*, présentée à droite de la figure 2.3 et illustrée par l'arbre syntaxique décoré de droite dans la figure 2.4, utilise un attribut synthétisé r et un attribut hérité h. Intuitivement, l'attribut hérité h initialisé à *nil* calcule la liste inversée dans les occurrences de ses différentes instances en *descendant* le long de l'arbre syntaxique, jusqu'à trouver la production *nil*, où il transmet la liste inversée à l'occurrence de l'attribut r, qui le *remonte* jusqu'à $Z.z$.

2.1.2 Notation fonctionnelle des grammaires attribuées

Les grammaires attribuées, habituellement spécifiées sur des grammaires indépendantes du contexte, peuvent également être vues comme des spécifications sur des types algébriques. Chirica et Martin [CM76, CM79] ou Ganzinger et Giegerich [Gan80, GG84] ont déjà étudié cette représentation, plus souple qu'avec des CFGs, dans laquelle les constructeurs de type jouent le rôle des productions de la CFG. Les terminaux d'une CFG sont alors les variables d'une définition de type algébrique (figure 2.5) ; en supposant que leur type est un paramètre, on obtient alors une représentation polymorphique. Pour des raisons historiques de domaine d'application, cette modification du formalisme original n'a pas été approfondie dans la majorité des études sur les grammaires attribuées, mais cela n'entame en rien leur principe et leurs méthodes d'évaluation.

Afin de rapprocher les notations des grammaires attribuées de celles classiquement utilisées

$$\begin{array}{lcl} list\ \alpha & = & cons\ \alpha\ (list\ \alpha) \\ & & \mid nil \\ tree\ \alpha & = & node\ (tree\ \alpha)\ (tree\ \alpha) \\ & & \mid leaf\ \alpha \end{array}$$

FIG. 2.5: *Exemples de types algébriques*

dans les programmes fonctionnels, je vais maintenant introduire une nouvelle notation pour les grammaires attribuées. Le principe de cette notation est de voir une production comme un filtrage syntaxique (*pattern matching*) implicite[6]. Au lieu d'utiliser des symboles de non-terminaux et de terminaux des productions classiques ($p : X_0 \rightarrow X_1 \ldots X_n$) qui correspondent aux constructeurs et à leurs paramètres du type de donnée représenté par la grammaire, j'utilise des noms de variables, comme dans un *pattern*. Le premier nom apparaissant dans le *pattern* représente alors le nom du constructeur de type, correspondant au nom de la production, soit $p\ var_1\ \ldots var_n$. Plus concrètement, la production de la grammaire des listes $cons : L_0 \rightarrow e\ L_1$ est par exemple utilisée comme dans un *pattern* fonctionnel : *cons head tail* où *cons*, *head* et *tail* sont des variables de type jouant respectivement le rôle des non-terminaux ou terminaux L_0, e et L_1. Une originalité, par rapport aux *patterns* fonctionnels classiques, qui peut être vue comme une ambiguïté dans un premier temps est alors de pouvoir utiliser le nom d'un constructeur, représentant une production, comme un non-terminal pouvant porter un attribut.

La figure 2.5 présente en rappel deux exemples de définitions de types algébriques simples qui sont couramment utilisés. Les grammaires attribuées peuvent donc être définies sur de tels types algébriques en ajoutant un **profil** pour chaque grammaire attribuée. Ce profil est une sorte de *constructeur* particulier qui porte le même nom que la grammaire attribuée. Il permet de palier l'absence de *racine* ($root : Z \rightarrow ...$) dans les types algébriques et distingue ainsi le premier *pattern matching* correspondant à l'*appel* de la grammaire attribuée, spécifiant son résultat par un attribut particulier noté *result* et tenant le rôle de l'attribut z classique. Outre ces petites différences purement syntaxiques, ce profil permet également de spécifier, en plus de l'argument syntaxique correspondant à l'arbre d'entrée, des paramètres supplémentaires pour la grammaire attribuée qui seront alors vus comme des attributs hérités en tête de l'arbre d'entrée. La souplesse de cette notation rapproche les spécifications de grammaires attribuées de celles des fonctions classiques, sans remettre en cause leurs principes fondamentaux.

La syntaxe donnée à la figure 2.6 formalise cette notation fonctionnelle pour les grammaires attribuées. Par sa forme, proche des spécifications fonctionnelles, elle facilite la compréhension des exemples et l'illustration des techniques que je vais présenter, ainsi que les comparaisons avec les différents formalismes fonctionnels présentés dans les prochains chapitres. Par ailleurs, elle est équivalente à la notation classique déjà présentée et j'utiliserai indifféremment l'une ou l'autre de ces notations. En fait, pour tous les résultats classiques des grammaires attribuées, concernant leur méthodes d'évaluation ou leurs analyses statiques, la notation utilisant les non-terminaux et leur position dans une production est mieux adaptée pour une présentation simple. En revanche, la spécification des fonctions est plus naturelle et plus facilement compréhensible sous la forme de variables de *patterns* expressives que sur des non-terminaux numérotés d'une grammaire indépendante du contexte.

Dans la syntaxe de la figure 2.6, une grammaire attribuée f est vue comme un bloc contenant plusieurs *patterns*, dont au moins un pour le profil. La notation surlignée $\overline{x}$ représente un nombre fini d'éléments et évite de spécifier les indices $x_1 \ldots x_n$. Chacune des règles sémantiques est une équation orientée, avec une occurrence d'attribut $x.a$ en partie gauche (éventuellement $f.result$ pour le profil) et avec en partie droite une expression. Une expres-

6. Cette idée a déjà été évoquée dans différents travaux, dont ceux de Johnsson, qui parlait alors de construction `case rec` [Joh87].

$$
\begin{array}{lcl}
\textit{bloc} & ::= & \texttt{aglet}\ f = \{f\ \overline{x} \rightarrow \overline{\textit{semrul}}\}\{\textit{pat} \rightarrow \overline{\textit{semrul}}\}^* \\
\textit{semrul} & ::= & \textit{occ} = \textit{exp} \\
\textit{occ} & ::= & x.a \mid f.\textit{result} \\
\textit{exp} & ::= & \textit{Constante} \\
 & | & y.b \in \textit{Occurences d'attributs} \\
 & | & x \in \textit{Variables} \\
 & | & g\ \overline{\textit{exp}}
\end{array}
$$

FIG. 2.6: *Une notation fonctionnelle pour les grammaires attribuées*

$$
\begin{array}{l}
\texttt{aglet}\ \textit{rev} = \\
\quad \textit{rev}\ x\ l \rightarrow \\
\qquad \textit{rev.result} = x.r \\
\qquad x.h = l \\
\quad \textit{cons}\ \textit{head}\ \textit{tail} \rightarrow \\
\qquad \textit{cons}.r = \textit{tail}.r \\
\qquad \textit{tail}.h = \textit{cons}\ \textit{head}\ \textit{cons}.h \\
\quad \textit{nil} \rightarrow \\
\qquad \textit{nil}.r = \textit{nil}.h
\end{array}
$$

FIG. 2.7: *Grammaire attribuée en notation fonctionnelle pour rev*

sion est construite à partir de constantes, de variables, d'occurrences d'attributs ou d'appels de fonctions.

Pour illustrer cette notation, la figure 2.7 présente la grammaire attribuée correspondant à la fonction *rev* qui retourne une liste passée en argument, en utilisant un paramètre (l) initialisée à *nil*. Par opposition à la grammaire attribuée classique *reverse* présentée à la figure 2.3, la valeur *nil* n'est plus une constante nécessairement fixée dans la règle sémantique initialisant l'attribut hérité h, mais peut être vue comme un paramètre de la grammaire attribuée, comme c'est le cas dans la définition de fonction correspondante. Autrement dit, l'appel de cette grammaire attribuée pour inverser une liste x se fait *"à la fonctionnelle"* par $rev\ x\ nil$.

Avec cette notation, le nom *rev* représente à la fois le nom de la grammaire attribuée et le nom du constructeur de profil. Cela peut semer la confusion chez le lecteur peu habitué à cette notation, mais ce choix est justifié par la nécessité de différencier (resp. identifier) les profils des grammaires attribuées différentes (resp. identiques) ; par ailleurs, le nom de la grammaire attribuée est plus pertinent que *root*. Le chapitre 7 traitant d'une transformation de programmes fonctionnels en grammaires attribuées permettra à cette remarque de prendre tout son sens.

De même, les noms des constructeurs (e.g., *cons*) peuvent être utilisés comme tels dans les *patterns* (e.g., *cons head tail*), comme des non-terminaux pouvant porter des attributs (e.g., $cons.r$) ou comme des fonctions de construction (e.g., $tail.h = cons\ head\ cons.h$). Cette ambiguïté est similaire à celle des grammaires attribuées classiques et, si elle semble compliquer les notations, elle est néanmoins très pratique lors de la spécification d'un programme.

```
aglet length =                  aglet sumleaves =
length x →                      sumleaves t →
  length.result = x.s             sumleaves.result = t.s
cons head tail →                node left right →
  cons.s = succ tail.s            node.s = left.s + right.s
nil →                           leaf n →
  nil.s = zero                    leaf.s = n
```

FIG. 2.8: *Définitions des grammaires attribuées length et sumleaves*

Pour familiariser le lecteur avec cette notation, deux autres exemples de grammaires attribuées simples sont donnés dans la figure 2.8 : la longueur d'une liste *length* et la somme des feuilles d'un arbre binaire *sumleaves*.

2.2 Évaluation des grammaires attribuées

Pour une spécification de grammaire attribuée donnée, différentes techniques d'évaluation de leur résultat sur un paramètre donné peuvent être employées. Je rappelle brièvement quelques-unes de ces techniques à la fin de cette section, mais elles sont toutes fondées sur la notion de *dépendance*, que je vais présenter après avoir défini la notion d'*arbre d'entrée*, fondamentale dans l'évaluation d'une grammaire attribuée.

Comme je l'ai déjà dit, la notation fonctionnelle des grammaires attribuées est particulièrement adaptée pour faciliter la compréhension des exemples et leur comparaison avec les programmes fonctionnels. Cependant, pour les techniques d'évaluations dont traite cette section, je lui préférerai la notation classique utilisant les non-terminaux numérotés de grammaire, mieux adaptée à la présentation de ces notions. Néanmoins, dans les exemples illustratifs, je préférerai les noms de variables, plus intuitifs, aux numéros de non-terminaux (*cons*, *tail* ou *nil* plutôt que L_0, L_1 ou L).

2.2.1 Arbre d'entrée

Une grammaire attribuée $AG = (G, A, F)$ est une méthode déclarative de spécification d'un calcul à effectuer sur un objet structuré que l'on appelle l'arbre d'entrée. Ce dernier peut être l'arbre correspondant à l'analyse de la "phrase" d'entrée dans le langage $\mathcal{L}(G)$. Ses feuilles portent les valeurs qui seront utilisées pour effectuer le calcul et ses nœuds portent des instances des attributs de A. Cela implique qu'un calcul ne peut se faire que sur un seul arbre d'entrée à la fois : les valeurs nécessaires aux calculs sont obtenues par l'analyse et le parcours de sa structure.

Définition 2.2.1 (Arbre d'entrée) *Un* ***arbre d'entrée*** *t pour une grammaire attribuée $AG = (G, A, F)$ est un arbre syntaxique associé à une phrase du langage associé à G.*

Un nœud de l'arbre t est noté u et un nœud feuille[7] *de celui-ci est noté l. A chaque nœud u est associée une production $label(u) = p : X_0 \to X_1 \cdots X_n$ telle que :*

- *u est associé au non-terminal X_0 ;*

7. Pour les productions vides, telles que $nil : L \to$, on considère le nœud feuille vide particulier, noté ϵ [Rou94].

- *u a n fils (u est d'arité n);*

- *si u_i est le nœud fils de u en position i, alors u_i est associé au non-terminal X_i;*

- *si l_j est le nœud feuille fils de u en position j, alors l_j est associé au terminal X_j.*

Sur un arbre d'entrée particulier t, $u \to \ldots l_j \ldots u_i \ldots$ est appelée une ***instance de la production*** *$label(u)$.*
□

Étant donnée une grammaire attribuée AG, l'évaluation des attributs sur un arbre d'entrée t consiste à évaluer l'ensemble des attributs associés à chaque nœud de l'arbre d'entrée, conformément aux règles sémantiques de AG.

Pour cela, à chaque nœud u de l'arbre d'entrée est associé l'ensemble des **instances d'attributs** de u, noté $IA(u)$, tel que

$$\text{si } label(u) = p : X_0 \to X_1 \ldots X_n \quad \text{alors} \quad IA(u) = \{u.a \mid a \in A(X_0)\}$$

De même, à chaque instance de production $p = label(u)$ correspondant à un nœud u de l'arbre d'entrée est associé l'ensemble des **instances des règles sémantiques** de $F(p)$. Plus précisément, à chaque règle sémantique de $F(label(u))$ de la forme

$$r_{label(u),a,X_i} : X_i.a = (f_{label(u),a,X_i}\ X_{j_1}.a_1 \ldots X_{j_q}.a_q)$$

est associée l'équation

$$u_i.a = (f_{label(u),a,X_i}\ u_{j_1}.a_1 \ldots u_{j_q}.a_q)$$

où u_i représente le nœud correspondant à l'instance de non-terminal X_i et où chaque u_{j_k} correspond au non-terminal ou au terminal X_{j_k}.

Toutes les instances de règles sémantiques sur toutes les instances d'attributs de tous les nœuds d'un arbre d'entrée donné t constituent un système d'équations orientées $K(t)$ dont les inconnues sont précisément les instances d'attributs.

Évaluer la grammaire attribuée sur l'arbre d'entrée t revient alors à résoudre le système $K(t)$.

Classiquement, l'évaluation d'une grammaire attribuée pour un arbre t est vue comme la décoration de l'arbre d'entrée permettant de donner une valeur à chaque instance d'attribut de t. Cependant, dans une vue plus fonctionnelle, le calcul du résultat de la grammaire attribuée pour un arbre t, appelé **valeur sémantique** de t, peut être restreint au calcul de la valeur de l'unique attribut synthétisé de la racine ($Z.z$ ou $AG.result$, selon la notation utilisée). Cela revient donc à calculer les instances d'attributs nécessaires au calcul de cet attribut.

Un système d'équations orientées tel que $K(t)$ est dit **circulaire** s'il possède une inconnue (une instance d'attribut) dont le calcul nécessite, par transitivité des équations, la connaissance de cette inconnue elle-même: sa valeur ne peut donc pas être déterminée de façon unique. Le système $K(t)$ possède une unique solution s'il n'est pas circulaire. Une grammaire attribuée est alors dite **non-circulaire** si, quel que soit l'arbre d'entrée t, le système $K(t)$ associé n'est pas circulaire. Dans ce cas, il existe un ordre simple de calcul des instances d'équations permettant d'obtenir la valeur sémantique d'un arbre d'entrée donné t.

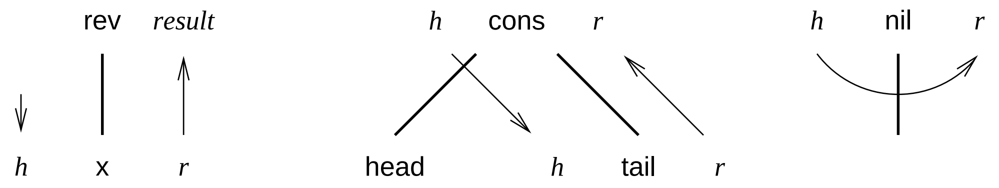

FIG. 2.9: *Graphes de dépendances locales des productions de la grammaire attribuée rev*

2.2.2 Graphe de dépendances

Sur une production $p : X_0 \rightarrow X_1 \ldots X_n$, une grammaire attribuée définit un ensemble de règles sémantiques qui impliquent des occurrences d'attributs sur les non-terminaux de p. Chacune de ces règles sémantiques définit en effet une occurrence d'attribut particulière $X_i.a$ en fonction d'un ensemble d'occurrences d'attributs $X_{j_k}.a_k$. Le calcul de $X_i.a$ nécessite donc le calcul préalable des $X_{j_k}.a_k$: on dit qu'il en *dépend*. Plus formellement, pour chaque production dans une grammaire attribuée, la notion de graphe de dépendances de production, ou graphe de dépendances locales, est définie comme suit.

Définition 2.2.2 (Graphe de dépendances locales) *Pour chaque production $p : X_0 \rightarrow X_1 \ldots X_n$ d'une grammaire attribuée, le **graphe de dépendances locales** $D(p)$ est défini par :*

- *l'ensemble de ses sommets, étant les occurrences d'attributs de $W(p)$;*
- *l'ensemble de ses arcs, sachant qu'il existe un arc allant de $X_{j_k}.b_k$ vers $X_i.a$ si et seulement si il existe une règle sémantique de la forme :*

$$r_{p,a,X_i} : X_i.a = (f_{p,a,X_i}\ X_{j_1}.a_1 \ldots X_{j_k}.b_k \ldots X_{j_q}.a_q)$$

□

Par abus de langage, ce graphe de dépendances locales est confondu avec la relation $\xrightarrow{p}$ qui lui est associée. Dans le cas de la définition 2.2.2, la **dépendance** entre $X_{j_k}.b_k$ et $X_i.a$ est alors notée $X_{j_k}.b_k \xrightarrow{p} X_i.a$.

Ces graphes de dépendances locales donnent l'ordre partiel d'évaluation à respecter pour les occurrences d'attributs sur chacune des productions.

Par exemple, pour la grammaire attribuée *rev* présentée à la figure 2.7, les graphes de dépendances locales sur chacune des productions sont donnés à la figure 2.9. Sur la production *nil*, l'occurrence d'attribut *nil.h* devra nécessairement être calculée avant *nil.r*. De même, sur la production *cons*, l'occurrence *cons.h* devra être calculée avant *tail.h* et l'occurrence *tail.r* avant *cons.r*. Par contre, ce graphe de dépendances locales sur *cons* ne dit rien sur l'ordre d'évaluation entre les occurrences de l'attribut h et celles de l'attribut r. Pour disposer d'informations sur cet ordre, il faut considérer les graphes de dépendances des autres productions.

Étant donné un arbre d'entrée t, pour obtenir un graphe de dépendances compatible avec un ordre d'évaluation possible des instances d'attributs dans $K(t)$, il faut considérer le graphe de dépendances **complet** sur l'arbre t, comme étant l'union des instanciations des graphes de dépendances locales sur l'ensemble des instances de productions de t. Intuitivement, ce graphe de dépendances complet est obtenu en collant les uns aux autres les graphes de dépendances locales de chaque production en suivant les instances de productions correspondant à la structure de t. La relation associée à ce graphe de dépendances complet est alors notée $\xrightarrow{t}$ et elle fournit un ordre d'évaluation des instances de $K(t)$.

2.2.3 Différentes méthodes d'évaluation

À partir d'une spécification de grammaire attribuée, le problème consiste donc à déterminer un ordre de calcul des instances d'attributs qui aboutit à la valeur sémantique d'un arbre d'entrée donné. Ce problème est pris en charge par un **évaluateur d'attributs**, qui peut être produit de différentes manières. Par ailleurs, ces différentes méthodes permettant de produire un évaluateur définissent des classes de grammaires attribuées pour lesquelles elles peuvent être appliquées.

Une première famille d'évaluateurs, basée sur la méthode dite méthode **dynamique**, est très proche de la résolution du système. Elle consiste à trouver dynamiquement pour un arbre d'entrée t l'ordre sur les instances d'attributs induit par $\xrightarrow{t}$ et à calculer les instances d'attributs en allant des plus petites aux plus grandes au sens de la clôture transitive de $\xrightarrow{t}$. Cette méthode très simple a été utilisée dans de nombreux systèmes de traitement de grammaires attribuées [Fan72, Lor74, Lor77, CH79, KR79, Mad80, Jal83, Jou84b]. La classe des grammaires attribuées acceptées par cette méthode est la classe des grammaires attribuées non-circulaires. Malheureusement, les évaluateurs basés sur cette méthode sont très peu efficaces car, pour chaque arbre d'entrée, ils doivent reconstruire l'ordre. D'autre part, même s'il est possible de tester statiquement la non-circularité d'une grammaire attribuée, le test est d'une complexité exponentielle en fonction de la taille de la grammaire [DJL84].

Par opposition à cette méthode, de nombreux auteurs ont cherché à déterminer de façon **statique** l'ordre d'évaluation.

Une deuxième famille d'évaluateurs, dits statiques, consiste alors à fixer d'une façon arbitraire la stratégie de parcours de l'arbre pour le calcul des instances d'attributs. Les classes acceptées les plus importantes sont les classes L et Sweep. La classe L permet de calculer tous les attributs en parcourant l'arbre en plusieurs passes de gauche à droite. La classe Sweep revient à la classe L moyennant une permutation de la partie droite de certaines productions. Un certain nombre de systèmes sont basés sur cette approche [DJL88]. Les évaluateurs de cette famille sont très efficaces, mais la classe des grammaires attribuées acceptées est très restreinte, de par le choix arbitraire du parcours.

Une troisième famille d'évaluateurs, également statiques, concerne ceux qui essaient, à partir des graphes de dépendances locales, de déduire statiquement un ordre d'évaluation des occurrences d'attributs d'une production qui soit valable quel que soit l'arbre d'entrée. Les différentes méthodes se différencient par le type de l'ordre qu'elles construisent. Par opposition à la famille précédente où l'ordre était fixé de façon arbitraire, ici c'est la structure de la grammaire attribuée qui induit l'ordre. La classe des grammaires attribuées acceptées est donc plus large et fait de cette famille d'évaluateurs la plus intéressante en matière de développement de systèmes de traitement de grammaires attribuées et de génération automatique d'évaluateurs efficaces.

Les nombreuses études théoriques et pratiques, menées dans les années 1970 et 1980 dans ce domaine, ont abouti à définir des classes de grammaires attribuées en fonction des méthodes permettant de leur construire un évaluateur. Le but n'est pas ici de rentrer dans les détails de cette classification, mais de rappeler brièvement les classes principales, des plus générales aux plus particulières.

La classe la plus générale, en terme de pouvoir d'expression, est la classe des grammaires attribuées **non-circulaires** (NC). Ensuite, viennent les grammaires attribuées **fortement non-circulaires** (FNC)[8], puis **doublement non-circulaires** (DNC). Plus restreinte, la classe des grammaires attribuées **l-ordonnées** permet de produire des évaluateurs statiques totalement déterministes, ce qui autorise des optimisations intéressantes. Ces évaluateurs sont basés sur des **séquences de visites** qui respectent un ordre d'évaluation total (cf. section 2.2.4). Théoriquement, le test d'appartenance à la classe l-ordonnée et la construction des ordres totaux sont des problèmes NP-complets [EF82], mais une sous-classe des grammaires attribuées l-ordonnées, la classe des grammaires OAG, peut être caractérisée en un temps polynômial [Kas80].

Par ailleurs, il est possible de transformer toute grammaire attribuée NC en une grammaire attribuée l-ordonnée, au risque d'augmenter la taille de la grammaire de façon exponentielle [Fil83]. Dans le cas des grammaires attribuées FNC, il existe une optimisation de cette transformation qui permet de faire disparaître en pratique ce facteur exponentiel [Par88]. Cette approche, développée et utilisée pour le système FNC-2 [JP89], permet de conserver le pouvoir d'expression de la classe FNC tout en autorisant un faible encombrement mémoire et un temps de traitement raisonnable.

Pour des détails et précisions concernant ces méthodes d'évaluation et ces transformations, le lecteur intéressé est invité à se référer aux travaux rapportés dans [Kas80, DJL84, Eng84, DJL88, Par88].

2.2.4 Mise en œuvre des évaluateurs

Comme je l'ai déjà brièvement dit, les grammaires attribuées l-ordonnées permettent de générer des évaluateurs basés sur les *séquences de visites*. Outre le faible encombrement mémoire qu'autorisent ces évaluateurs, leur forme est particulièrement bien adaptée pour une implantation fonctionnelle.

Plutôt que de donner les résultats théoriques et les algorithmes de cette méthode d'évaluation, je préfère en donner dans cette section l'illustration sur l'exemple d'une grammaire attribuée, avec la séquence de visites qu'il est possible de déterminer à partir de ses graphes de dépendances. Enfin, je donne un évaluateur fonctionnel de cette grammaire attribuée, basé sur ces séquences de visites.

Je considère donc la grammaire attribuée qui accepte un arbre binaire d'entiers en entrée et qui fournit comme résultat un arbre binaire ayant la même structure que l'arbre d'entrée,

8. La classe de ces grammaires attribuées est également connue sous le nom de SNC, de l'anglais *Strongly Non Circular*.

```
aglet bird =
  bird t →
    bird.result = t.tm
    t.h = t.m
  node left right →
    node.m = minimum left.m right.m
    node.tm = node left.tm right.tm
    left.h = node.h
    right.h = node.h
  leaf n →
    leaf.m = n
    leaf.tm = leaf leaf.h
```

FIG. 2.10: *Grammaire attribuée de bird*

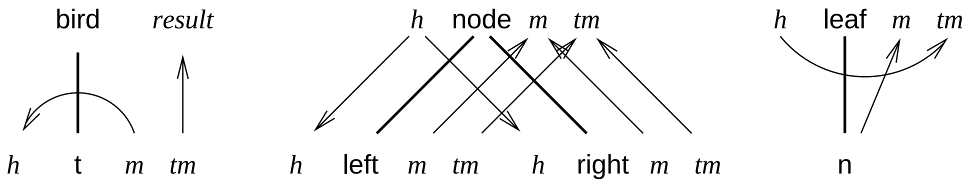

FIG. 2.11: *Graphes de dépendances locales des productions de la grammaire attribuée bird*

mais dont les feuilles sont toutes à la valeur minimale des feuilles de l'arbre d'entrée[9]. Cette grammaire attribuée *bird*, présentée à la figure 2.10 utilise deux attributs synthétisés, m et tm qui calculent respectivement la valeur de la plus petite feuille et l'arbre construit à partir de cette valeur. Elle utilise également un attribut hérité h qui propage la valeur de m, une fois que celle-ci est connue.

Intuitivement, sur chaque nœud de l'arbre, la valeur de m est obtenue en prenant la plus petite des deux valeurs correspondant aux deux sous-arbres (la fonction *minimum* est supposée définie). Une fois calculée dans m pour l'arbre entier, cette plus petite valeur est transmise jusqu'aux feuilles par l'attribut h. À partir de cette plus petite valeur, les feuilles et les nœuds de l'arbre résultat peuvent être construits dans tm de façon isomorphe à l'arbre d'entrée.

Cette petite explication fait bien pressentir que le calcul du résultat de *bird* nécessite plusieurs parcours de l'arbre d'entrée : un parcours montant pour calculer m, qui permet d'initialiser h qui propage cette valeur en descendant le même arbre, puis un dernier parcours montant pour construire le résultat : on parle alors de deux *passes*.

Les graphes de dépendances locales (définition 2.2.2) de chacune des productions sont présentés à la figure 2.11. Elles représentent des dépendances locales à chaque production entre les différentes occurrences d'attributs.

En considérant chacun des non-terminaux de cette grammaire attribuée, c'est-à-dire un non-terminal du type *tree* (comme *node* et *leaf*) et un non-terminal du type de *bird*, les tests de caractérisation FNC et DNC [CFZ82, Par88] permettent de déterminer des familles de

9. Cet exemple est dû à R.S. Bird [Bir84].

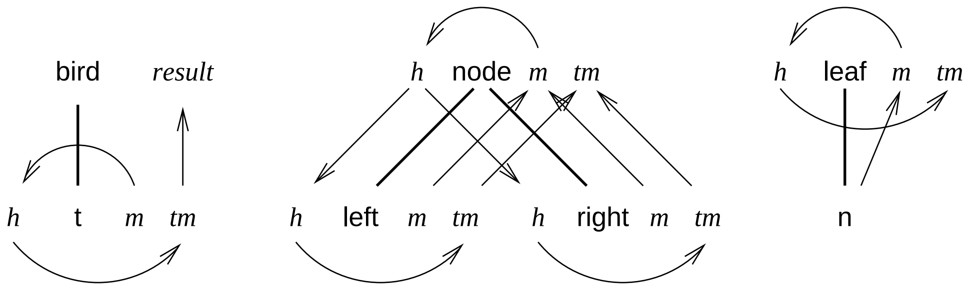

FIG. 2.12: *Graphes de dépendances locales augmentés*

relations d'ordre. Celles-ci peuvent être vues comme des graphes sur les occurrences d'attributs de chaque non-terminal qui donnent un ordre d'évaluation des occurrences d'attributs sur un non-terminal, valable quel que soit la production considérée. Dans le cas de la grammaire attribuée pour *bird*, cet ordre est trivial sur le profil *bird* qui ne possède qu'un seul attribut synthétisé *result*, et sur les non-terminaux du type *tree*, cet ordre est $m \prec h \prec tm$.

À partir de cet ordre, les **graphes de dépendances locales augmentés** sont considérés comme étant les graphes de dépendances locales des productions, augmentés par les relations de l'ordre total. Dans l'exemple de la grammaire attribuée *bird*, les graphes de dépendances locales augmentés de chacune des productions sont présentés à la figure 2.12.

Les grammaires attribuées l-ordonnées, dont *bird* fait partie, vérifient la propriété de non-circularité de ces graphes de dépendances augmentés pour toutes leurs productions. Il est alors possible de générer des **séquences de visites** pour chacune de leurs productions. L'idée intuitive d'une séquence de visites est qu'il est possible d'évaluer un sous-ensemble d'occurrences d'attributs synthétisés sur un nœud d'un arbre en un seul parcours du sous-arbre dont il est le père, à partir du moment où un sous-ensemble des occurrences d'attributs hérités sur ce nœud est déjà évalué. Le principe est alors de partitioner, à partir de l'ordre total, l'ensemble des occurrences d'attributs en sous-ensembles hérités et synthétisés consécutifs.

Dans le cas de la grammaire attribuée *bird*, la première tranche d'hérités est vide puisque le premier attribut de l'ordre total est m, synthétisé. La première visite est donc capable, en parcourant l'arbre, d'évaluer ce premier attribut synthétisé. À la fin de ce premier parcours d'arbre, l'attribut h peut être déterminé et peut servir de paramètre pour le second parcours. La deuxième visite permet donc, à partir du paramètre h évalué (en l'occurrence simplement transmis) sur chaque nœud de l'arbre, de calculer les occurrences de l'attribut synthétisé tm. Le tableau ci-dessous présente les attributs évalués par chacune des visites, pour chacun des constructeurs.

Visite	Première		Deuxième	
Constructeur	Hérités	Synthétisés	Hérités	Synthétisés
bird	$\emptyset$	*result*	$\emptyset$	$\emptyset$
node	$\emptyset$	m	h	tm
leaf	$\emptyset$	m	h	tm

Une séquence de visites est constituée de différentes opérations :

- `begin` i débute la i-ème visite sur le constructeur courant ;

bird t →
  `begin` 1
  `visit` 1 *t*
  `eval` *h t*
  `visit` 2 *t*
  `eval` *result bird*
  `leave` 1

node left right →
  `begin` 1
  `visit` 1 *left*
  `visit` 1 *right*
  `eval` *m node*
  `leave` 1
  `begin` 2
  `eval` *h left*
  `visit` 2 *left*
  `eval` *h right*
  `visit` 2 *right*
  `eval` *tm node*
  `leave` 2

leaf n →
  `begin` 1
  `eval` *m leaf*
  `leave` 1
  `begin` 2
  `eval` *tm leaf*
  `leave` 2

FIG. 2.13: *Séquences de visites générées pour la grammaire attribuée bird*

- `leave` i termine la i-ème visite sur le constructeur courant ;
- `visit` i y appelle la i-ème visite sur le fils spécifié par y ;
- `eval` a y évalue l'attribut $y.a$, en appelant simplement la règle sémantique qui le définit, puisque, d'après l'ordre choisi, toutes les occurrences d'attributs dont il dépend sont disponibles.

La figure 2.13 présente les séquences de visites générées pour la grammaire attribuée *bird*.

Un évaluateur déterminé à partir des séquences de visites est valable quel que soit l'arbre d'entrée fourni et est particulièrement efficace en temps. De plus, il est possible, grâce aux analyses statiques fines autorisées par les grammaires attribuées, en particulier sur la durée de vie des attributs, d'optimiser l'espace occupé par ce genre d'implantation (allocation en pile de certains attributs, non-construction de l'arbre, etc). Pour plus de détails, le lecteur est invité à se référer à [Par88, JP89].

Pour terminer l'étude de cet exemple et cette section sur l'évaluation des grammaires attribuées, la figure 2.14 présente une implantation fonctionnelle naïve de cet évaluateur à base de séquences de visites. Le programme fonctionnel obtenu est à mettre en parallèle avec la version standard [Bir84], telle que celle présentée dans la figure 2.15, en remplaçant les noms de fonctions *visit_bird_1*, *visit_tree_1* et *visit_tree_2* par *bird*, *tmin* et *replace*. Une telle implantation fonctionnelle est par exemple possible directement en *back-end* du système FNC-2 [Le 95].

Par ailleurs, Johnsson s'est également intéressé à cet exemple typique. Dans [Joh87], il donne une méthode permettant de déduire automatiquement, à partir de sa spécification simple en grammaire attribuée, un évaluateur en une passe de cette fonction qui peut être implanté avec un langage paresseux (*lazy*) tel que LML. L'idée de sa transformation est de considérer le résultat d'une grammaire attribuée comme un nouvel attribut, étant une fonction, qui prend (tous) les attributs hérités comme paramètres et qui retourne les attributs synthétisés (dont le résultat) comme un tuple. Dans le cas de l'exemple *bird* tel qu'il est spécifié par

```
let visit_bird_1 t =
  let
    m = visit_tree_1 t
    h = m
  in visit_tree_2 t h
```

```
let visit_tree_1 t = case t of
  node left right →
    let
      m_l = visit_tree_1 left
      m_r = visit_tree_1 right
    in minimum m_l m_r
  leaf n →
    n
```

```
let visit_tree_2 t h = case t of
  node left right →
    let
      h_l = h     h_r = h
      tm_l = visit_tree_2 left h_l
      tm_r = visit_tree_2 right h_r
    in node tm_l tm_r
  leaf n →
    leaf h
```

FIG. 2.14: *Évaluateur fonctionnel à séquences de visites pour la grammaire attribuée bird*

```
let bird t = replace t (tmin t)
```

```
let tmin t = case t of
  node left right →
    minimum (tmin left) (tmin right)
  leaf n → n
```

```
let replace t h = case t of
  node left right →
    node (replace left h) (replace right h)
  leaf n → leaf h
```

FIG. 2.15: *Programme fonctionnel classique pour bird*

```
let bird t = t1 in (t1, m1) = repmin t m1

let repmin t m = case t of
  node left right →
    let
      (t_l, m_l) = repmin left m
      (t_r, m_r) = repmin right m
    in (node t_l t_r, minimum m_l m_r)
  leaf n → (leaf m, n)
```

FIG. 2.16: *Programme fonctionnel paresseux en une passe pour bird*

la grammaire attribuée à la figure 2.10, cette transformation produit comme évaluateur le programme fonctionnel paresseux présenté à la figure 2.16, auquel aboutit également Bird dans [Bir84] et qui peut être évalué en une passe [Joh87].

Encore illustré par ce fameux exemple de Bird, citons également ici les travaux de Pettorossi sur la stratégie de généralisation d'ordre supérieur [PS87]. Cette technique permet d'obtenir un résultat similaire d'évaluation en une passe, sans recourir à l'évaluation paresseuse. L'idée pour cet exemple est de définir, en même temps que le calcul de la valeur minimum de l'arbre, des fonctions d'ordre supérieur permettant de construire le résultat lorsque le minimum global est connu. De manière intuitive, le résultat de cette transformation construit en une seule passe une paire d'objets : le premier objet du couple est une fonction d'ordre supérieur (une fermeture qui *mémorise la structure à construire*) et le second objet est la valeur de feuille minimum. Lorsque la passe est terminée, il suffit alors d'appliquer la fonction calculée dans le premier objet de la paire à la valeur calculée dans le second [PS87].

Sans aller plus loin dans la comparaison, ce résultat peut tout de même être mis en parallèle avec la transformation classique [Knu68, CM79] qui produit, pour toute grammaire attribuée, la grammaire attribuée purement synthétisée d'ordre supérieur correspondante (c'est-à-dire qui possède des attributs synthétisés et des règles sémantiques d'ordre supérieur, mais pas d'attribut hérité).

Cet exemple clôt la présentation de la mise en œuvre des évaluateurs abordée dans ce mémoire. Son but était simplement de donner une idée intuitive de la manière de générer un évaluateur à partir d'une spécification de grammaire attribuée. Néanmoins, pour une formalisation de ces techniques, le lecteur intéressé pourra se référer à [AM91, Paa95]. De plus, dans le chapitre 4, je reviendrai sur la technique de génération des séquences de visites dans le cadre des grammaires attribuées dynamiques.

Chapitre 3

Composition descriptionnelle

3.1 Grammaires attribuées composables

Pour permettre plus de modularité dans l'écriture des grammaires attribuées, Ganzinger et Giegerich ont introduit la notion de *grammaires couplées par attributs* [GG84], que Roussel appelle dans sa thèse les *grammaires attribuées fonctionnelles* [Rou94]. Ces objets ne sont pas une nouvelle forme de grammaires attribuées, mais simplement une nouvelle façon de les interpréter. L'idée est de considérer une grammaire attribuée, non plus comme un *décorateur* qui "attribue" des valeurs sur l'arbre d'entrée, mais comme une véritable fonction qui prend en entrée un arbre et qui en retourne un autre en sortie : $t_2 = GA(t_1)$. Une application complexe peut ainsi être exprimée sous la forme d'une succession de grammaires attribuées s'appelant chacune sur le résultat de la précédente.

Les grammaires couplées par attributs suivent la définition classique des grammaires attribuées, mais elles sont vues comme des fonctions [1] typées d'une grammaire dans une autre [2]. Ainsi, deux grammaires couplées par attributs sont vues comme une grammaire attribuée α qui fait passer d'un arbre de la grammaire $G_1 = (N_1, T_1, P_1, Z_1)$ vers un arbre de la grammaire $G_2 = (N_2, T_2, P_2, Z_2)$. Elle est notée $\alpha : G_1 \to G_2$ et sa syntaxe de base est celle de G_1 ; autrement dit, $\alpha = (G_1, A, F)$ où l'ensemble F des règles sémantiques vérifie des propriétés de typage par rapport à G_2. En particulier, l'attribut synthétisé de la racine de α est du type de Z_2. Pour simplifier les notations, j'appellerai désormais "grammaire attribuée" une paire de grammaires couplées par attributs.

Pour illustrer le principe de composition de deux grammaires attribuées, je commence par prendre un exemple très simple et assez caricatural.

Soit *map_square* : (*list int*) $\to$ (*list int*) la grammaire attribuée qui, à partir d'une liste d'entiers, construit la liste des carrés de ses entiers. Soit *sum* : (*list int*) $\to$ *int* la grammaire attribuée qui calcule la somme des éléments d'une liste. Elles sont présentées à la figure 3.1.

La composition de ces deux grammaires attribuées fournit donc la somme des carrés des éléments d'une liste donnée. Pour cela, elle construit d'abord la liste des carrés et ensuite calcule la somme des éléments de cette liste. La figure 3.2 présente sur un exemple l'effet de la composition de ces deux grammaires attribuées.

1. D'où leur nom selon Roussel.
2. D'où leur nom par Ganzinger et Giegerich.

$$\begin{array}{ll}
\mathit{root} : Z \to L & \mathit{root} : Z \to L \\
\quad Z.z = L.m & \quad Z.z = L.s \\
\mathit{cons} : L_0 \to e\ L_1 & \mathit{cons} : L_0 \to e\ L_1 \\
\quad L_0.m = \mathit{cons}\ (\mathit{square}\ e)\ L_1.m & \quad L_0.s = e + L_1.s \\
\mathit{nil} : L \to \quad L.m = \mathit{nil} & \mathit{nil} : L \to \quad L.s = 0
\end{array}$$

FIG. 3.1: *Les grammaires attribuées map_square et sum*

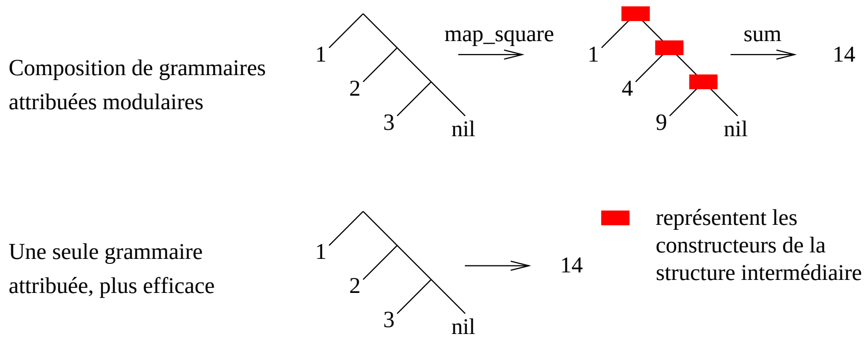

FIG. 3.2: *Modularité versus efficacité*

Cette approche compositionnelle permet une écriture modulaire d'une application. Elle peut également faciliter la spécification d'un problème, ainsi que la réutilisation de grammaires attribuées. Elle oblige cependant à construire des structures intermédiaires qui ne seraient pas utiles dans une grammaire attribuée qui décrirait, en une seule fois, l'ensemble de l'application. De ce fait, l'efficacité de l'évaluateur associé peut être sérieusement pénalisée par ces constructions. Par exemple, il est clair que connaître la somme des carrés des éléments d'une liste ne nécessite pas la construction de la liste des carrés, comme suggéré par la figure 3.2.

Ganzinger et Giegerich ont introduit une construction qui permet de transformer une suite de grammaires attribuées en une unique grammaire attribuée qui a la même sémantique que la suite de grammaires attribuées de départ [GG84]. Cette construction porte le nom de **composition descriptionnelle** ; elle a été plus récemment étudiée et développée par Roussel [Rou94] sous le nom de *méta-composition*.

3.2 Composition de spécifications

3.2.1 L'idée

L'idée de la composition descriptionnelle est simple. Soient deux grammaires attribuées $\alpha : G_1 \to G_2$ et $\beta : G_2 \to G_3$. Le but de la composition descriptionnelle est d'éliminer la construction de l'arbre intermédiaire de G_2. C'est également le but des transformations connues sous le nom de *déforestation*, que je traiterai plus longuement à partir du chapitre 5.

D'une part, les fonctions de construction de cet arbre intermédiaire sont connues dans α et, d'autre part, grâce à β, le schéma de calcul des occurrences d'attributs associées à ces constructions est connu. Il suffit donc de remplacer dans α, avec un renommage, les

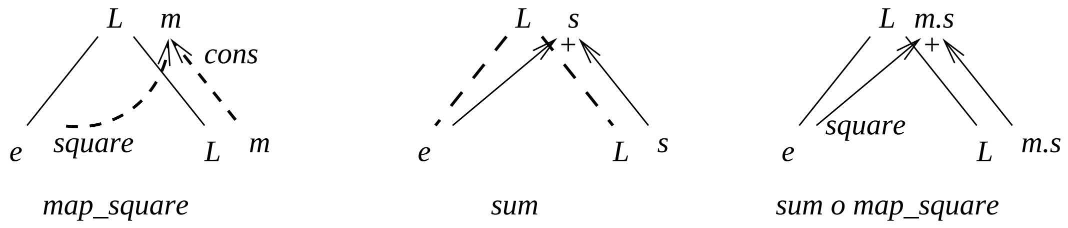

FIG. 3.3: *Projection du + de sum sur le cons de map_square*

constructions de G_2 par leurs schémas de calculs associés dans β. L'effet sur la grammaire attribuée résultant de cette transformation est d'effectuer, sur les productions où auraient été produits des morceaux d'arbre intermédiaire, les calculs qui auraient été effectués sur ces morceaux d'arbre. La figure 3.3 montre l'effet de cette transformation pour les productions *cons* de *map_square* et de *sum* de l'exemple précédent. Comme dans la production *cons* de *map_square*, l'occurrence d'attribut $L_0.m$ est définie par un *cons*, il s'agit d'un constructeur de la structure intermédiaire. Lors de l'application de *sum* sur cette structure intermédiaire, le calcul de $L_0.s$ sera effectué sur ce même *cons*. Ce dernier calcul peut donc être projeté, moyennant un renommage et une identification des termes, à l'endroit où devait être construit ce *cons*, le remplaçant donc par le calcul de la somme du carré sur la production initiale.

Je vais maintenant décrire plus formellement l'algorithme de base de la composition descriptionnelle, tel qu'il est présenté dans [Rou94].

3.2.2 L'algorithme de composition descriptionnelle

Soit donc une première grammaire attribuée $\alpha : G_1 \rightarrow G_2$ et une deuxième $\beta : G_2 \rightarrow G_3$; la composition descriptionnelle de ces deux grammaires attribuées produit une grammaire attribuée $\beta \circ \alpha = \gamma : G_1 \rightarrow G_3$.

Dans la présentation qui suit, afin qu'il soit plus facile de différencier les différents éléments des trois grammaires attribuées, le nom de la grammaire attribuée considérée est placé en indice. Par exemple l'ensemble des attributs synthétisés associés au non-terminal X dans la grammaire attribuée β est noté $S_\beta(X)$. Cette notation sera utilisée à chaque fois que plusieurs grammaires attribuées interviennent en même temps dans une expression. Par ailleurs, la formalisation des déclarations de nouveaux attributs ou de nouvelles règles sémantiques est représentée par $\hookrightarrow$.

Création de la syntaxe de base La syntaxe de base de la grammaire attribuée γ est la même que celle de α, c'est-à-dire G_1. En effet, γ doit prendre les mêmes arbres d'entrée que α, et doit donc conserver les ensembles de terminaux, de non-terminaux, de productions et l'axiome de α.

Déclaration des nouveaux attributs

- Pour chaque attribut de α ne participant pas directement à la construction d'une partie de la structure de G_2, un attribut de même nom, de même classe et de même type est déclaré dans γ.

- Pour chaque attribut synthétisé a de type τ, porté par un non terminal X dans α, et pour chaque attribut b, porté par un non-terminal Y de type τ dans β, un attribut $a.b$ est déclaré sur le non-terminal X dans γ ; il est de même type et de même classe que b.

$$\begin{array}{l}\forall a \in S_\alpha(X)\\ \quad \forall b \in S_\beta(Y) \mid type(Y) = type(a)\\ \qquad \hookrightarrow a.b \in S_\gamma(X) \mid type(a.b) = type(b)\\ \quad \forall b \in H_\beta(Y) \mid type(Y) = type(a)\\ \qquad \hookrightarrow a.b \in H_\gamma(X) \mid type(a.b) = type(b)\end{array}$$

- Pour chaque attribut hérité a de type τ, porté par un non terminal X dans α, et pour chaque attribut b, porté par un non-terminal Y de type τ dans β, un attribut $a.b$ est déclaré sur le non-terminal X dans γ ; il est de même type que b, mais de classe opposée.

$$\begin{array}{l}\forall a \in H_\alpha(X)\\ \quad \forall b \in S_\beta(Y) \mid type(Y) = type(a)\\ \qquad \hookrightarrow a.b \in H_\gamma(X) \mid type(a.b) = type(b)\\ \quad \forall b \in H_\beta(Y) \mid type(Y) = type(a)\\ \qquad \hookrightarrow a.b \in S_\gamma(X) \mid type(a.b) = type(b)\end{array}$$

Création des nouvelles règles sémantiques

- Chaque règle sémantique de α sur une production p qui **ne construit pas** de partie de la structure de type G_2, mais effectue simplement un calcul, est conservée telle quelle, soit :

$$\begin{array}{l}\forall\ r_{p,a_0,\rho(0)} : X_{\rho(0)}.a_0 = f_{p,a_0,\rho(0)}\ X_{\rho(1)}.a_1, \cdots, X_{\rho(n)}.a_n) \in F_\alpha(p)\\ \quad \text{tel que } f_{p,a_0,\rho(0)} \text{ est un calcul}\\ \qquad \hookrightarrow r_{p,a_0,\rho(0)} \in F_\gamma(p)\end{array}$$

- Pour chaque règle sémantique r de α sur une production p qui **construit** une partie de la structure de G_2 (avec un constructeur de production p_2), la projection des règles sémantiques associées à p_2 sur la règle sémantique r donne lieu à de nouvelles règles sémantiques, soit :

$$\begin{array}{l}\forall\ r_{p,a_0,\rho(0)} : X_{\rho(0)}.a_0 = (f_{p,a_0,\rho(0)}\ X_{\rho(1)}.a_1\ \cdots X_{\rho(n)}.a_n) \in F_\alpha(p)\\ \quad \text{tel que } f_{p,a_0,\rho(0)} \text{ est une construction } p_2 \text{ de } G_2\\ \quad \forall\ r_{p_2,b_0,\delta(0)} : X_{\delta(0)}.b_0 = (f_{p_2,b_0,\delta(0)}\ X_{\delta(1)}.b_1\ \cdots X_{\delta(q)}.b_q) \in F_\beta(p_2)\\ \qquad \hookrightarrow r_{p,a_{\delta(0)}.b_0,\rho\circ\delta(0)} \in F_\gamma(p) \text{ avec}\\ r_{p,a_{\delta(0)}.b_0,\rho\circ\delta(0)} : X_{\rho\circ\delta(0)}.a_{\delta(0)}.b_0 = (f_{p_2,b_0,\delta(0)}\ X_{\rho\circ\delta(1)}.a_{\delta(1)}.b_1 \cdots X_{\rho\circ\delta(q)}.a_{\delta(q)}.b_q)\\ \quad \text{où } \rho\circ\delta(i) = \rho(\delta(i))\end{array}$$

- Pour chaque règle sémantique de α sur une production p qui propage simplement une valeur d'attribut (règle de copie), une règle de copie correspondante dans γ est générée ; elle prend en compte les noms des nouveaux attributs et leur classe, soit :

$$\begin{array}{l}\forall\ r_{p,a_0,\rho(0)} : X_{\rho(0)}.a_0 = X_{\rho(1)}.a_1 \in F_\alpha(p)\\ \quad \forall b \in S_\beta(Y) \mid type(Y) = type(a_0)\\ \qquad \hookrightarrow r_{p,a_0.b,\rho(0)} \in F_\gamma(p) \text{ avec}\\ \qquad\quad r_{p,a_0.b,\rho(0)} : X_{\rho(0)}.a_0.b = X_{\rho(1)}.a_1.b\\ \quad \forall b \in H_\beta(Y) \mid type(Y) = type(a_0)\\ \qquad \hookrightarrow r_{p,a_1.b,\rho(1)} \in F_\gamma(p) \text{ avec}\\ \qquad\quad r_{p,a_1.b,\rho(1)} : X_{\rho(1)}.a_1.b = X_{\rho(0)}.a_0.b\end{array}$$

3.2.3 Des exemples

Pour illustrer l'algorithme de composition descriptionnelle, je prends d'abord un exemple simple où les deux grammaires attribuées ne possèdent qu'un seul attribut synthétisé chacune. Il s'agit de l'exemple présenté en introduction de ce chapitre, concernant la composition descriptionnelle $sum \circ map\_square$ (figure 3.1).

Par rapport à l'algorithme de composition descriptionnelle présentée à la section 3.2.2 précédente, $map\_square : (list\ int) \to (list\ int)$ joue le rôle de la grammaire attribuée $\alpha : G_1 \to G_2$ et $sum : (list\ int) \to int$ celui de la grammaire attribuée $\beta : G_2 \to G_3$. Par simplicité, la grammaire attribuée $sum \circ map\_square : (list\ int) \to int$ résultant de la composition descriptionnelle sera appelée γ.

Sa syntaxe de base est donc la même que celle de *map_square*, à savoir le type des listes.

Les attributs synthétisés de *map_square* sont z sur Z et m sur L. Pour chacun d'entre eux, de nouveaux attributs sont déclarés dans γ, formés grâce aux attributs portés par les non-terminaux de *sum* qui sont du type correspondant : sont donc créés les attributs $z.z$ sur Z et $m.s$ sur L.

Pour les règles sémantiques, le problème est assez simple puisqu'il n'en existe qu'une seule pour chaque production de chaque grammaire attribuée. La règle sémantique $\diamondsuit$ de *map_square* sur la production *cons* est

$$L_0.m = cons\ (square\ \ e)\ L_1.m \qquad \diamondsuit$$

Comme elle produit une partie de la structure acceptée en entrée par *sum* avec la construction *cons*, la règle associée à ce constructeur dans *sum*, à savoir

$$L_0.s = e + L_1.s$$

doit être projetée sur $\diamondsuit$ pour produire la nouvelle règle sémantique associée à *cons* dans γ. Cette projection associe les occurrences d'attributs et les termes $L_0.s$ à $L_0.m$, e à $(square\ \ e)$ et $L_1.s$ à $L_1.m$ et produit :

$$L_0.m.s = (square\ \ e) + L_1.m.s$$

De la même façon, la règle sémantique de *map_square* pour la production *nil* (qui produit un *nil*), sur laquelle est projetée la règle sémantique de *sum* pour la production *nil*, permet de générer la nouvelle règle sémantique de γ pour *nil* :

$$L.m.s = 0$$

Pour le constructeur *root*, la règle sémantique définissant l'attribut $Z.z$ en fonction de $L.m$ permet de générer la nouvelle règle dans γ :

$$Z.z.z = L.m.s$$

L'ensemble de la grammaire attribuée ainsi obtenue par composition descriptionnelle est présenté à la figure 3.4.

Cette unique grammaire attribuée ne contient alors aucun constructeur de liste. Son évaluation est donc nettement plus efficace que l'application successive des deux grammaires attribuées, comme suggéré à la figure 3.2 dans l'introduction de ce chapitre. Elle est *déforestée*, au sens où elle ne construit plus aucune structure intermédiaire.

$$\begin{array}{l} \mathit{root} : Z \to L \\ \quad Z.z.z = L.m.s \\ \mathit{cons} : L_0 \to e\ L_1 \\ \quad L_0.m.s = (\mathit{square}\ e) + L_1.m.s \\ \mathit{nil} : L \to \\ \quad L.m.s = 0 \end{array}$$

FIG. 3.4: *La composition descriptionnelle sum ∘ map_square*

$$\begin{array}{ll} \mathit{root} : Z \to T & \\ \quad Z.z = T.f \quad \heartsuit & \mathit{root} : Z \to L \\ \quad T.l = \mathit{nil} \quad \clubsuit & \quad Z.z = L.r \\ \mathit{node} : T_0 \to T_1\ T_2 & \quad L.h = \mathit{nil} \\ \quad T_0.f = T_1.f & \mathit{cons} : L_0 \to e\ L_1 \\ \quad T_1.l = T_2.f & \quad L_0.r = L_1.r \\ \quad T_2.l = T_0.l & \quad L_1.h = \mathit{cons}\ e\ L_0.h \\ \mathit{leaf} : T \to e & \mathit{nil} : L \to \\ \quad T.f = \mathit{cons}\ e\ T.l \quad \spadesuit & \quad L.r = L.h \end{array}$$

FIG. 3.5: *Les grammaires attribuées flatten et reverse*

Un exemple un peu plus complexe, qui met en œuvre des grammaires attribuées possédant toutes les deux des attributs synthétisés et des attributs hérités, est celui de l'inversion de la liste des feuilles d'un arbre binaire. Je considère donc les grammaires attribuées *flatten* et *reverse* telles qu'elles sont présentées dans la figure 3.5.

Pour *flatten*, l'attribut hérité l initialisé à *nil* dans le profil de la grammaire attribuée est propagé vers le fils droit de chaque nœud de l'arbre. La valeur de la première feuille trouvée dans cette descente à droite dans la structure de l'arbre est soumise à un *cons* avec l'attribut l ainsi propagé, définissant l'occurrence de l'attribut synthétisé f sur cette feuille. L'attribut hérité l sur chaque fils gauche d'un nœud est défini par la valeur de l'attribut synthétisé f du fils droit du même nœud. Ainsi, la liste des feuilles de l'arbre, construite à partir de *nil* de la droite vers la gauche, est synthétisée des fils gauches vers leurs pères jusqu'à la racine de l'arbre. Le croquis à gauche de la figure 3.6 illustre ce parcours pour un exemple d'arbre binaire.

Pour *reverse*, la liste inversée est construite en descendant dans la liste analysée, à partir de l'attribut hérité h initialisé à *nil* dans le profil de la grammaire attribuée. La valeur de h sur chaque sous-liste est définie par un *cons* entre l'élément associé au nœud courant et la valeur h sur ce nœud. En arrivant sur le *nil*, l'attribut synthétisé r reçoit la liste inversée ainsi construite et la propage jusqu'à la racine. Le croquis à droite de la figure 3.6 illustre cette construction pour un exemple de liste.

Je détaille maintenant les étapes de la composition descriptionnelle $\gamma = \mathit{reverse} \circ \mathit{flatten}$: la déclaration des nouveaux attributs et la génération des nouvelles règles sémantiques de γ, dont la syntaxe de base est la même que celle de *flatten*, à savoir celle des arbres binaires.

Pour l'attribut synthétisé f de *flatten*, deux nouveaux attributs sont créés dans γ : $f.r$ synthétisé et $f.h$ hérité. Pour l'attribut hérité l de *flatten*, deux nouveaux attributs sont créés

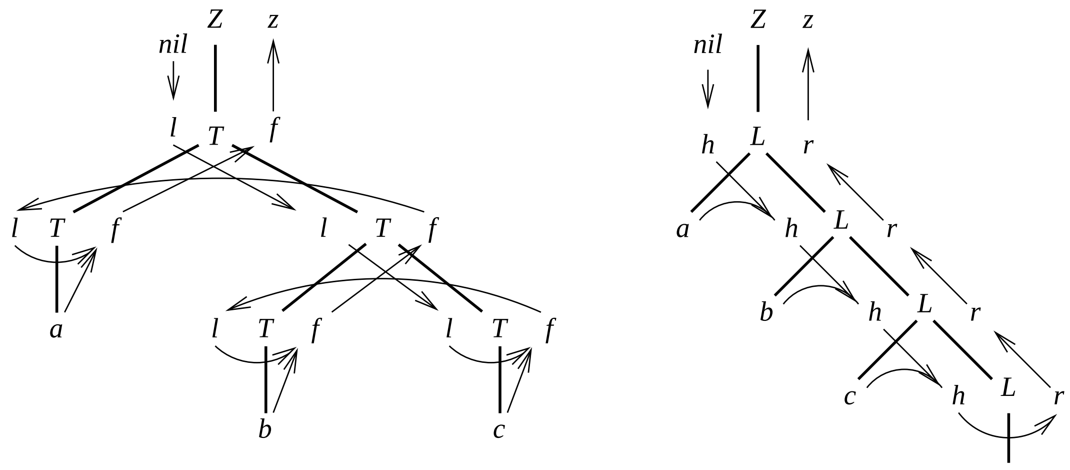

FIG. 3.6: *Illustrations sur des exemples des grammaires attribuées flatten et reverse*

dans γ : $l.r$ hérité et $l.h$ synthétisé. Pour l'attribut z de *flatten*, l'attribut $z.z$ est créé dans γ.

La règle sémantique ♣ (resp. ♠) de *flatten* construit par *nil* (resp. *cons*) une partie de la structure acceptée en entrée par *reverse* et donne lieu à la projection des règles sémantiques associées au constructeur *nil* (resp. *cons*) dans *reverse*. Pour ♣, cette projection produit la nouvelle règle de γ pour la production *root* :

$$T.l.r = T.l.h$$

Pour ♠, les règles produites dans γ pour la production *leaf* sont les suivantes :

$$\begin{aligned} T.f.r &= T.l.r \\ T.l.h &= cons\ e\ \ T.f.h \end{aligned}$$

Par ailleurs, la règle sémantique ♡ de *flatten* dans la production *root* qui définit $Z.z$ en fonction de $T.f$ n'est pas une véritable identité. En fait, il s'agit d'une construction abstraite qui devrait normalement être notée $Z.z = root\ \ T.f$. Par simplicité, et pour éviter de surcharger les exemples, j'ai laissé jusqu'ici de côté cette précision, mais c'est ce constructeur de racine qui permet comme pour ♣ et pour ♠ de générer la nouvelle règle sémantique de γ pour la production *root* :

$$T.f.h = nil$$

Ensuite, les règles sémantiques de *flatten* de propagation de valeur par simple recopie d'une occurrence d'attribut dans une autre permettent de produire les dernières règles sémantiques de γ pour la production *node*. La grammaire attribuée obtenue par cette composition descriptionnelle est présentée à la figure 3.7. Cette grammaire attribuée contient de nombreuses règles de copies d'occurrences d'attribut, mais elle ne construit plus de liste intermédiaire. Pour illustrer son comportement, la figure 3.8 présente le graphe de dépendance complet de cette grammaire attribuée sur un exemple d'arbre binaire.

En terme de "passe" d'un programme fonctionnel, la passe représentée en trait plein correspond à la fonction *revflat1 t nil* donnée dans l'introduction (figure 1.6), tandis que la passe représentée en pointillés correspond à la fonction *revflat2 t l* de la même figure, où l est le résultat de la première passe.

$$
\begin{array}{l}
root : Z \rightarrow T \\
\quad Z.z.z = T.f.r \\
\quad T.f.h = nil \\
\quad T.l.r = T.l.h \\
node : T_0 \rightarrow T_1 \ \ T_2 \\
\quad T_0.f.r = T_1.f.r \\
\quad T_1.f.h = T_0.f.h \\
\quad T_1.l.r = T_2.f.r \\
\quad T_2.f.h = T_1.l.h \\
\quad T_2.l.r = T_0.l.r \\
\quad T_0.l.h = T_2.l.h \\
leaf : T \rightarrow e \\
\quad T.f.r = T.l.r \\
\quad T.l.h = cons \ \ e \ \ T.f.h
\end{array}
$$

FIG. 3.7: *La composition descriptionnelle reverse* $\circ$ *flatten*

Les constructions de liste que cette grammaire attribuée contient sont uniquement celles qui produisent le résultat final. Là aussi, la composition descriptionnelle a *déforesté* la composition des deux grammaires attribuées initiales.

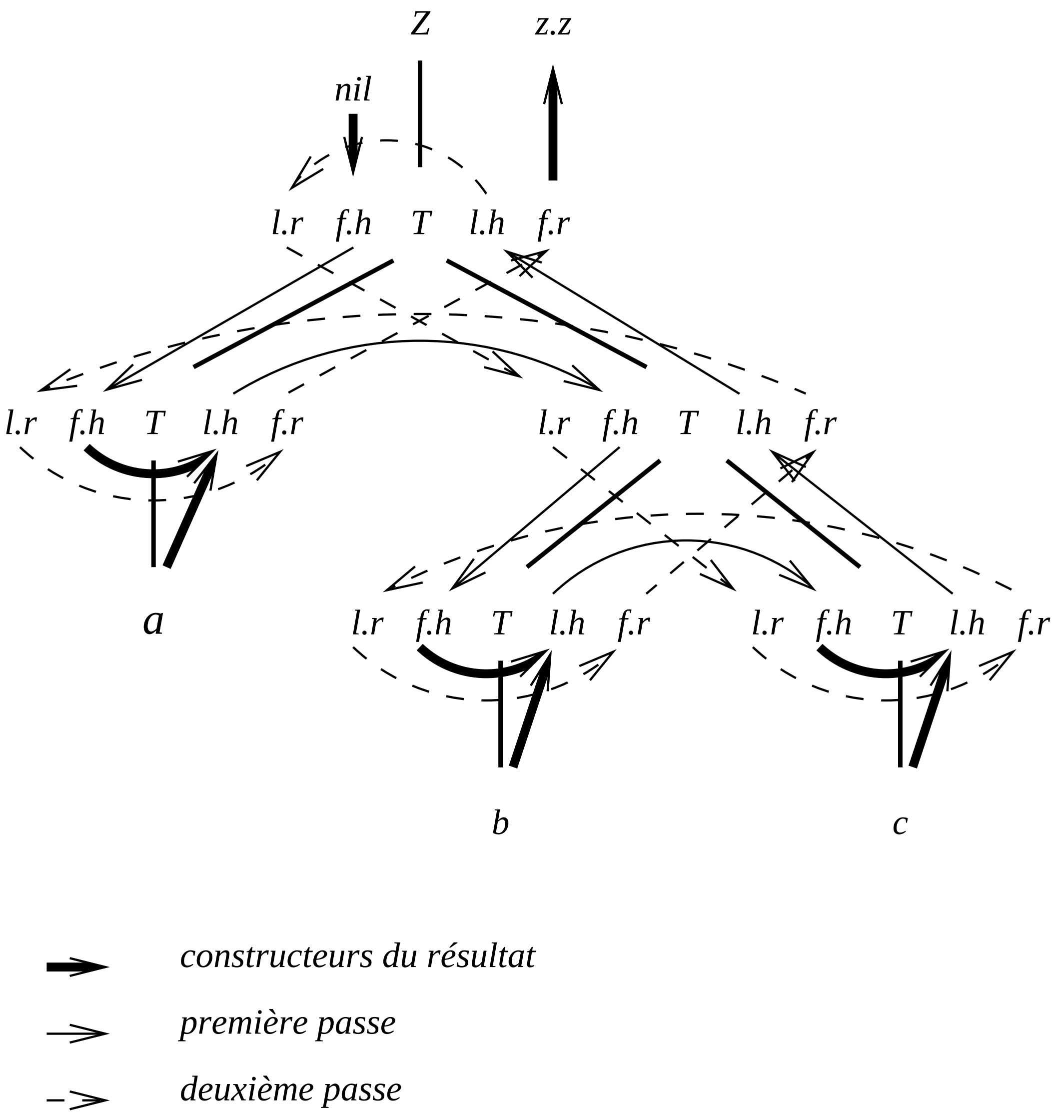

FIG. 3.8: *reverse* ∘ *flatten sur un exemple*

Chapitre 4

Grammaires attribuées dynamiques

4.1 Introduction et présentation du problème

Les chapitres précédents ont brièvement présenté le formalisme des grammaires attribuées et le type de transformation qu'il autorise. Je vais ici présenter les grammaires attribuées dynamiques, une extension au formalisme classique qui permet de représenter une plus large classe de programmes sans bouleverser les techniques d'évaluations classiques. Le but de cette extension est d'obtenir une expressivité permettant d'appliquer des techniques développées pour les grammaires attribuées dans un cadre plus large, facilitant ainsi leur comparaison avec les techniques fonctionnelles.

Bien qu'elles aient été introduites il y a trente ans et qu'elles aient été le sujet de nombreuses études [Paa95], les grammaires attribuées ne sont pas utilisées aussi largement qu'elles le pourraient. En effet, elles sont trop souvent considérées comme une partie inhérente d'un processus de compilation. Ceci est principalement dû au fait que peu de domaines en dehors de la compilation disposent naturellement en entrée d'un arbre qui dirige les calculs. Sans arbre, le modèle *classique* des grammaires attribuées est inutilisable: il est par exemple impossible de spécifier la fonction factorielle avec une grammaire attribuée classique.

Plusieurs travaux ont tenté de répondre à ce problème en proposant diverses extensions au formalisme original des grammaires attribuées, comme par exemple les grammaires attribuées circulaires [Far86], les grammaires multi-attribuées [Att89], les grammaires attribuées d'ordre supérieur [SV91] ou plus récemment les grammaires attribuées conditionnelles [Boy96].

Les grammaires attribuées dynamiques [PDRJ95b, PDRJ96, PRJD96a, PRJD96b] que je présente dans ce chapitre ont des similarités avec les deux dernières extensions citées : comme pour les grammaires attribuées d'ordre supérieur, l'arbre des calculs n'est pas nécessairement isomorphe à l'arbre d'entrée et comme pour les grammaires attribuées conditionnelles, la valeur des attributs peut influencer le choix d'une règle sémantique. Cependant, les grammaires attribuées dynamiques diffèrent des autres extensions par deux points fondamentaux. Premièrement, pour les grammaires attribuées dynamiques, la notion de grammaire n'implique pas nécessairement l'existence (physique) d'un arbre : en fait, les évaluateurs peuvent s'en passer. Deuxièmement, leur technique d'implantation est une simple dérivation des techniques traditionnelles d'évaluation par séquences de visites et ne nécessite la construction ou l'ajout d'aucun fragment d'arbre supplémentaire.

L'idée est de voir la grammaire sous-jacente à une grammaire attribuée comme la grammaire décrivant tous les *arbres d'appels* pour un programme fonctionnel donné, ou tous les *arbres de preuves* pour un programme logique : cette grammaire décrit précisément les différents *flots de contrôle* possibles. Dans ce contexte, une production décrit un schéma de récursion élémentaire (c'est le flot de contrôle) [CFZ82], tandis que les règles sémantiques décrivent les calculs associés à ce schéma (flot de données).

Il faut remarquer que tous les résultats théoriques et pratiques sur les grammaires attribuées sont basés *uniquement* sur l'abstraction du flot de contrôle induit par la grammaire et non sur la façon d'obtenir une instance de cette grammaire au moment de l'exécution. Ceci est vrai en particulier pour les algorithmes de construction d'évaluateurs efficaces pour les différentes classes de grammaires attribuées et pour les méthodes globales d'analyse statique qui sont spécifiques aux grammaires attribuées [AM91, Jou92, Paa95].

Deux notions vont donc être introduites pour formaliser et concrétiser ces idées intuitives :

- les *couples de grammaires* permettent de décrire des schémas de récursion indépendamment de toute structure de donnée physique. Ils permettent également d'exprimer des combinaisons différentes d'éléments d'une structure physique. Un couple de grammaires définit une association entre une grammaire dynamique et une grammaire concrète (éventuellement vide).

- les *grammaires attribuées dynamiques* permettent à une valeur d'attribut d'influencer le flot de contrôle en choisissant une production dynamique plutôt qu'une autre. La nouvelle notion de *blocs de règles sémantiques* définit les arbres de décision correspondant aux productions et à leurs règles sémantiques associées.

Les grammaires attribuées dynamiques contribuent ainsi à rendre aux grammaires attribuées leur puissance d'expression intrinsèque, au sens où elles permettent d'exprimer une classe de programmes plus large, sans remettre fondamentalement en cause les techniques d'évaluations classiques. L'expressivité des grammaires attribuées dynamiques ainsi obtenue est comparable à celle d'un langage du premier ordre, avec un parfum fonctionnel dû à la propriété d'affectation unique et tout en conservant le caractère déclaratif marqué des grammaires attribuées qui peut autoriser l'application des techniques d'analyses et de transformations qui leurs sont propres.

Pour montrer plus facilement que les grammaires attribuées dynamiques peuvent s'évaluer par les techniques classiques des grammaires attribuées, le formalisme utilisé dans la suite de ce chapitre sera plus proche de celui des grammaires attribuées classiques (c'est-à-dire nommage des productions, non-terminal gauche → non-terminaux droits, racines des grammaires sous-jacentes, etc.) que du formalisme fonctionnel pour les grammaires attribuées introduit à la section 2.1.2 (c'est-à-dire patterns de constructeurs, profil de grammaire attribuée, etc.). Néanmoins, avant de détailler plus précisément dans la section suivante les objets et les notations utiles aux grammaires attribuées dynamiques, je donne à la figure 4.1 une représentation de la fonction factorielle, dans une pseudo-syntaxe qui n'est pas définie ici. Le but est uniquement de motiver les travaux décrits dans ce chapitre, d'en donner une idée intuitive au lecteur et de convaincre de la possibilité de représenter l'exemple typique de la fonction factorielle par une grammaire attribuée dynamique.

`daglet` *fact* =	Grammaire attribuée dynamique
fact dynfact n →	Le profil fait référence
fact.result = *dynfact.f*	à un non-terminal dynamique (dynfact)
dynfact.h = *n*	Le paramètre n devient un attribut h
⟨ (*dynfact.h* >= *1*),	dont la valeur détermine l'application
⟨ *dynfact rec* →	d'une production dynamique (récursive)
dynfact.f = *dynfact.h* ∗ *rec.f*	
rec.h = *dynfact.h* − *1* ⟩	
⟨ *dynfact* →	ou de l'autre (terminale)
dynfact.f = *1* ⟩ ⟩	

FIG. 4.1: *La fonction factorielle dans une pseudo-syntaxe de grammaire attribuée dynamique*

4.2 Grammaires attribuées dynamiques

Dans les définitions des grammaires attribuées données au chapitre 2, il peut y avoir une ambiguïté dans l'utilisation du symbole X_i. Dans une définition de CFG, il représente un non-terminal tandis que dans une grammaire attribuée, il représente à la fois l'occurrence d'un non-terminal (libellé par sa position dans la production) et le non-terminal lui même (en tant que type). Cependant, la position d'un nom dans une production n'est importante que s'il s'agit de X_0, ou bien pour distinguer deux occurrences d'un non-terminal et de leur type. Je considérerai donc ici une production comme un ensemble de noms distincts (avec un nom particulier pour la partie gauche de la production), chacun associé à un type.

Définition 4.2.1 (Production) *Soit $\mathcal{V}$ un ensemble universel fini de noms de variables. Une production $p : X_0 \to X_1 \dots X_n$ dans une CFG est un tuple $((X_0, \mathcal{V}_p), type_p)$ dans lequel :*

i. $\mathcal{V}_p = \{X_1, X_2, \dots, X_n\} \subset \mathcal{V}$, *avec* $n = Card(\mathcal{V}_p)$ *et* $X_0 \in \mathcal{V} - \mathcal{V}_p$ *;*

ii. $type_p : \mathcal{V}_p^{\oplus} \to N \cup T$, *où* $\mathcal{V}_p^{\oplus} = \{X_0\} \cup \mathcal{V}_p$ *est une fonction qui associe à chaque nom un unique type dans l'ensemble des terminaux et des non-terminaux, tels que* $type_p(X_0) \in N$.

□

Dans la suite de ce chapitre, j'utiliserai pour une production p la plus claire des notations en fonction du contexte, entre $p : X_0 \to X_1 \dots X_n$ et $((X_0, \mathcal{V}_p), type_p)$.

Dans ce chapitre, j'utiliserai les notations usuelles relatives aux productions :

$$LHS(p) = X_0 \quad \text{et} \quad RHS(p) = \mathcal{V}_p$$

La base d'une grammaire attribuée dynamique est donc la grammaire décrivant le flot de contrôle (schéma de récursion) de l'application représentée. Ce flot de contrôle peut dépendre des valeurs des attributs, mais aussi également de la forme d'un arbre physique, qui doit alors être un paramètre particulier de l'évaluateur. Il faut donc non seulement faire la différence, mais de plus établir une relation entre la grammaire qui décrit la structure concrète et celle qui décrit le schéma de calcul et qui, d'une certaine façon, "contient" la première. C'est l'intuition de la notion de *couple de grammaires*. La définition 4.2.2 est d'abord donnée formellement, puis commentée dans le paragraphe suivant.

Définition 4.2.2 (Couple de grammaires) *Un* ***couple de grammaires*** *est un triplet* $G = (G_d, G_c, \mathcal{C}oncrete)$ *formé de deux grammaires indépendantes du contextes* G_d *et* G_c *et d'une fonction* $\mathcal{C}oncrete$*, où*

$G_d = (N_d, T_d, Z_d, P_d)$ *est appelée la grammaire* ***dynamique****,*
$G_c = (N_c, T_c, Z_c, P_c)$ *est appelée la grammaire* ***concrète*** *et où*
$\mathcal{C}oncrete : P_d \times \mathcal{V} \rightarrow (P_c \times \mathcal{V}) \cup \{\perp\}$,

telles qu'elles vérifient :

1. $N_c \subseteq N_d$;
   $T_d = T_c$;
   Si G_c *n'est pas vide*[1] *alors* $Z_d = Z_c$.
2. $\forall p_d \in P_d$, *on a :*
   - *i.* $\forall X \in \mathcal{V}^{\oplus}_{p_d}$, $type_{p_d}(X) \in (N_d - N_c) \Rightarrow \mathcal{C}oncrete(p_d, X) = \perp$;
   - *ii.* $type_{p_d}(LHS(p_d)) \in (N_d - N_c)$
     $\Rightarrow \forall X \in RHS(p_d)$, $type_{p_d}(X) \in (N_d - N_c)$;
   - *iii.* $type_{p_d}(LHS(p_d)) \in N_c \Rightarrow \exists!\ p_c \in P_c$ *tel que :*
     - $\mathcal{C}oncrete(p_d, LHS(p_d)) = (p_c, LHS(p_c))$ *et* $type_{p_d}(LHS(p_d)) = type_{p_c}(LHS(p_c))$;
     - $\forall X \in RHS(p_d)$, $type_{p_d}(X) \in N_c \Rightarrow \exists Y \in \mathcal{V}^{\oplus}_{p_c}$ *tel que* $\mathcal{C}oncrete(p_d, X) = (p_c, Y)$ *et* $type_{p_d}(X) = type_{p_c}(Y)$.
3. $\forall p, q \in P_d$ *tel que* $type_p(LHS(p)) = type_q(LHS(q))$ *et* $\mathcal{C}oncrete(p, LHS(p)) = \mathcal{C}oncrete(q, LHS(q))$, *on a :*
   - *i.* $LHS(p) = LHS(q)$;
   - *ii.* $\forall X \in \mathcal{V}_p \cap \mathcal{V}_q, type_p(X) = type_q(X)$;
   - *iii.* $\forall X \in \mathcal{V}_p \cap \mathcal{V}_q, \mathcal{C}oncrete(p, X) = \mathcal{C}oncrete(q, X)$.

□

Avec les contraintes ci-dessus, la fonction $\mathcal{C}oncrete$, qui est initialement définie pour un nom dans une production, peut être étendue sans ambiguïté aux productions p_d de P_d de la façon suivante :

$$\mathcal{C}oncrete(p_d) = \begin{cases} p_c & si \quad \mathcal{C}oncrete(p_d, LHS(P_d)) = (p_c, LHS(p_c)) \\ \perp & si \quad \mathcal{C}oncrete(p_d, LHS(P_d)) = \perp \end{cases}$$

Dans la définition 4.2.2, G_d et G_c représentent respectivement les grammaires *dynamique* et *concrète* ; la fonction $\mathcal{C}oncrete$ fournit la production (ou le nom) concrète correspondant à une production (ou un nom) dynamique donnée, c'est-à-dire un arbre (ou un nœud) physique.

1. Dans les cas de programmes sans aucune structure, tels que la fonction factorielle, G_c peut être vide et $\mathcal{C}oncrete$ fait correspondre $\perp$ à tous les éléments (cf. productions dynamiques récursive et terminale de *dynfact* dans la figure 4.1).

Production concrète $double \in P_c$

$double : lhs \rightarrow rhs$

Production dynamique $double_{dyn} \in P_d$

$double_{dyn} : simple \rightarrow first\ second$

avec

$Concrete(double_{dyn}) = double$

$Concrete(double_{dyn}, simple) = (double, lhs)$

$Concrete(double_{dyn}, first) = Concrete(double_{dyn}, second) = (double, rhs)$

FIG. 4.2: *Partie d'un couple de grammaires pour la fonction* ***double***

Lorsque la valeur fournie par cette fonction est $\perp$ (indéfini), cela signifie que l'argument est un objet purement dynamique, ou "abstrait" ; il correspond à un schéma de récursion pur.

La condition *2.i* de la définition 4.2.2 signifie que dans une production dynamique, un non-terminal qui n'appartient pas à la grammaire conrète est associé à $\perp$ par la fonction *Concrete* ; il est alors dit *purement dynamique*.

La condition *2.ii* précise que si le non-terminal en partie gauche d'une production dynamique est lui même purement dynamique, alors toute la production est purement dynamique, c'est-à-dire que tous les non-terminaux en partie droite le sont. Intuitivement, une structure concrète ne peut pas être dérivée à partir d'un schéma de récursion purement abstrait.

La condition *2.iii* stipule par contre que si le non-terminal en partie gauche d'une production dynamique p_d appartient également à la grammaire concrète, alors il existe une unique production concrète p_c associée, de même non-terminal en partie gauche. Dans ce cas, à tout non-terminal apparaissant en partie droite de la production dynamique p_d et appartenant à la grammaire concrète, il existe un non-terminal dans la production concrète p_c qui lui est associé. Intuitivement, il faut que les éléments concrets apparaissant dans une production dynamiques puissent être retrouvés dans la production concrète associée.

Une production dynamique p_d est donc soit purement abstraite, soit associée à une unique production concrète p_c, qui est de même type en partie gauche et pour tous les non-terminaux de type concret en partie droite de p_d, il existe un non-terminal correspondant de même type dans p_c.

Cependant, une structure physique donnée (non-terminal concret) peut être référencée plus d'une fois dans une production dynamique. Cette possibilité est illustrée par l'exemple de la fonction **double** (figure 4.2) qui est un exemple bien connu [DJL88] des limitations du formalisme classique des grammaires attribuées.

Par ailleurs, une partie gauche concrète, qui est par définition associée à une partie gauche dynamique (condition *1*), peut également être référencée dans la partie droite de la production dynamique. C'est le cas dans l'exemple du **while** (figure 4.3) qui servira tout au long de ce chapitre. Ces particularités et ces "effets spéciaux" sont l'essence même des grammaires attribuées dynamiques. Il permettent de spécifier des calculs qui n'étaient pas exprimables avec les grammaires attribuées classiques.

Finalement, la condition *3* de la définition 4.2.2 formalise la contrainte que, pour deux productions de même type en partie gauche et de même *Concrete*[2] associé, les parties gauches doivent avoir le même nom et tous les noms communs aux deux productions doivent être de

2. Avec éventuellement $Concrete = \perp$.

Production concrète $p \in P_c$

$$p : while \rightarrow cond\ body$$

Productions dynamiques p_r et $p_t \in P_d$

$$p_r : while \rightarrow cond\ body\ loop$$

$$p_t : while \rightarrow cond$$

avec

$$\begin{aligned}
&Concrete(p_r) = Concrete(p_t) = (p)\\
&Concrete(p_r, while) = Concrete(p_t, while) = (p, while)\\
&Concrete(p_r, cond) = Concrete(p_t, cond) = (p, cond)\\
&Concrete(p_r, body) = (p, body)\\
&Concrete(p_r, loop) = (p, while)
\end{aligned}$$

FIG. 4.3: *Partie d'un couple de grammaires pour l'instruction* ***while***

même type. Ceci implique en particulier que si le $Concrete$ d'un tel nom commun est défini ($\neq \perp$), alors ce nom représente effectivement le même objet concret.

Pour illustrer les notions présentées, je prends comme exemple la définition d'une partie de la sémantique dynamique d'un langage de programmation contenant des boucles, avec une grammaire attribuée dynamique décrivant un interpréteur. Cette application n'est pas traitable avec les grammaires attribuées classiques. La figure 4.3 présente la structure de l'instruction **while** comme une partie d'un couple de grammaires $(G_d, G_c, Concrete)$ où *cond* est du type $COND \in N_d \cup N_c$ représentant une condition booléenne et où tous les autres noms dans les productions sont du type $STMT \in N_d \cup N_c$ représentant des instructions. $p \in P_c$ est la production concrète qui spécifie qu'une instruction **while** est constituée d'une condition et d'un ensemble d'instructions. p_r et $p_t \in P_d$ sont deux productions dynamiques qui représentent respectivement le comportement récursif et terminal d'une boucle **while** ; la production concrète associée à ces deux productions dynamiques est p.

Un bloc de règles sémantiques est une structure conditionnelle (un arbre de décision) qui définit toutes les productions dynamiques qui sont applicables à un même endroit (c'est-à-dire qui sont associées soit à une même production concrète, soit au même non-terminal purement dynamique, cf. contraintes de la définition 4.2.5 un peu plus loin) avec leurs règles sémantiques et l'ensemble des conditions qui permettent de choisir la production à appliquer.

Définition 4.2.3 (Bloc de règles sémantiques) *Un* ***bloc de règles sémantiques*** *b est défini inductivement comme suit :*

$$b = \langle R, \langle e, b, b\rangle\rangle \mid \langle p, R\rangle$$

où R est un ensemble (éventuellement vide) de règles sémantiques (inconditionnelles), e est une condition (expression booléenne sur des occurrences d'attributs) et p est une production.
□

La figure 4.4 présente le bloc de règles sémantiques décrivant la sémantique (de façon dénotationnelle) de l'instruction **while**. Les attributs *henv* (hérité) et *senv* (synthétisé) représentent l'environnement d'exécution (mémoire, etc) d'une instruction et l'attribut *val* contient la valeur de la condition.

$$
\begin{array}{ll}
\langle\ cond.henv = while.henv, & \text{– règle sémantique classique R} \\
\quad \langle\ (cond.val), & \text{– expression booléenne} \\
\quad\quad \langle\ while \rightarrow cond\ body\ loop, & \text{– cas true} : \langle p_r, R' \rangle \\
\quad\quad\quad body.henv = while.henv & \\
\quad\quad\quad loop.henv = body.senv & \\
\quad\quad\quad while.senv = loop.senv\ \rangle & \\
\quad\quad \langle\ while \rightarrow cond, & \text{– cas false} : \langle p_t, R'' \rangle \\
\quad\quad\quad while.senv = while.henv\ \rangle\ \rangle\ \rangle &
\end{array}
$$

FIG. 4.4: *Le bloc de règles sémantiques pour l'instruction* ***while***

$$
\begin{array}{l}
\{\ (\ ((cond.val, true),\ \ while \rightarrow cond\ body\ loop), \\
\quad\quad cond.henv = while.henv \\
\quad\quad body.henv = while.henv \\
\quad\quad loop.henv = body.senv \\
\quad\quad while.senv = loop.senv\ ) \\
\ \ (\ ((cond.val, false),\ \ while \rightarrow cond), \\
\quad\quad cond.henv = while.henv \\
\quad\quad while.senv = while.henv\ )\ \}
\end{array}
$$

FIG. 4.5: *L'ensemble $\mathcal{R}^b$ pour l'instruction* ***while***

Dans un bloc, les règles sémantiques sont associées à n'importe quel nœud de l'arbre de décision tandis que les productions apparaissent uniquement aux feuilles. La définition suivante montre comment un bloc peut être "aplati" en une collection de productions classiques — avec leurs règles sémantiques.

Définition 4.2.4 (Ensemble $\mathcal{R}^b$) *Pour chaque bloc b, $\mathcal{R}^b$ est l'ensemble de toutes les règles sémantiques de b,* ***qualifiées*** *par la conjonction des conditions (qui peut être vue comme un "chemin") qui contraignent leur application et par la production à laquelle elles sont attachées. L'ensemble $\mathcal{R}^b$ est inductivement défini comme suit :*

- $\mathcal{R}^{\langle p,R \rangle} = \{((\varepsilon, p), R)\}$
- $\mathcal{R}^{\langle R, \langle e, b_{true}, b_{false} \rangle \rangle} =$

$$
\begin{array}{ll}
let & \mathcal{R}^{b_{true}} = \cup_i((c_i, p_i), R_i), \\
 & \mathcal{R}^{b_{false}} = \cup_j((c_j, p_j), R_j) \\
in & \cup_i\,(((e, true).c_i, p_i), R \cup R_i) \bigcup \cup_j(((e, false).c_j, p_j), R \cup R_j)
\end{array}
$$

□

L'ensemble $\mathcal{R}^b$ pour l'exemple du **while** est présenté à la figure 4.5.

Pour un bloc de règles sémantiques donné b, on définit $\mathcal{PR}^b$ comme l'ensemble de toutes les productions de b : $\mathcal{PR}^b = \{p \mid ((c,p), R) \in \mathcal{R}^b\}$. On dit que la paire $((c,p), R)$ est *bien formée* si l'ensemble R de règles sémantiques est bien formé pour la production p et que chaque condition e dans le chemin c (conjonction de conditions) ne réfère que des occurrences d'attributs d'entrée de p.

Il est maintenant possible de définir complètement les grammaires attribuées dynamiques.

Définition 4.2.5 (Grammaire attribuée dynamique) *Une* ***grammaire attribuée dynamique*** *est un tuple* $AG = (G, A, F)$ *où :*

- $G = (G_d, G_c, \mathcal{C}oncrete)$ *est un couple de grammaires;*
- $A = \bigcup_{X \in N_d} H(X) \uplus S(X)$ *est un ensemble d'attributs;*
- F *est un ensemble de blocs de règles sémantiques tel que :*
  1. $\forall b \in F$, *chaque* $((c,p), R) \in \mathcal{R}^b$ *est bien formé;*
  2. $\forall p \in P_d, \exists!\ b \in F$ *tel que* $p \in \mathcal{PR}^b$*;*
  3. $\forall p, q \in P_d$, *avec* $p \in \mathcal{PR}^{b_i}$ *et* $q \in \mathcal{PR}^{b_j}$*, tels que* $type_p(LHS(p)) = type_q(LHS(q)) = X$*, on a :*
     - $X \in (N_d - N_c) \Rightarrow b_i = b_j$*;*
     - $X \in N_c \Rightarrow (b_i = b_j \Leftrightarrow \mathcal{C}oncrete(p) = \mathcal{C}oncrete(q))$*.*

□

En fait, une grammaire attribuée dynamique décrit une fonction qui prend comme arguments

- les valeurs de tous les attributs hérités du symbole de départ (puisque ceux-ci ne sont plus interdits) et
- un arbre concret décrit par la grammaire, si dans le couple de grammaires celle-ci n'est pas vide,

et qui retourne la valeur de l'attribut synthétisé du symbole de départ. Le calcul des attributs est défini de façon triviale ; il est guidé à chaque nœud dynamique par les valeurs des différentes conditions et, le cas échéant, par la production appliquée au nœud concret correspondant.

4.3 Grammaires attribuées abstraites

Dans l'introduction de cette partie, je prétendais que les grammaires attribuées dynamiques pouvaient être implantées en utilisant les mêmes techniques que pour les grammaires attribuées classiques. L'idée de base est simple [Jou92] :

1. à partir d'une grammaire attribuée dynamique donnée, construire une grammaire attribuée classique qui possède le même "comportement" (syntaxe — c'est-à-dire le schéma de récursion — et dépendances — c'est-à-dire le flot de données) ;
2. générer l'évaluateur de cette grammaire attribuée classique ;
3. transformer cet évaluateur de façon à ce qu'il implante correctement la grammaire attribuée originale.

Je vais montrer comment construire cette grammaire attribuée équivalente, appelée la *grammaire attribuée abstraite* associée à une grammaire attribuée dynamique.

Soit $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$ la forme la plus simple d'un bloc (conditionnel). Les productions et les règles sémantiques de la grammaire attribuée abstraite qui vont reproduire le comportement de ce bloc sont, d'une part, p_T associée avec les règles de $R \cup R_T$ et, d'autre part, $(p_F, R \cup R_F)$. De plus, ceci est correct du point de vue des schémas de récursion et des flots de données ; la condition "bien formée" sur la grammaire attribuée dynamique assure que la grammaire attribuée abstraite résultante sera également bien formée. La définition ci-dessous formalise cette intuition et ajoute une contrainte très importante : aucun attribut défini par une règle dans les groupes (R_T et R_F) sujets à la condition ne peut être évalué avant l'évaluation de la condition.

Définition 4.3.1 (Grammaire attribuée abstraite) *Pour une grammaire attribuée dynamique donnée $DAG = (G, A, F)$, où $G = (G_d, G_c, \mathcal{C}oncrete)$, la **grammaire attribuée abstraite** correspondante est définie par le tuple $AAG = (G_a, A_a, F_a)$ dans lequel :*

- $G_a = (N_a, T_a, Z_a, P_a)$; $N_a = N_d$; $Z_a = Z_d$; $T_a = T_d$; $A_a = A$;
- $P_a = \{c.p_d : X_0 \rightarrow X_1 \dots X_{n_{p_d}} \mid \exists b \in F, ((c, p_d), R) \in \mathcal{R}^b$ *avec* $p_d : X_0 \rightarrow X_1 \dots X_{n_{p_d}} \in P_d\}$;
- $F_a = \cup_{p \in P_a} F_a(p)$ *est un ensemble de règles sémantiques,* *avec* $F_a(p) = R$ *tel que* $p = c.p_d$ *et* $\exists b \in F, ((c, p_d), R) \in \mathcal{F}^b$, *avec* $\mathcal{F}^b$ *est l'ensemble défini à la définition 4.3.2 ci-dessous.*

□

Dans cette définition, $c.p_d$ est juste un nom pour une production de AAG, qui encode ses origines dans DAG : la production p_d et la séquence de gardes c. Pour l'exemple de **while**, les deux productions de la grammaire attribuée abstraite présentées ci-dessous sont les mêmes que p_r et p_t dans la figure 4.3, au nom des productions près :

$$((cond.val), true).p_r : while \rightarrow cond\ body\ loop$$
$$((cond.val), false).p_t : while \rightarrow cond$$

On définit DP comme la transformation qui, à une règle sémantique donnée de la forme $f_{p,a,X} : X.a = exp$ et à une condition e vue comme une expression sur des occurrences d'attributs, associe la règle sémantique modifiée $DP(f_{p,a,X}, e) : X.a = dp(exp, e)$, où dp est la fonction polymorphe définie par $dp(x, y) = x$. La définition de DP s'étend à l'ensemble des règles sémantiques : $DP(R, e) = \{DP(f_{p,a,X}, e) \mid f_{p,a,X} \in R\}$. Le rôle de DP est d'assurer qu'un attribut donné ne peut pas être évalué avant la condition e, en introduisant des dépendances artificielles entre des occurrences d'attributs et les conditions auxquelles elles sont soumises.

Définition 4.3.2 (Ensemble $\mathcal{F}^b$) *Pour chaque bloc b, $\mathcal{F}^b$ est l'ensemble de toutes les règles sémantiques de b, **qualifiées** et **modifiées** par la conjonction de conditions qui contraignent*

$$
\begin{array}{l}
\{\ (\ ((cond.val, true),\ \ while \rightarrow cond\ body\ loop), \\
\qquad\qquad cond.henv = while.henv \\
\qquad\qquad body.henv = dp(while.henv, cond.val) \\
\qquad\qquad loop.henv = dp(body.senv, cond.val) \\
\qquad\qquad while.senv = dp(loop.senv, cond.val)\ \ ), \\
\quad (\ ((cond.val, false),\ \ while \rightarrow cond), \\
\qquad\qquad cond.henv = while.henv \\
\qquad\qquad while.senv = dp(while.henv, cond.val)\ \ )\ \}
\end{array}
$$

FIG. 4.6: *Ensemble $\mathcal{F}^b$ de la grammaire attribuée abstraite de l'instruction* ***while***

leur application et par la production à laquelle elles sont attachées. L'ensemble $\mathcal{F}^b$ est inductivement défini comme suit :

- $\mathcal{F}^{\langle p,R\rangle} = \{((\varepsilon, p), R)\}$
- $\mathcal{F}^{\langle R,\langle e,b_{true},b_{false}\rangle\rangle} =$
  $let\ \ \mathcal{F}^{b_{true}} = \cup_i((c_i, p_i), R_i),$
  $\quad\ \ \mathcal{F}^{b_{false}} = \cup_j((c_j, p_j), R_j)$
  $in\ \cup_i\ (((e, true).c_i, p_i), R \cup DP(R_i, e)) \bigcup \cup_j(((e, false).c_j, p_j), R \cup DP(R_j, e))$

□

Cette forme est pratiquement la même que celle "aplatie" de $\mathcal{R}_b$, mais elle rend cependant explicites les *dépendances de contrôle* sur les conditions. La figure 4.6 présente les productions et les règles sémantiques modifiées dans la grammaire attribuée abstraite de l'instruction **while**.

Il est clair qu'étant donné un arbre "abstrait" qui représente le même schéma de récursion qu'un calcul décrit par une grammaire attribuée dynamique, la grammaire attribuée abstraite associée décrit le même calcul sur cet arbre : les valeurs des attributs seront les mêmes et il est possible de vérifier *a posteriori* que les conditions auront également les mêmes valeurs. Les autres ajouts à la grammaire attribuée abstraite sont de pures dépendances qui assurent que l'évaluation des conditions et des attributs interviennent alternativement et dans "le bon ordre" ; ces problèmes sont abordés dans la prochaine section.

4.4 Implantation à base de séquences de visites

Cette section montre comment produire des évaluateurs à séquences de visites [Kas80, Eng84, Alb91] pour des grammaires attribuées dynamiques. Le choix de cette technique d'implantation plutôt qu'une autre est dû à plusieurs raisons : ce type d'évaluateur constitue le meilleur compromis entre l'efficacité en temps et en mémoire et la généralité des classes de grammaires attribuées qu'ils permettent d'implanter ; c'est justement ce paradigme qui est implanté dans le système FNC-2 [JPJ+90, JP91] (pour les raisons déjà mentionnées et pour la polyvalence de ces évaluateurs) ; finalement, ils sont facilement transformables en des fonctions, ce qui, dans le cadre de nos travaux sur les relations entre grammaires attribuées et programmation fonctionnelle, est un atout évident.

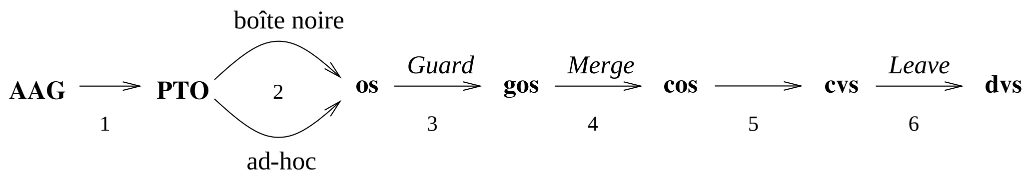

FIG. 4.7: *Les phases de la génération des séquences de visites dynamiques pour une grammaire attribuée dynamique*

Le processus général

Dans un premier temps, je présente donc rapidement l'ensemble du processus de génération des séquences de visites associées à une grammaire attribuée dynamique. La figure 4.7 illustre les différentes phases de ce processus en introduisant les abréviations des différents objets qui y sont manipulés.

1. La grammaire attribuée abstraite correspondant à une grammaire attribuée dynamique est construite et testée pour assurer qu'elle est bien l-ordonnée, en exhibant ou en construisant les partitions totalement ordonnées (PTO) appropriées des attributs de chaque non-terminal. Puisque toutes les grammaires attribuées ne sont pas l-ordonnées, ceci peut aboutir à un échec pour certaines grammaires attribuées dynamiques.

2. En utilisant ces PTO, une *séquence ordonnée (os)* d'occurrences d'attributs est générée pour chaque production de la grammaire attribuée abstraite.

3. Chaque production $c.p$ dans la grammaire attribuée abstraite correspond à une production "gardée" de la grammaire attribuée dynamique. Des *marques* (gardes) sont alors réintroduites dans chaque séquence ordonnée, pour l'évaluation et le test des différentes conditions de la production dynamique à laquelle elle correspond. Dans ces *séquences ordonnées gardées (gos)*, des marques pour chaque condition apparaissent aussitôt que les occurrences d'attributs dont elle dépend sont disponibles, dans l'ordre défini par le chemin de l'arbre de décision.

4. Les séquences ordonnées gardées correspondant à un même bloc sont alors remises ensemble, de sorte à obtenir une seule *séquence ordonnée conditionnelle* (*cos*), structurée exactement comme l'arbre de décision du bloc. Pour ce faire, il faut assurer que ces séquences de visites sont "compatibles", c'est-à-dire que, pour un bloc simple de la forme $\langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, les deux séquences ordonnées gardées pour p_T et p_F qui apparaissent avant la marque de la condition e calculent exactement la même collection d'occurrences d'attributs. Je présente un peu plus loin deux approches pour assurer cette condition de compatibilité.

5. Chaque séquence ordonnée conditionnelle est transformée en une *séquence de visites conditionnelle* (*cvs*) par le procédé de transformation classique d'une séquence ordonnée en une séquence de visites.

6. Chaque séquence de visites conditionnelle est découpée en "tranches" correspondant aux différentes visites au nœud de la partie gauche de la production, de sorte à faire de

chaque tranche une *fonction de visite* ; le code de branchement exécuté dans les visites précédentes est réintroduit au début de chacune de ces fonctions : il s'agit des *séquences de visites dynamiques (dvs).*

Dans la section 4.4.1 suivante, je présente la phase de construction partant de la grammaire attribuée dynamique et allant jusqu'aux séquences ordonnées gardées. Ensuite, je présente à la section 4.4.2 deux approches différentes permettant d'assurer que la construction de ces séquences ordonnées aboutit à des sous-séquences *compatibles* pour les alternatives d'une condition. Finalement, la phase de construction des séquences de visites dynamiques à partir de ces séquences ordonnées compatibles est présentée à la section 4.4.3.

4.4.1 Génération des séquences ordonnées gardées

J'ai déjà brièvement évoqué à la section 2.2.4 le principe pour des grammaires attribuées classiques. Je vais préciser ici quelques notions relatives aux séquences ordonnées, puisqu'elles seront fondamentales dans les preuves des théorèmes de compatibilité présentés un peu plus loin.

Je commence donc par formaliser quelques notions relatives aux grammaires l-ordonnées.

Définition 4.4.1 (Grammaire attribuée l-ordonnée) *Soit $AG = (G, A, F)$ une grammaire attribuée classique avec $G = (N, T, Z, P)$.*

- *Pour un non-terminal $X \in N$, une* ***partition totalement ordonnée (PTO)*** *sur X, T_X, est une séquence ordonnée*

$$H_1(X), S_1(X), H_2(X), S_2(X), \ldots, H_{n_X}(X), S_{n_X}(X)$$

  où n_X est le nombre de visites sur X et telle que

$$H(X) = \uplus_{1 \leq i \leq n_X} H_i(X) \qquad S(X) = \uplus_{1 \leq i \leq n_X} S_i(X)$$
$$\forall i, 1 < i \leq n_X, H_i(X) \neq \emptyset \qquad \forall i, 1 \leq i < n_X, S_i(X) \neq \emptyset$$

- *Pour une production donnée $p : X_0 \to X_1 \ldots X_n \in P$ et pour une famille de partitions totalement ordonnées $T_{X_i}, 0 \leq i \leq n$, le* ***graphe de dépendances augmenté*** *de la production, $\gamma(p) = D(p)[T_{X_0}, T_{X_1}, \ldots, T_{X_n}]$, est défini par $\gamma(p) = (W(p), E_\gamma(p))$ où les sommets $W(p)$ sont les mêmes que ceux du graphe de dépendances locales $D(p) = (W(p), E(p))$ et où les arcs $E_\gamma(p)$ sont définis comme ceci :*

$$\begin{aligned} E_\gamma(p) \quad = \quad E(p) \cup \{\ & X_i.a \to X_i.b \mid 0 \leq i \leq n, \exists j, 1 \leq j \leq n_{X_i}, \\ & (a \in H_j(X_i) \ \wedge \ b \in S_j(X_i)) \\ & \bigvee (a \in S_j(X_i) \ \wedge \ b \in H_{j+1}(X_i)) \ \} \end{aligned}$$

- *La grammaire attribuée AG est* ***l-ordonnée*** *si et seulement si il existe une famille de partitions totalement ordonnées $\{T_X \mid X \in N\}$ telle que $\forall p \in P$, $\gamma(p)$ soit acyclique.*

□

Je rappelle que le problème consistant à trouver une famille de partitions totalement ordonnées telle que la grammaire attribuée correspondante soit l-ordonnée est un problème théoriquement NP-complet, mais qu'il existe différents algorithmes d'approximations polynômiaux [Kas80, Par88]. Aussi, dans le reste de cette présentation, je ne considère que les grammaires attribuées dynamiques et les grammaires attribuées abstraites l-ordonnées[3] pour lesquelles la famille de partitions totalement ordonnées peut être obtenue par un algorithme qui ne sera pas détaillé ici.

Je définis maintenant plusieurs notions qui vont être utiles dans la suite de cette section.

Définition 4.4.2 (Dépendances) *Étant donnée une production p d'une grammaire attribuée possédant une famille de partitions totalement ordonnées et son graphe de dépendances augmenté $\gamma(p)$ associé, plusieurs notions sont définies :*

- *Étant donnée une occurrence d'attribut $X.a \in W(p)$, le graphe $\mathcal{DD}(X.a)$ est la projection des arcs de $E_\gamma(p)$ sur les occurrences d'attributs dont $X.a$ dépend. $\mathcal{DD}(X.a)$ est défini par :*

  - *ses sommets :* $\{Y.b \in W(p) \mid X.a \leftarrow Y.b \in \gamma(p)\}$
  - *ses arcs, qui sont ceux de $\gamma(p)$.*

  En cas d'ambiguïté, $\mathcal{DD}$ peut être qualifié par le nom de la production : $\mathcal{DD}_p$. Il est possible de voir $\mathcal{DD}(X.a)$ comme l'ensemble des dépendances directes menant à $X.a$.

- *La clôture transitive de $\mathcal{DD}$ est notée $\mathcal{DD}^+$. Pour une occurrence d'attribut $X.a$ donnée, le graphe $\mathcal{DD}^+(X.a)$ est appelé le* ***cône de dépendance*** *de $X.a$ et représente l'ensemble des occurrences d'attributs dont $X.a$ dépend transitivement. Comme $\gamma(p)$ est acyclique, $\mathcal{DD}^+(X.a)$ est également un graphe acyclique, qui possède une* ***frontière*** $\mathcal{DD}^\infty(X.a) = \{u \in \mathcal{DD}^+(X.a) \mid \mathcal{DD}(u) = \emptyset\}$ *telle que tout chemin de $X.a$ à un élément de la frontière est de longueur bornée. Ces notations sur les graphes sont étendues aux ensembles d'occurrences d'attributs et aux expressions sur les occurrences d'attributs.*

- *Une* ***séquence ordonnée*** *os sur p est un sous-ensemble ordonné de $W(p)$ tel que l'ordre total sur os respecte l'ordre partiel $\gamma(p)$.*

- *Une séquence ordonnée os sur p est* ***complète*** *si tout $X.a \in W(p)$ apparaît dans os. Une séquence ordonnée complète fournit par conséquent un ordre d'évaluation valide de toutes les occurrences d'attributs dans p.*

- *Pour une production donnée p, l'ensemble des* ***occurrences d'attributs évaluables*** *après os est l'ensemble $\mathcal{E}(os) = \{X.a \in W(p) \mid \mathcal{DD}(X.a) \subset os, X.a \notin os\}$.*

- *La fonction $\mathcal{P}ick(\mathcal{E}(os)) = X.a, X.a \in \mathcal{E}(os)$ est une fonction de choix d'un élément dans un ensemble.*

□

---

3. Une grammaire attribuée dynamique est dite d'une classe C (ici, l-ordonnée) si et seulement si sa grammaire attribuée abstraite associée est de classe C.

$os \leftarrow \epsilon$;
Répéter
  calculer $\mathcal{E}(os)$;
  $X.a \leftarrow \mathcal{P}ick(\mathcal{E}(os))$;
  $os \leftarrow os ++ X.a$;
Jusqu'à *os* complet

FIG. 4.8: *Tri topologique de* $\gamma(p)$

La notion de séquence ordonnée (complète) sera la base des preuves de compatibilité des séquences de visites construites, puisqu'il y a une correspondance entre les séquences ordonnées et les séquences de visites, permettant de préserver l'ordre de $\gamma(p)$.

Du point de vue des notations, une séquence ordonnée est une liste ordonnée d'occurrences d'attributs. Par analogie avec les listes et par abus de langage, je noterai par $++$ la concaténation de deux séquences ordonnées ou l'ajout d'une occurrence d'attribut à une séquence ordonnée :

$$os = (X.a,\ X.b,\ X.c) \wedge os' = (X.a,\ X.b) \Leftrightarrow os = os' ++ X.c$$

La construction d'une séquence ordonnée complète à partir d'un graphe de dépendances augmenté est un simple tri topologique, qu'il est possible de schématiser par l'algorithme[4] de la figure 4.8.

Dans le processus de génération d'un évaluateur pour une grammaire attribuée dynamique, il est donc possible de générer de façon classique une séquence ordonnée complète pour chaque production de la grammaire attribuée abstraite correspondante. Ces séquences ne contiennent alors aucune notion de *condition* et doivent les réintégrer grâce aux noms des productions de la grammaire attribuée abstraite qui contiennent ces informations : cette réintroduction des conditions dans les séquences conduit à la notion de *séquence ordonnée gardée.*

Définition 4.4.3 (Séquence ordonnée gardée) *Étant donnée une séquence ordonnée complète os sur une production $p = c.p_d \in P_a$ de la grammaire attribuée abstraite, la **séquence ordonnée gardée** $gos = \mathcal{G}uard(p, os)$ est définie comme le résultat de la transformation $\mathcal{G}uard$, qui ajoute à la séquence ordonnée initiale des marques $\mathtt{cond}_e$ pour l'évaluation de chaque condition e, dans le même ordre que dans le chemin de conditions encodé dans c. Cette transformation est inductivement définie par :*

- $\mathcal{G}uard(\varepsilon.p_d, os) = os$;
- $\mathcal{G}uard((e,v).p_e, os) = os' ++ \mathtt{cond}_e ++ \mathcal{G}uard(p_e, os'')$,
  où $os = os' ++ os''$ *avec* os' *tel que*
  $\mathcal{DD}(e) \subset os'$ *et* $\forall\ os_1 \neq os', os' = os_1 ++ os_2, \mathcal{DD}(e) \not\subset os_1$.

□

La définition de $\mathcal{G}uard$ assure que chaque condition est évaluée aussi tôt que possible, juste après l'évaluation du dernier attribut dont elle dépend.

4. Dans cet algorithme, $\leftarrow$ représente l'affectation et ϵ représente la liste vide.

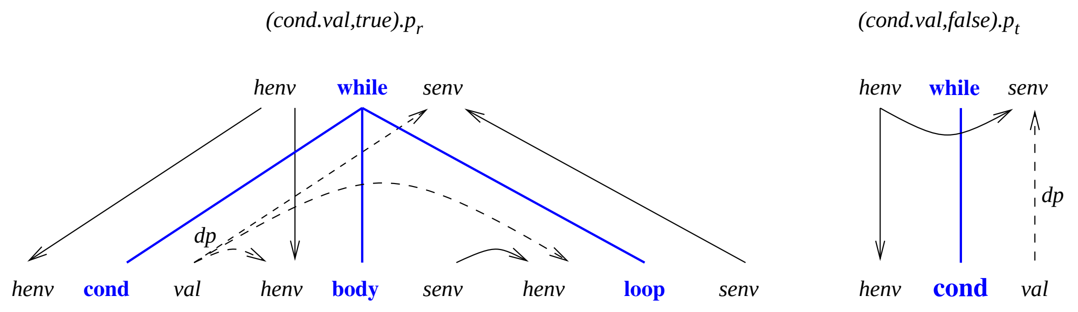

FIG. 4.9: *Les dépendances des productions du* **while** *dans la grammaire attribuée abstraite*

À partir des productions p_r et p_t de la grammaire attribuée dynamique de l'exemple du **while** (figure 4.3), j'ai déjà présenté comment il était possible de déterminer la grammaire attribuée abstraite correspondante, avec l'ensemble des règles sémantiques modifiées telles qu'elles sont présentées à la figure 4.6.

L'introduction dans ces règles de la fonction polymorphe dp ajoute de nouvelles dépendances dans les graphes de dépendances de chacune des productions. La figure 4.9 présente les graphes de dépendances de la grammaire attribuée abstraite correspondant aux productions $(cond.val, true).p_r$ et $(cond.val, false).p_t$, avec les nouvelles dépendances dues à dp représentées par des traits pointillés.

Les partitions totalement ordonnées associées aux productions de la grammaire abstraite sont les suivantes :

- sur les non-terminaux de type $STMT$ représentant des instructions, $T_{STMT} = \{henv\}\{senv\}$;
- sur les non-terminaux de type $COND$ représentant les conditions, $T_{COND} = \{henv\}\{val\}$.

À partir des graphes de dépendances de chacune des productions, augmentés par ces partitions totalement ordonnées, le tri topologique permet de déterminer une séquence ordonnée pour chaque production, que j'appelle os_{p_r} pour la production $(cond.val, true).p_r$ et os_{p_t} pour la production $(cond.val, false).p_t$:

$$\begin{array}{rcl} os_{p_r} & = & while.henv,\ cond.henv,\ cond.val,\ body.henv, \\ & & body.senv,\ loop.henv,\ loop.senv,\ while.senv \\ os_{p_t} & = & while.henv,\ cond.henv,\ cond.val,\ while.senv \end{array}$$

L'application de la transformation $\mathcal{G}uard$ sur ces séquences ordonnées permet de leur réintroduire la marque $\texttt{cond}_{cond.val}$, produisant ainsi les séquences ordonnées gardées présentées à la figure 4.10.

À ce stade du processus, étant donné un simple bloc conditionnel

$$b = \langle R, \langle e, \langle p_T, R_T\rangle, \langle p_F, R_F\rangle\rangle\rangle$$

et les deux séquences ordonnées gardées

$$\begin{array}{l} gos_T = \mathcal{G}uard(p_T, os_T) = os'_T +\!\!+ \texttt{cond}_e +\!\!+ os''_T \\ gos_F = \mathcal{G}uard(p_F, os_F) = os'_F +\!\!+ \texttt{cond}_e +\!\!+ os''_F \end{array}$$

$$\begin{aligned}
&\mathcal{G}uard((cond.val, true).p_r, os_{p_r}) = \\
&\quad while.henv,\ cond.henv,\ cond.val,\ \texttt{cond}_{cond.val},\ body.henv, \\
&\quad body.senv,\ loop.henv,\ loop.senv,\ while.senv \\
&\mathcal{G}uard((cond.val, false).p_t, os_{p_t}) = \\
&\quad while.henv,\ cond.henv,\ cond.val,\ \texttt{cond}_{cond.val},\ while.senv
\end{aligned}$$

FIG. 4.10: *Les séquences ordonnées gardées des productions du* ***while***

construites pour p_T et p_F, il faut maintenant produire une *séquence ordonnée conditionnelle* qui permette de :

1. évaluer les attributs pré-conditionnels ;
2. évaluer la condition ;
3. selon la valeur de celle-ci, continuer avec l'une ou l'autre des séquences.

Pour cela, il est nécessaire d'assurer que os'_T et os'_F sont **compatibles**, c'est-à-dire qu'elles contiennent exactement le même ensemble d'occurrences d'attributs.

4.4.2 Compatibilité des séquences ordonnées

Je vais donc maintenant présenter deux approches de construction de séquences ordonnées permettant de prouver qu'elles conduisent à des séquences ordonnées compatibles. La première approche, dite de *la boîte noire*, utilise n'importe quelle méthode de construction classique de séquences ordonnées mais nécessite l'utilisation d'une grammaire attribuée abstraite plus rigide que celle que j'ai présenté jusqu'ici. La seconde approche, dite *ad-hoc*, utilise la grammaire attribuée abstraite standard, mais implique que la construction des séquences ordonnées tienne compte des conditions.

Ensuite, il sera possible de construire les évaluateurs complets.

L'approche boîte noire

Le théorème fondamental qui permet d'assurer que les séquences ordonnées sont compatibles dans chaque alternative d'une conditionnelle est le suivant :

Théorème 4.4.1 (Compatibilité boîte noire) *Soit un bloc* $b = \langle R, \langle e, (p_1, R_T), (p_2, R_F)\rangle\rangle$ *pour lequel :*

- $p_T = (e, true).p_1$ *et* $p_F = (e, false).p_2 \in P_a$ *sont des productions de la grammaire attribuée abstraite,*
- os_T *et* os_F *sont les séquences ordonnées générées par l'algorithme de tri topologique,*
- $gos_T = \mathcal{G}uard(p_T, os_T) = os'_T +\!\!+ \texttt{cond}_e +\!\!+ os''_T$ *et*
  $gos_F = \mathcal{G}uard(p_F, os_F) = os'_F +\!\!+ \texttt{cond}_e +\!\!+ os''_F$
  sont les séquences ordonnées gardées correspondantes.

Si la fonction de choix utilisée dans le tri topologique est déterministe, alors $os'_T = os'_F$.
□

Cela signifie que si la génération des séquences ordonnées individuelles est basée sur un générateur traditionnel déterministe — ce qui est fréquemment le cas en pratique — alors les séquences seront compatibles. En fait, le résultat qui va être présenté ici est plus précisément que les sous-séquences alternatives d'une condition sont *identiques*, ce qui les rend bien évidemment compatibles.

Malheureusement, à partir des grammaires attribuées abstraites telles qu'elles ont été définies (cf. définition 4.3.1), il est impossible de prouver ce théorème.

Le problème qui est susceptible d'être rencontré est le suivant. Supposons que, dans un bloc $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, un non-terminal X apparaisse uniquement dans la production p_T et que s soit un attribut purement synthétisé attaché à X ou, plus exactement, que s ne dépende d'aucun autre attribut de X dans la partition totalement ordonnée[5]. Dans ce cas, l'algorithme de tri topologique peut décider de prendre l'occurrence d'attribut $X.s$ avant que la condition e ne soit prête à être évaluée. Ainsi, par la transformation $\mathcal{G}uard$, l'évaluation de $X.s$ apparaîtra dans os'_T et comme $X.s$ n'apparaît pas dans gos_F, os'_T et os'_F seront donc incompatibles.

Ce petit exemple montre que, pour prouver le théorème 4.4.1, il est nécessaire d'imposer des conditions supplémentaires sur la grammaire attribuée dynamique et de modifier légèrement la grammaire attribuée abstraite qui lui est associée.

Cependant, ces conditions sont fréquemment réalisables en pratique et ne réduisent pas à néant l'intérêt de cette approche *boîte noire*. Je présente un peu plus loin l'approche *ad-hoc* qui ne nécessite pas d'imposer ces contraintes, mais qui demande à ce que le générateur de séquences ordonnées soit légèrement modifié.

La grammaire attribuée abstraite étendue nécessaire pour l'approche *boîte noire* est donc légèrement enrichie par rapport à celle définie initialement (définition 4.3.1) :

Définition 4.4.4 (Grammaire attribuée abstraite étendue) *La **grammaire attribuée abstraite étendue** associée à une grammaire attribuée dynamique $DAG = (G, A, F)$ donnée, où $G = (G_d, G_c, \mathcal{C}oncrete)$, est un tuple $EAAG = (G_a, A_a, F_a)$ où :*

- $G_a = (N_a, T_a, Z_a, P_a)$; $N_a = N_d$; $Z_a = Z_d$; $T_a = T_d$;
- $A_a = \bigcup_{X \in N_a} A_a(X)$, *avec* $A_a(Z_a) = A(Z_d)$ *et*
  $\forall X \in N_a, X \neq Z_a, A_a(X) = A(X) \cup \{\mathtt{x}\}$
  où $\mathtt{x}$ *est un nouvel attribut hérité de type booléen ;*
- $P_a = \{c.p_d : X_0 \to X_1 \dots X_{n_{p_d}} \mid \exists b \in F, ((c, p_d), R) \in \mathcal{R}^b$ *avec* $p_d : X_0 \to X_1 \dots X_{n_{p_d}}\}$
- $F_a = \cup_{p \in P_a} F_a(p)$ *est un ensemble de règles sémantiques,*
  avec $F_a(p) = R$ *tel que* $p = c.p_d$ *et* $\exists b \in F, ((c, p_d), R) \in \mathcal{F}^b_{\mathtt{x}}$,
  où $\mathcal{F}^b_{\mathtt{x}}$ *est l'ensemble défini par la définition 4.4.5.*

□

Définition 4.4.5 ($\mathcal{F}^b_{\mathtt{x}}$) *Pour chaque bloc b, $\mathcal{F}^b_{\mathtt{x}}$ est l'ensemble de toutes les règles sémantiques de b, **qualifiées** et **modifiées** par la conjonction de conditions (le chemin) qui contraignent leur application et par la production à laquelle elles sont attachées.*

5. L'exemple donné à la fin de ce chapitre, à la figure 4.16, illustre ce genre de situation avec l'attribut s.

Soit $\mathcal{F}_{\mathtt{x}}$ la transformation qui est inductivement définie comme suit :

- $\mathcal{F}_{\mathtt{x}}(\langle p, R\rangle) = \{((\varepsilon, p), R)\}$

- $\mathcal{F}_{\mathtt{x}}(\langle R, \langle e, b_T, b_F\rangle\rangle) =$
$$\begin{aligned} & let\ \mathcal{F}_{\mathtt{x}}(b_T) = \textstyle\bigcup_i((c_i, p_i), R_i), \\ & \quad \mathcal{F}_{\mathtt{x}}(b_F) = \textstyle\bigcup_j((c_j, p_j), R_j), \\ & \quad V_T = \textstyle\bigcap_i RHS(p_i), \\ & \quad V_F = \textstyle\bigcap_j RHS(p_j), \\ & \quad V = V_T \cap V_F, \\ & \quad \phi_T = \textstyle\bigcup_i (((e, true).c_i, p_i), R \cup DP(R_i, e) \cup R_{\mathtt{x}}(p_i, V_T - V, e)), \\ & \quad \phi_F = \textstyle\bigcup_j (((e, false).c_j, p_j), R \cup DP(R_j, e) \cup R_{\mathtt{x}}(p_j, V_F - V, e)) \\ & in\ \phi_T \cup \phi_F \end{aligned}$$

où, étant donnée une production p, un ensemble de noms $V \subset RHS(p)$ et une condition e sur p, $R_{\mathtt{x}}(p, V, e) = \{X.\mathtt{x} = e \mid X \in V\}$.

Alors, $\mathcal{F}^b_{\mathtt{x}} = \{((c, p), \mathcal{T}(R)) \mid ((c, p), R) \in \mathcal{F}_{\mathtt{x}}(b)\}$, où $\mathcal{T}(R)$ est dérivé de R par les transformations suivantes :

- *pour tout $X \in RHS(p)$ tel qu'il n'existe pas de règle dans R définissant $X.\mathtt{x}$, ajouter la règle sémantique $X.\mathtt{x} = LHS(p).\mathtt{x}$ (si $LHS(p) = Z_a$, ajouter $X.\mathtt{x} = true$) ;*

- *si $LHS(p) \neq Z_a$, pour tout $s \in S(type_p(LHS(p)))$, transformer sa définition $LHS(p).s = exp$ en la règle $LHS(p).s = dp(exp, LHS(p).\mathtt{x})$.*

□

La figure 4.11 présente les graphes de dépendances des productions de l'exemple du **while**, modifiées par $\mathcal{F}_{\mathtt{x}}$ (les dépendances relatives à l'attribut `x` apparaissent en gras et celles dues à *dp* sont en pointillés). Bien que cet exemple ne soit pas très significatif, il est possible de constater que, pour chaque non-terminal en partie droite d'une production, tous ses attributs synthétisés dépendent de l'attribut `x`. Cet attribut dépend lui même soit de la valeur de la condition *cond.val* (si le non-terminal auquel il se rapporte n'est pas dans l'intersection des productions alternatives), soit de la valeur de `x` sur le non-terminal en partie gauche de la production.

Les principales différences entre la transformation $\mathcal{F}_{\mathtt{x}}$ (définition 4.4.5) et la transformation $\mathcal{F}$ (définition 4.3.2) de la construction standard de grammaire attribuée abstraite sont les suivantes :

1. *Ajout de l'attribut* `x`.
   $X.\mathtt{x}$ est une occurrence d'attribut dont la valeur est insignifiante, mais qui ne devient évaluable que lorsque qu'il devient possible de référer à X. En particulier, si X n'apparaît que dans une seule alternative d'une production conditionnelle, référer X n'a de sens qu'après l'évaluation de la condition et c'est pour cela que $X.\mathtt{x}$ dépend de la condition. Ainsi, même s'il est principalement introduit pour des raisons techniques, cet attribut a le rôle sémantique de confirmer l'existence du nœud de l'arbre auquel il est attaché.

2. *Ajout des dépendances vers chaque attribut synthétisé de chaque non-terminal, depuis leur occurrence d'attribut* `x` *correspondant (à l'exception du non-terminal de la racine).*
   Ceci assure qu'aucune visite à un nœud ne peut être effectuée avant que l'attribut `x` correspondant ne soit évalué.

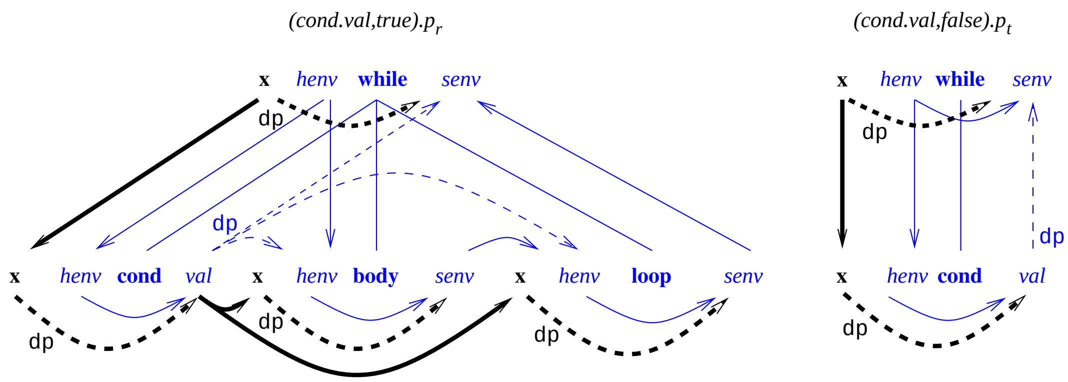

FIG. 4.11: *Les dépendances des productions du* ***while*** *avec l'attribut* `x`

3. *Ajout de dépendances de "contrôle".*
   Considérons un nom de non-terminal X dans un bloc simple $b = \langle R, \langle e, \langle p_T, R_T\rangle, \langle p_F, R_F\rangle\rangle\rangle$. Comme je l'ai déjà expliqué, si X n'est pas dans l'intersection de p_T et de p_F, il est nécessaire d'interdire que ce non-terminal soit visité avant l'évaluation de la condition e. Les règles ajoutées par les appels à $R_{\mathtt{x}}$ dans la transformation $\mathcal{F}_{\mathtt{x}}$ assurent précisément cette condition.

La condition 3 ci-dessus est une restriction dans le sens où cela peut conduire au rejet de certaines grammaires attribuées dynamiques qui pourraient être traitées par d'autres méthodes (en particulier par l'approche *ad-hoc* présentée plus loin). Les grammaires attribuées dynamiques rejetées sont celles dans lesquelles il existe une production conditionnelle et un non-terminal X en partie droite de celle-ci tel que la condition dépend d'un attribut de X (calculé par exemple lors d'une première visite) et qu'un autre attribut de X (calculé lors d'une visite ultérieure) dépende de la valeur de la condition : dans ce cas, l'introduction de l'attribut `x` et des dépendances associées à la condition provoque une circularité artificielle.

Il n'est malheureusement pas possible d'assouplir cette restriction dans l'approche *boîte noire*, car elle ne permet de disposer d'aucune information pour distinguer, à partir des dépendances locales, les attributs d'un non-terminal en partie droite qui dépendent réellement de la condition de ceux du même non-terminal qui ne dépendent pas vraiment de la condition. Par conséquent, il est nécessaire d'obliger un nom de non-terminal en partie droite d'une production conditionnelle à appartenir complètement à la partie commune — auquel cas aucun de ses attributs ne dépend de la condition — ou complètement aux parties alternatives — auquel cas tous ses attributs dépendent de la condition.

Finalement, il est intéressant de remarquer que, lorsqu'un bloc contient deux conditions imbriquées, les règles $R_{\mathtt{x}}$ générées pour la condition interne sont ensuite correctement prises en compte par la transformation DP pour la condition externe, de telle manière que les attributs `x` correspondant dépendent des deux conditions.

Après ces remarques, je peux maintenant présenter la démonstration du théorème 4.4.1.

Démonstration 4.4.1 *La démonstration suit les étapes de l'algorithme de tri topologique et procède par induction sur la longueur des séquences ordonnées.*

Tout d'abord, il faut remarquer que $\mathcal{DD}_{p_T}(e) = \mathcal{DD}_{p_F}(e)$. *Par conséquent, si la grammaire attribuée dynamique est bien formée,* e *est une expression sur les occurrences d'attributs qui* *doit être bien formée à la fois dans* p_T *et dans* p_F*, de sorte que les éléments de* $\mathcal{DD}_{p_T}(e)$ *qui sont exactement les occurrences d'attributs apparaissant dans e doivent également exister dans* p_F *et vice versa. Il est donc raisonnable d'abstraire le raisonnement sur* p_T *ou sur* p_F *et de considérer* $\mathcal{DD}(e)$.

Hypothèse d'induction: Soit e est évaluable, soit $os_T^n = os_F^n \subset W(p_T) \cap W(p_F)$*, où n est la longueur de la séquence ordonnée.*

- *Pas initial:* $os_T^0 = os_F^0 = \emptyset$*, CQFD.*

- *Pas d'induction: Supposons que l'hypothèse d'induction soit vraie après le pas n.*

  Si $\mathcal{DD}(e) \subset os_T^n = os_F^n$*, e est évaluable et l'induction termine.*

  Sinon, les deux ensembles suivants sont définis :

$$\mathcal{E}_T = \mathcal{E}(os_T^n, p_T) = \{X.a \in W(p_T) \mid \mathcal{DD}_{p_T}(X.a) \subset os_T^n, X.a \notin os_T^n\}$$
$$\mathcal{E}_F = \mathcal{E}(os_F^n, p_F) = \{X.a \in W(p_F) \mid \mathcal{DD}_{p_F}(X.a) \subset os_F^n, X.a \notin os_F^n\}$$

  Il faut alors montrer que $\mathcal{E}_T = \mathcal{E}_F \subset W(p_T) \cap W(p_F)$*, sachant que* $os_T^n = os_F^n$ *et* $\mathcal{DD}(e) \not\subset os_T^n = os_F^n$.

  Supposons donc que $X.a \in \mathcal{E}_T$ *; il faut alors montrer que* $X.a \in \mathcal{E}_F$ *également. Il y a alors deux possibilités :*

  $X.a \in W_u(p_T)$ *: Montrons tout d'abord que* $X \in p_T \cap p_F$.

  i. *Si* $X = LHS(p_T)$*, c'est vrai par définition.*
  ii. *Si* $X \in RHS(p_T)$*, alors* $a \in S(type_{p_T}(X))$ *et, par propriété de* $\mathcal{F}^b_{\mathtt{x}}$*,* $\mathcal{DD}_{p_T}(X.a) \neq \emptyset$*, puisqu'il contient au moins*[6] $X.\mathtt{x}$*. Comme X.a est prêt à être évalué,* $X.\mathtt{x} \in os_T^n = os_F^n$.
      Si $X \notin p_T \cap p_F$*, par les propriétés des règles de* $R_{\mathtt{x}}(p_T, p_T - (p_T \cap p_F), e)$ *dans la grammaire attribuée abstraite étendue,* $X.\mathtt{x}$ *dépend de e, ce qui contredit l'hypothèse selon laquelle e n'est pas évaluable.*

  Comme $X \in p_T \cap p_F$ *et que, par définition, il a le même type dans les deux productions, la partition totalement ordonnée* T_X *utilisée dans* $\gamma(p_F)$ *est la même que dans* $\gamma(p_T)$*. Puisque X.a est une occurrence d'entrée, toutes ses dépendances sont dans* T_X *et donc* $\mathcal{DD}_{p_F}(X.a) = \mathcal{DD}_{p_T}(X.a) \subset os_T^n = os_F^n$*, ce qui implique que* $X.a \in \mathcal{E}_F$.

  $X.a \in W_d(p_T)$ *: Soit a est l'attribut* $\mathtt{x}$ *d'un nom de non-terminal en partie droite, soit la règle sémantique* $f_{p_T,a,X}$ *qui définit X.a est dans* $R \cup R_T$.

  i. *Si* $a = \mathtt{x}$ *et si* $X \notin p_T \cap p_F$*, le même raisonnement qu'en ii ci-dessus conduit à une contradiction. Donc,* $X \in p_T \cap p_F$ *et* $X.\mathtt{x}$ *dépend uniquement de* $LHS(p_T).\mathtt{x}$*, qui est le même que* $LHS(p_F).\mathtt{x}$.[7]
  ii. *Si* $f_{p_T,a,X} \in R_T$*, cette règle a été produite par la transformation* $DP(R_T, e)$ *de* $\mathcal{F}_{\mathtt{x}}$*, donc* $\mathcal{DD}(e) \subset \mathcal{DD}_{p_T}(X.a)$ *; ceci contredit la supposition selon laquelle e n'est pas évaluable, donc* $f_{p_T,a,X} \in R$*, ce qui implique que* $X \in p_T \cap p_F$ *et*

6. C'est là le point technique qui justifie l'introduction de l'attribut $\mathtt{x}$.
7. Dans un bloc plus général, cela peut se traduire par "$X.\mathtt{x}$ dépend uniquement des conditions plus hautes que e dans l'arbre de décision qui s'appliquent à la fois à p_T et à p_F".

$f_{p_F,a,X} = f_{p_T,a,X}$. *Ces deux assertions impliquent à leur tour que* $X.a$ *dépend des mêmes attributs dans* p_F *et dans* p_T*, puisque toutes ses dépendances sont soit dans la règle sémantique, soit dans le (même)* T_X.

Dans tous les cas, la conclusion est que $X.a \in W_d(p_F)$ *avec la même définition et les mêmes dépendances que dans* p_T. *Par conséquent,* $\mathcal{DD}_{p_F}(X.a) = \mathcal{DD}_{p_T}(X.a) \subset os^n_T = os^n_F$ *et, comme plus haut,* $X.a \in \mathcal{E}_F$.

Puisque le raisonnement ci-dessus est symétrique pour p_T *et pour* p_F*, il montre que* $\mathcal{E}_T = \mathcal{E}_F \subset W(p_T) \cap W(p_F)$. *Donc, si* $\mathcal{P}ick$ *est déterministe,* $\mathcal{P}ick(\mathcal{E}_T) = \mathcal{P}ick(\mathcal{E}_F) = X.a \in W(p_T) \cap W(p_F)$ *et* $os^{n+1}_T = os^n_T \mathbin{+\!\!+} X.a = os^n_F \mathbin{+\!\!+} X.a = os^{n+1}_F \subset W(p_T) \cap W(p_F)$, *ce qui conclue la démonstration du pas d'induction.*

Cette preuve s'étend naturellement pour un arbre de décision de hauteur supérieure à 1.
□

L'approche *ad-hoc*

Dans cette section, je présente la seconde approche permettant d'assurer que les séquences de visites générées pour les productions alternatives d'un bloc simple sont compatibles. Par opposition à l'approche *boîte noire* précédemment présentée, cette approche *ad-hoc* fonctionne sur la définition standard de grammaire attribuée abstraite (cf. définition 4.3.1), mais elle requiert des modifications dans le tri topologique utilisé pour produire les séquences ordonnées. Plus précisément, ce tri topologique modifié doit tenir compte des conditions. L'idée de base de cette modification est de réduire, à chaque pas de tri, l'ensemble des occurrences d'attributs aux seuls qui sont nécessaires pour calculer la prochaine condition. L'algorithme ainsi modifié est donné à la figure 4.12.

Pour tout $\gamma(p)$ de la forme $p = c.p_d = (e_1, t_1).(e_2, t_2) \dots (e_n, t_n).p_d$ **Faire**
$os \leftarrow \epsilon$; $i \leftarrow 0$; $S \leftarrow \emptyset$;
Répéter
    $i \leftarrow i + 1$;
    Si $i = n + 1$ **Alors** $S \leftarrow W(p)$
    Sinon $S \leftarrow S \cup \mathcal{DD}^+(e_i)$ — *cône de dépendance de la condition*
    Répéter
        calculer $\mathcal{E}(os)$; — *l'ensemble des attributs prêts à être évalués*
        $X_i.a \leftarrow \mathcal{P}ick(\mathcal{E}(os) \cap S)$;
        $os \leftarrow os \mathbin{+\!\!+} X_i.a$;
    Jusqu'à $\mathcal{E}(os) \cap S = \emptyset$
Jusqu'à os complet.

FIG. 4.12: *Tri topologique* <u>*conditionnel*</u> *de* $W(p)$.

L'utilisation de cet algorithme de tri topologique conditionnel, sur les graphes de dépendances augmentés par les partitions totalement ordonnées dérivées de la grammaire attribuée abstraite originale, permet d'établir un théorème équivalent à celui de l'approche boîte noire (théorème 4.4.1). La preuve de ce théorème 4.4.4, énoncé un peu plus loin, nécessite l'utilisation des deux lemmes suivants.

Le lemme 4.4.2 est le plus important : il montre qu'avant l'évaluation d'une condition, sur chaque production sujette à cette condition, le tri topologique conditionnel ordonne exactement le même ensemble d'occurrences d'attributs.

Lemme 4.4.2 *Étant donné un bloc* $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, *alors* $\mathcal{DD}^+_{p_T}(e) = \mathcal{DD}^+_{p_F}(e)$.
□

Démonstration 4.4.2 *La preuve se fait par récurrence sur la transitivité des dépendances dans* $\mathcal{DD}^+$.

Pas initial : $\mathcal{DD}_{p_T}(e) = \mathcal{DD}_{p_F}(e)$

Si la grammaire attribuée est bien formée, e est une expression sur les occurrences d'attributs qui doit être bien formée à la fois pour p_T *et pour* p_F *; donc, les éléments de* $\mathcal{DD}_{p_T}(e)$, *qui sont exactement les occurrences d'attributs apparaissant dans l'expression e, doivent également exister dans* p_F *et vice versa.*

Pas d'occurrence d'entrée $\mathcal{DD}^2_{p_T}(e) = \mathcal{DD}^2_{p_F}(e)$

Pour chaque $X.a$ *dans* $\mathcal{DD}_{p_T}(e)$, $X.a$ *doit être une occurrence d'attribut d'entrée (utilisée) dans* p_T *et* X *doit apparaître à la fois dans* p_T *et dans* p_F. *Donc, comme* $type_{p_T}(X) = type_{p_F}(X) = type_b(X)$, *le même* T_X *est utilisé dans* $\gamma(p_T)$ *et dans* $\gamma(p_F)$. *Par conséquent,* $\mathcal{DD}_{p_T}(X.a) = \mathcal{DD}_{p_F}(X.a)$.
De plus, comme $X.a$ *est une occurrence d'attribut d'entrée dans* p_T, *toutes ses dépendances directes sont dans* T_X : $\forall Y.b \in \mathcal{DD}_{p_T}(X.a)$, $Y = X$.

Pas d'occurrence de sortie : $\mathcal{DD}^3_{p_T}(e) = \mathcal{DD}^3_{p_F}(e)$

Supposons maintenant qu'avec un $X.a$ *tel que dans le pas précédent, il existe une occurrence d'attribut* $X.b$ *dans* $\mathcal{DD}_{p_T}(X.a)$ *telle que* $\mathcal{DD}_{p_T}(X.b) \neq \mathcal{DD}_{p_F}(X.b)$.
Comme $X.a \in W_u(p_T)$, $X.b \in W_d(p_T)$ *et les dépendances directes de* $X.b$ *(les éléments de* $\mathcal{DD}_{p_T}(X.b)$*) sont induites soit par la partition totalement ordonnée* T_X, *soit par la règle sémantique définissant* $X.b$. *La différence entre* $\mathcal{DD}_{p_T}(X.b)$ *et* $\mathcal{DD}_{p_F}(X.b)$ *ne peut pas venir de* T_X *puisque c'est le même qui est utilisé dans* $\gamma(p_T)$ *et dans* $\gamma(p_F)$. *Par conséquent,* $X.b$ *doit être défini par des règles sémantiques différentes dans* p_T *et dans* p_F, *ce qui implique que* $f_{p_T,b,X} \in R_T$.
Mais dans ce cas, cette règle sémantique doit avoir été soumise à la transformation $DP(R_T, e)$ *dans* $\mathcal{F}$, *donc* $X.b \leftarrow e$. *Cependant, par hypothèse, on a également* $e \leftarrow X.a \leftarrow X.b$ *et il y a donc une circularité dans* $\gamma(p_T)$ *et dans* $\gamma(p_F)$. *Puisque la grammaire attribuée est supposée être l-ordonnée, cela conduit à une contradiction.*
Par conséquent, pour tout $X.a \in \mathcal{DD}_{p_T}(e)$ *et* $X.b \in \mathcal{DD}_{p_T}(X.a)$, $\mathcal{DD}_{p_T}(X.b) = \mathcal{DD}_{p_F}(X.b)$.

Pas de récurrence

Puisqu'il est équivalent de raisonner sur p_T *ou sur* p_F, *ces deux pas précédemment détaillés peuvent être alternativement appliqués, aussi longtemps qu'il existe des dépendances inexplorées dans* $\mathcal{DD}^+_{p_T}(e)$, *pour montrer que* $\forall n, \mathcal{DD}^n_{p_T}(e) = \mathcal{DD}^n_{p_F}(e)$. *Puisque* $\mathcal{DD}^+_{p_T}(e)$ *est acyclique et fini, la conclusion est que* $\mathcal{DD}^+_{p_T}(e) = \mathcal{DD}^+_{p_F}(e)$. CQFD.

□

Le lemme 4.4.3 suivant prouve que l'algorithme décrit à la figure 4.12 termine et qu'il ne rencontre pas de situation de blocage.

Lemme 4.4.3 *Pour toutes les exécutions du tri topologique conditionnel, si S^i et os^i sont les valeurs de S et de os à la fin de la i-ème boucle interne, $1 \leq i \leq n$, alors $S^i = os^i$.*
□

Démonstration 4.4.3

$os^i \subset S^i$ par construction.

$S^i \subset os^i$ par l'absurde :
Supposons qu'à la fin de la i-ème boucle interne, $\exists x \in S = \bigcup_{j \leq i} \mathcal{DD}^+(e_j)$ tel que $x \notin os$.

- *Comme la boucle est terminée, $\mathcal{E}(os) \cap S = \emptyset$, ce qui implique que $x \notin \mathcal{E}(os)$. Par définition de $\mathcal{E}$ et par l'hypothèse que $x \notin os$, on peut inférer que $\mathcal{DD}(x) \not\subset os$. De plus, par définition de $\mathcal{DD}^+$ (notion de transitivité), $x \in \mathcal{DD}^+(e) \Rightarrow \mathcal{DD}(x) \subset \mathcal{DD}^+(e)$, ce qui implique que $x \in \bigcup_{j \leq i} \mathcal{DD}^+(e_j) \Rightarrow \mathcal{DD}(x) \subset \bigcup_{j \leq i} \mathcal{DD}^+(e_j)$. Donc $\exists a \in \mathcal{DD}(x), a \notin os, a \in S$.*
- *Cependant, $a \notin \mathcal{E}(os)$, puisque $\mathcal{E}(os) \cap S = \emptyset$ (hypothèse de boucle). Donc, $a \in \mathcal{DD}(x)$ est exactement dans la même situation que le x hypothétique et on peut appliquer le même raisonnement pour montrer que $\exists b \in \mathcal{DD}^2(x), b \notin os, b \in S$ et, par induction, $\forall n > 0, \exists c \in \mathcal{DD}^n(x), c \notin os, c \in S$.*
- *Mais comme $\gamma(p)$ est acyclique, $\exists n_0 > 0, \forall d \in \mathcal{DD}^{n_0}(x), \mathcal{DD}(d) = \emptyset$, ce qui contredit le fait que $d \notin os$.* CQFD.

□

À partir de ces deux lemmes, le théorème 4.4.4 assurant la compatibilité des séquences ordonnées générées correspondant à celui de l'approche boîte noire (théorème 4.4.1) peut être établi pour l'approche *ad-hoc*.

Théorème 4.4.4 *Soit un bloc $b = \langle R, \langle e, (p_1, R_T), (p_2, R_F) \rangle \rangle$ pour lequel :*

- *$p_T = (e, true).p_1$ et $p_F = (e, false).p_2 \in P_a$,*
- *os_T et os_F sont les séquences ordonnées générées par l'algorithme de tri topologique conditionnel,*
- *$gos_T = \mathcal{G}uard(p_T, os_T) = os'_T ++ \texttt{cond}_e ++ os''_T$ et*
  $gos_F = \mathcal{G}uard(p_F, os_F) = os'_F ++ \texttt{cond}_e ++ os''_F$
  sont les séquences ordonnées gardées correspondantes.

Si la fonction de choix utilisée dans le tri topologique est déterministe, alors $os'_T = os'_F$.
□

Démonstration 4.4.4 *Par le lemme 4.4.2, $\mathcal{DD}^+_{p_T}(e) = \mathcal{DD}^+_{p_F}(e)$. De plus, comme l'algorithme de tri topologique conditionnel fonctionne sans blocage jusqu'à détermination exhaustive de $\mathcal{DD}^+(e)$ (lemme 4.4.3) et que $\mathcal{P}ick$ est une fonction déterministe qui fonctionne sur deux ensembles identiques, alors $os'_T = os'_F$.*
□

4.4.3 Génération des séquences de visites dynamiques

Les deux méthodes décrites précédemment (boîte noire et *ad-hoc*) permettent au générateur d'évaluateur de produire des séquences ordonnées gardées *compatibles* pour toute production $c.p$ d'une grammaire attribuée abstraite. Dans cette section, je montre comment produire un évaluateur complet pour cette collection de séquences ordonnées gardées.

Définition 4.4.6 (Séquence ordonnée conditionnelle) *Pour un bloc $b \in F$ donné, soit V^b l'ensemble des paires $(c.p, gos)$ où $c.p \in \mathcal{PR}^b$ et où gos est sa séquence ordonnée gardée associée. Si tous les gos dans V^b sont compatibles, la* ***séquence ordonnée conditionnelle*** *cos associée à b est le résultat de la transformation $\mathcal{M}erge$ sur V^b, définie comme suit :*

Si $V^b = \{(\varepsilon.p, os)\}$ ***Alors*** $\mathcal{M}erge(V^b) = os$

Sinon $\mathcal{M}erge(V^b) = os +\!\!+ \langle e, \langle \mathcal{M}erge(V_T), \mathcal{M}erge(V_F) \rangle \rangle$
où
$V_T = \{\ (c_i.p_i, gos'_i) \mid (c.p_i, gos_i) \in V^b,\ c.p_i = (e, true).c_i.p_i,\ gos_i = os_i +\!\!+ \texttt{cond}_e +\!\!+ gos'_i\ \}$
$V_F = \{\ (c_j.p_j, gos'_j) \mid (c.p_j, gos_j) \in V^b,\ c.p_j = (e, false).c_j.p_j,\ gos_j = os_j +\!\!+ \texttt{cond}_e +\!\!+ gos'_j\ \}$
et où $os = os_i \simeq os_j$ *(elles sont compatibles).*

□

Comme les séquences ordonnées gardées associées à la grammaire attribuée abstraite (données à la figure 4.10) sont compatibles, il est possible de construire la séquence ordonnée conditionnelle pour le bloc du **while** : elle est présenté figure 4.13.

$$
\begin{array}{l}
while.henv,\ cond.henv,\ cond.val,\\
\langle\ cond.val,\\
\quad \langle\ body.henv,\ body.senv,\ loop.henv,\ loop.senv,\ while.senv\ \rangle\\
\quad \langle\ while.senv\ \rangle\ \rangle
\end{array}
$$

FIG. 4.13: *Séquence ordonnée conditionnelle pour le bloc* ***while***

De la même manière qu'en grammaires attribuées classiques, le générateur d'évaluateur construit une séquence de visites pour une séquence ordonnée donnée, il permet de construire une *séquence de visites conditionnelle* pour une séquence ordonnée conditionnelle donnée.

Afin d'expliquer comment ces séquences de visites conditionnelles seront exploitées, je rappelle brièvement la notion classique d'évaluateur basé sur le paradigme des séquences de visites [Kas80, Alb91, Kas91]. Un premier exemple a déjà été donné à la section 2.2.4 et je rappelle ici qu'il existe une séquence de visites pour chaque production, qui est une séquence d'instructions parmi les suivantes :

- `begin` i débute la i-ème visite du nœud courant ;
- `eval` a X évalue l'occurrence d'attribut $X.a$;
- `visit` i X appelle récursivement la i-ème visite au fils X ;
- `leave` i termine la i-ème visite sur le constructeur courant.

Pour tout p et os **Faire**
$n \leftarrow n_{LHS(p)}$ — *nombre de visites à p*
$vs \leftarrow$ `begin` 1;
Répéter
  $X.a \leftarrow head(os)$;
  Si $X = LHS(p) \wedge a \in S_i(X)$ **Alors**
    $vs \leftarrow vs$ ++`eval` $a\ X$;
    Si $X.a = Last(os, S_i(X)) \wedge i \neq n$ **Alors**
      $vs \leftarrow vs$ ++`leave` i ++`begin` $i+1$
    FinSi
  SinonSi $X \neq LHS(p) \wedge a \in H_j(X)$ **Alors**
    $vs \leftarrow vs$ ++`eval` $a\ X$
  SinonSi $X \neq LHS(p) \wedge a(X) = First(os, S_j(X))$ **Alors**
    $vs \leftarrow vs$ ++`visit` $j\ X$
  FinSi
  $os \leftarrow tail(os)$
Jusqu'à $os = \epsilon$;
$vs \leftarrow vs$ ++`leave` n.

FIG. 4.14: *Algorithme de production d'une séquence de visites à partir d'une séquence ordonnée*

L'algorithme classique de production d'une séquence de visites à partir d'une séquence ordonnée est présenté à la figure 4.14, dans lequel les définitions suivantes sont nécessaires :

Définition 4.4.7 *Étant donné une production p et une séquence ordonnée complète os sur $W(p)$ et S un sous ensemble de $W(p)$. Alors :*

$$Last(os, S) = \begin{cases} \perp & si\ S = \emptyset \\ w & sinon,\ où\ os = os' +\!\!+ w +\!\!+ os'', w \in S, os'' \cap S = \emptyset \end{cases}$$

$$First(os, S) = \begin{cases} \perp & si\ S = \emptyset \\ w & sinon,\ où\ os = os' +\!\!+ w +\!\!+ os'', w \in S, os' \cap S = \emptyset \end{cases}$$

□

Une simple extension de l'algorithme de la figure 4.14, prenant en compte la structure des séquences ordonnées conditionnelles, permet alors de générer les séquences de visites conditionnelles correspondantes. Je présente à la figure 4.15 la séquence de visites conditionnelle du bloc **while** produite à partir de la séquence ordonnée conditionnelle de la figure 4.13.

Les séquences de visites sont facilement implantables comme des procédures récursives qui modifient des informations et des valeurs d'attributs aux nœuds de l'arbre qu'elles traversent [Kas91]. Des travaux sur l'optimisation mémoire [Kas87, JP90] ont permis de réduire la place mémoire totale nécessaire, mais également la proportion d'attributs qui doivent être stockés dans l'arbre. Il est important de préciser que la plus probante de ces techniques, qui est basée sur une analyse statique de l'entière collection des séquences de visites [JP90], s'applique également aux grammaires attribuées dynamiques (aux séquences de visites conditionnelles), car elles ne sont pas liées au mécanisme de sélection des séquences.

```
begin 1, eval henv cond, visit 1 cond,
⟨ cond.val,
   ⟨ eval henv body, visit 1 body, eval henv loop,
     visit 1 loop, eval senv while, leave 1 ⟩ ,
   ⟨ eval senv while, leave 1 ⟩ ⟩
```

FIG. 4.15: *Séquence de visites conditionnelle pour le bloc* ***while***

Une autre implantation, plus appropriée dans le contexte de mon étude, est le paradigme de *fonction de visite* [SV91]. Une propriété importante de cette implantation est que, grâce aux techniques d'optimisation mémoire ou, en dernier ressort, grâce à la technique des *binding trees* [SV91], elle ne nécessite plus le stockage d'aucun attribut dans l'arbre. C'est une propriété particulièrement intéressante pour l'implantation des grammaires attribuées dynamiques, dans lesquelles l'arbre physique n'est pas nécessairement isomorphe à l'*arbre* des calculs, ou peut même ne pas exister du tout.

Il manque donc une dernière étape avant de pouvoir générer des fonctions de visite. Classiquement, il y a une fonction de visite pour chaque visite à un non-terminal, qui teste la production qui est appliquée à la racine du sous-arbre courant et qui *branche* la sous-séquence appropriée (délimitée par `begin` i and `leave` i). Cependant, avec les grammaires attribuées dynamiques, la sous-séquence appropriée ne dépend pas uniquement de la production appliquée à la racine du sous-arbre considéré (qui peut d'ailleurs ne pas exister physiquement), mais aussi du chemin (de la suite) de conditions qui ont été évaluées dans les visites précédentes.

Aussi, lorsqu'il faut couper une séquence de visites conditionnelle en "tranches" correspondant aux différentes visites à la partie gauche d'une production, il est nécessaire de réintroduire dans chacune de ces tranches le code de branchement exécuté dans les visites précédentes. La transformation $\mathcal{L}eave$ de la définition 4.4.8 prend en charge ces opérations de découpage et produit une *séquence de visites dynamique* qui est directement transformable en une collection de fonctions de visite.

Définition 4.4.8 (Séquence de visites dynamique) *Pour un bloc donné et sa séquence de visites conditionnelle cvs associée, la* ***séquence de visites dynamique*** *dvs correspondante est le résultat de la transformation $\mathcal{L}eave$ sur cvs, définie par :*

- ***Si*** *pas de condition dans cvs* ***Alors*** $\mathcal{L}eave(cvs) = cvs$
- ***Si*** $cvs = vs \mathbin{+\!\!+} \langle e\langle vs_T, vs_F\rangle\rangle$ *et pas de condition dans* vs
  Alors $\mathcal{L}eave(cvs) = vs \mathbin{+\!\!+} \mathcal{L}eave\langle e\langle vs_T, vs_F\rangle\rangle$
- ***Si*** $cvs = \langle e\langle vs_T, vs_F\rangle\rangle$ *et*
  $\mathcal{L}eave(vs_T) = vs_T^1 \mathbin{+\!\!+} \texttt{leave} \mathbin{+\!\!+} vs_T^2$ *avec* $\texttt{leave} \notin vs_T^1$
  $\mathcal{L}eave(vs_F) = vs_F^1 \mathbin{+\!\!+} \texttt{leave} \mathbin{+\!\!+} vs_F^2$ *avec* $\texttt{leave} \notin vs_F^1$
  Alors ***Si*** $vs_T^2 = vs_F^2 = \varepsilon$
  Alors $\mathcal{L}eave(cvs) = \langle e\langle vs_T^1, vs_F^1\rangle\rangle \mathbin{+\!\!+} \texttt{leave}$
  Sinon $\mathcal{L}eave(cvs) = \langle e\langle vs_T^1, vs_F^1\rangle\rangle \mathbin{+\!\!+} \texttt{leave} \mathbin{+\!\!+} \mathcal{L}eave(\langle e\langle vs_T^2, vs_F^2\rangle\rangle)$

□

Pour illustrer de façon significative cette transformation, l'exemple du **while** est trop simple (il n'y a qu'une seule visite). Aussi, je présente à la figure 4.16 les graphes de dépendances de deux productions d'une grammaire attribuée dynamique imaginaire. Ces productions dépendent d'une condition sur un attribut purement synthétisé d'une variable commune aux deux productions, $W.s$. Si cette condition est vraie, alors p_t est appliquée, sinon c'est p_f.

La figure 4.17 présente successivement la séquence ordonnée conditionnelle associée à ces productions, la séquence de visites conditionnelle correspondante et la séquence de visites dynamique qui est générée par la transformation $\mathcal{L}eave$.

Il est alors facile de transformer de telles séquences de visites dynamiques en une collection de fonctions de visites, dès lors que les valeurs des différentes conditions calculées dans une visite sont correctement transmises aux visites subséquentes comme des attributs locaux non temporaires. Il est à noter également que certaines de ces valeurs de conditions qui passent d'une visite à la suivante peuvent être indéfinies, mais cela ne pose pas de problème puisque toutes les valeurs qui seront effectivement utilisées dans le code de branchement auront été correctement calculées.

Ceci termine la construction des évaluateurs basés sur des séquences de visites pour les grammaires attribuées dynamiques. Tout comme pour les grammaires attribuées traditionnelles, ces évaluateurs sont aussi efficaces que possible. Lorsqu'une grammaire attribuée dynamique est évaluable en une passe, les fonctions de visites générées sont les mêmes que celles que l'on pourrait écrire à la main dans un langage muni de fonctions récursives. Cependant, lorsque les dépendances sont plus complexes, écrire un évaluateur à la main devient quasiment impossible à moins d'utiliser un mécanisme d'évaluation retardée — par exemple l'évaluation paresseuse de programmes fonctionnels —, mais dans ce cas, les évaluateurs impatients (*eager*) générés par les systèmes tels que FNC-2 peuvent être plus efficaces.

4.5 Conclusion et ouverture

Ce chapitre a montré que, dans la terminologie “grammaire attribuée”, la notion de *grammaire* n'impliquait pas nécessairement l'existence d'un arbre physique sous-jacent et que la notion d'attribut ne signifiait donc pas nécessairement décoration d'un tel arbre. Les grammaires attribuées dynamiques ainsi présentées constituent donc un nouveau formalisme qui est une simple extension de celui des grammaires attribuées. Étant en adéquation avec l'esprit général des grammaires attribuées, les grammaires attribuées dynamiques peuvent ainsi bénéficier de la puissance et de l'étendue des résultats et des techniques déjà développées dans ce domaine. Notre but, en fournissant une telle extension du formalisme des grammaires attribuées, est de promouvoir cet outil de spécification puissant dans un contexte plus large, non pas comme langage de spécification, mais plutôt comme formalisme de représentation. Le style déclaratif et structuré, ainsi que les techniques d'analyses statiques propres aux grammaires attribuées, deviennent grâce à cette extension plus généraux et plus accessibles. Les grammaires attribuées se révèlent alors, de par leur fort caractère déclaratif et local (spécification d'équations orientées au niveau des productions), complémentaires d'autres formalismes tels que la programmation fonctionnelle qui spécifie plus généralement un comportement par équations récursives, ou tels que la programmation par règles d'inférences [PDRJ95a]. Cette augmentation de l'expressivité des grammaires attribuées dynamiques offre des opportunités pour la comparaison entre formalismes et pour le transfert des résultats intéressants obtenus en grammaires attribuées vers d'autres paradigmes de programmation.

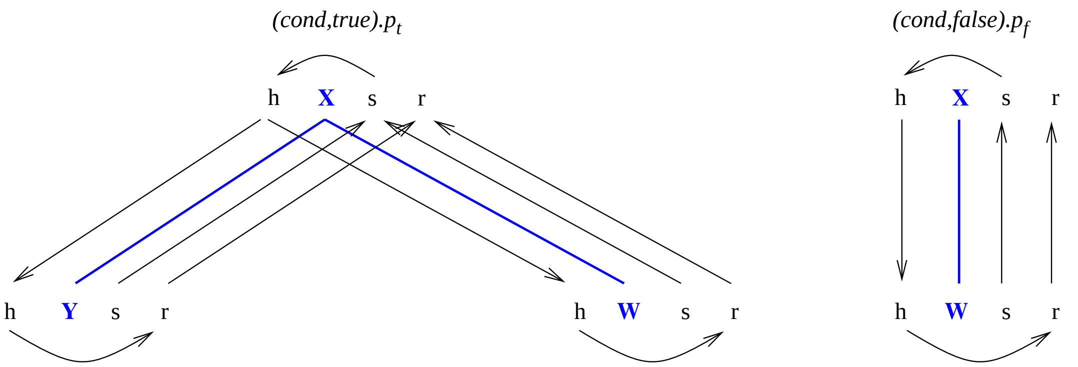

FIG. 4.16: *Un exemple de graphes de dépendances*

$W.s,$
$\langle\ cond,$ — condition sur $W.s$
$\langle\ Y.s,\ X.s,\ X.h,\ Y.h,\ W.h,\ Y.r,\ W.r,\ X.r\ \rangle,$
$\langle\ X.s,\ X.h,\ W.h,\ W.r,\ X.r\ \rangle\ \rangle$

(a) La séquence ordonnée conditionnelle

$\texttt{begin}\ \mathit{1},\ \texttt{visit}\ \mathit{1}\ W,$
$\langle\ cond,$
$\langle\ \texttt{visit}\ \mathit{1}\ Y,\ \texttt{eval}\ s\ X,\ \texttt{leave}\ \mathit{1},\ \texttt{begin}\ \mathit{2},\ \texttt{eval}\ h\ Y,$
$\texttt{eval}\ h\ W,\ \texttt{visit}\ \mathit{2}\ Y,\ \texttt{visit}\ \mathit{2}\ W,\ \texttt{eval}\ r\ X,\ \texttt{leave}\ \mathit{2}\ \rangle\ ,$
$\langle\ \texttt{eval}\ s\ X,\ \texttt{leave}\ \mathit{1},\ \texttt{begin}\ \mathit{2},\ \texttt{eval}\ h\ W,\ \texttt{visit}\ \mathit{2}\ W,\ \texttt{eval}\ r\ X,\ \texttt{leave}\ \mathit{2}\ \rangle\ \rangle$

(b) La séquence de visites conditionnelle

$\texttt{begin}\ \mathit{1},\ \texttt{visit}\ \mathit{1}\ W,$
$\langle\ cond,$
$\langle\ \texttt{visit}\ \mathit{1}\ Y,\ \texttt{eval}\ s\ X\ \rangle\ ,$
$\langle\ \texttt{eval}\ s\ X\ \rangle\ \rangle\ ,$
$\texttt{leave}\ \mathit{1},$
$\texttt{begin}\ \mathit{2},$
$\langle\ cond,$
$\langle\ \texttt{eval}\ h\ Y,\ \texttt{eval}\ h\ W,\ \texttt{visit}\ \mathit{2}\ Y,\ \texttt{visit}\ \mathit{2}\ W,\ \texttt{eval}\ r\ X\ \rangle\ ,$
$\langle\ \texttt{eval}\ h\ W,\ \texttt{visit}\ \mathit{2}\ W,\ \texttt{eval}\ r\ X\ \rangle\ \rangle$
$\texttt{leave}\ \mathit{2}$

(c) La séquence de visites dynamique.

FIG. 4.17: *Exemple de transformation de séquence ordonnée conditionnelle en séquence de visites dynamique*

Chapitre 5

Déforestation de programmes fonctionnels

5.1 Introduction et présentation du problème

Un des principaux avantages du style *applicatif* de la programmation fonctionnelle tient à l'écriture simple, modulaire et sure des programmes qu'elle autorise. En fait, pour "bien" programmer en fonctionnel, il suffit de penser les parties d'un programme en termes de suite d'applications de fonctions générales, chacune d'entre elles effectuant une tâche simple de façon très distincte, et produisant un résultat intermédiaire qui constitue l'entrée de la fonction suivante, jusqu'à obtention du résultat final. Un tel style de programmation est particulièrement agréable à utiliser et à comprendre, puisqu'il permet de découper une application en *modules*, ou fonctions, idéalement indépendants les uns des autres. De plus, il est plus facile de prouver la validité d'une succession de manipulations simples que celle d'un ensemble important d'instructions non ordonnées.

Les listes, les arbres et plus généralement toute structure intermédiaire produite par un module ou une fonction sont donc la base de ce style de programmation fonctionnelle puisqu'elles constituent la *colle* qui permet de lier les fonctions entre elles à l'intérieur d'un programme général [Hug89]. Comme le remarque Wadler [Wad88], elles en sont également le fléau dans le sens où elles exigent un coût important à l'exécution de ce programme, par leur allocation, leur parcours et leur désallocation. Elles ne sont en effet *que* des listes intermédiaires et ne font donc pas, *a priori*, directement partie du résultat final.

Ce dilemme a suscité de nombreux travaux et surtout créé un mode de développement de programmes visant à transformer automatiquement des spécifications intuitives en programmes efficaces. Burstall et Darlington [BD77], ainsi que Manna, Waldinger, Bird ou Turchin [MW75, MW79, Bir84, Tur86], furent parmi les premiers à œuvrer dans ce domaine avec les transformations dites de "dépliage/pliage" [1]. Le principe de ces transformations est un système d'inférence formel dont les termes sont des équations récursives de la forme $lhs \Leftarrow rhs$. Burstall et Darlington définissent dans [BD77] un petit nombre de règles d'inférence, parmi lesquelles la règle de *définition* de nouvelle équation récursive (appelée également *eureka* ou *généralisation* et qui relève de l'inspiration de l'utilisateur), la règle d'*instanciation* d'une

1. Le terme anglo-saxon est *unfold/fold*.

équation récursive existante et ces deux règles fondamentales de *dépliage* et de *pliage* :

- pour deux équations $E \Leftarrow E'$ et $F \Leftarrow F'$ telles que F' contienne une instance de E, le **dépliage** introduit la nouvelle équation $F \Leftarrow F''$ où F'' est le terme F' dans lequel l'instance de E a été remplacée par E' ;

- pour deux équations $E \Leftarrow E'$ et $F \Leftarrow F'$ telles que F' contienne une instance de E', le **pliage** introduit la nouvelle équation $F \Leftarrow F''$ où F'' est le terme F' dans lequel l'instance de E' a été remplacée par E.

Intuitivement, le "dépliage" fait apparaître des termes dans lesquels il est possible d'effectuer des instanciations et d'appliquer des lois sur les primitives du langages (associativité, commutativité, etc.) qui permettent de reformuler une expression avant de la "replier". Différentes stratégies d'application de ces règles ont été proposées, permettant de transformer des programmes sous la forme d'équations récursives de manière semi-automatique, de sorte que le programme résultant de la transformation soit plus efficace [BD77].

Cette idée de dépliage et pliage a inspiré différentes approches de transformation de programmes visant à en améliorer l'efficacité, que ce soit par des méthodes d'élimination des parcours parallèles d'une même structure, de spécialisation de programme ou d'élimination de structures intermédiaires. Ce dernier type de transformation a été baptisé *déforestation* par Wadler [Wad88], puisqu'il permet *d'éliminer des arbres.* Le terme arbre désigne ici les structures de données intermédiaires qui sont "inutiles" au sens où elles ne font pas partie du résultat final du programme. Il peut s'agir d'arbres binaires, de listes ou de toute autre structure de donnée basée sur un type abstrait, même si les listes ont fait l'objet d'un intérêt particulier de par leur importance dans les langages fonctionnels.

Les règles de transformation de dépliage/pliage ont également été étudiées sous la forme de règles de réécriture de termes par Reddy dans le système FOCUS [Red88]. Dans cette optique, et à partir des travaux de Dershowitz [Der88], Bellegarde a développé cette méthode de transformation de programmes basée sur des outils de réécriture [Bel91b]. Cette approche lui a permis de donner une nouvelle formulation des règles d'inférence de Burstall et Darlington et d'utiliser une dérivation des *procédures de complétions partielles* permettant de prouver la correction du processus de transformation [Bel91b]. La combinaison de techniques d'évaluation partielle [CD93], de réécriture et de *fusion de boucles* [2] [Chi94] a abouti à des transformations automatiques de programmes prises en charges par le système ASTRE [Bel91a, Bel93, Bel95b, Bel95a].

Notons au passage que principe des règles de dépliage/pliage a également été largement étudié en tant que méthode de spécialisation et d'optimisation de programmes logiques, principalement par Pettorossi et Proietti [PP96, PP97, PPR97].

L'algorithme de déforestation proposé par Wadler [Wad88] permet d'effectuer automatiquement de la déforestation sur une classe de termes restreints appelés *treeless.* Outre les différentes extensions et améliorations qui ont été apportées à cet algorithme pour étendre la classe de termes acceptés [CK95, Ham96, SS97a, SGJ94], d'autres formalismes ont été proposés pour étendre la puissance de ces transformations de programmes fonctionnels [FSS92,

2. J'ai traduit par *fusion de boucles* l'expression anglaise *two-loops fusion* qui représente la mise en commun dans un tuple plusieurs fonctions qui parcourent parallèlement une même structure.

GLJ93, SF93, FSZ94, TM95]. Je me suis particulièrement intéressé aux méthodes algébriques qui sont basées sur un style de programmation *dirigé par la structure*, où les fonctions sont décrites avec des schémas de récursion explicites. Simultanément aux travaux de Gill, Launchbury et Peyton Jones sur la déforestation des listes par application automatique de règles d'élimination `foldr/build`, implantés dans Haskell [GLJ93], Sheard et Fegaras ont introduit l'opérateur de contrôle générique *fold* [SF93, FSZ94], qui permet de déforester de façon générique différents types de données (listes, arbres ou autres). Ces méthodes sont fondées sur un style de programmation algébrique et sur des résultats de la théorie des catégories [BW88, MFP91, Fok95, Mee95]. L'opérateur *fold* permet, à l'aide de l'*algorithme de normalisation*, d'effectuer de la *déforestation* ou de la *fusion* [3] de programmes.

D'autres travaux, eux aussi fondés sur la théorie des catégories, ont été initiés par Meijer et Takano qui ont introduit le terme générique de *déforestation en forme calculationnelle* [4] [TM95] pour regrouper ces formalismes algébriques. Ces derniers travaux ont récemment abouti à un traitement de déforestation automatique pour une large sous-classe de programmes fonctionnels, dans le système HYLO développé par Hu, Iwasaki et Takeichi [HIT96b, OHIT97].

Dans la suite de ce chapitre, je commence par faire la présentation de l'algorithme de déforestation introduit par Wadler ; je présente également un rapide survol des différentes extensions de cet algorithme qui ont été étudiées, ainsi que les règles d'élimination `foldr/build` de Gill *et al.*, qui donnent une bonne intuition des transformations présentées ensuite.

La seconde approche que je présente est celle des *folds*, avec leur *algorithme de normalisation* [SF93].

Comme la dernière approche calculationnelle que je présente fait référence à des notions issues de la théorie des catégories, je propose alors un rappel sur les formalismes algébriques.

Munis de ces notions, je présente donc le formalisme des hylomorphismes et leurs théorèmes de fusion [TM95].

Ce choix de présentation est en fait assez représentatif de la démarche adoptée pour l'étude que j'ai menée. Le concept de déforestation puis les *folds* m'ont permis, de par leurs analogies avec la composition descriptionnelle (cf. chapitre 3) et les grammaires attribuées, de comprendre le fonctionnement et le principe de ces manipulations d'un programme récursif dirigé par la structure. Ensuite, les transformations liées aux lois fondamentales de la théorie des catégories et relatives aux hylomorphismes me sont apparues plus aisées à appréhender. La présentation que je fais de ces différents formalismes dans la suite de ce chapitre se veut volontairement didactique pour faciliter leur compréhension à un lecteur peu familier. Ce chapitre peut être vu comme un *survey* de ces trois méthodes de déforestation fonctionnelles ; aussi, pour les détails de chaque méthode de déforestation, je renverrai aux publications des auteurs et, pour les notions catégorielles, je conseille les travaux de Meijer [MFP91], de Fokkinga [Fok95] ou de Meerteens [Mee95].

3. Le terme de fusion vient du fait qu'on transforme la composition de deux fonctions en une seule dans laquelle des constructions de structures intermédiaires ont été éliminées.

4. Le terme anglophone, introduit dans [TM95], dû aux bases théoriques de ces travaux dans l'Algorithmique Constructive, est *deforestation in calculational form*.

```
let append l1 l2 =
  case l1 of
    cons head tail → cons head (append tail l2)
    nil → l2
```

Fig. 5.1: *La fonction append qui concatène deux listes*

```
let tri_append l1 l2 l3 =
  case l1 of
    cons head tail → cons head (tri_append tail l2 l3)
    nil → append l2 l3
```

Fig. 5.2: *La fonction tri_append qui concatène trois listes.*

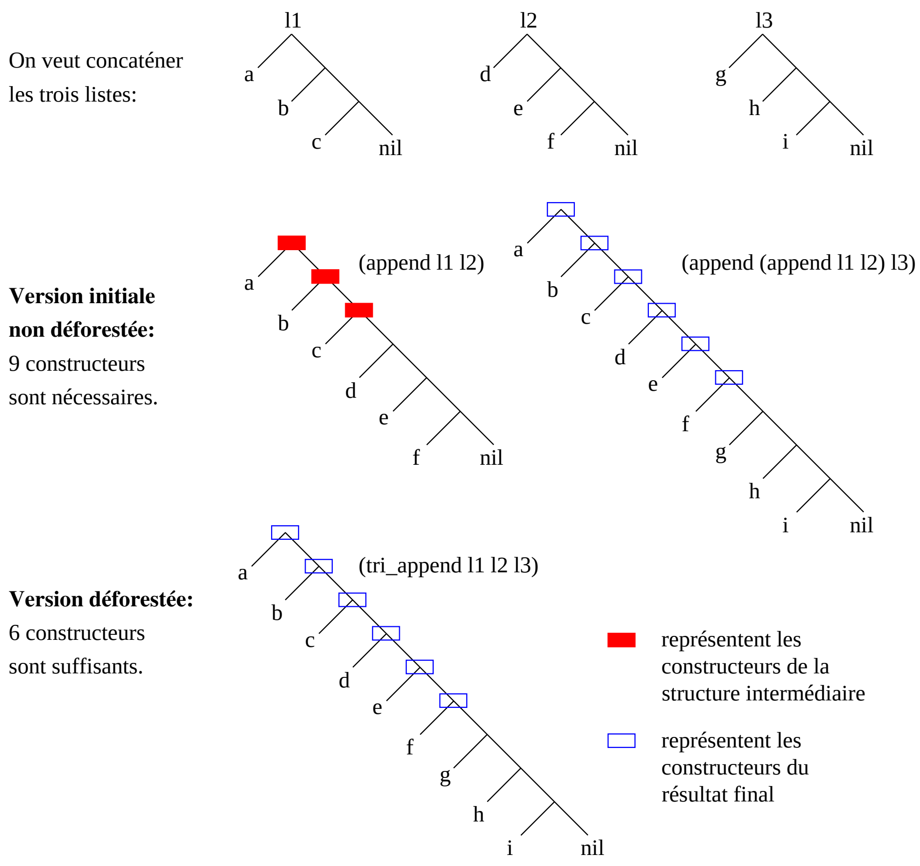

Fig. 5.3: *Les bienfaits de la déforestation*

5.2 Déforestation de Wadler

Ce chapitre présente donc l'algorithme de déforestation de Wadler. Je rappelle que l'idée de la déforestation est née de la constatation que l'élégance des programmes fonctionnels est confrontée à des problèmes d'efficacité. Considérons par exemple la définition de la fonction *append* qui concatène deux listes l_1 et l_2, présentée figure 5.1. Considérons alors la concaténation de trois listes l_1, l_2 et l_3 obtenue par la composition :

$$append\ (append\ l_1\ l_2)\ l_3 \qquad \heartsuit$$

La concaténation de l_1 et de l_2 construit une liste intermédiaire à laquelle l_3 est concaténée. L'allocation et la désallocation de cette structure intermédiaire est coûteuse au moment de l'exécution du programme. En sacrifiant la lisibilité à l'efficacité, il serait préférable de choisir plutôt la définition présentée figure 5.2 pour obtenir le même résultat en appelant $tri\_append\ l_1\ l_2\ l_3$.

En effet, dans cette version, la liste intermédiaire n'est pas construite ; Seidl et Sørensen rapportent dans [SS97a] qu'avec le langage *Gofer* de Marc Jones par exemple, le premier programme est plus coûteux que le second d'environ 7% en place mémoire et 13% en temps d'exécution. La figure 5.3 montre l'application de ces deux programmes sur le même exemple.

Toute l'idée de la déforestation tient dans le principe originel exprimé par Burstall et Darlington [BD77] : écrire un programme clair (mais inefficace) et le transformer en un programme efficace (quitte à ce qu'il soit moins clair). Plus précisément pour notre exemple, on voudrait permettre à l'utilisateur d'écrire la première version du programme ($\heartsuit$) et qu'un mécanisme la traduise automatiquement dans la seconde version (figure 5.2), plus efficace.

5.2.1 Le langage

Pour présenter l'algorithme de déforestation de Wadler, je me place dans le même contexte que lui et je considère maintenant un langage fonctionnel du premier ordre dans une syntaxe décrite à la figure 5.4 [Wad88].

Dans l'application d'un constructeur ou d'une fonction, on appelle les termes $t_1, \ldots, t_l$ les *arguments*. Dans un terme `case`, t_0 est appelé le *sélecteur* et $p_1 \rightarrow t_1, \ldots, p_n \rightarrow t_n$ sont les *branches*. Les définitions de fonctions sont de la forme $f\ v_1 \ldots v_k = t$.

def	$::=$	$\texttt{let}\ f\ t_1 \ldots t_m = t$	définition de fonction
t	$::=$	v	variable
	$\mid$	$c\ t_1 \ldots t_i$	application d'un constructeur c
	$\mid$	$f\ t_1 \ldots t_j$	application d'une fonction f
	$\mid$	$\texttt{case}\ t_0\ \texttt{of}$	terme `case`
		$p_1 \rightarrow t_1$	filtre syntaxique
		$\ldots$	(*pattern matching*)
		$p_n \rightarrow t_n$	
p	$::=$	$c\ v_1 \ldots v_k$	(*pattern*)

FIG. 5.4: *Syntaxe du langage fonctionnel*

$\texttt{let}$ *flip* $t =$
  $\texttt{case}$ t $\texttt{of}$
    node left right $\rightarrow$ *node* (*flip right*) (*flip left*)
    leaf $n \rightarrow$ *leaf* n

FIG. 5.5: *La fonction flip*

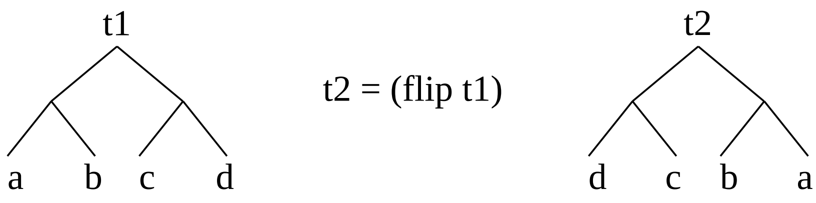

FIG. 5.6: *La fonction flip sur un exemple*

Pour illustrer ce langage, outre la définition de *append* présentée figure 5.1, je donne dans la figure 5.5 la définition de la fonction *flip* qui échange récursivement chacune des deux branches d'un arbre binaire, comme illustré figure 5.6.

Habituellement, un terme est dit *linéaire* s'il ne contient pas plusieurs fois la même variable. Par exemple, *append* l_1 (*append* l_2 l_3) est linéaire, tandis que *append* l_1 l_1 ne l'est pas. Dans le cadre de la déforestation de Wadler présentée ici, cette définition de la linéarité est assouplie (cf. définition 5.2.1) pour les termes `case` par la condition suivante : une variable ne peut pas apparaître à la fois dans une branche et dans le sélecteur, ni plusieurs fois dans une même branche. Ainsi, la définition de *append* est linéaire, même si l_2 apparaît dans chaque branche.

Définition 5.2.1 (Terme linéaire) *Un terme t est dit **linéaire** dans le cadre de la déforestation de Wadler si :*

- *il est de la forme $c\ t_1 \dots t_n$ ou $f\ t_1 \dots t_n$: dans ce cas la même variable n'apparaît pas plusieurs fois dans ce terme.*

- *il est de la forme* `case` t_0 `of` $p_1 \rightarrow t_1 \dots p_n \rightarrow t_n$ *: dans ce cas t_0 n'apparaît pas dans les termes des branches $t_1, \dots, t_n$ et la même variable n'apparaît pas plusieurs fois dans un même terme de branche t_i. En revanche, une même variable peut apparaître en même temps dans deux branches différentes t_i et t_j avec $i \neq j$.*

□

5.2.2 La forme *treeless*

L'algorithme proposé initialement par Wadler [Wad88] élimine des structures de données intermédiaires pour des programmes fonctionnels du premier ordre vérifiant des restrictions basées sur la notion de terme *treeless*, définie comme suit.

Définition 5.2.2 (Terme *treeless*) *Soit F un ensemble de noms de fonctions. Un terme tt du langage est dit* ***treeless*** *par rapport à F si :*

1. *il est linéaire ;*
2. *il ne contient que des appels de fonctions de F ;*
3. *tous les arguments des applications de fonctions dans tt sont des variables ;*
4. *tous les* sélecteurs *de terme* `case` *dans tt sont des variables ;*

Un ensemble de définitions de fonctions F est dit ***treeless*** *si chaque terme définissant une fonction est* ***treeless*** *par rapport à F.*
□

- La contrainte *2* permet de définir de façon inductive la notion de *treeless*.
- Les contraintes *3* et *4* garantissent qu'aucune structure construite dans le terme n'est consommée dans celui-ci par l'appel d'une fonction. En particulier, cela interdit les termes tels que (*flip* (*flip t*)) dans lequel (*flip t*) est un arbre intermédiaire. Par contre, les applications de constructeurs ne sont pas soumises aux mêmes restrictions que les applications de fonctions dans la restriction *3*. Cette distinction autorise les termes tels que (*node* (*flip t_1*) (*flip t_2*)) où les arbres retournés par (*flip t_1*) et (*flip t_2*) ne sont pas des arbres intermédiaires, mais sont bel et bien des *parties du résultat*. Autrement dit, toute consommation (dépliage) de structure est effectuée par un *pattern matching* “visible”, c'est-à-dire à l'extérieur du terme.
- La contrainte *1* garantit que la transformation de certains programmes n'introduit pas de répétitions de calculs dans la phase de “dépliage” [BD77], c'est-à-dire le remplacement de la partie gauche d'une équation par sa partie droite (remplacement d'un appel de fonction par le terme qui la définit). Un tel dépliage d'une fonction qui a une définition (partie droite) non linéaire risquerait de dupliquer un terme potentiellement coûteux à calculer. Prenons par exemple la fonction non linéaire classique $square\ x = x * x$. Il est en effet préférable que le programme contienne $square\ t$ plutôt que son équivalent déplié $t * t$. Par contre, la définition $square\ x = exp(2 * log\ x)$ est linéaire et il est alors possible de déplier sans risque. Une telle propriété de linéarité permet donc de déplier sans perdre d'efficacité.

Notons pour finir que “être *treeless*” est la propriété d'une définition de fonction et non d'une fonction elle-même. Considérons par exemple le type des listes de listes de α, soit $list\ (list\ \alpha)$, et supposons que l'on veuille aplatir une telle liste de listes en une liste simple, soit $list\ \alpha$, en concaténant chacun de ses élément. La figure 5.7 présente, pour la même fonction qui aplatit une liste de listes, une définition $flatlist_0$ qui n'est pas *treeless* et une définition $flatlist_1$ qui est *treeless*. En effet, dans $flatlist_0$, pour le *pattern cons head tail*, le second argument de la fonction *append* n'est pas une variable, mais un appel de fonction (*flatlist$_0$ tail*). Cela ne satisfait donc pas la contrainte *3* de la définition 5.2.2. Par contre, dans la définition de la fonction $flatlist_1$, les paramètres *head* et *tail* de l'appel à la fonction *flatlist$'_1$* sont des variables et dans la définition de *flatlist$'_1$*, le terme (*flatlist$'_1$ tail ll*), qui n'est pas une variable, est l'argument d'un *constructeur* et non d'une fonction. La définition 5.2.2 qualifie donc ces définitions de fonctions de *treeless*.

$\mathit{flatlist_0} : \mathit{list}\ (\mathit{list}\ \alpha) \rightarrow \mathit{list}\ \alpha$
`let` $\mathit{flatlist_0}\ \mathit{ll} =$
  `case` ll `of`
    $\mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \mathit{append}\ \mathit{head}\ (\mathit{flatlist_0}\ \mathit{tail})$
    $\mathit{nil} \rightarrow \mathit{nil}$

$\mathit{flatlist_1} : \mathit{list}\ (\mathit{list}\ \alpha) \rightarrow \mathit{list}\ \alpha$
`let` $\mathit{flatlist_1}\ \mathit{ll} =$
  `case` ll `of`
    $\mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \mathit{flatlist'_1}\ \mathit{head}\ \mathit{tail}$
    $\mathit{nil} \rightarrow \mathit{nil}$
avec
$\mathit{flatlist'_1} : \mathit{list}\ \alpha \rightarrow \mathit{list}\ (\mathit{list}\ \alpha) \rightarrow \mathit{list}\ \alpha$
`let` $\mathit{flatlist'_1}\ l\ \mathit{ll} =$
  `case` l `of`
    $\mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \mathit{cons}\ \mathit{head}\ (\mathit{flatlist'_1}\ \mathit{tail}\ \mathit{ll})$
    $\mathit{nil} \rightarrow \mathit{nil}$

FIG. 5.7: *Définitions treeless et non-treeless pour une même fonction*

À partir de cette notion de définition de fonction *treeless*, Wadler énonce le théorème suivant [Wad88] :

Théorème 5.2.1 (Théorème de déforestation) *Toute composition de fonctions ayant des définitions treeless peut être transformée en une seule fonction ayant une définition treeless, sans perte d'efficacité*[5].
□

5.2.3 Algorithme de déforestation

La transformation annoncée par le théorème de déforestation est effectuée par l'ensemble de règles de transformations présenté à la figure 5.8.

La caractérisation des définitions *treeless* et les hypothèses du théorème de déforestation étant purement syntaxiques, il est facile de savoir quand la transformation est applicable. Par ailleurs, l'utilisateur n'a pas besoin d'être familier avec les détails des règles en elles-mêmes. La déforestation de base [Wad88] est effectuée grâce aux sept règles de transformation de la figure 5.8. On note par $T[\![t]\!]$ le résultat de la conversion d'un terme t dans sa forme *treeless*. On doit donc avoir, du point de vue de la sémantique du terme t,

$$t = T[\![t]\!]$$

c'est-à-dire que t et $T[\![t]\!]$ doivent calculer la même valeur. Le principe (très général) de cette déforestation est en fait de faire passer les constructeurs vers l'extérieur (vers la gauche) des termes et les fonctions vers l'intérieur. Cela a pour effet de retarder au maximum, lors d'un

5. Dans ce cadre, la sémantique opérationnelle du langage est la réduction de graphe dans l'ordre normal (de gauche à droite et en profondeur d'abord). Un terme est dit plus efficace qu'un autre si, pour chaque instanciation possible des variables libres, le premier requiert moins de pas de réduction que le second.

$$
\begin{array}{ll}
(1) & T[\![v]\!] = v \\[2ex]
(2) & T[\![c\ t_1 \ldots t_k]\!] = c\ (T[\![t_1]\!]) \ldots (T[\![t_k]\!]) \\[2ex]
(3) & T[\![f\ t_1 \ldots t_k]\!] = T[\![t[t_1/v_1, \ldots, t_k/v_k]]\!] \\
 & \qquad \text{où } f \text{ est défini par } f\ v_1\ \ldots\ v_k\ = t \\[2ex]
(4) & T[\![\texttt{case}\ v\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n]\!] \\
 & \quad = \texttt{case}\ v\ \texttt{of}\ \ p'_1 \to T[\![t'_1]\!] \ldots p'_n \to T[\![t'_n]\!] \\[2ex]
(5) & T[\![\texttt{case}\ c\ t_1 \ldots t_k\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n]\!] \\
 & \quad = T[\![t'_i[t_1/v_1, \ldots, t_k/v_k]]\!] \\
 & \quad \text{où } p'_i = c\ v_1 \ldots v_k \\[2ex]
(6) & T[\![\texttt{case}\ f\ t_1 \ldots t_k\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n]\!] \\
 & \quad = T[\![\texttt{case}\ t[t_1/v_1, \ldots, t_k/v_k]\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n]\!] \\
 & \quad \text{où } f \text{ est défini par } f\ v_1\ \ldots\ v_k\ = t \\[2ex]
(7) & T[\![\texttt{case}\ (\texttt{case}\ t_0\ \texttt{of}\ \ p_1 \to t_1 \ldots p_m \to t_m)\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n]\!] \\
 & \quad = T[\![\texttt{case}\ t_0\ \texttt{of} \\
 & \qquad\qquad p_1 \to (\texttt{case}\ t_1\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n) \\
 & \qquad\qquad \ldots \\
 & \qquad\qquad p_m \to (\texttt{case}\ t_m\ \texttt{of}\ \ p'_1 \to t'_1 \ldots p'_n \to t'_n)]\!]
\end{array}
$$

FIG. 5.8: *Règles de transformations de l'algorithme de déforestation.*

calcul, la construction d'une structure et d'éviter de la manipuler (par une fonction) après sa construction. Ceci est bien dans l'esprit de la suppression des structures intermédiaires.

Cependant, la terminaison de l'algorithme tel qu'il est présenté ici ne peut pas être assurée. Il est donc nécessaire d'introduire, en plus, un mécanisme de renommage des expressions qui, lorsqu'elles sont reconnues comme ayant déjà été rencontrées par l'algorithme, donnent lieu à des remplacements par des fonctions récursives (phase de “pliage” dans [BD77]).

Considérons par exemple le terme *flip* (*flip* t) qui, intuitivement, aboutit à une fonction qui laisse intact un arbre binaire passé en paramètre. Je vais dérouler pas à pas l'application des règles de transformation sur cet exemple :

$$
\begin{array}{rl}
 & T[\![\mathit{flip}\ (\mathit{flip}\ t)]\!] \\
\overset{(3)}{=} & T[\![\texttt{case}\ (\mathit{flip}\ t)\ \texttt{of} \\
 & \qquad \mathit{node}\ \mathit{left}'\ \mathit{right}' \to \mathit{node}\ (\mathit{flip}\ \mathit{right}')\ (\mathit{flip}\ \mathit{left}') \\
 & \qquad \mathit{leaf}\ \ n' \to \mathit{leaf}\ \ n']\!]
\end{array}
$$

Le terme initial étant un appel de fonction, il faut appliquer la règle (3), puis, le sélecteur apparaissant étant lui même un appel de fonction, il faut appliquer la règle (6), qui a pour effet de *déplier* cet appel de fonction :

$$\overset{(6)}{=} \quad T[\![\texttt{case}\ (\texttt{case}\ t\ \texttt{of}\\
\qquad\qquad \mathit{node\ left\ right} \rightarrow \mathit{node}\ (\mathit{flip\ right})\ (\mathit{flip\ left})\\
\qquad\qquad \mathit{leaf}\ \ n \rightarrow \mathit{leaf}\ \ n)\ \texttt{of}\\
\qquad \mathit{node\ left'\ right'} \rightarrow \mathit{node}\ (\mathit{flip\ right'})\ (\mathit{flip\ left'})\\
\qquad \mathit{leaf}\ \ n' \rightarrow \mathit{leaf}\ \ n']\!]$$

En cas de terme `case` apparaissant dans le sélecteur, l'application de la règle (7) a pour effet de *distribuer* le `case` externe dans les branches du `case` du sélecteur interne, pour chacun de ses *patterns* :

$$\overset{(7)}{=} \quad T[\![\texttt{case}\ t\ \texttt{of}\\
\quad \mathit{node\ left\ right} \rightarrow (\texttt{case}\ \mathit{node}\ (\mathit{flip\ right})\ (\mathit{flip\ left})\ \texttt{of}\\
\qquad\qquad \mathit{node\ left'\ right'} \rightarrow \mathit{node}\ (\mathit{flip\ right'})\ (\mathit{flip\ left'})\\
\qquad\qquad \mathit{leaf}\ \ n' \rightarrow \mathit{leaf}\ \ n'\ )\\
\quad \mathit{leaf}\ \ n \rightarrow (\texttt{case}\ \mathit{leaf}\ \ n\ \texttt{of}\\
\qquad\qquad \mathit{node\ left'\ right'} \rightarrow \mathit{node}\ (\mathit{flip\ right'})\ (\mathit{flip\ left'})\\
\qquad\qquad \mathit{leaf}\ \ n' \rightarrow \mathit{leaf}\ \ n'\ )]\!]$$

L'application de la règle (4), dans le cadre d'un `case` portant sur une variable, reporte alors l'application de l'algorithme sur les termes de ses branches :

$$\overset{(4)}{=} \quad \texttt{case}\ t\ \texttt{of}\\
\quad \mathit{node\ left\ right} \rightarrow T[\![(\texttt{case}\ \mathit{node}\ (\mathit{flip\ right})\ (\mathit{flip\ left})\ \texttt{of}\\
\qquad\qquad \mathit{node\ left'\ right'} \rightarrow \mathit{node}\ (\mathit{flip\ right'})\ (\mathit{flip\ left'})\\
\qquad\qquad \mathit{leaf}\ \ n' \rightarrow \mathit{leaf}\ \ n'\ )]\!]\\
\quad \mathit{leaf}\ \ n \rightarrow T[\![(\texttt{case}\ \mathit{leaf}\ \ n\ \texttt{of}\\
\qquad\qquad \mathit{node\ left'\ right'} \rightarrow \mathit{node}\ (\mathit{flip\ right'})\ (\mathit{flip\ left'})\\
\qquad\qquad \mathit{leaf}\ \ n' \rightarrow \mathit{leaf}\ \ n'\ )]\!]$$

Pour chacune des branches du `case` externe, il est alors possible d'appliquer la règle (5) qui *spécialise* en quelque sorte l'application d'un `case` interne, en fonction de chacun des *patterns* dans lequel il intervient :

$$\overset{(5),(5)}{=} \quad \texttt{case}\ t\ \texttt{of}\\
\quad \mathit{node\ left\ right} \rightarrow T[\![\mathit{node}\ (\mathit{flip}\ (\mathit{flip\ left}))\ (\mathit{flip}\ (\mathit{flip\ right}))]\!]\\
\quad \mathit{leaf}\ \ n \rightarrow T[\![\mathit{leaf}\ \ n]\!]$$

À ce stade, l'application des règles (1) et (2) pour les applications de constructeurs ou pour les variables permettent d'obtenir le terme suivant :

$$\overset{(2),(1),(2)}{=} \quad \texttt{case}\ t\ \texttt{of}\\
\quad \mathit{node\ left\ right} \rightarrow\\
\qquad \mathit{node}\ \ (T[\![(\mathit{flip}\ (\mathit{flip\ left}))]\!])\ \ (T[\![(\mathit{flip}\ (\mathit{flip\ right}))]\!])\\
\quad \mathit{leaf}\ \ n \rightarrow \mathit{leaf}\ \ n$$

Il est alors primordial pour la terminaison de l'algorithme de reconnaître dans ce terme deux renommages du terme initial à déforester, sous peine de voir l'algorithme boucler infiniment. En effet, les pas d'application des règles précédemment détaillées sur *flip* (*flip* t)

$$
\begin{array}{l}
\quad T[\![append\ (append\ l_1\ l_2)\ l_3]\!] = h_0\ l_1\ l_2\ l_3 \\
\text{avec} \\
\quad \texttt{let}\ \ h_0\ l_1\ l_2\ l_3 = \texttt{case}\ l_1\ \texttt{of} \\
\qquad\quad cons\ head\ tail \rightarrow cons\ head\ (h_0\ tail\ l_2\ l_3) \\
\qquad\quad nil \rightarrow h_1\ l_2\ l_3 \\
\text{et} \\
\quad \texttt{let}\ \ h_1\ l_2\ l_3 = \texttt{case}\ l_2\ \texttt{of} \\
\qquad\quad cons\ head\ tail \rightarrow cons\ head\ (h_1\ tail\ l_3) \\
\qquad\quad nil \rightarrow l_3
\end{array}
$$

FIG. 5.9: *Déforestation de append* $(append\ l_1\ l_2)\ l_3$.

$$
\begin{array}{l}
\quad T[\![flip\ (flip\ t)]\!] = h_0\ t \\
\text{avec} \\
\quad \texttt{let}\ \ h_0\ t = \texttt{case}\ t\ \texttt{of} \\
\qquad\quad node\ left\ right \rightarrow node\ (h_0\ left)\ (h_0\ right) \\
\qquad\quad leaf\ \ n \rightarrow leaf\ \ n
\end{array}
$$

FIG. 5.10: *Déforestation de flip* $(flip\ t)$.

---

peuvent s'appliquer à nouveau sur $(flip\ (flip\ left))$ et sur $(flip\ (flip\ right))$; l'algorithme peut ainsi boucler infiniment si on ne reconnaît pas que cette application de règle a déjà eu lieu sur une expression équivalente.

La méthode de Wadler consiste alors à établir, au fur et à mesure de l'application de l'algorithme, une liste des termes sur lesquels la déforestation est tentée et d'essayer de reconnaître, à chaque nouveau terme apparaissant, un renommage de l'un de ces termes.

C'est le cas dans notre exemple, où une nouvelle fonction h_0 doit être introduite, qui vérifie

$$h_0\ t = T[\![flip\ (flip\ t)]\!]$$

Les deux occurrences de $T[\![\ldots]\!]$ dans le dernier terme obtenu ci-dessus sont alors des renommages de la partie droite de cette définition de fonction, et peuvent alors être remplacés par la partie gauche (c'est le "pliage") : ceci conduit à la définition suivante de h_0 :

$$
\begin{array}{l}
\texttt{let}\ \ h_0\ t = \texttt{case}\ t\ \texttt{of} \\
\qquad node\ left\ right \rightarrow node\ (h_0\ left)\ (h_0\ right) \\
\qquad leaf\ \ n \rightarrow leaf\ \ n
\end{array}
$$

Cette étape complète la transformation, puisqu'elle permet d'exhiber une définition *treeless* pour une fonction h_0 équivalente au terme $flip\ (flip\ t)$ (cf. figure 5.10).

Par ailleurs, si le théorème assure qu'il n'y a pas de perte d'efficacité, il y a en fait un gain lorsque le terme initial contient des structures intermédiaires. Par exemple, pour les définitions des fonctions *append* et *flip* présentées aux figures 5.1 et 5.5, les termes $append\ (append\ l_1\ l_2)\ l_3$ et $flip\ (flip\ t)$ satisfont les hypothèses du théorème et sont transformés comme l'illustrent les figures 5.9 et 5.10.

Dans la figure 5.9, on peut reconnaître par de simples renommages les définitions précédemment énoncées des fonctions *tri_append* pour h_0 et *append* pour h_1. L'effet de la déforestation est d'éliminer la construction de la structure intermédiaire (*append* l_1 l_2) comme déjà illustré à la figure 5.3.

L'exemple de la déforestation de *flip* (*flip t*) est particulièrement probant (figure 5.10), puisqu'il produit automatiquement la définition d'une fonction de recopie d'un arbre.

5.2.4 Différentes extensions ou améliorations

L'algorithme de déforestation de Wadler que je viens de présenter comporte deux problèmes. Premièrement, la terminaison de l'algorithme nécessite un renommage et une reconnaissance des termes qui ne sont pas complètement satisfaisants. Deuxièmement, la condition de *treeless* est trop restrictive sur les programmes d'entrée de la déforestation. Par ailleurs, je n'ai pas parlé de l'extension concernant les fonctions d'ordre supérieur, mais l'algorithme de Wadler ne permet de prendre en compte que certaines fonctions d'ordre supérieur simples pouvant être exprimées sous la forme de macros non récursives [Wad88]. Je donne dans cette section un bref aperçu des différentes améliorations qui ont été apportées à cet algorithme par la technique de *marquage*.

Ensuite, je présente une autre méthode d'élimination de structures intermédiaires qui ne pose pas ce genre de problèmes, même si elle est limitée au type des listes : il s'agit des règles d'élimination `foldr/build`.

Déforestation marquée

L'algorithme de base, tel que présenté dans la section précédente, a donné lieu à de nombreuses extensions, dont celles de Wadler lui-même [Wad88, Wad90]. La technique de *marquage* [6] consiste à marquer certains termes qui seront ensuite ignorés par les transformations de l'algorithme. L'avantage de cette méthode est d'éviter de déforester des termes ne contenant pas de constructeurs de structures intermédiaires, ou des termes pouvant poser des problèmes à l'algorithme. Tout le problème consiste alors à déterminer les critères de marquage, mais cette technique a permis d'introduire différentes conditions assurant l'efficacité, la sûreté ou la terminaison de la déforestation.

Wadler détermine les critères de marquage à partir du type des termes [Wad88, Wad90]. Par exemple, un terme de type entier ne produit pas de résultat étant une structure de données : marquer ce sous-terme ne fait pas perdre d'efficacité à la déforestation. Je cite ici quelques-unes des extensions ou améliorations de cette méthode. Chin et Khoo ont introduit différentes techniques de marquage des termes qui violent le critère de *treeless*, élargissant ainsi la classe des termes acceptés par l'algorithme ; ils ont développé différentes extensions où les arguments de fonctions non récursives peuvent être marqués et où des conditions syntaxiques peuvent être remplacées par des conditions sémantiques [CK95]. Hamilton et Jones ont développé des analyses statiques permettant d'obtenir des conditions de marquage similaires à celles de Chin, en utilisant des analyses de comptage du nombre d'utilisation des termes [HJ91, Ham96]. Sørensen et Seidl ont développé une méthode d'analyse de flot de contrôle, basée sur des contraintes, qui permet d'approximer la terminaison de l'algorithme de déforestation et d'utiliser les informations de cette analyse pour le marquage [Sør94, Sei96, SS97a, SS97b].

Ces derniers auteurs ont par ailleurs étudié des rapprochements entre les méthodes de déforestation, de super-compilation et d'évaluation partielle [Tur86, GJ94, SGJ94]. Ces travaux

6. De l'anglo-saxon *blazed deforestation*.

ne seront pas présentés dans cette thèse.

Un raccourci de déforestation : la règle `foldr/build`

Je présente brièvement ici une méthode développée par Gill, Launchbury et Jones, qui permet d'éliminer la construction de listes intermédiaires produites dans des programmes fonctionnels [GLJ93, Gil96]. Cette méthode n'est pas directement une application de la méthode de Wadler, mais plutôt un cas particulier, dédié aux listes, des méthodes calculationnelles qui seront présentées dans les sections 5.3 et 5.5.

L'intérêt de cette méthode et la raison pour laquelle je la présente ici sont qu'elle permet d'avoir une idée intuitive de ce qu'est un *consommateur* et un *producteur* de structure, comme en sont le *folds* qui sera présenté dans la section 5.3 et son opérateur dual, appelé ici `build`. Par ailleurs, le principe d'une règle automatique permettant d'effectuer de la déforestation sur une représentation particulière d'un programme sera approfondi par les théorèmes de fusion des hylomorphismes, à la section 5.5.

Cette technique de déforestation spécifique pour les listes a été implantée dans le compilateur GHC du langage de programmation Haskell [HJW$^+$92, GLJ93]. Pour cette raison, j'utilise les notations de ce langage, où le constructeur de liste (*cons x xs*) est noté par `(x:xs)`, la liste vide (*nil*) est notée `[]`, et les λ-abstractions ($\lambda\ a\ b\ .\ (f\ a\ b)$) sont notées `\a b -> (f a b)`.

La définition du *consommateur* de liste représenté par l'opérateur `foldr` est la suivante :

```
foldr k z []      =  z
foldr k z (x:xs)  =  k x (foldr k z xs)
```

Dans cette définition, le consommateur de liste `foldr` accepte en paramètre une opération binaire `k`, une valeur `z` et une liste sur laquelle il s'applique. Son effet est de remplacer tous les constructeurs `(x:xs)` de la liste par l'application de `k` sur `x` et sur l'appel récursif du `foldr k z` sur `xs` ; par ailleurs, le constructeur `[]` est remplacé par la valeur `z`.

Quelques exemples de définitions de fonctions classiques permettent de se familiariser avec cette notation. La fonction `sum` calcule la somme des éléments d'une liste, `append` concatène deux listes et `map f` applique la fonction `f` à tous les éléments d'une liste :

```
sum xs        =  foldr (+) 0 xs
append xs ys  =  foldr (:) ys xs
map f xs      =  foldr (\a b -> (f a) : b) [] xs
```

De manière duale, l'opérateur de *production* de liste est défini comme ceci :

```
build g = g (:) []
```

Intuitivement, `g` représente le schéma de récursion d'une liste, dans lequel les opérateurs de constructions `(:)` et `[]` sont abstraits. Les fournir en argument au `build` revient à instancier ces abstractions et donc à construire une liste. C'est en effet ce qui permet de définir l'équivalence des opérateurs `foldr` et `build` dans la règle (•) suivante, où pour tout [7] `g`, `k` et `z` :

7. Il existe des restrictions sur la fonction `g` qui ne sont pas détaillées ici. Le lecteur est invité à se référer à [GLJ93] pour plus de détails.

```
[foldr] k z ([build] g) = g k z             (•)
```

Cette règle est appelée *règle d'élimination* `foldr/build`.

Je considère comme exemple la fonction `from`, qui prend deux entiers `a` et `b` en paramètres et qui construit la liste des entiers de `a` jusqu'à `b`. Cette fonction peut être définie par :

```
from a b  =  if a>b
             then []
             else a : from (a+1) b
```

Alors, l'abstraction des constructeurs `(:)` et `[]` par les variables liées `c` et `n` permet de définir la fonction `from'` suivante :

```
from' a b  =  \c n ->  if a>b
                       then n
                       else c a from' (a+1) b
```

Typiquement, la fonction `from` peut être obtenue par l'application de `build` à `from'` :

```
from a b = build (from' a b)
```

L'idée de la déforestation par application de règles `foldr/build` est simplement d'appliquer cette propriété lors de la composition d'une fonction exprimée par un `foldr` avec une autre exprimée par un `build`. Je prends maintenant l'exemple du calcul de la somme des éléments de la liste des entiers de `a` jusqu'à `b`. Cette composition, `sum (from a b)`, peut s'écrire grâce aux opérateurs `foldr` et `build` ; la propriété (•) permet alors d'effectuer des simplifications dans cette composition, comme illustré ci-dessous :

```
sum (from a b)  =  foldr (+) 0 (from a b)
                =  [foldr](+) 0 ([build] (from' a b))
                =  (from' a b) (+) 0
```

Cette dernière forme ne construisant plus la liste intermédiaire, elle a bien été déforestée par l'application de la règle `foldr/build`.

Cette technique donne de bons résultats de déforestation, mais uniquement pour des fonctions définies sur des listes. Toutefois, elle nécessite de disposer des définitions en `foldr` et en `build` des fonctions qui sont impliquées dans la composition, ce qui n'est pas nécessairement évident. Dans le cas de l'implantation qui a été faite de ce mécanisme dans le compilateur du langage Haskell, une bibliothèque des versions en `foldr` et en `build` des fonctions les plus simples ou les plus couramment utilisées a été mise en place ; l'optimisation de déforestation remplace alors ces fonctions par leur définition en `foldr` ou en `build`, en fonction du contexte de composition, pour pouvoir appliquer la règle d'élimination.

Il est important de signaler ici qu'une même fonction peut être vue à la fois comme un `foldr` et comme `build` ; par exemple, la fonction `map`, déjà présentée en tant que `foldr`, peut s'exprimer avec un `build` de la façon suivante :

```
map f xs = build(\c n -> foldr (\a b -> c (f a) b) n xs)
```

Cette caractéristique sera rappelée plus tard, en particulier à la section 5.5.1, dans un contexte plus général que le seul type des listes. Par ailleurs, la règle de simplification `foldr/build` qui élimine des structures intermédiaires, assez simple à comprendre, sera rappelée à titre de comparaison dans l'exposé des différentes méthodes de déforestations calculationnelles qui est fait jusqu'à la fin de ce chapitre et dont les mécanismes sous-jacents ne sont pas toujours aussi faciles à appréhender.

5.3 Normalisation des *folds*

Le but de cette section est de présenter l'opérateur de contrôle générique *fold*, tel qu'il a été introduit par Sheard et Fegaras [SF93]. L'opérateur *fold* peut être vu comme une généralisation de l'opérateur `foldr` présenté à la section 5.2.4, au sens où il n'est pas exclusivement réservé au type des listes — d'où le terme d'opérateur de contrôle *générique*. Cet opérateur permet d'exprimer des fonctions dans un formalisme qui facilite la déforestation de leur composition. Cette déforestation s'appuie sur le *théorème de promotion* et elle est effectuée par un algorithme dit de *normalisation*. Dans ce chapitre, je présente ces notions et leur mode de fonctionnement en commençant par présenter les types considérés dans cette approche et la façon d'extraire leurs opérateurs *folds* associés. Encore une fois, la présentation que je fais ici a comme objectif de donner les idées de base de cette technique ; aussi, pour de plus amples détails, le lecteur pourra se référer à [MFP91, SF93, FSZ94].

Par ailleurs, certaines notions algébriques comme les *foncteurs* ou les *catamorphismes* seront utilisées dans cette section sans rentrer dans leurs explications. Ces notions seront plus clairement détaillées et formalisées dans la section 5.4.

5.3.1 Types simples

Dans cette présentation, les définitions de types considérés sont des simples *sommes-de-produits* définis par des équations récursives. De manière assez classique, le terme de *somme-de-produits* vient de l'approche algébrique des types : le choix entre les différentes alternatives de constructeurs d'un type est considéré comme étant une somme et, pour chaque constructeur, le m-uplet de ses arguments est considéré comme étant un produit. La définition d'un type T, simple *somme-de-produits*, est donc de la forme :

$$\begin{array}{rcll} T\ \alpha_1 \ldots \alpha_p & = & & c_1\ t_{1,1} \ldots t_{1,m_1} \\ & & \mid & \ldots \\ & & \mid & c_n\ t_{n,1} \ldots t_{n,m_n} \end{array}$$

où les $\alpha_1, \ldots, \alpha_p$ représentent des variables de types, les c_i sont des noms de constructeurs et les $t_{i,j}$ sont soit des variables de type (dans l'ensemble $\alpha_1, \ldots, \alpha_p$), soit des instanciations de types *sommes-de-produits*.

Voici quelques exemples de définitions de types *sommes-de-produits* :

$$\begin{array}{rcl} \mathit{list}\ \alpha & = & \mathit{nil} \mid \mathit{cons}\ \alpha\ (\mathit{list}\ \alpha) \\ \mathit{nat} & = & \mathit{zero} \mid \mathit{succ}\ \mathit{nat} \\ \mathit{tree}\ \alpha & = & \mathit{leaf}\ \alpha \mid \mathit{node}\ (\mathit{tree}\ \alpha)\ (\mathit{tree}\ \alpha) \end{array}$$

5.3.2 Foncteurs

À partir d'une telle définition de type, il est possible d'extraire un *foncteur* E associé qui est une abstraction de la structure récursive du type [SF93], au sens où il affranchit une structure de données du nom de ses constructeurs.

E vérifie comme tous les foncteurs la propriété suivante qui sera très utile :

$$(E\ f_1 \ldots f_n) \circ (E\ h_1 \ldots h_n) = E\ (f_1 \circ h_1) \ldots (f_n \circ h_n)$$

À un type simple somme-de-produits donné, Sheard et Fegaras associent donc un foncteur, dont la définition constructive[8] est donnée à la définition 5.3.1.

Définition 5.3.1 (Foncteur E) *Soit $T\ \alpha_1 \ldots \alpha_p$ un type. Le **foncteur** E^T associé à ce type est défini pour chaque constructeur $c_i : t_{i,1} \ldots t_{i,m_i} \rightarrow T\ \alpha_1 \ldots \alpha_p$ par le foncteur (p+1)-adique E_i^T suivant :*

$$E_i^T\ f_1 \ldots f_p\ f_T = \lambda x_{i,1}, \ldots, x_{i,m_i}.(K[t_{i,1}]x_{i,1}) \ldots (K[t_{i,m_i}]x_{i,m_i})$$

où les variables liées $x_{i,j}$ sont de type $t_{i,j}$ et $K[t_{i,j}]$ représente une fonction qui peut être obtenue par les règles suivantes :

$$\begin{array}{rcl} K[\alpha_k] & = & f_k \\ K[T\ \alpha_1 \ldots \alpha_p] & = & f_T \end{array}$$

□

Pour l'exemple du type *list*, les foncteurs associés aux constructeurs *nil* et *cons* sont :

$$\begin{array}{rcl} E_{nil}^{list}\ f\ g & = & \lambda().() \\ E_{cons}^{list}\ f\ g & = & \lambda x, y.(f\ x)\ (g\ y) \end{array}$$

Pour le type *tree*, les foncteurs associés aux deux constructeurs *leaf* et *node* sont :

$$\begin{array}{rcl} E_{leaf}^{tree}\ f\ g & = & \lambda x.f\ x \\ E_{node}^{tree}\ f\ g & = & \lambda x, y.(g\ x)\ (g\ y) \end{array}$$

Un foncteur peut donc être vu comme une description abstraite d'une structure.

5.3.3 Les *folds* et ce qu'ils représentent

L'introduction d'un tel foncteur et sa définition pour chacun des constructeurs d'un type permet de définir un opérateur de contrôle générique, couramment connu sous le nom de *fold*. Ce dernier est défini pour chaque type simple *somme-de-produits*. Tout comme le foncteur E défini précédemment décrit une abstraction de la structure récursive d'un type, le *fold* basé sur cette abstraction est un opérateur qui décrit comment appliquer une fonction ayant la même structure récursive sur un tel type de donnée.

Définition 5.3.2 (*Fold*) *Soit $T\ \alpha_1 \ldots \alpha_p$ un type. La fonction $fold^T$ associée à ce type est définie par l'ensemble des équations récursives suivantes, une pour chaque constructeur c_i :*

$$(fold^T\ \overline{f}) \circ c_i = f_i \circ (E_i^T\ id_1 \ldots id_p\ (fold^T\ \overline{f}))$$

où $\overline{f} = f_1 \ldots f_n$ et où id_j est la fonction identité.

□

8. Cette version est allégée par rapport à celle de [SF93], mais elle est suffisante pour la présentation qui est faite ici.

Chaque f_i est appelée une *fonction accumulative*. L'exemple ci-dessous présente l'opérateur *fold* pour le type *list* qui est donc défini par deux fonctions, une pour chaque constructeur, et grâce au foncteur E^{list} défini dans la section 5.3.2.

$$\begin{array}{lcl} (fold^{list}\ f_n\ f_c) \circ nil & = & f_n \circ (E^{list}_{nil}\ id_1\ (fold^{list}\ f_n\ f_c)) \\ & = & f_n \circ (\lambda().()) \end{array}$$

et

$$\begin{array}{lcl} (fold^{list}\ f_n\ f_c) \circ cons & = & f_c \circ (E^{list}_{cons}\ id_1\ (fold^{list}\ f_n\ f_c)) \\ & = & f_c \circ (\lambda x, y.(id_1\ x)\ ((fold^{list}\ f_n\ f_c)\ y)) \end{array}$$

d'où

$$\begin{array}{lcl} (fold^{list}\ f_n\ f_c)\ nil & = & f_n \\ (fold^{list}\ f_n\ f_c)\ cons\ head\ tail & = & f_c\ head\ ((fold^{list}\ f_n\ f_c)\ tail) \end{array}$$

Intuitivement, la fonction accumulative f_n est appliquée lorsque le $fold^{list}$ agit sur un *nil*. Ce dernier n'ayant pas d'argument, f_n n'en a pas non plus. Cela correspond à la *terminaison* de la récursion de la structure du type *list*. Par contre, si le $fold^{list}$ est appliqué à un constructeur *cons* (c'est le cas de *récursion* de la structure *list*), la fonction accumulative f_c prend en paramètre deux arguments : le premier est l'élément courant en tête de la liste (*head* est du type α, paramètre du type *list*) et le second correspond à l'appel récursif du $fold^{list}$ sur le reste de la liste (*tail* est du type *list* ; il correspond à la partie récursive de la définition du type *list*).

Par exemple, l'application de $(fold^{list}\ f_n\ f_c)$ sur une liste l à deux éléments donne :

$$(fold^{list}\ f_n\ f_c)\ (cons\ a\ (cons\ b\ nil)) = f_c\ a\ (f_c\ b\ f_n)$$

Afin de mieux comprendre l'utilisation et le comportement du $fold^{list}$, je donne ci-dessous des exemples simples de définitions de fonctions classiques sur les listes, en utilisant cet opérateur. La fonction *copy* crée une copie de la liste qui lui est passée en paramètre. La fonction $(append\ x\ y)$ concatène les listes x et y. La fonction $(length\ x)$ calcule la longueur de la liste x en utilisant la définition des entiers naturels (*nat*) présentée dans la section 5.3.1.

$$\begin{array}{lcl} copy\ x & = & (fold^{list}\ (\lambda().nil)\ (\lambda h, r.cons\ h\ r))\ x \\ append\ x\ y & = & (fold^{list}\ (\lambda().y)\ (\lambda h, r.cons\ h\ r))\ x \\ length\ x & = & (fold^{list}\ (\lambda().zero)\ (\lambda h, r.succ\ r))\ x \end{array}$$

La fonction *length*, définie par un *fold*, peut donc être appliquée sur une liste à deux éléments $l = cons\ a\ (cons\ b\ nil)$ de la façon suivante :

$$\begin{array}{ll} & length\ l \\ = & length\ (cons\ a\ (cons\ b\ nil)) \\ = & (fold^{list}\ (\lambda().zero)\ (\lambda h, r.succ\ r))\ (cons\ a\ (cons\ b\ nil)) \\ = & (\lambda h, r.succ\ r)\ a\ ((\lambda h, r.succ\ r)\ b\ (\lambda().zero)) \\ = & (\lambda h, r.succ\ r)\ a\ ((\lambda h, r.succ\ r)\ b\ zero) \\ = & (\lambda h, r.succ\ r)\ a\ (succ\ zero) \\ = & succ\ (succ\ zero) \end{array}$$

Une manière intuitive de se représenter ce qu'est une fonction définie par un *fold* est de se souvenir que cet opérateur abstrait en même temps le schéma de récursion du type des données et le schéma de récursion de la fonction. On obtient le résultat de la fonction en

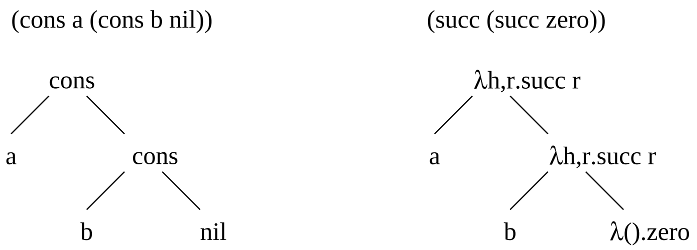

FIG. 5.11: *L'application d'un fold vu comme remplacement des constructeurs par les fonctions accumulatives : exemple pour length*

remplaçant, dans le terme qui lui est passé en argument, tous les constructeurs de type par les fonctions accumulatives correspondantes du *fold*. Ainsi, l'exécution déroulée ci-dessus peut se visualiser comme dans la figure 5.11, en remplaçant les *cons* par des $(\lambda h, r.succ\ r)$ et les *nil* par des $(\lambda().zero)$.

Je termine cette présentation en donnant, pour diversifier les exemples, la définition de l'opérateur *fold* pour le type *tree*, utilisant le foncteur E^{tree} défini dans la section 5.3.2. Il accepte également deux fonctions accumulatives, f_l et f_n correspondant aux constructeurs *leaf* et *node*.

$$
\begin{array}{lll}
(fold^{tree}\ f_l\ f_n) \circ leaf & = & f_l \circ (E^{tree}_{leaf}\ id_1\ (fold^{tree}\ f_l\ f_n)) \\
 & = & f_l \circ (\lambda x.id_1\ x)
\end{array}
$$

et

$$
\begin{array}{lll}
(fold^{tree}\ f_l\ f_n) \circ node & = & f_n \circ (E^{tree}_{node}\ id_1\ (fold^{tree}\ f_l\ f_n)) \\
 & = & f_n \circ (\lambda x, y.((fold^{tree}\ f_l\ f_n)\ x)\ ((fold^{tree}\ f_l\ f_n)\ y))
\end{array}
$$

d'où

$$
\begin{array}{lll}
(fold^{tree}\ f_l\ f_n)\ leaf\ n & = & f_l\ n \\
(fold^{tree}\ f_l\ f_n)\ node\ left\ right & = & f_n\ ((fold^{tree}\ f_l\ f_n)\ left)\ ((fold^{tree}\ f_l\ f_n)\ right)
\end{array}
$$

Le formalisme des *folds* permet donc d'abstraire le schéma de récursion des fonctions sur le schéma de récursion du type de donnée sous-jacent et de spécifier les calculs à effectuer sur ce schéma de récursion abstrait plutôt que sur les constructeurs de la structure.

5.3.4 L'algorithme de normalisation

Il est donc désormais possible de définir et d'évaluer des fonctions récursives avec des *folds*. Il suffit pour cela d'avoir une définition algébrique du type de donnée sur lequel une fonction agit, puis de décomposer la définition de la fonction sous la forme de fonctions accumulatives, une pour chaque constructeur du type.

Je reviens donc au problème initial : la déforestation. Exprimer des fonctions avec des *folds* n'empêche pas la production de structures de données intermédiaires lors de la composition de fonctions. Cependant, les définitions de fonctions à l'aide de l'opérateur *fold* sont dirigées par la structure du schéma de récursion du type ; cette propriété permet d'appliquer plus facilement un algorithme de déforestation. L'algorithme de normalisation mis au point par Sheard et Fegaras [SF93] est basé sur une loi générale qui s'applique à tous les *folds*, appelée le *théorème de promotion* ou encore *fixed point fusion* dans [MFP91]. Ce théorème établit que la composition de n'importe quelle fonction g avec un *fold* est encore un *fold*, dont les fonctions accumulatives ne dépendent que de celles du *fold* initial et de g.

Théorème 5.3.1 (Théorème de promotion du *fold*)

$$\frac{\forall i : \phi_i \circ (E_i^T \ id_1 \ldots id_p \ g) = g \circ f_i}{g \circ (fold^T \ \overline{f}) = (fold^T \ \overline{\phi})}$$

□

Je donne ci-dessous l'expression de ce théorème pour le type *list*, basée sur les définitions des foncteurs E_{nil}^{list} et E_{cons}^{list} (section 5.3.2), et sur la propriété que, pour tout foncteur E, $(E \ f_1 \ldots f_n) \circ (E \ h_1 \ldots h_n) = E \ (f_1 \circ h_1) \ldots (f_n \circ h_n)$.

$$\frac{\begin{array}{lcl} \phi_n & = & g \ f_n \\ \phi_c \ h \ (g \ r) & = & g \ (f_c \ h \ r) \end{array}}{g \ ((fold^{list} \ f_n \ f_c) \ x) = (fold^{list} \ \phi_n \ \phi_c) \ x}$$

De même, le théorème de promotion du *fold* sur le type *tree* s'exprime à partir des définitions des foncteurs E_{leaf}^{tree} et E_{node}^{tree} de la façon suivante :

$$\frac{\begin{array}{lcl} \phi_l \ n & = & g \ (f_l \ n) \\ \phi_n \ (g \ r_l) \ (g \ r_r) & = & g \ (f_n \ r_l \ r_r) \end{array}}{g \ ((fold^{tree} \ f_l \ f_n) \ t) = (fold^{tree} \ \phi_l \ \phi_n) \ t}$$

Comme le schéma de récursion du *fold* est entièrement décrit par le foncteur du type associé et que ses fonctions accumulatives sont bien séparées, il est possible de savoir où la récursion va s'appliquer : reconnaître une nouvelle fonction accumulative dans l'application de g sur chaque schéma de récursion suffit pour définir un nouveau *fold*.

En utilisant ce théorème, l'algorithme de normalisation [SF93] permet de déforester la composition d'une fonction g avec une fonction définie par un *fold* en intégrant g dans les fonctions accumulatives du *fold*. Dans le cas où g est elle même exprimée avec un *fold*, le but de l'algorithme de normalisation est de réduire des *folds* appliqués à des *folds* en des *folds* appliqués à des variables (éliminer des structures intermédiaires). Intuitivement, cette normalisation est accomplie en *poussant* le *fold* externe dans la fonction d'accumulation du *fold* interne.

Dans le langage des termes qui est utilisé ici, un programme a la forme $\lambda x_1, \ldots, x_n.t$, où chaque x_i est une variable et le terme t est soit :

- *une variable* : x, liée dans une lambda abstraction externe ;
- *une construction* : $c \ t_1 \ldots t_n$ où c est un constructeur et chaque t_i un terme ;
- *un fold* : $(fold^T \ f_1 \ldots f_n) \ t$ sur un terme t de type T. Chaque f_i a la forme $\lambda x_1, \ldots, x_m.t_i$ où chaque x_j est une variable liée et t_i est un terme.

Les variables liées x_j des fonctions accumulatives sont dites *variables d'accumulation de résultat* si et seulement si elles sont associées à un paramètre de type T dans le constructeur correspondant ; en fait, elles représentent le résultat d'un appel récursif du *fold*.

Par exemple, dans la définition de *append*, **r** est une variable d'accumulation de résultat :

$$append\ x\ y = (fold^{list}\ (\lambda().y)\ (\lambda h, \mathbf{r}.cons\ h\ \mathbf{r}))\ x$$

Intuitivement, ceci veut dire que dans l'application du *fold* sur un terme $x = cons\ head\ tail$, le h représente *head* et le $\mathbf{r}$ représente le résultat de l'appel de $append\ tail\ y$. Cette notion est utile dans la définition suivante.

Définition 5.3.3 (Terme potentiellement normalisable) *Un programme dans le langage des termes est dit* ***potentiellement normalisable*** *s'il ne contient pas de terme* $(fold^T\ \overline{f})\ t$ *tel que t fasse référence à une variable d'accumulation de résultat qui soit liée dans une lambda abstraction englobante (externe).*
□

Un contre-exemple assez simple est la fonction d'inversion de liste, *naif_rev*, exprimée de la façon naïve[9] suivante :

```
let naif_rev x =
  case x of
    cons head tail → append (naif_rev tail (cons head nil))
    nil → nil
```

Cette fonction s'exprime par un *fold* de la façon suivante :

$$(fold^{list}\ (\lambda().nil)\ (\lambda h, r.append\ r\ (cons\ h\ nil)))\ x$$

soit, d'après la définition de *append*

$$(fold^{list}\ (\lambda().nil)\ (\lambda h, \mathbf{r}.(fold^{list}\ (\lambda().cons\ h\ nil)\ (\lambda i, s.cons\ i\ s))\ \mathbf{r}))\ x$$

où le *fold* interne porte sur $\mathbf{r}$, qui est une variable d'accumulation de résultat. Cette fonction *naif_rev* exprimée ainsi n'est donc pas potentiellement normalisable. L'idée intuitive est que le *fold* interne accepte en paramètre une variable d'accumulation de résultat, "en cours de construction" dans le *fold* externe. La promotion sur une fonction utilisant une telle variable n'est donc pas envisageable.

Un terme est dit *canonique* s'il ne contient que des *folds* sur des variables et qu'aucune de ces variables n'est une variable d'accumulation de résultat ; un terme canonique sera donc toujours potentiellement normalisable.

Algorithme de normalisation

Pour une définition de fonction donnée sous la forme d'un *fold*, l'algorithme de normalisation permet de déforester la composition d'une fonction g avec ce *fold*. Ceci est fait en intégrant g dans les fonctions accumulatives du *fold*, évitant ainsi des constructions de structures intermédiaires.

Dans l'algorithme de normalisation présenté ci-dessous, ρ est une fonction partielle des termes vers les variables. $\rho[g/r]$ étend la fonction partielle ρ en associant le terme g à la variable r et $\rho[(g_1, \ldots, g_n)/(r_1, \ldots, r_n)]$ étend ρ en associant les g_i aux r_i. L'algorithme de normalisation comprend les trois parties suivantes :

9. Naïve au sens où même si elle n'est pas très naturelle, cette forme est syntaxiquement la plus simple, la plus basique.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{length = fold^{list}\ (\lambda().zero)\ (\lambda h, r.succ\ r)} \\
length\ (cons\ head\ tail) & = & (fold^{list}\ f_n\ f_c)\ (cons\ head\ tail) \\
 & = & f_c\ head\ ((fold^{list}\ f_n\ f_c)\ tail) \\
 & = & (\lambda h, r.succ\ r)\ head\ ((fold^{list}\ f_n\ f_c)\ tail) \\
 & = & succ\ ((fold^{list}\ f_n\ f_c)\ tail) \\
length\ (cons\ head\ tail) & = & succ\ (length\ tail)
\end{array}
$$

FIG. 5.12: *Un pas d'application à une construction pour length*

$$
\begin{array}{lcl}
\multicolumn{3}{l}{g((fold^{list}\ f_n\ f_c)\ x) = (fold^{list}\ \phi_n\ \phi_c)\ x} \\
\multicolumn{3}{l}{\text{où}} \\
\phi_n & = & g\ f_n \\
\phi_c\ r_1\ r_2 & = & g\ (f_c\ x_1\ x_2) \\
\text{et comme} & & (E^{list}_{cons}\ id_1\ g)\ x_1\ x_2 = (id_1\ x_1)\ (g\ x_2) = (x_1)\ (g\ x_2) \\
\text{on a} & & \rho[x_1/r_1, (g\ x_2)/r_2]
\end{array}
$$

FIG. 5.13: *Promotion du fold pour le type list*

- **Généralisation** : si l'algorithme de normalisation produit un terme associé dans ρ à une variable v, alors ce terme est remplacé par v :

$$t \text{ est remplacé par } v \text{ si } (t/v) \in \rho$$

- **Application à une construction** : d'après la définition du *fold* :

$$(fold^T\ \overline{f})\ (c_i\ \overline{u}) \qquad \text{est remplacé par} \qquad f_i((E_i^T\ id_1 \ldots id_p\ (fold^T\ \overline{f}))\ \overline{u})$$

  voir la figure 5.12 pour l'exemple de *length*.

- **Promotion du Fold** : si le terme est une composition d'une fonction g avec un *fold*,

$$(g\ ((fold^T\ f_1 \ldots f_n)\ x)) \qquad \text{est remplacé par} \qquad (fold^T\ \phi_1 \ldots \phi_n)\ x$$

  Le théorème de promotion du *fold* est appliqué pour déterminer chaque ϕ_i en appliquant récursivement l'équation :

$$(\phi_i\ r_1 \ldots r_{m_i}) = (g\ (f_i\ x_1 \ldots x_{m_i}))$$

  où chaque x_i et r_i sont de nouvelles variables. Dans chacun des cas, ρ est étendu avec le filtrage des termes $(E_i^T\ id_1 \ldots \ldots\ g)\ x_1 \ldots x_{m_i}$ sur les variables $r_1 \ldots r_{m_i}$. Voir la figure 5.13 pour le type *list*.

Pour illustrer cet algorithme, je déroule ci-dessous un exemple de normalisation de la composition de deux fonctions dans le terme *length* (*append* x y) :

$$
\begin{array}{lcl}
length\ x & = & (fold^{list}\ (\lambda().zero)\ (\lambda h, r.succ\ r))\ x \\
append\ x\ y & = & (fold^{list}\ f_n\ f_c)\ x \\
 & & \text{avec} \begin{cases} f_n = \lambda().y \\ f_c = \lambda h, r.cons\ h\ r \end{cases}
\end{array}
$$

Il faut alors trouver un $(\mathit{fold}^{list}\ \phi_n\ \phi_c)\ x$ équivalent à la composition

$$\mathit{length}\ (\mathit{append}\ x\ y)$$

L'algorithme de normalisation pour $g = \mathit{length}$ donne :

$$\begin{array}{lcl}
\phi_n & = & \mathit{length}\ f_n\ \text{, avec } \rho = [\,]\quad \text{obtenu par Promotion} \\
& = & \mathit{length}\ y \\
& = & (\mathit{fold}^{list}\ (\lambda().\mathit{zero})\ (\lambda h, r.\mathit{succ}\ r))\ y \\
\phi_c\ r_1\ r_2 & = & \mathit{length}\ (f_c\ x_1\ x_2)\ \text{, avec } \rho = [x_1/r_1, (\mathit{length}\ x_2)/r_2] \\
& & \quad \text{obtenu par Promotion} \\
& = & \mathit{length}\ (\mathit{cons}\ x_1\ x_2) \\
& = & \mathit{succ}\ (\mathit{length}\ x_2)\quad \text{par Application à une construction} \\
& = & \mathit{succ}\ r_2\quad \text{par Généralisation d'après } \rho
\end{array}$$

On obtient donc pour la composition de $\mathit{length}\ (\mathit{append}\ x\ y)$:

$$\begin{array}{ll}
(\mathit{fold}^{list} & (\lambda().(\mathit{fold}^{list}\ (\lambda().\mathit{zero})\ (\lambda h, r.\mathit{succ}\ r))\ y), \\
& (\lambda r_1, r_2.\mathit{succ}\ r_2))\ x
\end{array}$$

La liste intermédiaire n'est alors plus construite et la composition est donc déforestée.

5.3.5 Comparaison avec l'algorithme de Wadler

À ce stade de la présentation, il apparaît déjà des similitudes entre l'algorithme de normalisation des *folds* et l'algorithme de déforestation de Wadler présenté à la section 5.2.

En effet, le pas d'application à une construction de l'algorithme de normalisation, qui permet d'*exposer* les constructeurs d'une structure à l'effet d'une fonction exprimée par un *fold*, produit le même effet que la règle (5) de l'algorithme de déforestation de Wadler (cf. figure 5.8) qui spécialise l'application d'un `case` en fonction du *pattern* sur lequel il intervient. Dans ce cas, la définition du *fold* à l'aide du foncteur, qui explicite la récursion de la fonction et qui la décompose relativement à la structure récursive du type considéré, facilite l'association des fonctions accumulatives de résultat aux constructeurs correspondants. Dans l'algorithme de Wadler, cette phase est réalisée par le *pattern matching* du `case`, mais l'effet obtenu est le même : l'exposition directe d'un constructeur à une fonction permet, par dépliage[10], de faire *ressortir* les constructeurs vers l'extérieur (vers la gauche) des termes. Comme les structures intermédiaires sont construites par des constructeurs auxquels sont appliquées des fonctions, faire ressortir les constructeurs équivaut à éliminer des constructions de structures intermédiaires.

Par ailleurs, la promotion du *fold* de l'algorithme de normalisation peut être vue, en partie, comme l'action conjuguée des règles (6) et (7) de l'algorithme de déforestation de Wadler (cf. figure 5.8), qui ont pour but de déplier un appel de fonction et, dans le cas où cette fonction est un terme `case`, de distribuer le `case` externe dans les branches du `case` interne. Si un *fold* peut être vu comme une sorte de `case`, qui distribue les fonctions accumulatives de résultats sur leurs constructeurs correspondants, la promotion du *fold* a bien pour effet de distribuer le calcul de g — que g soit un fold (règle (7)) ou non (règle (6)) — dans les fonctions accumulatives de résultat du *fold* interne, créant ainsi les fonctions ϕ_i.

10. Au sens de Burstall et Darlington, c'est-à-dire remplacement de l'appel de la fonction par sa définition.

Pour terminer, la phase de généralisation de l'algorithme de normalisation permet de résoudre le problème de la terminaison de l'algorithme de Wadler. En effet, l'association dans ρ entre les variables r_i introduites par l'algorithme lors de la promotion du *fold* et les termes x_i ou $(g\ x_i)$ qui leur correspondent n'a pour but que de pouvoir réintroduire les fonctions récursives et d'assurer la terminaison de l'algorithme. C'est le cas dans l'exemple déroulé à la section précédente (section 5.3.4), lorsque le terme $succ\ (length\ x_2)$ est remplacé par r_2 dans un pas de généralisation : cela permet de *définir* la fonction ϕ_c relativement à son paramètre r_2. C'est bien ce type de remplacement qui permet à l'algorithme de terminer en reconnaissant des sous-expressions comme ayant déjà été rencontrées.

Malgré les différences apparentes et un traitement de la terminaison directement inclus dans l'algorithme de normalisation des *folds* tandis qu'il est plutôt extérieur dans l'algorithme de Wadler, ces deux approches sont basées sur des principes similaires mais avec des techniques de mise en œuvre différentes. Quant aux termes pris en compte par les algorithmes respectifs, il ne manque à un terme canonique que la linéarité pour être un terme *treeless*. Cette condition de linéarité est introduite dans l'algorithme de Wadler pour éviter la duplication de termes coûteux ; ces considérations ne sont pas prises en compte dans l'algorithme de normalisation.

5.3.6 Limitations et extensions

La normalisation des *folds*, telle qu'elle a été présentée jusqu'ici, ne permet pas de normaliser n'importe quel programme. C'est le cas de la définition naïve de l'inversion de liste déjà présentée plus haut :

$$(fold^{list}\ (\lambda().nil)\ (\lambda h, r.append\ r\ (cons\ h\ nil)))\ x$$

où *append* est un *fold* qui traverse la variable d'accumulation de résultat r. Comme le précisent les auteurs dans [SF93], de nombreuses fonctions peuvent être exprimées de la sorte, en utilisant un paramètre supplémentaire qui joue le rôle d'un accumulateur. De telles fonctions peuvent fréquemment être exprimées sous la forme de *fold du second ordre*, que je vais introduire ci-dessous, et certaines d'entre elles peuvent alors être normalisables tandis qu'elles ne l'étaient pas sous leur forme en *fold* du premier ordre.

Ainsi, si l'on considère la définition de la fonction qui inverse une liste sous la forme suivante à deux paramètres, plus répandue dans le monde fonctionnel que la version naïve *naif_rev* précédemment présentée, l'appel pour inverser la liste x sera alors fait par $rev\ x\ nil$:

```
let rev x w =
  case x of
    cons head tail → rev tail (cons head w)
    nil → w
```

Le *fold* de second ordre correspondant à cette définition de fonction est le suivant :

$$rev\ x = (fold^{list}\ (\lambda().(\lambda w.w))\ (\lambda h, r.(\lambda w.r\ (cons\ h\ w))))\ x\ nil$$

Sheard et Fegaras [SF93] ont introduit une extension de l'algorithme de normalisation pour les *folds* de second ordre qui permet d'amener cette forme à normalisation. Je ne présente pas ici cette méthode dans le détail, mais j'en donne les grandes lignes. Le lecteur pourra bien sûr se reporter à [SF93] pour plus de détails.

Comme je l'ai dit, la version naïve de *reverse* en *fold* n'est pas directement normalisable, car la fonction *append* qu'elle contient agit sur la variable d'accumulation de résultat r qui représente le résultat du *fold* et qui est en cours de construction. Plus précisément, cette forme naïve n'est pas normalisable parce que le *fold* intérieur porte sur r, une variable du type *list* liée par le *fold* externe.

Pour résoudre ce problème, en plus du théorème de promotion du second ordre, une transformation[11] $\mathcal{F}_{\mathcal{G}}$ est introduite dans [SF93]. Pour l'exemple de *reverse*, cette transformation traduit automatiquement le programme *fold* naïf $naif\_rev$ en son équivalent en second ordre, qui est normalisable via le théorème de promotion de second ordre.

Il faut cependant noter que la transformation $\mathcal{F}_{\mathcal{G}}$ impose des restrictions sur le type de base et sur les fonctions accumulatives, ce qui fait perdre beaucoup de sa généralité à la méthode. Sans rentrer dans les détails (voir [SF93]), le type doit posséder un *zéro-constructeur* et les fonctions accumulatives doivent utiliser des *fonctions de zéro-remplacement*. Grâce à ces conditions particulières, la forme naïve en *fold* de *reverse* peut être transformée par $\mathcal{F}_{\mathcal{G}}$ et le *fold* de second ordre résultant est normalisable avec le théorème de promotion de second ordre. Cela autorise des déforestations telles que *length (reverse x) = length x* ou *reverse (reverse x) = copy x.*[12]

Cependant, on ne sait toujours pas comment opérer une telle transformation dans le cas où le type de base ne possède pas naturellement de zéro-constructeur, comme le type *tree* précédemment défini.

5.4 Formalismes algébriques

Cette section traite des travaux sur l'*algorithmique constructive*[13] qui ont abouti à des formalismes, des méthodes et des systèmes de transformation de programmes à des fins de déforestation. Le but de l'algorithmique constructive est de *calculer* des programmes à partir de leur spécification. Le sens du mot *calculer* est moins clair pour des fonctions que pour des opérations arithmétiques comme l'addition ou la multiplication. En effet, comme le soulignent Meijer, Fokkinga et Paterson dans [MFP91], il est plus difficile d'établir, de prouver et d'utiliser des lois pour la composition de fonctions arbitraires définies récursivement que pour la composition d'opérations bien connues de l'arithmétique. Le principal problème avec les fonctions récursives est de traiter leur schéma de récursion de façon isolée ; la structure algorithmique de telles fonctions est obscurcie par des définitions récursives non structurées. Dans les *folds*, les foncteurs sont utilisés pour représenter les schémas de récursion des structures et pour guider la définition des fonctions. L'approche de l'algorithmique constructive est d'augmenter encore cette abstraction et d'utiliser les foncteurs pour définir les types eux-mêmes. Les schémas de récursion de ces types sont alors vus comme des fonctions d'ordre supérieur, distincts les uns des autres et indépendants des ingrédients qui leur permettent de spécifier des définitions de fonctions récursives.

Cette approche est à la base de différentes recherches qui ont tenté de définir des opérateurs de récursion qui sont naturellement associés à des définitions de types algébriques. Par ailleurs, de nombreuses lois ont été établies concernant ces opérateurs et leur composition. Les principaux opérateurs intéressants dans le cadre de mon étude travaillent sur des types

11. Cette transformation est basée sur les notions de passage de continuation (CPS).
12. En utilisant, en plus du théorème de promotion de second ordre, la transformation $\mathcal{F}_{\mathcal{G}}^{-1}$ définie dans [SF93].
13. Le terme anglo-saxon est *Constructive Algorithmics.*

de données définis récursivement, tels que le type liste :

$$list\ A = nil \mid cons\ A\ (list\ A)$$

La structure récursive de cette définition est très fréquemment utilisée pour écrire des fonctions qui traitent de ce type. Ceci a permis de distinguer différentes familles de fonctions, relativement à leur comportement général par rapport à une telle définition récursive de type. Meijer *et al.* les décrivent assez intuitivement dans [MFP91]. Une fonction du type $(list\ A) \rightarrow B$ qui détruit une liste est appelée un *catamorphisme* (du grec $\kappa\alpha\tau\alpha$, qui signifie *en bas*). C'est l'équivalent d'un *fold*, une sorte de *pliage* du paramètre, évoqué dans [BD77, BW88]. À l'opposé, une fonction du type $B \rightarrow (list\ A)$, qui construit une liste à partir d'un élément, est appelée un *anamorphisme* (du grec $\alpha\nu\alpha$, qui signifie *vers le haut*) ; c'est en fait l'opération duale de *dépliage* du paramètre, *unfold*, quelquefois appelée constructeur (*build*), comme dans [GLJ93]. Par ailleurs, une fonction de type $A \rightarrow B$ dont le schéma récursif d'appel a la forme d'une liste, ou plus généralement d'un type de donnée défini récursivement, est appelée un *hylomorphisme* (du grec $\upsilon\lambda o$ qui signifie *matière*, et de la philosophie d'Aristote considérant que la forme et la matière ne font qu'un).

Après ce détour culturel, je vais présenter assez brièvement dans cette section les bases de l'algorithmique constructive ayant trait à la déforestation. Pour de plus amples détails, je conseille au lecteur de se reporter à [BW88, Mal90, MFP91, Fok95, Mee95, TM95, HIT96b].

5.4.1 Quelques rappels sur les catégories

Bien que ce ne soit pas le sujet de cette thèse, ni fondamental pour la compréhension de ce qui suit, je présente ici les objets et le formalisme de base de la théorie des catégories, auxquels je ferai référence plus loin, dans la section 5.5.

Au sens mathématique du terme, un *morphisme* est synonyme d'homomorphisme. Si on considère un ensemble E muni d'une opération $\top$ et un ensemble E' muni d'une opération $\perp$, un morphisme est une application f de E dans E' telle que $f(x \top y) = f(x) \perp f(y)$ où x et y appartiennent à E.

Définition 5.4.1 (Catégorie) *Une **catégorie** $\mathcal{C}$ est définie par la donnée de :*

- *une collection d'*objets*, A, B, C ...*
- *une collection de* morphismes*, f, g, h ...*
- *une relation entre les morphismes et les paires d'objets, appelée* typage *et notée :* $f : A \rightarrow_{\mathcal{C}} B$ *où f est le morphisme et où $A \rightarrow_{\mathcal{C}} B$ est le* type *de f ;*
- *une opération binaire sur les morphismes, appelée* composition *et notée $g \circ f$ ou gf ;*
- *pour chaque objet A, un morphisme particulier, appelé* identité *sur A et noté id ou id_A ;*

qui vérifient les axiomes suivants :

- *Unicité du type*
  $f : A \rightarrow_{\mathcal{C}} B$ et $f : A' \rightarrow_{\mathcal{C}} B' \quad \Rightarrow \quad A = A'$ et $B = B'$
- *Typage de la composition*
  $f : A \rightarrow_{\mathcal{C}} B$ et $g : B \rightarrow_{\mathcal{C}} C \quad \Rightarrow \quad g \circ f : A \rightarrow_{\mathcal{C}} C$

- *Typage de l'identité*
  $id_A \; : A \rightarrow_{\mathcal{C}} A$
- *Associativité de la composition*
  $h \circ (g \circ f) = (h \circ g) \circ f$
- *Identité, pour tout A*
  $f \circ id_A = f = id_A \circ f$

□

Parmi les catégories les plus utilisées, on peut citer la catégorie $\mathcal{S}et$ dont les objets sont les ensembles et dont les morphismes sont les fonctions totales typées ; la composition et les identités de $\mathcal{S}et$ sont celles des fonctions. La catégorie $\mathcal{CPO}$ est la catégorie des Ordres Partiels Complets dont les morphismes sont les fonctions continues. Elle sera très utile dans la suite de cette présentation.

Je présente ici quelques définitions relatives aux catégories, à commencer par les notions d'*objet initial* et d'*objet final* d'une catégorie.

Définition 5.4.2 (objet initial) *Un objet A d'une catégorie $\mathcal{C}$ est dit* ***initial*** *si pour tout objet B de $\mathcal{C}$, il existe un morphisme et un seul de A vers B :*

$$\forall\ B\ \exists !\ f \ \textit{tel que}\ f : A \rightarrow B$$

□

Définition 5.4.3 (objet final) *Un objet A d'une catégorie $\mathcal{C}$ est dit* ***final*** *si pour tout objet B de $\mathcal{C}$, il existe un morphisme et un seul de B vers A :*

$$\forall\ B\ \exists !\ f \ \textit{tel que}\ f : B \rightarrow A$$

□

D'une autre façon, l'objet A est dit final s'il est initial dans la catégorie duale de $\mathcal{C}$.

Dans la définition d'une catégorie, les objets sont juste des *choses* sur lesquelles aucune structure interne ne peut être observée par des moyens *catégoriels* (compositions, identités, morphismes, typages). Les *foncteurs* fournissent un outil pour manipuler les objets structurés [Fok95], en agissant sur les objets et les morphismes de sorte à ce que les propriétés de typage soient conservées, comme le spécifie la définition 5.4.4 et comme l'illustre le diagramme suivant.

Définition 5.4.4 (Foncteur) *Soient $\mathcal{C}$ et $\mathcal{D}$ deux catégories ; un* ***foncteur*** *F de $\mathcal{C}$ dans $\mathcal{D}$ associe des objets de $\mathcal{C}$ à des objets de $\mathcal{D}$ et des morphismes de $\mathcal{C}$ à des morphismes de $\mathcal{D}$ de sorte que :*

$$\begin{array}{rcll}
\mathsf{F}\,f & : & \mathsf{F}\,A \rightarrow_{\mathcal{D}} \mathsf{F}\,B & \textit{si}\ f : A \rightarrow_{\mathcal{C}} B \\
\mathsf{F}\,id_A & = & id_{\mathsf{F}A} & \textit{pour chaque objet } A \textit{ dans } \mathcal{C} \\
\mathsf{F}\,(g \circ f) & = & (\mathsf{F}\,g) \circ (\mathsf{F}\,f) & \textit{si}\ g \circ f \text{ est bien typé.}
\end{array}$$

□

Les diagrammes suivants permettent de visualiser les relations qui lient un foncteur aux morphismes et aux objets des catégories.

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "\mathsf{F}"'] & B \arrow[r, "g"] \arrow[d, "\mathsf{F}"'] & C \arrow[d, "\mathsf{F}"'] \\ \mathsf{F}A \arrow[r, "\mathsf{F}f"] & \mathsf{F}B \arrow[r, "\mathsf{F}g"] & \mathsf{F}C \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "(g\circ f)"] \arrow[d, "\mathsf{F}"'] & C \arrow[d, "\mathsf{F}"'] \\ \mathsf{F}A \arrow[r, "(\mathsf{F}g)\circ(\mathsf{F}f)", "\mathsf{F}(g\circ f)"'] & \mathsf{F}C \end{tikzcd}$$

De nombreuses lois et propriétés sont relatives aux catégories et à leur manipulation ; j'en rappelle maintenant quelques unes.

Les **endofoncteurs** sur une catégorie $\mathcal{C}$ sont des foncteurs de $\mathcal{C}$ dans $\mathcal{C}$. La donnée d'un tel endofoncteur F permet de définir la notion catégorielle de F-algèbre, représentant classiquement un monoïde, c'est-à-dire un ensemble (un objet de la catégorie) muni d'une opération associative (la composition) et d'un élément neutre pour cette opération (l'identité).

Définition 5.4.5 (F-algèbre) *Soit une catégorie $\mathcal{C}$ et un endofoncteur F dans $\mathcal{C}$. Une* ***F-algèbre*** *est une paire (A, ϕ) où A est un objet de $\mathcal{C}$, appelé le* ***support*** *de l'algèbre*[14] *et $\phi : \mathsf{F}A \to A$ est un morphisme appelé l'****opération*** *de l'algèbre.*

La F-algèbre peut aussi être dénotée par son opération, le support étant alors implicite : on parlera de la F-algèbre $\phi : \mathsf{F}A \to A$.

□

De même, il est possible de retranscrire la notion classique d'homomorphisme en termes catégoriels et relativement à un endofoncteur.

Définition 5.4.6 (F-homomorphisme) *Soit une catégorie $\mathcal{C}$ et un endofoncteur F sur $\mathcal{C}$. Étant donné deux F-algèbres (A, ϕ_A) et (B, ϕ_B), le* ***F-homomorphisme*** *de (A, ϕ_A) vers (B, ϕ_B) est un morphisme h de l'objet A vers l'objet B dans la catégorie $\mathcal{C}$, qui satisfait $h \circ \phi_A = \phi_B \circ \mathsf{F}h$.*

Dans la notation plus concise, le F-homomorphisme de ϕ_A dans ϕ_B est dénoté par $h : \phi_A \to_F \phi_B$.

□

La notion de F-homomorphisme peut être intuitivement illustrée par le diagramme suivant, où la propriété $h \circ \phi_A = \phi_B \circ \mathsf{F}h$ signifie que le diagramme commute :

$$\begin{tikzcd} A \arrow[r, "h"] & B \\ \mathsf{F}A \arrow[u, "\phi_A"] \arrow[r, "\mathsf{F}h"] & \mathsf{F}B \arrow[u, "\phi_B"'] \end{tikzcd}$$

14. Le terme anglo-saxon est *carrier of the algebra*.

À partir de la définition de ces objets, une nouvelle catégorie va permettre de se rapprocher de la notion de type de donnée.

Définition 5.4.7 (Catégorie des F-algèbres) *Soit une catégorie $\mathcal{C}$ et un endofoncteur F sur $\mathcal{C}$. La* ***catégorie $\mathcal{ALG}(\mathsf{F})$ des F-algèbres*** *a pour objets les F-algèbres de $\mathcal{C}$ et pour morphismes tous les F-homomorphismes entre les F-algèbres. La composition et l'identité dans la catégorie des F-algèbres sont ceux de la catégorie $\mathcal{C}$.*
□

La définition d'un objet initial d'une catégorie (Définition 5.4.2) est tout à fait applicable sur $\mathcal{ALG}(\mathsf{F})$ et cet objet, s'il en existe un, est alors lui-même une F-algèbre.

Définition 5.4.8 (F-algèbre initiale) *Soit une catégorie $\mathcal{C}$ et un endofoncteur F sur $\mathcal{C}$. Une* ***F-algèbre initiale*** *est un objet initial de la catégorie $\mathcal{ALG}(\mathsf{F})$.*
□

Si je suppose, en prenant un peu d'avance sur la section suivante, l'existence d'une F-algèbre initiale pour un foncteur donné F, et que je décide de la noter $in_F : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$, cette F-algèbre initiale vérifie d'après la définition 5.4.2 l'existence d'un unique F-homomorphisme h_B pour toute F-algèbre $\phi_B : \mathsf{F}B \to B$. Cette propriété est illustrée par le diagramme suivant :

$$\begin{tikzcd} \mu\mathsf{F} \arrow[r, "h_B"] & B \\ \mathsf{F}\mu\mathsf{F} \arrow[u, "in_F"] \arrow[r, "\mathsf{F}h_B"] & \mathsf{F}B \arrow[u, "\phi_B"'] \end{tikzcd}$$

5.4.2 Types de donnée comme points fixes de foncteurs

Le but de l'utilisation de ces notions des catégories est de pouvoir représenter, par un endofoncteur, à la fois la structure des données et la structure de contrôle dans la définition d'un type — c'est-à-dire à la fois le type récursif et la récursion naturelle sur ce type. Par la suite, cela permet de représenter plus facilement des définitions de fonctions sur ces types en utilisant ces foncteurs.

En général, dans le contexte des définitions de fonctions récursives sur des types algébriques, la catégorie $\mathcal{C}$ utilisée par défaut est $\mathcal{CPO}$, la catégorie des ordres partiels complets avec les fonctions continues, ce qui permet de prendre en compte des équations récursives arbitraires dans une structure proche des langages fonctionnels paresseux [MFP91, HIT96b]. Dans la suite, j'utiliserai donc $\mathcal{C}$ pour $\mathcal{CPO}$.

De plus, dans ce contexte, il a été prouvé [Mal90] que la catégorie $\mathcal{ALG}(\mathsf{F})$ possède un objet initial dès que l'endofoncteur F est *polynômial*, c'est-à-dire construit à partir des quatre foncteurs de base *identité*, *constant*, *produit* et *somme disjointe* définis ci-dessous. Les objets de la catégorie $\mathcal{C}$ manipulés dans la suite seront appelés des **types**.

Définition 5.4.9 (Identité) *Le foncteur* ***identité*** *Id est défini par son effet sur un type X et sur une fonction f comme suit :*

$$\begin{array}{lcl} Id\ X & = & X \\ Id\ f & = & f \end{array}$$

□

Définition 5.4.10 (Constant) *Le foncteur* ***constant*** $!A$ *est défini par son effet sur un type* X *et sur une fonction* f *comme suit :*

$$\begin{array}{lcl} !A\ X & = & A \\ !A\ f & = & id \end{array}$$

□

Définition 5.4.11 (Produit) *Le foncteur* ***produit*** $X \times Y$ *de deux types* X *et* Y *et son effet sur les fonctions* f *et* g *sont définis comme suit (par commodité et par habitude, ce foncteur* $\times$ *est utilisé en notation infixe) :*

$$\begin{array}{lcl} X \times Y & = & \{(x,y) \mid x \in X, y \in Y\} \\ (f \times g)\ (x,y) & = & (f\ x, g\ y) \end{array}$$

Les opérateurs de ***projection*** *(*π_1 *et* π_2*) et de* ***split***[15] *(*$\vartriangle$*) relatifs au produit sont :*

$$\begin{array}{lcl} \pi_1\ (a,b) & = & a \\ \pi_2\ (a,b) & = & b \\ (f \vartriangle g)\ a & = & (f\ a, g\ a) \end{array}$$

□

Définition 5.4.12 (Somme disjointe) *Le foncteur de* ***somme disjointe*** $X+Y$ *de deux types* X *et* Y *et son effet sur les fonctions* f *et* g *sont définis comme suit :*

$$\begin{array}{lcl} X + Y & = & \{1\} \times X \cup \{2\} \times Y \\ (f+g)\ (1,x) & = & (1, fx) \\ (f+g)\ (2,y) & = & (2, gy) \end{array}$$

*Les opérateurs d'****injection*** *(*ι_1 *et* ι_1*) et de* ***junc***[15] *(*$\triangledown$*) relatifs à la somme disjointe sont :*

$$\begin{array}{lcl} \iota_1\ a & = & (1,a) \\ \iota_2\ b & = & (2,b) \\ (f \triangledown g)\ (1,x) & = & fx \\ (f \triangledown g)\ (2,y) & = & gy \end{array}$$

□

Donc, dans $\mathcal{C}$, pour un endofoncteur polynômial F donné, la catégorie des F-algèbres $\mathcal{ALG}(\mathsf{F})$ possède une F-algèbre initiale [Mal90]. Elle est notée $in_F : \mathsf{F}\mu\mathsf{F} \rightarrow \mu\mathsf{F}$.

De plus, la catégorie duale possède une F-co-algèbre finale, notée $out_F : \mu\mathsf{F} \rightarrow \mathsf{F}\mu\mathsf{F}$.

Elles sont inverses l'une de l'autre et établissent un isomorphisme $\mu\mathsf{F} \cong \mathsf{F}\mu\mathsf{F}$ dans $\mathcal{C}$. Elles satisfont également l'équation

$$\mu(\lambda f.in_F \circ \mathsf{F}f \circ out_F) = id_{\mu F}$$

où μ est l'opérateur de point fixe qui satisfait $\mu h = h(\mu h)$, par analogie avec la définition d'une fonction récursive f, qui peut être exprimée comme le point fixe de l'équation $f\ x = x$. C'est également pourquoi le support de ces F-algèbres est noté $\mu\mathsf{F}$.

15. Je préfère utiliser les termes anglo-saxons *split* et *junc* qui sont plus intuitifs et plus concis que leurs traductions françaises *séparation* et *jonction*.

$$
\begin{array}{c}
\mu\,\mathsf{F} \\
in_F \Big\uparrow \qquad \Big\downarrow out_F \\
\mathsf{F}\,\mu\,\mathsf{F}
\end{array}
$$

On dit que le type $\mu\mathsf{F}$ est le *type de donnée (algébrique)* défini par le foncteur F. L'opération $in_F : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ est quelques fois appelée le *constructeur de donnée* et $out_F : \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F}$ le *destructeur (ou abstracteur)*[16] *de données.*

En somme, aux déclarations de types de données classiques correspondent des foncteurs. Pour illustrer ces notions abstraites, je propose deux exemples simples : les entiers naturels et les listes.

Le type des entiers naturels, classiquement défini par :

$$nat = zero \mid succ\ nat$$

correspond à la déclaration de la N-algèbre initiale,

$$in_N = zero \triangledown succ : \mathsf{N}\ nat \to nat$$

où le foncteur N est défini par $\mathsf{N} = !\mathbf{1} + Id$, et où $nat = \mu\mathsf{N}$ ($\mathbf{1}$ est l'objet terminal (ou final) de $\mathcal{C}$). L'opérateur inverse de in_N est out_N, défini par :

$$
\begin{array}{rcl}
out_N & = & \lambda x.\texttt{case}\ x\ \texttt{of} \\
& & \qquad zero \to (1, ()) \\
& & \qquad succ\ n \to (2, (n))
\end{array}
$$

De même, le type de donnée représentant les listes d'éléments de type A, classiquement défini par

$$list\ A = nil \mid cons\ A\ (list\ A)$$

peut être vu par sa définition catégorielle, en tant que L_A-algèbre initiale

$$in_{L_A} = nil \triangledown cons : \mathsf{L}_A(list\ A) \to (list\ A)$$

où le foncteur est $\mathsf{L}_A = !\mathbf{1} + !A \times Id$ et où $(list\ A) = \mu\mathsf{L}_A$.

La L_A-co-algèbre finale peut alors être définie par :

$$
\begin{array}{rcl}
out_{L_A} & = & \lambda x.\texttt{case}\ x\ \texttt{of} \\
& & \qquad nil \to (1, ()) \\
& & \qquad cons\ head\ tail \to (2, (head, tail))
\end{array}
$$

16. Abstracteur au sens où il remplace les constructeurs par des numéros (*tags*) de constructeurs. Par exemple, *nil* devient $(1, ())$ et $(cons\ a\ l)$ devient $(2, (a, l))$.

5.4.3 Catamorphismes et anamorphismes

Le fait que $in_F : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ soit une F-algèbre initiale dans $\mathcal{ALG}(\mathsf{F})$ implique que pour toute F-algèbre $\phi : \mathsf{F}A \to A$, il existe un unique F-homomorphisme $h : in_F \to_F \phi$, c'est-à-dire un morphisme de l'objet $\mu\mathsf{F}$ vers l'objet A dans la catégorie $\mathcal{C}$ qui vérifie $h \circ in_F = \phi \circ \mathsf{F}h$. Cet homomorphisme est appelé un *catamorphisme* et est noté $(\!|\phi|\!)_F$.

De façon duale, le fait que $out_F : \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F}$ soit une F-co-algèbre finale dans $\mathcal{COALG}(\mathsf{F})$ signifie que pour toute F-co-algèbre $\psi : A \to \mathsf{F}A$, il existe un unique F-co-homomorphisme $h : \psi \to_F out_F$, c'est-à-dire un un morphisme de l'objet A vers l'objet $\mu\mathsf{F}$ dans la catégorie $\mathcal{C}$ qui vérifie $out_F \circ h = \mathsf{F}h \circ \psi$. Ce F-co-homomorphisme est appelé *anamorphisme* et noté $[\!(\psi)\!]_F$.

Intuitivement, cela peut se représenter par les diagrammes suivants, où pour toute F-algèbre $\phi : \mathsf{F}A \to A$, et pour toute F-co-algèbre $\psi : A \to \mathsf{F}A$, il existe un unique $(\!|\phi|\!)_F$ et un unique $[\!(\psi)\!]_F$.

$$\begin{tikzcd} \mu\,\mathsf{F} \arrow[r, "(\!|\phi|\!)_F"] & A \\ \mathsf{F}\,\mu\,\mathsf{F} \arrow[u, "in_F"] \arrow[r, "\mathsf{F}(\!|\phi|\!)_F"] & \mathsf{F}\,A \arrow[u, "\phi"'] \end{tikzcd} \qquad \begin{tikzcd} \mu\,\mathsf{F} \arrow[d, "out_F"'] & A \arrow[l, "{[\!(\phi)\!]_F}"'] \arrow[d, "\psi"] \\ \mathsf{F}\,\mu\,\mathsf{F} & \mathsf{F}\,A \arrow[l, "{\mathsf{F}[\!(\psi)\!]_F}"'] \end{tikzcd}$$

Ces deux morphismes peuvent aussi être définis comme des plus petits points fixes :

$$\begin{array}{rcl} (\!|\_|\!)_F & : & (\mathsf{F}A \to A) \to \mu\mathsf{F} \to A \\ (\!|\phi|\!)_F & = & \mu(\lambda f.\phi \circ \mathsf{F}f \circ out_F) \\ [\!(\_)\!]_F & : & (A \to \mathsf{F}A) \to A \to \mu\mathsf{F} \\ [\!(\psi)\!]_F & = & \mu(\lambda f.in_F \circ \mathsf{F}f \circ \psi) \end{array}$$

Par ailleurs, les propriétés de μ, in_F et out_F permettent de déduire les propriétés suivantes :

$$\begin{array}{rcl} (\!|\phi|\!)_F & = & \phi \circ \mathsf{F}(\!|\phi|\!)_F \circ out_F \\ (\!|\phi|\!)_F \circ in_F & = & \phi \circ \mathsf{F}(\!|\phi|\!)_F \\ [\!(\psi)\!]_F & = & in_F \circ \mathsf{F}[\!(\psi)\!]_F \circ \psi \\ out_F \circ [\!(\psi)\!]_F & = & \mathsf{F}[\!(\psi)\!]_F \circ \psi \end{array}$$

Dans les sections suivantes, je montrerai de façon plus intuitive que les catamorphismes sont des opérateurs *fold* généralisés qui substituent les constructeurs d'un type de donnée par des opérations de même signature. Ce sont des *consommateurs* de structures. De façon duale, les anamorphismes sont des *producteurs* de structures et correspondent à l'opérateur inverse du *fold*. De nombreuses fonctions classiques sur des structures de données en programmation fonctionnelle peuvent être exprimées avec des catamorphismes [SF93, GLJ93].

5.4.4 Hylomorphismes

Il a déjà été remarqué qu'un *fold* était un catamorphisme (un *consommateur* de structure), et que l'opérateur dual était un anamorphisme (un *producteur* de structure). Un *hylomorphisme* est ce que l'on obtient en composant un catamorphisme avec un anamorphisme : $(\!|\phi|\!) \circ [\!(\psi)\!]$. Le diagramme présenté figure 5.14 permet d'en avoir une notion intuitive. En fait, un hylomorphisme représente la partie figée d'une récursion associée à un foncteur particulier.

$$
\begin{tikzcd}
B \arrow[rr, bend left, "{[\![\phi,\psi]\!]_F}"] \arrow[r, "{[\!(\psi)\!]_F}"'] \arrow[d, "\psi"'] & \mu\mathsf{F} \arrow[r, "{(\!|\phi|\!)_F}"'] \arrow[d, shift left, "out_F"] & A \\
\mathsf{F}B \arrow[r, "{\mathsf{F}[\!(\psi)\!]_F}"] \arrow[rr, bend right, "{\mathsf{F}[\![\phi,\psi]\!]_F}"'] & \mathsf{F}\mu\mathsf{F} \arrow[u, shift left, "in_F"] \arrow[r, "{\mathsf{F}(\!|\phi|\!)_F}"] & \mathsf{F}A \arrow[u, "\phi"']
\end{tikzcd}
$$

FIG. 5.14: *Un hylomorphisme est la composition d'un catamorphisme et d'un anamorphisme*

$$
\begin{array}{lcl}
[\![\_,\_]\!]_F & : & (\mathsf{F}A \to A) \times (B \to \mathsf{F}B) \to B \to A \\
[\![\phi,\psi]\!]_F & = & \mu(\lambda f.\phi \circ \mathsf{F}f \circ \psi)
\end{array}
$$

Ces définitions font donc des catamorphismes et des anamorphismes des cas particuliers d'hylomorphismes.

$$(\!|\phi|\!)_F = [\![\phi, out_F]\!]_F \qquad \text{et} \qquad [\!(\psi)\!]_F = [\![in_F, \psi]\!]_F$$

Un hylomorphisme $[\![\phi,\psi]\!]_F$ est une fonction récursive dont le graphe (schéma) d'appel est isomorphique au type de donnée algébrique $\mu\mathsf{F}$. La plupart des fonctions couramment utilisées peuvent être exprimées avec des hylomorphismes [BM94]. Par ailleurs, toute fonction primitive récursive sur un type algébrique peut être représentée par un hylomorphisme sur un certain type algébrique [Mee92].

Parmi les nombreux théorèmes vérifiés par ces objets, issus de la théorie des catégories, certains sont d'un intérêt tout particulier pour les transformations de programmes écrits dans ce formalisme. Avant de présenter l'un de ces théorèmes, appelé *HyloShift*, il est nécessaire de définir la notion de *transformation naturelle*.

Définition 5.4.13 (Transformation naturelle) *Soient $\mathcal{C}$ et $\mathcal{D}$ deux catégories, et $\mathsf{F},\mathsf{G}: \mathcal{C} \to \mathcal{D}$ deux foncteurs. Une **transformation** dans $\mathcal{D}$ de F vers G est une famille t de morphismes t_A :*

$$t_A : \mathsf{F}A \to_{\mathcal{D}} \mathsf{G}A \text{ pour tout } A \text{ dans } \mathcal{C}$$

*Une transformation t dans $\mathcal{D}$ de F vers G est **naturelle**, et notée $t : \mathsf{F} \dot{\to} \mathsf{G}$ si*

$$t_A \circ \mathsf{F}f = \mathsf{G}f \circ t_B \text{ pour tout } f : B \to_{\mathcal{C}} A$$

□

En fait, une transformation naturelle n'est rien d'autre qu'une sorte de "map" préservant la structure entre deux foncteurs [Fok95]. Le dessin de la figure 5.15 donne une idée intuitive de la définition. Prenons l'exemple de la catégorie $\mathcal{S}et$ des ensembles. Si F, $\mathsf{G} : \mathcal{C} \to \mathcal{S}et$ sont des foncteurs d'une catégorie $\mathcal{C}$ dans $\mathcal{S}et$, alors pour chaque A dans $\mathcal{C}$, $\mathsf{F}A$ représente un ensemble structuré et F représente la structure elle-même. Supposons alors que F représente

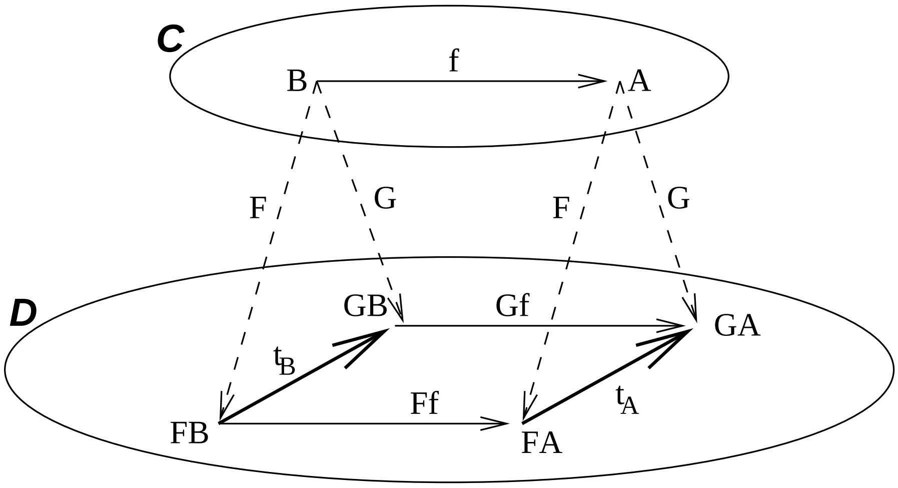

FIG. 5.15: *Transformation naturelle t dans $\mathcal{D}$ de F vers G*

la structure des paires, et G la structure des listes. Dans ce cas, FG représente la structure des paires de listes, GG la structure des listes de listes, etc. Une transformation de la structure F dans la structure G est alors une famille t de fonctions $t_A : \mathsf{F}A \to \mathsf{G}A$ effectuant un *mapping* de l'ensemble $\mathsf{F}A$ vers l'ensemble $\mathsf{G}A$ pour chaque A de $\mathcal{C}$. Une telle transformation t est dite *naturelle* si chaque t_A *n'affecte pas les constituants des éléments structurés de* $\mathsf{F}A$, mais *reforme* simplement la structure des éléments, d'une F-structure vers une G-structure. Dans cet exemple des listes et des paires [Fok95], si on considère les fonctions $join_A : \mathsf{FG}A \to \mathsf{G}A$ dans la catégorie $\mathcal{S}et$ qui transforment des paires de listes de A en des listes de A (par exemple, en concaténant les constituants de la paire), alors la famille des $join$ est une transformation naturelle $join : \mathsf{FG} \dot{\to} \mathsf{G}$ puisque :

$$join_B \circ \mathsf{FG}f = \mathsf{G}f \circ join_A \text{ pour chaque } f : A \to B$$

Cette notion de transformation naturelle permet d'énoncer un théorème, issu de la théorie des catégories, qui est d'un intérêt tout particulier pour les transformations de programmes écrits dans le formalisme des hylomorphismes. Ce théorème, qui sera utilisé dans la section suivante, établit qu'il est possible de déplacer les transformations naturelles d'un constituant à l'autre dans un hylomorphisme, soit :

Théorème 5.4.1 (HyloShift) *Si η est une transformation naturelle de F vers G, alors on a :*

$$\eta : \mathsf{F} \dot{\to} \mathsf{G} \quad \Rightarrow \quad [\![\phi \circ \eta, \psi]\!]_F = [\![\phi, \eta \circ \psi]\!]_G$$

□

5.5 Fusion d'hylomorphismes

L'idée sous-jacente aux différents formalismes de déforestation en forme calculationnelle est qu'un type de données récursif et une fonction récursive s'appliquant sur ce type peuvent être exprimés en fonction du même *foncteur*. Celui-ci représente le schéma de récursion de l'opération à effectuer en fonction du schéma de récursion de la définition du type.

Le formalisme des *folds*, qui sont des catamorphismes, exploite déjà cette relation. Cependant, un catamorphisme est un cas particulier d'un hylomorphisme et Takano et Meijer [TM95] ont développé un formalisme représentant la notion d'hylomorphisme sous une forme particulièrement adaptée à son traitement automatique, appelé *hylomorphisme sous forme de triplet.*

Un hylomorphisme sous forme de triplet est un *pattern* de récursion expressif constitué de trois parties : un opérateur d'abstraction du type, un opérateur de transformation dans cet espace abstrait et un opérateur de concrétisation de cet espace abstrait vers le type résultant. Cette section présente ce formalisme et ses atouts.

5.5.1 Hylomorphismes sous forme de triplets

À la section 5.4.4, il a déjà été signalé qu'un hylomorphisme pouvait être vu comme la composition d'un catamorphisme et d'un anamorphisme. L'idée de Takano et de Meijer [TM95] est de se servir des hylomorphismes comme composants de base pour représenter la structure des programmes et pour pouvoir ensuite les déforester par des règles simples (cf. *Acid Rain Theorems*, section 5.5.2). Cependant, l'observation suivante permet de simplifier l'application de ces théorèmes : de nombreuses fonctions se comportent à la fois comme des catamorphismes vis-à-vis de leurs paramètres et comme des anamorphismes vis-à-vis de leur résultat.

Par exemple, la fonction *length*, qui calcule la longueur d'une liste donnée, se comporte comme un catamorphisme par rapport au type *list A*, de foncteur associé $\mathsf{L}_A = !1 + !A \times Id$, et comme un anamorphisme par rapport au type *nat*, de foncteur associé $\mathsf{N} = !1 + Id$:

$$
\begin{array}{lcl}
length & = & (\![zero \triangledown (succ \circ \pi_2)]\!)_{L_A} \\
 & = & [\!(\psi)\!]_N \\
\text{où} & & \\
\psi & : & (list\ A) \rightarrow \mathsf{N}(list\ A) \\
 & = & \lambda x.\texttt{case}\ x\ \texttt{of} \\
 & & \quad nil \rightarrow (1, ()) \\
 & & \quad cons\ head\ tail \rightarrow (2, (tail))
\end{array}
$$

Par ailleurs, la loi *HyloShift* du théorème 5.4.1 spécifie que pour toute transformation naturelle $\eta : \mathsf{F} \dot{\rightarrow} \mathsf{G}$, on a :

$$
\begin{array}{lclcl}
(\![in_G \circ \eta]\!)_F & = & [\![in_G \circ \eta, out_F]\!]_F & & \\
 & = & [\![in_G, \eta \circ out_F]\!]_G & = & [\!(\eta \circ out_F)\!]_G
\end{array}
$$

De ces remarques vient l'idée de représenter les hylomorphismes en factorisant explicitement la transformation naturelle comme un troisième paramètre, ce qui permettra, dans les différentes lois utilisées pour la déforestation, de ne considérer qu'une seule représentation.

Définition 5.5.1 (Hylomorphismes sous forme de triplets) *Étant donnés ϕ: $\mathsf{G}A \rightarrow A$ et $\psi : B \rightarrow \mathsf{F}B$ deux morphismes et une transformation naturelle η: $\mathsf{F} \dot{\rightarrow} \mathsf{G}$, l'**hylomorphisme***

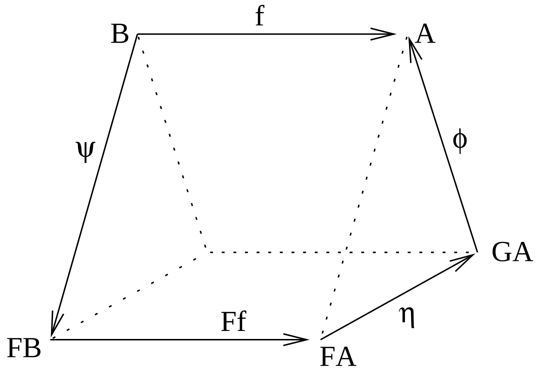

FIG. 5.16: *Idée intuitive de l'effet d'un hylomorphisme sous forme de triplet* $[\![\phi, \eta, \psi]\!]_{G,F}$

sous forme de triplet $[\![\phi, \eta, \psi]\!]_{G,F}$ *est défini comme le plus petit morphisme* $f : B \to A$ *vérifiant l'équation*

$$f = \phi \circ (\eta \circ \mathsf{F}f) \circ \psi$$

D'une manière générale, un hylomorphisme sous forme de triplet $[\![\phi, \eta, \psi]\!]_{G,F}$ *peut être défini par :*

$$\begin{array}{lcl} [\![\_, \_, \_]\!]_{G,F} & : & \forall A, B.(\mathsf{G}A \to A) \times (F \dot{\to} G) \times (B \to \mathsf{F}B) \to (B \to A) \\ [\![\phi, \eta, \psi]\!]_{G,F} & = & \mu(\lambda f.\phi \circ \eta \circ \mathsf{F}f \circ \psi) \end{array}$$

□

Plus intuitivement, on peut décomposer l'effet d'un hylomorphisme $[\![\phi, \eta, \psi]\!]_{G,F}$, défini comme le plus petit morphisme f vérifiant $f = \phi \circ (\eta \circ \mathsf{F}f) \circ \psi$, de la façon suivante (le diagramme de la figure 5.16 illustre ce propos) :

- ψ prend le paramètre d'entrée de l'hylomorphisme (de type B) et génère, à partir de son constructeur, une F-structure ($\psi : B \to \mathsf{F}B$) ;
- $\mathsf{F}f$ représente l'application du morphisme ($f : B \to A$) sur ce constructeur "abstrait" en une F-structure, en suivant la **récursion** imposée par le foncteur F autrement dit, l'application récursive de $\mathsf{F}f : \mathsf{F}B \to \mathsf{F}A$, sur le reste du paramètre d'entrée ;
- η (transformation *naturelle*) transforme la F-structure en une G-structure (η: $\mathsf{F} \dot{\to} \mathsf{G}$) ;
- ϕ construit le résultat (de type A) à partir des éléments de la G-structure, soit le résultat de l'hylomorphisme (ϕ: $\mathsf{G}A \to A$).

Pour fixer les idées, je donne ici la représentation en hylomorphisme sous forme de triplet (je dirai désormais plus simplement "hylomorphisme") de la fonction *length*.

$$length = [\![in_N, id + \pi_2, out_{L_A}]\!]_{N,L_A}$$

Par définition, cette notation représente donc le plus petit morphisme f tel que

$$f = in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f) \circ out_{L_A}$$

où je rappelle que $in_N = zero \triangledown succ$, $\mathsf{L}_A = !1 + !A \times Id$ et

$$\begin{array}{lcl} out_{L_A} & = & \lambda x.\texttt{case } x \texttt{ of} \\ & & \quad nil \rightarrow (1, ()) \\ & & \quad cons\ head\ tail \rightarrow (2, (head, tail)) \end{array}$$

L'application de cet hylomorphisme sur la liste $(cons\ a\ (cons\ b\ nil))$ donne alors :

$$\begin{array}{ll} & (f\ (cons\ a\ (cons\ b\ nil))) \\ = & ((in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f) \circ out_{L_A})\ (cons\ a\ (cons\ b\ nil))) \\ = & ((in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f))\ (2, (a, (cons\ b\ nil)))) \end{array}$$

Or, d'après les définitions et les propriétés des foncteurs,

$$\begin{array}{ll} & ((id + \pi_2) \circ \mathsf{L}_A f) \\ = & ((id + \pi_2) \circ ((!1 + !A \times Id)\ f)) \\ = & ((id + \pi_2) \circ (id + id \times f)) \\ = & id + (f \circ \pi_2) \end{array}$$

Dans le cas présent, le *tag* représentant le constructeur de la L_A-structure étant un 2, f sera donc appliquée récursivement sur le reste de la liste, soit :

$$\begin{array}{ll} & (f\ (cons\ a\ (cons\ b\ nil))) \\ = & (in_N\ ((id + (f \circ \pi_2))\ (2, (a, (cons\ b\ nil))))) \\ = & (in_N\ (2, ((f \circ \pi_2)\ (a, (cons\ b\ nil))))) \\ = & (in_N\ (2, (f\ (cons\ b\ nil)))) \\ = & ((zero \triangledown succ)\ (2, (f\ (cons\ b\ nil)))) \\ = & (succ\ (f\ (cons\ b\ nil))) \\ = & (succ\ ((in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f) \circ out_{L_A})\ (cons\ b\ nil))) \\ = & (succ\ ((in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f))\ (2, (b, nil)))) \\ = & \ldots \\ = & (succ\ (succ\ (f\ nil))) \\ = & (succ\ (succ\ ((in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f) \circ out_{L_A})\ nil))) \\ = & (succ\ (succ\ ((in_N \circ ((id + \pi_2) \circ \mathsf{L}_A f))\ (1, ())))) \\ = & (succ\ (succ\ (in_N\ ((id + (f \circ \pi_2))\ (1, ()))))) \\ = & (succ\ (succ\ (in_N\ (1, ())))) \\ = & (succ\ (succ\ ((zero \triangledown succ)\ (1, (id\ ()))))) \\ = & (succ\ (succ\ zero)) \end{array}$$

Cette représentation permet d'exprimer des fonctions, mais elle met surtout en évidence le fait que l'hylomorphisme qui la représente est un catamorphisme et/ou un anamorphisme : si le troisième paramètre est out_F, il s'agit d'un F-catamorphisme et si le premier paramètre est in_G, c'est un G-anamorphisme. L'exemple de la longueur d'une liste illustre ce propos, puisque sa représentation en hylomorphisme sous forme de triplet montre qu'il est à la fois un catamorphisme et un anamorphisme.

Plus généralement, les catamorphismes et anamorphismes s'expriment simplement en terme d'hylomorphismes sous forme de triplets :

$$(\!|\phi|\!)_F = [\![\phi, id, out_F]\!]_{F,F} \qquad [\!(\psi)\!]_F = [\![in_F, id, \psi]\!]_{F,F}$$

$$[\![\phi, \eta, \psi]\!]_{G,F}$$

$$(\mathsf{G}A \rightarrow A) \times (\mathsf{F} \dot{\rightarrow} \mathsf{G}) \times (B \rightarrow \mathsf{F}B)$$

$$[\![\phi \circ \eta, id, \psi]\!]_{F,F}$$

$$(\mathsf{F}A \rightarrow A) \times (\mathsf{F} \dot{\rightarrow} \mathsf{F}) \times (B \rightarrow \mathsf{F}B)$$

$$[\![\phi, id, \eta \circ \psi]\!]_{G,G}$$

$$(\mathsf{G}A \rightarrow A) \times (\mathsf{G} \dot{\rightarrow} \mathsf{G}) \times (B \rightarrow \mathsf{G}B)$$

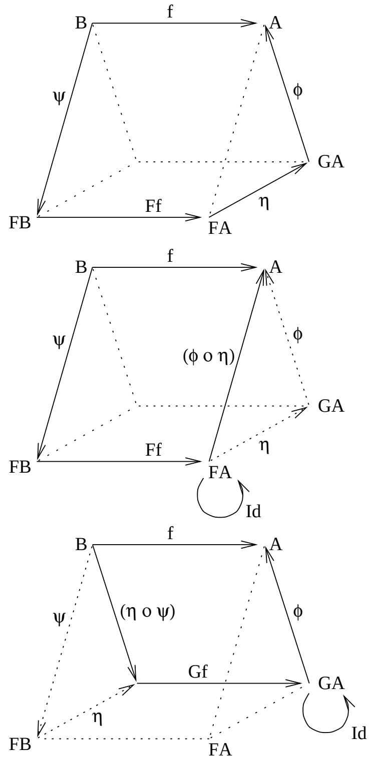

FIG. 5.17: *Illustration du théorème HyloShift*

5.5.2 Des lois intéressantes pour la déforestation

Parmi les nombreuses lois qui s'appliquent aux hylomorphismes, je donne ici celles qui sont particulièrement utiles dans le cadre de la déforestation, exprimées pour des hylomorphismes en forme de triplets.

Tout d'abord, la loi qui permet de déplacer la transformation naturelle d'un argument sur l'autre, déjà présentée sous sa forme classique au théorème 5.4.1 et qui est illustrée à la figure 5.17.

Théorème 5.5.1 (HyloShift)

$$[\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi \circ \eta, id, \psi]\!]_{F,F} = [\![\phi, id, \eta \circ \psi]\!]_{G,G}$$

□

Ensuite, une loi très générale, appelée *théorème de promotion du fold* dans [SF93] (Théorème 5.3.1) ou encore *fixed point fusion* dans [MFP91] et que j'appellerai ici *HyloFusion* comme dans [TM95] ; les deux déclinaisons de *HyloFusion* sont illustrées à la figure 5.18.

Théorème 5.5.2 (HyloFusion) *Cette loi se dérive pour la fusion à gauche et à droite d'un hylomorphisme et d'une fonction (qui est du type adéquat) :*

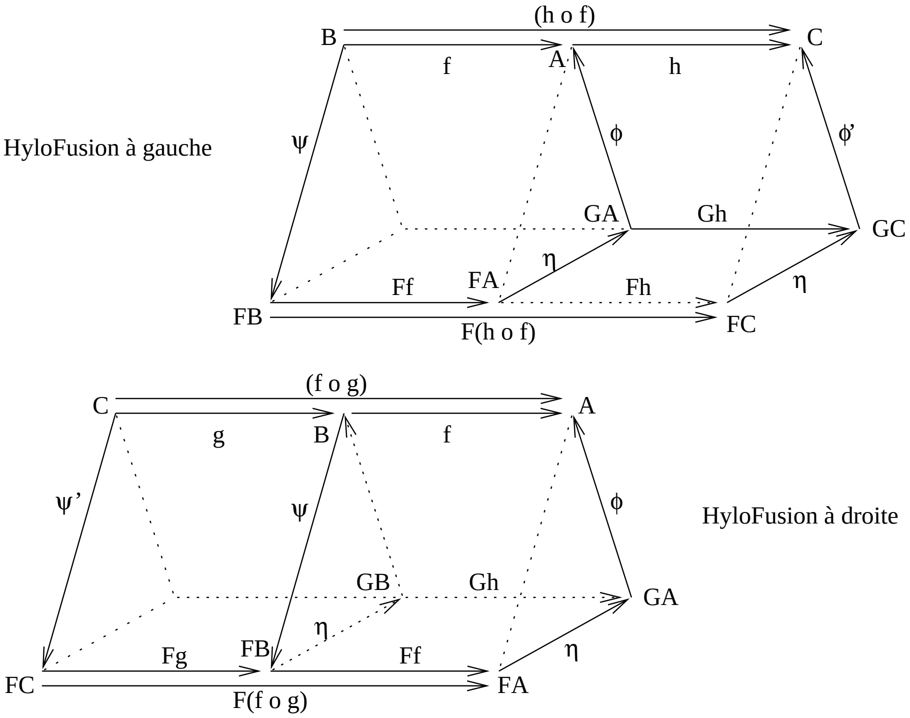

FIG. 5.18: *Illustration du théorème HyloFusion à gauche et à droite*

HyloFusion à gauche *:*

$$\frac{h \circ \phi = \phi' \circ Gh}{h \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F}}$$

HyloFusion à droite *:*

$$\frac{\psi \circ g = Fg \circ \psi'}{[\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F}}$$

□

Ces deux lois sont générales, mais elles prennent une forme particulièrement simple dans le cas où la fonction que l'on fusionne avec l'hylomorphisme est elle même un hylomorphisme particulier. Elles se dérivent alors dans le fameux théorème de Meijer, *Acid Rain* qui déforeste les compositions d'hylomorphismes autant que les pluies acides déforestent les forêts de chênes verts … Les deux lois sont illustrées dans la figure 5.19.

Théorème 5.5.3 (Acid Rain) *Les deux lois suivantes sont symétriques et permettent de fusionner simplement la composition entre un hylomorphisme et une fonction qui est soit un catamorphisme, soit un anamorphisme.*

Loi de fusion à gauche, aussi appelée ***Cata-HyloFusion*** *:*

$$\tau : \forall A.(FA \to A) \to F'A \to A$$
$$\Rightarrow\ [\![\phi, \eta_1, out_F]\!]_{G,F} \ \circ\ [\![\tau in_F, \eta_2, \psi]\!]_{F',L} \ =\ [\![\tau(\phi \circ \eta_1), \eta_2, \psi]\!]_{F',L}$$

Loi de fusion à droite, aussi appelée ***Hylo-AnaFusion****:*

$$\begin{aligned} & \sigma : \forall A.(A \to \mathsf{F}A) \to A \to \mathsf{F}'A \\ \Rightarrow\ & [\![\phi, \eta_1, \sigma out_F]\!]_{G,F'} \ \circ\ [\![in_F, \eta_2, \psi]\!]_{F,L} \ =\ [\![\phi, \eta_1, \sigma(\eta_2 \circ \psi)]\!]_{G,F'} \end{aligned}$$

□

Dans le théorème 5.5.3, les *foncteurs d'algèbres* τ et σ sont polymorphes, c'est-à-dire qu'ils sont valables pour tout A (et donc en particulier pour $\mu\mathsf{F}$, cf. illustration figure 5.19) ; ils permettent de passer d'une F-algèbre vers une F'-algèbre de même support. Ce polymorphisme permet de prouver les lois de Cata-HyloFusion et de Hylo-AnaFusion, qui ne sont que des instances particulières d'un théorème plus général également appelé *Acid Rain* par Meijer [TM95]. Ce théorème est lui même dérivé des *theorems for free* de Wadler, fondés sur la paramétrisation [Wad89]. Je ne rentrerai pas plus loin dans les détails et les preuves de ces théorèmes, mais je vais plutôt m'intéresser à la façon dont ils peuvent être utilisés.

Avec ces lois, il est désormais possible de fusionner deux hylomorphismes et donc d'espérer déforester leur composition. Takano et Meijer [TM95] se sont tout d'abord placés dans le cadre théorique où les programmes sont entièrement spécifiés par des compositions d'hylomorphismes.

5.5.3 Déforestation par fusion d'hylomorphismes

Je présente dans cette section un premier exemple de fusion. Ensuite, je présente brièvement la stratégie de fusion de plusieurs hylomorphismes et finalement une transformation qui permet de dériver des hylomorphismes à partir de définitions récursives classiques. Je reviendrai plus longuement à la section 5.5.5 sur le détail de quelques exemples de fusion.

Un exemple

Pour l'instant, je veux illustrer les notions déjà abordées dans le contexte de la déforestation. J'ai déjà présenté l'hylomorphisme *length* spécifiant la longueur d'une liste ; il va être utilisé dans cet exemple. Je rappelle que le foncteur du type *list* A est $\mathsf{L}_A = !\mathbf{1} + !A \times Id$ et que *length* retourne un entier naturel *nat*, dont le foncteur associé est $\mathsf{N} = !\mathbf{1} + Id$. Cet hylomorphisme est donc défini par :

$$\begin{array}{lcl} length & : & (list\ A) \to nat \\ length\ x & = & [\![in_N, id + \pi_2, out_{L_A}]\!]_{N,L_A}\ x \end{array}$$

Je considère maintenant la fonction *append* qui concatène deux listes du même type, *list* A. L'hylomorphisme représentant cette fonction peut être défini par :

$$\begin{array}{lcl} append & : & (list\ A) \times (list\ A) \to (list\ A) \\ append\ x_1\ x_2 & = & \lambda x_1, x_2.[\![\tau in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A}\ x_1 \\ \text{où} & & \\ & & \tau = \lambda n \triangledown c.([\![n \triangledown c, id, out_{L_A}]\!]_{L_A,L_A}\ x_2) \triangledown c \end{array}$$

Intuitivement, out_{L_A} abstrait la liste x_1 sous la forme de $(1, ())$ pour le *nil* et $(2, (h, t))$ pour les (*cons* h t). Ensuite, τin_{L_A} *concrétise* les $(2, (h, t))$ en des (*cons* h t) et effectue pour les $(1, ())$ une abstraction/concrétisation de x_2. En d'autres termes, chaque *cons* de x_1 est recopié tel quel et le *nil* de x_1 est remplacé par une recopie de x_2.

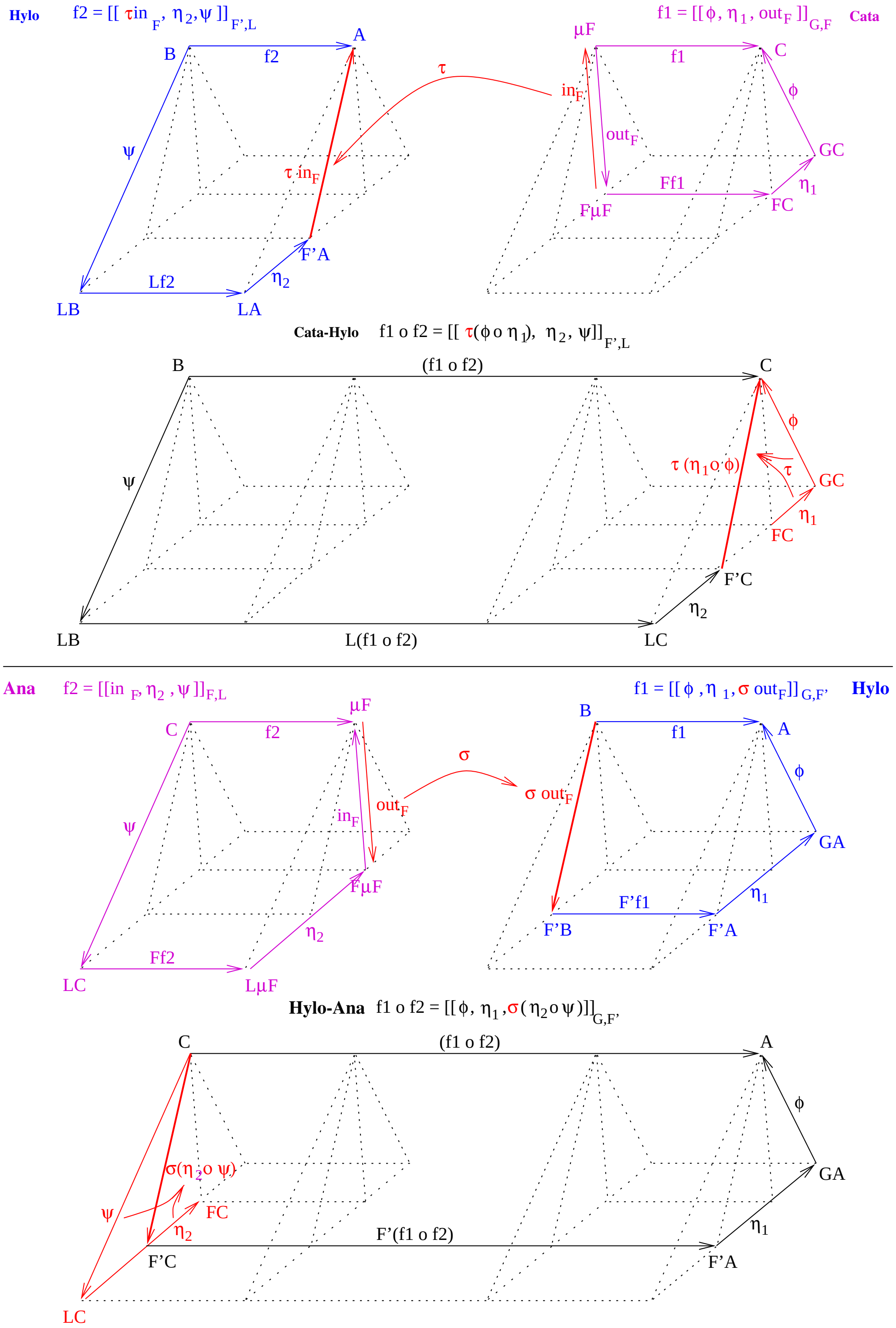

FIG. 5.19: *Illustration des Acid Rain Theorems*

Je m'intéresse maintenant au programme formé par la composition de *length* et de *append* ; je présente ci-dessous l'application des résultats énoncés à la section précédente sur cet exemple, dans le but de fusionner les deux hylomorphismes correspondants :

$$
\begin{array}{l}
length \circ append \\
\qquad \text{par définition de } length \text{ et } append \\
= [\![in_N, id + \pi_2, out_{L_A}]\!]_{N,L_A} \circ [\![\tau in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A} \\
\qquad \text{par Cata-HyloFusion} \\
= [\![\tau(in_N \circ (id + \pi_2)), id, out_{L_A}]\!]_{L_A,L_A} \\
\qquad \text{par définition de } in_N \\
= [\![\tau((zero \triangledown succ) \circ (id + \pi_2)), id, out_{L_A}]\!]_{L_A,L_A} \\
\qquad \text{par propriétés des foncteurs de base} \\
= [\![\tau(zero \triangledown (succ \circ \pi_2)), id, out_{L_A}]\!]_{L_A,L_A} \\
\qquad \text{par définition de } \tau \\
= [\![\ ([\![zero \triangledown (succ \circ \pi_2), id, out_{L_A}]\!]_{L_A,L_A} \ x_2) \triangledown (succ \circ \pi_2), id, out_{L_A}]\!]_{L_A,L_A}
\end{array}
$$

Cette définition, un peu alambiquée, devient plus claire lorsqu'on la traduit en définition récursive classique :

$$
\begin{array}{l}
length \circ append = f \\
\qquad \text{où} \\
\quad f\ x_1\ x_2 = \texttt{case}\ x_1\ \texttt{of} \\
\qquad nil \rightarrow g\ x_2 \\
\qquad cons\ head\ tail \rightarrow ((succ \circ \pi_2)\ head\ (f\ tail\ x_2)) \\
\qquad \qquad \qquad \qquad \quad = succ\ (f\ tail\ x_2) \\
\qquad \text{avec} \\
\quad g\ x_2 = \texttt{case}\ x_2\ \texttt{of} \\
\qquad nil \rightarrow zero \\
\qquad cons\ head\ tail \rightarrow ((succ \circ \pi_2)\ head\ (g\ tail)) \\
\qquad \qquad \qquad \qquad \quad = succ\ (g\ tail)
\end{array}
$$

En effet, dans cette version, après fusion des hylomorphismes, plus aucun constructeur de liste n'apparaît : la construction de la liste intermédiaire a été éliminée et le programme correspondant est déforesté.

Ordre de fusion des compositions d'hylomorphismes

Tout le problème consiste alors à trouver une stratégie pour l'ordre d'application des règles de transformations. On peut remarquer que l'application de Cata-HyloFusion sur la composition de deux hylomorphismes $h_1 \circ h_2$ laisse toujours inchangé le troisième paramètre (ψ) de h_2 dans l'hylomorphisme résultant. De façon duale, l'application de Hylo-AnaFusion laisse inchangé le premier paramètre (ϕ) de h_1 dans l'hylomorphisme résultant. La question de l'ordre d'application de ces règles se pose lorsque l'on est en présence de la composition de trois (ou plus) hylomorphismes $h_1 \circ h_2 \circ h_3$ dans un programme et qu'il est possible d'appliquer une loi sur $h_1 \circ h_2$ ou sur $h_2 \circ h_3$. On dit alors qu'il y a deux *redex* possibles, ces *redex* pouvant être *Cata-Hylo* ou *Hylo-Ana*. Dans [TM95], Meijer et Takano font remarquer que, comme il y a deux types de redex possibles et qu'on cherche à connaître leur ordre de réduction en fusionnant deux compositions, il y a quatre combinaisons possibles :

1. Deux redex Cata-Hylo :

$$[\![\phi, \eta_1, out_F]\!] \circ [\![\tau_1 in_F, \eta_2, out_G]\!] \circ [\![\tau_2 in_G, \eta_3, \psi]\!]$$

   Dans ce cas, la réduction du redex de gauche ne détruit pas le redex de droite.

2. Deux redex Hylo-Ana :

$$[\![\phi, \eta_1, \sigma_1 out_F]\!] \circ [\![in_F, \eta_2, \sigma_2 out_G]\!] \circ [\![in_G, \eta_3, \psi]\!]$$

   Dans ce cas, la réduction du redex de droite ne détruit pas le redex de gauche.

3. Un redex Cata-Hylo à gauche avec un redex Hylo-Ana à droite :

$$[\![\phi, \eta_1, out_F]\!] \circ [\![\tau in_F, \eta_2, \sigma out_G]\!] \circ [\![in_G, \eta_3, \psi]\!]$$

   Dans ce cas, la réduction d'un redex ne détruit pas l'autre redex.

4. Un redex Hylo-Ana à gauche avec un redex Cata-Hylo à droite :

$$[\![\phi, \eta_1, \sigma out_F]\!] \circ [\![in_F, \eta_2, out_G]\!] \circ [\![\tau in_G, \eta_3, \psi]\!]$$

   Dans ce cas, la réduction d'un redex ne détruit pas l'autre redex.

Les auteurs introduisent alors la notion de *chaîne de redex Cata-Hylo* – respectivement *chaîne de redex Hylo-Ana* – comme étant une série de redex Cata-Hylo – resp. Hylo-Ana. Ils donnent alors l'ordre de réduction préconisé pour les fusions d'hylomorphismes, qui définit leur algorithme de fusion :

1. Trouver toutes les chaînes maximales de redex Cata-Hylo et réduire chacune de ces chaînes de gauche à droite.

2. Trouver toutes les chaînes maximales de redex Hylo-Ana et réduire chacune de ces chaînes de droite à gauche.

3. Simplifier les composants de chaque hylomorphisme grâce aux propriétés des foncteurs de base et de leurs opérateurs associés ($\triangledown$, π, etc ...).

4. S'il existe encore des redex pour des HyloFusions, retourner en 1 et continuer la réduction.

Appliquer la fusion d'hylomorphismes à des définitions récursives classiques

L'ensemble de la présentation qui a été faite jusqu'ici permet donc de fusionner, tout en déforestant, des programmes fonctionnels qui sont exprimés en terme de composition d'hylomorphismes. Le problème est qu'il n'est pas forcément aisé, ni intuitif, d'écrire directement des programmes dans le formalisme des hylomorphismes. L'idée s'est donc imposée d'une traduction automatique de programmes d'un langage fonctionnel classique en composition d'hylomorphismes. Hu, Iwasaki et Takeichi [HIT96b] se sont attaqués à ce problème, en partant d'un langage de description de définitions récursives proche des langages fonctionnels classiques et en donnant un algorithme qui traduit automatiquement ces définitions récursives en hylomorphismes. Je ne présente pas ici cet algorithme dans le détail, mais j'en donne le principe général (le lecteur pourra se référer à [HIT96b, HIT96a, OHIT97] pour plus de détails).

Je considère une définition récursive de la forme :

$$
\begin{aligned}
f = \lambda\overline{v}.\texttt{case}\ & t_0\ \texttt{of} \\
& p_1 \to t_1 \\
& \ldots \\
& p_n \to t_n \\
\text{où } \overline{v} = v \mid (v_1 \ldots v_n) & \text{ sont des variables}
\end{aligned}
$$

Le but est alors de transformer la partie droite de la définition de f sous la forme $\phi \circ \mathsf{F} f \circ \psi$, puisqu'alors, par définition des hylomorphismes (Définition 5.5.1), on aura $f = [\![\phi, id, \psi]\!]_{F,F}$. L'idée pour cette transformation est d'utiliser les appels récursifs de f dans les t_i ; en fait, il faut tenter d'exprimer chaque t_i sous la forme d'un $g_i\ t'_i$. Grâce à cela, en extrayant chacun de ces g_i, il est possible, en ajoutant une marque i à chaque t_i, de représenter f par une description compositionnelle :

$$
\begin{aligned}
f = (g_1 \triangledown \ldots \triangledown g_n) \quad \circ \quad (\lambda\overline{v}.\texttt{case}\ & t_0\ \texttt{of} \\
& p_1 \to (\mathsf{1}, t'_1) \\
& \ldots \\
& p_n \to (\mathsf{n}, t'_n))
\end{aligned}
$$

La tâche ardue est ensuite d'exprimer chaque g_i par $\phi_i \circ \mathsf{F}_i f$ où F_i est un foncteur, ce qui permet de représenter les g_i comme ceci :

$$g_1 \triangledown \ldots \triangledown g_n = (\phi_1 \triangledown \ldots \triangledown \phi_n) \circ (\mathsf{F}_1 + \ldots + \mathsf{F}_n) f$$

et en intégrant cette définition de g dans la description compositionnelle de f, on obtient :

$$
\begin{aligned}
f = (\phi_1 \triangledown \ldots \triangledown \phi_n) \quad \circ \quad (\mathsf{F}_1 + \ldots + \mathsf{F}_n) f \quad \circ \quad (\lambda\overline{v}.\texttt{case}\ & t_0\ \texttt{of} \\
& p_1 \to (\mathsf{1}, t'_1) \\
& \ldots \\
& p_n \to (\mathsf{n}, t'_n))
\end{aligned}
$$

soit la définition de l'hylomorphisme recherché $f = [\![\phi, id, \psi]\!]_{F,F}$. C'est cette dérivation de la fonction ϕ_i, du foncteur F_i et du nouveau terme t'_i qu'effectue l'algorithme $\mathcal{D}$ décrit dans [HIT96b] et implanté dans le système HYLO [OHIT97].

Reste à ce système HYLO à fusionner de tels hylomorphismes, à l'aide des lois classiques, à savoir HyloFusion et Acid Rain. Hu *et al.* précisent dans [HIT96b] que la forme la mieux adaptée pour un hylomorphisme à fusionner dépend de la loi utilisée. En effet, pour HyloFusion, on recherche une forme où la partie ϕ (resp. ψ) contient le maximum de calculs pour fusionner à gauche (resp. à droite). Par contre, pour Acid Rain, la forme recherchée est celle qui fait apparaître ϕ (resp. ψ) sous la forme τin_F (resp. σout_F) pour fusionner à gauche (resp. à droite).

Comme dans un hylomorphisme $[\![\phi, \eta, \psi]\!]_{G,F}$, la partie transformation naturelle η peut être déplacée dans la partie ϕ ou dans la partie ψ par HyloShift (Théorème 5.5.1), on dit que ϕ (ψ)

contient les *plus petits calculs* s'il ne contient pas de calculs que η pourrait contenir. Aussi, les auteurs introduisent la notion d'*hylomorphisme structurel* : un hylomorphisme $[\![\phi, \eta, \psi]\!]_{G,F}$ est dit *structurel* si à la fois ϕ et ψ contiennent les plus petits calculs. Les hylomorphismes structurels sont donc assez bien adaptés pour Acid Rain puisque leurs parties ϕ et ψ sont simples ; ils sont également bien adaptés pour HyloFusion puisque qu'en faisant passer la partie η (qui contient, en un sens, le maximum de calcul) dans ϕ (resp. ψ), on obtient un hylomorphisme idéal pour la fusion à gauche (resp. droite). Des algorithmes permettant de restructurer les hylomorphismes en ce sens sont donnés dans [HIT96b, OHIT97] et le système HYLO en implante une partie.

5.5.4 Comparaison avec les autres méthodes de déforestation

Avant de revenir plus précisément sur certaines extensions et limitations de la déforestation par fusion d'hylomorphismes (section 5.5.5), je compare ici le principe et l'effet de cette approche par rapport aux autres méthodes de déforestation fonctionnelles déjà présentées.

Tout d'abord, la notion d'hylomorphisme généralise la notion de *fold*. En effet, les *folds* sont des cas particuliers ou des instances de catamorphismes. Or, ces catamorphismes sont eux mêmes des hylomorphismes particuliers. Le formalisme des hylomorphismes autorise donc la représentation de programmes plus généraux que ceux représentés par des *folds* et de plus, la représentation sous forme de triplet permet de reconnaître, par exemple, un *fold* dans un hylomorphisme. La comparaison entre l'algorithme de déforestation de Wadler et la fusion d'hylomorphismes étant rendue difficile par l'approche très *calculationnelle* de ce dernier formalisme, je me contenterai de rappeler l'équivalence de déforestation effectuée par l'algorithme de normalisation des *folds* sur les termes canoniques et l'algorithme de déforestation de Wadler sur les termes *treeless* du premier ordre (cf. section 5.3.5).

L'opérateur dual du *fold*, quelquefois appelé *unfold* ou *build*, tel que celui utilisé pour les règles d'éliminations `foldr/build` sur des listes dans [GLJ93] (cf. section5.2.4), correspond également à une instance particulière d'un anamorphisme, lui-même cas particulier d'hylomorphisme.

En terme d'expressivité, le formalisme des hylomorphismes sous forme de triplets permet donc de généraliser les précédents travaux, en représentant les programmes d'une façon homogène avec ces objets de base.

En terme de déforestation, les théorèmes présentés pour les hylomorphismes et les règles de transformation qui leur sont associées (*HyloShift*, *HyloFusion*, *Acid Rain*) constituent une double généralisation par rapport aux travaux précédents. Par la représentation catégorielle des types, elles ne s'appliquent plus uniquement aux listes, comme la règle `foldr/build` de [GLJ93]. Par la représentation dans le même formalisme des *fold* et *unfold*, les lois de fusion des hylomorphismes permettent de généraliser la règle `foldr/build` et son dual, ainsi que l'algorithme de normalisation des *folds*, dans une forme plus facilement automatisable, par application de règles simples.

Finalement, une différence importante entre la normalisation des *folds* et la fusion des hylomorphismes tient à la forme du foncteur utilisé dans chacun des cas. Dans les travaux présentés à la section 5.3, les *folds* pris en compte sont dirigés par des foncteurs générés uniquement à partir des définitions des types algébriques. Ces foncteurs ne sont réellement adaptés qu'à la représentation de fonctions dont le schéma de récursion est isomorphe au type de donnée considéré. Grâce à l'algorithme de dérivation d'un hylomorphisme à partir de définitions récursives classiques introduit par Hu *et al.* [HIT96b, OHIT97], dont j'ai brièvement présenté l'idée à la fin de la section 5.5.3, les foncteurs ne sont plus uniquement générés par la définition du type de donnée considéré, mais également par le schéma de récursion de la fonction prise en compte. Cette caractéristique permet de limiter l'effet d'*encapsulation* des calculs difficiles à déforester, relevé à la section 5.3.5 au sujet des *folds* dans le cas du *reverse* par exemple. En effet, la déforestation étant guidée par les foncteurs, ceux-ci doivent être le plus près possible de la construction des structures intermédiaires. Malgré cet assouplissement, ces foncteurs restent encore trop "rigides" pour permettre la déforestation des constructions effectuées dans les paramètres d'accumulation des fonctions ; je présenterai à la fin de la prochaine section des exemples illustrant ce problème.

5.5.5 Problèmes et limites de la méthode

Parmi les différentes méthodes de déforestation en forme calculationnelle, la fusion des hylomorphismes est particulièrement riche, mais également complexe de par l'obscurité des notations. Aussi, le déroulement d'un exemple permettra au lecteur, si besoin est, de se familiariser avec ce formalisme.

De plus, il me permettra de faire état d'un problème relevé dans [TM95] et d'en proposer une explication.

Finalement, cet exemple me permettra de mettre en évidence des limitations de cette technique de fusion d'hylomorphismes dans le cadre de la déforestation de programmes.

L'exemple en filigrane dans toute cette partie est une fois de plus la fonction d'inversion de liste.

Un exemple

Je considère donc l'hylomorphisme *rev_naif*, correspondant à l'inversion de liste vue sous sa forme naïve, tel qu'il est donné dans [TM95] :

$$\begin{array}{lcl} rev\_naif & : & (list\ A) \to (list\ A) \\ rev\_naif\ x & = & [\![\sigma in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A}\ x \\ \text{où} & & \\ & & \sigma = \lambda n_\triangledown c.(n_\triangledown(\lambda h, r.[\![(c\ h\ n)_\triangledown c, id, out_{L_A}]\!]_{L_A,L_A}\ r)) \end{array}$$

Cet hylomorphisme exprimé sous la forme d'une fonction récursive classique peut être vu comme ceci :

$$\begin{array}{l} rev\_naif = f \\ \quad\quad \text{où} \\ \quad f\ x = \texttt{case}\ x\ \texttt{of} \\ \quad\quad nil \to nil \\ \quad\quad cons\ head\ tail \to g\ head\ (f\ tail) \\ \quad\quad\quad \text{avec} \\ \quad g\ h\ r\ = \texttt{case}\ r\ \texttt{of} \\ \quad\quad nil \to cons\ h\ nil \\ \quad\quad cons\ head\ tail \to cons\ head\ (g\ h\ tail) \end{array}$$

Intuitivement, par cet hylomorphisme *rev_naif*, une liste $(cons\ a\ (cons\ b\ (nil)))$ est récursivement abstraite par out_{L_A} en $(2,(a,(2,(b,(1,()))))))$.

La transformation naturelle identité et le foncteur $\mathsf{L}_A =!1+!A \times Id$ conservant cette abstraction à chaque récursion, il reste à $\sigma in_{L_A} = \sigma(nil \triangledown cons)$ à lui être appliquée récursivement.

Les définitions de σ et de in_{L_A} permettent d'établir :

$$\sigma(nil \triangledown cons) = (nil \triangledown (\lambda h, r. [\![(cons\ h\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ r))$$

où, ici, r est la variable d'accumulation de résultat déjà rencontrée pour les *folds*. En fait, cette liste résultante est construite à partir de la fin de l'abstraction, en "remontant". Chaque application de σin_F sur un couple $(2,(h,t))$ entraîne l'abstraction, puis la concrétisation de la sous liste t.

Voici ce que cela donne sur une liste prise en exemple :

$$\begin{array}{ll}
 & \sigma in_F\ (2,(a,(\sigma in_F\ (2,(b,(\sigma in_F\ (1,())))))))) \\
= & \sigma in_F\ (2,(a,(\sigma in_F\ (2,(b, \\
 & \quad (nil \triangledown (\lambda h, r. [\![(cons\ h\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ r))\ (1,()))))))) \\
= & \sigma in_F\ (2,(a,(\sigma in_F\ (2,(b, nil))))) \\
= & \sigma in_F\ (2,(a,(\\
 & \quad (nil \triangledown (\lambda h, r. [\![(cons\ h\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ r))\ (2,(b, nil))))) \\
= & \sigma in_F\ (2,(a,(\\
 & \quad [\![(cons\ b\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ nil))) \\
= & \sigma in_F\ (2,(a,(cons\ b\ nil))) \\
= & (nil \triangledown (\lambda h, r. [\![(cons\ h\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ r))\ (2,(a,(cons\ b\ nil))) \\
= & [\![(cons\ a\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ (cons\ b\ nil) \\
= & ((cons\ a\ nil) \triangledown cons)\ (2,(b, [\![(cons\ a\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ nil)) \\
= & ((cons\ a\ nil) \triangledown cons)\ (2,(b, [\![(cons\ a\ nil) \triangledown cons, id, out_{L_A}]\!]_{L_A, L_A}\ nil)) \\
= & ((cons\ a\ nil) \triangledown cons)\ (2,(b,(cons\ a\ nil))) \\
= & cons\ b\ (cons\ a\ nil)
\end{array}$$

Un problème

Je considère maintenant un programme qui calcule la longueur d'une liste inversée, soit la composition *length* ∘ *rev_naif*. Les lois sur les hylomorphismes vues précédemment permettent de fusionner les deux fonctions de ce programme de la manière suivante :

$$\begin{array}{l}
length \circ rev_naif \\
\qquad \text{par définitions en hylomorphismes} \\
= [\![in_N, id + \pi_2, out_{L_A}]\!]_{N, L_A} \circ [\![\sigma in_{L_A}, id, out_{L_A}]\!]_{L_A, L_A} \\
\qquad \text{par Cata-HyloFusion} \\
= [\![\sigma(in_N \circ (id + \pi_2)), id, out_{L_A}]\!]_{L_A, L_A} \\
\qquad \text{par définition de } in_N \text{ et propriétés des foncteurs} \\
= [\![\sigma(zero \triangledown (succ \circ \pi_2)), id, out_{L_A}]\!]_{L_A, L_A} \\
\qquad \text{par définition de } \sigma \\
= [\![zero \triangledown (\lambda h, r. [\![((succ \circ \pi_2)\ h\ zero) \triangledown (succ \circ \pi_2), id, out_{L_A}]\!]_{L_A, L_A}\ r), \\
\qquad id, out_{L_A}]\!]_{L_A, L_A}
\end{array}$$

Un problème se pose dans le cadre de cet exemple, et je vais tenter de l'exposer. En effet, dans [TM95], les auteurs simplifient cet exemple pour obtenir la forme suivante :

$$\begin{array}{c} length \circ rev\_naif = \quad (\clubsuit) \\ [\![zero_\triangledown (\lambda h, r.[\![(succ\ zero)_\triangledown (succ \circ \pi_2), id, out_{L_A}]\!]_{L_A, L_A}\ r), id, out_{L_A}]\!]_{L_A, L_A} \end{array}$$

Je vais présenter ci-dessous ce que donne cette dernière forme pour $length \circ rev\_naif$ lorsque elle est appliquée à la liste $(cons\ a\ (cons\ b\ (nil)))$. Pour simplifier, j'adopte localement ici les notations :

$$\begin{array}{ll} & length \circ rev\_naif = f = [\![\phi, \eta, \psi]\!]_{L_A, L_A} \\ \text{où} & \phi = zero_\triangledown(\lambda h, r.(g\ r)) \\ \text{avec} & \quad g = [\![\phi', id, out_{L_A}]\!]_{L_A, L_A} \\ & \qquad \phi' = (succ\ zero)_\triangledown(succ \circ \pi_2) \\ & \eta = id \\ & \psi = out_{L_A} \end{array}$$

D'après la définition des hylomorphismes, je rappelle que f est défini récursivement comme ceci :

$$f = \phi \circ (\eta \circ \mathsf{L}_A f) \circ \psi$$

Cette définition appliquée à notre exemple produit ceci :

$$\begin{array}{ll} & f\ (cons\ a\ (cons\ b\ (nil))) \\ = & \phi \circ (\eta \circ \mathsf{L}_A f) \circ \psi\ (cons\ a\ (cons\ b\ (nil))) \\ = & \phi \circ (\eta \circ \mathsf{L}_A f)\ (2, (a, (cons\ b\ (nil)))) \\ = & \phi(2, (a, (f\ (cons\ b\ (nil))))) \\ = & \ldots \\ = & \phi(2, (a, (\phi(2, (b, (\phi(1, ())))))))) \end{array}$$

L'application de ϕ sur $(1, ())$ génère directement $zero$, donc :

$$\begin{array}{ll} & f\ (cons\ a\ (cons\ b\ (nil))) \\ = & \phi(2, (a, (\phi(2, (b, zero))))) \end{array}$$

Par contre, l'application de ϕ sur $(2, (b, zero))$ fait appel à la fonction notée g plus haut, où les paramètres formels h et r sont respectivement associés à b et à $zero$:

$$\begin{array}{ll} & f\ (cons\ a\ (cons\ b\ (nil))) \\ = & \phi(2, (a, (g\ zero))) \\ = & \phi(2, (a, ([\![\phi', id, out_{L_A}]\!]_{L_A, L_A}\ zero))) \end{array}$$

Le problème ici est que out_{L_A} ne peut pas être appliqué à $zero$. Par ailleurs, si c'était out_N pour les naturels, ça n'irait pas non plus, puisque le constructeur 2 de ϕ' (deuxième membre du $_\triangledown$) commence par faire une projection sur le deuxième argument d'un couple de paramètres, or il n'y a qu'un seul paramètre dans le cas des naturels.

Après avoir longtemps cherché à savoir d'où provient ce problème, laissé dans l'ombre des articles publiés dans des conférences internationales, je pense que le typage de la fonction out_{L_A}, qui provient de la définition de σ dans la définition originale de rev_naif, *doit* dépendre des paramètres formels n et c qui représentent des constructeurs du type traité.

J'ai envisagé deux manières de remédier à ce problème, sans qu'elles soient pour autant complètement satisfaisantes.

La première méthode consiste, en plus du **paramètrage de** σ par n et c à ne **pas effectuer le dernier pas de simplification** (♣) mis en évidence plus haut et à considérer les constructeurs n et c comme des *entités* que l'on ne doit pas modifier ou simplifier avant la fin d'un calcul.

En effet, si on considère la définition de l'hylomorphisme un pas de simplification plus tôt, et que l'on utilise une fonction $out_{n,c}$ au lieu de out_{L_A}, on obtient :

$$
\begin{array}{ll}
 & length \circ rev\_naif = f = [\![\phi, \eta, \psi]\!]_{L_A,L_A} \\
\text{où} & \phi = zero_{\triangledown}(\lambda h, r.(g_h\ r)) \\
\text{avec} & \quad g_h = [\![\phi''_h, id, \psi'']\!]_{L_A,L_A} \\
 & \qquad \phi''_h = ((succ \circ \pi_2)\ h\ zero)_{\triangledown}(succ \circ \pi_2) \\
 & \qquad \psi'' = out_{n,c} \\
 & \eta = id \\
 & \psi = out_{L_A}
\end{array}
$$

Il est alors possible de définir la fonction $out_{n,c}$ comme effectuant l'abstraction d'une *donnée* définie à l'aide des constructeurs n et c considérés, soit ici $zero$ et $(succ \circ \pi_2)$, en les considérant dans l'intégrité de leur forme transmise lors de l'instanciation du σ :

$$
\begin{array}{l}
out_{n,c}\ x = \texttt{case}\ x\ \texttt{of} \\
\quad zero \rightarrow (1, ()) \\
\quad (succ \circ \pi_2)\ j\ k \rightarrow (2, (j, k))
\end{array}
$$

Je reprends alors l'exécution qui posait problème plus haut :

$$
\begin{array}{ll}
 & f\ (cons\ a\ (cons\ b\ (nil))) \\
= & \phi(2, (a, (\phi(2, (b, (\phi(1, ()))))))) \\
= & \phi(2, (a, (\phi(2, (b, zero))))) \\
= & \phi(2, (a, ((\lambda h, r.(g_h\ r))\ (b, zero)))) \\
= & \phi(2, (a, (g_b\ zero))) \\
= & \phi(2, (a, ([\![\phi''_b, id, out_{n,c}]\!]_{L_A,L_A}\ zero))) \\
= & \phi(2, (a, ([\![((succ \circ \pi_2)\ b\ zero)_{\triangledown}(succ \circ \pi_2), id, out_{n,c}]\!]_{L_A,L_A}\ zero)))
\end{array}
$$

L'application de ϕ sur $(1, ())$ a généré directement $zero$, mais l'application de ϕ sur $(2, (b, zero))$ fait appel à $(g_b\ zero)$, puisque les paramètres formels h et r sont respectivement associés à b et à $zero$. $out_{n,c}$ peut alors être appliqué à $zero$:

$$
\begin{array}{ll}
 & f\ (cons\ a\ (cons\ b\ (nil))) \\
= & \phi(2, (a, (\phi''_b\ (1, ())))) \\
= & \phi(2, (a, (((succ \circ \pi_2)\ b\ zero)_{\triangledown}(succ \circ \pi_2))\ (1, ()))) \\
= & \phi(2, (a, ((succ \circ \pi_2)\ b\ zero)))
\end{array}
$$

Récursivement, l'application de ϕ fait appel à l'instance g_a de g_h, où les paramètres formels h et r sont respectivement associés à a et à $((succ \circ \pi_2)\ b\ zero))$, ce qui permet à $out_{n,c}$ d'être appliqué sur cette dernière expression :

$$
\begin{array}{ll}
 & f\ (cons\ a\ (cons\ b\ (nil))) \\
= & ((\lambda h, r.(g_h\ r))\ a\ ((succ \circ \pi_2)\ b\ zero)) \\
= & g_a\ ((succ \circ \pi_2)\ b\ zero) \\
= & [\![\phi''_a, id, out_{n,c}]\!]_{L_A,L_A}\ ((succ \circ \pi_2)\ b\ zero) \\
= & \phi''_a \circ (id \circ \mathsf{L}_A g_a) \circ out_{n,c}\ ((succ \circ \pi_2)\ b\ zero) \\
= & \phi''_a \circ (id \circ \mathsf{L}_A g_a)(2, (b, zero)) \\
= & \phi''_a(2, (b, (g_a\ zero))) \\
= & \phi''_a(2, (b, [\![\phi''_a, id, out_{n,c}]\!]_{L_A,L_A}\ zero)) \\
= & \phi''_a(2, (b, (\phi''_a(1, ())))) \\
= & \phi''_a(2, (b, (((succ \circ \pi_2)\ a\ zero) \triangledown (succ \circ \pi_2)(1, ())))) \\
= & \phi''_a(2, (b, ((succ \circ \pi_2)\ a\ zero))) \\
= & (((succ \circ \pi_2)\ a\ zero) \triangledown (succ \circ \pi_2))\ (2, (b\ ((succ \circ \pi_2)\ a\ zero))) \\
= & (succ \circ \pi_2)\ b\ ((succ \circ \pi_2)\ a\ zero) \\
= & succ\ (succ\ zero)
\end{array}
$$

Cet hylomorphisme résultant de la fusion peut être traduit en sa définition récursive classique :

$$
\begin{array}{l}
length \circ rev\_naif = f \\
\qquad \text{où} \\
\quad f\ x = \texttt{case}\ x\ \texttt{of} \\
\qquad nil \rightarrow zero \\
\qquad cons\ head\ tail \rightarrow g\ head\ (f\ tail) \\
\qquad \text{avec} \\
\quad g\ h\ r = \texttt{case}\ r\ \texttt{of} \\
\qquad zero \rightarrow (succ\ \circ \pi_2)\ h\ zero \\
\qquad (succ \circ \pi_2)\ j\ k \rightarrow (succ \circ \pi_2)\ j\ (g\ h\ k)
\end{array}
$$

qui, sans être ce qu'il existe de plus simple pour calculer la longueur d'une liste, est tout de même une fonction qui ne construit plus la liste intermédiaire produite par le *rev_naif* de la composition initiale. Cette composition a donc été déforestée.

Concernant le problème du mauvais typage de out_{L_A} par rapport à son paramètre, relevé dans [TM95], il est clair que faire dépendre le σ des constructeurs n et c semble indispensable.

Cependant, ne pas s'autoriser à effectuer la simplification (♣), comme je l'ai fait dans l'exposé précédent, n'est pas acceptable[17]. En effet, il semble bien naturel de pouvoir remplacer dans la partie ϕ'' l'expression $((succ \circ \pi_2)\ h\ zero)$ par l'expression $(succ\ zero)$. Cependant, telle quelle, cette simplification ne permet pas d'avoir une définition cohérente. Il serait alors nécessaire de répercuter cette simplification à la fois dans le $out_{n,c}$, mais également dans le foncteur de l'hylomorphisme interne, qui ne devrait plus être L_A, mais un foncteur dépendant des constructeurs, $\mathsf{F}_{n,c}$.

La deuxième méthode consiste à considérer un σ **davantage paramétré.**

Considérons donc la définition de *rev_naif* suivante, dans laquelle les constituants du σ dépendent complètement des constructeurs qui permettent de l'instancier, n et c :

17. Françoise Bellegarde n'était pas convaincue de cette "solution" et m'a fait part de sa méfiance concernant l'interdiction d'une telle simplification.

$$\begin{array}{lcl} rev\_naif & : & (list\ A) \to (list\ A) \\ rev\_naif\ x & = & [\![\sigma in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A}\ x \\ \text{où} & & \\ & & \sigma = \lambda n_\triangledown c.(n_\triangledown(\lambda h, r.[\![(c\ h\ n)_\triangledown c, id, out_{n,c}]\!]_{F_{n,c},F_{n,c}}\ r)) \end{array}$$

Dans cette définition, l'abstracteur de l'hylomorphisme interne à σ, le $out_{n,c}$, peut être défini grâce aux constructeurs qui instancient le foncteur d'algèbre σ. Ainsi, lors de la fusion de *length* avec *rev_naif*, on obtient donc, comme précédemment :

$$length \circ rev\_naif = [\![\sigma(zero_\triangledown(succ \circ \pi_2)), id, out_{L_A}]\!]_{L_A,L_A}$$

Ici, les constructeurs considérés pour instancier n et c sont $zero$ et $(succ \circ \pi_2)$. Ces constructeurs peuvent permettre de définir un $out_{n,c}$ succeptible d'abstraire des données générées par ces constructeurs :

$$\begin{array}{l} out_{n,c}\ x = \texttt{case}\ x\ \texttt{of} \\ \quad zero \to (1, ()) \\ \quad (succ \circ \pi_2)\ j\ k \to (2, (j, k)) \end{array}$$

De même, au regard de ces constructeurs et de l'identité comme constituant central de l'hylomorphisme dans la définition du σ, les foncteurs $\mathsf{F}_{n,c}$ associés à cette instance de σ peuvent être déterminés ; je les appelle F :

$$\mathsf{F} = !1 + !A \times Id = \mathsf{L}_A$$

On obtient alors la définition suivante pour l'hylomorphisme résultant de la fusion de *length* et de *rev_naif* :

$$[\![zero_\triangledown(\lambda h, r.[\![((succ \circ \pi_2)\ h\ zero)_\triangledown(succ \circ \pi_2), id, out_{n,c}]\!]_{F,F}\ r), id, out_{L_A}]\!]_{L_A,L_A}$$

Intervient alors la possibilité de simplification de $((succ \circ \pi_2)\ h\ zero)$ en $(succ\ zero)$, déjà abordée plus haut (♣). Au niveau de l'hylomorphisme interne, cette simplification permet de restructurer l'hylomorphsime par HyloShift, comme suit :

$$\begin{array}{cl} & [\![((succ \circ \pi_2)\ h\ zero)_\triangledown(succ \circ \pi_2), id, out_{n,c}]\!]_{F,F} \\ & \quad \text{simplification } (\clubsuit) \\ = & [\![(succ\ zero)_\triangledown(succ \circ \pi_2), id, out_{n,c}]\!]_{F,F} \\ = & [\![((succ\ zero) \circ id)_\triangledown(succ \circ \pi_2), id, out_{n,c}]\!]_{F,F} \\ & \quad \text{propriétés des foncteurs} \\ = & [\![((succ\ zero)_\triangledown succ) \circ (id + \pi_2), id, out_{n,c}]\!]_{F,F} \\ & \quad \text{HyloShift (théorème 5.5.1)} \\ = & [\![((succ\ zero)_\triangledown succ), (id + \pi_2), out_{n,c}]\!]_{G,F} \end{array}$$

Dans ce cas, l'hylomorphisme transforme désormais une F-structure en une G-structure, et la transformation naturelle déplacée dans la partie centrale de l'hylomorphisme permet de définir :

$$\mathsf{G} = !1 + Id = \mathsf{N}$$

Cependant, la simplification (♣) de la première partie de l'hylomorphisme, couplée à l'application d'Hylo-Shift, nécessite une reformulation du $out_{n,c}$. En effet, le constructeur portant le *tag* 2, fourni par $out_{n,c}$, doit toujours avoir deux paramètres pour être une F-structure. La reformulation de la partie gauche de l'hylomorphisme a en fait "oublié" un paramètre inutile (le h), mais il doit être présent, par respect du foncteur F, dans le résultat du $out_{n,c}$.

D'où la modification, du $out_{n,c}$:

$$\begin{array}{l} out_{n,c}\ x = \texttt{case}\ x\ \texttt{of} \\ \quad zero \rightarrow (1,()) \\ \quad (succ \circ \pi_2)\ j\ k \rightarrow (2,(j,k)) \end{array}$$

qui doit devenir :

$$\begin{array}{l} out2_{n,c}\ x = \texttt{case}\ x\ \texttt{of} \\ \quad zero \rightarrow (1,()) \\ \quad succ\ k \rightarrow (2,(blob,k)) \end{array}$$

En supposant qu'il est possible d'inférer toutes ces modifications, on obtiendrait alors finalement une définition de la composition initiale sous la forme :

$$\begin{array}{ll} & length \circ rev\_naif = f = [\![\phi,\eta,\psi]\!]_{L_A,L_A} \\ \text{où} & \phi = zero_\triangledown (\lambda h,r.(g\ r)) \\ \text{avec} & \quad g = [\![\phi'',\eta'',\psi'']\!]_{G,F} \\ & \qquad \phi'' = (succ\ zero)_\triangledown succ \\ & \qquad \eta'' = (id + \pi_2) \\ & \qquad \psi'' = out2_{n,c} \\ & \eta = id \\ & \psi = out_{L_A} \\ & \mathsf{L}_A = \mathsf{F} = !1 + !A \times Id \\ & \mathsf{N} = \mathsf{G} = !1 + Id \end{array}$$

En reprennant alors l'exécution qui posait problème plus haut avec cette nouvelle définition :

$$\begin{array}{ll} & f\ (cons\ a\ (cons\ b\ (nil))) \\ = & \phi(2,(a,(\phi(2,(b,(\phi(1,()))))))) \\ = & \phi(2,(a,(\phi(2,(b,zero))))) \\ = & \phi(2,(a,((\lambda h,r.(g\ r))\ (b,zero)))) \\ = & \phi(2,(a,(g\ zero))) \\ = & \phi(2,(a,[\![((succ\ zero)_\triangledown succ),\eta'',out2_{n,c}]\!]_{G,F}\ zero)) \end{array}$$

L'application de ϕ sur $(1,())$ a généré directement $zero$, mais l'application de ϕ sur $(2,(b,zero))$ fait appel à g, où les paramètres formels h et r sont respectivement associés à b et à $zero$. Le b, correspondant au paramètre formel h, est donc "oublié" ; $out2_{n,c}$ peut alors être appliqué à $zero$:

$$\begin{array}{ll} & f\ (cons\ a\ (cons\ b\ (nil))) \\ = & \phi(2,(a,(((succ\ zero)_\triangledown succ)\ (1,())))) \\ = & \phi(2,(a,(succ\ zero))) \end{array}$$

Récursivement, l'application de ϕ fait appel à g, où les paramètres formels h et r sont respectivement associés à a et à $((succ\ zero))$; a étant "oublié", $out2_{n,c}$ va être appliqué sur cette dernière expression en introduisant le parmètre *blob* qui ne sera pas utilisé, mais qui permet de fournir à l'application récursive le bon nombre de paramètres et ainsi de compenser l'oubli du paramètre dû à la simplification :

$$
\begin{array}{cl}
 & f\ (cons\ a\ (cons\ b\ (nil))) \\
= & (\lambda h, r.(g\ r))\ a\ (succ\ zero) \\
= & g\ (succ\ zero) \\
= & [\![\phi'', \eta'', out2_{n,c}]\!]_{G,F}\ (succ\ zero) \\
= & \phi'' \circ ((id + \pi_2) \circ \mathsf{F}\ g) \circ out2_{n,c}(succ\ zero) \\
= & \phi'' \circ (id + (g \circ \pi_2))(2, (blob, zero)) \\
= & \phi''(2, (g\ zero)) \\
= & \ldots \\
= & \phi''(2, (succ\ zero)) \\
= & ((succ\ zero) \triangledown succ)\ (2, (succ\ zero)) \\
= & succ\ (succ\ zero)
\end{array}
$$

Cet hylomorphisme résultant de la fusion peut être traduit en sa définition récursive classique :

$$
\begin{array}{l}
length \circ rev\_naif = f \\
\qquad \text{où} \\
\quad f\ x = \texttt{case}\ x\ \texttt{of} \\
\qquad nil \rightarrow zero \\
\qquad cons\ head\ tail \rightarrow g\ head\ (f\ tail) \\
\qquad\quad \text{avec} \\
\quad g\ h\ r = \texttt{case}\ r\ \texttt{of} \\
\qquad zero \rightarrow succ\ zero \\
\qquad succ\ k \rightarrow succ\ (g\ h\ k)
\end{array}
$$

qui est une version déforestée de la composition initiale, puisqu'elle ne contient plus de construction de structure de liste.

Je viens de présenter deux manières légèrement différentes de contourner le problème de typage relevé dans [TM95]. Cependant, l'une comme l'autre n'est pas complètement satisfaisante.

Dans le premier cas, la conservation *in extenso* des deux constructeurs $((succ \circ \pi_2)\ h\ zero)$ et $(succ \circ \pi_2)$ dans la partie ϕ'' de l'hylomorpisme interne permet de conserver une version de $out_{n,c}$ assez simplement déterminable : cette méthode permet d'obtenir une version correcte de la fusion des deux hylomorphismes. En revanche, cette première méthode nécessite de ne pas simplifier le $((succ \circ \pi_2)\ h\ zero)$ en $(succ\ zero)$ (♣), ce qui est difficile à justifier.

Dans le second cas, plus général au sens où le paramètrage du σ est plus complet, cette simplification nécessite de nombreuses répercussions sur les différents objets constituant l'hylomorphisme, ainsi que sa restructuration. Même si les modifications que j'ai proposé aboutissent à une version correcte de l'hylomorphisme résultant de la composition, qui est bien déforesté, il semble difficile d'imaginer comment elle peut être obtenue automatiquement ; en particulier, l'ajout du paramètre *blob* dans le $out2_{n,c}$ n'est clairement pas satisfaisant et, en tout cas, semble difficile à automatiser.

Remarquons pour finir que, dans l'exemple précédent ($length \circ append$), le problème ne se posait pas car la substitution

$$\tau = \lambda n \triangledown c.([\![n \triangledown c, id, out_{L_A}]\!]_{L_A, L_A}\ x_2)\ \triangledown\ c$$

s'appliquait sur un paramètre (x_2) qui était un paramètre constant et non une variable d'accumulation de résultat comme dans le cas du σ posant problème. Reste que ce sont des problèmes qui n'ont, à ma connaissance, jamais été évoqués jusqu'à présent dans la littérature relative à ces méthodes de fusion.

Des limitations

L'exemple suivant est une illustration de ce qu'il est possible d'obtenir avec les transformations de déforestation par fusion d'hylomorphismes. J'ai montré plus haut comment représenter la fonction *rev_naif* d'inversion d'une liste. J'ai également montré comment une composition telle que $length \circ rev\_naif$ pouvait être déforestée si elle était représentée par des hylomorphismes. Je m'intéresse maintenant à la fonction d'inversion *reverse* définie grâce à la fonction *rev*, utilisant un paramètre d'accumulation : $reverse\ x = rev\ x\ nil$

L'algorithme de dérivation d'hylomorphismes à partir de définitions récursives fourni par Hu *et al.* [HIT96b, OHIT97] produit automatiquement pour cette fonction l'hylomorphisme en forme de triplet suivant [18] :

$$\begin{array}{lcl}
reverse & : & (list\ A) \to (list\ A) \\
reverse\ x & = & rev\ x\ nil \\
rev & : & (list\ A) \times (list\ A) \to (list\ A) \\
rev\ x\ l & = & [\![\phi, id, \psi]\!]_{F,F}\ x\ l \\
\text{où} & & \\
 & & \phi = id \triangledown id \\
 & & \psi = \lambda x, l.\mathtt{case}\ (x, l)\ \mathtt{of} \\
 & & \quad (nil, l) \to (1, (l)) \\
 & & \quad ((cons\ head\ tail), l) \to (2, (tail, (cons\ head\ l))) \\
\text{avec} & & \\
 & & \mathsf{F} = !(list\ A) + Id
\end{array}$$

Il est important de remarquer ici que le foncteur F n'est pas le foncteur des listes. Il représente le schéma récursif de la fonction et la construction de son résultat dans le paramètre d'accumulation ; il est donc distinct du schéma de récursion du type de donnée manipulé. Cette particularité pose des problèmes. Par exemple, il n'est plus possible, dans le cadre de la composition $length \circ reverse$, d'appliquer l'*Acid Rain Theorem* (Cata-HyloFusion), puisque la différence de foncteur ($\mathsf{L}_A \neq \mathsf{F}$) et la forme $id \triangledown id$ de la partie ϕ de l'hylomorphisme pour *reverse* ne permet pas de remplir les hypothèses de ce théorème.

Un des avantages des hylomorphismes sous la forme de triplets est de pouvoir appliquer les règles du théorème *Acid Rain*, qui sont plus aisées à mettre en œuvre automatiquement. La

18. Cette forme correspondant à la définition n'est pas nécessairement la seule possible, mais c'est celle qui est fournie par l'algorithme de dérivation [HIT96b, OHIT97] qui, à ma connaissance, est le seul existant à lors actuelle.

dérivation automatique proposée dans [OHIT97] de la fonction *reverse* produit un foncteur F et un hylomorphisme qui ne permettent pas l'application des règles simples : c'est une première limitation de la technique.

Par contre, en se plaçant dans le cadre plus général de HyloFusion (à gauche), on peut tout de même fusionner *length*, en tant que fonction, avec *rev*, en tant qu'hylomorphisme :

$$\frac{length \circ \phi = \phi' \circ \mathsf{F}\ length}{length \circ [\![\phi, \eta, \psi]\!]_{F,F} = [\![\phi', \eta, \psi]\!]_{F,F}}$$

Reste alors à déterminer ϕ' :

$$\begin{array}{rcll} length \circ \phi & = & length \circ (id \triangledown id) & \textit{par définition de } \phi \\ & = & length \triangledown length & \textit{propriétés des foncteurs} \\ & = & \phi' \circ \mathsf{F}\ length & \textit{hypothèse d'HyloFusion} \\ & = & \phi' \circ (id + length) & \textit{puisque } \mathsf{F}\ f = id + f \\ & = & (length \triangledown id) \circ (id + length) & \textit{par déduction} \\ \textit{d'où} & & \phi' = (length \triangledown id) & \end{array}$$

On obtient donc pour la fusion de *length* avec *reverse* l'hylomorphisme suivant :

$$\begin{array}{lcl} (length \circ reverse) & : & (list\ A) \to nat \\ (length \circ reverse)\ x & = & (length \circ rev)\ x\ nil \\ (length \circ rev) & : & (list\ A) \times (list\ A) \to nat \\ (length \circ rev)\ x\ l & = & [\![length \triangledown id, id, \psi]\!]_{F,F}\ x\ l \end{array}$$

où

$$\begin{array}{l} \psi = \lambda x, l.\mathtt{case}\ (x, l)\ \mathtt{of} \\ \quad (nil, l) \to (1, (l)) \\ \quad ((cons\ head\ tail), l) \to (2, (tail, (cons\ head\ l))) \end{array}$$

avec

$$\mathsf{F} = !(list\ A) + Id$$

Le problème, avec cette version fusionnée de $length \circ reverse$, est qu'elle n'est pas déforestée. En effet, l'accumulation dans le second paramètre de ψ n'est pas *visible* par le foncteur F, et donc n'est pas atteint par le processus de déforestation. Au final, on n'a plus qu'un seul hylomorphisme, qui calcule la longueur de la liste inversée, *mais après l'avoir construite* !

Je déroule l'effet de cet hylomorphisme, noté f, sur un petit exemple. L'application récursive sur $(cons\ a\ (cons\ b\ nil))$ et nil donne :

$$\begin{array}{cl} & f\ (cons\ a\ (cons\ b\ nil))nil \\ = & [\![\phi', \eta, \psi]\!]_{F,F}\ (cons\ a\ (cons\ b\ nil))nil \\ = & \phi' \circ (\eta \circ \mathsf{F}f) \circ \psi\ (cons\ a\ (cons\ b\ nil))\ nil \\ = & \phi' \circ (\eta \circ \mathsf{F}f)\ (2, ((cons\ b\ nil), (cons\ a\ nil))) \\ = & \phi'\ (2, (f\ (cons\ b\ nil)\ (cons\ a\ nil))) \\ = & \ldots \\ = & \phi'\ (2, (\phi'\ (2, (\phi'\ (1, (cons\ b\ (cons\ a\ nil))))))) \\ = & \phi'\ (2, (\phi'\ (2, (length \triangledown id(1, (cons\ b\ (cons\ a\ nil))))))) \\ = & \phi'\ (2, (\phi'\ (2, (length(cons\ b\ (cons\ a\ nil)))))) \\ = & (id\ (id\ (length\ (cons\ b\ (cons\ a\ nil))))) \end{array}$$

Il y a donc bien eu fusion, mais pas déforestation, puisque la construction de la liste intermédiaire n'a pas été éliminée. Le problème est exactement le même si on tente de fusionner *reverse* avec lui même. On arrive à fusionner, mais sans déforester. Reste alors, comme pour l'exemple précédent, à prendre la définition naïve de l'inversion de liste, pour tenter de déforester $rev\_naif \circ rev\_naif$.

$$\begin{array}{lcl} rev\_naif & : & (list\ A) \to (list\ A) \\ rev\_naif\ x & = & [\![\sigma in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A}\ x \\ \text{où} & & \\ & & \sigma = \lambda n_\triangledown c.(n_\triangledown(\lambda h, r.[\![(c\ h\ n)_\triangledown c, id, out_{L_A}]\!]_{L_A,L_A}\ r)) \end{array}$$

Dans ce cas, le théorème *Acid Rain* voit ses hypothèses vérifiées et il est donc appliqué comme suit :

$$\begin{array}{l} rev\_naif \circ rev\_naif \\ \qquad \text{par définitions en hylomorphismes} \\ = [\![\sigma in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A} \circ [\![\sigma in_{L_A}, id, out_{L_A}]\!]_{L_A,L_A} \\ \qquad \text{par Cata-HyloFusion} \\ = [\![\sigma(\sigma(in_{L_A} \circ id)), id, out_{L_A}]\!]_{L_A,L_A} \\ \qquad \text{par définition de } in_{L_A} \text{ et propriétés des foncteurs} \\ = [\![\sigma(\sigma in_{L_A}), id, out_{L_A}]\!]_{L_A,L_A} \end{array}$$

Cependant, le nombre de constructeurs *cons* utilisés pour inverser deux fois une liste de longueur n est $\frac{n\ (n+1)}{2}$, soit une complexité de $\mathcal{O}(n^2)$ (il s'agit en fait de la complexité intrinsèque de la fonction *rev_naif*), alors qu'on espère une fonction proche de la recopie de liste ... Cet exemple est assez caricatural, mais il représente une classe de programmes pour lesquels l'ensemble des méthodes de déforestation fonctionnelles existantes ne sont pas efficaces : cela constitue une deuxième limitation de la technique.

Finalement, je considère la fonction *flat* qui construit la liste des feuilles d'un arbre binaire.

$$\begin{array}{l} \texttt{let}\ \mathit{flat}\ t\ l = \texttt{case}\ t\ \texttt{of} \\ \quad \mathit{leaf}\ n \to \mathit{cons}\ n\ l \\ \quad \mathit{node}\ \mathit{left}\ \mathit{right} \to \\ \qquad \mathit{flat}\ \mathit{left}\ (\mathit{flat}\ \mathit{right}\ l) \end{array}$$

L'hylomorphisme correspondant à cette fonction, dérivé par l'algorithme présenté à la section 5.5.3, est le suivant :

$$\begin{array}{lcl} \mathit{flat}\ t\ l & = & [\![\phi', id, \psi']\!]_{G,G}\ t\ l \\ \text{où} & & \\ & & \phi' = \mathit{cons}_\triangledown \mathit{id} \\ & & \psi = \lambda t, l.\texttt{case}\ (t, l)\ \texttt{of} \\ & & \qquad ((\mathit{leaf}\ n), l) \to (\mathit{1}, (n, l)) \\ & & \qquad ((\mathit{node}\ \mathit{left}\ \mathit{right}), l) \to (\mathit{2}, (\mathit{left}, (\mathit{flat}\ \mathit{right}\ l))) \\ \text{avec} & & \\ & & \mathsf{G} = !A \times !(list\ A) + Id \end{array}$$

Si je considère la composition de *rev* avec *flat*, afin d'obtenir la liste inversée des feuilles d'un arbre binaire t, le résultat optimal attendu serait l'élimination de la liste intermédiaire, produite par *flat* et consommée par *rev*. L'incompatibilité entre les formes des hylomorphismes et les foncteurs F et G pose alors la troisième limitation du système de fusion d'hylomorphismes : à ma connaissance, il ne permet pas de traiter la fusion et actuellement, aucune déforestation automatique ne peut donc être effectuée.

5.6 Conclusion et ouverture

Sans être exhaustif sur toutes les méthodes de déforestations existant en programmation fonctionnelle, ce chapitre permet d'avoir une vue assez complète des techniques utilisées, d'une part, dans l'algorithme de déforestation de Wadler et, d'autre part, dans les deux principales approches *calculationnelles* de la déforestation que sont la normalisation des *folds* et la fusion des hylomorphismes.

Les différentes comparaisons qui ont pu être faites entre ces méthodes font principalement ressortir la nécessité, pour faciliter la déforestation, de décrire un programme par une spécification dirigée par la structure des données.

Dans l'algorithme de Wadler, le guidage de la déforestation est fait syntaxiquement, sur les mots clé du langage (e.g. sur le `case`), tandis que les méthodes calculationnelles font appel à une abstraction des schémas récursifs des types de données et des fonctions, qui est représentée par la notion de foncteur. Cette technique d'abstraction permet de traiter les problèmes de manière formelle, grâce à des résultats de la théorie des catégories et permet également d'automatiser les transformation de programmes. Cependant, la notion de foncteur possède une certaine rigidité, puisqu'elle est déterminée à partir des schémas de récursions globaux des types de données et des fonctions. Elle ne permet pas toujours de suivre fidèlement les constructions de structures intermédiaires pouvant être “encapsulées” dans des variables d'accumulations. Cette abstraction des foncteurs, parfois trop éloignée des constructions de structures intermédiaires, peut engendrer des difficultés lors du processus de déforestation et ne pas les éliminer complètement.

Dans le prochain chapitre, je vais comparer la composition descriptionnelle, technique de déforestation dédiée aux grammaires attribuées (cf. section 3), aux méthodes de déforestations calculationnelles qui viennent d'être présentées. Cette comparaison tend à montrer que les spécifications de grammaires attribuées, déclaratives, dirigées par la structure des données et indépendantes de l'ordre ou du modèle d'exécution, sont particulièrement adaptées aux transformations de programmes telles que la déforestation et peuvent apporter une réponse aux problèmes restés irrésolus par les techniques présentées dans ce dernier chapitre.

Chapitre 6

Déforestation fonctionnelle *versus* composition descriptionnelle

Dans le contexte des transformations de programmes, et plus particulièrement quand la structure des données est une information primordiale comme c'est le cas pour la déforestation, le formalisme des grammaires attribuées possède des atouts indéniables et se révèle très puissant.

Afin de mettre en évidence les caractéristiques intéressantes du formalisme des grammaires attribuées et les similitudes qu'il possède avec la programmation fonctionnelle, en termes tant de pouvoir d'expression que de transformation de programmes, je compare dans ce chapitre les notations, l'expressivité, la sémantique et l'évaluation des les grammaires attribuées avec celles des *folds* et des hylomorphismes, qui sont des styles de programmation fonctionnelle dédiés aux transformations de programmes. Leur point commun évident est de rendre la structure des données explicite dans les programmes et de s'appuyer sur cette structure pour décrire des calculs, des parcours ou des transformations.

Les *folds* et les grammaires attribuées seront donc comparés en tant que paradigmes de programmation dirigés par la structure des données. La comparaison sera tout d'abord limitée aux folds du premier ordre qui sont équivalents aux grammaires attribuées purement synthétisées. Après la comparaison des notations et des pouvoirs d'expressions, je comparerai les effets respectifs des méthodes de déforestation sur chacun de ces deux formalismes.

Cette comparaison sera ensuite étendue aux *folds* du second ordre qui permettent d'exprimer des fonctions plus complexes et que les grammaires attribuées représentent généralement à l'aide d'attributs hérités. Je comparerai ces deux représentations et la déforestation qu'elles autorisent dans leurs domaines respectifs. Cette comparaison permettra ensuite de montrer les limitations du formalisme des *folds* et de son algorithme de déforestation associé.

Je présenterai alors les atouts du formalisme des hylomorphismes et les apports de leur méthode de fusion. Introduits pour palier certaines des limitations des *folds*, ce formalisme et ces techniques apparaîtront alors comme étonnamment proches des grammaires attribuées, dans leur principe et dans leurs effets.

Finalement, les constatations de limitations, à la fois des méthodes de déforestation et des moyens permettant de les comparer, me donneront l'occasion de motiver les travaux qui sont décrits par la suite.

$$
\begin{array}{lcl}
length\ x & = & (fold^{list} \\
 & & \quad \clubsuit \quad (\lambda().zero) \\
 & & \quad \spadesuit \quad (\lambda e, r.succ\ r) \ \ )\ x
\end{array}
$$

FIG. 6.1: *Définition de length avec un fold*

$$
\begin{array}{lcl}
append\ x\ y & = & (fold^{list} \\
 & & \quad \diamondsuit \quad (\lambda().y) \\
 & & \quad \heartsuit \quad (\lambda e, r.cons\ e\ r) \ \ )\ x
\end{array}
$$

FIG. 6.2: *Définition de append avec un fold*

6.1 *Folds* du premier ordre et grammaires attribuées

6.1.1 Similarités de notations

À l'aide de l'opérateur de contrôle générique *fold*, il est possible de décrire des programmes comme cela est présenté à la section 5.3. Sur le type *list*, les fonctions calculant la longueur d'une liste (*length*, figure 6.1) et la concaténation de deux listes (*append*, figure 6.2) peuvent être exprimées assez simplement à l'aide de *folds* [SF93].

Les deux lambda-expressions (♣ et ♠) sont les *fonctions accumulatives* du *fold* représentant la fonction *length* (figure 6.1). Ces fonctions accumulatives représentent les calculs à effectuer sur chacun des constructeurs du type *list* : l'une, pour le constructeur *nil*, n'a aucun paramètre ; l'autre, pour le constructeur *cons*, accepte deux paramètres déterminés par la définition générique du fold^{list} (cf. section 5.3.3).

Sur le type *list*, les grammaires attribuées permettent également de spécifier la fonction *length*. La figure 6.3 présente une grammaire attribuée spécifiant *length* dans le formalisme fonctionnel présenté section 2.1.2. Dans cette grammaire attribuée, les règles sémantiques ♣ et ♠ pour les constructeurs *nil* et *cons* sont aisément représentables sous la forme de lambda-expressions et correspondent exactement aux fonctions accumulatives de résultat du programme représentant la fonction *length* avec un *fold*,(figure 6.1). Il en est de même pour la grammaire attribuée spécifiant *append*, présentée à la figure 6.4.

La différence principale entre ces deux syntaxes (*fold* et grammaire attribuée) est que chaque variable utilisée dans une grammaire attribuée est explicitement mentionnée par un nom spécifique : c'est une occurrence d'attribut. Par exemple, dans le *pattern* (*cons head tail*) de la figure 6.3, le résultat attendu sur *tail* est explicitement nommé *tail.s* tandis que dans

$$
\begin{array}{lcl}
\texttt{aglet}\ length = & & \\
\quad length\ x \rightarrow & & \\
\quad\quad length.result = x.s & & \\
\quad nil \rightarrow & & \\
\quad\quad nil.s = zero & \clubsuit & nil.s = (\lambda().zero) \\
\quad cons\ head\ tail \rightarrow & & \\
\quad\quad cons.s = succ\ tail.s & \spadesuit & cons.s = (\lambda e, r.succ\ r)\ head\ tail.s
\end{array}
$$

FIG. 6.3: *Définition de length en grammaire attribuée*

$$
\begin{array}{lll}
\texttt{aglet}\ append = & & \\
\quad append\ x\ y \rightarrow & & \\
\qquad append.result = x.v & & \\
\quad nil \rightarrow & & \\
\qquad nil.v = y & \diamondsuit & nil.v = (\lambda().y) \\
\quad cons\ head\ tail \rightarrow & & \\
\qquad cons.v = cons\ head\ tail.v & \heartsuit & cons.v = (\lambda e, r.cons\ e\ r)\ head\ tail.v
\end{array}
$$

FIG. 6.4: *Définition de append en grammaire attribuée*

$$
\begin{array}{rcl}
flip\ t & = & (fold^{tree} \\
& & \quad \bullet \quad (\lambda(i).leaf\ i) \\
& & \quad \star \quad (\lambda p, q.node\ q\ p) \quad)\ t
\end{array}
$$

FIG. 6.5: *Définition de flip avec un fold*

la version en *fold* ce résultat est implicitement représenté par r, qui est une variable accumulative de résultat. Cette notation explicite en grammaire attribuée permet d'exprimer des fonctions qui retournent plus d'un résultat (plusieurs attributs synthétisés) et qui possèdent plus d'un paramètre (plusieurs attributs hérités). Dans cette section, je me contenterai de comparer les grammaires attribuées purement synthétisées, qui n'utilisent qu'un seul attribut synthétisé, avec les *folds* du premier ordre. Cependant, il est d'ores et déjà intéressant de remarquer que les règles sémantiques de la forme en grammaire attribuée sont exactement les fonctions accumulatives de la forme en *fold*, appliquées aux paramètres explicites, c'est-à-dire aux occurrences d'attributs appropriées.

Remarquons également que, dans l'exemple de *append*, le paramètre y est une variable libre qui joue le rôle d'une *variable globale*, dans le *fold* comme dans la grammaire attribuée.

Pour prendre un exemple sur un autre type, la fonction *flip*, qui prend en entrée un arbre binaire de type *tree* et qui échange toutes ses branches, peut être représentée par un *fold*, comme le présente la figure 6.5, ou par une grammaire attribuée, comme dans la figure 6.6. Les fonctions accumulatives du *fold* ($\bullet$ et $\star$) sont là aussi des λ-abstractions des règles sémantiques de la grammaire attribuée, où les occurrences d'attributs ont été abstraites par des paramètres implicites : les variables accumulatives de résultat i, p et q.

$$
\begin{array}{lll}
\texttt{aglet}\ flip = & & \\
\quad flip\ t \rightarrow & & \\
\qquad flip.result = t.m & & \\
\quad leaf\ elem \rightarrow & & \\
\qquad leaf.m = leaf\ elem & \bullet & leaf.m = (\lambda(i).leaf\ i)\ elem \\
\quad node\ left\ right \rightarrow & & \\
\qquad node.m = node\ right.m\ left.m & \star & node.m = \\
& & \quad (\lambda p, q.node\ q\ p)\ left.m\ right.m
\end{array}
$$

FIG. 6.6: *Définition de flip en grammaire attribuée*

6.1.2 Foncteurs et évaluateurs d'attributs

Après la comparaison des notations, je m'intéresse ici à la sémantique et à l'évaluation des *folds* et des grammaires attribuées. La sémantique d'une fonction exprimée avec un *fold* est donnée par la définition de cet opérateur, qui utilise la notion de *foncteur* [SF93]. Sans revenir sur le détail du calcul de ces foncteurs (cf. section 5.3.3), qui sont déterminés de façon statique à partir d'un type algébrique, je rappelle ci-dessous les foncteurs générés à partir des types *list* et *tree* et les équations définissant l'opérateur *fold* pour ces types.

$$\begin{array}{lcl} E^{list}_{nil}\ f\ g & = & \lambda().() \\ E^{list}_{cons}\ f\ g & = & \lambda x, y.(fx)\ (gy) \end{array}$$

$$\begin{array}{lcl} (fold^{list}\ f_n\ f_c)\ nil & = & f_n \\ (fold^{list}\ f_n\ f_c)\ (cons\ head\ tail) & = & f_c\ head\ ((fold^{list}\ f_n\ f_c)\ tail) \end{array}$$

$$\begin{array}{lcl} E^{tree}_{leaf}\ f\ g & = & \lambda x.fx \\ E^{tree}_{node}\ f\ g & = & \lambda x, y.(gx)\ (gy) \end{array}$$

$$\begin{array}{lcl} (fold^{tree}\ f_l\ f_n)\ (leaf\ elem) & = & f_l\ elem \\ (fold^{tree}\ f_l\ f_n)\ (node\ left\ right) & = & f_n\ ((fold^{tree}\ f_l\ f_n)\ left)\ ((fold^{tree}\ f_l\ f_n)\ right) \end{array}$$

Ces définitions permettent d'évaluer n'importe quelle fonction écrite à l'aide de *fold* sur le type des listes ou des arbres binaires.

Par exemple, la définition de $fold^{list}$ permet d'évaluer la fonction *length* donnée à la figure 6.1, puisqu'elle permet d'identifier les paramètres de f_c (c'est-à-dire e et r). Le rôle du foncteur est donc de déterminer les paramètres à fournir aux fonctions accumulatives. Pour le constructeur *cons head tail*, la variable accumulative de résultat est définie récursivement sur *tail* (le reste de la liste). L'opérateur $fold^{list}$ est défini avec des foncteurs qui sont déterminés statiquement à partir du type *list*. Ces foncteurs fournissent un sens, une sémantique, à l'évaluation des fonctions exprimées avec un $fold^{list}$.

Plus généralement, les foncteurs E_i (cf. section 5.3.2) définissent la façon de calculer récursivement les valeurs des fonctions accumulatives f_i sur les constructeurs c_i d'une structure récursive. Il est important de noter ici que la définition d'un ensemble de foncteurs dépend uniquement du type considéré. Ces foncteurs sont définis statiquement à partir des constructeurs du type et ils sont valides pour n'importe quelle fonction accumulative de n'importe quel programme défini sur ce type : le *fold* est un opérateur de contrôle *générique*, défini automatiquement et définitivement pour un type donné.

Pour une grammaire attribuée, la sémantique est donnée par la solution du système d'équations orientées défini par l'ensemble des règles sémantiques entre les instances d'attributs, pour une structure de donnée d'entrée particulière et ceci quelle que soit la méthode utilisée pour résoudre ce système d'équations.

La forme des règles sémantiques (équations orientées) permet de déterminer des dépendances entre les occurrences d'attributs sur les productions (*patterns*). C'est à partir de ces dépendances que sera déterminé l'ordre d'évaluation de toutes les instances d'occurrences d'attributs pour un paramètre d'entrée donné, parmi lesquelles la valeur de l'attribut synthétisé à la racine de la structure (*result*) représente le résultat de la fonction.

$$\begin{array}{l}
\texttt{aglet}\ (\mathit{fold}\ f_n\ f_c) = \\
\quad (\mathit{fold}\ f_n\ f_c)\ x \rightarrow \\
\quad\quad \mathit{fold.result} = x.s \\
\quad \mathit{nil} \rightarrow \\
\quad\quad \mathit{nil.s} = f_n \\
\quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \\
\quad\quad \mathit{cons.s} = f_c\ \mathit{head}\ \mathit{tail.s}
\end{array}$$

FIG. 6.7: *Définition de* $fold^{list}$ *en grammaire attribuée* S^1 *générique*

En d'autres termes, pour une spécification (grammaire attribuée) donnée, différentes techniques [Eng84] peuvent être employées pour générer un évaluateur d'attributs, utilisant éventuellement différentes méthodes d'évaluation, mais conduisant toujours à la même solution. Ainsi, la sémantique d'une grammaire attribuée repose uniquement sur sa spécification (la signature de ses règles sémantiques) et elle est indépendante de la méthode – et éventuellement de l'ordre – d'évaluation. Du point de vue des grammaires attribuées, les foncteurs associés aux constructeurs d'un type peuvent donc être considérés comme une méthode d'évaluation particulière, figée par le type.

Pour l'exemple de *length* (figure 6.3), la règle sémantique $cons.s = succ\ tail.s$ associée au constructeur *cons* détermine la récursion implicite nécessitant le calcul de l'attribut s sur le reste de la liste (*tail*) avant de pouvoir calculer sa valeur sur un *cons* donné. La règle sémantique $nil.s = zero$ donne la condition de terminaison de cette récursion, puisque aucun attribut n'apparaît en partie droite de cette équation.

De même, pour la grammaire attribuée définissant *flip* (figure 6.6), la règle sémantique associée au *pattern* (*node left right*) fournit simplement l'information suivante : l'attribut m d'un nœud donné dépend des attributs m calculés sur chacune de ses branches.

Le cadre d'étude est ici celui de fonctions assez simples, c'est-à-dire les fonctions qui peuvent s'exprimer par des grammaires attribuées ayant un seul attribut synthétisé et notées S^1. Pour ces fonctions, l'ordre d'évaluation fourni par les foncteurs qui servent à la définitions des *folds* pour un type donné est valide pour l'évaluation des grammaires attribuées correspondantes. Autrement dit, ces foncteurs constituent des évaluateurs valides pour les grammaires attribuées [Jou84a].

Il est donc possible de définir l'opérateur $fold^{list}\ f_n\ f_c$ dans le formalisme des grammaires attribuées, comme dans la figure 6.7, et de considérer qu'elles possèdent le même pouvoir d'expression.

6.1.3 Effets de déforestation

En restant dans le cadre des *folds* du premier ordre et des grammaires attribuées purement synthétisées, cette section compare maintenant leurs méthodes de déforestation : l'algorithme de normalisation (cf. section 5.3.4) pour les *folds* de premier ordre et la composition descriptionnelle (cf. chapitre 3) pour les grammaires attribuées S^1. Le fonctionnement de l'algorithme de normalisation a déjà été détaillé sur l'exemple *length* (*append x y*), mais j'en redonne ici les grandes lignes ; son principe et ses effets seront comparés sur cette composition simple avec ceux de la composition descriptionnelle. Malgré des moyens (algorithmes) *a priori* différents, cette comparaison aboutit à des résultats similaires.

Algorithme de normalisation

Rappelons que l'algorithme de normalisation est composé de trois parties : la *généralisation* qui est un remplacement (substitution) de terme, l'*application à une construction* qui est l'application de la définition de l'opérateur *fold* à un constructeur et à ses paramètres et la *promotion* qui établit que la composition d'une fonction g avec une fonction exprimée en *fold* est une nouvelle fonction exprimée en *fold*, dont les fonctions accumulatives doivent pouvoir être déterminées comme solutions d'équations. Cette dernière partie est basée sur le théorème de promotion (théorème 5.3.1 section 5.3.4) :

$$\frac{\begin{array}{lcl}\phi_n & = & g\ f_n \\ \phi_c\ a\ (g\ r) & = & g\ (f_c\ a\ r)\end{array}}{g\ ((fold^{list}\ f_n\ f_c)\ x) = (fold^{list}\ \phi_n\ \phi_c)\ x}$$

Le théorème de promotion assure la validité du *fold* résultant pour des fonctions ϕ construites localement sur chaque constructeur, c'est-à-dire que chaque fonction accumulative ϕ_i du résultat ne dépend que de la fonction g et de la fonction accumulative f_i du *fold* originel. Dans le cas des listes par exemple, il doit déterminer le ϕ_c, c'est-à-dire trouver une solution de l'équation $\phi_c\ a\ (g\ r) = g\ (f_c\ a\ r)$. Je vais montrer que la composition descriptionnelle est une transformation qui possède le même caractère de *localité* que le théorème de promotion du fold.

Dans l'exemple de *length* $\circ$ *append*, le rôle du théorème de promotion est de donner une définition pour les fonctions accumulatives ϕ_i (où plus précisément de donner des équations qu'elles doivent vérifier) et de prouver la correction du *fold* résultant. Il permet donc de déterminer de nouvelles fonctions accumulatives sur lesquelles il sera possible d'appliquer les pas d'*application à une construction* et de *généralisation*. Ce sont ces deux derniers pas qui effectuent véritablement la déforestation.

Plus précisément, la fonction g (*length* pour notre exemple) se retrouve à l'intérieur des fonctions accumulatives ϕ_i du résultat et est donc appliquée directement sur les f_i originales (c'est-à-dire sur leurs constructeurs). Cependant, la véritable déforestation n'est effectuée sur ces nouvelles fonctions ϕ_i que par les étapes d'*application à une construction* et de *généralisation*.

Voilà ce que cela donne sur l'exemple *length* $\circ$ *append* (plus de détails sont fournis section 5.3.4). Rappelons d'abord les définitions :

$$\begin{array}{lcl} length\ x & = & (fold^{list}\ (\lambda().zero)\ (\lambda e, r.succ\ r))\ x \\ append\ x\ y & = & (fold^{list}\ f_n\ f_c)\ x \\ & \text{avec} & f_n = (\lambda().y) \\ & \text{et} & f_c = (\lambda e, r.cons\ e\ r) \end{array}$$

En considérant $g = length$, le théorème de promotion du *fold* donne :

$$\begin{array}{llcll} & \phi_n & = & length\ y & \\ \text{et} & \phi_c\ r_1\ r_2 & = & length\ (cons\ x_1\ x_2) & \text{avec} \quad [x_1/r_1, (length\ x_2)/r_2] \end{array}$$

L'*application à une construction* donne :

$$\phi_c\ r_1\ r_2 = succ\ (length\ x_2)$$

Et pour terminer, la *généralisation* de *length* x_2 par r_2 permet d'obtenir :

$$\phi_c\ r_1\ r_2 = succ\ r_2$$

```
aglet length =
  length x →
    length.result = x.s
  nil →
    nil.s = zero              ♣
  cons head tail →
    cons.s = succ tail.s      ♠
```

FIG. 6.8: *Définition de length en grammaire attribuée*

Le résultat de l'algorithme de normalisation sur *length (append x y)* est donc le *fold* :

$$\begin{array}{ll}(fold^{list} & (\lambda().(fold^{list}\ (\lambda().zero)\ (\lambda e, r.succ\ r))y)\\ & (\lambda r_1, r_2.succ\ r_2))\ x\ y\end{array}$$

dans lequel plus aucune structure intermédiaire n'est construite.

Composition descriptionnelle

Rappelons brièvement que le but de la composition descriptionnelle est de construire statiquement, à partir de deux grammaires attribuées f et g telles que $f : T_1 \rightarrow T_2$ et $g : T_2 \rightarrow T_3$, une nouvelle grammaire attribuée $(g\_f) : T_1 \rightarrow T_3$ qui a la même sémantique que l'application successive de f et de g, mais telle que cette nouvelle grammaire attribuée ne construise plus la structure intermédiaire correspondant au type T_2.

L'idée de base est la notion de *projection de règle sémantique*. Intuitivement, (g_f) est construite à partir de f en remplaçant chaque règle sémantique qui construit un terme de type T_2 par la projection des règles sémantiques de g sur ce terme.

Plus précisément, soit r_i^f une règle sémantique de f, correspondant au pattern de constructeur $c_i^{T_1}$, qui construit un terme de type T_2 avec le constructeur $c_j^{T_2}$. Ce terme étant une structure de type T_2 sur lequel la grammaire g doit être appliquée, il représente donc une structure intermédiaire, puisque les règles $\overline{r_j^g}$ associées au pattern de constructeur $c_j^{T_2}$ dans g lui seront appliquées. Dans ce cas, pour le pattern de constructeur $c_i^{T_1}$ dans (g_f), la règle r_i^f sera remplacée par la projection des règles $\overline{r_j^g}$ sur r_i^f. Cette projection suit la structure et l'ordre des paramètres du constructeur original $c_j^{T_2}$ dans r_i^f. Par ailleurs, de nouveaux attributs sont créés : pour chaque attribut a d'une variable de type X dans f tel que le type de a est celui d'un non-terminal Y de g, un nouvel attribut a_b sur X est déclaré dans (g_f) pour chaque attribut b de Y dans g. Le type du nouvel attribut a_b est celui de b.

Cette présentation de la composition descriptionnelle est quelque peu restreinte, puisqu'elle ne distingue pas les attributs syntaxiques des attributs sémantiques et qu'elle ne traite pas la projection des règles sémantiques ayant des *if–then–else* ; cependant, elle est suffisante pour notre propos. Par ailleurs, le lecteur pourra trouver une définition plus formelle au chapitre 3 de cette thèse ou dans l'article original [GG84].

Pour l'exemple de (*length* ∘ *append*), les définitions en grammaire attribuée de chacune des fonctions sont rappelées dans les figures 6.8 et 6.9. La règle sémantique ♡ de *append* crée un *cons*. Donc, la règle sémantique ♠ pour la production *cons* de *length* est projetée sur le constructeur *cons* de la règle sémantique ♡ de *append*, conformément à ses paramètres. Un

$$
\begin{array}{ll}
\texttt{aglet}\ append = & \\
\quad append\ x\ y \rightarrow & \\
\qquad append.result = x.v & \\
\quad nil \rightarrow & \\
\qquad nil.v = y & \diamondsuit \\
\quad cons\ head\ tail \rightarrow & \\
\qquad cons.v = cons\ head\ tail.v & \heartsuit
\end{array}
$$

FIG. 6.9: *Définition de append en grammaire attribuée*

$$
\begin{array}{l}
\texttt{aglet}\ lengthappend = \\
\quad lengthappend\ x\ y \rightarrow \\
\qquad lengthappend.result = x.v\_s \\
\quad nil \rightarrow \\
\qquad nil.v\_s = (length\ y) \\
\quad cons\ head\ tail \rightarrow \\
\qquad cons.v\_s = succ\ tail.v\_s
\end{array}
$$

FIG. 6.10: *La composition descriptionnelle de length avec append*

---

nouvel attribut v_s est créé dans la règle sémantique résultante sur la production *cons* de *lengthappend* (figure 6.10).

6.1.4 Une première comparaison

Tout comme la différence entre l'évaluation des *folds* et l'évaluation des grammaires attribuées (la définition des foncteurs dépend uniquement du type tandis que le générateur d'évaluateur dépend de la forme des règles sémantiques), la différence entre l'algorithme de normalisation et la composition descriptionnelle tient à la connaissance de la méthode d'évaluation. La composition descriptionnelle ne nécessite pas du tout cette connaissance. En fait, la méthode d'évaluation choisie pour la grammaire attribuée résultante d'une composition descriptionnelle peut être différente des méthodes d'évaluations qui auraient été choisies pour les grammaires attribuées d'entrée. On distingue différentes classes de grammaires attribuées en fonction de leurs techniques d'évaluation possibles. La plus large de ces classes est celle des grammaires attribuées *non-circulaires* et elle est stable par la composition descriptionnelle[1], c'est-à-dire le résultat de la composition descriptionnelle de deux grammaires attribuées non-circulaires est encore une grammaire attribuée non-circulaire [Gie88].

Pour interpréter la composition descriptionnelle dans les termes de l'algorithme de normalisation, la projection des règles sémantiques de la composition descriptionnelle produit directement la version déforestée des ϕ_i, tandis que les ϕ_i obtenues par le théorème de promotion doivent encore être déforestées par les étapes d'*application à une construction* et de *généralisation* de l'algorithme de normalisation. Il est donc possible de voir la composition descriptionnelle comme un algorithme de normalisation plus symbolique, au sens où elle est

1. Cette propriété de stabilité (clôture) des classes de grammaires attribuées par la composition descriptionnelle, principalement étudiée dans [Gie88], n'est pas systématiquement réalisée pour toutes les classes.

$$
\begin{aligned}
naif\_rev\ x \quad = \quad & (fold^{list} \\
& \quad (\lambda().nil) \\
& \quad (\lambda e, r.append\ r\ (cons\ e\ nil))\ )\ x
\end{aligned}
$$

FIG. 6.11: *La forme "naïve" de l'inversion de liste en fold*

$$
\begin{array}{l}
\texttt{aglet}\ naif\_rev = \\
\quad naif\_rev\ x \rightarrow \\
\quad\quad naif\_rev.result = x.s \\
\quad nil \rightarrow \\
\quad\quad nil.s = nil \\
\quad cons\ head\ tail \rightarrow \\
\quad\quad cons.s = (append\ tail.s\ (cons\ head\ nil))
\end{array}
$$

FIG. 6.12: *La grammaire attribuée correspondant au fold pour naif_rev*

indépendante [2] de la méthode ou de l'ordre d'évaluation des attributs alors que le théorème de promotion est fortement lié à la définition des foncteurs, guidant cette évaluation. La composition descriptionnelle est donc une transformation de source à source, indépendante de toute méthode d'évaluation : c'est une composition purement symbolique qui n'est guidée que par la forme des règles sémantiques et non par une abstraction générale en terme de schéma de récursion liée aux *patterns* de type auxquelles elles sont associées.

Cependant, dans le cadre des fonctions pouvant s'exprimer par un *fold* du premier ordre, le foncteur étant valable pour l'évaluation de la grammaire attribuée S^1 correspondante, l'algorithme de normalisation et la composition descritpionnelle sont équivalents en terme de déforestation.

6.2 *Folds* de 2^{nd} ordre et grammaires attribuées générales

Dans cette section, je m'intéresse aux programmes plus complexes qui ne peuvent pas être déforestés dans leur représentation en *fold* du premier ordre. La normalisation de certains de ces programmes nécessite une transformation en *folds* de second ordre [3]. Par ailleurs, ces programmes peuvent être représentés par des grammaires attribuées *générales* (avec des attributs hérités).

6.2.1 Des problèmes de déforestation

L'algorithme de normalisation, tel qu'il a été présenté, ne peut pas être appliqué sur tous les programmes écrits à l'aide de *folds*, de même que la composition descriptionnelle ne s'applique pas sur toutes les grammaires attribuées. Par exemple la fonction d'inversion de liste exprimée avec un *fold* dans sa forme naïve (*naif_rev*, figure 6.11) n'est pas *potentiellement normalisable* [SF93]. En effet, la fonction *append* qu'elle contient agit sur la variable accumulative de résultat r qui est en cours de construction. Plus précisément, le *fold* interne (de la

2. En prenant garde cependant au problème de la clôture des classes de grammaires attribuées par la composition descriptionnelle.

3. Un *fold* est de second ordre lorsque l'une de ses fonctions accumulatives de résultat l'est.

$$\begin{array}{lcl} reverse\ x & = & (fold^{list} \\ & & \quad (\lambda().\lambda w.w) \\ & & \quad (\lambda e, r.\lambda w.(r\ (cons\ e\ w)))\ )\ x\ nil \end{array}$$

FIG. 6.13: *La forme en fold de second ordre pour l'inversion de liste*

fonction *append*) porte sur r qui est une variable du type *list* liée par le *fold* externe.

Pour une raison similaire, la composition descriptionnelle ne peut pas être appliquée sur la grammaire attribuée correspondante ($naif\_rev$, figure 6.12). Pour appliquer la composition descriptionnelle sur des grammaires attribuées, les constructeurs de la structure intermédiaire à éliminer doivent être directement visibles, pour que la projection des règles sémantiques sur ces constructeurs soit possible et correcte. Dans la grammaire attribuée pour $naif\_rev$ (figure 6.12) correspondant au *fold* naïf, la fonction *append* "cache" les constructeurs et interdit l'application de la composition descriptionnelle.

6.2.2 L'approche "ordre supérieur"

Pour résoudre ce problème, Sheard et Fegaras [SF93] ont introduit le *théorème de promotion du second ordre.* En effet, les fonctions qui posent problème peuvent dans certains cas être transformées en des *folds* de second ordre, obtenus grâce à une transformation $\mathcal{F}_{\mathcal{G}}$ [SF93]. Par exemple, la fonction d'inversion de liste naïve, qui n'est pas normalisable sous sa forme en *fold* du premier ordre (figure 6.11), est transformée en un *fold* de second ordre présenté à la figure 6.13.

Cependant, la transformation $\mathcal{F}_{\mathcal{G}}$ impose des restrictions sur le type de base et sur les fonctions accumulatives de la forme naïve. Sans rentrer dans les détails (voir [SF93]), le type doit posséder un *zéro-constructeur* (constructeur sans arguments), et les fonctions accumulatives apparaissant dans le programme fold à transformer doivent utiliser des *fonctions de zéro-remplacement.* Par exemple, dans le cas de *reverse*, *nil* est un zéro-constructeur pour le type *list* et la fonction *append* est une fonction de zéro-remplacement. Grâce à ces conditions très particulières, la forme naïve en *fold* de *reverse* peut être transformée par $\mathcal{F}_{\mathcal{G}}$, et le fold de second ordre résultant est normalisable avec le Théorème de Promotion de second ordre. Grâce à cette extension, l'algorithme de normalisation peut effectuer des déforestations telles que *length(reverse(x)) = length(x)* ou *reverse(reverse(x)) = copy(x).*[4]

En aparté, je signale ici qu'il existe une transformation classique [Knu68, CM79] qui produit, pour toute grammaire attribuée, la grammaire attribuée purement synthétisée d'ordre supérieur correspondante[5]. À partir de la grammaire attribuée naturelle pour l'inversion ($reverse$, présentée figure 6.14), cette transformation produit la grammaire attribuée S^1 d'ordre supérieur de la figure 6.15. Même si la transformation en grammaire attribuée d'ordre supérieur est plus générale que la transformation $\mathcal{F}_{\mathcal{G}}$ de [SF93] (en particulier, elle ne possède pas de restriction de type concernant des zéro-constructeurs), il est intéressant de remarquer la ressemblance entre cette grammaire attribuée S^1 et le *fold* d'ordre supérieur de la figure 6.13, produit par la transformation $\mathcal{F}_{\mathcal{G}}$ à partir du *fold* naïf.

4. En utilisant, en plus du Théorème de Promotion de second ordre, la transformation $\mathcal{F}_{\mathcal{G}}^{-1}$ définie dans [SF93].

5. Ici, "grammaire attribuée d'ordre supérieur" ne désigne pas l'extension du formalisme des grammaires attribuées de Vogt *et al.* [VSK89], mais une grammaire attribuée (classique) avec des attributs et des règles sémantiques d'ordre supérieur.

$$
\begin{array}{l}
\texttt{aglet}\ reverse = \\
\quad reverse\ x \rightarrow \\
\qquad reverse.result = x.s \\
\qquad x.h = nil \\
\quad nil \rightarrow \\
\qquad nil.s = nil.h \\
\quad cons\ head\ tail \rightarrow \\
\qquad cons.s = tail.s \\
\qquad tail.h = cons\ head\ cons.h
\end{array}
$$

FIG. 6.14: *La grammaire attribuée "naturelle" pour reverse*

$$
\begin{array}{l}
\texttt{aglet}\ reverse = \\
\quad reverse\ x \rightarrow \\
\qquad reverse.result = (x.s\ nil) \\
\quad nil \rightarrow \\
\qquad nil.s = (\lambda w.w) \\
\quad cons\ head\ tail \rightarrow \\
\qquad cons.s = (\lambda w.((tail.s)\ (cons\ head\ w)))
\end{array}
$$

FIG. 6.15: *La grammaire attribuée d'ordre supérieur correspondant à la grammaire attribuée "naturelle" pour reverse*

6.2.3 L'approche "héritée"

Pour la composition descriptionnelle, l'impuissance à traiter les grammaires attribuées comme celle de la figure 6.12, n'a jamais été un problème crucial. En effet, la forme "naturelle" en grammaire attribuée pour exprimer l'inversion de liste n'est pas celle de la figure 6.12, mais plutôt une grammaire attribuée (*reverse*, figure 6.14) utilisant un attribut hérité, h, initialisé à *nil*. On peut alors appliquer directement la composition descriptionnelle sur (la composition de) cette grammaire attribuée naturelle (avec elle-même ou avec une autre grammaire attribuée, comme *length*).

La notion d'attributs synthétisés et hérités est le concept de base du formalisme des grammaires attribuées. Introduits dès le début de la théorie des grammaires attribuées [Knu68], ils permettent de décrire à la fois les calculs "montants" et "descendants" sur la structure. La composition descriptionnelle, qui a été définie sur les grammaires attribuées générales (ayant des attributs synthétisés et hérités), peut être appliquée indifféremment avec n'importe quelle grammaire attribuée, qu'elle soit S^1 ou non. Par exemple, elle agit directement sur la grammaire attribuée naturelle pour *reverse* (figure 6.14), qui utilise un attribut hérité.

La figure 6.16 présente le résultat *reverse_reverse* de la composition descriptionnelle sur l'exemple *reverse* (*reverse x*), sans rentrer dans les détails de la transformation. L'idée de base, même avec les attributs hérités, reste la projection des règles sémantiques (cf. chapitre 3).

Les grammaires attribuées produites par la composition descriptionnelle contiennent souvent beaucoup de règles de recopie entre les attributs, qui n'ont pas d'autre rôle que de propager les valeurs le long de la structure ; une analyse statique globale simple [Rou94] permet

$$
\begin{array}{l}
\texttt{aglet}\ reverse\_reverse = \\
\quad reverse\_reverse\ x \rightarrow \\
\quad\quad reverse\_reverse.result = x.s\_s \\
\quad\quad x.h\_h = x.h\_s \\
\quad\quad x.s\_h = nil \\
\quad nil \rightarrow \\
\quad\quad nil.s\_s = nil.h\_s \\
\quad\quad nil.h\_h = nil.s\_h \\
\quad cons\ head\ tail \rightarrow \\
\quad\quad cons.s\_s = tail.s\_s \\
\quad\quad tail.h\_s = cons.h\_s \\
\quad\quad cons.h\_h = cons\ head\ tail.h\_h \\
\quad\quad tail.s\_h = cons.s\_h
\end{array}
$$

FIG. 6.16: *La grammaire attribuée résultante de la composition descriptionnelle de reverse avec elle-même*

d'éliminer la plupart de ces règles de recopie ; je reviendrai sur ces transformations au chapitre 9. À partir de la grammaire attribuée présentée à la figure 6.16, cette élimination des règles de copie conduit à la grammaire attribuée de recopie d'une liste, ce qui est équivalent au résultat obtenu par l'algorithme de normalisation de second ordre, utilisant les transformations $\mathcal{F}_{\mathcal{G}}$ et $\mathcal{F}_{\mathcal{G}}^{-1}$, avec des conditions restrictives sur les constructeurs des types considérés.

Un exemple avec le type *tree*

Pour diversifier les types des exemples étudiés et pour illustrer le cas d'un type ne possédant pas naturellement de zéro-constructeur, je donne ici un petit exemple de composition descriptionnelle utilisant le type *tree* (arbres binaires d'entiers). La fonction *flip* prend un arbre comme argument dont elle inverse les sous-arbres à chaque nœud. Par exemple :

$$
\begin{array}{rl}
 & flip\ (node\ (node\ (leaf\ 1)\ (leaf\ 2))\ (leaf\ 3)) \\
= & (node\ (leaf\ 3)\ (node\ (leaf\ 2)(leaf\ 1)))
\end{array}
$$

La fonction *sigma* prend également un arbre comme argument et retourne un arbre qui a la même structure, mais où les valeurs aux feuilles ont été ajoutées entre elles, de gauche à droite. Par exemple :

$$
\begin{array}{rl}
 & sigma(node\ (node\ (leaf\ 2)\ (leaf\ 5))\ (leaf\ 3)) \\
= & (node\ (node\ (leaf\ 2)\ (leaf\ 7))\ (leaf\ 10))
\end{array}
$$

La grammaire attribuée pour *flip* est donnée à la figure 6.17 et celle pour *sigma*[6] à la figure 6.18. Notre but est d'avoir une version de la composition des fonctions ($flip \circ sigma$) qui ne construise pas l'arbre intermédiaire. La composition descriptionnelle de *flip* avec *sigma* produit la grammaire attribuée présentée à la figure 6.19.

6. On peut remarquer que cette grammaire attribuée pour *sigma* n'est pas en forme normale, à cause de la règle sémantique $leaf.s = leaf\ leaf.v$. Cela permet de simplifier sa présentation et ne pose pas de problèmes puisque, de toute façon, il est possible de la transformer dans sa forme normale [Boc76].

$$
\begin{array}{l}
\texttt{aglet}\ \mathit{flip} = \\
\quad \mathit{flip}\ t \rightarrow \\
\qquad \mathit{flip.result} = t.m \\
\quad \mathit{leaf}\ i \rightarrow \\
\qquad \mathit{leaf}.m = \mathit{leaf}\ i \\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\qquad \mathit{node}.m = \mathit{node}\ \mathit{right}.m\ \mathit{left}.m
\end{array}
$$

FIG. 6.17: *La grammaire attribuée pour flip*

$$
\begin{array}{l}
\texttt{aglet}\ \mathit{sigma} = \\
\quad \mathit{sigma}\ t \rightarrow \\
\qquad \mathit{sigma.result} = t.s \\
\qquad t.h = \mathit{zero} \\
\quad \mathit{leaf}\ i \rightarrow \\
\qquad \mathit{leaf}.s = \mathit{leaf}\ \mathit{leaf}.v \\
\qquad \mathit{leaf}.v = \mathit{plus}\ i\ \mathit{leaf}.h \\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\qquad \mathit{node}.s = \mathit{node}\ \mathit{left}.s\ \mathit{right}.s \\
\qquad \mathit{node}.v = \mathit{right}.v \\
\qquad \mathit{right}.h = \mathit{left}.v \\
\qquad \mathit{left}.h = \mathit{node}.h
\end{array}
$$

FIG. 6.18: *La grammaire attribuée pour sigma*

La composition descriptionnelle produit donc une grammaire attribuée (figure 6.19) qui évite la construction de l'arbre intermédiaire et qui effectue **uniquement** les constructions nécessaires au résultat final. Le type *tree* n'ayant pas de zéro-constructeurs, il n'est pas possible d'exprimer ces fonctions avec des fonctions accumulatives de zéro-remplacement et l'algorithme de l'algorithme de normalisation des *folds* ne peut donc pas déforester ce programme.

6.2.4 Comparaison

Avec le passage à des programmes plus complexes, les différences dans le traitement de la déforestation s'affirment entre les *folds* de second ordre et les grammaires attribuées générales, c'est-à-dire utilisant éventuellement des attributs hérités.

La notion d'attribut hérité n'existant pas dans le formalisme des *folds*, ces derniers doivent utiliser des fonctions d'ordre supérieur pour obtenir la même expressivité, c'est-à-dire pour décrire des calculs descendants dans une structure.

Les variables d'accumulation des fonctions récursives (comme celle dans laquelle l'inversion de liste est construite dans *reverse*) sont des arguments "supplémentaires" ou "secondaires" par rapport à l'argument "principal" ou *pattern matché* sur lequel s'applique la récursion (comme la liste à inverser dans l'exemple de *reverse*) qui est guidée par la structure de son type. Ces variables d'accumulation peuvent contenir des constructions qui ne suivent pas le schéma de récursion de la fonction. En grammaires attribuées, ces variables d'accumulation

$\texttt{aglet}\ \mathit{flip\_sigma} =$
$\quad \mathit{flip\_sigma}\ t \rightarrow$
$\quad\quad \mathit{flip\_sigma.result} = t.m\_s$
$\quad\quad t.m\_h = \mathit{zero}$
$\quad \mathit{leaf}\ i \rightarrow$
$\quad\quad \mathit{leaf}.m\_s = \mathit{leaf}\ \mathit{leaf}.m\_v$
$\quad\quad \mathit{leaf}.m\_v = \mathit{plus}\ i\ \mathit{leaf}.m\_h$
$\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow$
$\quad\quad \mathit{node}.m\_s = \mathit{node}\ \mathit{right}.m\_s\ \mathit{left}.m\_s$
$\quad\quad \mathit{node}.m\_v = \mathit{left}.m\_v$
$\quad\quad \mathit{left}.m\_h = \mathit{right}.m\_v$
$\quad\quad \mathit{right}.m\_h = \mathit{mode}.m\_h$

FIG. 6.19: *La résultat de la composition descriptionnelle de flip avec sigma*

peuvent être représentées comme des objets de première classe, en tant qu'attributs hérités ; ce pouvoir d'expression n'est pas accessible par les *folds* qui, n'étant guidés que par le schéma de récursion du type de l'argument *pattern matché*, ne peuvent représenter l'accumulation d'un résultat dans un paramètre supplémentaire que par une fonction d'ordre supérieur (une clôture).

Ceci est dû aux définitions des *folds* qui sont intimement liées aux définitions des foncteurs, eux-mêmes statiquement déterminés à partir des définitions algébriques des types. Pour introduire la notion d'*hérité* dans le formalisme des *folds*, les foncteurs ne devraient plus être définis uniquement en fonction de la structure du type, mais aussi en fonction des signatures des fonctions accumulatives qui décrivent le résultat et les arguments du calcul ; ceci interdirait la définition statique des foncteurs à partir du type seul. Un tel *foncteur étendu*, qui tiendrait compte à la fois de la structure (type) et de la forme des calculs (signatures des *fonctions accumulatives étendues*), pourrait éventuellement être défini en utilisant les techniques de génération d'évaluateurs d'attributs. En effet, la plupart des évaluateurs d'attributs (pour les classes de grammaires attribuées plus générales) sont produits statiquement à la fois à partir des types algébriques et des signatures des règles sémantiques.

Par ailleurs, l'algorithme de normalisation et la composition descriptionnelle diffèrent par la forme de leur programme d'entrée. Pour les programmes qui ne sont pas directement normalisables au premier ordre, l'algorithme de normalisation impose, pour le passage au second ordre, que ce programme soit exprimé avec des fonctions de zéro-remplacement, tandis que la composition descriptionnelle ne nécessite pas une telle contrainte. Même si ces fonctions particulières peuvent être déduites automatiquement à partir du zéro-constructeur du type considéré, il existe des types sans zéro-constructeur (*tree* par exemple). Dans ces cas, la composition descriptionnelle peut être appliquée directement, mais l'algorithme de normalisation échoue, faute de savoir écrire ces exemples avec des fonctions de zéro-remplacement. Dans [FSZ94], la contrainte de zéro-constructeur semble avoir été éliminée, en passant systématiquement d'une définition récursive à un *fold* d'ordre supérieur (*Conversion de programmes fonctionnels en programmes algébriques*). L'approche de [FSZ94] permet de convertir la version fonctionnelle de l'exemple de *reverse* (version itérative) en une version inductive, en ordre supérieur, sur laquelle l'algorithme de normalisation peut être appliqué tout comme la composition descriptionnelle peut être appliquée sur la grammaire attribuée naturelle pour *reverse*. Cependant, des problèmes subsistent car cette conversion, qui utilise un algorithme de normalisation étendu (sous forme de système de règles d'inférences), peut tout de même échouer dans certains cas. De plus, les résultats sont obtenus dans une forme en ordre supérieur et ne peuvent pas toujours être ramenés au premier ordre ; c'est le cas de l'exemple de *reverse* : la forme naïve avec *append* n'est plus nécessaire, mais la forme naturelle conduit à une version en *fold* du second ordre qu'on ne peut pas ramener au premier ordre. Au contraire, la composition descriptionnelle travaille directement sur une spécification de grammaire attribuée avec des attributs hérités, sans la convertir préalablement en ordre supérieur.

Finalement, même s'ils aboutissent dans certains cas à des résultats de déforestation équivalents, l'algorithme de normalisation et la composition descriptionnelle utilisent des techniques de transformation assez différentes. Plus précisément, la transformation effectuée par l'algorithme de normalisation dépend fortement d'une évaluation partielle du terme d'entrée tandis que la composition descriptionnelle ne réalise la déforestation que par une transformation symbolique indépendante de la méthode d'évaluation.

Les limites actuelles de la composition descriptionnelle sont la nécessité de "visibilité" des constructeurs dans les règles sémantiques et l'impossibilité de traiter l'ordre supérieur (notion inhabituelle dans les domaines d'applications classiques des grammaires attribuées). Cependant, la notion d'attribut hérité en grammaire attribuée permet de représenter des programmes qui nécessitent l'ordre supérieur dans le formalisme des *folds* ; de plus, la normalisation des *folds* a dû être étendue et améliorée pour résoudre les problèmes posés par l'ordre supérieur, nécessaire aux *folds* pour atteindre un pouvoir d'expression suffisant. Il est intéressant de constater que la puissance d'expression induite par la notion d'attribut hérité permet de traiter la plupart des extensions nécessaires pour les *folds*. De plus, cette notion étant classique en grammaire attribuée, elle a été prise en compte initialement dans les nombreux travaux de recherche sur les grammaires attribuées et plus particulièrement ceux qui concernent la composition descriptionnelle.

6.3 L'apport des hylomorphismes

Outre les *folds*, un formalisme largement étudié dans le cadre de la déforestation de programmes est celui des hylomorphismes, plus précisément sous forme de triplets. La section 5.5 détaille assez précisément ce formalisme et les techniques qui lui sont dédiées.

Généralisation des *folds* et de la forme d'entrée

Les hylomorphismes et leurs règles de transformation permettent de généraliser les *folds* et leur algorithme de normalisation, comme cela a été mentionné à la section 5.5.4. L'exemple de $length \circ append$, détaillé section 5.5.3, présente un cas où l'application des règles spécifiques aux hylomorphismes simplifie la déforestation et permet d'envisager plus facilement son automatisation.

De plus, l'algorithme de dérivation d'un hylomorphisme à partir d'une définition de fonction récursive classique, développé par Hu *et al.* et brièvement présenté à la section 5.5.3, permet de résoudre le problème de représentation d'une fonction, au moins du point de vue du programmeur, dans sa forme d'entrée pour le système de fusion. En effet, l'utilisation du système de fusion d'hylomorphismes n'est guère réaliste s'il faut demander à l'utilisateur d'écrire directement ses programmes comme des compositions d'hylomorphismes sous forme de triplets et de déterminer lui-même les foncteurs associés.

Les grammaires attribuées classiques ne sont pas un formalisme très répandu ni très connu dans la communauté fonctionnelle. Elles sont cependant plus intuitives que les notions catégorielles. Par contre, l'algorithme de dérivation d'un hylomorphisme à partir d'une définition de fonction récursive classique libère l'utilisateur de la contrainte d'écrire ses programmes dans un formalisme qui ne lui est pas familier. Cet algorithme de dérivation, accompagné de la traduction inverse, permet de rendre le système de fusion d'hylomorphismes "utilisable" pour les langages de programmation fonctionnelle classiques.

Assouplissement des foncteurs

J'ai déjà signalé à la section 6.2.4 que les foncteurs utilisés dans les folds, déterminés statiquement à partir de la définition d'un type algébrique, étaient trop rigides pour permettre d'exprimer les calculs spécifiés par des fonctions construisant leur résultat grâce à un paramètre d'accumulation.

Les foncteurs qui guident la récursion des hylomorphismes ne sont plus simplement les foncteurs des types algébriques. Ils correspondent plus précisément au schéma de récursion de la fonction représentée et peuvent être pour cela générés par l'algorithme de dérivation d'un hylomorphisme à partir d'une définition de fonction récursive. Par ailleurs, les diverses transformations de fusion ou de restructuration d'hylomorphismes permettent de modifier ces foncteurs. C'est un début de réponse au problème du trop grand isomorphisme entre les foncteurs guidant la déforestation et les types à partir desquels ils sont générés. En tout cas, c'est un rapprochement de la technique utilisée pour les grammaires attribuées où l'évaluateur, qui est généré en fonction de la *forme* des règles sémantiques, ne correspond au schéma de récursion de la fonction que dans les cas les plus simples. L'avantage de cette souplesse des foncteurs se traduit par une meilleure représentation des constructions intervenant dans des variables d'accumulation.

Dépendance entre déforestation et évaluation

Malheureusement, les inconvénients des hylomorphismes apparaîssent lors de la tentative de fusion. En effet, comme je l'ai présenté à la fin de la section 5.5.5, la déforestation des programmes tels que $length \circ reverse$ n'est plus possible par les règles "agréables" d'*Acid Rain* (théorème 5.5.3) ; la raison en est la "spécificité" du foncteur de l'hylomorphisme correspondant à *reverse*, qui n'est plus compatible avec celui du type algébrique des listes. Les deux fonctions du programme de cet exemple peuvent être fusionnées par *HyloFusion* (théorème 5.5.2), qui est moins simple et moins facile à mettre en œuvre, mais la déforestation obtenue n'est pas satisfaisante : cette règle produit bien un hylomorphisme, mais qui ne calcule la longueur de la liste qu'après l'avoir construite.

Pour les grammaires attribuées, la détermination de la méthode d'évaluation est complètement distincte du mécanisme de déforestation et réciproquement. Grâce à cette indépendance, les programmes pour lesquels la déforestation pose des problèmes aux *folds* et aux hylomorphismes sont exprimables sous la forme de grammaires attribuées qui sont correctement déforestées par la composition descriptionnelle.

Par conséquent, bien que la fusion des hylomorphismes constitue une amélioration des méthodes de déforestations calculationnelles, particulièrement appréciable dans le cadre d'un traitement automatisé de la déforestation, il reste une classe de programmes, à laquelle la double inversion de liste et l'inversion de la liste des feuilles d'un arbre appartiennent, pour lesquelles elle n'apporte pas de déforestation suffisante ou ne peut pas être appliquée. Ces programmes peuvent être caractérisés par la construction d'une structure intermédiaire dans un paramètre d'accumulation, comme c'est le cas pour *rev* ou *flat*. Le foncteur associé à ces fonctions, correspondant à leur schéma de récursion, ne permet pas de déforester cette construction "cachée" dans le paramètre d'accumulation. Les travaux de Meijer et Hutton [MH95] sur les *difoncteurs*, permettant de représenter des schémas récursifs plus complexes et notablement moins réguliers (c'est-à-dire pouvant exprimer différents schémas de récursions dans une même fonction) pourront peut être apporter une plus grande correspondance entre l'abstraction d'une fonction et la construction des structures intermédiaires qui y sont effectuées. Dès à présent, les grammaires attribuées peuvent exprimer ces fonctions à l'aide d'un attribut hérité qui spécifie la construction au même titre que tous les autres calculs et la composition descriptionnelle permet de les déforester.

6.4 Comparaison, limitations et objectifs

Cette section dresse un bilan de l'évolution des méthodes de déforestations fonctionnelles étudiées et de leur comparaison avec la composition descriptionnelle des grammaires attribuées. Elle établit les limitations et présente les objectifs qui motivent les travaux décrits dans les chapitres suivants.

Comparaison des méthodes

Je présente ici un récapitulatif en trois points des méthodes de déforestation pour les *folds*, pour les hylomorphismes et pour les grammaires attribuées. Le premier point caractérise l'algorithme de déforestation dans sa faculté à évoluer, dans sa simplicité, dans sa mise en œuvre. Le second point représente la puissance de l'algorithme, son efficacité en terme de déforestation. Finalement, le troisième et dernier point caractérise la facilité d'utilisation, en fonction de la forme d'entrée acceptée par la méthode.

Les folds

1. Grâce aux *folds*, il est possible d'exprimer des types ou des schémas de calculs complexes. Cependant, chaque nouveau concept étendant la puissance d'expression initiale des *folds* (*folds* d'ordre supérieur, algorithme de normalisation étendu) introduit de profondes modifications dans les algorithmes de transformation. Ces modifications rendent la normalisation et la déforestation assez peu *maintenable*.

2. L'algorithme de normalisation étendu est un système de réécriture complexe, qui n'est pas simple à automatiser et qui ne conduit pas nécessairement à une forme déforestée. Les programmes ne sont pas toujours acceptables, comme sur les types n'ayant pas de zéro-constructeurs, ou peuvent être pris en compte sans pouvoir être normalisés (cf. exceptions dans [FSZ94]).

3. Pour pouvoir être normalisé, un programme doit nécessairement être exprimé dans le formalisme spécifique des *folds*. Ceci semble être trop exigent pour être utilisé pratiquement par un programmeur et la mise à disposition de bibliothèques de fonctions dans leur version "*fold*" est coûteuse, ou inadaptée.

Les hylomorphismes

1. Les hylomorphismes sous forme de triplets couvrent toutes les fonctions représentables avec des *folds*. Ces spécifications abstraites associées à des foncteurs polynômiaux permettent à l'algorithme de déforestation d'être plus homogène, plus général et plus facile à maintenir et à faire évoluer.

2. Le théorème de promotion du *fold* possède son équivalent dans le formalisme des hylomorphismes sous forme de triplets (HyloFusion), mais d'autres propriétés agréables de ce formalisme peuvent être dédiées au contexte de fusion (Hylo Shift, Acid Rain). Ces propriétés facilitent l'automatisation des transformations. Cependant, les fusions ne correspondent pas nécessairement à des déforestations satisfaisantes.

3. Il existe une transformation automatique de programmes fonctionnels classiques dans leur forme en hylomorphismes. Ceci épargne au programmeur des spécifications laborieuses dans un formalisme assez peu trivial, tout autant que la nécessité d'une bibliothèque figée de fonctions avec leurs équivalents en hylomorphismes.

Les grammaires attribuées

1. Comme dans les formalismes calculationnels tels que les hylomorphismes, la notion de type abstrait existe dans les grammaires attribuées sous forme de grammaire ou de type algébrique. En fait, toutes les transformations de grammaires attribuées exploitent largement cette notion. La composition descriptionnelle en fait partie et de plus, c'est une transformation purement syntaxique, indépendante de tout foncteur lié à ce type. Elle est pleinement générique en ce sens qu'elle est indépendante de toute représentation de type (algèbre) et de tout ordre d'évaluation (foncteur).

2. Comme la composition descriptionnelle est seulement basée sur la projection des règles sémantiques, purement syntaxique, elle peut être appliquée sans aucune modification quel que soit le type. De plus, la composition descriptionnelle fonctionne naturellement au niveau des spécifications en grammaires attribuées.

3. Pour pouvoir être appliquée, la composition descriptionnelle doit disposer d'une spécification déclarative, sous la forme d'un ensemble d'équations orientées pour chaque constructeur du type de donnée considéré, dans lesquels les constructions doivent être visibles. Cette forme correspond aux spécifications habituelles de grammaires attribuées, utilisant la notion d'attribut hérité. Bien que ce formalisme soit assez simple et déclaratif, il est différent des spécifications de programmes fonctionnels classiques, sur lesquels la composition descriptionnelle ne peut être appliquée directement.

Limitations

En dépit du traitement efficace de nombreux programmes par l'algorithme de normalisation, les cas particuliers et la possibilité d'exceptions avec ce type de traitement a conduit à développer l'approche des hylomorphismes sous forme de triplets, qui peuvent traiter indifféremment les fonctions de premier ou de second ordre. Malheureusement, l'encapsulation de certains calculs dans des paramètres de fonctions de second ordre peut quelques fois *cacher* des constructions déforestables, comme cela a été présenté à la fin de la section 5.5.5. Ainsi, les compositions d'hylomorphismes correspondant aux fonctions *append* ou *reverse* peuvent être fusionnées automatiquement, mais la déforestation effectuée est insuffisante, comme le montrent les exemples détaillés dans la section 5.5.5. Par exemple, $reverse \circ reverse$ peut être traité par ce système, mais produit une fonction qui est loin d'être une simple recopie de liste (en terme de nombre de constructions), comme c'est le cas avec la composition descriptionnelle.

Les comparaisons faites dans la section précédente, après avoir présenté différentes méthodes dans différents formalismes, permettent d'établir des limitations dans chaque technique.

Les *folds* du premier ordre et les grammaires attribuées purement synthétisées apparaissent comme ayant le même pouvoir d'expression et une puissance similaire, malgré des techniques différentes. Cependant, la comparaison d'un programme écrit en *fold* avec son équivalent en grammaire attribuée S^1 est obscurcie par les différences de notations (implicite/explicite). Par contre, sur ces fonctions, les deux méthodes conduisent à une déforestation efficace.

Dans le cas des *folds* d'ordre supérieur, permettant d'exprimer des fonctions moins triviales, cette différence devient beaucoup plus importante : elle n'est plus uniquement au niveau des notations, mais au niveau de la façon de spécifier, ou de considérer le calcul à effectuer. Les grammaires attribuées autorisent encore une spécification très déclarative, au niveau des constructeurs du type considéré. Ceci est rendu possible par l'existence des attributs hérités, permettant la description simple de calculs *descendant la récursion*. Pour les *folds* d'ordre supérieur, ne bénéficiant pas de ces objets, les calculs doivent être décrits par l'intermédiaire de fonctions (d'ordre supérieur) permettant de modéliser l'attente d'un résultat, sachant que le parcours récursif de la structure est figé par le foncteur associé au type. Les résultats de déforestation obtenus pour ces programmes plus complexes sont moins probants. La normalisation du second ordre nécessite une expression particulière des *folds* pris en compte (fonctions de zéro-remplacement), ou peut aboutir à des impasses (exceptions dans l'algorithme de normalisation étendu). Dans certains cas, deux fonctions décrites en premier ordre ne peuvent être fusionnées qu'en une fonction d'ordre supérieur qui ne peut être ramenée au premier ordre. Pour la composition descriptionnelle, l'utilisation de la puissance d'expression procurée par les attributs hérités permet une déforestation efficace.

Cependant, pour une large utilisation, le formalisme des grammaires attribuées souffre de ses origines liées aux types et à la structure des données. La différence entre les spécifications par fonctions d'ordre supérieur et par attributs hérités rend la comparaison des programmes d'entrée plus difficile. Les spécifications de grammaires attribuées sont suffisamment différentes des programmes fonctionnels classiques pour accentuer la difficulté de comparaison de la composition descriptionnelle avec les autres méthodes de déforestation et pour interdire son application directe sur des programmes fonctionnels classiques.

C'est également un problème dont souffraient les hylomorphismes, même sous forme de triplet : comment exiger d'un programmeur qu'il spécifie ses fonctions dans un tel formalisme, peu habituel sinon rébarbatif ? Le problème fut en partie résolu par Hu *et al.* dans [HIT96b, OHIT97], grâce à une transformation permettant de déterminer des hylomorphismes sous forme de triplets à partir d'une définition de fonction récursive classique. Malheureusement, il semble que la fusion effectuée sur des programmes décrits avec de tels hylomorphismes, tout comme pour les folds, n'aboutit pas toujours à une déforestation suffisante. Les programmes tels que la double inversion de liste ou l'inversion de la liste des feuilles d'un arbre ne sont en effet pas correctement déforestés par ces méthodes, même lorsqu'ils sont acceptés par les algorithmes de fusion d'hylomorphismes ou de normalisation de *folds*. La raison de ces points d'achoppement semble être à la fois l'encapsulation de certains calculs dans des fonctions (cachant les constructeurs à déforester) et la rigidité imposée par les foncteurs qui guident la récursion des calculs et des transformations.

Objectifs

En revanche, les qualités déclaratives des grammaires attribuées, dues à la fois au concept d'attribut hérité et au nommage explicite des occurrences d'attributs, permettent à la composition descriptionnelle, qui exploite au maximum ces caractéristiques, de déforester pleinement ces classes de programmes. Malheureusement, les programmes pris en compte par la composition descriptionnelle doivent être écrits en grammaire attribuée. De plus, dans les différents formalismes fonctionnels, l'intégration aux algorithmes de déforestation d'une certaine forme d'évaluation partielle paraît judicieuse et, jusqu'à présent, la composition descriptionnelle n'en fait pas l'usage.

Les remarques issues de la longue phase d'étude des formalismes et méthodes existants et inspirées par leurs comparaisons, ont permis d'établir des critères que devait remplir une méthode de déforestation plus efficace que celles existant jusqu'alors. La composition descriptionnelle semble intrinsèquement plus puissante, de par son indépendance par rapport aux types manipulés et par rapport aux méthodes d'évaluation. Par contre, il est indispensable qu'elle puisse prendre en charge des définitions de fonctions récursives classiques, d'une part pour mieux comparer ses effets et d'autre part pour la rendre plus largement utilisable. De plus, il semblait bénéfique de pouvoir lui incorporer un mécanisme d'évaluation partielle locale et symbolique afin de détecter des simplifications lors de la transformation, tout en restant indépendant des méthodes ou des ordres d'évaluation.

Ces remarques motivent les travaux qui sont présentés dans les chapitres suivants.

Chapitre 7

Transformation de programmes fonctionnels en grammaires attribuées

Pour pouvoir appliquer la composition descriptionnelle aux programmes fonctionnels et d'une manière générale pour faciliter le transfert des techniques d'analyses et de transformation des grammaires attribuées aux programmes fonctionnels, il devient nécessaire de disposer d'une transformation automatisable de programmes fonctionnels en grammaires attribuées. C'est l'objet de cette section.

L'idée intuitive de la transformation FP-to-AG d'un programme fonctionnel en sa notation par grammaires attribuées [1] est la suivante. Chaque terme fonctionnel associé à un *pattern* (filtre syntaxique correspondant à un constructeur de type) doit être démonté en un ensemble d'équations orientées, qui seront appelées des règles sémantiques. Les paramètres du programme fonctionnel deviendront des attributs explicites, attachés à des variables du *pattern*, formant ainsi des occurrences d'attributs, définis par les règles sémantiques. Les appels récursifs explicites deviennent donc implicites sur la structure sous-jacente du type de donnée et les règles sémantiques rendent le flot des données explicite.

La transformation FP-to-AG est décomposée en deux principales étapes : la *transformation préliminaire* et l'*évaluation symbolique du profil.*

7.1 Langage fonctionnel

Afin de présenter les étapes de base de la transformation FP-to-AG de manière simple et claire, je restreins délibérément ici la présentation à une sous-classe des programmes fonctionnels du premier ordre où les λ-abstractions ne seront pas autorisées.

La syntaxe du langage fonctionnel pris en compte est présentée à la figure 7.1. Par ailleurs, le *pattern matching* imbriqué n'est pas autorisé, mais des méthodes permettant de transformer ces expressions en plusieurs fonctions séparées sont connues et facilement applicables. De plus,

1. Cette notation *fonctionnelle* de grammaires attribuées n'est pas la plus classique, mais celle qui est précisément introduite à la section 2.1.2 pour faciliter ce rapprochement. De plus, c'est une notation *minimale* pour la présentation qui est faite ici.

$$
\begin{array}{lcl}
prog & ::= & \overline{def} \\
def & ::= & \texttt{let}\ f\ \overline{x} = exp \\
 & | & \texttt{let}\ f\ \overline{x} = \texttt{case}\ x_k\ \texttt{of}\ \{pat \rightarrow exp\}^{+} \\
pat & ::= & c\ \overline{x} \\
exp & ::= & Constante \\
 & | & x \in Variables \\
 & | & f\ \overline{exp}
\end{array}
$$

FIG. 7.1: *Langage fonctionnel*

$$
\begin{array}{ll}
\texttt{let}\ rev\ x\ l = \texttt{case}\ x\ \texttt{of} & \texttt{let}\ flat\ t\ l = \texttt{case}\ t\ \texttt{of} \\
\quad cons\ head\ tail \rightarrow & \quad node\ left\ right \rightarrow \\
\qquad rev\ tail\ (cons\ head\ l) & \qquad flat\ left\ (flat\ right\ l) \\
\quad nil \rightarrow l & \quad leaf\ \ n \rightarrow cons\ n\ l
\end{array}
$$

FIG. 7.2: *Exemples de programmes fonctionnels*

les instructions `if-then-else` peuvent être prises en compte grâce aux grammaires attribuées dynamiques [PRJD96a] ; pour éviter les confusions, je ne développerai pas ces points dans cette présentation. Remarquons cependant que les autres formalismes de déforestation traitent des classes de programmes similaires (cf. le système HYLO [OHIT97]).

Les fonctions *rev* et *flat*, présentées à la figure 7.2, illustrent le langage fonctionnel que nous prenons en compte.

Je suppose ici que les transformations qui seront considérées prennent en entrée des programmes fonctionnels bien typés ; cela permet de déduire des informations pour la grammaire attribuée produite correspondante. Par exemple, la classe (synthétisée ou héritée) et le type des attributs d'une grammaire attribuée produite à partir d'un programme fonctionnel bien typé peuvent être déduits directement à partir du programme d'entrée.

Dans les algorithmes de transformations présentés dans ce qui suit les notations et conventions suivantes seront utilisées :

$\stackrel{def}{=}$	:	une définition locale dans un algorithme
$\overline{x}$	:	un n-uple $x_1, \ldots, x_n$
$x.a = exp$	:	une règle sémantique définissant $x.a$
$e[x := y]$	:	substitution de x par y dans une expression e
Σ	:	un ensemble de règles sémantiques
Π	:	un *pattern* avec son ensemble de règles sémantiques associé
$\mathcal{C} \vdash A \Rightarrow B$	:	transformation de A en B dans le contexte $\mathcal{C}$
$\mathcal{E}[e]$	:	une expression qui contient e comme sous-expression.

$$\frac{\forall i \quad \overset{exp}{\vdash} a_i \Rightarrow b_i \quad ; \quad f \text{ est un nom de fonction}}{\overset{exp}{\vdash} (f\ \overline{a}) \Rightarrow (f\ \overline{b}).result} \qquad (App)$$

$$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{f, \{x_j\}_{j\neq k}, x_k \overset{pat}{\vdash} c\ \overline{y} \rightarrow e \Rightarrow c\ \overline{y} \rightarrow c.f = e'[x_k := c][x_j := c.x_j]_{\forall j\neq k}} \qquad (Pattern)$$

$$\frac{\begin{array}{l}\forall i \quad f, \{x_j\}_{j\neq k}, x_k \overset{pat}{\vdash} p_i \rightarrow e_i \Rightarrow \Pi_i \\ \overline{\Pi} \overset{def}{=} \left(\begin{array}{l} f\ \overline{x} \rightarrow \\ \quad f.result = x_k.f \\ \quad x_k.x_j = x_j \quad {\scriptstyle (\forall j\neq k)} \end{array}\right) \cup \Pi_i\end{array}}{\overset{let}{\vdash} \texttt{let } f\ \overline{x} = \texttt{case } x_k \texttt{ of } \overline{p \rightarrow e} \Rightarrow \texttt{aglet } f = \overline{\Pi}} \qquad (Let)$$

$$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{\overset{let}{\vdash} \texttt{let } f\ \overline{x} = e \Rightarrow \texttt{aglet } f = f\ \overline{x} \rightarrow f.result = e'} \qquad (Let')$$

$\overset{exp}{\vdash} e \Rightarrow e'$	signifie que l'expression e est traduite en l'expression e'.
$env \overset{pat}{\vdash} p \rightarrow e \Rightarrow p \rightarrow \mathcal{R}$	signifie que l'expression associée au *pattern* p est traduite en l'ensemble des règles sémantiques $\mathcal{R}$, dans le contexte de l'environnement *env*.
$\overset{let}{\vdash} \mathcal{D} \Rightarrow \mathcal{B}$	signifie que la définition de fonction $\mathcal{D}$ est traduite en un bloc $\mathcal{B}$.

FIG. 7.3: *Transformation préliminaire*

7.2 Schéma de récursion *versus* système d'équation

7.2.1 Transformation préliminaire

Le but de la transformation préliminaire, présentée à la figure 7.3, est de déterminer la forme générale de la future grammaire attribuée. Elle introduit le profil de la grammaire attribuée, avec ses règles sémantiques, ainsi qu'une unique règle sémantique pour chacun des *patterns* des autres constructeurs.

Je vais commenter ces règles de transformation et détailler l'exécution de cette transformation sur l'exemple de la fonction *flat*, présentée à la figure 7.2.

Pour toute fonction donnée en entrée, l'attribut *result* est défini comme un attribut synthétisé du profil de la grammaire attribuée correspondante. Il contient le résultat de la fonction (règle *Let'* de la figure 7.3). Dans le cas intéressant où la définition de fonction possède une instruction `case`, le résultat est calculé par des attributs portés par la variable syntaxique, soumise au *pattern matching*, que j'appelle par barbarisme la variable *pattern matchée* ; en particulier, le résultat $f.result$ est défini par la valeur d'un attribut synthétisé sur la variable *pattern matchée* et cet attribut porte le même nom que la fonction, c'est-à-dire $x_k.f$ (règle *Let*). Les autres arguments sont traduits en des règles sémantiques définissant des attributs hérités (ayant le même nom que les arguments correspondant) attachés à la variable *pattern matchée*, c'est-à-dire $x_k.x_j = x_j$ (règle *Let*).

C'est le cas dans l'exemple de la fonction *flat* pour laquelle l'application de la règle *Let* est présentée ci-dessous.

$$\frac{\begin{array}{l} \forall i \quad \mathit{flat},\ l,\ t \overset{pat}{\vdash} p_i \rightarrow e_i \ \Rightarrow\ \Pi_i \\ \overline{\Pi} \overset{def}{=} \begin{pmatrix} \mathit{flat}\ t\ l \rightarrow \\ \quad \mathit{flat.result} = t.\mathit{flat} \\ \quad t.l = l \end{pmatrix} \cup \Pi_i \end{array}}{\overset{let}{\vdash} \texttt{let}\ \mathit{flat}\ t\ l = \texttt{case}\ t\ \texttt{of}\ \overline{p \rightarrow e} \ \Rightarrow\ \texttt{aglet}\ \mathit{flat} = \overline{\Pi}} \quad (Let)$$

La transformation des expressions associées aux *patterns* $\overline{p \rightarrow e}$ en des ensembles de règles sémantiques se fait en plusieurs phases. La première est prise en charge par la règle *Pattern*. Elle rend explicite l'attachement des attributs aux variables de types, en remplaçant la variable *pattern matchée* de la fonction (x_k) par le nom de la variable de type du constructeur du *pattern* courant (c). De même, les arguments de la fonctions (x_j) sont renommés par les occurrences d'attributs hérités qui leurs sont associés ($c.x_j$).

Par exemple, pour les *patterns* de la fonction *flat*, toute apparition du terme t, paramètre de la fonction représentant l'arbre binaire actuellement *pattern matché*, est remplacée par le nom du constructeur courant. De même, toute apparition de la variable l, argument de la fonction, devra être remplacée par l'occurrence d'attribut qui la représente et qui est portée par la variable du constructeur du *pattern* courant (c'est-à-dire $\mathit{leaf}.l$ ou $\mathit{node}.l$).

$$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{\mathit{flat},\ l,\ t \overset{pat}{\vdash} \begin{array}{l} \mathit{leaf}\ \ n \rightarrow e \\ \Rightarrow\ \mathit{leaf}\ \ n \rightarrow \\ \qquad \mathit{leaf.flat} = e'[t := \mathit{leaf}][l := \mathit{leaf}.l] \end{array}} \quad (Pattern)$$

$$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{\mathit{flat},\ l,\ t \overset{pat}{\vdash} \begin{array}{l} \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow e \\ \Rightarrow\ \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\ \qquad \mathit{node.flat} = e'[t := \mathit{node}][l := \mathit{node}.l] \end{array}} \quad (Pattern)$$

Restent alors à transformer les expressions e subsistant dans ces termes en expressions e' attendues. Chaque appel de fonction ($f\ \overline{a}$) est traduit en une notation *pointée* $(f\ \overline{b}).result$ par la règle *App*. Cette règle fait la différence entre les appels de fonctions et les appels de constructeurs de type. Cette distinction est rendue possible à partir des informations issues du typage du programme fonctionnel pris en entrée de la transformation. Chaque expression apparaissant dans un *pattern* est transformée en une seule règle sémantique qui définit l'attribut synthétisé calculant le résultat (règle *App*). Ceci implique des renommages, pris en charge par la règle *Pattern*.

Par exemple, sur la fonction *flat*, l'expression correspondant au *pattern* ($leaf\ n \rightarrow$) est ($cons\ n\ l$). Dans ce cas, puisque le *cons* est un constructeur de type, l'expression reste inchangée et seul le renommage induit par la règle *Pattern* sera effectué. Par contre, dans le cas du *pattern* ($node\ left\ right \rightarrow$), l'expression ($flat\ left\ (flat\ right\ l)$) est un appel de fonction. Dans ce cas, la règle *App* doit être appliquée. Cette règle est appliquée avec la stratégie *en profondeur d'abord.* Elle commencera donc par être appliquée à tous les paramètres sous-termes de l'expression principale. Comme *left* reste inchangé par la transformation, il reste à la même règle *App* à être appliquée sur le sous-terme ($flat\ right\ l$) qui est lui même un appel de fonction :

$$\frac{\overset{exp}{\vdash} \begin{array}{c} right \Rightarrow right \\ l \Rightarrow l \end{array} \quad \mathit{flat} \text{ est un nom de fonction}}{\overset{exp}{\vdash} (\mathit{flat}\ right\ l) \Rightarrow (\mathit{flat}\ right\ l).result} \quad (App)$$

Ceci permet d'appliquer la règle *App* à l'expression principale du *pattern* ($node\ left\ right \rightarrow$) :

$$\frac{\overset{exp}{\vdash} \begin{array}{c} left \Rightarrow left \\ (flat\ right\ l) \Rightarrow (flat\ right\ l).result \end{array} \quad \mathit{flat} \text{ est un nom de fonction}}{\overset{exp}{\vdash} \mathit{flat}\ left\ (\mathit{flat}\ right\ l) \Rightarrow (\mathit{flat}\ left\ (\mathit{flat}\ right\ l).result).result} \quad (App)$$

Finalement, l'application du renommage déterminé lors de l'application des règles *Pattern* permet d'obtenir le résultat de la transformation préliminaire pour la fonction *flat* :

```
aglet flat =
  flat t l →
    flat.result = t.flat
    t.l = l
  node left right →
    node.flat = (flat left (flat right node.l).result).result
  leaf n →
    leaf.flat = cons n leaf.l
```

7.2.2 Évaluation symbolique du profil

Le résultat obtenu par la transformation préliminaire n'est pas encore une véritable grammaire attribuée. Chaque définition de fonction dans le programme initial doit être transformée en un bloc (cf. Figure 2.6) qui contient le profil de la fonction et ses *patterns* correspondants. Cependant, les appels récursifs explicites ont déjà été transformés sous la forme $(f\ \overline{a}).result$. Ces dernières expressions doivent maintenant être traduites en un ensemble de règles sémantiques, en *cassant* les récursions explicites par le nommage des attributs et leur attachement aux variables de *pattern*. Les règles sémantiques ainsi produites définissent alors implicitement la récursion, à la manière des grammaires attribuées.

Ces diverses opérations sont effectuées par un algorithme appelé *évaluation symbolique du profil*, qui est présenté dans la figure 7.4.

$$\frac{\left(\begin{array}{l} f\ \overline{x} \rightarrow \\ \quad f.result = \varphi \\ \quad \Sigma_f \end{array}\right) \in \mathcal{P} \qquad \begin{array}{l} \sigma \stackrel{def}{=} [x_i := a_i] \\ \Sigma \stackrel{def}{=} \left\{\begin{array}{l} u = \mathcal{E}[\sigma(\varphi)] \\ \sigma(\Sigma_f) \\ \Sigma_{aux} \end{array}\right. \end{array} \qquad Check(c, f, \Sigma)}{\mathcal{P} \vdash \left(\begin{array}{l} c\ \overline{y} \rightarrow \\ \quad u = \mathcal{E}[(f\ \overline{a}).result] \\ \quad \Sigma_{aux} \end{array}\right) \Rightarrow \left(\begin{array}{l} c\ \overline{y} \rightarrow \\ \quad \Sigma \end{array}\right)} \quad (PSE)$$

$\mathcal{P} \vdash p \rightarrow \Sigma_1 \ \Rightarrow\ p \rightarrow \Sigma_2$ signifie que dans le programme $\mathcal{P}$, l'ensemble des équations Σ_1 d'un *pattern* p est transformé en Σ_2.

FIG. 7.4: *Évaluation symbolique du profil*

Partout où apparaît une expression $(f\ \overline{a}).result$, l'évaluation symbolique du profil *PSE* y projette les règles sémantiques du profil f de la grammaire attribuée. L'application de cette transformation doit être faite avec une stratégie d'application *en profondeur d'abord*.

Le prédicat *Check* assure que la grammaire attribuée résultante est bien formée. Essentiellement, il vérifie que chaque occurrence d'attribut est définie une fois et une seule. C'est en général le cas puisque les paramètres des programmes fonctionnels acceptés en entrée de la transformation sont bien définis. Cependant, le prédicat *Check* interdit certains termes non linéaires tels que $g\ (f\ x\ 1)\ (f\ x\ 2)$.

De plus, pour l'instant et dans le cadre de la transformation FP-to-AG, les termes tels que $(x.a).b$ ne sont pas autorisés dans l'évaluation symbolique du profil. Je montrerai dans la section 8.2 qu'ils sont importants : ils seront alors autorisés pour pouvoir être eux-mêmes transformés.

D'une façon générale, partout où le prédicat $Check(c, f, \Sigma)$ n'est pas vérifié, l'expression $(f\ \overline{a}).result$ est simplement remplacée par l'appel de la fonction correspondante $(f\ a)$.

Dans l'exemple précédent de la fonction *flat*, la règle sémantique qui est associée au *pattern* (*node left right* $\rightarrow$) est la suivante :

$$node.flat = (flat\ left\ (flat\ right\ node.l).result).result$$

Les deux applications[2] successives de l'évaluation symbolique du profil sur cette règle sémantique sont présentées par les figures 7.5 et 7.6.

L'application complète de l'évaluation symbolique du profil pour la fonction *flat* conduit à la grammaire attribuée bien formée présentée dans la figure 7.7. L'application successive de la transformation préliminaire et de l'évaluation symbolique du profil génère donc une véritable grammaire attribuée. Cette traduction sera appelée la transformation FP-to-AG.

2. Les *termes* soulignés mettent en évidence les endroits où la règle est appliquée.

$$\dfrac{\begin{array}{l} \left(\begin{array}{l} \mathit{flat}\ t\ l \rightarrow \\ \quad \mathit{flat}.\mathit{result} = t.\mathit{flat} \\ \quad t.l = l \end{array}\right) \in \mathcal{P} \\ \sigma \overset{def}{=} [t := \mathit{right}][l := \mathit{node}.l] \\ \Sigma \overset{def}{=} \left\{\begin{array}{l} \mathit{node}.\mathit{flat} = (\mathit{flat}\ \mathit{left}\ \mathit{right}.\mathit{flat}).\mathit{result} \\ \mathit{right}.l = \mathit{node}.l \end{array}\right. \\ \mathit{Check}(\mathit{node}, \mathit{flat}, \Sigma) \end{array}}{\mathit{flat} \vdash \begin{array}{l} \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\ \quad \mathit{node}.\mathit{flat} = (\mathit{flat}\ \mathit{left}\ (\underline{\mathit{flat}}\ \mathit{right}\ \mathit{node}.l).\underline{\mathit{result}}).\mathit{result} \\ \Rightarrow \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \Sigma \end{array}} \quad (PSE)$$

FIG. 7.5: *Premier pas d'application de PSE pour le pattern node left right*

$$\dfrac{\begin{array}{l} \left(\begin{array}{l} \mathit{flat}\ t\ l \rightarrow \\ \quad \mathit{flat}.\mathit{result} = t.\mathit{flat} \\ \quad t.l = l \end{array}\right) \in \mathcal{P} \\ \sigma \overset{def}{=} [t := \mathit{left}][l := \mathit{right}.\mathit{flat}] \\ \Sigma \overset{def}{=} \left\{\begin{array}{l} \mathit{node}.\mathit{flat} = \mathit{left}.\mathit{flat} \\ \mathit{left}.l = \mathit{right}.\mathit{flat} \end{array}\right. \\ \mathit{Check}(\mathit{node}, \mathit{flat}, \Sigma) \end{array}}{\mathit{flat} \vdash \begin{array}{l} \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\ \quad \mathit{node}.\mathit{flat} = (\underline{\mathit{flat}}\ \mathit{left}\ \mathit{right}.\mathit{flat}).\underline{\mathit{result}} \\ \quad \mathit{right}.l = \mathit{node}.l \\ \Rightarrow \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \Sigma \end{array}} \quad (PSE)$$

FIG. 7.6: *Deuxième pas d'application de PSE pour le pattern node left right*

$$\begin{array}{l} \texttt{aglet}\ \mathit{flat} = \\ \quad \mathit{flat}\ t\ l \rightarrow \\ \quad\quad \mathit{flat}.\mathit{result} = t.\mathit{flat} \\ \quad\quad t.l = l \\ \quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\ \quad\quad \mathit{node}.\mathit{flat} = \mathit{left}.\mathit{flat} \\ \quad\quad \mathit{left}.l = \mathit{right}.\mathit{flat} \\ \quad\quad \mathit{right}.l = \mathit{node}.l \\ \quad \mathit{leaf}\ n \rightarrow \\ \quad\quad \mathit{leaf}.\mathit{flat} = \mathit{cons}\ n\ \mathit{leaf}.l \end{array}$$

FIG. 7.7: *Grammaire attribuée obtenue par FP-to-AG pour la fonction flat*

Le coût de cette transformation

Cette transformation n'est évidemment pas gratuite. Néanmoins, son coût en pratique reste très raisonnable. En effet, la complexité de la transformation préliminaire est linéaire en fonction de la profondeur des termes du programme fonctionnel accepté en entrée. Pour chaque définition de fonction, un bloc est créé pour le profil, avec autant de règles sémantiques que la fonction accepte de paramètres. Pour chaque cas du filtrage de la définition fonctionnelle, un nouveau bloc est créé, comprenant une règle de définition de l'occurrence d'attribut représentant la valeur de la fonction dans ce cas. C'est alors dans ces règles qu'apparaissent généralement les appels de fonctions qui devront être démontés par l'évaluation symbolique du profil. Plus le terme correspondant à un cas de filtrage contient d'appels de fonctions récursifs, plus le pas de *PSE* doit être appliqué. C'est en ce sens que la complexité d'application de FP-to-AG dépend linéairement de la taille et de la profondeur (en nombre d'appels de fonctions) du terme du programme fonctionnel initial.

Chapitre 8

Composition symbolique

Il est désormais possible d'appliquer les méthodes de déforestation spécifiques aux grammaires attribuées sur les programmes fonctionnels transformables, grâce à la transformation FP-to-AG, en grammaires attribuées équivalentes. Je présente dans cette section une nouvelle méthode de déforestation qui étend le pouvoir et le champ d'application de la composition descriptionnelle, mais qui est fondée sur le même principe de projection de règles sémantiques. Je commence pour cela par étendre l'algorithme d'évaluation symbolique du profil déjà utilisé dans FP-to-AG, qui sera largement utilisé dans la composition symbolique. Ensuite, je présente le cœur de la transformation, comme une extension de la composition descriptionnelle par la projection de schémas de calculs. Finalement, je montre en quoi cette nouvelle technique de déforestation est plus efficace que les méthodes fonctionnelles développées jusqu'à présent.

Il est important de noter ici, en avertissement, que même si les résultats fournis par la composition symbolique sont effectivement des grammaires attribuées, les objets qui sont manipulés au cours des diverses transformations sont plus des *blocs* de grammaires attribuées que des grammaires attribuées complètes. De plus, les expressions de la forme $(x.a).b$, qui sont classiquement interdites dans les grammaires attribuées et qui ont été interdites en première approche dans la transformation FP-to-AG qui vient d'être présentée, seront ici temporairement autorisées. En effet, ces expressions sont représentatives des compositions qu'il est souhaitable d'éliminer dans le cadre de la déforestation. Elles seront donc utiles dans le mécanisme de la composition symbolique, mais n'apparaîtront plus dans les résultats.

8.1 Évaluation symbolique généralisée

Avant de présenter la composition symbolique, je commence donc par généraliser l'évaluation symbolique du profil (*PSE*) en une nouvelle transformation appelée simplement *évaluation symbolique (SE)*, qui incorpore au précédent mécanisme l'évaluation partielle de certains termes finis. L'idée de cette évaluation symbolique est de projeter récursivement des règles sémantiques sur des termes finis, non plus seulement au niveau des profils des fonctions, mais plus généralement partout où apparaissent des compositions, permettant ainsi d'éliminer les attributs intermédiaires qui sont définis et utilisés dans les ensembles de règles sémantiques produits par les transformations.

L'algorithme de transformation de l'évaluation symbolique est présenté à la figure 8.1.

$$\frac{\begin{pmatrix} f\ \overline{x} \rightarrow \\ \quad f.w = \varphi \\ \quad \Sigma_f \end{pmatrix} \in \mathcal{P} \qquad \begin{array}{l} \sigma \stackrel{def}{=} [x_i := a_i][f.h := \varphi_h] \\ \Sigma \stackrel{def}{=} \left\{ \begin{array}{l} c.u = \mathcal{E}[\sigma(\varphi)] \\ \sigma(\Sigma_f) \\ \Sigma_{aux} \end{array} \right. \end{array} \qquad Check(c, f, \Sigma)}{\mathcal{P} \vdash \begin{pmatrix} c\ \overline{y} \rightarrow \\ \quad c.u = \mathcal{E}[(f\ \overline{a}).w] \\ \quad (f\ \overline{a}).h = \varphi_h \\ \quad \Sigma_{aux} \end{pmatrix} \Rightarrow \begin{pmatrix} c\ \overline{y} \rightarrow \\ \Sigma \end{pmatrix}} \quad (SE)$$

FIG. 8.1: *Évaluation symbolique (SE)*

Pour illustrer l'utilisation de cette évaluation symbolique et pour montrer en quoi elle effectue de l'évaluation partielle, je considère l'expression g définie par un appel à la fonction *rev* :

$$\texttt{let}\ g = rev\ (cons\ a\ (cons\ b\ nil))\ nil$$

La transformation FP-to-AG appliquée à cette fonction produit, grâce à la règle *Let'* de la transformation préliminaire (figure 7.3) puis par l'application de l'évaluation symbolique du profil (*PSE*, figure 7.4), la grammaire attribuée suivante :

$$\begin{array}{l} \texttt{aglet}\ g = \\ \quad g \rightarrow \\ \quad\quad g.result = (cons\ a\ (cons\ b\ nil)).rev \\ \quad\quad (cons\ a\ (cons\ b\ nil)).h = nil \end{array}$$

Les deux règles sémantiques ci-dessus peuvent alors subir les transformations spécifiées par l'évaluation symbolique (*SE*, figure 8.1). Le premier pas d'application de *SE* sur ces règles est détaillé figure 8.2. Deux autres pas de cette même règle de transformation, présentés figures 8.3 et 8.4, conduisent au terme évalué suivant :

$$g.result = (cons\ b\ (cons\ a\ nil))$$

L'évaluation symbolique effectue donc de l'**évaluation partielle** sur les termes finis.

8.2 Projection des schémas de calculs

Je vais maintenant présenter une transformation qui s'inspire des principes de la composition descriptionnelle mais qui met à profit le formalisme fonctionnel adopté pour les grammaires attribuées, ainsi que de l'évaluation partielle incluse dans l'évaluation symbolique précédemment généralisée.

L'idée

Pour donner l'idée de cette transformation, je reprends ici l'exemple illustratif de composition de la fonction *rev* qui inverse une liste avec la fonction *flat* qui crée la liste des feuilles d'un arbre (cf. figure 7.2). Cette composition aboutit bien sûr à la liste des feuilles de l'arbre,

$$\dfrac{\begin{array}{l} \left(\begin{array}{l} cons\ head\ tail \rightarrow \\ \quad cons.rev = tail.rev \\ \quad tail.h = cons\ head\ cons.h \end{array}\right) \in \mathcal{P} \\ \sigma \stackrel{def}{=} [head := a][tail := (cons\ b\ nil)][cons.h := nil] \\ \Sigma \stackrel{def}{=} \left\{ \begin{array}{l} g.result = (cons\ b\ nil).rev \\ (cons\ b\ nil).h = (cons\ a\ nil) \end{array}\right. \\ Check(g, cons, \Sigma) \end{array}}{\mathcal{P} \vdash \begin{array}{l} g \rightarrow \\ \quad g.result = (cons\ a\ (cons\ b\ nil)).rev \\ \quad (cons\ a\ (cons\ b\ nil)).h = nil \end{array} \Rightarrow g \rightarrow \Sigma} \quad (SE)$$

FIG. 8.2: *Premier pas d'application de SE*

$$\dfrac{\begin{array}{l} \left(\begin{array}{l} cons\ head\ tail \rightarrow \\ \quad cons.rev = tail.rev \\ \quad tail.h = cons\ head\ cons.h \end{array}\right) \in \mathcal{P} \\ \sigma \stackrel{def}{=} [head := b][tail := nil][cons.h := (cons\ a\ nil)] \\ \Sigma \stackrel{def}{=} \left\{ \begin{array}{l} g.result = (nil).rev \\ (nil).h = (cons\ b\ (cons\ a\ nil)) \end{array}\right. \\ Check(c, cons, \Sigma) \end{array}}{\mathcal{P} \vdash \begin{array}{l} g \rightarrow \\ \quad g.result = (cons\ b\ nil).rev \\ \quad (cons\ b\ nil).h = (cons\ a\ nil) \end{array} \Rightarrow g \rightarrow \Sigma} \quad (SE)$$

FIG. 8.3: *Second pas d'application de SE*

$$\dfrac{\begin{array}{l} \left(\begin{array}{l} nil \rightarrow \\ \quad nil.rev = nil.h \end{array}\right) \in \mathcal{P} \\ \sigma \stackrel{def}{=} [nil.h := (cons\ b\ (cons\ a\ nil))] \\ \Sigma \stackrel{def}{=} \left\{ \ g.result = (cons\ b\ (cons\ a\ nil)) \right. \\ Check(g, nil, \Sigma) \end{array}}{\mathcal{P} \vdash \begin{array}{l} g \rightarrow \\ \quad g.result = (nil).rev \\ \quad (nil).h = (cons\ b\ (cons\ a\ nil)) \end{array} \Rightarrow g \rightarrow \Sigma} \quad (SE)$$

FIG. 8.4: *Troisième pas d'application de SE*

```
aglet rev =
  rev x h →
    rev.result = x.rev
    rev.h = h
  cons head tail →
    cons.rev = tail.rev
    tail.h = (cons head cons.h)
  nil →
    nil.rev = nil.h
```

```
aglet flat =
  flat t l →
    flat.result = t.flat
    t.l = l
  node left right →
    node.flat = left.flat
    left.l = right.flat
    right.l = node.l
  leaf n →
    leaf.flat = cons n leaf.l
```

FIG. 8.5: *Les grammaires attribuées rev et flat*

dans le sens inverse de celui décrit par *flat*, mais le but ici est de la transformer de sorte à ce qu'elle ne construise pas la première liste des feuilles, mais plutôt qu'elle projette le calcul de son inversion directement sur le parcours de la structure de l'arbre.

Soit donc la définition de la fonction *revflat* :

$$\texttt{let}\ \mathit{revflat}\ t = (\mathit{rev}\ (\mathit{flat}\ t\ \mathit{nil})\ \mathit{nil})$$

De manière intuitive, dans le contexte des grammaires attribuées correspondant à ces fonctions et générées par FP-to-AG (cf. figure 8.5), cette composition implique les deux ensembles d'attributs suivants :

$$\mathit{Att}_{\mathit{flat}} = \{\mathit{flat}\ ,\ l\} \quad \text{and} \quad \mathit{Att}_{\mathit{rev}} = \{\mathit{rev}\ ,\ h\}$$

Le pas de projection

Plus généralement, considérons une grammaire attribuée $\mathcal{F}$ (par exemple *flat*), produisant une structure de donnée intermédiaire qui doit être consommée par une grammaire attribuée $\mathcal{G}$ (par exemple *rev*). Deux ensembles d'attributs sont impliqués dans cette composition. Le premier ensemble, $\mathit{Att}_{\mathcal{F}}$, contient tous les attributs utilisés pour construire la structure de donnée intermédiaire. Le second, $\mathit{Att}_{\mathcal{G}}$, contient les attributs de $\mathcal{G}$.

Tout comme dans la composition descriptionnelle, l'idée de la composition symbolique est de projeter les attributs de $\mathit{Att}_{\mathcal{G}}$ ($\mathit{Att}_{\mathit{rev}}$) partout où un attribut de $\mathit{Att}_{\mathcal{F}}$ ($\mathit{Att}_{\mathit{flat}}$) est défini. Cette opération globale *transporte* les équations spécifiant le calcul sur la structure intermédiaire à l'intérieur même de sa construction. Le pas de base de cette projection est présenté à la figure 8.6. La transformation *Proj* fait apparaître, par cette projection, des expressions qui peuvent ensuite être traitées par l'évaluation symbolique (*SE*, figure 8.1). Celle-ci se chargera d'éliminer les constructeurs inutiles de la structure intermédiaire.

C'est le principe de base de la déforestation proposée par la composition symbolique.

Les sites d'applications

Il reste cependant une chose à déterminer : comment trouver les sites d'application des pas de projection *Proj* ? Comme je l'ai évoqué dans l'introduction de ce chapitre, le prédicat *Check*, qui interdisait dans l'évaluation symbolique du profil l'introduction d'expressions de la

$$\frac{a \in Att_{\mathcal{F}} \quad \overline{s} = Att\_S_{\mathcal{G}} \quad \overline{h} = Att\_H_{\mathcal{G}}}{Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash x.a = e \;\Rightarrow \left\{ \begin{array}{ll} (x.a).s = (e).s & \forall s \in \overline{s} \\ (e).h = (x.a).h & \forall h \in \overline{h} \end{array} \right.} \qquad (Proj)$$

$Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash eq \Rightarrow \Sigma$	signifie que, en considérant $\mathcal{G} \circ \mathcal{F}$, l'équation eq est transformée en l'ensemble d'équations Σ.
$Att\_S_{\mathcal{G}}$	est l'ensemble des attributs synthétisés de $Att_{\mathcal{G}}$.
$Att\_H_{\mathcal{G}}$	est l'ensemble des attributs hérités de $Att_{\mathcal{G}}$.

FIG. 8.6: *Le pas de projection (Proj)*

$$\begin{array}{l} revflat\ t \rightarrow \\ \quad revflat.result = (t.flat).rev \\ \quad (t.flat).h = nil \\ \quad \underline{t.l = nil} \quad * \end{array}$$

$$\begin{array}{l} node\ left\ right \rightarrow \\ \quad \underline{node.flat = left.flat} \\ \quad \underline{left.l = right.flat} \\ \quad \underline{right.l = node.l} \\ leaf\ n \rightarrow \\ \quad \underline{leaf.flat = cons\ n\ leaf.l} \quad * \end{array} \qquad \begin{array}{l} cons\ head\ tail \rightarrow \\ \quad cons.rev = tail.rev \\ \quad tail.h = cons\ head\ cons.h \\ nil \rightarrow \\ \quad nil.rev = nil.h \end{array}$$

FIG. 8.7: *Le bloc de grammaire attribuée correspondant au profil de revflat, avec les blocs correspondants à flat et à rev*

forme $(x.a).b$, va être temporairement assoupli. En fait, ces expressions sont représentatives des compositions qui doivent avoir lieu et elles indiquent précisément les sites où la déforestation doit être effectuée localement (comme $(t.flat).rev$). Je parle ici d'assouplissement *temporaire* du prédicat *Check* car, en sortie de la composition symbolique, après l'application de toutes les transformations, ces expressions n'apparaîtront plus dans la grammaire attribuée résultante.

Avec le prédicat *Check* assoupli et à partir de la définition de la fonction *revflat*, les applications successives de la transformation préliminaire et de l'évaluation symbolique du profil aboutissent aux blocs décrits à la figure 8.7. Dans les règles sémantiques des blocs participant à la construction de la structure intermédiaire, les sites d'applications des pas de projection sont soulignés et une étoile $(*)$ marque les constructions à déforester.

L'application du pas de projection *Proj* est détaillée à la figure 8.8 pour la règle sémantique

$$\frac{flat \in Att_{flat} \quad \begin{array}{l} \overline{s} = Att\_S_{rev} = \{rev\} \\ \overline{h} = Att\_H_{rev} = \{h\} \end{array}}{Att_{rev}, Att_{flat} \vdash \begin{array}{l} leaf.flat = cons\ n\ leaf.l \\ \Rightarrow \left\{ \begin{array}{l} (leaf.flat).rev = (cons\ n\ leaf.l).rev \\ (cons\ n\ cons.l).h = (leaf.flat).h \end{array} \right. \end{array}} \qquad (Proj)$$

FIG. 8.8: *Exemple d'application de Proj*

$$
\begin{array}{l}
\mathit{revflat}\ t \rightarrow \\
\qquad \mathit{revflat.result} = (t.\mathit{flat}).\mathit{rev} \\
\qquad (t.\mathit{flat}).h = \mathit{nil} \\
\qquad \left.\begin{array}{l}(t.l).\mathit{rev} = (\mathit{nil}).\mathit{rev} \\ (\mathit{nil}).h = (t.l).h\end{array}\right\} \text{ site d'application pour SE} \\
\mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\qquad (\mathit{node.flat}).\mathit{rev} = (\mathit{left.flat}).\mathit{rev} \\
\qquad (\mathit{left.flat}).h = (\mathit{node.flat}).h \\
\qquad (\mathit{left}.l).\mathit{rev} = (\mathit{right.flat}).\mathit{rev} \\
\qquad (\mathit{right.flat}).h = (\mathit{left}.l).h \\
\qquad (\mathit{right}.l).\mathit{rev} = (\mathit{node}.l).\mathit{rev} \\
\qquad (\mathit{node}.l).h = (\mathit{right}.l).h \\
\mathit{leaf}\ \ n \rightarrow \\
\qquad \left.\begin{array}{l}(\mathit{leaf.flat}).\mathit{rev} = (\mathit{cons}\ \ n\ \ \mathit{leaf}.l).\mathit{rev} \\ (\mathit{cons}\ \ n\ \ \mathit{leaf}.l).h = (\mathit{leaf.flat}).h\end{array}\right\} \text{ site d'application pour SE}
\end{array}
$$

FIG. 8.9: *Blocs produits par les différentes applications de Proj*

du pattern (*leaf* $n \rightarrow$). Après toutes les applications possibles de cette règle, les blocs produits sont présentés à la figure 8.9.

À ce stade de la transformation, l'évaluation symbolique peut être appliquée sur les sites annotés dans la figure 8.9, ce qui permet d'obtenir la véritable déforestation par l'élimination locale, dans les règles sémantiques, des constructeurs de la structure intermédiaire. Dans ce cas, de nouveaux attributs sont créés par un renommage des expressions $a.b$ en a_b lorsque $a \in Att_{\mathcal{F}}$ et $b \in Att_{\mathcal{G}}$. Plus précisément, les compositions repérées par les expressions $(x.a).b$ sont transformées dans leur version déforestée par $x.a\_b$. Cette dernière étape complète l'ensemble des transformations qui permettent de définir la composition symbolique :

$$\boxed{\mathit{Composition\ symbolique} = \mathit{Renommage} \ \circ\ (SE)\ \circ\ (Proj)}$$

Ainsi, pour la fonction *revflat* qui illustre les transformations, l'application de la composition symbolique permet d'obtenir la grammaire attribuée présentée à la figure 8.10. Quatre attributs ont été créés. La liste finale est construite grâce aux attributs *l_h* and *flat_h*. Une fois construite, cette liste est propagée grâce aux attributs *flat_rev* and *flat_h* en suivant, à l'envers, le parcours de sa construction avant d'être assignée à l'attribut *result*[1]. Ce parcours peut paraître inutile et je présenterai dans le chapitre 9 une optimisation statique, appelée *élimination des règles de copies*, qui permet de se débarrasser de ce simple *transport* d'attribut. Néanmoins, dès à présent, cette grammaire attribuée obtenue par composition symbolique ne construit plus la structure (liste) intermédiaire. Elle est donc déforestée.

Il est possible de voir cette grammaire attribuée comme un programme fonctionnel, correspondant à un évaluateur fonctionnel de la grammaire attribuée.

Les fonctions de visites, produites en sortie d'un générateur d'évaluateur tel que FNC-2 [PDRJ95a, Le 95], correspondant à la grammaire attribuée *revflat* déforestée sont présentées à

1. La notation *result* peut surprendre le lecteur, mais c'est une simplification, pour cet attribut particulier, de la notation *result_result*, qui est normalement générée par la phase de renommage de la composition symbolique.

$$
\begin{array}{l}
\texttt{aglet}\ \mathit{revflat} = \\
\quad \mathit{revflat}\ t \rightarrow \\
\qquad \mathit{revflat.result} = t.\mathit{flat\_rev} \\
\qquad t.\mathit{flat\_h} = \mathit{nil} \\
\qquad t.\mathit{l\_rev} = t.\mathit{l\_h} \\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\qquad \mathit{node.flat\_rev} = \mathit{left.flat\_rev} \\
\qquad \mathit{left.flat\_h} = \mathit{node.flat\_h} \\
\qquad \mathit{left.l\_rev} = \mathit{right.flat\_rev} \\
\qquad \mathit{right.flat\_h} = \mathit{left.l\_h} \\
\qquad \mathit{right.l\_rev} = \mathit{node.l\_rev} \\
\qquad \mathit{node.l\_h} = \mathit{right.l\_h} \\
\quad \mathit{leaf}\ \ n \rightarrow \\
\qquad \mathit{leaf.flat\_rev} = \mathit{leaf.l\_rev} \\
\qquad \mathit{leaf.l\_h} = \mathit{cons}\ \ n\ \ \mathit{leaf.flat\_h}
\end{array}
$$

FIG. 8.10: *La grammaire attribuée revflat déforestée par la composition symbolique*

la figure 8.11 [2]. La première fonction, *revflat* appelle deux sous-fonctions, la première, *revflat1*, construisant la structure de liste résultat du parcours de l'arbre de la droite vers la gauche, et la seconde, *revflat2*, se contentant de propager cette liste le long de l'arbre, de la gauche vers la droite.

Une autre façon de voir cette grammaire attribuée comme un programme fonctionnel est de considérer les travaux de Johnsson [Joh87], qui permettent de générer, à partir d'une grammaire attribuée non-circulaire, une fonction équivalente qui peut être évaluée dans un langage paresseux. Le principe général de ces travaux est de considérer que pour chaque production, une fonction paresseuse peut évaluer l'ensemble des attributs synthétisés du constructeur à partir de ses attributs hérités. La fonction qui est produite par cette méthode pour l'exemple de la grammaire attribuée déforestée *revflat* de la figure 8.10 est présentée à la figure 8.12, d'abord dans sa version de base, puis après quelques simplifications.

Remarques

Je viens d'illustrer la composition symbolique sur un exemple assez simple. Pour des programmes plus complexes, les transformations spécifiées peuvent parfois être interdites par le prédicat *Check* évitant que le résultat de cette transformation soit une grammaire attribuée mal formée. Les raisons sont les suivantes, qui correspondent essentiellement aux contraintes introduites par Giegerich et Ganzinger au sujet de la composition descriptionnelle des grammaires attribuées classiques.

Les pas de projection de la composition symbolique ne doivent être effectués que sur les termes qui sont *impliqués* dans la construction de la structure de données intermédiaire. C'est le problème de la détermination de l'ensemble $Att_{\mathcal{F}}$, qui correspond à la séparation évoquée par Ganzinger et Giegerich entre les attributs syntaxiques et sémantiques. Cette remarque nécessite en particulier que la construction de la structure de données intermédiaire

2. Une présentation de la technique de génération de telles séquences de visites à partir d'une grammaire attribuée est présentée à la section 2.2.4 et illustrée sur l'exemple *bird*.

```
let revflat t = revflat2 t (revflat1 t nil)

let revflat1 t l = case t of
  node left right →
     revflat1 right (revflat1 left l)
  leaf n →
     cons n l

let revflat2 t l = case t of
  node left right →
     revflat2 left (revflat2 right l)
  leaf n →
     l
```

FIG. 8.11: *Les fonctions de visites générées pour la grammaire attribuée revflat déforestée*

soit complètement visible et *accessible*.

De plus, pour obtenir une grammaire attribuée résultante bien formée par la composition symbolique, chaque occurrence d'attribut doit être définie une fois et une seule. Des problèmes peuvent donc arriver en présence de termes non linéaires.

Les contraintes énoncées par Ganzinger and Giegerich peuvent être utilisées en première approche pour résoudre ces problèmes. Néanmoins, le contexte particulier de programmes fonctionnels correctement typés, celui dans lequel se place la transformation FP-to-AG, permet de reformuler ces problèmes en termes d'analyses statiques du programme fonctionnel initial. En effet, les informations requises par le mécanisme de composition peuvent être déterminées séparément, à partir d'une analyse du programme d'entrée. Même si cette analyse reste un problème ouvert intéressant qui n'est pas étudié dans cette thèse, sa prise en compte indépendante ne remet pas en question la composition symbolique.

Le coût de la composition symbolique

La complexité de la composition symbolique est plus délicate à estimer que celle de la transformation FP-to-AG présentée dans le chapitre précédent. Il est cependant possible de faire quelques remarques.

Tout d'abord, il est important de noter que la généralisation de l'évaluation symbolique du profil (*PSE*) en l'évaluation symbolique (*SE*) utilisée dans la composition symbolique implique des répercussions sur la complexité qui sont de deux ordres. Premièrement, elle implique la transformation de plusieurs règles et non plus une seule puisqu'elle doit prendre en compte les attributs hérités ; ce point conserve la linéarité mais augmente le facteur. Deuxièmement, elle peut être utilisée comme mécanisme d'évaluation partielle. En tant que telle, la complexité du terme à évaluer partiellement est répercutée sur la complexité de la transformation. En pratique, il est nécessaire de limiter cette utilisation de la règle *SE* pour éviter que cette transformation ne boucle infiniment dans le cas où, par exemple, elle tente d'évaluer partiellement l'inversion d'une liste infinie.

La complexité de la règle de projection *Proj* est due à la création de nouveaux attributs et des règles sémantiques qui les définissent. Essentiellement, elle est la même que celle de

Version déduite directement d'après la grammaire attribuée

```
let revflat t =
    let
       (flat_rev, l_h) = revflatlazy t l_rev flat_h
       flat_h = nil
       l_rev = l_h
    in flat_rev

let revflatlazy t l_rev flat_h = case t of
  node left right →
    let
       (flat_rev_left, l_h_left) = revflatlazy left l_rev_left flat_h_left
       (flat_rev_right, l_h_right) = revflatlazy right l_rev_right flat_h_right
       flat_rev = flat_rev_left
       flat_h_left = flat_h
       l_rev_left = flat_rev_right
       flat_h_right = l_h_left
       l_rev_right = l_rev
       l_h = l_h_right
    in (flat_rev, l_h)
  leaf n →
    let
       flat_rev = l_rev
       l_h = cons n flat_h
    in (flat_rev, l_h)
```

Version après simplifications

```
let revflat t = flat_rev in (flat_rev, l_h) = revflatlazy t l_h nil

let revflatlazy t l_rev flat_h = case t of
  node left right →
    let
       (flat_rev_left, l_h_left) = revflatlazy left flat_rev_right flat_h
       (flat_rev_right, l_h_right) = revflatlazy right l_rev l_h_left
    in (flat_rev_left, l_h_right)
  leaf n →
    (l_rev, cons n flat_h)
```

FIG. 8.12: *La fonction paresseuse correspondant à la grammaire attribuée revflat déforestée*

la composition descriptionnelle. Pour la composition d'une grammaire attribuée possédant n attributs avec une grammaire attribuée en possédant m, le nombre d'attributs crées est potentiellement $n * m$, avec autant de règles sémantiques. L'évaluation symbolique, qui permet d'éliminer les constructeurs de structures intermédiaires, peut alors potentiellement être appliquée sur chacun des termes ainsi créés dans ces règles. On retrouve ici la nécessité de borner en pratique le nombre d'applications successives de SE sur un terme généré par un pas de projection. D'une manière générale, la composition symbolique ainsi utilisée dépend de façon quadratique de la taille et de la profondeur des termes des grammaires attribuées à composer.

Notons finalement que ces remarques sont indicatives et assez informelles. Il serait tout aussi intéressant d'étudier ces problèmes de complexité de façon plus formelle que de les comparer avec la complexité des autres techniques de déforestation des programmes fonctionnels présentées dans ce mémoire. À ma connaissance, ce problème n'a pas été abordé, ou du moins publié, dans le cadre de la normalisation des *folds* ni dans celui de la fusion des hylomorphismes. Les résultats qui ont été rapportés dans le cadre de la règle d'élimination `foldr/build` sont relatifs à l'efficacité de son implantation dans le compilateur d'Haskell [Gil96] ; j'y reviendrai brièvement dans la conclusion (cf. section 10.4).

8.3 Une nouvelle technique de déforestation

J'ai déjà montré dans le chapitre 6 que pour des programmes simples et plus particulièrement pour les grammaires attribuées S^1, les *folds* du premier ordre et certains hylomorphismes, la composition descriptionnelle et les techniques de déforestation fonctionnelles aboutissaient à des résultats équivalents.

En dépit des restrictions associées à la transformation FP-to-AG, la composition symbolique qui vient d'être présentée permet de déforester une classe de programmes pour laquelle les déforestations fonctionnelles étaient insuffisantes. Cette classe de programme regroupe les fonctions dans lesquelles un paramètre d'accumulation construit le résultat de la fonction sans suivre exactement le schéma de récursion du type de donnée sous-jacent ou de la fonction elle-même. Les compositions $(rev\ (flat\ t\ nil)\ nil)$ ou $(rev\ (rev\ x\ nil)\ nil)$ appartiennent à cette classe pour laquelle, si SC représente la composition symbolique, on a :

$$\boxed{\text{SC} \circ \text{FP-to-AG} \;>\; \text{FP-to-AG} \circ \text{Déforestation Fonctionnelle}}$$

La suite de cette section tente de mettre en évidence certaines des raisons qui expliquent cet avantage de la composition symbolique par rapport aux différentes méthodes calculationnelles.

En fait, la plupart des méthodes de fusion ou de déforestation dirigées par la structure des données en programmation fonctionnelle s'appuient sur la notion de foncteur et utilisent des opérateurs de contrôle génériques permettant de capturer *en même temps* le schéma de récursion des fonctions et celui des types de données. D'abord, les règles de `foldr/build` de la *shortcut deforestation* de Gill *et al.* [GLJ93] implantée dans Haskell, rendaient possibles l'élimination de listes intermédiaires pour des compositions de fonctions dont une version en *fold* était disponible. Ensuite, la normalisation des *folds* de Sheard et Fegaras [SF93] généralisait cette technique à tous les types de donnée, en produisant automatiquement leurs foncteurs associés à partir de leurs définitions algébriques. Cependant, ces foncteurs étaient trop *isomorphes* aux types pour représenter correctement certains calculs, aussi les hylomorphismes sous forme de triplet furent introduits par Takano et Meijer [TM95], puis développés dans des systèmes tels qu'ADL ou HYLO [KL94, OHIT97]. Ces derniers permettent de déforester automatiquement certains programmes fonctionnels complexes, mais échouent encore, en terme de déforestation, sur des programmes tels que $(rev\ (flat\ t\ nil)\ nil)$ ou $(rev\ (rev\ x\ nil)\ nil)$, qui peuvent désormais être traités efficacement grâce à la composition symbolique.

Une meilleure déforestation

Dans les méthodes de déforestation fonctionnelles, en dépit des généralisations ou raffinements successifs, la classe des fonctions telles que $rev \circ rev$ ou $rev \circ flat$ restait problématique. Je donne ici quelques remarques qui peuvent expliquer pourquoi.

Les méthodes calculationnelles utilisent toujours des foncteurs pour diriger leurs transformations et leurs calculs, ce qui n'est pas le cas de la composition symbolique et des grammaires attribuées. L'exemple de la fonction *rev* est assez représentatif :

$$\texttt{let}\ rev\ x\ l = \texttt{case}\ x\ \texttt{of}$$
$$cons\ head\ tail \rightarrow \underline{rev}\ tail\ (\underline{cons}\ head\ l) \quad [\ldots]$$

Si F_{rev} est le foncteur qui dirige la récursion du type liste, il ne suit pas la construction de la liste inversée, mais les appels récursifs de la fonction. Ce faisant, il *cache* la construction du résultat. Plus précisément le constructeur $\underline{cons}$ est caché dans le second paramètre de l'appel récursif à $\underline{rev}$. Dans ces conditions, la déforestation locale sur ce $\underline{cons}$ inaccessible est impossible.

Grâce à l'expression de cette fonction en grammaire attribuée, la composition symbolique permet d'atteindre chaque constructeur du résultat, directement à partir de la spécification. De plus, ceci est possible sans l'utilisation d'une forme intermédiaire abstraite telle que les foncteurs. La raison de cette différence est que toutes les constructions du résultat sont visibles dans la grammaire attribuée, même s'ils ne suivent pas exactement le schéma de récursion (foncteur) du type sous-jacent. Ceci explique que la composition symbolique permette une meilleure déforestation sur les programmes construisant une structure intermédiaire dans un paramètre d'accumulation, comme c'est le cas dans *revflat* où le $\underline{cons}$ ci-dessus est déforesté.

Pour permettre aux méthodes calculationnelles d'effectuer de telles déforestations, il faudrait envisager un foncteur qui décrive complètement le schéma de récursion de la construction du résultat, et qui prenne en compte les paramètres d'accumulations dans lesquels des fragments de résultats sont construits ou transmis. Les travaux de Meijer et Hutton sur les *difoncteurs* [MH95], qui permettent d'exprimer des schémas récursifs plus complexes que ceux pris en charge par les endofoncteurs polynômiaux considérés dans ces méthodes de déforestation, constituent peut être une piste pour apporter une amélioration à ce problème.

Pour conclure ici, je rappelle que la composition symbolique est une méthode de déforestation *générale*. Grâce à la transformation FP-to-AG et à la transformation inverse, elle n'est plus limitée uniquement aux grammaires attribuées.

Du point de vue des grammaires attribuées

Dans le domaine des grammaires attribuées, l'introduction de la composition symbolique constitue une intégration plus complète de la composition descriptionnelle et la rend plus largement utilisable. L'idée initiale introduite par Ganzinger et Giegerich, plus récemment développée et mise en œuvre par Roussel, était d'appliquer un mécanisme de composition entre deux grammaires attribuées séparées. La composition symbolique, grâce à la localité de ses pas de projections et de son évaluation symbolique, permet d'effectuer de la déforestation sur des termes *à l'intérieur* d'une grammaire attribuée. De plus, les termes finis peuvent être évalués pendant ces transformations et une utilisation particulière de la composition symbolique peut être vue comme de l'évaluation partielle.

Finalement, il faut rappeler que le formalisme des grammaires attribuées n'est pas seulement une notation abstraite pour écrire des équations sémantiques. C'est également un langage de programmation complet, reconnu pour sa puissance et son efficacité dans la conception de grosses applications, ou pour l'écriture de compilateurs (cf. système FNC-2 [JPJ+90]). De plus, pour effectuer de la déforestation, même pour les programmes complexes qui posent des problèmes aux autres méthodes, aucune extension spécifique n'a été requise dans le formalisme initial et simple des grammaires attribuées. C'est une preuve supplémentaire de la possibilité d'utiliser les grammaires attribuées à des fins de déforestation comme une alternative avantageuse à d'autres formalismes classiques dans ce contexte.

Chapitre 9

Elimination de règles de copie

Les grammaires attribuées, particulièrement celles qui sont produites par la composition symbolique, peuvent contenir de nombreuses règles de copie qui servent uniquement à transporter des valeurs ou des constantes le long de la structure de l'arbre d'entrée. Cependant, il est possible d'éliminer ces règles de copie dans de nombreux cas, grâce à une analyse statique globale de la grammaire attribuée [Rou94]. Je rappelle ici que les transformations souhaitées doivent être valables quel que soit le paramètre d'entrée de la grammaire attribuée. Elles ont lieu sur la spécification de la grammaire attribuée. D'autres optimisations, liées à un paramètre donné, peuvent être faites de façon dynamique par l'évaluateur produit par le système de traitement de grammaires attribuées, comme c'est le cas pour FNC-2.

Dans le cadre de l'application de la composition symbolique sur une grammaire attribuée produite par la transformation FP-to-AG, c'est-à-dire correspondant à un programme fonctionnel donné dont on attend le programme fonctionnel correspondant à la version déforestée par la composition symbolique, l'application de ces éliminations de règles de copie est conditionnée par des contraintes sur la sémantique d'évaluation du programme fonctionnel considéré. En fait, plus particulièrement dans le cas d'évaluation paresseuse, ces éliminations peuvent conduire à une non-préservation de la sémantique du programme déforesté. Après avoir donné l'idée intuitive des transformations d'élimination des règles de copie, je reviendrai à la fin de cette section sur ses conditions d'application.

9.1 Deux types d'élimination de règles de copie

Pour une grammaire attribuée donnée, deux types d'élimination de règles de copie peuvent être effectués. Le premier type est celui des **règles de copie menant à une constante** et le second concerne les **boucles de règles de copie**. Dans le premier cas, il s'agit des suites de règles de copie entre occurrences d'attributs telles que la première occurrence d'attribut de cette suite soit initialisée par une constante. Dans le second cas, il s'agit des suites de règles de copie qui débutent et terminent dans la même production, c'est-à-dire qui débutent et qui terminent sur deux attributs d'un même non-terminal.

Les algorithmes d'élimination des règles de copie que je présente ici de façon informelle sont dûs à Roussel [Rou94]. Le but de cette présentation est de montrer que, en plus des résultats de déforestation obtenus grâce au passage par le formalisme des grammaires attribuées, ce même formalisme et les analyses statiques qu'il autorise permettent de *réarranger* la grammaire attribuée obtenue et, le cas échéant, le programme fonctionnel qui peut en être dérivé.

```
aglet dblrev =
  dblrev x →
    dblrev.result = x.rr
    x.hr = x.hh
    x.rh = nil
  cons head tail →
    cons.rr = tail.rr
    tail.hr = cons.hr
    cons.hh = cons head tail.hh
    tail.rh = cons.rh
  nil →
    nil.rr = nil.hr
    nil.hh = nil.rh
```

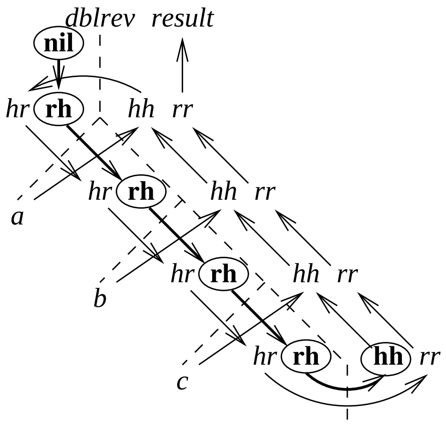

FIG. 9.1: *Règle de copie de constante sur dblrev*

Les principes d'évaluation des grammaires attribuées sont basés sur la notion de dépendance entre occurrences d'attributs (section 2.2.2). Ces dépendances représentent l'utilisation d'une occurrence d'attribut dans la définition d'une autre mais elles ne modélisent pas le genre de fonction utilisée dans la règle sémantique de cette définition. Pour l'élimination des règles de copie, la notion de dépendance est étendue à la notion de **relation** engendrée par une règle sémantique. Celle-ci permet de modéliser, en plus de la notion de dépendance classique, à la fois le fait qu'une occurrence d'attribut dépend d'une constante ou non et le fait que la règle sémantique qui induit la dépendance est une règle d'identité (simple recopie) ou non. Brièvement, une relation est un triplet (A, B, K) qui représente le fait que B dépend de A par une fonction modélisée par K. Dans une telle relation, B est une occurrence d'attribut, A peut être soit une occurrence d'attribut, soit le symbole $\perp$ qui modélise une constante et $K \in \{id, fonc\}$ modélise la fonction d'une règle sémantique comme étant une règle de copie (id) ou une autre fonction ($fonc$) [Rou94].

À partir de cette notion de relation, des analyses similaires à celles effectuées sur les dépendances peuvent être faites sur une grammaire attribuée : elles représentent précisément le flot de données des attributs et plus particulièrement la propagation des valeurs constantes. Elles permettent alors d'éliminer des règles de copie, par des algorithmes de transformation qui sont détaillés dans [Rou94] et implantés dans le système FNC-2. Le but ici n'étant pas de retranscrire ces algorithmes, je me contenterai d'exposer leurs effets dans chacun des cas d'élimination, sur des exemples qui ont déjà été rencontrés dans cette thèse.

9.1.1 Élimination des règles de copie menant à une constante

Ce type d'élimination consiste, grâce aux analyses effectuées sur les notions de relations, à repérer une propagation, par des règles sémantiques étant des identités, d'une valeur identifiée comme étant une constante. Les occurrences d'attributs concernées par cette propagation peuvent alors être définies par l'affectation de cette valeur constante plutôt que par la règle sémantique initiale.

Pour illustrer ce type de règles de copie, je présente à la figure 9.1 l'exemple de la double inversion de liste *dblrev*, telle qu'elle est produite par la composition symbolique. Les occur-

```
aglet dblrev2 =
  dblrev2 x →
    dblrev2.result = x.rr
    x.hr = x.hh
  cons head tail →
    cons.rr = tail.rr
    tail.hr = cons.hr
    cons.hh = cons head tail.hh
  nil →
    nil.rr = nil.hr
    nil.hh = nil
```

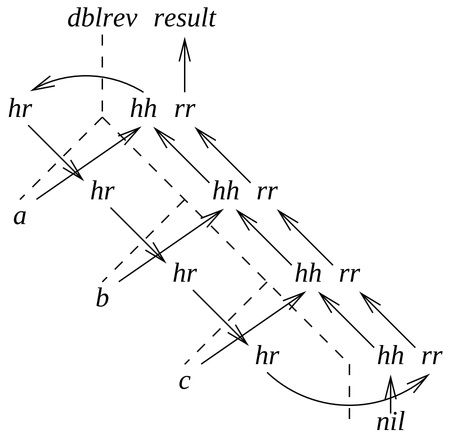

FIG. 9.2: *Après élimination des règle de copie de constante sur dblrev*

rences de l'attribut rh sont toutes initialisées soit par une recopie d'une autre occurrence de rh, soit par la valeur constante *nil* dans le cas du profil (propagation visualisée en gras dans la figure 9.1). Il existe en fait une suite de recopies, depuis l'occurrence d'attribut $x.rh$ du profil défini par une constante, jusqu'à l'occurrence d'attribut $nil.hh$ du *pattern* ($nil \rightarrow$) ; cette suite de recopies existe quel que soit l'arbre d'entrée. Il est alors clair que ces règles de copie peuvent être remplacées par une affectation de la valeur constante *nil*.

Dans ce cas, la grammaire attribuée transformée devient celle présentée à la figure 9.2. Les techniques d'évaluation des grammaires attribuées, telles que la génération de séquences de visites à partir des graphes de dépendances (section 2.2.4), déterminent alors directement la valeur de l'occurrence d'attribut $nil.hh$ du *pattern* ($nil \rightarrow$) ; les occurrences de l'attribut rh deviennent alors des *impasses statiques* qui sont éliminées statiquement. Le principe d'évaluation d'une telle grammaire attribuée, après l'élimination de la suite de règles de copie conduisant à une constante, est représenté sur le dessin de la figure 9.2.

9.1.2 Élimination des boucles de règles de copie

L'autre type d'élimination qu'il est possible d'effectuer concerne les boucles de règles de copie. Intuitivement, il s'agit d'une passe ou d'un parcours sur (un sous-arbre de) l'arbre d'entrée, qui propage une valeur sans la modifier. Dans ce cas, il existe deux attributs sur un même non-terminal qui sont respectivement le début et la fin de la suite de règles de copie. L'exemple de la figure 9.1 contient une telle boucle, qui va de $x.hh$ à $x.rr$ dans le profil, mais je vais illustrer sur un exemple encore plus probant.

Je présente à la figure 9.3 l'exemple de la grammaire attribuée *revflat* produite par la composition symbolique de *rev* et de *flat* (cette composition symbolique est décrite à la section 8.2). Cette grammaire attribuée *revflat* contient également une boucle de règle de copie, qui va de $t.lh$ à $t.fr$ sur le profil ; elle est mise en évidence en gras sur l'exemple de la figure 9.3. Dans ce cas, l'algorithme d'élimination des boucles de règles de copie permet de remplacer la règle sémantique du profil $revflat.result = t.fr$ par $revflat.result = t.lh$, rendant ainsi les calculs de fr, et donc de lr, inutiles pour la valeur sémantique de l'arbre d'entrée. La grammaire attribuée *Rflat* ainsi obtenue est présentée à la figure 9.4, avec un dessin illustrant son évaluation sur un exemple.

```
aglet revflat =
  revflat t →
    revflat.result = t.fr
    t.fh = nil
    t.lr = t.lh
  node left right →
    node.fr = left.fr
    left.fh = node.fh
    left.lr = right.fr
    right.fh = left.lh
    right.lr = node.lr
    node.lh = right.lh
  leaf n →
    leaf.fr = leaf.lr
    leaf.lh = cons n leaf.fh
```

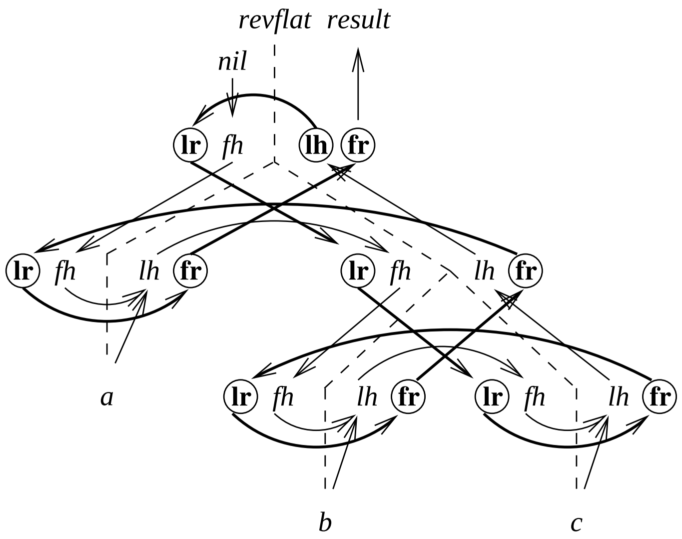

FIG. 9.3: *Boucle de règles de copie dans revflat*

```
aglet Rflat =
  Rflat t →
    Rflat.result = t.lh
    t.fh = nil
  node left right →
    left.fh = node.fh
    right.fh = left.lh
    node.lh = right.lh
  leaf n →
    leaf.lh = cons n leaf.fh
```

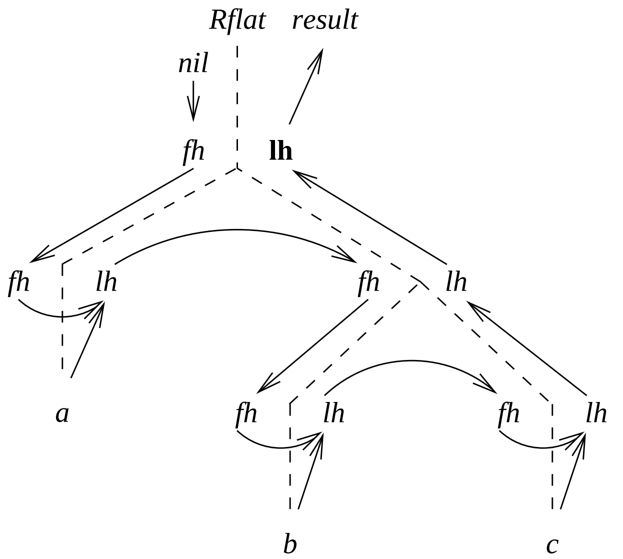

FIG. 9.4: *Après élimination de la boucle de règles de copie dans revflat*

```
let rev x h = case x of            let flat t l = case t of            let Rflat t l = case t of
  cons head tail →                   node left right →                   node left right →
    rev tail (cons head h)             flat left (flat right l)            Rflat right (Rflat left l)
  nil → h                            leaf n → cons n l                   leaf n → cons n l
```

FIG. 9.5: *Les fonctions rev, flat et Rflat*

Par ailleurs, l'application de cette règle d'élimination de boucle de règles de copie sur la grammaire attribuée *dblrev* obtenue à la figure 9.2, après élimination des règles de copie menant à une constante, abouti à la grammaire attribuée de simple recopie de liste.

9.2 Conditions d'application

Ces algorithmes de repérages et d'élimination de règles de copie ont été développés dans le cadre des grammaires attribuées. Il est important de noter qu'ils peuvent **changer l'ordre d'évaluation** de la grammaire attribuée et en particulier le sens de parcours de l'arbre abstrait correspondant au paramètre d'entrée.

Par exemple, la grammaire attribuée *flat* peut être évaluée en parcourant l'arbre d'entrée des feuilles de droite vers les feuilles de gauche. Après l'application de la composition symbolique sur la composition de *rev* avec *flat*, la grammaire attribuée *revflat* obtenue peut être évaluée en parcourant l'arbre d'entrée des feuilles de gauche vers les feuilles de droite, puis en transportant le résultat construit lors de ce parcours en sens inverse, le long de sa construction, des feuilles de droite vers les feuilles de gauche. L'application de l'algorithme d'élimination des boucles de règles de copie sur la grammaire attribuée *revflat* élimine ce dernier parcours de "transport" du résultat des feuilles de droite vers les feuilles de gauche. Il en résulte une grammaire attribuée *Rflat* dont l'ordre d'évaluation est l'opposé de celui de *flat*. Ce résultat, illustré par la figure 9.4, est très probant en terme d'efficacité du programme déforesté obtenu.

Néanmoins, cette modification possible de l'ordre d'évaluation d'une grammaire attribuée par l'élimination des boucles de règles de copie ne peut être acceptable que si les arguments de cette grammaire attribuée correspondent à des arbres abstraits totalement définis et ne comportant aucune branche infinie. Cette remarque concerne principalement la déforestation de programmes fonctionnels, transformés en grammaires attribuées par FP-to-AG, déforestés par la composition symbolique, puis retranscrits en programmes fonctionnels. Dans ce cas, l'élimination des règles de copie ne doit être effectuée que dans le cas où les paramètres d'entrée du programme fonctionnel peuvent être garantis comme totalement définis, sans branches infinies. En effet, considérons le cas du programme d'entrée

$$(rev \circ flat)\ t = rev\ (flat\ t\ nil)\ nil$$

considéré dans une sémantique d'évaluation paresseuse (cf. figure 9.5). Après transformation de ces fonctions en grammaires attribuées par FP-to-AG, déforestation de leur composition par la composition symbolique, puis élimination des règles de copie, le programme fonctionnel correspondant à la grammaire attribuée *Rflat* (présentée à la figure 9.5) parcourt l'arbre dans le sens opposé du programme *flat* initial.

Programme initial

$$
\begin{aligned}
& \textit{rev}\ (\textit{flat}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ \textit{nil})\ \textit{nil} \\
= \;& \textit{rev}\ (\textit{flat}\ (\perp)\ (\textit{flat}\ (\textit{leaf}\ \ a)\ \textit{nil}))\ \textit{nil} \\
= \;& \textit{rev}\ (\perp)\ \textit{nil} \\
= \;& (\perp)
\end{aligned}
$$

Après composition symbolique et élimination des règles de copie

$$
\begin{aligned}
& \textit{Rflat}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ \textit{nil} \\
= \;& \textit{Rflat}\ (\textit{leaf}\ \ a)\ (\textit{Rflat}\ (\perp)\ \textit{nil}) \\
= \;& \textit{cons}\ a\ (\textit{Rflat}\ (\perp)\ \textit{nil}) \\
= \;& \textit{cons}\ a\ (\perp)
\end{aligned}
$$

FIG. 9.6: *Traces d'exécution en évaluation paresseuse sur un exemple de rev (flat t nil) nil et de Rflat t nil*

Dans ce cas, sur un arbre binaire dont la branche la plus à gauche est infinie ou conduit à $\perp$, l'évaluation paresseuse de *rev (flat t nil) nil* produira également $\perp$, et ne nécessite aucun *cons*. Par contre, la fonction *Rflat* correspondant à la grammaire attribuée déforestée après élimination des règles de copie (figure 9.4) nécessite dans la même sémantique d'évaluation autant de constructeurs *cons* que l'arbre possède de feuilles définies, avant de rencontrer la branche la plus à gauche et de produire $\perp$. Sur l'exemple d'un tel arbre avec une branche conduisant à $\perp$, la figure 9.6 présente les traces d'exécutions des deux fonctions dans une sémantique d'évaluation paresseuse.

Un autre exemple flagrant de non-préservation de la sémantique paresseuse sur un argument partiellement défini est celui de la double inversion de liste, *dblrev2*. L'élimination des règles de copie sur la grammaire attribuée correspondante conduit à la recopie du paramètre. Or, l'évaluation de la recopie d'une liste infinie en évaluation paresseuse n'est pas sémantiquement équivalente à l'évaluation de la double inversion de cette liste.

Il faut donc considérer plusieurs cas, selon la sémantique d'évaluation, stricte ou paresseuse, et selon le type de paramètres acceptés en entrée d'un programme, avec des branches infinies ou non.

Tout d'abord, je rappelle que les techniques de déforestation, quelles qu'elles soient, ne préservent pas la sémantique d'évaluation stricte. L'exemple typique est la composition de la longueur d'une liste avec un *map* qui consiste à diviser chaque élément par lui même. Si un élément de la liste vaut zéro, la composition de ces fonctions produit une erreur, tandis qu'une fois déforesté, le programme résultant se contente de calculer la longueur de la liste initiale, et produit un résultat. En ce sens, la déforestation peut être vue comme la spécialisation de deux programmes en fonction de la sémantique de leur composition : la longueur d'une liste, quoi que l'on fasse sur ses éléments, ne dépend que du nombre de ces éléments. Cela ne signifie pas que la déforestation soit dénuée d'intérêt dans le cadre d'une sémantique d'évaluation stricte. Au contraire, en tant qu'outil de généricité, en matière de spécialisation de composition de briques logicielles, le fait que la déforestation "oublie" une partie de la sémantique d'un composant dans le cadre précis d'une de ses utilisations peut être vu comme un atout.

```
let revflat t l = revflat2 t (revflat1 t l)

let revflat1 t l = case t of
   node left right →  revflat1 right (revflat1 left l)
   leaf n →  cons n l

let revflat2 t l = case t of
   node left right →  revflat2 left (revflat2 right l)
   leaf n →  l
```

FIG. 9.7: *La fonction revflat déforestée par composition symbolique*

Dans le cadre d'une sémantique d'évaluation paresseuse, la composition symbolique, comme les autres techniques de déforestation, préserve la sémantique du programme initial, dans la mesure où les opérations qui sont nécessaires au calcul d'un résultat dans la composition initiale sont conservées dans le programme résultant. De plus, la composition symbolique préserve l'ordre du premier parcours de la structure du paramètre d'entrée, même si ce parcours ne spécifie qu'un transport de valeur : il s'agit alors d'un ensemble de règles de copie.

Je reprends une dernière fois l'exemple de la composition des fonctions *rev* et *flat*. La fonction *revflat* obtenue par composition symbolique (section 8.2) est rappelée à la figure 9.7. L'ordre de parcours de la structure spécifié par la fonction externe *revflat2* est le même que celui spécifié par la fonction initiale *flat*, même s'il ne calcule aucune valeur et se contente de propager un résultat. Dans le cas où le paramètre d'entrée du programme contient une branche infinie, cela permet à la version déforestée *revflat* d'avoir un comportement compatible avec la fonction *flat* initiale. Plus exactement, en cas de branche infinie et d'évaluation conduisant à $\perp$, le programme déforesté aboutit au même résultat en *au plus* autant de constructions.

La figure 9.8 présente les traces d'exécutions sur un exemple d'arbre d'entrée t avec une branche gauche infinie ou menant à $\perp$. Dans ce cas, le programme initial *rev* (*flat t nil*) *nil* et le programme déforesté *revflat* aboutissent tous les deux à $\perp$ sans aucune construction.

La figure 9.9 présente les traces d'exécutions sur un exemple d'arbre d'entrée t avec une branche droite infinie ou menant à $\perp$. Dans ce cas, le programme initial *rev* (*flat t nil*) *nil* construit autant de *cons* (en gras dans l'exemple) qu'il rencontre de feuilles avant $\perp$ tandis que le programme déforesté *revflat* aboutit à $\perp$ sans construire aucun *cons*.

Il est donc possible de considérer que la déforestation évite également de construire des structures inutiles dans le cas où la structure d'entrée possède des branches infinies ou menant à $\perp$.

En conclusion, je tiens à rappeler qu'en matière d'optimisation de programmes, c'est-à-dire de transformation dont l'application améliore l'efficacité de l'évaluation d'un programme, il est fréquement admis que ces transformations puissent changer le comportement des cas d'exceptions. C'est le cas de la déforestation effectuée par la composition symbolique et couplée à l'élimination des règles de copies.

Programme initial

$$\begin{array}{ll} & \textit{rev}\ (\textit{flat}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ \textit{nil})\ \textit{nil} \\ = & \textit{rev}\ (\textit{flat}\ (\perp)\ (\textit{flat}\ (\textit{leaf}\ \ a)\ \textit{nil}))\ \textit{nil} \\ = & \textit{rev}\ (\perp)\ \textit{nil} \\ = & (\perp) \end{array}$$

Après composition symbolique

$$\begin{array}{ll} & \textit{revflat}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ \textit{nil} \\ = & \textit{revflat2}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ (\textit{revflat1}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ \textit{nil}) \\ = & \textit{revflat2}\ (\perp)\ (\textit{revflat2}\ (\textit{leaf}\ \ a)\ (\textit{revflat1}\ (\textit{node}\ (\perp)\ (\textit{leaf}\ \ a))\ \textit{nil})) \\ = & (\perp) \end{array}$$

FIG. 9.8: *Traces d'exécution en évaluation paresseuse sur un exemple de (rev (flat t nil) nil) et de (revflat t nil)*

Programme initial

$$\begin{array}{ll} & \textit{rev}\ (\textit{flat}\ (\textit{node}\ (\textit{leaf}\ \ a)\ (\perp))\ \textit{nil})\ \textit{nil} \\ = & \textit{rev}\ (\textit{flat}\ (\textit{leaf}\ \ a)\ (\textit{flat}\ (\perp)\ \textit{nil}))\ \textit{nil} \\ = & \textit{rev}\ (\mathbf{cons}\ a\ (\textit{flat}\ (\perp)\ \textit{nil}))\ \textit{nil} \\ = & \textit{rev}\ (\textit{flat}\ (\perp)\ \textit{nil})\ (\mathbf{cons}\ a\ \textit{nil}) \\ = & \textit{rev}\ (\perp)\ (\mathbf{cons}\ a\ \textit{nil}) \\ = & (\perp) \end{array}$$

Après composition symbolique

$$\begin{array}{ll} & \textit{revflat}\ (\textit{node}\ (\textit{leaf}\ \ a)\ (\perp))\ \textit{nil} \\ = & \textit{revflat2}\ (\textit{node}\ (\textit{leaf}\ \ a)\ (\perp))\ (\textit{revflat1}\ (\textit{node}\ (\textit{leaf}\ \ a)\ (\perp))\ \textit{nil}) \\ = & \textit{revflat2}\ (\textit{leaf}\ \ a)\ (\textit{revflat2}\ (\perp)\ (\textit{revflat1}\ (\textit{node}\ (\textit{leaf}\ \ a)\ (\perp))\ \textit{nil})) \\ = & \textit{revflat2}\ (\perp)\ (\textit{revflat1}\ (\textit{node}\ (\textit{leaf}\ \ a)\ (\perp))\ \textit{nil}) \\ = & (\perp) \end{array}$$

FIG. 9.9: *Traces d'exécution en évaluation paresseuse sur un exemple de (rev (flat t nil) nil) et de (revflat t nil)*

Chapitre 10

Conclusion

Les différents paradigmes de programmation qui existent ont été étudiés et développés afin de répondre à des exigences spécifiques. La programmation par objets, la programmation fonctionnelle, la programmation logique ou la programmation séquentielle impérative sont des paradigmes bien connus, dont chacun est adapté à une approche particulière d'un problème. Malheureusement, la diversité de ces paradigmes et les différences de formalismes qui les séparent cachent souvent des relations, similitudes ou complémentarités entre les techniques de traitement et de résolution de problèmes voisins ou identiques.

La suppression des structures de données intermédiaires apparaissant lors de la composition des différentes parties d'une application modulaire, néfastes pour son efficacité, est un problème difficile qui a suscité de nombreux travaux dans le cadre de la programmation fonctionnelle, mais également dans le domaine des grammaires attribuées. Ce type de transformations de programmes particulier a été le cas d'étude privilégié de ma thèse dont l'objectif était d'apporter une contribution aux relations pouvant exister entre les grammaires attribuées et la programmation fonctionnelle.

10.1 Une plus large utilisation des grammaires attribuées

Comme chaque paradigme de programmation, les grammaires attribuées possèdent leur domaine de prédilection ; ce fut très longtemps la compilation et la génération de code. Dans ce cadre particulier, elles ont fait la preuve de leur puissance et de leur efficacité, mais se sont cependant confinées dans un domaine où la structure syntaxique est, sinon indispensable, au moins prédominante. Ainsi, la plupart des développements et des études qui ont été menées concernant les grammaires attribuées sont fortement liés à la présence *concrète* d'un arbre syntaxique qui guide les calculs.

Des grammaires attribuées plus proches des programmes fonctionnels

Dans le cadre de leur comparaison avec la programmation fonctionnelle, cette apparente nécessité d'une structure concrète était un premier obstacle en terme d'expressivité des programmes, l'exemple typique étant l'impossibilité de représenter une fonction comme factorielle, qui ne s'appuie sur aucune structure de donnée concrète. La première contribution apportée dans cette thèse, rapprochant les grammaires attribuées des programmes fonctionnels, a donc été de montrer que les grammaires attribuées pouvaient se passer de l'existence physique d'un arbre. J'ai donc présenté les grammaires attribuées dynamiques, qui sont une extension naturelle du formalisme classique des grammaires attribuées. Grâce à l'association d'une grammaire *concrète* représentant une structure syntaxique classique et d'une grammaire *abstraite* représentant un schéma de récursion, les grammaires attribuées dynamiques permettent de considérer le schéma récursif d'une fonction comme un arbre *virtuel* guidant les calculs spécifiés par des règles sémantiques *dynamiques*, ces dernières pouvant être choisies dynamiquement en fonction de la valeur d'un attribut. Les cas où la grammaire attribuée concrète est réduite à néant sont alors envisageables et c'est par exemple le cas de la fonction factorielle qui peut donc être représentée dans ce formalisme en considérant une structure de récursion purement abstraite en guise d'arbre guidant les calculs.

De plus, les grammaires attribuées dynamiques sont compatibles avec les méthodes d'évaluations classiques des grammaires attribuées et deux approches ont été présentées permettant de générer des évaluateurs à séquences de visites correspondants. Ces grammaires attribuées dynamiques ont été implantées en tant qu'extension au système de traitement de grammaires attribuées FNC-2, développé à l'INRIA.

Cette extension de l'expressivité des grammaires attribuées permet de se rapprocher des programmes fonctionnels. Elle comble une lacune qui existait dans ce formalisme en dépit de sa puissance intrinsèque et, par la possibilité de représenter une plus large classe de programmes, elle facilite la comparaison entre les techniques de transformation développées pour les grammaires attribuées et pour les programmes fonctionnels.

Un formalisme adapté aux analyses et aux transformations de programmes

L'intérêt fondamental de ce rapprochement n'est pas de considérer les grammaires attribuées comme pouvant *remplacer* les programmes fonctionnels, ou n'importe quel autre paradigme de programmation. Autrement dit, l'intérêt pour l'utilisateur n'est pas d'écrire la fonction factorielle en grammaire attribuée dynamique plutôt qu'en Lisp, en Haskell ou en ML qui sont bien mieux adaptés à sa spécification.

L'intérêt est plutôt de pouvoir *représenter* une plus large classe de programmes dans le formalisme des grammaires attribuées qui est particulièrement bien adapté aux analyses et aux transformations de programmes et qui dispose déjà d'un ensemble de techniques puissantes (traductions, analyses statiques, optimisations, etc). Pouvoir représenter un programme en grammaire attribuée signifie alors qu'il est possible de mettre à sa disposition cet ensemble de techniques. Ainsi, l'objectif de notre démarche est de considérer les grammaires attribuées comme un formalisme de représentation d'un programme, plus général et plus abstrait que le langage dans lequel ce programme peut être spécifié ou exécuté. L'étude décrite dans cette thèse au sujet de la déforestation est une illustration probante de ce principe.

10.2 La déforestation comme cas d'étude

La comparaison des techniques développées dans les grammaires attribuées et dans la programmation fonctionnelle qui est présentée dans cette thèse a donc été basée sur le cas particulier de transformations de programmes qu'est la déforestation. Ce problème de l'élimination des structures de données servant de *colle* entre les différents composants modulaires d'un programme constitue la *caution* du style de programmation par composition de multiples spécifications de modules simples et indépendants. En effet, les avantages de ce style de développement sont nombreux : lisibilité, clarté, traçabilité et réutilisabilité de petits composants, dont la validation individuelle est facilitée. Cependant, au sein de l'agencement de ces composants dans une application, l'ensemble des structures produites par les uns pour être consommées par les autres engendre un coût d'allocation, de parcours et de désallocation qui peut remettre en cause, de part la perte d'efficacité, l'intérêt même d'une telle approche modulaire.

L'idée de transformer un programme, en vue de supprimer les structures intermédiaires occasionnées par les compositions qu'il contient, a donc été très largement étudiée là où la composition et la modularité étaient utilisées. Ce fut le cas dans les grammaires attribuées avec la composition descriptionnelle de Ganzinger et Giegerich, technique dédiée à ces spécifications déclaratives guidées par la structure des données. Ce fut également le cas en programmation fonctionnelle, où la modularité et la composition sont fondamentales. Dans ce contexte, de nombreux travaux de recherche ont été menés. Le principe de transformation de dépliage/pliage de Burstall et Darlington a inspiré différentes études et en particulier l'algorithme de Wadler qui a donné le nom de déforestation à ce type de transformation de programme.

Des restrictions sur l'ensemble de méthodes de déforestation étudiées

L'ensemble des travaux sur les méthodes de déforestation en programmation fonctionnelle qui sont présentés et comparés dans cette thèse n'est pas exhaustif. Il est représentatif d'un ensemble d'approches de ces transformations de programmes où la localité des transformations est guidée par la structure des données. Ce critère de choix a été dicté par le contexte de comparaison des techniques dédiées aux grammaires attribuées avec celles utilisées en programmation fonctionnelle.

L'étude des méthodes de transformation de programmes basées sur des systèmes de réécriture, tels que ceux de Reddy ou de Bellegarde, apporterait probablement des éléments de comparaison intéressants. De même, les techniques de déforestation développées dans le domaine de la programmation logique, principalement par Pettorossi et Proietti, pourraient être comparées à celles que j'ai étudié. Par ailleurs, les travaux de Sørensen, Glück *et al.* sur l'unification de l'évaluation partielle et de la super-compilation avec la déforestation mériteraient d'être étudiés dans la continuité de cette thèse.

Néanmoins, je me suis intéressé en priorité aux méthodes fonctionnelles qui semblaient être les plus "proches" des spécificités des grammaires attribuées, au sens ou la structure des données dirige à la fois la spécification des programmes et la technique de transformation.

Les méthodes de déforestation fonctionnelles considérées

Pour sa popularité et l'intérêt qu'elle à suscité dans la communauté fonctionnelle, j'ai considéré l'algorithme de Wadler comme référence historique du principe de la déforestation de programmes fonctionnels. Outre les nombreux développements, extensions et améliorations qui ont été apportés à l'algorithme de Wadler et à ce type de transformation, une autre approche à vu le jour, initiée par les travaux sur l'algorithmique constructive de Fokkinga, Meertens, Meijer *et al.*, souvent appelée déforestation en forme calculationnelle. En 1993, deux techniques de transformations ont été introduites parallèlement.

D'une part, la règle d'élimination `foldr/build` de Gill, Launchbury et Peyton Jones qui permet d'éliminer d'une façon facilement automatisable les listes intermédiaires générées par un constructeur et consommées par un destructeur ; cependant, cette approche ne permet de traiter que le type des listes.

D'autre part, l'algorithme de normalisation des *folds* de Sheard et Fegaras qui utilise la notion de foncteur inhérente au type algébrique d'une structure de donnée ; grâce au théorème de promotion relatif aux propriétés des foncteurs, cet algorithme permet de transformer une composition d'un programme où les calculs sont représentés en fonction des différents *patterns* de récursion du type de donnée sous-jacent.

Plus récemment, une approche généralisant les deux précédentes à été introduite par Meijer et Takano, puis développée par Hu, Iwasaki et Takeichi. Basée sur la théorie des catégories, la notion d'hylomorphisme permet de représenter des programmes en fonction de foncteurs qui ne dépendent plus uniquement du type de donnée sous-jacent, mais également du schéma de récursion des fonctions. Les théorèmes classiques de la théorie des catégories sont alors dédiés à cette utilisation particulière et fournissent des règles de transformation simples permettant d'automatiser la fusion d'hylomorphismes.

L'étude comparative des ces méthodes fonctionnelles

Pour pouvoir établir des comparaisons entre les techniques utilisées dans ces approches de la déforestation et la composition descriptionnelle des grammaires attribuées, cette thèse a tout d'abord présenté ces différentes méthodes fonctionnelles de déforestation. Leurs formalismes de base étant très différents, j'ai voulu aborder chacune de ces approches de manière formelle et précise pour en permettre une étude détaillée, mais également illustrée d'exemples pour donner au lecteur une idée intuitive de leurs principes dans une vue d'ensemble comparative.

Ainsi, j'ai pu mettre en évidence de profondes ressemblances entre l'algorithme de normalisation des *folds* et l'algorithme de déforestation de Wadler, dans leur principe de fonctionnement et dans l'ensemble des programmes qu'ils permettent de traiter. Par ailleurs, sans égaler la simplicité d'application de la règle `foldr/build`, les *folds* permettent de traiter différents types de manière générique et non plus simplement celui des listes. La notion d'hylomorphisme est ensuite apparue comme étant une généralisation de l'opérateur *fold*, mais également de son opérateur dual grâce aux notions de catamorphisme et d'anamorphisme qu'elle regroupe. Finalement, les théorèmes de fusion des hylomorphismes (*Acid Rain*) ont été identifiés comme des moyen d'automatisation de la déforestation qui englobent et généralisent à la fois l'algorithme de normalisation des *folds* et la règle d'élimination `foldr/build`.

Au delà de ces remarques comparatives, cette thèse a également permis de déterminer certaines limitations de ces formalismes. En particulier, lors de la composition d'un programme qui construit son résultat dans un paramètre d'accumulation, comme l'inversion de liste par exemple, j'ai montré que le schéma de récursion du type de données (pour les *folds*) ou de la fonction (pour les hylomorphismes), permettant de déterminer le foncteur correspondant à ce programme, ne permettait pas de suivre fidèlement la construction de la structure intermédiaire dans le paramètre d'accumulation. Les méthodes de déforestation calculationnelles étant guidées par ces foncteurs, elles ne permettent pas d'intervenir sur ces constructions intermédiaires. Cette particularité pose des problèmes aux algorithmes de normalisation ou de fusion, qui s'ils fonctionnent — c'est-à-dire génèrent une seule fonction à partir d'une composition — ne produisent pas le résultat optimal attendu en terme de déforestation, c'est-à-dire d'élimination des structures intermédiaires. Par exemple, la double inversion de liste et l'inversion de la liste des feuilles d'un arbre binaire sont des programmes que les méthodes fonctionnelles ne permettent pas, pour l'instant, de déforester de manière satisfaisante.

Par ailleurs, l'étude des méthodes calculationnelles a permis de mettre en évidence un problème dans la méthode de fusion des hylomorphismes présentée par Takano et Meijer, et d'en proposer une explication et un début de solution. Ce problème est resté jusqu'à présent dans l'ombre et n'a pas encore pu être éclairci par les auteurs.

Une déforestation facilitée dans les grammaires attribuées

Après la comparaison des méthodes fonctionnelles de déforestation entre elles, j'ai présenté leur comparaison avec la composition descriptionnelle des grammaires attribuées. J'ai d'abord montré que les *folds* de premier ordre et les grammaires attribuées purement synthétisées permettaient de représenter des programmes simples de manière équivalente et qu'ils aboutissaient, avec leurs méthodes respectives, aux mêmes résultats de déforestation. J'ai ensuite montré que le traitement de programmes plus complexes avait nécessité des extensions dans le formalisme des *folds* sans être pour autant complètement satisfaisant (algorithme de normalisation de second ordre, transformations *ad hoc* basées sur des types possédant des zéro-constructeurs, exceptions dans l'algorithme de normalisation étendu). Au contraire, les grammaires attribuées, avec leur notion d'attribut hérité, permettent de représenter ces programmes plus complexes dans leur formalisme de base et de traiter leur déforestation avec le même algorithme de composition descriptionnelle. En fait, un attribut hérité permet d'exprimer un calcul descendant dans une structure, ce que les *folds* nécessitent d'exprimer sous la forme d'un paramètre d'accumulation ou d'une fonction d'ordre supérieur. Ces dernières formes rendent la déforestation des structures qui y sont construites plus difficile puisqu'elles y sont encapsulées, tandis que sous la forme d'attributs hérités, qui sont des objets de première classe dans les grammaires attribuées, ces structures sont facilement accessibles et déforestables.

Concernant les hylomorphismes, qui sont guidés par des foncteurs représentant le schéma de récursion des fonctions, les structures intermédiaires intervenant dans des paramètres d'accumulation sont également difficiles à éliminer, puisque qu'elles ne sont pas construites en suivant le schéma de récursion déterminé par le foncteur. Les exemples de double inversion de liste ou d'inversion de la liste des feuilles d'un arbre binaire, déjà évoqués comme faisant échec à ces méthodes calculationnelles, sont naturellement traités par la composition descriptionnelle des grammaires attribuées correspondantes, où ces calculs sont directement accessibles puisque représentés dans des attributs hérités et non *cachés* dans des fonctions de second ordre ou des paramètres d'accumulation. Les travaux de Meijer et Hutton sur les *difoncteurs*, permettant d'exprimer des schémas récursifs plus complexes que ceux pris en charge par les endofoncteurs polynômiaux considérés dans ces méthodes de déforestation, pourraient peut être apporter une amélioration à ce problème.

D'une manière plus générale, le formalisme des *folds* tend à représenter une fonction dans les termes du schéma de récursion du type de donnée sous-jacent. Autrement dit, il s'agit de *plaquer* le flot de données sur un flot de contrôle arbitrairement fixé par le type du paramètre d'entrée. Il en va de même pour les hylomorphismes, ou cependant le flot de contrôle n'est plus seulement fixé par le type du paramètre d'entrée, mais prend également en compte le schéma de récursion de la fonction. Néanmoins, le flot de donnée est guidé par un flot de contrôle figé par le foncteur qui ne représente pas nécessairement la construction d'une structure intermédiaire intervenant dans un paramètre d'accumulation. La différence fondamentale entre ces approches et l'approche des grammaires attribuées est que dans ces dernières, les productions qui représentent le flot de contrôle (le type de donnée sous-jacent) sont tout à fait distinctes du flot de données représenté par les règles sémantiques. De plus, l'existence des attributs hérités ainsi que l'attachement des attributs aux variables de *pattern* permettent d'expliciter le flot de données de manière plus souple et pas nécessairement dans les termes d'un schéma de récursion figé.

10.3 Une nouvelle méthode de déforestation

Ces études comparatives de différents formalismes et différentes techniques de déforestation semblent indiquer que la composition descriptionnelle des grammaires attribuées possède des atouts intrinsèques par rapport aux méthodes calculationnelles existantes. Cependant, elle n'est applicable que sur un programme spécifié sous la forme d'une grammaire attribuée.

La fusion des hylomorphismes, spécifiée pour leur formalisme spécifique, n'a pu être utilisée dans le cadre de programmes fonctionnels que grâce à un algorithme de dérivation d'hylomorphismes à partir de définitions d'équations récursives. De la même façon, il est devenu indispensable pour permettre aux techniques des grammaires attribuées d'être utilisables sur des programmes fonctionnels, de déterminer une traduction de programmes fonctionnels vers leur forme grammaire attribuée.

Une traduction de programmes fonctionnels en grammaires attribuées

Cette thèse a donc présenté un algorithme de transformation de programmes fonctionnels en grammaires attribuées équivalentes, FP-to-AG, dont le principe général est de considérer le paramètre syntaxique d'une fonction (sur lequel est effectué le *pattern matching*) comme la structure grammaticale sur laquelle s'appuie la grammaire attribuée. Les autres paramètres de la fonction sont alors considérés comme des attributs hérités du profil de cette grammaire attribuée résultante. Pour chaque constructeur de production de la grammaire, un attribut portant le nom de la fonction est défini comme étant le résultat de l'expression du programme fonctionnel correspondant à ce constructeur. Le schéma de récursion de cette expression du programme fonctionnel est ensuite transformé en un ensemble d'équations orientées.

Les paramètres de la fonction étant devenus des occurrences d'attributs attachés à des variables de *pattern*, leur propagation au sein des appels récursifs pour chaque *pattern* permet de transformer le flot de contrôle spécifié par récursion dans le programme fonctionnel initial en un flot de contrôle spécifié par l'ensemble des règles sémantiques générées. Afin que la grammaire attribuée générée soit bien formée, certains appels de fonctions peuvent ne pas être démontés complètement et sont alors laissés sous leur forme initiale.

Le but de cette transformation était de permettre un "transport" des techniques des grammaires attribuées sur les programmes fonctionnels. Aussi, la version de FP-to-AG présentée dans cette thèse est simple et basique. Elle permet de transformer une classe de programmes fonctionnels qui n'est déterminée que de manière constructive par l'algorithme de transformation. Il serait maintenant intéressant de déterminer plus formellement cette classe de programmes et de l'élargir, notamment par l'utilisation des grammaires attribuées dynamiques. Par ailleurs, une forme étendue de cette traduction FP-to-AG est en cours d'implantation à l'INRIA dans le cadre des travaux de Correnson, mais le prototype produit déjà l'ensemble des exemples qui sont présentés dans cette thèse.

La composition symbolique

Cette traduction FP-to-AG permet donc l'application des techniques de déforestation des grammaires attribuées sur les programmes fonctionnels qu'elle permet de transformer. Cependant, la composition descriptionnelle classique nécessite d'être appliquée sur la composition de deux grammaires attribuées complètes. Il semblait alors intéressant d'assouplir cette contrainte afin d'augmenter les sites d'application de son principe de base : la projection de schéma de calcul. En effet, des pas de projections plus “locaux” pouvaient être appliqués à l'intérieur même d'un ensemble de règles sémantiques, permettant ainsi de faire apparaître de nouveaux termes, sujets à d'autres phases de transformation.

L'opportunité offerte par le principe de l'évaluation symbolique du profil utilisée dans le cadre de FP-to-AG (transformation locale et symbolique sur une expression d'une équation orientée) a permis de développer la composition symbolique. Cette transformation de programme incorpore plusieurs mécanismes. En particulier, les pas de projections qu'elle utilise font apparaître de nouveaux termes sur lesquels le processus d'évaluation symbolique généralisée peut être appliqué. Ce dernier simplifie les expressions traitées et peut effectuer des pas d'évaluation partielle sur des termes finis. La composition symbolique ainsi obtenue est donc une extension de la composition descriptionnelle classique, tant dans les effets qu'elle produit que dans ses possibilités d'application.

La traduction FP-to-AG couplée avec la composition symbolique produit une nouvelle méthode de déforestation de programmes fonctionnels, qui sont d'abord traduits sous la forme de grammaires attribuées, puis transformés en des grammaires attribuées déforestées. Les grammaires attribuées obtenues peuvent finalement être retranscrites en programmes fonctionnels par une transformation telle que celle suggérée par Johnsson, ou en générant les fonctions de visites permettant d'évaluer ces grammaires attribuées. En utilisant les techniques spécifiques des grammaires attribuées, cette nouvelle méthode de déforestation est capable de déforester des programmes fonctionnels pour lesquels les autres méthodes de déforestation échouent. En particulier, les structures intermédiaires construites dans des paramètres d'accumulation des programmes fonctionnels peuvent être éliminées. Par exemple, la déforestation par composition symbolique de l'inversion de la liste des feuilles d'un arbre binaire produit un programme qui construit exactement autant de constructeurs de liste que l'arbre binaire a de feuilles, au lieu du double dans le programme initial.

Par ailleurs, les grammaires attribuées déforestées par la composition symbolique contiennent fréquemment de nombreuses règles de copie qui ne font que de propager des valeurs le long des structures. Dans les cas où le changement de l'ordre de parcours des structures examinées par les programmes est envisageable, des analyses d'éliminations de règles de copies spécifiques aux grammaires attribuées peuvent supprimer ces parcours inutiles, simplifiant ainsi le résultat déforesté. Ces optimisations permettent par exemple d'aboutir à un programme de simple recopie de liste après déforestation de la double inversion de liste.

10.4 Conclusion et travaux futurs

À partir de l'étude comparative des différentes techniques développées dans un même objectif, cette thèse propose une nouvelle méthode qui permet de résoudre des problèmes pour lesquels les méthodes existantes étaient impuissantes. Cependant, les algorithmes proposés sont illustrés sur des exemples d'école. Même s'il prouvent une supériorité théorique et intrinsèque de la composition symbolique sur les méthodes calculationnelles étudiées dans le cadre d'une classe de programmes, la méthode de déforestation proposée reste sujette à la caution d'une utilisation à grande échelle dans un système de transformation de programmes. Le problème qui se pose dans des termes précis est alors le suivant : à quoi peut ou doit servir la déforestation et est-elle utile ?

Des implantations de la déforestation assez peu satisfaisantes

En fait, la plupart des systèmes proposés sont motivés en tant qu'optimisation de compilateur, mais très peu de techniques de déforestation ont été réellement implantées.

Un des rares exemples est celui de la règle `foldr/build` de Gill *et al.* implanté dans le compilateur GHC de Haskell. Dans ce cas, le gain d'efficacité rapporté par cette technique en tant qu'optimisation du compilateur est en moyenne de 3% en temps d'exécution, de moins de 5% en occupation du tas et nécessite une augmentation de la taille maximale des piles notable [Gil96]. Ces mesures ayant été réalisées sur un ensemble de programmes réaliste, elles ont conduit les auteurs à se poser la question de l'intérêt de cette technique. La réponse est qu'en tant qu'optimisation de programme incluse dans un compilateur, la déforestation de Gill *et al.* n'apporte une réelle amélioration que si le programmeur est conscient de son existence et spécifie délibérément ses programmes en utilisant largement des compositions de listes, ce qui dans la réalité est assez peu courant.

Concernant le système de fusion d'hylomorphismes HYLO, dont une version préliminaire a été implantée par Onoue [OHIT97], il n'existe pas a ma connaissance de version diffusée de ce prototype, ni de résultats de mesures. Il est donc difficile de se prononcer sur son efficacité pratique.

La composition descriptionnelle entre deux grammaires attribuées a été implantée par Roussel [Rou94] dans le système de traitement de grammaires attribuées FNC-2. Par ailleurs, les travaux actuels de Correnson sur une reformulation du formalisme des grammaires attribuées sous la forme de sémantique équationnelle comportent des prototypes d'implantations de la traduction FP-to-AG et de la composition symbolique. À lors actuelle, le système n'est pas en mesure de pouvoir être évalué en terme d'efficacité mais nous espérons y parvenir assez rapidement. Même si ce prototype étend assez largement les travaux qui sont présentés dans cette thèse, il pourrait donner des informations quantitatives intéressantes, permettant au moins de comparer notre approche à celle implantée dans GHC.

Un outil logiciel de spécialisation de composants

Néanmoins, le problème auquel le système implanté dans GHC a été confronté est riche d'enseignement : sur des programmes développés par un utilisateur et en tant qu'optimisation de compilateur, l'efficacité de la déforestation est loin d'être flagrante.

En réalité, il semble qu'elle possède un plus grand potentiel d'efficacité et d'utilité sur des programmes plus naïfs que ceux qu'un programmeur écrit naturellement. Par exemple, dans le cadre de programmes générés automatiquement, la composition symbolique présentée dans cette thèse peut être vue comme un outil de spécialisation de composants logiciels, par composition de spécifications. Des travaux [DPRJ97, CDPR98a, CDPR98b] ont déjà pu comparer les méthodes de généricité dans les grammaires attribuées avec la programmation polytypique dans les langages fonctionnels [JJ96, JJ97] ou la programmation adaptive des langages à objets [LL95, Lie96, PPSL96]. L'intérêt croissant pour les méthodologies de développement de programmes génériques par composition de briques logicielles et pour la conception semi-automatique de petits langages dédiés à des applications spécifiques, constitue un cadre dans lequel la déforestation possède un potentiel d'intérêt, en tant que technique de spécialisation et d'optimisation de programmes générés automatiquement.

Ce domaine de l'aide au développement de logiciel donne aux méthodes de déforestation et en particulier à la composition symbolique des champs d'applications, à un niveau "méta", où elles pourront probablement être plus efficaces que sous la forme d'optimisation classique de compilateurs. Cette même approche, d'utilisation dédiée à la génération de programmes d'une technique initialement prévue comme optimisation de compilateur, fait actuellement ses preuves dans le contexte de l'évaluation partielle vue comme un outil de spécialisation de programmes [Con96, TMC97].

D'autres champs d'investigation

Dans de nombreux domaines, il apparaît que les grammaires attribuées peuvent permettre de modéliser des problèmes par leurs spécifications déclaratives et dirigées par une syntaxe. C'est le cas par exemple dans la modélisation des architectures de développement logiciel par composants et par *design patterns* [BG96, MWSS96, Hed97], ou dans les bases de données [NdB98, FMS93]. Les travaux rapportés dans cette thèse, au sujet de la déforestation et de l'utilisation du formalisme des grammaires attribuées pour permettre l'utilisation de techniques existantes (la composition descriptionnelle) dans le contexte des programmes fonctionnels, peuvent être étendus dans plusieurs directions.

D'une part, il serait intéressant d'étudier et d'utiliser ces techniques de déforestation des compositions dans le cadre d'autres paradigmes de programmation, comme par exemple la programmation par objets. Des premiers liens [CDPR98a, CDPR98b] ont déjà pu être établis avec les *tree traversal* et la programmation adaptive qui suscite actuellement beaucoup d'intérêt dans la communauté objet.

D'autre part, d'autres techniques déjà développées pour le formalisme des grammaires attribuées pourraient être étudiées dans le but d'être appliquées à d'autres domaines. Je pense par exemple aux travaux sur l'évaluation parallèle ou sur l'évaluation incrémentale des grammaires attribuées, dont le transfert dans le domaine des bases de données semble prometteur.

Bibliographie

[Alb91] Henk Alblas. Attribute evaluation methods. In Alblas and Melichar [AM91], pages 48–113. Prague.

[AM91] Henk Alblas and Bořivoj Melichar, editors. *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.* Springer-Verlag, New York–Heidelberg–Berlin, June 1991. Prague.

[Att89] Isabelle Attali. *Compilation de programmes TYPOL par attributs sémantiques.* PhD thesis, Université de Nice, April 1989.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bel91a] F. Bellegarde. ASTRE, a Transformation System using Completion. Technical report, 1991.

[Bel91b] F. Bellegarde. Program Transformation and Rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, volume 488 of *Lect. Notes in Comp. Sci.*, pages 226–239. Springer-Verlag, 1991.

[Bel93] F. Bellegarde. A transformation system combining partial evaluation with term rewriting. In *Higher Order Algebra, Logic and Term Rewriting (HOA '93)*, volume 816 of *Lect. Notes in Comp. Sci.*, pages 40–58. Springer-Verlag, September 1993.

[Bel95a] F. Bellegarde. ASTRE: Towards a Fully Automated Program Transformation System. In *Proceedings of the sixth conference on Rewriting Techniques and Applications*, volume 914 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1995.

[Bel95b] F. Bellegarde. Automatic Synthesis by Completion. In *Journées Francophones sur les Langages Applicatifs*, INRIA, collection didactiques, 1995.

[BG96] Don Batory and Bart J. Geraci. Validating component compositions in software system generators. In *International Conference on Software Reuse*, Florida, 1996.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversal of data. *Acta Informatica*, 21:239–250, 1984.

[BM94] Richard Bird and Oege De Moor. Relational program derivation and context-free language recognition. In A. W. Roscoe, editor, *A Classical Mind: Essays dedicated to C.A.R. Hoare*, pages 17–35. Prentice Hall International, 1994.

[Boc76] Gregor V. Bochmann. Semantic evaluation from left to right. *Communications of the Association for Computing Machinery*, 19(2):55–62, February 1976.

[Boy96] John Boyland. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, January 1996.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM Press, 1993.

[CDPR98a] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic programming by program composition (position paper). In *Workshop on Generic Programming*, Marstrand, Sweden, June 1998. conjunction with MPC'98.

[CDPR98b] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Schéma générique de développement par composition. In *Approches Formelles dans l'Assistance au Développement de Logiciel AFADL'98*, Poitiers - futuroscope, 1998.

[CFZ82] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17(2 and 3):163–191 and 235–257, 1982. part I and II See also: rapport 8008, University de Bordeaux I (April 1980).

[CH79] R. Cohen and E. Harry. Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. In *6th ACM Symp. on Principles of Progr. Languages*, pages 121–134. ACM press, San Antonio, TX, January 1979.

[Chi94] W. N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.

[CK95] W.-N. Chin and S.-C. Khoo. Better consumers for deforestation. In S. Doaitse Swierstra, editor, *Programming Languages: Implementations, Logic and Programs (PLILP '95)*, volume 982 of *LNCS*, pages 223–240. Springer-Verlag, 1995.

[CM76] Laurian M. Chirica and David F. Martin. An algebraic formulation of knuthian semantics. In *17th IEEE Conf. on Foundations of Comput. Sc.*, pages 127–136. Houston, TX, October 1976.

[CM79] Laurian M. Chirica and David F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979. See also: report TRCS78-2, Dept. of Elec. Eng. and Computer Science, University of California, Santa Barbara, CA (October 1978).

[Con96] Charles Consel. A framework of application generator design. Technical Report 3005, INRIA, 1996.

[Der88] N. Dershowitz. Completion and its applications. In *Resolution of Equations in Algebraic Structures*, Academic Press, New York, 1988.

[DJL84] Pierre Deransart, Martin Jourdan, and Bernard Lorho. Speeding up circularity tests for attribute grammars. *Acta Informatica*, 21:375–391, December 1984. See also: rapport RR 211, INRIA, Rocquencourt (May 1983).

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lect. Notes in Comp. Sci.* Springer-Verlag, New York–Heidelberg–Berlin, August 1988.

[DPRJ97] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

[EF82] Joost Engelfriet and Gilberto Filé. Simple multi-visit attribute grammars. *Journal of Computer and System Sciences*, 24(3):283–314, June 1982. See also: memorandum 314, Onderafdeling der Informatica, Tech. Hogeschool Twente (August 1980).

[Eng84] Joost Engelfriet. Attribute grammars: Attribute evaluation methods. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, New York, 1984.

[Fan72] Isu Fang. *FOLDS, a Declarative Formal Language Definition System.* Ph.D. thesis, Comp. Sc. Department, Stanford University, December 1972. Abstract in: t Séminaires Structure et Programmation des Calculateurs 1973, ed. M. Kronental and Bernard Lorho, INRIA, Rocquencourt, pp. 275-290 (1973).

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 85–98. ACM press, Palo Alto, CA, June 1986. Published as ACM SIGPLAN Notices, volume 21, number 7.

[Fil83] Gilberto Filé. Interpretation and reduction of attribute grammars. *Acta Informatica*, 19:115–150, 1983. See also: memorandum 359, Onderafdeling der Informatica, Tech. Hogeschool Twente (1981).

[FMS93] L. Fegaras, D. Maier, and T. Sheard. Specifying rule-based query optimizers in a reflective framework. In *Third International Conference on Deductive and Object-Oriented Databases*, pages 146–168, Phoenix, Arizona, December 1993.

[Fok95] Maarten M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1995.

[FSS92] Leonidas Fegaras, Tim Sheard, and David Stemple. Uniform traversal combinators: Definition, use and properties. In *11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lect. Notes in Comp. Sci.*, pages 148–162, Saratoga Springs, New York, June 1992. Springer-Verlag.

[FSZ94] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pages 21–32, Orlando, Florida, June 1994.

[Gan80] Harald Ganzinger. Transforming denotational semantics into practical attribute grammars. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, New York–Heidelberg–Berlin, 1980.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984. ACM press. Published as *ACM SIGPLAN Notices*, 19(6).

[Gie88] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[Gil96] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, Glasgow University, January 1996.

[GJ94] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

[GLJ93] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

[Ham96] G. W. Hamilton. Higher order deforestation. 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, September 1996. Springer-Verlag.

[Hed97] Görel Hedin. Language support for design patterns using attribute extension. In *Language Support for Design Patterns and Frameworks*, Jyväskylä, Finland, 1997. Workshop in conjunction with ECOOP'97.

[HIT96a] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeishi. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundations of Computer Science (MFCS'96)*, volume 1113 of *Lect. Notes in Comp. Sci.*, pages 407–418, Cracow, September 1996. Springer-Verlag.

[HIT96b] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeishi. Deriving structural hylomorphisms from recursive definitions. In *Proc.of the International Conference on Functional Programming (ICFP'96)*, pages 73–82, Philadelphia, May 1996. ACM Press.

[HJ91] G. Hamilton and S. B. Jones. Extending deforestation for first order functional programs. In *Glasgow Workshop on Functional Programming*, pages 134–145, 1991.

[HJW$^+$92] P. Hudak, S. L. Peyton Jones, P. L. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. S Nikhil, W. Partain, and J. Peterson. Report on the functional programming language haskell, version 1.2. SIGPLAN Notice 27, May 1992.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Jal83] Fahimeh Jalili. A general linear-time evaluator for attribute grammars. *ACM SIGPLAN Notices*, 18(9):35–44, September 1983.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *24th ACM Symp. on Principles of Programming Languages*, 1997.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Func. Prog. Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, New York–Heidelberg–Berlin, September 1987. Portland.

[Jou84a] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and Bernard Robinet, editors, *6th Int. Symp. on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer-Verlag, New York–Heidelberg–Berlin, April 1984. Toulouse.

[Jou84b] Martin Jourdan. Recursive evaluators for attribute grammars: an implementation. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 139–164. Cambridge University Press, New York, New York, 1984.

[Jou92] Martin Jourdan. *Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées.* Mémoire d'habilitation, Département de Mathématiques et d'Informatique, Université d'Orléans, 1992.

[JP89] Martin Jourdan and Didier Parigot. *The FNC-2 System User's Guide and Reference Manual.* INRIA, Rocquencourt, February 1989. release 0.4 This manual is periodically updated.

[JP90] Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes in Comp. Sci.*, pages 29–45, Paris, September 1990. Springer-Verlag.

[JP91] Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Evaluation Methods*, volume 545 of *Lect. Notes in Comp. Sci.*, pages 485–504, New York–Heidelberg–Berlin, June 1991. Springer-Verlag. Prague.

[JPJ$^+$90] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980. See also: Bericht 7/78, Institut für Informatik II, University Karlsruhe (1978).

[Kas87] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24(6):633–652, November 1987.

[Kas91] Uwe Kastens. Implementation of visit-oriented attribute evaluators. In Alblas and Melichar [AM91], pages 114–139. Prague.

[KL94] Richard Kieburtz and Jeffrey Lewis. Algebraic design language. Technical report, Oregon Graduate Institute, 1994.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[KR79] Ken Kennedy and J. Ramanathan. A deterministic attribute grammar evaluator based on dynamic sequencing. *ACM Trans. Progr. Languages and Systems*, 1(1):142–160, July 1979.

[Le 95] Gilles Le Bâtard. Réalisation dans le système FNC-2 d'un traducteur vers ML. rapport de stage de maîtrise, IFI, Université de Marne-la-Vallée, July 1995.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[LL95] Cristina Videira Lopes and Karl J. Lieberherr. AP/S++: A CASE-study of a MOP for purposes of software evolution. Technical Report NU-CCS-95-?, Xerox PARC and Northeastern University, November 1995.

[Lor74] Bernard Lorho. De la définition à la traduction des langages de programmation: méthode des attributs sémantiques. thèse d'état, University Paul Sabatier, Toulouse, November 1974.

[Lor77] Bernard Lorho. Semantic attributes processing in the system DELTA. In A. Ershov and Cornelius H. A. Koster., editors, *Methods of Algorithmic Language Implementation*, volume 47 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, New York–Heidelberg–Berlin, 1977.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 314–323, La Jolla, CA, USA, 1995. ACM Press.

[Mad80] Ole L. Madsen. On defining semantics by means of extended attribute grammars. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 259–299. Springer-Verlag, New York–Heidelberg–Berlin, 1980. See also: report DAIMI PB-109, Comp. Sc. Department, Aarhus University (January 1980).

[Mal90] Grant Malcolm. *Algebraic Types and Program Transformation.* PhD thesis, University of Gronigen, 1990.

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Mee95] Lambert Meertens. Category theory for program construction by calculation, 1995.

[MFP91] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144, Cambridge, September 1991. Springer-Verlag.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA'95*, 1995.

[MW75] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. *Artif. Intel. J.*, 6(2):175–208, 1975.

[MW79] Z. Manna and R. Waldinger. Synthesis: Dreams =¿programs. *IEEE Transaction on Software Engineering*, 5(4):157–164, 1979.

[MWSS96] T . Murer, A. Würtz, D. Scherer, and D. Schweizer. GIPSY: Generating integrated process support systems - project overview. Ik-report no. 22, Swiss Federal Institute of Technology Zurick, December 1996.

[NdB98] Frank Neven and Jan Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *PODS '98. Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–17. ACM press, 1998.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *In Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997.

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

[Par88] Didier Parigot. *Transformations, Évaluation Incrémentale et Optimisations des Grammaires Attribués: Le Système FNC-2.* PhD thesis, Université de Paris-Sud, Orsay, 1988.

[PDRJ95a] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. Rapport de Recherche 2662, INRIA, October 1995.

[PDRJ95b] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *journées du GDR de programmation*, Grenoble, November 1995.

[PDRJ96] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *Journées Francophones des Langages Applicatifs*, pages 263–279, Val-Morin, Québec, January 1996.

[PP96] A. Pettorossi and M. Proietti. A theory of logic program specialization and generalization for dealing with input data properties. volume 1110 of *Lecture Notes in Computer Science*, pages 386–408, Dagstuhl Castle, Germany, 1996. Springer.

[PP97] Alberto Pettorossi and Maurizio Proietti. Flexible continuations in logic programs via unfold/fold transformations and goal generalization. In *2nd ACM Sigplan Workshop on Continuations (CW'97)*, La Sorbonne, Paris, France, January 1997.

[PPR97] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Reducing nondeterminism while specializing logic programs. In *24th ACM Symp. on Principles of Programming Languages (POPL'97)*, La Sorbonne, Paris, France, January 1997.

[PPSL96] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag.

[PRJD96a] 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, September 1996. Springer-Verlag.

[PRJD96b] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996.

[PS87] A. Pettorossi and A. Skowron. Higher Order Generalization in Program Derivation. In Springer Verlag, editor, *Proceedings of TAPSOFT '87: Theory and Practice of Software Development*, volume 250 of *Lect. Notes in Comp. Sci.*, 1987.

[Red88] U. S. Reddy. Transformational derivation of programs using the focus system. In *Symposium Practical Software Development Environments*, ACM, pages 163–172, December 1988.

[Rou94] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[Sei96] Helmut Seidl. Integer constraints to stop deforestation. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 326–340, Linköping, Sweden, April 1996. Springer.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1994.

[Sør94] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 1994.

[SS97a] H. Seidl and M. H. Sørensen. Constraints to stop deforestation. *Science of Computer Programming*, 1997. to appear.

[SS97b] H. Seidl and M. H. Sørensen. Constraints to stop higher-order deforestation. In *24th ACM Symp. on Principles of Programming Languages*, pages 400–413, 1997.

[SV91] S. Doaitse Swierstra and Harald H. Vogt. Higher Order Attribute Grammars. In Alblas and Melichar [AM91], pages 256–296. Prague.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 306–313, La Jolla, CA, USA, 1995. ACM Press.

[TMC97] S. Thibault, R. Marlet, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *the IEEE conference on Automated Software Engineering (ASE'97)*, Lake Tahoe, Novembre 1997.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conf. on Progr. Lang. Design and Implementation*, pages 131–145, Portland, OR, July 1989. ACM press. Published as *ACM SIGPLAN Notices*, 24(7).

[Wad84] Philip Wadler. Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas,, 1984.

[Wad85] Philip Wadler. Listlessness is better than laziness II: composing listless functions. In *Workshop on Programs as Data Objects*, volume 217 of *LNCS*, Copenhagen, Denmark, October 1985. Springer-Verlag.

[Wad88] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, March 1988. Springer-Verlag.

[Wad89] Philip Wadler. Theorems for free! In *Conf. on Func. Prog. Languages and Computer Architecture*, London, September 1989.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science*, volume 73, pages 231–248, 1990. (Special issue of selected papers from 2'nd European Symposium on Programming).

Contribution to the relation between attribute grammars and functional programming

Abstract

Software engineering has to reconcile modularity, that is required for development and maintenance phases, with efficiency, obviously essential in the practical implementation of applications. This dilemma implies that methods and techniques must be developed in order to increase the efficiency of modular programs.

The aim of deforestation transformations is to discard intermediate data structures that appear when software components are composed. Thus, these transformations are of great interest, especially to attribute grammar and functional programming communities. In spite of the variety of formalisms they used, this thesis compares several existing techniques and develops a new general deforestation method drawn from their advantages.

First, a natural attribute grammar extension is introduced, allowing a larger functional programming class to be expressed. Then, dynamic attribute grammars are no more tied to concrete trees, to direct computations and transformations. Nevertheless, they could always be evaluated with classical attribute grammar evaluation methods.

Next, the main functional deforestation methods (Wadler's algorithm, elimination of foldr/build rule, normalization of folds, fusion of hylomorphisms) are studied and compared with the descriptional composition of attribute grammars. Limitations of each method are established and allow suitable features for these program transformations to be determined.

Finally, a new deforestation method is introduced. The symbolic composition uses the power of attribute grammar formalism and also includes a partial evaluation mechanism. This general technique can be applied to attribute grammars or to functional programs and it deforests programs for which existing methods were insufficient.

Key words :
Attribute grammars, functional programming, structure directed programming, algebraic formalisms, program transformations, deforestation, genericity, static analysis.

Résumé

L'ingénierie du logiciel doit concilier, d'une part, la modularité requise par les phases de développement et de maintenance et, d'autre part, l'efficacité indispensable dans la mise en œuvre des applications. Ce dilemme nécessite des méthodes et des techniques de transformation permettant d'accroître l'efficacité des programmes modulaires.

La *déforestation*, qui consiste à éliminer les structures intermédiaires apparaissant lors de la composition des différentes parties d'un programme, a suscité beaucoup d'intérêt, notamment en grammaires attribuées et en programmation fonctionnelle. En dépit de la diversité des formalismes utilisés, cette thèse compare les différentes techniques existantes et s'inspire de leurs atouts pour développer une nouvelle méthode de déforestation plus générale.

Tout d'abord, une extension naturelle des grammaires attribuées est introduite pour permettre de représenter une plus large classe de programmes fonctionnels. Les *grammaires attribuées dynamiques* peuvent se passer de la présence physique d'un arbre pour guider les calculs et les transformations, mais bénéficient des méthodes classiques d'évaluation des grammaires attribuées.

Ensuite, les principales méthodes fonctionnelles de déforestation (algorithme de Wadler, règle d'élimination *foldr/build*, normalisation des *folds*, fusion d'hylomorphismes) sont étudiées et comparées avec la composition descriptionnelle des grammaires attribuées. Les limitations de chaque méthode sont établies et permettent de déterminer les atouts nécessaires pour ces transformations de programmes.

Finalement, une nouvelle méthode de déforestation est proposée. La *composition symbolique* utilise la puissance du formalisme des grammaires attribuées et incorpore un mécanisme d'évaluation partielle. Cette technique générale peut être appliquée sur des grammaires attribuées ou sur des programmes fonctionnels et permet de déforester des programmes pour lesquelles les méthodes existantes restaient impuissantes.

Mots clés :
Grammaires attribuées, programmation fonctionnelle, programmation dirigée par la structure, formalismes algébriques, transformations de programmes, déforestation, généricité, analyses statiques.

INRIA

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex