Mémoire

présenté à

l'Université d'Orléans

par

Martin JOURDAN

en vue d'obtenir

l'Habilitation à Diriger des Recherches en Sciences

Spécialité : Informatique

Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées

Mémoire présenté en séance publique
le 16 Avril 1992
devant le jury composé de

MM.	Bernard LORHO	Président
	Henk ALBLAS	Rapporteurs
	Paul FRANCHI-ZANNETTACCI	
Mme	Marie-Claude GAUDEL	Examinateurs
MM.	François LE BERRE	
	Patrick COUSOT	

Composition du dossier

Outre le présent mémoire, le dossier que je présente en vue d'obtenir l'Habilitation à Diriger des Recherches en Sciences comprend les publications suivantes :

- M. Jourdan : "An Optimal-Time Recursive Attribute Evaluator" [Jou84a]
- M. Jourdan : "Strongly Non-Circular Attribute Grammars and their Recursive Evaluation" [Jou84b]
- P. Deransart, M. Jourdan & B. Lorho : "Speeding up Attribute Grammar Non-Circularity Tests" [DJL84]
- P. Deransart, M. Jourdan & B. Lorho : *Attribute Grammars: Definitions, Systems and Bibliography* [DJL88]
- M. Jourdan & D. Parigot : "Advances in Attribute Grammar Technology" [JoP89]
- M. Jourdan & D. Parigot : *The* FNC-2 *System User's Guide and Reference Manual* [JoP92]
- M. Jourdan & D. Parigot : "Techniques for Improving Grammar Flow Analysis" [JoP90a]
- M. Jourdan, D. Parigot, C. Julié, C. Le Bellec & O. Durin : "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System" [JPJ90]
- M. Jourdan, C. Le Bellec & D. Parigot : "The OLGA Attribute Grammar Description Language: Design, Implementation and Evaluation" [JLP90]
- M. Jourdan & D. Parigot : "Application Development with the FNC-2 Attribute Grammar System" [JoP90b]
- P. Franchi-Zannettacci & M. Jourdan : "Conception des environnements de programmation, première partie : Attribute Grammars" [FrJ91]
- M. Jourdan : "A Survey of Parallel Attribute Evaluation Techniques" [Jou91]
- M. Jourdan & D. Parigot : "Internals and Externals of the FNC-2 Attribute Grammar System" [JoP91]

Parmi mes autres publications significatives mais un peu trop éloignées du sujet de ce mémoire pour faire partie du dossier, je citerais [BDJ83, BDJ84, BoJ87, Jou84].

Remerciements

Une fois n'est pas de coutume, je ne commencerai pas cette liste de remerciements par les membres du jury (j'espère qu'ils me pardonneront !) mais par celui qui a d'abord été mon élève, puis mon bras droit et maintenant mon égal (au moins sur le plan scientifique) : je veux parler de Didier Parigot, avec qui je travaille depuis bientôt sept ans. FNC-2 est notre œuvre commune et je dois humblement reconnaître que, dans FNC-2 aussi bien que dans nos publications communes et même dans ce mémoire, je ne suis guère capable de séparer clairement la part de Didier de la mienne, sinon que, pour une question de dons, il a manié plutôt le clavier et moi plutôt la plume. Notre collaboration a, je crois, été productive et, en tout état de cause, elle a été très enrichissante pour moi. J'espère qu'elle se poursuivra longtemps encore. Que Didier trouve ici l'expression de toute la joie que j'ai eue et que j'ai encore à travailler avec lui.

Bien qu'il ait déjà quitté l'INRIA à l'époque, c'est Bernard Lorho qui m'a réellement introduit aux grammaires attribuées, dont il a été le pionnier en France. S'il existe un qualificatif décrivant notre relation, c'est certainement celle de mentor à disciple : il me prodigue constamment conseils, encouragements et opportunités de progresser (et depuis qu'il est revenu diriger le centre INRIA de Rocquencourt, il se décharge sur moi d'un certain nombre de tâches qu'il ne veut ou peut pas assurer, c'est dire !). Je suis donc très heureux qu'au moment où il quitte le Département d'Informatique de l'Université d'Orléans, qu'il a animé pendant de longues années, il ait accepté et trouvé le temps de juger mon travail et de présider ce jury.

Mes pas croisent de plus en plus fréquemment ceux de Henk Alblas, aussi bien sur le plan scientifique (par exemple notre intérêt commun pour les systèmes de transformation d'arbres attribués ou l'évaluation d'attributs sur machines parallèles) que sur le plan physique (groupe 2.4 de l'IFIP), et à chaque fois je suis agréablement surpris de notre communauté de vues. Henk est un membre éminent de l'école européenne sur les grammaires attribuées et je suis heureux qu'il ait accepté d'évaluer mon travail malgré le barrage de la langue. *Dank je wel, Henk*! (ce qui résume à peu près ma connaissance du néerlandais).

Les contributions de Paul Franchi-Zannettacci à l'étude des grammaires attribuées [CoF82, Fra82] ont formé la base de mes premiers travaux [Jou84b, Jou84] mais, curieusement, Paul n'avait jamais eu encore l'occasion de porter un jugement "officiel" sur mes œuvres ; cette injustice est aujourd'hui réparée et je le remercie d'avoir accepté cette charge avec autant d'enthousiasme. Je ne doute pas qu'il saura me remercier, par exemple en me nommant rapporteur des thèses de ses étudiants...

C'est Marie-Claude Gaudel qui m'a donné la possibilité de faire mes premiers pas dans la recherche, en m'accueillant dans le projet "Langages et Traducteurs", qu'elle dirigeait alors à l'INRIA, il y a déjà plus de dix ans. Depuis, nos chemins se sont séparés, mais j'ai toujours gardé pour elle beaucoup de reconnaissance et une certaine tendresse. J'espère lui montrer, avec ce mémoire et l'ensemble de mon œuvre, que le temps qu'elle a passé à me porter conseil n'aura pas été inutile.

J'apprécie la présence dans ce jury de François Le Berre, qui témoigne ainsi de tout l'intérêt que le Département d'Informatique de l'Université d'Orléans porte à mes travaux et, par voie de conséquence, à mon enseignement. Bien qu'il ne soit pas "officiellement" un spécialiste des grammaires attribuées, je pense que ce travail peut l'intéresser personnellement et, qui sait, lui suggérer de nouvelles pistes pour de futures recherches.

Patrick Cousot a joué à l'École Polytechnique, où j'enseigne depuis un certain nombre d'années maintenant, un rôle semblable à celui de Bernard Lorho vis-à-vis de ma recherche. Il m'a servi de guide en me donnant beaucoup de conseils pédagogiques, même si ce ne fut pas toujours de manière très directe ; ceux-ci me furent en tout cas très précieux. En outre, c'est un spécialiste mondialement reconnu de l'analyse statique. Ses travaux ne s'appliquent certes pas directement au domaine des grammaires attribuées, mais je pense qu'un certain nombre de parallèles entre nos travaux s'imposent, et nous permettront de développer une collaboration que j'ai toujours souhaitée.

Les travaux décrits dans ce mémoire sont en réalité l'œuvre de toute une équipe qui a compris ou comprend, outre Didier Parigot et moi-même, qui la dirigeons, Catherine Julié, Joël Bonnet, Christine Ayrault, Souad Taouil, Olivier Durin, Étienne Planes, Carole Le Bellec, Catherine Zylberman, Bruno Marmol, David Devillard, Gilles Roussel, Roberto Gomez Cardenas et Jean-Philippe Jouve.

Je voudrais enfin remercier tous les membres de l'ex-projet "Langages et Traducteurs" et du nouveau projet "ChLoÉ" de l'INRIA, avec qui je travaille en toute confiance et dans une ambiance tout à fait sympathique et productive. Parmi ceux-ci, je voudrais particulièrement saluer, d'une part, les "anciens", qui m'ont si bien accueilli à l'INRIA et qui m'ont permis de travailler dans un environnement scientifique et technique tout à fait motivant et, d'autre part, ceux qui m'ont fait confiance récemment pour qu'à mon tour, j'aie la responsabilité de fournir à d'autres chercheurs l'environnement scientifique et technique dans lequel ils peuvent développer une recherche fructueuse.

Un dernier mot pour, d'une part, Farida Ennebati et Agnès Frapy, dont la participation matérielle à la préparation de ce mémoire a été essentielle, d'autre part, pour tous les organismes qui m'ont permis de me dégager des obligations que j'avais vis-à-vis d'eux pendant le temps nécessaire à la rédaction de ce mémoire, ensuite, pour mon médecin, qui m'a permis de dormir pendant cette même période et, enfin et surtout, pour mon épouse Nicole, qui a pris en charge presque totalement depuis longtemps l'environnement familial, qu'elle rend merveilleusement confortable, et qui m'a ainsi permis de supporter le stress engendré par l'épreuve qui trouve son aboutissement dans le présent mémoire, ainsi que dans la soutenance correspondante.

Table des matières

Plan

Intro: les GA comme outil de spécification. Problème: ça ne passe pas chez les industriels. Solutions: 1: augmenter l'efficacité, 2: augmenter le niveau d'abstraction (pouvoir d'expression) et la facilité d'utilisation (programming in the large). Comment réaliser les deux à la fois? par l'analyse statique.

Il se trouve que le formalisme des GA est suffisamment simple pour qu'une analyse statique, même simple, puisse fournir des résultats très précis, voire exacts; par exemple, le test de circularité bien connu est un test statique qui fournit des résultats exacts.

Analyse statique pour la construction des évaluateurs: séquences de visite (meilleur compromis puissance/efficacité, un seul test par nœud), transformations NC $\rightarrow$ FNC $\rightarrow$ l-ordonnés, optimisations mémoire, évaluateurs // à séquences de visite (bonne/meilleure efficacité, au sens speedup/nb.procs), méta-composition des GCA + plus optimisations source à source. Améliorations de l'analyse statique: GFA.

Facilité d'utilisation: GCA, modularité; construction automatique des règles de copie et même plus (OLGA++); GA génériques: OLGA: typage fort, inférence de types; rapprochement/unification des domaines syntaxique et sémantique.

Future work: $X.a := f(Y.b) \Rightarrow v := f(v) \Rightarrow p(v)$; augmenter le pouvoir d'expression: GA dynamiques.

Conclusion: ça finit par venir (STERIA, Compare).

Chapitre 1

Introduction

Dans une communication invitée qu'il a donnée au colloque WAGA en 1990 [Knu90], Donald Knuth décrit de manière savoureuse le processus intellectuel qui l'a amené à inventer les grammaires attribuées. Il explique qu'il cherchait essentiellement une notation pour pouvoir spécifier, de manière agréable et formelle à la fois, la sémantique des langages de programmation. L'article qu'il a publié en 1968 [Knu68], dans lequel il décrit effectivement la théorie quasiment complète des grammaires attribuées, ainsi qu'une première méthode d'évaluation et un premier test de circularité (légèrement faux d'ailleurs), a eu depuis un retentissement considérable ; j'en prendrai pour preuve la liste de plus de 600 références que nous avons rassemblée dans une étude sur ces grammaires attribuées en 1988 [DJL88], et qui continue à grossir régulièrement depuis cette date, quoiqu'à un rythme un peu moins soutenu.

Cependant, à côté de leur utilisation comme méthode de spécification formelle (à laquelle je ne crois guère, d'ailleurs), de nombreux travaux ont été consacrés à l'utilisation des grammaires attribuées comme un outil de programmation réellement utilisable en pratique ; je donne dans l'annexe A un exemple de développement pas à pas d'une petite grammaire attribuée (celle que Knuth avait lui-même donnée dans [Knu68]), qui démontre — autant que faire se peut en trois pages — la qualité des grammaires attribuées comme outil de programmation pour des calculs dirigés par la syntaxe. En examinant la façon dont cette grammaire est développée, ainsi que le résultat de ce processus, on constate facilement que les grammaires attribuées sont intéressantes, en tant qu'outil de programmation, sur au moins quatre plans.

- C'est une méthode *déclarative* : on ne spécifie que *ce* qu'il faut calculer et pas *comment* il faut le calculer, en particulier pas dans quel ordre il faut calculer les attributs.

- C'est une méthode qui est *structurée* par la grammaire sous-jacente : chaque production constitue un *bloc* qui ne s'interface avec les autres blocs que de manière bien spécifiée, par les non-terminaux et leurs attributs.

- C'est une méthode dans laquelle chacun de ces blocs ne peut pas voir vers l'extérieur, c'est-à-dire qu'il y a une *localité de référence* très forte.

- Enfin, et cela n'est apparu que relativement récemment, c'est une méthode de développement *incrémentale* [JoP90b], c'est-à-dire qu'on peut développer une grammaire attribuée petit à petit, et tester chaque partie de spécification indépendemment des autres ; je crois que c'est un très grand avantage par rapport à d'autres méthodes.

De nombreux travaux ont montré d'ailleurs que le gain de productivité du programmeur apporté par la méthode des grammaires attribuées sur des exemples réels, comme des parties de compilateurs, est tout à fait impressionnant ; il suffit par exemple de l'ordre d'un homme-mois pour développer une partie avant de compilateur pour un langage relativement simple, et au pire un an, à comparer aux plusieurs centaines d'hommes-années qu'a demandées la réalisation (à la main) du compilateur Ada de la société Alsys.

Enfin, bien entendu, une des qualités les plus importantes des grammaires attribuées, comme outil de spécification et de programmation, est qu'elles forment une méthode *exécutable* : de nombreux travaux ont montré qu'il était possible, et même relativement facile, de produire automatiquement, à partir d'une grammaire attribuée décrivant un calcul donné, un programme qui réalise ce calcul.

Malheureusement, force est de constater que, malgré toutes ces qualités, les grammaires attribuées ne sont pas ou quasiment pas utilisées en milieu industriel. Les deux seuls systèmes que je connaisse qui soient plus ou moins effectivement utilisés en milieu industriel sont d'une part Mira, le successeur de Lila, dont on peut trouver une description dans [DJL88], mais qui est à peine plus puissant que *Yacc* avec des attributs purement synthétisés, et le système de Rodney Farrow, Linguist, autour duquel il a monté une petite entreprise [1].

Une des causes de cette situation pour le moins indésirable est que les industriels sont toujours un peu en retard sur la recherche [2]. Je pense cependant qu'il y a deux autres raisons vraiment importantes qui provoquent cette sous-utilisation. La première est que les évaluateurs, c'est-à-dire les programmes produits automatiquement à partir d'une GA pour réaliser le calcul qu'elle décrit, sont encore trop inefficaces pour être acceptables par les industriels. La seconde est que les GA montrent quelques faiblesses quand il s'agit de décrire des applications de grande taille, en particulier des applications qui nécessitent d'être développées de front par plusieurs programmeurs. Un exemple typique en est la GA décrivant la partie avant d'un compilateur Ada, écrite pour le système GAG, qui est une spécification totalement monolithique, en un seul fichier, formant un livre de plus de *500 pages* !

Les deux seules solutions que je vois pour améliorer cet état de fait sont, d'une part, d'augmenter l'efficacité des évaluateurs produits et, d'autre part, d'augmenter la facilité d'utilisation des GA — en particulier en s'intéressant fortement à ces problèmes de génie logiciel qui sont couverts par l'expression *programming in the large* — ainsi que leur pouvoir d'expression. Comment atteindre ces deux buts à la fois, alors qu'ils paraissent quelque peu contradictoires [3] ? C'est *l'analyse statique de grammaires attribuées* qui fournit la solution : avec celle-ci on peut en effet, plus ou moins facilement, rassembler une quantité importante d'informations à partir de la seule donnée d'une GA, ce qui permet ensuite, d'une part, d'optimiser l'évaluateur produit et, d'autre part, de simplifier le travail du programmeur en lui offrant la modularité, en diminuant la redondance nécessaire et, d'une manière générale, en inférant un certain nombre d'informations qu'il n'a pas à donner explicitement, ce qui lui facilite la tâche.

1. Cependant, à chaque fois que je le rencontre, il me dit bien que, d'une part, il ne fait que vivoter et que, d'autre part, ce n'est pas en vendant Linguist lui-même qu'il gagne quelque argent, mais en vendant des applications développées avec ce système.

2. Voir par exemple le temps qu'ont mis les générateurs d'analyseurs lexico-syntaxiques pour pénétrer ce milieu, alors que leur utilité est encore plus évidente que celle des GA.

3. La contradiction vient de ce qu'il est toujours facile d'augmenter l'efficacité en restreignant la puissance d'expression et en obligeant l'utilisateur à donner des informations supplémentaires qui facilitent le travail du compilateur.

Cette démarche n'est absolument pas nouvelle : l'analyse statique est utilisée depuis très longtemps dans les compilateurs des langages de programmation, et en particulier dans les compilateurs optimisants, sous forme d'analyse de flot de données classique [JoM81] ou d'interprétation abstraite [CoC77]. La thèse que je développe dans ce mémoire est cependant que, lorsqu'on l'applique aux grammaires attribuées, cette idée d'analyse statique fournit d'excellents résultats, qui dépassent de loin en tous cas ce qu'on peut faire sur des formalismes plus puissants. En effet le formalisme des grammaires attribuées et leur sémantique sont suffisamment simples pour qu'une analyse statique, même simple, puisse fournir des résultats extrêmement précis, voire exacts (non approximés) ; par exemple, je montre dans l'annexe B que le test de circularité bien connu est un exemple d'analyse statique qui fournit des résultats exacts sur le comportement dynamique d'une GA [4].

Le but de ce mémoire est donc, afin d'étayer cette thèse, d'exhiber un certain nombre d'exemples d'analyse statique sur les grammaires attribuées qui permettent, d'une part, d'améliorer les performances des évaluateurs et, d'autre part, d'améliorer la facilité de leur utilisation pour développer de grosses applications. La plupart de ces exemples d'analyse statique sont implantés dans le système de traitement de grammaires attribuées FNC-2, que nous développons à l'INRIA depuis une demi-douzaine d'années maintenant et qui, je crois pouvoir le dire sans paraître trop présomptueux, est actuellement le meilleur système de GA au monde.

Le plan de ce mémoire s'établit ainsi : les deux prochains chapitres sont consacrés aux exemple d'analyse statique que j'ai cités ci-dessus, le premier étant consacré essentiellement à l'optimisation des évaluateurs produits et le second à l'amélioration de la facilité d'utilisation des GA. Dans ce second chapitre je donne en outre quelques pistes pour de futurs travaux. Je montre enfin en conclusion que l'ensemble de ces résultats aboutit à un système qui est réellement utilisable et qui commence à être effectivement utilisé, au moins comme outil de prototypage rapide, chez quelques industriels.

Il n'est pas dans l'esprit de ce que doit être un mémoire d'habilitation de reproduire intégralement les publications jointes au dossier ; en conséquence, je ne ferai qu'esquisser certains développements et résultats, souvent à l'aide d'illustrations ; dans tous les cas, je renverrai le lecteur aux publications idoines. En outre, bien que je m'efforcerai de rester aussi didactique que possible, je supposerai que le lecteur est capable de me suivre sans trop de difficultés ; là encore, les publications jointes au dossier pourront se révéler utiles, ainsi que l'annexe B.

Je me suis efforcé d'utiliser le mot "nous" pour désigner tout notre groupe travaillant sur les GA à l'INRIA et le mot "je" quand il s'agit réellement de mes propres opinions. Malheureusement, il m'a été souvent très difficile d'établir une séparation nette entre les deux.

4. Dans cette annexe B je donne aussi quelques définitions et résultats fondamentaux sur les GA, qui peuvent aider le lecteur à comprendre la description du test de circularité, bien sûr, mais aussi les autres constructions plus ou moins développées dans ce mémoire. En outre, j'y insiste sur la dualité entre vue statique et vue dynamique des GA, qui est vraiment l'idée fondamentale sur laquelle repose toute ma thèse ; je crois donc que même un lecteur averti pourra trouver quelque intérêt à la lire.

Chapitre 2

Analyse statique des GA pour améliorer les évaluateurs

La thèse que je développe dans ce mémoire est que l'analyse statique de grammaires attribuées permet, entre autres choses, d'améliorer le comportement des évaluateurs que l'on peut produire à partir d'une grammaire attribuée donnée. C'est en fait sur ce sujet, même si ce n'est que rarement dit explicitement, que la plupart des chercheurs sur les GA ont travaillé. En effet, il a très vite été reconnu que la méthode naïve d'évaluation d'attributs proposée par Knuth lui-même [Knu68], qui est fondée sur la construction explicite du graphe de dépendance complet (voir annexe B) et le tri topologique de ce graphe, est totalement inefficace, aussi bien en temps qu'en place.

En fait, si l'on examine de près le comportement d'un évaluateur d'attribut, et si on le met en perspective avec la définition de la sémantique des GA (que je donne aussi en annexe B), il est facile de voir que, fondamentalement, tout évaluateur doit accomplir les trois tâches suivantes :

1. évaluer les règles sémantiques, vues comme des affectations ;
2. parcourir l'arbre de nœud en nœud, pour pouvoir calculer ou accéder à la valeur des attributs ;
3. savoir à tout instant où il en est, c'est-à-dire tenir à jour un certain nombre de structures de données qui lui permettent de déterminer quelle est la prochaine action (évaluation d'attribut ou mouvement dans l'arbre) à effectuer pour progresser vers son but final.

Manifestement, tout évaluateur doit effectuer les tâches 1 et 2, puisqu'elles constituent exactement la sémantique d'une GA. Donc, le seul moyen de gagner en efficacité est de réduire le plus possible le point 3. L'idée fondamentale pour atteindre ce but est (d'essayer) de remplacer l'ordre d'évaluation dynamique, induit par le tri topologique du graphe de dépendance complet, par un ordre plus statique, déterminé *a priori* ou par analyse statique de la GA elle-même.

Pour comprendre comment l'analyse statique peut aider à déterminer un ordre d'évaluation, il faut se reporter à la discussion sur les graphes de dépendance synthétisés et hérités de la section B.2.3. Il y est dit en effet que ces graphes représentent, en les résumant, des dépendances *réelles*, qui apparaissent effectivement dans les graphes de dépendance inférieur et supérieur d'un nœud particulier dans un arbre particulier, et constituent donc des contraintes sur l'ordre dans lequel on peut évaluer les attributs de ce nœud. En d'autres termes, pour un nœud donné dans un arbre donné, et de façon dynamique, le graphe de dépendance hérité (GDH) est la seule information qui soit nécessaire pour trouver un ordre d'évaluation dans le graphe de dépendance inférieur, et de même pour le graphe de dépendance synthétisé (GDS) et le graphe de dépendance supérieur. Il est alors clair que, si l'on veut déterminer statiquement un ordre "universel" d'évaluation des attributs d'un non-terminal, applicable à tous les nœuds de tous les arbres qui sont étiquetés par ce non-terminal, il faut au moins que cet ordre soit compatible avec l'union de tous les GDS et GDH de ce non-terminal (supposée définir un ordre partiel, ce qui n'est pas le cas pour toutes les GA non circulaires).

De nombreux travaux ont donc pris comme base d'autres genres de ces "graphes de dépendance de non-terminaux", plus simples, calculables statiquement et qui conduisent à des méthodes d'évaluation différentes, plus efficaces mais qui s'appliquent à des classes de GA plus petites.

En particulier, les évaluateurs applicatifs pour la classe fortement non circulaire (FNC), que j'ai développés dans ma thèse de docteur-ingénieur [Jou84], sont fondés sur des graphes de dépendance, nommés *sélecteurs d'argument* [CoF82], qui ressemblent aux graphes de dépendance synthétisés, sauf qu'il y en a qu'un seul par non-terminal, au lieu d'un ensemble contenant éventuellement plusieurs graphes, ce qui permet de les calculer en temps polynomial. Ceci conduit à des évaluateurs applicatifs qui sont faciles à construire et raisonnablement efficaces. Cependant, d'une part, la classe FNC est plus petite que la classe NC (bien qu'en pratique la différence soit très faible) et, d'autre part, ces évaluateurs sont encore un peu trop éloignés de l'idéal d'efficacité de l'évaluation que nous voulons atteindre car, en ce qui concerne la quantité de structures de données à maintenir et de calculs à effectuer pour décider que faire à chaque étape, ils en font encore trop. Par exemple, puisque les sélecteurs d'argument ne définissent qu'un ordre *partiel* sur les attributs de chaque non-terminal, ces évaluateurs doivent tester *dynamiquement* si l'attribut qu'ils doivent calculer à un instant donné ne le serait pas déjà ; on constate qu'en pratique le coût de ces tests est loin d'être négligeable.

L'idéal serait de trouver un évaluateur qui, outre les points 1 et 2 ci-dessus, qui sont absolument nécessaires et qui coûtent à peu près le même prix dans tout évaluateur, réduise le point 3 à sa plus simple expression. Or, l'évaluateur doit au minimum être capable d'adapter son comportement à l'arbre qui lui est passé en entrée ; il est donc nécessaire qu'il puisse déterminer sur chaque nœud de l'arbre quelle production s'y applique. Mais le point 3 devrait être réduit à cet unique test par nœud.

Les seuls évaluateurs connus qui réalisent cet idéal sont les évaluateurs à *séquences de visite*. Je ne vais pas refaire ici toute la théorie des évaluateurs à séquences de visite et de la classe des GA l-ordonnées qui lui est associée (voir pour cela [Alb91, EnF82, Fil87, Kas80] et bien d'autres). Je me contenterai de rappeler que, dans la définition initiale de la condition selon laquelle une GA est l-ordonnée [EnF82], Engelfriet et Filé caractérisaient cette condition de deux façons équivalentes, l'une dynamique et l'autre statique. Donc, eux aussi avaient déjà cette idée de dualité entre le comportement dynamique et la vue statique d'un évaluateur. Je rappellerai aussi que cette condition statique met en jeu un graphe de production augmenté (comme on s'en sert déjà dans l'annexe B) par des relations de dépendance sur les attributs de chaque non-terminal, qui doivent d'une part être un ordre *total* plutôt qu'un ordre partiel et, d'autre part, vérifier des conditions de non-circularité et de clôture [1].

1. Notons que, d'une part, si ces conditions sont assouplies, elles définissent une classe de GA plus large mais pour laquelle on ne peut plus construire d'évaluateurs à séquences de visite et que, d'autre part, si elles

Dans ce chapitre nous allons donc partir de ces évaluateurs à séquences de visite et de grammaires l-ordonnées — que nous employons bien entendu dans le système FNC-2, puisque nous recherchons en particulier l'efficacité — et montrer qu'une analyse un peu plus poussée permet d'améliorer encore le compromis déjà tout à fait intéressant obtenu sur cette base.

2.1 Transformation d'une GA FNC en une GA l-ordonnée

Le problème essentiel de la classe l-ordonnée est que la caractérisation d'une GA de cette classe est un problème NP-complet [EnF82] : fondamentalement, il n'y a pas d'autre moyen pour caractériser une GA l-ordonnée que de "deviner" un certain ordre total sur les attributs de chaque non-terminal et de vérifier la condition l-ordonnée avec cette famille d'ordres totaux. Bien entendu, ceci pose un gros problème pratique, et un certain nombre d'auteurs se sont attaqués à le contourner.

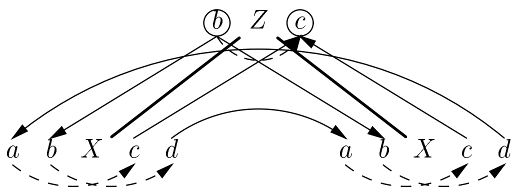
Une de ces techniques est de choisir *a priori* l'algorithme pour totaliser des ordres partiels, que l'on obtient en temps polynomial [Fil87, Kas80], en des ordres totaux, et de vérifier ensuite la condition l-ordonnée avec cette famille d'ordres totaux. C'est la méthode de Kastens avec sa classe des GA ordonnées (OAG) [Kas80]. Le résultat obtenu est intéressant, puisqu'effectivement le test OAG est polynomial, mais malheureusement la classe ainsi obtenue est (sensiblement) plus petite que la classe l-ordonnée. Cette technique ne nous satisfaisait donc *a priori* pas, puisque nous voulions pour FNC-2, outre l'efficacité, la généralité.

L'autre façon de contourner le problème du test l-ordonné est d'essayer de réduire une classe de GA plus grande à la classe l-ordonnée par une transformation adéquate. De fait, la transformation d'une GA FNC en une GA l-ordonnée existe depuis très longtemps ; elle était même déjà définie dans l'article initial d'Engelfriet et Filé [EnF82]. On la retrouve aussi dans [Der84, Fil87, FrJ91, Rii83] [2]. Elle repose sur la construction de *plusieurs* ordres totaux sur les attributs de chaque non-terminal (on les appelle aussi *partitions totalement ordonnées*) et elle est illustrée dans la figure 2.1. Elle atteint effectivement son but, puisque la GA résultante est l-ordonnée et que l'on peut en dériver un évaluateur à séquence de visite qui possède les mêmes bonnes propriétés que l'évaluateur classique pour une GA qui serait directement l-ordonnée. En particulier l'interprétation (au sens de [Fil83]) de la GA qui est construite au cours de cette transformation peut être "cachée" à l'exécution dans le fonctionnement normal de l'évaluateur à séquences de visite, sans aucune perte d'efficacité.

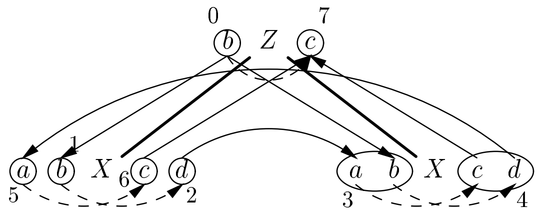
Cette transformation d'une GA FNC en une GA l-ordonnée réalise, pour autant que je le sache, le meilleur compromis entre l'efficacité des évaluateurs produits (à séquences de visite) et la puissance d'expression (classe FNC). Malheureusement, telle quelle, elle demande un temps exponentiel et les évaluateurs produits peuvent être exponentiellement gros. Une partie de nos travaux [JoP91, JPJ90, Par88] a donc consisté à modifier cette construction de façon à réduire le nombre de partitions totalement ordonnées par non-terminal jusqu'à une valeur raisonnable. Cette réduction est fondée sur le même genre d'"approximation" que le *covering* dans le test de circularité [DJL84, JoP90a, LoP75] et consiste à ne garder que les partitions totalement ordonnées qui en "contiennent" d'autres (cf. la figure 2.2). En pratique nous obtenons des "facteurs d'expansion" qui sont extrêmement proche de 1, sans rien perdre

sont renforcées, elles définissent une classe de GA plus petite (par exemple les GA évaluables en k passes) sans que les évaluateurs correspondants soient plus efficaces. Voilà pourquoi je dis que les évaluateurs à séquences de visite sont les meilleurs.

2. Il a aussi été reconnu depuis que la méthode de construction d'évaluateurs de Kennedy et Warren [KeW76], absolument incompréhensible, lui est équivalente.



(a) Graphe de dépendance d'une production avec des sélecteurs d'arguments en partie droite et une partition totalement ordonnée en partie gauche



(b) Tri topologique

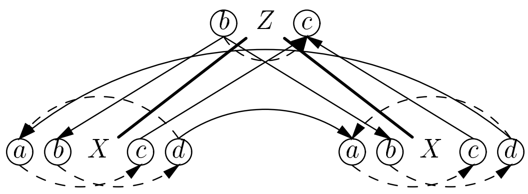
π_1 π_2

(c) Les deux partitions engendrées pour X

BEGIN 1; **EVAL** $\{X_1.b\}$; **VISIT** $1, X_1, \pi_1$;
EVAL $\{X_2.a, X_2.b\}$; **VISIT** $1, X_2, \pi_2$;
EVAL $\{X_1.a\}$; **VISIT** $2, X_1, \pi_1$;
EVAL $\{Z.c\}$; **LEAVE** 1.

(d) La séquence de visite correspondante

FIG. 2.1 –. *La transformation classique d'une GA FNC en une GA l-ordonnée*

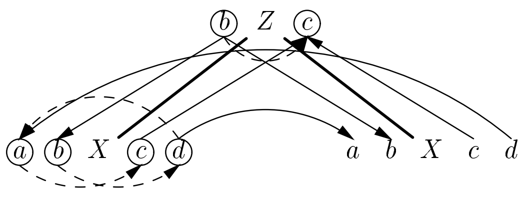


(a) Graphe de dépendance de la production avec π_1 à la fois sur X_1 et X_2

BEGIN 1; **EVAL** $\{X_1.b\}$; **VISIT** $1, X_1, \pi_1$;
EVAL $\{X_2.b\}$; **VISIT** $1, X_2, \pi_1$;
EVAL $\{X_2.a\}$; **VISIT** $2, X_2, \pi_1$;
EVAL $\{X_1.a\}$; **VISIT** $2, X_1, \pi_1$;
EVAL $\{Z.c\}$; **LEAVE** 1.

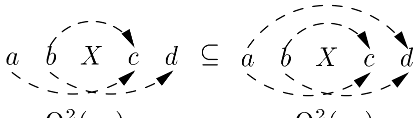
(b) La séquence de visite correspondante

(c) La projection du GDA $D(p)[\pi_0, \pi_1, \emptyset]$



sur X_2 est $a \quad b \quad X \quad c \quad d \quad \subset \pi_1$

(d) Comparaison des contributions :



$\Theta_p^2(\pi_1)$ $\Theta_p^2(\pi_2)$

FIG. 2.2 –. π_1 *peut remplacer* π_2 *dans la production, et je le prouve !*

(ou presque) en efficacité dynamique : la seule différence est qu'un évaluateur ainsi modifié peut avoir à effectuer plus de visites à un certain non-terminal que celui construit par la transformation normale, mais le surcoût pratique de cette différence est totalement négligeable [JPJ90]. Un effet secondaire intéressant est que, le temps d'exécution de la transformation étant directement lié au nombre de partitions totalement ordonnées produites, celle-ci devient maintenant quasi-polynomiale (et en tous cas bien plus rapide que la transformation normale).

Il existe aussi une transformation NC → FNC → l-ordonnée [Fil83], fondée sur une double interprétation, mais nous l'avons écartée car la première phase d'interprétation est ascendante, ce qui ne correspond pas au parcours d'arbre descendant d'un évaluateur à séquences de visite (contrairement à l'interprétation FNC → l-ordonnée). Elle nécessite donc à l'exécution une passe préliminaire, coûteuse en temps et, surtout, qui stocke le résultat de cette interprétation à chaque nœud (sauf si l'on effectue un renommage "physique" comme dans [Kas91], ce qui pose d'autres problèmes dans le cadre de FNC-2). Non seulement ceci est coûteux en place mais, en outre, ceci compromet la faisabilité d'un certain nombre d'améliorations que nous voulons introduire dans FNC-2 : optimisations mémoire (cf. la section suivante), GA dynamiques, unification des domaines syntaxique et sémantiques (cf. le chapitre 3), etc.

2.2 Amélioration de la gestion de la mémoire

Il est bien connu que l'un des principaux problèmes qui empêchent l'utilisation d'évaluateurs d'attributs puissants en milieu industriel est leur très forte consommation de mémoire. En effet, un évaluateur naïf, ou même un évaluateur à séquences de visite non optimisé quant à la gestion de la mémoire, stockera de manière "naturelle" tous les attributs aux nœuds de l'arbre. Or il est connu depuis longtemps que, si le résultat désiré n'est pas l'arbre totalement attribué mais seulement un sous-ensemble de ses attributs, ce qui est souvent le cas (cf. par exemple le chapitre 3), alors la plupart des (autres) instances d'attributs calculées au cours de l'évaluation ont une durée de vie très limitée. Ceci laisse penser qu'on peut essayer de récupérer la place qu'elles occupent dès lors qu'elles ne sont plus utiles.

Ce problème a été abordé de différentes façons. Celle que nous avons choisie dans FNC-2, en particulier parce qu'elle effectue un maximum de calculs statiquement, lors de l'analyse de la grammaire attribuée, et un minimum dynamiquement, lors de l'évaluation proprement dite, est celle qui a été étudiée initialement par Kastens [Kas84, Kas87, KHZ82] et reprise depuis par d'autres auteurs [AkS91, EnJ90, FaY86, Jul89, JuP90]. Cette approche repose, encore une fois, sur l'analyse statique de la durée de vie des différentes instances d'attributs ; il faut noter cependant que cette durée de vie dépend de l'ordre d'évaluation (dynamique) des attributs beaucoup plus que de la GA elle-même. Dans notre cas cette analyse statique est possible, puisqu'avec les séquences de visite, on a un moyen de simuler de façon statique le comportement dynamique de l'évaluateur et, en particulier, de calculer une approximation statique des durées de vie de ces instances d'attributs. Si l'on se limite à un certain type de moyen de stockage, par exemple des variables globales ou des piles globales, on peut alors vérifier que les conditions dans lesquelles il est correct de stocker toutes les instances d'une occurrence d'attribut donnée dans une variable ou une pile sont effectivement respectées par l'évaluateur à séquences de visite donné. Il faut noter ici que, si l'on considère un ensemble de séquences de visite comme un ensemble de procédures mutuellement récursives, ce genre d'analyse est similaire à une analyse de flot de données inter-procédurale [JoM81].

Nous avons consacré une partie de nos efforts à ce problème et nous[3] avons obtenu un certain nombre de résultats [Jul89, JuP90] améliorant ceux qui existaient alors [Kas84], bien que d'autres les aient aussi obtenus indépendamment en même temps que nous [Kas87]. Bien entendu, tous ces résultats ont été intégrés à FNC-2 et lui donnent des performances tout à fait intéressantes (l'article [JPJ90] présente les résultats de quelques mesures ; ces chiffres sont encore valables).

Le premier de ces résultats est que nous avons assoupli les conditions de Kastens sur la gestion d'une pile, en permettant d'accéder sous le sommet de la pile, à condition que la profondeur d'accès soit calculable statiquement. En outre, on peut allonger artificiellement la durée de vie d'une occurrence d'attribut dans une production donnée, ce qui permet en fin de compte de stocker dans une pile n'importe quel attribut *temporaire*, c'est-à-dire tout attribut dont la durée de vie ne "déborde" pas de la visite du nœud qui le porte.

Nous avons aussi amélioré les conditions qui permettent de stocker un attribut temporaire dans une variable globale. En outre, la méthode que nous employons pour ce faire est, à notre avis, plus simple et plus intuitive que celles des travaux antérieurs. Elle est fondée sur la *grammaire des visites*, qui permet de simuler le comportement dynamique de l'évaluateur à séquences de visite : si l'on reprend le parallèle avec l'analyse de flot de données inter-procédurale, cette grammaire est analogue au graphe d'appels.

De même, en utilisant une grammaire plus compliquée, appelée *grammaire des visites et des contextes*, on obtient des informations plus précises sur ce comportement dynamique de l'évaluateur, ce qui permet par exemple de déterminer, comme le font Engelfriet et De Jong [EnJ90] avec un formalisme plus compliqué, quand un attribut non temporaire peut être stocké dans une variable globale.

Nous avons aussi amélioré l'algorithme de Kastens qui permet de grouper en une seule plusieurs variables ou plusieurs piles stockant des valeurs d'attribut, en le dirigeant non pas par des critères de simple faisabilité mais par des critères numériques d'optimalité ; ces critères font intervenir le nombre de règles de copie qu'un groupage permet d'éliminer.

Enfin, nous avons défini des conditions qui permettent de stocker certains attributs non temporaires dans des piles globales. Il semble cependant que cette analyse ne soit pas très rentable en pratique (nous ne l'avons pas encore implantée), en particulier parce qu'elle peut entraîner une passe supplémentaire de l'évaluateur pour dépiler ce genre de piles.

Quant à la dernière catégorie d'attribut, c'est-à-dire les attributs non temporaires qui ne peuvent pas être stockés dans des variables globales, un argument de complexité simple — à un instant donné entre la visite au cours de laquelle ils sont calculés et la visite au cours de laquelle ils sont utilisés pour la dernière fois, il en existe un nombre plus ou moins proportionnel à la *taille* de l'arbre plutôt qu'à sa hauteur — montre que, si l'on veut les stocker hors de l'arbre, il faudra les stocker dans une structure de données dont la taille est liée à celle de l'arbre, ce qui exclut variables et piles. Ceci constitue donc une limite à l'analyse statique des séquences de visite pour la gestion de la mémoire. Plutôt que de perdre du temps à essayer d'atteindre cette limite, nous pensons donc réutiliser la notion de liaisons (*bindings*) définie dans [SwV91] et qui, même si son effet sur la consommation mémoire est négligeable, répond exactement à notre problème : stocker *tous* les attributs hors de l'arbre.

---

3. Rendons à César ce qui est à César : les travaux décrits dans cette section sont essentiellement l'œuvre de Catherine Julié et de Didier Parigot, que je n'ai accompagnés que de très loin. Je me permets cependant de présenter ici leurs résultats car, d'une part, ils constituent un bon exemple d'analyse statique et, d'autre part, les résultats obtenus sont à la base d'un certain nombre de travaux du chapitre 3.

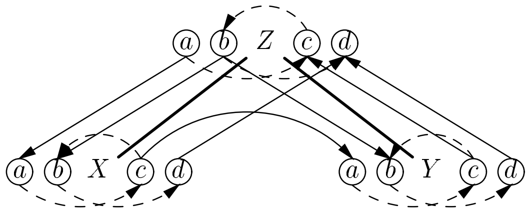
En effet, le fait de stocker tous les attributs hors de l'arbre implique que celui-ci n'a plus besoin d'être un objet *physique*. On peut alors diriger l'évaluateur à séquences de visite par des structures de données ou par des programmes quelconques, à condition qu'ils apparaissent à l'évaluateur comme un arbre. Par exemple, l'évaluation d'attributs sur des graphes acycliques (c'est-à-dire des arbres dont certains sous-arbres sont partagés) devient faisable sans autre forme de procès. Cela nous permet aussi d'augmenter considérablement la puissance d'expression des GA (voir le chapitre 3).

2.3 Évaluateurs parallèles à séquences de visite

La possibilité d'utiliser le parallélisme pour l'évaluation d'attributs est connue depuis le début des travaux sur les grammaires attribuées. Il est en effet évident que, dans un graphe de dépendance complet, il existe beaucoup de chemins de dépendance qui sont disjoints les uns des autres et qui ne se rencontrent qu'en quelques points seulement dans l'ensemble du graphe ; on est alors tenté d'évaluer les instances d'attribut le long de ces chemins en parallèle. C'est tellement évident que la première implantation historique des grammaires attribuées, le système FOLDS [Fan72], utilisait le parallélisme de manière explicite et complète. En gros, pour chaque instance d'attribut dans l'arbre, FOLDS lançait un processus séparé ; ce processus attendait que toutes les valeurs d'instances d'attribut dont il avait besoin, c'est-à-dire celles des prédécesseurs de la dite instance dans le GDC, soient disponibles ; il calculait ensuite la valeur adéquate, puis il signalait à tous les processus qui ont besoin de cette valeur qu'elle est disponible. Ceci peut être implanté facilement en associant un sémaphore à chaque instance d'attribut.

Cependant, il est évident que cette méthode d'évaluation parallèle est au parallélisme ce que la méthode d'évaluation naïve de Knuth est à l'évaluation séquentielle : elle est totalement inefficace, parce que chaque processus n'exécute qu'un tout petit nombre d'instructions utiles et que le surcoût dû à la gestion et à la synchronisation de tous ces processus est alors bien plus important que ce qu'on gagne par leur exécution concurrente. Il est donc utile, de la même façon que cela a été fait pour l'évaluation séquentielle, d'essayer de trouver des méthodes qui sont plus efficaces, en ce sens qu'elles utilisent le parallélisme de manière plus *utile* (j'insiste sur ce dernier mot). En effet, le principal problème de l'approche dynamique de Fang est que la première chose que la plupart des processus doivent faire, quand ils sont lancés, est d'attendre des données calculées par d'autres processus, alors qu'une analyse simple des dépendances dans la GA aurait pu montrer que tel ou tel processus serait bloqué immédiatement après son lancement, sans pouvoir faire quoi que ce soit d'utile, et ne ferait qu'augmenter le coût de la gestion des processus en encombrant la file d'attente. Ce problème est très proche de celui de l'évaluation séquentielle dynamique, dans laquelle il faut déterminer l'ordre d'évaluation à l'exécution, ce qui est équivalent à déterminer l'ordonnancement et la synchronisation des processus dans la méthode de Fang.

Il est donc tout à fait naturel que le problème de l'évaluation parallèle ait été attaqué avec les mêmes techniques d'analyse statique que celui de l'évaluation séquentielle. En effet, la détermination d'un ordre total pour évaluer les instances d'attributs de chaque nœud est le seul moyen d'éviter des opérations coûteuses, comme le verrouillage d'instances d'attributs et la gestion de processus inactifs qui attendent sur des sémaphores.



(a) Le graphe de production augmenté par
des partitions totalement ordonnées sur chaque NT
est acyclique et contient des chemins indépendants

```
BEGIN 1; EVAL –X.a″; VISIT 1,X;
   EVAL –Y.a″; VISIT 1,Y;
   EVAL –Z.c″; LEAVE 1;
BEGIN 2; EVAL –X.b″; VISIT 2,X;
   EVAL –Y.b″; VISIT 2,Y;
   EVAL –Z.d″; LEAVE 2.
```

(b) Une séquence de visite séquentielle

```
BEGIN 1; EVAL –X.a″; VISIT 1,X;
   EVAL –Y.a″; VISIT 1,Y;
   EVAL –Z.c″; LEAVE 1;
BEGIN 2; FORK
   EVAL –X.b″; VISIT 2,X ||
   EVAL –Y.b″; VISIT 2,Y
JOIN; EVAL –Z.d″; LEAVE 2.
```

(c) Une séquence de visite parallèle

Fig. 2.3 –. *Construction de séquences de visite séquentielles et parallèles*

Nous avons donc essayé, comme beaucoup d'autres, d'adapter le paradigme d'évaluation à séquences de visite pour qu'il tire avantage des machines parallèles [4]. L'idée fondamentale de toutes les méthodes d'évaluation parallèle fondées sur les séquences de visite, et en particulier la nôtre [Jou91, Mar90], est de conserver un ordre total pour évaluer les attributs de chaque non-terminal, c'est-à-dire de chaque nœud, mais de permettre d'évaluer les attributs de nœuds différents, ainsi que de visiter ces nœuds différents, en parallèle. La seule différence avec la construction des séquences de visite séquentielles est la suivante : quand on trie topologiquement un graphe de dépendance augmenté, on utilise la séquentialité chaque fois que c'est nécessaire, c'est-à-dire quand les dépendances se réduisent à un chemin unique, mais on utilise le parallélisme quand on rencontre un nœud qui a plus d'un successeur. Ceci est illustré dans la figure 2.3 ; on y voit effectivement que les premières visites de X et Y doivent être exécutées séquentiellement mais que, au cours de la deuxième visite de Z, les visites de X et de Y peuvent être exécutées concurremment.

Cette méthode a l'avantage très important que la synchronisation entre les tâches, qui est un des principaux composants du surcoût induit par le parallélisme, et donc une des principales causes d'inefficacité, est réduite au minimum. En effet, grâce à l'ordre total d'évaluation des attributs de chaque nœud, cette construction garantit que :

1. toutes les données nécessaires à un processus sont calculées avant que le processus ne soit lancé et

2. deux processus différents, qui s'exécutent concurremment, n'accèdent jamais à la même partie de l'arbre et, en particulier, n'écrivent jamais dans les mêmes cellules mémoires.

En d'autres termes, tout le surcoût est concentré dans les opérations `FORK` et `JOIN`, qui sont inévitables si l'on veut effectivement mettre en œuvre le parallélisme. En conséquence, chaque

4. J'ai eu l'occasion récemment de faire une étude de toutes les méthodes concurrentes d'évaluation qui ont été proposées dans la littérature [Jou91], et celle-ci m'a permis de me convaincre que l'approche que nous avions suivie dès le début était effectivement la meilleure.

processus s'exécutera aussi vite que la partie correspondante de l'évaluateur séquentiel. Notons ici que la classe des GA l-ordonnées, avec l'ordre total d'évaluation qui lui est associé, est la plus grande classe de GA pour laquelle il est possible d'utiliser le parallélisme synchrone (`FORK` et `JOIN`). Bien entendu, ce résultat s'étend à la classe FNC grâce à la transformation de la section 2.1.

Dans [Jou91] nous présentons une des implantations possibles de cette méthode, fondée sur un interprète des séquences de visite parallèles, avec force détails sur le fonctionnement fin de l'évaluateur. Dans ce même article, nous donnons aussi les résultats pratiques que nous avons pu mesurer sur une implantation réelle, légèrement différente et *a priori* légèrement moins performante [Mar90]. Ces résultats sont déjà tout à fait intéressants : pour deux grammaires attribuées réalistes, qui n'ont pas du tout été écrites pour tirer parti de l'évaluation parallèle et qui sont typiques d'applications de compilation — la première décrit la vérification de contraintes contextuelles et la seconde décrit la construction d'une forme intermédiaire —, nous obtenons une accélération de l'ordre d'un facteur cinq avec six processeurs sur une machine Sequent (un peu plus pour la première GA, qui est plus "calculatoire", et un peu moins pour la seconde, qui est limitée par les performances de l'allocateur de mémoire partagée). Pour juger de ces chiffres, il faut garder en tête que les différentes tâches concurrentes sont extrêmement petites par rapport à celles d'une application parallèle "normale" comme un serveur de base de données.

D'autres travaux, et en particulier d'autres expérimentations, sont nécessaires pour confirmer ces bonnes performances, mais nous pensons que seule cette méthode est capable d'utiliser correctement le parallélisme que presque toutes les grammaires attribuées exposent. En fait, on peut même dire qu'une GA "normale" expose *trop* de parallélisme dès que le texte ou l'arbre d'entrée atteint une taille "normale", ce qui fait que, même en se limitant au parallélisme utile, le surcoût dû à la gestion de ce parallélisme est encore trop important. Nous pensons attaquer ce problème en affinant la notion de parallélisme utile, par exemple en restreignant les visites parallèles à des non-terminaux récursifs qui conduisent potentiellement à de gros sous-arbres ; au contraire de la plupart des auteurs (cf. [Jou91]), nous aimerions pouvoir déterminer un certain optimum sans que l'utilisateur ne donne aucune information supplémentaire (encore l'analyse statique !).

2.4 Méta-composition et optimisation des grammaires couplées par attributs

Ganzinger et Giegerich ont introduit en 1984 [GaG84] une variation du formalisme des GA qu'ils ont appelée *grammaires couplées par attributs*. Cette variation consiste à voir toute GA comme une *fonction* qui prend en entrée un arbre attribué, décrit par une certaine syntaxe attribuée, et qui produit en sortie un autre arbre attribué, décrit par une autre syntaxe attribuée. Ceci se fait assez simplement en considérant une syntaxe attribuée comme un ensemble de définitions de types, correspondant aux non-terminaux, et de constructeurs de ces types, correspondant aux productions. On déclare alors dans une telle GCA des attributs de type "non-terminal X" et des règles sémantiques qui utilisent les productions dont X est partie gauche pour construire des sous-arbres de l'arbre de sortie dont la racine est étiquetée par X et qui sont les valeurs de ces attributs de type X. L'arbre de sortie complet est obtenu à la racine : c'est la valeur sémantique de la GCA, au sens de l'annexe B.

Nous verrons dans la section 3.1 comment cette notion de GCA améliore la facilité d'utilisation des GA en général et du système FNC-2 en particulier. Cependant, la raison pour laquelle je présente ici cette notion de GCA est qu'elle donne lieu à un des exemples les plus fascinants de la puissance de l'analyse statique des GA : je veux parler de la *méta-composition* de ces grammaires couplées par attributs, que Ganzinger et Giegerich avaient baptisée de l'affreux néologisme *composition descriptionnelle*. L'idée est la suivante : si l'on voit une GA comme décrivant une fonction prenant un certain arbre attribué et rendant un autre arbre attribué, il est tout à fait naturel de chercher à composer ces fonctions. Cette composition peut être réalisée très simplement de manière dynamique, en exécutant séquentiellement les évaluateurs produits pour chacune de ces deux GA, le résultat du premier étant la donnée du second. Or, Ganzinger et Giegerich ont montré que cette composition pouvait aussi se réaliser *statiquement*, c'est-à-dire qu'à partir de la seule donnée de deux GCA composables, on peut construire mécaniquement une unique GA qui réalise la même fonction (traduction) que la composition des deux GA données.

Ceci est déjà en soi un résultat important, puisqu'il permet de profiter de la modularité introduite par cette notion de GCA (cf. la section 3.1) sans perte d'efficacité. Mais la méta-composition va encore plus loin puisque, non seulement la GA résultante est construite statiquement, mais en outre et surtout, elle réalise cette même traduction de façon *plus efficace*[5], parce que l'arbre intermédiaire, c'est-à-dire le résultat du premier évaluateur et la donnée du second, n'est en fait *pas construit*. L'idée fondamentale de la méta-composition consiste en effet à plaquer les règles sémantiques d'une certaine production p de la deuxième GA sur les productions de la première GA dans lesquelles p était utilisée pour construire une partie de l'arbre intermédiaire.

Ce résultat est édifiant, car cette construction permet d'améliorer à la fois le confort du programmeur et l'efficacité du programme résultant, ce qui est exceptionnel dans le monde de l'analyse statique.

Bien entendu, nous avons implanté cette méta-composition dans un des processeurs de FNC-2 ; nous en avons profité pour l'étendre sur un certain nombre de points, en particulier les constructions d'arbre contrôlées par des conditionnelles.

Cependant, le résultat brut de la méta-composition n'est pas entièrement satisfaisant. Par exemple, si l'on écrit une GA qui prend en entrée un arbre qui est une liste, et qui construit l'inverse de cette liste (ceci se réalise très facilement en une passe haut-bas gauche-droite), et qu'on compose cette GA d'inversion avec elle-même, on obtient effectivement une GA qui recopie l'arbre d'entrée à l'identique. Malheureusement cette GA de recopie est moins efficace qu'une GA "naturelle" qui réalise le même calcul, car elle fait, avant la même passe de construction que celle de la GA naturelle, une passe entière sur toute la liste qui est totalement inutile, puisqu'elle se contente de propager la valeur *liste vide*.

Nous avons donc défini certaines optimisations "source à source" de GA, c'est-à-dire qui transforment une GA en une autre GA bâtie sur la même syntaxe, et qui permettent d'éliminer beaucoup de chaînes de règles de copie inutiles : d'une part, des propagations de valeurs constantes (cf. l'exemple ci-dessus) et, d'autre part, des chaînes de règles de copie en forme de *canne*, c'est-à-dire reliant deux attributs du même non-terminal, soit vers le bas soit vers le haut [Rou92]. Là encore, cette analyse est faite totalement statiquement, en utilisant

5. On peut définir qu'une GA G_1 est plus efficace qu'une GA G_2 si et seulement si G_1 et G_2 décrivent la même fonction et, quelle que soit la méthode d'évaluation d'attributs M, l'évaluateur construit selon M pour G_1 est plus efficace que celui construit selon M pour G_2.

des algorithmes semblables au test de non-circularité ou au test FNC. Il est intéressant de remarquer ici que, alors que la plupart des tests de caractérisation de GA (NC, FNC, etc.) recherchent l'existence d'*une* dépendance dans *au moins un* sous-arbre, l'analyse permettant d'éliminer des chaînes de règles de copie recherche l'existence de *la même* dépendance dans *tous* les sous-arbres issus d'un même non-terminal [Rou92].

Les résultats que nous obtenons sont tout à fait intéressants : la GA résultante est à la fois beaucoup plus facile à lire et plus efficace que la GA non optimisée, car beaucoup de règles sémantiques inutiles sont éliminées.

2.5 Améliorations des algorithmes d'analyse statique de GA

Dans tout ce chapitre, j'ai présenté un certain nombre d'exemples d'analyse statique de grammaires attribuées, destinés à améliorer l'efficacité des évaluateurs d'attributs correspondants. Dans cette section, je voudrais passer à un niveau un peu "méta", en présentant des améliorations qui s'appliquent aux algorithmes d'analyse eux-mêmes.

Tous les algorithmes d'optimisation que nous avons vus dans ce chapitre, et la quasi-totalité des autres algorithmes d'analyse de GA qui ont été présentés dans la littérature, comme le test de circularité de l'annexe B, rentrent dans un cadre commun, que ses inventeurs, Möncke et Wilhelm [MöW91], ont appelé, par analogie avec l'analyse de flot de données, *analyse de flot de grammaire* (AFG). En effet, ils partagent tous la caractéristique de fonctionner sur le graphe de la grammaire, dont on trouvera la définition précise dans [JoP90a, MöW91], ainsi que de calculer et de propager des informations, attachées aux non-terminaux, *exclusivement* le long des arcs de ce graphe. On distingue des problèmes d'AFG ascendants et des problèmes descendants. Un problème ascendant (resp. descendant) est défini par la donnée de fonctions de propagation, qui définissent comment, sur une production donnée, les informations disponibles sur les non-terminaux en partie droite servent à calculer l'information sur le non-terminal en partie gauche (resp. de la partie gauche vers la partie droite), et des fonctions de combinaison, qui définissent comment les informations produites pour un certain non-terminal sur toutes les productions où il apparaît en partie gauche (resp. en partie droite) sont associées pour former l'(unique) élément d'information à attacher à ce non-terminal. La solution du problème est alors le point fixe de ces équations de propagation et de combinaison, qui sont mutuellement récursives. On retrouve ici un parallèle tout à fait direct avec l'analyse de flot de données, dans laquelle il y a aussi un graphe (le graphe de flot de contrôle), de l'information qui se propage à travers les nœuds de ce graphe (les blocs de base) et qui est combinée aux extrémités des arcs de ce graphe, et un calcul de point fixe.

L'intérêt de ce cadre qu'est l'analyse de flot de grammaire est, comme dans l'analyse de flot de données, que les améliorations que l'on peut apporter à la résolution d'un des problèmes qui tombent dans cette classe s'appliquent aussi, en général, à tous les autres problèmes. Or, ceux-ci sont très nombreux [JoP90a, MöW91] et dépassent largement le cadre des GA : le calcul des ensembles $First_k$ et $Follow_k$ sur une grammaire, ainsi que le pré-calcul des ensembles de modèles d'arbre pouvant être reconnus sur un non-terminal donné dans un *pattern-matcher* d'arbre ascendant [HoO82], sont des problèmes d'AFG.

Möncke et Wilhelm ont étudié les propriétés théoriques de l'analyse de flot de grammaire, mais n'ont abordé que très superficiellement l'implantation effective des algorithmes de résolution. A l'inverse, nous avons beaucoup étudié un de ces algorithmes particuliers, le test de non-circularité simple présenté dans l'annexe B, et nous avons défini plusieurs techniques d'implantation qui l'améliorent considérablement par rapport à l'algorithme naïf de la figure B.8. Parmi celles-ci, il y en a quatre qui s'appliquent globalement à tout problème d'analyse de flot de grammaire et qui améliorent considérablement la vitesse de la résolution [JoP90a]. Ces techniques utilisent la structure du graphe pour converger plus vite vers le point fixe, de même qu'on le fait depuis longtemps dans l'analyse de flot de données [JoM81]. Ce sont :

- le partitionnement de la grammaire, selon lequel la grammaire est décomposée en sous-grammaires selon la relation "dérive de" et les sous-grammaires sont traitées dans un ordre dérivé de la relation quotient ;
- l'ordonnancement statique des non-terminaux, qui utilise cette même relation "dérive de" pour déterminer un ordre quasi-optimal pour traiter les non-terminaux ;
- la stabilité faible, qui permet d'éviter les traitements redondants de non-terminaux et de productions dont on peut assurer qu'ils ne produisent que des arbres terminaux ;
- et enfin la stabilité sémantique, qui prend en compte l'âge de chaque élément d'information pour éviter des calculs redondants.

On peut considérer, d'un certain point de vue, que les deux premières techniques constituent aussi une certaine forme d'analyse statique de la grammaire, ou du moins de pré-traitement de celle-ci.

Je ne rentrerai pas plus dans les détails ici (cf. [JoP90a] et les publications qu'il référence), mais je voudrais insister sur le fait que ces algorithmes sont tout à fait efficaces, puisque sur des GA normales, le temps nécessité par le test de non-circularité simple est divisé en général par des facteurs supérieurs à 10. Ceci est particulièrement intéressant pour nous puisque, dans le système FNC-2, nous utilisons de façon très intensive des algorithmes qui rentrent dans le cadre de l'analyse de flot de grammaire : test FNC (ascendant), test DNC (descendant), transformation FNC → l-ordonnée (descendant), élimination des chaînes de règles de copie (ascendant et descendant), etc. Sans ces améliorations de l'AFG, notre système ne serait guère utilisable dans un contexte industriel.

2.6 Conclusion

Tout au long de ce chapitre, nous avons présenté de nombreux exemples d'analyse statique de grammaires attribuées visant à augmenter l'efficacité, en temps et en place, des évaluateurs produits à partir d'une GA donnée.

Ma conviction aujourd'hui, que je tire de notre expérience avec le système FNC-2, est qu'il n'y a plus grand-chose à faire sur ce plan ; on peut même dire qu'il n'y a plus aucune raison de faire quoi que ce soit sur ce plan, puisque, si l'on ne tient pas compte de la traduction des règles sémantiques elles-mêmes, qui n'est pas encore assez puissante pour donner des résultats efficaces (cf. le chapitre 3), les évaluateurs produits par FNC-2, c'est-à-dire les séquences de visite et leur implantation proprement dite sur une machine donnée, sont aussi efficaces[6] que des programmes écrits à la main qui seraient fondés sur le même paradigme de calcul,

6. Voire même plus, car ils ne font que ce qui est strictement minimum pour respecter les dépendances, alors que l'humain qui écrit son programme n'a pas forcément une idée claire de celles-ci et peut donc "en faire trop".

c'est-à-dire qui travailleraient sur des arbres ayant une existence physique. En ce qui concerne donc la construction et l'implantation d'évaluateurs d'attributs *exhaustifs* et *séquentiels*, j'ose dire que tous les problèmes sont résolus ; en revanche, la réévaluation incrémentale [Rep84] et l'évaluation sur machines parallèles [Jou91] requièrent encore quelque effort.

Je suis convaincu que la simplicité du formalisme des GA est le fondement de cette conclusion un peu brutale, et cette simplicité me satisfait pleinement. En revanche, il reste encore beaucoup à faire sur les extensions de ce formalisme, c'est-à-dire les possibilités d'y "faire entrer" certaines applications qui jusqu'ici en étaient exclues. Ce sujet, entre autres, est abordé dans le prochain chapitre.

Chapitre 3

Analyse statique des GA pour améliorer leur facilité d'utilisation et leur puissance d'expression

Dans le chapitre précédent, nous avons vu un certain nombre d'exemples d'analyse statique de grammaires attribuées destinés à améliorer l'efficacité des évaluateurs produits automatiquement à partir de ces GA. Nous nous intéressons maintenant au second problème mentionné dans l'introduction et qui freine l'utilisation des GA en milieu industriel ; je veux parler des difficultés à programmer à plusieurs sur une GA de taille importante pour réaliser une unique application. D'autre part, une autre raison de ce manque d'utilisation est que, telles quelles, les GA ont un pouvoir d'expression relativement faible ; ceci est dû essentiellement à ce que, d'une part, la valeur des attributs n'a aucune influence sur l'ordre dans lequel ils sont calculés et, d'autre part, l'évaluateur est strictement guidé par l'arbre d'entrée, qui doit préexister au calcul ; ces "limitations" n'existent pas dans la plupart des autres paradigmes de programmation.

Je voudrais donc expliquer ici comment nous avons abordé un certain nombre de ces problèmes dans le système FNC-2. Les aspects qui me semblent les plus importants pour faire de FNC-2 un système réellement utilisable en pratique peuvent se classer en trois catégories :

1. la vue globale de ce qu'est une application, en particulier la notion de modularité avec tous ses avantages ;
2. le langage de description de GA et les constructions qu'il offre au programmeur, dont les "micro-extensions" par rapport aux GA classiques ;
3. enfin, les extensions "lourdes", qui dépassent le simple cadre du langage de programmation et qui visent des aspects plus fondamentaux de l'utilisation d'un système de traitement de GA et des évaluateurs produits.

Je vais présenter les résultats que nous avons obtenus sur ces trois points dans les sections qui vont suivre. Certains de ces résultats ne sont pas réellement des applications de l'analyse statique, mais je les citerai tout de même pour placer les véritables analyses statiques dans un cadre plus complet.

3.1 Ce qu'est une application pour FNC-2

Comme dit dans l'introduction, le point le plus gênant pour faire accepter les GA dans le monde industriel est qu'une application que l'on voudrait réaliser à l'aide de ce paradigme doit théoriquement être programmée en *une seule* GA ; le code de cette application se retrouve donc intégralement rassemblé dans un unique fichier, monolithique et relativement mal structuré[1]. Ce n'est évidemment pas tenable en pratique, d'autant plus que les applications naturelles des GA en compilation deviennent de plus en plus volumineuses, soit parce que le langage source devient plus complexe, soit parce que produire le code le plus efficace, en particulier pour les nouvelles architectures, demande des analyses et des optimisations toujours plus fines. Il est donc absolument nécessaire de fournir au programmeur de GA des outils de modularité qui, d'une part, facilitent son travail personnel en découpant la spécification d'une tâche en entités plus abordables, c'est-à-dire plus petites, plus agréables à lire et plus faciles à développer, et, d'autre part, permettent à plusieurs programmeurs de travailler séparément et concurremment sur chacune de ces entités.

Le concept de modularité que nous avons utilisé dans FNC-2 pour résoudre ce problème était déjà connu avant que nous ne débutions nos travaux, puisque c'est celui de grammaires couplées par attributs, déjà évoqué dans la section 2.4. Je rappelle que cette notion consiste à considérer une GA comme une fonction, prenant en entrée un arbre attribué et construisant en sortie un autre arbre attribué. Ainsi, on peut découper une application en une suite de phases, chaque phase étant décrite par une GA.

Cependant, ce formalisme a d'autres avantages que la modularité "brute". La figure 3.1 illustre la structure d'une application typique construite avec FNC-2. Cette application, dont le modèle d'exécution est en partie droite de la figure, peut s'apparenter à un compilateur classique, avec une phase d'analyse lexico-syntaxique et de construction d'arbre, une phase de vérification des contraintes contextuelles et d'analyse sémantique, une phase de traduction vers un certain langage objet et enfin une production de "code" sous forme textuelle. Dans ce cas précis, l'application est construite uniquement avec des outils proposés par FNC-2 ; en particulier, les deux phases centrales sont toutes deux décrites par des GA.

Cependant, rien n'empêche, dans ce modèle d'application, et si l'on connaît exactement le type des données qui transitent d'une phase à une autre, de programmer l'une de ces phases, soit à la main, soit avec un outil plus adapté que les GA au travail effectué dans cette phase. Par exemple, une phase d'optimisation dans un compilateur *modifie* l'arbre qu'il a en entrée de façon relativement locale, mais en fonction d'un contexte global. Si l'on veut programmer un tel optimiseur avec une GA, on est obligé de construire complètement l'arbre de sortie, sans pouvoir réutiliser quoi que ce soit de l'arbre d'entrée, alors que, justement, l'arbre de sortie est quasiment identique à celui d'entrée et que les modifications ne sont effectuées que dans quelques règles sémantiques. Ceci est évidemment très fastidieux. Il existe un formalisme, les systèmes de transformation d'arbres attribués, et des outils, comme OPTRAN [LMW88] ou TRANSAT [Sou91], qui sont bien mieux adaptés à la spécification et à l'implantation ce genre d'application que ne peuvent l'être les GA.

1. *A priori*, la structure d'une GA est strictement calquée sur la syntaxe sous-jacente ; toutes les règles sémantiques qui traitent d'une même notion sémantique (dans un compilateur, l'identification, la vérification de types, la production de code, ...) sont donc "éclatées" sur toute la syntaxe, alors qu'on voudrait souvent les voir rassemblées en un même "chapitre", en particulier si la GA est de taille importante. C'est exactement ce à quoi répond la construction **phase** d'OLGA.

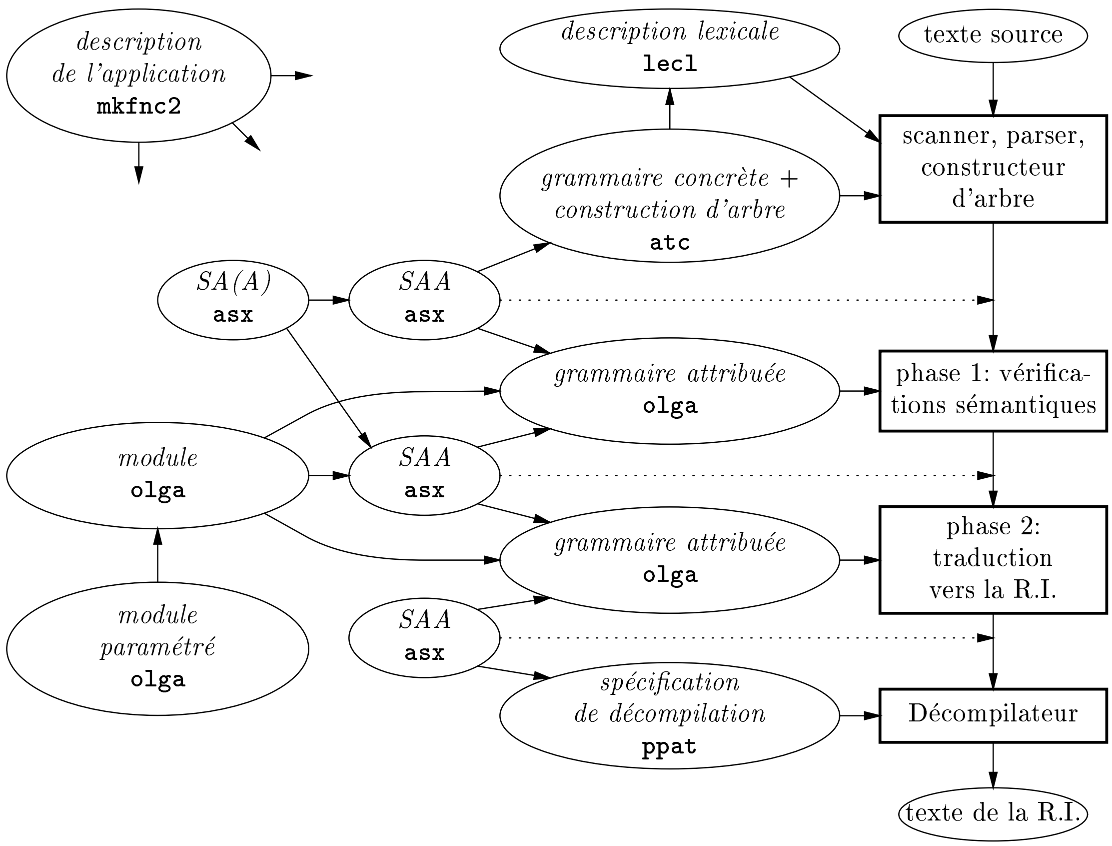


FIG. 3.1 –. *Une application de* FNC-2 *typique*

De même, dans cette application de compilation, je n'ai pas inclus de générateur de code : la dernière boîte n'est qu'un simple décompilateur de la forme intermédiaire, dont le résultat est du texte lisible par un humain, mais qui n'est pas à proprement parler du code. Là encore, rien n'empêche de remplacer cette boîte par un générateur de code complet, avec sélection des instructions, allocation des registres, etc., tel qu'en produit le système PAGODE, développé dans le projet CHLOÉ de l'INRIA [DMR90].

Outre la modularité, on gagne donc, avec ces grammaires couplées par attributs, la souplesse, puisqu'on peut associer, à des modules traités et construits par FNC-2, d'autres modules construits par d'autres outils (ou à la main). La condition pour que ce soit faisable est, comme dit ci-dessus, que les données transitant d'une phase à une autre soient bien identifiées ; nous verrons plus loin que c'est le cas avec FNC-2, puisque ces données sont des *arbres abstraits attribués*, qui sont définis précisément avec un langage adapté, `asx` (cf. la figure 3.1).

Pour conclure sur les grammaires couplées par attributs et les facilités de programmation qu'elles apportent, je voudrais faire remarquer que la méta-composition qui leur est associée, et qui est une forme d'analyse statique, permet de profiter de cette modularité sans aucune perte d'efficacité, bien au contraire, comme je l'ai montré dans la section 2.4. Cependant, cette possibilité de combiner deux évaluateurs en un seul ne met pas en cause un des avantages importants de la modularité, la compilation séparée. En effet, dans une première étape de développement et de mise au point, les programmeurs développent leurs GA séparément, la composition des évaluateurs correspondants se faisant à l'exécution, dynamiquement ; le

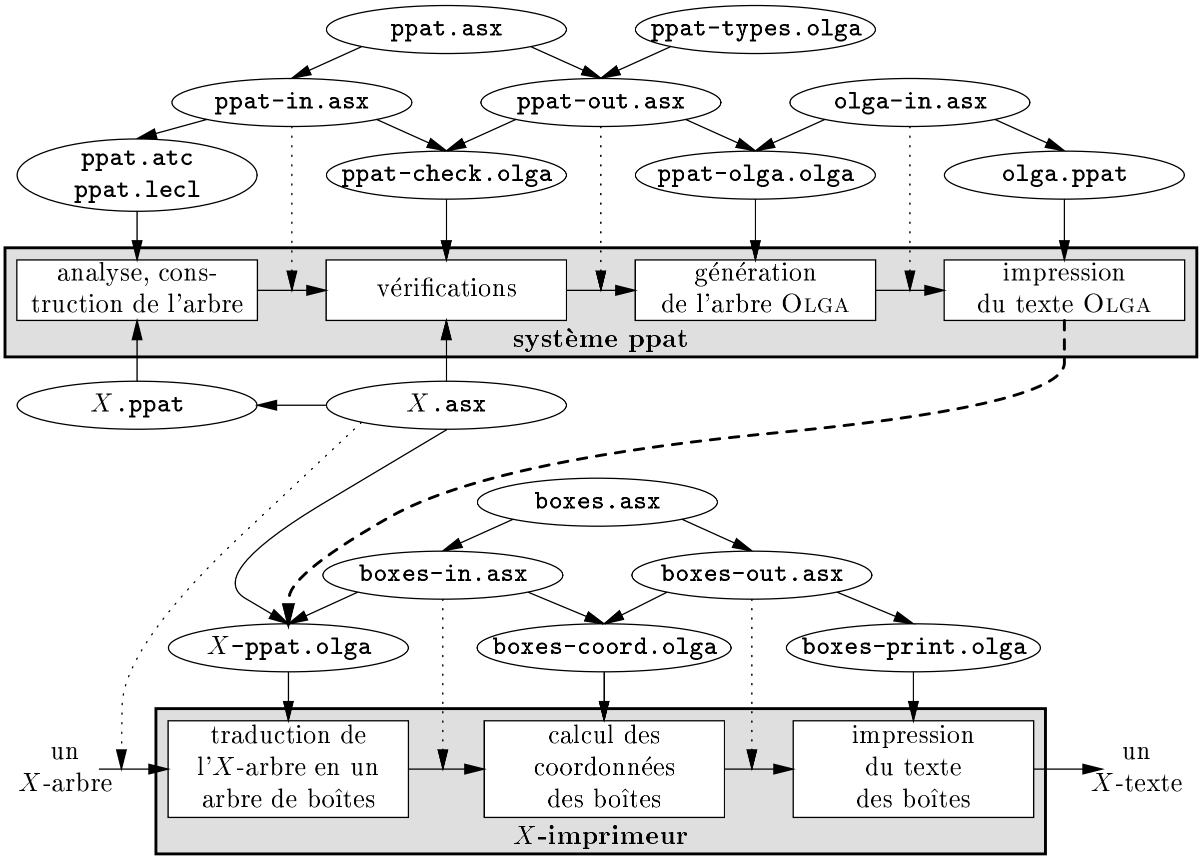


FIG. 3.2 –. *Le générateur de décompilateurs* PPAT

temps de construction de l'application est alors réduit. Dans une deuxième phase de production, on invoque la méta-composition pour gagner en efficacité, mais ceci se fait de manière transparente, avec les mêmes fichiers source (par exemple à l'aide d'un *makefile*).

Je voudrais terminer cette section en illustrant, sur un exemple, quelques autres avantages de la modularité. Dans la figure 3.2, j'expose la structure de notre générateur de décompilateurs d'arbres attribués, PPAT, et du décompilateur qu'il produit pour une syntaxe abstraite attribuée particulière X, définie dans X`.asx`, selon la spécification de décompilation X`.ppat`. Comme on peut le remarquer, le nombre de modules mis en jeu est important, pour une application qui est relativement simple, ce qui permet de réduire la taille et la complexité de chacun de ces modules. Dans l'état actuel des choses, PPAT et la partie fixe des décompilateurs qu'il produit sont répartis sur une quinzaine de fichiers totalisant environ 8000 lignes ; on voit donc que la taille moyenne d'un de ces fichiers est d'environ 500 lignes, ce qui reste largement à la portée de tout programmeur.

Le deuxième avantage de la modularité illustré par la figure 3.2 est la notion de *réutilisation*. La troisième phase du système PPAT, dénotée “génération de l'arbre OLGA” dans la figure 3.2, construit la première phase du décompilateur sous la forme d'une GA écrite dans notre langage OLGA (fichier X`-ppat.olga`). La syntaxe attribuée des arbres qui sont construits par cette phase est donc la même que celle des arbres produits par l'analyseur syntaxique et le constructeur d'arbre du langage OLGA lui-même (fichier `olga-in.asx`). Ce fichier est partagé entre le composant de FNC-2 qui analyse des GA écrites en OLGA, et

PPAT qui en fabrique. Un autre exemple de réutilisation se situe au niveau des décompilateurs d'arbres eux-mêmes. En effet, le modèle de décompilation sur lequel est fondé PPAT est celui de boîtes de texte imbriquées les unes dans les autres, à la manière de TEX. Tout décompilateur est donc organisé en trois phases très simples. La première, qui correspond à la GA engendrée par PPAT, consiste simplement à traduire un certain arbre de type X en un arbre de boîtes, selon la spécification donnée en entrée à PPAT. Tout le reste du décompilateur, c'est-à-dire d'abord le calcul des coordonnées des boîtes, puis l'impression du texte contenu dans les feuilles de cet arbre de boîtes aux coordonnées ainsi calculées, est réutilisable (et effectivement réutilisé) d'un décompilateur à un autre. L'efficacité du décompilateur est garantie par la méta-composition. Enfin, dans cette application particulière, le langage PPAT étant très voisin d'OLGA, notre langage de description de GA, la deuxième phase du système PPAT, qui vérifie les contraintes contextuelles sur une description de décompilation, emprunte beaucoup de fonctions et de schémas d'attributs à son homologue dans le compilateur OLGA.

Pour terminer sur cet exemple, je voudrais juste faire remarquer que nous utilisons autant que possible la notion d'auto-amorçage (*bootstrap*). En effet, la quatrième et dernière phase du système PPAT, qui produit le texte proprement dit de la GA X`-ppat.olga`, est un décompilateur engendré par PPAT lui-même (fichier `olga.ppat`).

Notre propre expérience montre que la modularité et la réutilisabilité offertes par les GCA, qu'on peut qualifier d'"extérieures" à une GA proprement dite, sont très efficaces en termes de productivité du programmeur. A titre d'exemple, le système PPAT, composé, comme je l'ai dit plus haut, de 8000 lignes environ d'OLGA et autres langages associés, a été réalisé et testé en environ 3 mois par un étudiant qui ne connaissait initialement que superficiellement les grammaires attribuées et pas du tout FNC-2.

3.2 Le langage de description de grammaires attribuées OLGA

3.2.1 Généralités

Dans cette section, je voudrais passer à un niveau un petit peu plus "microscopique", du point de vue du programmeur, puisque je vais parler du langage OLGA, que nous avons conçu pour la description de GA proprement dites (par opposition à l'association de plusieurs GA et autres modules pour construire une application complète, comme décrit dans la section précédente). Je ne voudrais pas détailler ici tous les traits d'OLGA (on en trouvera une bonne description résumée dans [JLP90] et la description complète dans [JoP92]), mais seulement insister sur quelques points qui, à mon avis, rendent OLGA particulièrement agréable aux programmeurs parce que le système qui l'implante, FNC-2, utilise l'analyse statique pour simplifier leur tâche.

Le premier aspect, un peu trivial mais qui a son importance pour le programmeur (ainsi que pour nous, en tant que concepteurs du système, car cela évite bien des problèmes [JLP90]), est qu'OLGA est un langage *réellement applicatif* ; il n'y a pas de variables globales et très peu d'effets de bord, qui sont en outre parfaitement contrôlés. Bien entendu, dans un langage applicatif, il n'y a pas d'itération mais la récursion ; comme il se doit, le compilateur OLGA détecte les cas de récursion terminale et les remplace par des boucles. Un deuxième aspect, presqu'aussi trivial que le précédent, est le fait qu'OLGA est un langage fortement typé, ce qui est évidemment crucial pour le programmeur ; la vérification des contraintes contextuelles constitue, elle aussi, un exemple d'analyse statique très connu, sur lequel je n'insisterai donc pas.

$$
\begin{tikzcd}
S \arrow[r, "réduction"] \arrow[d, "?"'] & G \arrow[d, "construction"] \\
I & E \arrow[l, "interprétation"]
\end{tikzcd}
$$

FIG. 3.3 –. *La notion de changement de plan*

Un autre aspect d'OLGA, qui permet de gagner beaucoup en puissance d'expression, au moins descriptive, est l'abolition de la séparation qui est faite, dans le formalisme classique des GA, entre le domaine syntaxique, c'est-à-dire l'arbre d'entrée, support des règles sémantiques et des attributs, et le domaine sémantique, c'est-à-dire les attributs eux-mêmes et leurs valeurs, dont l'arbre de sortie ; il faut bien dire que cette séparation stricte est assez artificielle. Loin de moi l'idée de vouloir faire croire au lecteur que j'ai inventé ce concept : il est connu depuis au moins les travaux de Reps sur le Synthesizer Generator [Rep84]. Cependant il apporte beaucoup en facilité d'utilisation, tout en restant simple. On permet donc, d'une part, qu'un attribut ait pour valeur un (sous-)arbre, comme dans les grammaires couplées par attribut. D'autre part, on permet d'utiliser des sous-arbres de l'arbre d'entrée comme des valeurs, en particulier des valeurs d'attributs. Enfin — et c'est le plus important, en ce qui concerne le pouvoir descriptif —, on permet d'appliquer à de telles valeurs de type arbre des fonctions qui sont décrites par des GA ; en effet, la vue d'une GA comme une fonction, introduite avec les GCA, s'étend naturellement à des GA qui, d'une part, rendent autre chose que des arbres et qui, d'autre part, prennent plus d'un argument, le premier étant l'arbre d'entrée et les autres des attributs hérités de la racine.

Les deux sous-sections qui suivent présentent d'autres aspects d'OLGA dans lesquels l'analyse statique joue un rôle important, aspects que je juge plus intéressants car plus originaux.

3.2.2 Nœuds listes et changement de plan

Le premier de ces aspects que je voudrais aborder peut sembler trivial mais illustre bien un concept très important, que je nomme *changement de plan*.

Dans FNC-2, les objets qui transitent d'une phase à une autre, en particulier d'une GA à une autre, sont, comme je l'ai dit, des arbres abstraits attribués. Le formalisme que nous utilisons pour les décrire, `asx`, est très proche de METAL, celui qui avait été introduit pour MENTOR et CENTAUR [KLM83]. La différence essentielle de METAL et donc d'`asx` par rapport à la BNF normale, outre l'attachement éventuel d'attributs aux non-terminaux d'une grammaire, est la possibilité d'avoir des nœuds listes, c'est-à-dire des nœuds qui ont un nombre variable de fils. Notre expérience montre que cette facilité est très agréable à utiliser et fait gagner en productivité de façon importante. Mais le problème que nous avions à résoudre, après avoir fait ce choix, est que les algorithmes de construction d'évaluateurs, que j'ai présentés dans le chapitre 2, sont tous fondés sur le formalisme classique des GA et en particulier sur des syntaxes décrites en BNF, sans listes. Il nous fallait donc trouver un moyen d'associer la notion de liste à une notion "équivalente" dans le formalisme classique des GA.

Ce problème est un exemple d'application de la notion de *changement de plan*, illustrée dans la figure 3.3. On a une spécification S d'un certain calcul, exprimée dans un formalisme voisin de celui des GA sans lui être totalement identique. On veut en déduire le plus automatiquement possible une certaine implantation I de ce calcul, connaissant la construction qui, à partir d'une GA "normale" G, produit l'évaluateur E qui l'implante. Le changement de plan consiste alors à définir, d'une part, la fonction de *réduction* d'une spécification S vers une GA G "équivalente" et, d'autre part, la fonction d'*interprétation* qui permet d'obtenir l'implantation I de S à partir de l'évaluateur E de G, de sorte, bien sûr, que le diagramme commute ! Bien entendu, les fonctions de réduction et d'interprétation sont supposées simples à définir et à appliquer, ce qui permet d'étendre simplement à d'autres formalismes tous les travaux effectués sur les GA.

En ce qui concerne les nœuds listes, la difficulté de la fonction de réduction est d'avoir une description finie, en BNF et en GA classique, d'un ensemble potentiellement infini de règles sémantiques, tel qu'il peut *a priori* en exister sur un nœud liste. Nous avons donc décidé de restreindre les schémas de dépendance que l'on peut définir sur un nœud liste, en imposant que les attributs d'un fils quelconque "au milieu" de la liste, c'est-à-dire autre que le premier et le dernier fils, ne puissent être définis qu'en fonction de ceux du père et des frères de gauche et de droite de ce fils. Il s'avère que cette contrainte est raisonnable en pratique. La réduction consiste alors à construire, à partir d'une spécification de dépendances sur un nœud liste, les schémas de dépendance correspondants pour 0, 1, 2 et 3 fils. En effet, les restrictions que nous avons imposées sont telles qu'à partir de 4 fils, les schémas de dépendance sur tous les fils "du milieu" sont identiques à celles du fils du milieu dans une liste à 3 fils. L'interprétation se résume alors à ajouter, dans ce dernier cas, une boucle autour des actions de l'évaluateur sur le second fils pour qu'il soit ainsi capable de traiter tous les fils "du milieu" d'un nœud liste, quel que soit leur nombre.

3.2.3 Construction automatique de règles sémantiques manquantes

Le deuxième aspect intéressant d'Olga qui mette en jeu de façon importante l'analyse statique est la possibilité offerte au programmeur d'"oublier" de définir certaines règles sémantiques et de laisser le système les construire automatiquement. Cette facilité, quand les règles à construire sont limitées à des règles de copies, est connue depuis longtemps. On la trouve déjà dans les systèmes Delta [Lor77] et FNC/ERN [Jou84]. Nous l'avons étendue aux nœuds listes et implantée dans Fnc-2, puisque nous avions déjà pu tester, avec FNC/ERN, son intérêt et son efficacité pour la productivité du programmeur.

Cependant, il reste encore, dans la plupart des GA rencontrées en pratique, des groupes de règles sémantiques qui sont très similaires d'une production à une autre et qui expriment le même calcul dans des contextes légèrement différents. D'une part, pour le programmeur, écrire ces multiples variations du même modèle est fastidieux et guère gratifiant ; d'autre part, pour le lecteur, ces règles "inintéressantes" sont plutôt un obstacle à la compréhension car elles noient les calculs importants dans une masse de détails insignifiants. Notons ici que la construction automatique de règles de copie a été conçue pour répondre exactement à ces défauts, mais qu'elle ne suffit pas.

Nous avons donc défini et implanté dans Fnc-2 une extension d'Olga [JLP90, Le B89], qui consiste à définir des classes d'attributs et à donner, pour chacune de ces classes, un ensemble de *modèles* de règles sémantiques susceptibles de définir tout attribut qui lui appartient. En outre, chacun de ces modèles de règles sémantiques est subordonné à un modèle de production. Lorsqu'il s'agit de construire une règle sémantique manquante pour un certain attribut sur une certaine production, le système commence par déterminer quels sont les modèles de production qui correspondent à celle qu'il considère, puis essaye d'instancier les modèles de règles sémantiques qui leur sont attachés dans le contexte de cette production.

Si cela est possible pour un et un seul de ces modèles, la règle sémantique correspondante est effectivement créée. On pourra trouver un exemple d'application de ce mécanisme dans [JoP92].

Cette idée est similaire à celle présentée dans [DuC90], quoique la philosophie et les détails soient assez différents. On peut remarquer aussi que la construction automatique de règles de copie n'est qu'un cas particulier de ce mécanisme plus général, sauf que les modèles de règles n'ont pas à être donnés par l'utilisateur ; c'est d'ailleurs ainsi que nous l'avons (ré)implantée dans FNC-2.

Il est clair que la mise en œuvre de cette construction automatique de règles sémantiques manquantes exige une analyse assez sophistiquée, mais toujours statique, de la GA.

En pratique, cette facilité est tout à fait intéressante car, d'une part, elle réduit considérablement la taille d'une grammaire attribuée typique et, d'autre part — et c'était bien le but visé —, la lisibilité de la GA est très améliorée, puisque les règles sémantiques qui restent dans le "corps" proprement dit de la GA sont les seules qui apportent réellement de l'information. En outre, en combinant ce mécanisme de définition "par défaut", plutôt orienté vers les attributs que vers la syntaxe car les modèles de règles sont groupés en fonction des classes d'attributs, avec la notion de **phase**, au sens OLGA du terme, qui permet de séparer les règles sémantiques en plusieurs blocs (en répétant éventuellement les productions concernées, si cela est nécessaire), il est possible de structurer une GA, non plus selon la syntaxe, comme c'est le cas dans le formalisme classique, mais selon la sémantique, c'est-à-dire les attributs.

3.3 Les grammaires attribuées génériques

Avec cette section commence la partie de ce chapitre qui est consacrée aux extensions "lourdes" de la notion de grammaire attribuée, extensions destinées à accroître leur pouvoir d'expression et leur facilité d'utilisation. Les travaux qui seront décrits ne sont pas encore terminés et n'ont pas encore donné lieu à publication ; la présentation que j'en ferai sera donc informelle et, malheureusement, je ne pourrais pas renvoyer le lecteur à des descriptions plus détaillées.

Les *GA génériques* sont (ou seront) aux GA normales ce que la notion de module générique, en Ada par exemple, est à la notion de module simple. Un module générique décrit un algorithme, de manière plus ou moins générale, qui pourra ensuite s'appliquer, par *instanciation*, à différents types de données et mettre en jeu différentes fonctions spécifiés par l'utilisateur.

C'est cette même idée que nous voulons transporter dans le monde des GA avec les GA génériques. Un programmeur expérimenté décide un jour d'écrire une grammaire attribuée dans laquelle il veut spécifier un algorithme intéressant et de portée assez universelle, par exemple la résolution de la surcharge en Ada. Ce qui l'intéresse dans cette GA, c'est l'algorithme lui-même bien plus que la syntaxe de l'arbre qui le supporte. Il va donc choisir une syntaxe vraiment abstraite, réduite aux seules structures qui jouent un rôle dans cet algorithme. Par exemple, dans un algorithme d'identification et de manipulation de tables des symboles pour les langages dont les règles de visibilité sont celles de C ou d'Ada, la syntaxe ne comportera que la notion de bloc, la notion de déclaration d'un identificateur et la notion d'utilisation d'un identificateur. Il est intéressant de noter que cette même syntaxe peut aussi servir de support à un algorithme d'identification pour des règles de visibilité différentes, par exemple celles de Pascal.

Ainsi, la motivation fondamentale pour écrire une telle grammaire *gène* est de fournir sous une forme facilement réutilisable un concentré de l'expérience accumulée en écrivant des GA pour résoudre des problèmes classiques, en compilation ou dans d'autres domaines. Il est bien connu que cet accès à la connaissance et sa réutilisation facile sont une des clés de la productivité du programmeur et un des problèmes les plus importants du génie logiciel.

Pour que cette transmission de la connaissance s'effectue dans les meilleures conditions, il est indispensable, d'une part, qu'une GA générique soit facile à écrire par son concepteur et, d'autre part, que les programmeurs puissent aussi facilement récupérer la connaissance accumulée dans une GA générique et l'adapter à la syntaxe particulière sur laquelle ils travaillent. En ce qui concerne le premier point, il est évident que le plus simple est de faire en sorte qu'une GA destinée à former un gène soit une GA normale, sans autre contrainte ni obligation.

En ce qui concerne le second point, l'utilisateur a d'un côté, une GA gène, et de l'autre une syntaxe "nue". Le résultat qu'il veut obtenir est que les calculs décrits dans la GA gène soient "plaqués" ou "transportés" sur sa syntaxe nue. Or, dans la syntaxe de la GA gène, il n'existe, comme dit ci-dessus, que les structures utiles et nécessaires à l'expression de l'algorithme qu'elle spécifie. Il vient donc naturellement à l'idée de chercher à spécifier ce transport des calculs (instanciation) en associant, à chaque non-terminal de la GA gène, qui représente donc une structure jouant un rôle "sémantique" important, l'ensemble des non-terminaux de la syntaxe nue qui représentent la même structure. Ces non-terminaux "importants" sont appelés *non-terminaux clés*. Les ensembles de non-terminaux clés doivent être disjoints mais, bien entendu, ils ne sont pas sensés couvrir toute la grammaire à instancier.

Voyons maintenant les détails du processus d'instanciation. L'idée est qu'à tout arbre de la syntaxe à instancier, on veut associer un arbre de la syntaxe gène, sur lequel on pourra effectuer les calculs décrits par les règles sémantiques de la GA gène et les reporter sur l'arbre de départ. On commence donc par partitionner celui-ci en *régions*, c'est-à-dire en parties connexes dont la frontière (racine et feuilles) est formée uniquement de non-terminaux clés, tandis qu'aucun nœud intérieur n'est un non-terminal clé. On est ainsi certain de n'oublier dans l'arbre de départ aucune structure importante pour le calcul. On associe alors à chacune de ces régions le plus petit modèle d'arbre de la syntaxe gène qui a la même frontière, moyennant l'association donnée. Si ceci peut se faire de manière univoque, on peut alors transporter les sous-calculs effectués sur chaque modèle d'arbre du gène dans les régions correspondantes de l'arbre à instancier (cf. ci-dessous).

Le problème est que cette description du processus d'instanciation est éminemment dynamique, au sens de l'annexe B, alors que le programmeur voudrait obtenir statiquement une véritable GA, sur laquelle il pourrait continuer à travailler comme s'il l'avait écrite lui-même. Cependant, cette description dynamique fournit une piste vers la solution statique. En effet, cette construction de l'arbre gène à partir de l'arbre à instancier peut, moyennant certaines conditions, être décrite statiquement par une GA.

S'il est effectivement possible de construire cette GA d'instanciation à partir de la GA gène, de la syntaxe à instancier et de l'association des non-terminaux clés de cette dernière aux non-terminaux du gène (cf. ci-dessous), le transport des règles sémantiques du gène sur la syntaxe à instancier devient très simple : il se résume en effet à composer la GA d'instanciation avec la GA gène et de déclarer, dans la GA résultante, dont la syntaxe est celle qu'on voulait instancier, que les attributs de sortie sont ceux qui correspondent à ceux du gène, que l'on retrouve attachés aux non-terminaux clés. Le tour est joué. . .

Notre vue de ces GA génériques est sensiblement la même que la notion de GA composables, apparue récemment [2] [FMY92]. Dans cet article, les auteurs, qui ont à peu près les mêmes motivations que nous, soulèvent deux problèmes et énoncent les solutions qu'ils leur ont apportées : le premier est justement l'association d'un arbre gène à un arbre à instancier et le second est le replaquage des calculs de la GA gène sur l'arbre à instancier. En ce qui concerne ce dernier point, ils mentionnent que cela peut être fait par méta-composition, comme je l'ai montré ci-dessus, mais ils adoptent une autre méthode qui ne me semble pas très satisfaisante car très *ad hoc*. En ce qui concerne le premier problème, ils l'évitent complètement en demandant à l'utilisateur d'écrire lui-même la GA qui, à partir d'un arbre à instancier, construit effectivement l'arbre correspondant de la grammaire gène. Cette façon de procéder, si elle évite beaucoup de difficultés et reste la plus souple et la plus puissante, ne correspond pas du tout à notre conception de la généricité et de l'instanciation. En effet, pour nous, la généricité s'appuie beaucoup plus sur la *sémantique* d'une GA ou d'un langage que sur la syntaxe. En outre, comme dans Ada, nous voulons que le processus d'instanciation reste aussi transparent et simple à spécifier que possible. Or, ce qui porte vraiment la sémantique dans un arbre, ce sont les non-terminaux bien plus que les productions elles-mêmes. Je suis donc convaincu qu'il est utile de passer du temps à essayer de mettre réellement en œuvre la notion d'association entre non-terminaux, telle que je l'ai décrite ci-dessus.

Nous sommes actuellement en train de travailler sur ce problème, et nous avons déjà identifié des algorithmes et des restrictions sur les associations de non-terminaux clés de la syntaxe à instancier aux non-terminaux du gène, qui permettent d'assurer l'existence et l'unicité de l'arbre gène à associer à tout arbre à instancier et de construire statiquement une GA qui définit cette association par une construction d'arbre. La publication de ces résultats est prévue dans un futur proche.

Il nous manque de l'expérience, bien entendu, pour juger de l'utilité des GA génériques, mais je suis fermement convaincu que cela rendra les GA encore plus faciles à utiliser, car elles correspondent assez fidèlement à la notion de généricité qui existe dans plusieurs langages de programmation (Ada, ML, etc.) et qui est très appréciée des programmeurs par le confort et la productivité qu'elle leur apporte. La publication de l'article cité ci-dessus [FMY92] ne fait que confirmer que ce problème est intéressant à étudier et à résoudre.

Pour terminer, il serait intéressant d'étudier comment "abstraire" une GA normale en une GA générique.

3.4 Les grammaires attribuées dynamiques

La dernière utilisation d'analyse statique que je voudrais présenter maintenant, et qui est destinée à l'augmentation du pouvoir d'expression des GA, est la notion de grammaires attribuées *dynamiques*.

La motivation de nos travaux dans cette direction est la constatation qu'un évaluateur de GA à base de séquences de visite s'implante extrêmement facilement dans n'importe quel langage de programmation disposant de la récursion. Or, nous ne tirons pas parti, avec un tel évaluateur, de toute la puissance de ces langages. En particulier, chaque séquence de visite est une séquence de code tout à fait *linéaire*, qui ne contient pas d'instructions ou d'expressions conditionnelles qui pourraient agir sur le flot de contrôle de l'évaluateur ; en effet, dans une

2. Je voudrais cependant signaler que nos travaux sur les GA génériques ont commencé bien antérieurement à la publication de cet article, comme le prouvent les rapports d'activité de l'INRIA (éditions 1990 et 1991).

```
where while → EXPR STMT use
    $pre-state (EXPR) := $pre-state;
    if $value (EXPR) = false then
        view while → EXPR use
            $post-state := $post-state (EXPR);
        end view
    else view while → EXPR STMT W (:= while) use
            $pre-state (STMT) := $post-state (EXPR);
            $pre-state (W) := $post-state (STMT);
            $post-state := $post-state (W);
        end view
    end if;
end where;
```

FIG. 3.4 –. *Exemple de GA dynamique*

GA, les valeurs des attributs n'influent pas sur ce contrôle, et l'évaluateur n'est dirigé que par la structure de l'arbre. D'autre part, la récursion naturelle d'un évaluateur et des GA elles-mêmes est strictement calquée sur cette structure de l'arbre, alors que rien n'empêche, dans un langage de programmation classique, de faire de la récursion sur la même structure que celle qui est passée en entrée.

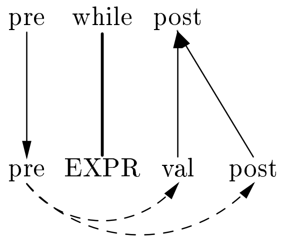
Nous voulons donc, avec cette notion de GA dynamique, incorporer ces deux constructions — la conditionnelle agissant sur le flot de contrôle, d'une part, et la récursion non stricte, d'autre part —, dans la définition même d'une GA. Comme on le voit dans la figure 3.4, qui présente l'exemple classique de la sémantique d'une boucle **while** dans un style plus ou moins dénotationnel, cette incorporation est fondée sur deux constructions :

1. la notion de *condition*, c'est-à-dire de prédicat sur des valeurs d'attributs, qui contrôle l'utilisation d'un jeu de règles sémantiques parmi plusieurs ;

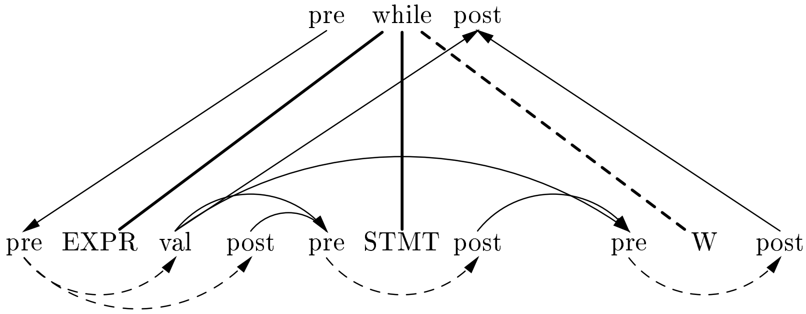
2. la notion de multiples *vues* de la même production, et en particulier le fait que l'on puisse considérer le nœud courant comme un de ses propres fils ; ceci permet d'exprimer des structures virtuelles de graphes fondées sur des structures physiques d'arbres et de faire, ainsi, de multiples passes sous un même sous-arbre.

Nos travaux sur ce sujet sont assez embryonnaires pour l'instant et demandent, d'une part, à être implantés pratiquement, par exemple dans FNC-2, et, d'autre part, à être testés et validés sur de multiples exemples.

En ce qui concerne l'implantation, nous ne pensons rencontrer aucun problème, car on peut la réaliser par *changement de plan*, notion que j'avais introduite dans la section 3.2.2. En effet, chaque feuille de l'arbre des conditionnelles (car elles peuvent être imbriquées, bien sûr), correspond à une unique vue, c'est-à-dire à une production virtuelle, et à un unique jeu de règles sémantiques qui définit bien toutes les occurrences d'attribut de sortie de cette vue (du moins c'est sensé être le cas). On peut donc, à partir d'une telle production "dynamique", construire plusieurs productions virtuelles, chacune ayant un schéma de dépendances bien formé. A titre d'exemple, les deux productions et schémas de dépendance correspondant à l'exemple de la boucle **while** (figure 3.4) sont présentés en figure 3.5. Il faut bien entendu ajouter des arcs de dépendance depuis tous les attributs utilisés dans une condition (ici, $va-



(a) \$value(EXPR) = false



(b) \$value(EXPR) = true

FIG. 3.5 –. *Les schémas de dépendance correspondant à la production "dynamique" de la figure 3.4*

lue(EXPR)) vers tous les attributs définis dans les sous-jeux de règles sémantiques contrôlés par cette condition. Ceci constitue l'étape de réduction.

Une fois l'évaluateur construit de façon standard à partir de ces productions virtuelles, il ne reste plus qu'à rassembler toutes les séquences de visite correspondantes en une seule séquence de visite, attachée à la production "physique" correspondante, en y insérant les instructions conditionnelles adéquates et en remplaçant les visites aux nœuds virtuels (par exemple le nœud W dans l'exemple ci-dessus) par des visites aux nœuds physiques correspondants (étape d'interprétation). Il est important de noter, cependant, que pour que ce changement de plan soit correct, il est indispensable que nous sachions stocker *tous* les attributs hors de l'arbre, puisque la récursion des calculs n'est plus strictement calquée sur la structure de l'arbre et qu'on ne peut donc pas se servir de de dernier comme support physique pour y accrocher des valeurs d'attributs. Nous avons montré, dans la section 2.2, que ceci est un problème résolu.

En ce qui concerne l'utilisation et la validation de la notion de GA dynamique, il nous reste à écrire de nombreux exemples la mettant en jeu, afin de déterminer quelle est son utilité pratique. Il est, bien entendu, encore trop tôt pour disserter sur les résultats, mais il me semble tout à fait raisonnable de penser que nous pouvons, ainsi, atteindre le pouvoir de la sémantique dénotationnelle ; je n'en ai pas encore de preuve, mais nous allons travailler dans cette direction. Une autre utilisation des GA dynamiques, et en particulier de la possibilité de récursion non stricte, est de définir et d'implanter des calculs de points fixes en restant presque strictement dans le cadre des GA ; la notion de GA circulaires est en effet un vieux serpent de mer, et aucune des propositions existant dans la littérature ne m'a jusqu'ici vraiment satisfait.

Enfin, il sera tout à fait intéressant de combiner la notion de GA dynamique avec celle de GA générique. En particulier, un des sujets de recherche que j'ai imaginé il y a déjà quelque temps concerne la génération automatique d'analyseurs de flot de données (AFD). Comme l'ont montré Kildall en 1973 [Kil73] et beaucoup d'autres depuis, tous les problèmes d'AFD se ressemblent et tous, jusqu'à un certain point, peuvent être résolus par la même famille d'algorithmes. Initialement, mon idée était de construire un générateur d'AFD, totalement indépendant de FNC-2, pour lequel je définirais un langage de description de problèmes d'AFD et qui produirait, à partir de la spécification d'un de ces problèmes, un module entier qui serait l'analyseur correspondant. Il me semble maintenant plus judicieux d'essayer d'obtenir le même résultat en combinant GA génériques et GA dynamiques, de la façon suivante. Presque tous les algorithmes d'AFD peuvent effectivement être décrits par des GA dynamiques, puisque

```
where int-decl → ID
declare
    value
        correct := ...
use
    ...
    $s-temp-st :=       if correct then
                            insert ($id(ID), varid, $h-temp-st)
                        else
                            $h-temp-st
                        end if;
end where;
```

FIG. 3.6 –. *Extrait de la GA décrivant une déclaration de variable en* simproc

celles-ci permettent de spécifier des calculs itératifs (par exemple calculs de points fixes) ainsi que certains calculs sur des structures de graphe. On peut ensuite considérer une telle GA comme une GA générique, que l'on pourrait instancier sur une syntaxe particulière et un problème d'AFD donné. Cette approche à base de GA me semble plus puissante et plus souple d'emploi que celle du générateur *ad hoc* ; par exemple, quand un nouvel algorithme d'AFD apparaît, il suffit de le décrire dans une nouvelle GA dynamique et générique pour pouvoir l'utiliser tout de suite (notion de bibliothèque de GA), alors que, dans l'autre cas, il faut modifier le générateur lui-même.

3.5 Travaux futurs

Outre l'exploration de la puissance d'expression et des applications des notions de GA dynamiques et de GA génériques, une des idées qu'il m'intéresse de développer est la suivante. De nombreuses règles sémantiques servent à exprimer la “mise à jour” d'une certaine structure de données par de nouvelles informations. Par exemple, la figure 3.6 présente un extrait de la grammaire attribuée de *simproc* [JoP92, ann. D] qui illustre la mise à jour de la table des symboles à la suite d'une nouvelle déclaration de variable. D'une manière générale, j'appelle “opération de mise à jour” toute fonction dont le résultat est du même type que l'un de ses paramètres. Ces opérations sont décrites ailleurs que dans la règle sémantique elle-même, par exemple par une fonction OLGA.

Il apparaît que les algorithmes d'optimisation de la gestion de la mémoire, présentés dans la section 2.2, permettent souvent de déterminer qu'après mise à jour, l'ancienne valeur d'une de ces structures n'est plus utile et qu'on peut stocker la nouvelle valeur, mise à jour, dans la même variable globale (ou dans le même sommet de pile). Ainsi, certaines règles sémantiques de la forme $X.a := f(Y.b, \dots)$ sont traduites, dans l'évaluateur, par des affectations de la forme $v := f(v, \dots)$.

Or, notre expérience montre que, si la traduction d'OLGA vers un langage d'implantation impératif comme C est relativement peu efficace en temps et surtout en place, c'est essentiellement parce que les opérations sur les structures de données sont des opérations *applicatives* qui ne doivent pas modifier physiquement une valeur existante (cf. dans la figure 3.7(a) le code OLGA de la fonction “insert”, utilisée dans la figure 3.6 pour mettre à jour la table des symboles). Or, dans le cas qui nous intéresse, le générateur d'évaluateurs détermine que l'ancienne valeur n'est plus réellement utilisée et qu'on pourrait la combiner, d'une manière

```
function insert (key: key-type; info: info-type; st: symbol-table): symbol-table is
declare
    function insert-in-tree (key: key-type; info: info-type; node: bin-tree): bin-tree is
        case node match
            bin-tree (null: void): { insert here }
                bin-tree (tree-node (key, info, node, node));
            bin-tree (tree-node(the-key, the-info, left-s, right-s)):
                if the-key = key then { replace old entry, while sharing as much as possible }
                    bin-tree (tree-node (key, info, left-s, right-s))
                elsif the-key < key then { insert in left subtree and reconstruct path to root }
                    bin-tree (tree-node (the-key, the-info, insert-in-tree (key, info, left-s), right-s))
                else
                    bin-tree (tree-node (the-key, the-info, left-s, insert-in-tree (key, info, right-s)))
                end if;
        end case
    end function { insert-in-tree };
in
    symbol-table (insert-in-tree (key, info, head (st))) + tail (st)
end function { insert };
```

(a) La fonction non destructrice *insert* en Olga

```
procedure insert (key: key-type; info: info-type; var st: symbol-table);
    procedure insert-in-tree (key: key-type; info: info-type; var node: bin-tree);
    begin
        if node = nil then { insert here }
            node := make_node (key, info, nil, nil)
        else if node↑.key = key then { replace old entry }
            node↑.info := info
        else if node↑.key < key then { insert in left subtree }
            insert-in-tree (key, info, node↑.left)
        else
            insert-in-tree (key, info, node↑.right)
    end { insert-in-tree };
begin { insert }
    insert-in-tree (key, info, st↑.head)
end { insert };
```

(b) La procédure destructrice *insert* en Pascal

Fig. 3.7 –. *Transformation de fonctions de mise à jour en procédures destructrices*

ou d'une autre, avec la nouvelle valeur. On aimerait donc, dans ce cas, qu'il existe, à côté de la fonction f, opération applicative et non destructrice, une procédure p, telle que $p(v, \dots)$ ait la même sémantique que $v := f(v, \dots)$, mais qui utiliserait des effets de bord destructeurs (remplacement physique de certains champs d'une structure existante), ce qui est plus efficace, bien sûr, mais aussi correct, puisque l'ancienne valeur de v ne sera plus jamais accédée. Par exemple, une version procédurale destructrice de l'opération d'insertion, codée en Pascal, est présentée dans la figure 3.7(b).

J'aimerais trouver le temps d'étudier le problème consistant à construire automatiquement le code de la procédure p à partir de la spécification, en OLGA ou autre, de la fonction f. Ce problème n'a apparemment fait l'objet que de peu de publications (il y en a des traces dans le monde de la programmation fonctionnelle, pour tirer parti des analyses de *single threading*), mais il est intéressant dans l'absolu et sa résolution peut considérablement augmenter l'efficacité de FNC-2 lui-même, et en particulier de l'utilisation d'OLGA. Il est possible que ce problème puisse être abordé plus élégamment, ou plus facilement, si ces fonctions de mise à jour sont spécifiées par des GA.

3.6 Conclusion

Dans ce chapitre, nous avons discuté plusieurs problèmes mettant en jeu l'analyse statique de grammaires attribuées afin d'augmenter, soit leur facilité d'utilisation, soit leur puissance d'expression. Puisque j'avais conclu, dans le chapitre précédent, qu'il n'y avait plus rien à faire sur les évaluateurs, je vais évidemment faire ici la conclusion inverse : je suis effectivement convaincu qu'il reste beaucoup de sujets de recherche, très intéressants à étudier, sur la notion de grammaire attribuée et, en particulier, son utilisation et ses extensions. Nous nous y attachons en ce moment, et nous continuerons dans les années qui viennent.

Chapitre 4

Conclusion

Tous les travaux que j'ai présentés dans ce mémoire partagent deux caractéristiques : la première est qu'ils concernent les grammaires attribuées et la seconde est qu'ils sont tous, plus ou moins, fondés sur la notion d'analyse statique. Le résultat brut de ces travaux est qu'il est plus facile, maintenant, de concevoir l'utilisation de ces GA en milieu industriel comme une solution raisonnable à certains problèmes de programmation et de génie logiciel. C'est justement le but que nous nous étions fixé en entreprenant la spécification et la construction du système FNC-2.

Certes, il reste énormément de travail pour que cet objectif devienne une réalité, mais mon pronostic est résolument optimiste. Déjà, certains industriels commencent à s'intéresser de près à FNC-2 et ont demandé à en disposer pour évaluer ses possibilités et ses performances. En particulier, certains de nos partenaires industriels du projet ESPRIT COMPARE ont décidé d'utiliser FNC-2 pour développer, au moins de manière prototypique, quelques applications dans le domaine de la compilation. Bien que les conditions de cette utilisation soient un peu particulières, à cause de ce partenariat ESPRIT, cet intérêt porté à FNC-2 nous montre que nous sommes sur la bonne voie.

Pour terminer en élargissant un peu le débat, il est clair que les travaux que j'ai présentés ici participent au même mouvement général que la plupart des autres travaux sur les langages de programmation : on cherche maintenant, ce qui n'était pas le cas jusqu'au début des années 1970, à privilégier essentiellement la propreté (mathématique), la facilité d'utilisation et la puissance d'expression des langages de programmation, même si les compilateurs doivent en contrepartie effectuer beaucoup de travail pour assurer l'efficacité du programme résultant. Un exemple que je connais assez bien et qui m'intéresse particulièrement est la parallélisation automatique de programmes écrits dans des langages *a priori* séquentiels, comme FORTRAN, de sorte que leur exécution sur des machines parallèles soit aussi efficace que possible. La thèse que j'ai soutenue dans ce mémoire au sujet des grammaires attribuées s'applique en fait à tous ces travaux : le seul moyen de concilier le "confort" d'un langage de programmation et l'efficacité des programmes exécutables résultants passe par l'analyse statique. Ce sujet n'a pas fini de faire couler de l'encre...

Annexe A

Un exemple de développement pas à pas d'une GA

Cet exemple est apparu pour la première fois dans l'article initial de Knuth sur les GA [Knu68], et a été maintenant adopté comme l'exemple standard par la plupart des chercheurs. C'est en effet un exemple très court, mais qui exhibe la plupart des notions de base et même les premières difficultés que posent les grammaires attribuées. Il est donc tout à fait pédagogique, à la fois pour étudier la théorie et les méthodes d'évaluation des grammaires attribuées, mais aussi, dans la mesure où je vais le présenter d'une façon incrémentale, pas à pas, comme exemple du processus intellectuel de développement d'une GA, qui montre que celles-ci sont aussi une bonne méthode de spécification et de développement de logiciel. Pour ne pas être trop long, je supposerai que le lecteur est assez familier avec les GA pour être capable d'en lire une.

L'objet de cette grammaire attribuée exemple est de calculer la valeur décimale de nombres réels écrits en notation binaire à virgule fixe. C'est un problème tellement simple que personne n'utiliserait, en pratique, une GA pour résoudre ce problème, mais dans tous les cas une GA est une bonne spécification pour quelque implantation que ce soit.

Les grammaires attribuées peuvent être vues comme des extensions des grammaires indépendantes du contexte ; la première étape que nous avons à effectuer pour définir cette GA est donc d'écrire une grammaire (syntaxique) décrivant les nombres binaires à virgule fixe. Les terminaux de cette grammaire seront les bits `0` et `1` et le point (décimal). Les non-terminaux sont B, qui représente un bit unique, L, qui représente une liste de bits consécutifs, c'est-à-dire un nombre ou une partie de nombre, et Z, qui est l'axiome et qui représente ainsi un nombre en virgule fixe complet. Les productions qui suivent ont une signification évidente.

$$
\begin{array}{llll}
p_1\colon & Z ::= L & p_4\colon & L ::= B \\
p_2\colon & Z ::= L\ .\ L & p_5\colon & B ::= \mathtt{1} \\
p_3\colon & L ::= L\ B & p_6\colon & B ::= \mathtt{0}
\end{array}
$$

Le lecteur se convaincra aisément que le langage engendré par cette grammaire est effectivement ce que nous voulons, c'est-à-dire l'ensemble des nombres binaires en virgule fixe.

A cette grammaire nous devons ajouter des attributs. Nous avons d'abord un attribut appelé v, pour valeur, qui représente la valeur décimale de la sous-chaîne engendrée par chaque non-terminal, *dans son contexte* ; il est de type réel et sera attaché aux non-terminaux Z, L et B.

En sus des attributs attachés aux non-terminaux, une GA étend sa grammaire sous-jacente par la donnée d'un ensemble de règles sémantiques, attachées aux productions. En l'occurrence, puisque nous avons défini que l'attribut v de chaque non-terminal représente la valeur exacte de la sous-chaîne qu'il dérive *dans son contexte*, nous aurons les règles sémantiques suivantes :

$$
\begin{array}{llcll}
p_1\colon & Z ::= L & \qquad & p_3\colon & L_0 ::= L_1\ B \\
& Z.v = L.v & & & L_0.v = L_1.v + B.v \\[1em]
p_2\colon & Z ::= L_1\ \texttt{.}\ L_2 & & p_4\colon & L ::= B \\
& Z.v = L_1.v + L_2.v & & & L.v = B.v
\end{array}
$$

Dans les productions p_2 et p_3 nous utilisons des indices pour distinguer les différentes occurrences des non-terminaux L. Les règles sémantiques des productions p_1 et p_4 s'appellent des *règles de copie*.

La valeur représentée par un chiffre `0` est toujours 0, donc nous pouvons aussi écrire :

$$
\begin{array}{ll}
p_6\colon & B ::= \texttt{0} \\
& B.v = 0
\end{array}
$$

mais la valeur représentée par un chiffre `1` dépend de la position de ce chiffre par rapport au point décimal. Puisque cette dernière information ne peut pas être calculée localement dans la production p_5, nous devons inventer un deuxième attribut, dont le rôle sera de transmettre cette position aux différents nœuds étiquetés par B. Appelons cet attribut s, pour échelle (*scale* en anglais); il sera de type entier. On peut donc écrire maintenant :

$$
\begin{array}{ll}
p_5\colon & B ::= \texttt{1} \\
& B.v = 2^{B.s}
\end{array}
$$

$B.s$ représente ainsi la position du chiffre par rapport au point décimal ; cette position est 0 pour le chiffre qui est immédiatement à gauche du point, 1 pour le deuxième plus à gauche, et ainsi de suite en croissant vers la gauche, et -1 pour le chiffre qui est immédiatement à droite du point, et ainsi de suite en décroissant vers la droite. Il ne reste plus qu'à résoudre le problème de calculer et de transmettre cette valeur de position aux nœuds B. Cette transmission doit, bien entendu, être effectuée par l'intermédiaire des nœuds L. La façon la plus simple de le faire est de décider que l'échelle d'une chaîne de chiffres dérivée d'un nœud L est celle de son chiffre le plus à droite ; cela conduit aux règles sémantiques suivantes :

$$
\begin{array}{llcll}
p_3\colon & L_0 ::= L_1\ B & \qquad & p_4\colon & L ::= B \\
& B.s = L_0.s & & & B.s = L.s \\
& L_1.s = L_0.s + 1 & & &
\end{array}
$$

Nous arrivons à la fin, c'est-à-dire au calcul de l'échelle des deux listes de chiffre dans les productions racines p_1 et p_2. Il n'y a aucun problème pour le L à gauche du point, ou l'unique L dans la production p_1, dont l'échelle est 0, d'après la discussion précédente :

$$
\begin{array}{llcll}
p_1\colon & Z ::= L & \qquad & p_2\colon & Z ::= L_1\ \texttt{.}\ L_2 \\
& L.s = 0 & & & L_1.s = 0
\end{array}
$$

D'après cette même discussion, la valeur de $L_2.s$ dans p_2 doit être l'échelle de son chiffre le plus à droite, c'est-à-dire l'opposé de la longueur de cette chaîne de chiffres. Mais cette information n'est pas non plus directement disponible dans p_2 ! Nous devons donc introduire

$$G = (N, T, P, Z), \text{ avec } N = \{Z, L, B\},\ T = \{\texttt{.}, \texttt{1}, \texttt{0}\},\ P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$$
$$A = \{v, s, l\},\ A(Z) = \{v\},\ A(L) = \{v, s, l\},\ A(B) = \{v, s\},\ S = \{v, l\},\ I = \{s\}$$
$$\text{type}(v) = \text{réel},\ \text{type}(s) = \text{type}(l) = \text{entier}$$

$$\begin{array}{llll}
p_1\colon\ Z ::= L & p_3\colon\ L_0 ::= L_1\ B & p_4\colon\ L ::= B & p_5\colon\ B ::= \texttt{1} \\
\quad Z.v = L.v & \quad L_0.v = L_1.v + B.v & \quad L.v = B.v & \quad B.v = 2^{B.s} \\
\quad L.s = 0 & \quad B.s = L_0.s & \quad B.s = L.s & \\
 & \quad L_1.s = L_0.s + 1 & \quad L.l = 1 & p_6\colon\ B ::= \texttt{0} \\
p_2\colon\ Z ::= L_1\ \texttt{.}\ L_2 & \quad L_0.l = L_1.l + 1 & & \quad B.v = 0 \\
\quad Z.v = L_1.v + L_2.v & & & \\
\quad L_1.s = 0 & & & \\
\quad L_2.s = -L_2.l & & &
\end{array}$$

FIG. A.1 –. *La GA complète des nombres binaires*

un troisième et dernier attribut, l comme longueur, qui sera attaché au non-terminal L, et de type entier. On peut donc terminer la spécification du calcul de $L_2.s$ dans p_2 en écrivant :

$$\begin{array}{l} p_2\colon\ Z ::= L_1\ \texttt{.}\ L_2 \\ \quad L_2.s = -L_2.l \end{array}$$

Les règles nécessaires pour calculer cette longueur sont très simples :

$$\begin{array}{ll} p_3\colon\ L_0 ::= L_1\ B & \qquad p_4\colon\ L ::= B \\ \quad L_0.l = L_1.l + 1 & \qquad \quad L.l = 1 \end{array}$$

Ceci conclut la spécification de notre grammaire attribuée, qui est rassemblée dans la figure A.1.

Comme il est dit dans l'introduction, on voit bien, sur cet exemple suivi pas à pas, que chaque étape du développement d'une GA consiste à résoudre une partie très simple du problème, qui se traduit par :

1. l'introduction d'un nouvel attribut pour résoudre un problème précis ;
2. la définition de sa sémantique globale ;
3. la spécification des règles sémantiques définissant la valeur de cet attribut dans chacune des productions où il apparaît comme occurrence d'attribut de sortie.

Cette méthode de développement s'applique aussi à des applications très importantes, comme une partie avant de compilateur. Les règles de typage de Pascal, par exemple, se déduisent facilement de la sémantique décrite dans la norme [AFN82] et de la sémantique globale d'un attribut choisi pour représenter le type d'une expression. De même, la construction et l'utilisation d'une table des symboles, ainsi que la vérification des règles de visibilité, se déterminent à partir de la norme, avec une étape "Eurêka" supplémentaire qui consiste à savoir que toute notion d'identification (association de la déclaration d'un identificateur à ses utilisations) se résout en général par la manipulation d'une table des symboles.

Annexe B

Un exemple d'analyse statique exacte : le test de non-circularité

Cette annexe comporte deux parties : d'une part, une présentation des définitions et résultats essentiels sur les GA, d'autre part ce qui correspond effectivement au titre de l'annexe, c'est-à-dire le test de non-circularité ; si j'inclus la première partie, qui n'apporte guère d'informations nouvelles, c'est parce que j'en ai besoin dans la seconde, bien sûr.

Il n'y ici aucun résultat nouveau, *a fortiori* aucun de *mes* résultats, mais ceux que je présente insistent sur la dualité entre vue statique et vue dynamique d'une GA, ce qui permet de justifier la thèse que je soutiens dans ce mémoire ; j'ai employé ce point de vue pour la première fois dans [FrJ91] et, de fait, cette annexe est une traduction plus ou moins fidèle de la section 3 du chapitre 2 de ce dernier document.

Le style de la présentation ci-dessous est assez formel, et je suppose donc que le lecteur est familier avec la notion de GA, même s'il n'en connaît pas tous les détails théoriques.

Dans toute cette annexe, "notre GA exemple" désigne la GA des nombres binaires décrite dans l'annexe précédente et présentée dans la figure A.1. La phrase exemple qui servira de base à tous les autres exemples est `1101.11`, dont l'arbre dérivation est présenté dans la figure B.1.

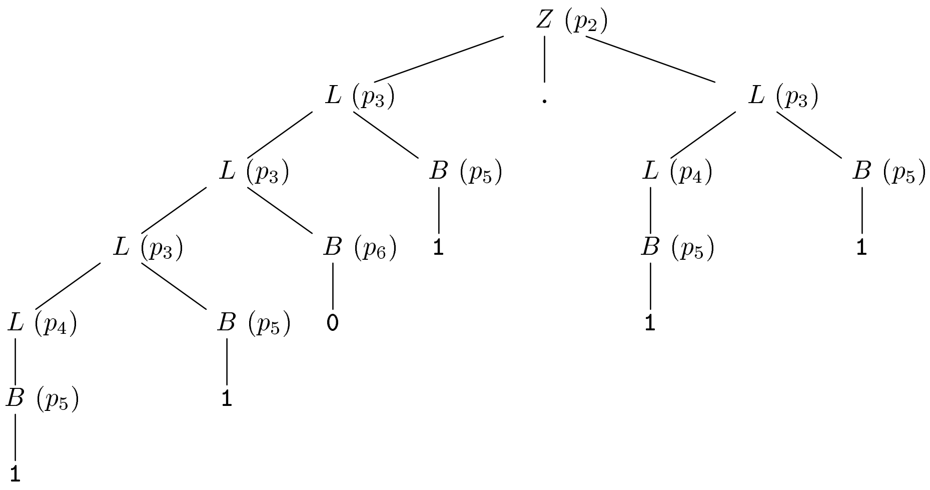


Fig. B.1 –. *Arbre de dérivation du texte* `1101.11`

B.1 Principales définitions sur les GA

B.1.1 Rappels

Je supposerai que le lecteur est familier avec les grammaires indépendantes du contexte, les arbres définis par une grammaire (classique) ou une algèbre d'opérateurs (qui sont isomorphes, de toutes façons, puisque j'ignorerai totalement les terminaux), les relations et les graphes. Puisque je n'utiliserai que des relations binaires, qui sont isomorphes aux graphes, il m'arrivera souvent de confondre les unes avec les autres sans le dire explicitement.

J'utiliserai les notations suivantes :

- Une grammaire indépendante du contexte est un quadruplet $G = (N, T, P, Z)$ où N est l'ensemble des non-terminaux, T l'ensemble des terminaux, $P \subset N \times (N \cup T)^*$ l'ensemble des productions et $Z \in N$ l'axiome.

  Je supposerai que G est réduite. En outre, puisque les terminaux ne jouent aucun rôle dans cette théorie, je les oublierai purement et simplement et supposerai donc que chaque production est de la forme :

  $$p: \; X_0 \to X_1 X_2 \cdots X_{n_p}$$

  où $\forall i, 0 \leq i \leq n_p, X_i \in N$.

- Soit t un arbre défini par une certaine grammaire G, c'est-à-dire, sans perte de généralité, un arbre de dérivation pour une certaine phrase de $L(G)$. Pour tout nœud u dans t, soit $P(u) \in P$ la production appliquée en u et $N(u) \in N$ le non-terminal qui étiquette u (c'est-à-dire $N(u) = \mathrm{LHS}(P(u))$). Si $P(u) : \; X_0 \to X_1 X_2 \cdots X_{n_p}$, les fils de u seront notés $u.1, u.2, \ldots, u.n_p$; on assimile $u.0 = u$. On utilisera aussi les notions bien connues de sous-arbre de t issu de u, noté t/u, et de contexte de u dans t, noté $t \setminus u$.

B.1.2 Définitions fondamentales

Définition 1 (Grammaire attribuée (vue statique)) [Knu68] *Une* grammaire attribuée *est un triplet $GA = (G, A, F)$ où :*

1. *G est une grammaire indépendante du contexte $G = (N, T, P, Z)$ telle que définie ci-dessus.*

2. *A est un ensemble fini d'*attributs. *A est une partition $A = I \uplus S$ d'attributs* hérités *et d'attributs* synthétisés. *A chaque non-terminal $X \in N$ est attaché un sous-ensemble de ces attributs $A(X) = I(X) \cup S(X)$, avec $I(X) \subseteq I$ et $S(X) \subseteq S$. L'axiome n'a aucun attribut hérité : $I(Z) = \emptyset$.*

   Dans chaque production $p: \; X_0 \to X_1 X_2 \cdots X_{n_p}$, on définit l'ensemble des occurrences d'attribut *de p :*

   $$W(p) = \{X_i.a \mid 0 \leq i \leq n_p \wedge a \in A(X_i)\}$$

   *Cet ensemble est partitionné en les occurrences d'*entrée *et de* sortie *de p :*

   $$\begin{array}{rcl} IW(p) & = & \{X_i.a \mid i = 0 \wedge a \in I(X_0) \vee 1 \leq i \leq n_p \wedge a \in S(X_i)\} \\ OW(p) & = & \{X_i.a \mid i = 0 \wedge a \in S(X_0) \vee 1 \leq i \leq n_p \wedge a \in I(X_i)\} \end{array}$$

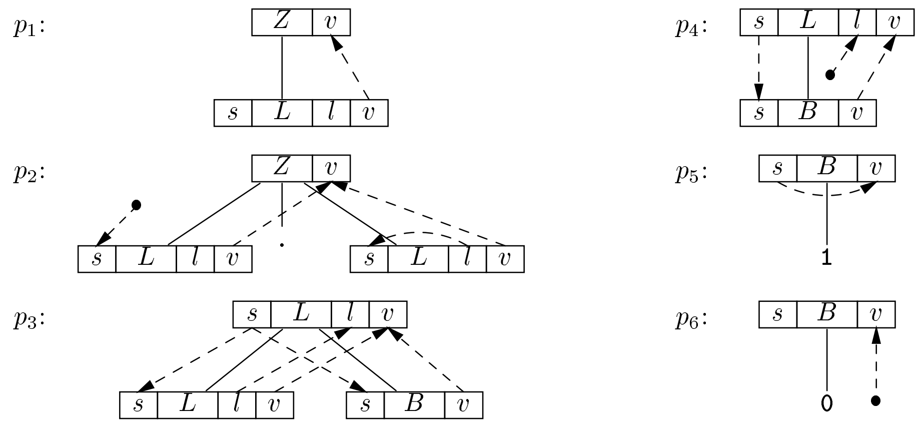


FIG. B.2 –. *Graphes de dépendance des productions pour la GA de la figure A.1*

3. *F est un ensemble de* règles sémantiques $F = \{F(p) \mid p \in P\}$. *Pour chaque $p \in P$, $F(p)$ doit contenir exactement une règle sémantique définissant chaque occurrence d'attribut de sortie de p et aucune autre règle : $F(p) = \{r_{p,i,a} \mid X_i.a \in OW(p)\}$.*

   Une règle sémantique exprime comment la valeur d'une occurrence d'attribut de sortie est calculée en termes des occurrences d'attribut utilisées *de p :*

$$r_{p,i,a}\ : X_i.a = f_{p,i,a}(X_{k_1}.a_1, X_{k_2}.a_2, \ldots, X_{k_m}.a_m)$$

   où $f_{p,i,a}$ est une fonction appropriée, $m \geq 0$ et $\forall j, 1 \leq j \leq m, X_{k_j}.a_j \in UW(p)$.

   $UW(p)$ est soit $W(p)$ tout entier, soit réduit à $IW(p)$.

La discussion sur $UW(p)$ est retardée jusqu'après la définition suivante.

Cette définition des GA est fondée sur les grammaires "concrètes" définies en BNF. Elle s'étend facilement à des formes de grammaires plus abstraites, en particulier la BNF étendue [JuD84] et les grammaires abstraites avec des opérateurs de liste [JLP90, VoM82].

Définition 2 (Dépendances (1)) *Soit $GA = (G, A, F)$ une grammaire attribuée et $r_{p,i,a}$ une règle sémantique dans F :*

$$r_{p,i,a}\ : X_i.a = f_{p,i,a}(X_{k_1}.a_1, X_{k_2}.a_2, \ldots, X_{k_m}.a_m)$$

On dit, à cause de cette règle, que $X_i.a$ dépend *de $X_{k_1}.a_1, X_{k_2}.a_2, \ldots, X_{k_m}.a_m$ dans la production p, ce qu'on note $X_{k_j}.a_j \underset{p}{\rightarrow} X_i.a$.*

Le graphe de dépendance de la production *p, noté $D(p)$, est celui de la relation $\underset{p}{\rightarrow}$, dont les sommets sont les éléments de $W(p)$.*

La figure B.2 représente l'ensemble des graphes de dépendance de toutes les productions de notre GA exemple ; le graphe de dépendance d'une production est, par convention, dessiné par dessus le fragment d'arbre qui représente cette production.

Si $UW(p) = IW(p)$ la GA est dite être en *forme normale* (de Bochmann) [Boc76]. Notre GA exemple est en forme normale. Les GA en forme normale sont considérées plus faciles à lire et plus faciles à utiliser dans les discussions théoriques ; de plus, il existe une macro-expansion textuelle simple transformant toute GA en une GA équivalente en forme normale

[Boc76], à condition qu'aucun des graphes de dépendance de production n'ait de cycle, ce qui fait qu'imposer la forme normale n'est guère restrictif. Dans toute la suite je supposerai que toute GA sera en forme normale.

Définition 3 (grammaire attribuée (vue dynamique)) *Soit* $GA = (G, A, F)$ *une grammaire attribuée et* t *un arbre défini par* G.

A chaque nœud $u \in t$ *est attaché un ensemble d'*instances d'attribut $A(u) = \{u.a \mid a \in A(N(u))\}$. *On note* $W(t) = \bigcup_{u \in t} A(u)$ *l'ensemble de toutes les instances d'attribut dans l'arbre* t, *de même pour* $W(t/u)$ *et* $W(t \setminus u)$. *Notons que, puisque* u *appartient à la fois à* t/u *et à* $t \setminus u$, *les instances d'attribut de* $A(u)$ *appartiennent à la fois à* $W(t/u)$ *et à* $W(t \setminus u)$.

Soit $K(t)$ *le système d'équations construit comme suit. L'ensemble des inconnues est* $W(t)$. *Pour chacune de ces inconnues* $u.a$, *l'équation la définissant est construite comme suit :*

- *Si* $a \in S$, *soit* $p = P(u)$. *Dans* $F(p)$ *il existe une et une seule règle sémantique définissant* $X_0.a$, $r_{p,0,a} : X_0.a = f_{p,0,a}(X_{k_1}.a_1, X_{k_2}.a_2, \ldots, X_{k_m}.a_m)$. *Alors l'équation suivante est dans* $K(t)$ *:*
$$u.a = f_{p,0,a}(u.k_1.a_1, u.k_2.a_2, \ldots, u.k_m.a_m)$$
- *Sinon,* $a \in I$. *Soit alors* v *le père de* u *dans* t, $p = P(v)$ *et* i *tel que* $u = v.i$, *avec* $1 \leq i \leq n_p$. *Dans* $F(p)$ *il existe une et une seule règle sémantique définissant* $X_i.a$, $r_{p,i,a} : X_i.a = f_{p,i,a}(X_{k_1}.a_1, X_{k_2}.a_2, \ldots, X_{k_m}.a_m)$. *Alors l'équation suivante est dans* $K(t)$ *:*
$$u.a = f_{p,i,a}(v.k_1.a_1, v.k_2.a_2, \ldots, v.k_m.a_m)$$

Aucune autre équation n'est dans K(t).

Toutes les définitions que nous avons vues jusqu'ici nous permettent d'établir certaines correspondances entre des entités *statiques*, c'est-à-dire existant dans une GA elle-même, et des entités *dynamiques*, c'est-à-dire n'existant qu'à "l'exécution" de cette GA (cf. la section B.1.3) :

vue statique	*vue dynamique*
un(e)	plusieurs
grammaire	arbre
non-terminal	nœud d'arbre
production	étiquette de nœud
occurrence d'attribut	instance d'attribut
règle sémantique	équation

Le système d'équations correspondant à la GA exemple de la figure A.1 et l'arbre de dérivation de la figure B.1 est présenté dans la figure B.3, dans laquelle les inconnues sont notées a_i, où a est le nom de l'attribut et i le numéro du nœud dans un ordre gauche-droite en largeur d'abord.

Propriété 1 [Knu68] *Pour chaque arbre* t *le système d'équations* $K(t)$ *est bien formé, c'est-à-dire que toute inconnue dans* $W(t)$ *est définie par une et une seule équation.*

$v_1 = v_2 + v_3$	$s_7 = s_3$	$v_6 = v_{10}$	$v_9 = 0$
$s_2 = 0$	$s_6 = s_3 + 1$	$s_{10} = s_6$	$v_{10} = 2^{s_{10}}$
$s_3 = -l_3$	$l_3 = l_6 + 1$	$l_6 = 1$	$v_{11} = v_{13}$
$v_2 = v_4 + v_5$	$v_4 = v_8 + v_9$	$v_7 = 2^{s_7}$	$s_{13} = s_{11}$
$s_4 = s_2 + 1$	$s_9 = s_4$	$v_8 = v_{11} + v_{12}$	$l_{11} = 1$
$s_5 = s_2$	$s_8 = s_4 + 1$	$s_{12} = s_8$	$v_{12} = 2^{s_{12}}$
$l_2 = l_4 + 1$	$l_4 = l_8 + 1$	$s_{11} = s_8 + 1$	$v_{13} = 2^{s_{13}}$
$v_3 = v_6 + v_7$	$v_5 = 2^{s_5}$	$l_8 = l_{11} + 1$	

FIG. B.3 –. *Le système d'équations pour décorer l'arbre de dérivation de la figure B.1*

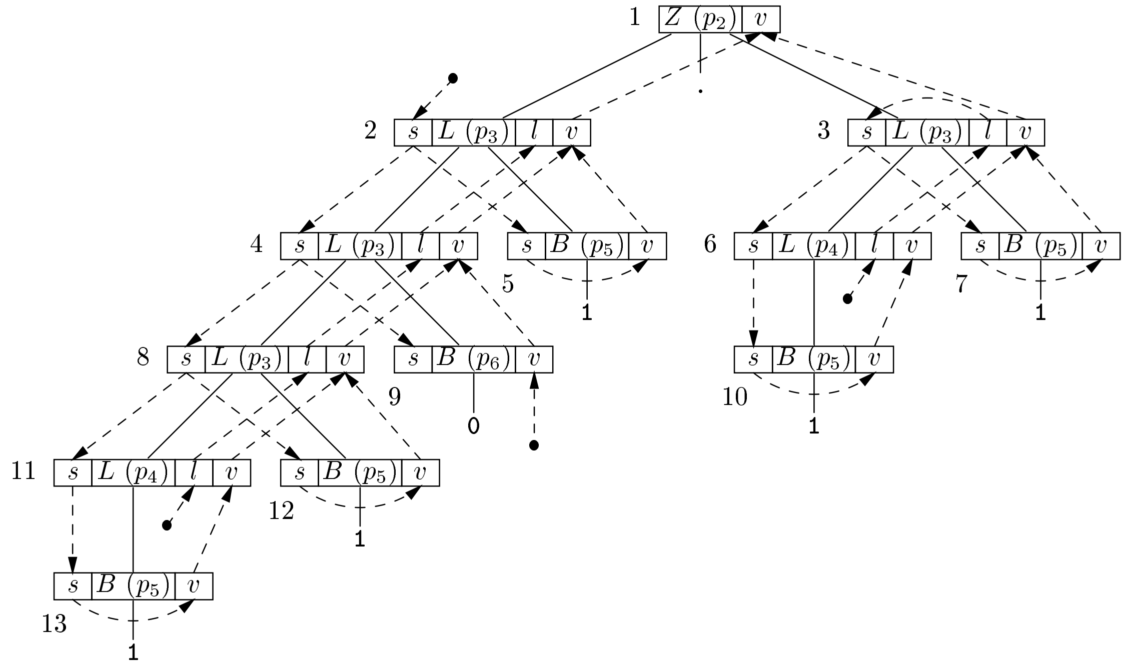


FIG. B.4 –. *Le graphe de dépendance complet (arcs orientés) de l'arbre de la figure B.1*

Définition 4 (Dépendances (2)) *Soit $GA = (G, A, F)$ une grammaire attribuée, t un arbre défini par G et $W(t)$ et $K(t)$ l'ensemble d'inconnues et le système d'équations associés. Soit une équation quelconque de $K(t)$:*

$$w = f_{p,i,a}(w_1, w_2, \ldots, w_m)$$

On dit, à cause de cette équation, que w dépend *de $w_1, w_2, \ldots, w_m$ dans l'arbre t, ce qu'on note $w_j \underset{t}{\rightarrow} w$.*

Le graphe de dépendance complet *de t, noté $D(t)$, est celui de la relation $\underset{t}{\rightarrow}$, dont les sommets sont les éléments de $W(t)$.*

La figure B.4 représente le graphe de dépendance complet pour l'arbre de la figure B.1 ; le graphe de dépendance complet associé à un arbre est, par convention, dessiné par dessus cet arbre.

Les dépendances statiques (de production) et dynamiques (complètes) sont liées par le théorème suivant, qui montre que le graphe de dépendance complet est construit en "collant" les unes aux autres des copies des graphes de dépendance de production selon la structure de l'arbre. Par exemple, à partir des figures B.1 et B.2 on peut construire la figure B.4.

Théorème 1 *Soit $GA = (G, A, F)$ une grammaire attribuée et $D(p)_{p \in P}$ ses graphes de dépendance de production ; soit t un arbre défini par G et $D(t)$ le graphe de dépendance complet qui lui est associé.*

Pour chaque nœud u de t, soit $D(u)$ le graphe construit comme suit. Soit $p = P(u)$, $p : X_0 \to X_1 X_2 \cdots X_{n_p}$. Les sommets de $D(u)$ sont les instances d'attribut $\{u.i.a \mid 0 \leq i \leq n_p \wedge a \in A(X_i)\}$. Les arcs de $D(u)$ sont dérivés de ceux de $D(p)$: pour tout arc $X_i.a \to X_j.b$ dans $D(p)$, il y a un arc $u.i.a \to u.j.b$ dans $D(u)$.

Alors :

$$D(t) = \bigcup_{u \in t} D(u)$$

Ce théorème nous permet d'ajouter une nouvelle ligne à notre table des correspondances statiques/dynamiques :

vue statique	*vue dynamique*
plusieurs	beaucoup
graphe de dépendance de production	graphe de dépendance complet

Le fait que tous les graphes de dépendance complets possibles soient composés à partir des graphes de dépendance de production, qui sont connus statiquement, est important car il permet de *simuler statiquement toutes les situations dynamiques possibles.* C'est sur cette observation, ainsi que sur le fait que la sémantique d'une GA est directement liée aux dépendances (cf. la définition suivante), que repose la thèse que je développe dans ce mémoire. Voir aussi la proposition 3 et le théorème 3 ci-dessous.

B.1.3 Sémantique d'une GA

Définition 5 *Soit $GA = (G, A, F)$ une grammaire attribuée, t un arbre défini par G et $W(t)$ et $K(t)$ l'ensemble d'inconnues et le système d'équations associés.*

La signification *ou la* valeur sémantique *de t selon GA est la solution de $K(t)$, c'est-à-dire les valeurs de toutes les inconnues de $W(t)$ ou d'un sous-ensemble particulier d'entre elles, généralement les attributs (synthétisés) de la racine de t.*

Une GA définit ainsi une traduction ou une application de $L(G)$ ou de l'ensemble des arbres définis par G vers un certain domaine sémantique.

D'un point de vue purement formel, il n'y a rien à ajouter à cette définition de la sémantique d'une GA. D'un point de vue opérationnel cependant, elle implique, à moins que les fonctions utilisées dans les règles sémantiques n'aient des propriétés particulières, que la seule façon de calculer la valeur sémantique d'un arbre consiste à interpréter les équations comme des affectations (orientées) et à “exécuter” ces affectations dans un ordre compatible avec celui (partiellement) imposé par les dépendances, afin de calculer la valeur des inconnues demandées. Là encore, l'analyse statique de ces dépendances permet d'améliorer le comportement des évaluateurs (cf. le chapitre 2). Mais revenons à notre exemple d'analyse statique exacte, le test de non-circularité.

B.2 Analyse statique des GA : l'exemple de la non-circularité

B.2.1 La non-circularité

Puisque nous n'imposons aucune propriété particulière des fonctions sémantiques $f_{p,i,a}$ et de leurs domaines qui permettrait de calculer des points fixes, la bonne formation de $K(t)$ n'est pas suffisante pour assurer qu'il est résoluble : on doit imposer en plus une propriété syntaxique de *non-circularité*.

Définition 6 (Non-circularité) *Un graphe de dépendance complet $D(t)$ associé à un certain arbre t d'une grammaire attribuée GA est dit* non circulaire *s'il est acyclique, c'est-à-dire* $\nexists w \in W(t), w \underset{t}{\rightarrow}^{+} w$.

Une grammaire attribuée GA est dite elle-même non circulaire si, pour tout arbre t qu'elle définit, $D(t)$ est non circulaire.

Notons que ceci définit la non-circularité comme une propriété *dynamique*, mais le théorème suivant montre qu'en fait on peut la tester statiquement.

Théorème 2 [Knu68, *corr.*] *La propriété de non-circularité est intrinsèque à une GA donnée et peut être vérifiée statiquement.*

Ce théorème sera prouvé en exhibant un algorithme qui teste la propriété de non-circularité avec une GA comme seule donnée, mais il nous faut d'abord introduire quelques nouvelles notions techniques.

B.2.2 De nouvelles définitions

Définition 7 (Graphe de dépendance augmenté) *Soit $GA = (G, A, F)$ une grammaire attribuée et p une production dans P, $p\ :\ X_0 \rightarrow X_1X_2 \cdots X_{n_p}$. Soit $(\gamma_i)_{0\leq i\leq n_p}$ une famille de relations (graphes) telle que, $\forall i, 0 \leq i \leq n_p$, γ_i est définie sur $A(X_i)$.*

Le graphe de dépendance augmenté $D(p)[\gamma_0, \gamma_1, \ldots, \gamma_{n_p}]$ *est défini comme suit :*

- *Ses sommets sont les éléments de $W(p)$.*
- *Il y a deux sortes d'arcs :*
  1. *ceux de $D(p)$;*
  2. *ceux provenant des γ_i : pour tout arc $a \rightarrow b$ dans γ_i, il y a un arc $X_i.a \rightarrow X_i.b$ dans $D(p)[\gamma_0, \gamma_1, \ldots, \gamma_{n_p}]$.*

  Aucun autre arc ne figure dans $D(p)[\gamma_0, \gamma_1, \ldots, \gamma_{n_p}]$.

Cette notion sera utilisée la plupart du temps quand les γ_i représentent des ordres partiels ou totaux sur les attributs de X_i, que nous interpréterons comme des dépendances entre les attributs. De plus, comme le contexte définira toujours de manière non ambiguë l'ensemble des sommets dans un tel γ_i, nous réduirons quelquefois ce dernier à son ensemble d'arcs.

Définition 8 (Projection d'un graphe de production sur un non-terminal) *Soit $GA = (G, A, F)$ une grammaire attribuée et p une production de P, $p\ :\ X_0 \rightarrow X_1X_2 \cdots X_{n_p}$. Soit Γ une relation (graphe) sur $W(p)$.*

Des “graphes de dépendance de non-terminaux”

Graphe de dépendance de la production p_3

Graphe de production augmenté : les arcs en trait continu viennent du graphe de dépendance de la production, ceux en pointillés viennent des graphes de dépendance des non-terminaux

Projection de la fermeture transitive de ce graphe augmenté sur le non-terminal en partie gauche = (nouveau) graphe de dépendance pour ce non-terminal

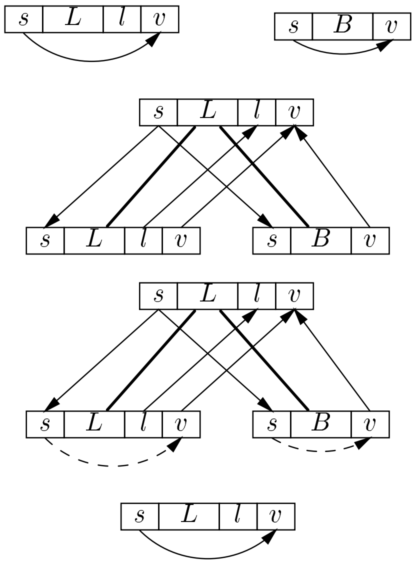


FIG. B.5 –. *Construction d'un graphe de dépendance de production augmenté et sa projection sur un certain non-terminal*

La projection de Γ sur le non-terminal X_i, $0 \leq i \leq n_p$, *notée* $\Gamma|_{X_i}$ *ou* $\Gamma|_i$, *est la relation (graphe) définie comme suit :*

- *Ses sommets sont les attributs de* X_i, $A(X_i)$.
- *Il y a un arc de a à b dans* $\Gamma|_{X_i}$ ($a \underset{\Gamma|_{X_i}}{\longrightarrow} b$) *pour tout chemin non vide de* $X_i.a$ *à* $X_i.b$ *dans* Γ ($X_i.a \underset{\Gamma}{\longrightarrow}^{+} X_i.b$) *qui ne passe par aucun autre attribut de* X_i. *Aucun autre arc ne figure dans* $\Gamma|_{X_i}$.

Cette notion sera utilisée la plupart du temps quand Γ est un graphe de dépendance augmenté construit comme ci-dessus. Dans ce cas, et si ce graphe est acyclique, les projections résultantes seront elles aussi des ordres partiels ou totaux sur les attributs d'un même non-terminal, que nous interpréterons encore comme des dépendances entre ses attributs.

Notez que, puisque nos GA sont en forme normale, les arcs dans ces “graphes de dépendance de non-terminal” vont d'un attribut hérité à un synthétisé ou vice-versa.

Les processus de construction d'un graphe de dépendance de production augmenté et de sa projection sur un certain non-terminal sont tous deux illustrés dans la figure B.5.

B.2.3 Graphes de dépendance synthétisés et hérités

Le test de non-circularité est fondé sur la construction d'un ensemble complet de tels “graphes de dépendance de non-terminal”, appelés *graphes de dépendance synthétisés* (ou graphes caractéristiques inférieurs [Rep84], etc.). Je présenterai aussi la notion duale de graphes de dépendance hérités, bien qu'elle soit inutile pour le test de non-circularité.

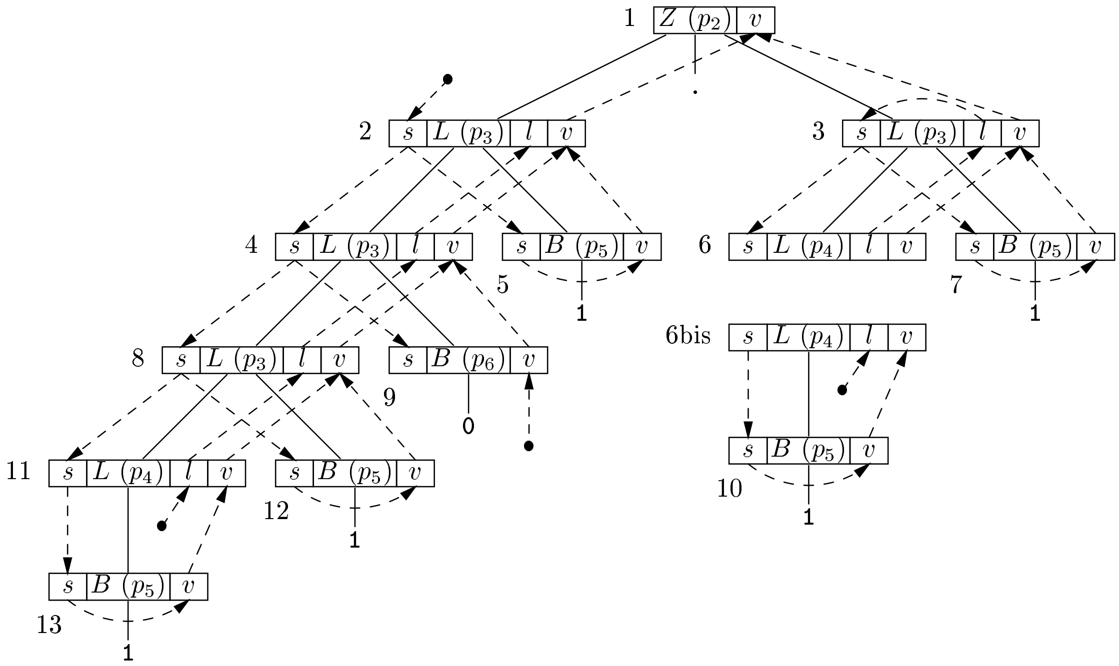


FIG. B.6 –. *Graphe de dépendance complet de la figure B.4 "coupé" au nœud 6*

Observons d'abord que les graphes de dépendance complets construits à partir d'une GA ont une forme très spéciale : en un certain sens, chaque nœud dans un arbre "partitionne" le GDC associé en deux sous-graphes, de la même façon qu'il partitionne l'arbre en deux morceaux disjoints (cf. le théorème 1). C'est ce que dit la proposition suivante.

Propriété 2 *Soit $GA = (G, A, F)$ une grammaire attribuée, t un arbre défini par G, $D(t)$ le graphe de dépendance complet associé et u un nœud dans t.*

Tout chemin dans $D(t)$ d'une instance d'attribut de $D(t/u)$ à une instance de $D(t \setminus u)$ ou vice-versa doit passer par au moins une instance d'attribut de u.

On peut alors écrire :

Définition 9 (Dépendances (3)) *Soit $GA = (G, A, F)$ une grammaire attribuée, t un arbre défini par G, $D(t)$ le graphe de dépendance complet associé et u un nœud dans t.*

Le graphe de dépendance inférieur *de u dans t (resp. le* graphe de dépendance supérieur*), noté $D(t/u)$ (resp. $D(t \setminus u)$), est défini comme la restriction de $D(t)$ à $W(t/u)$ (resp. $W(t \setminus u)$).*

Cette restriction est bien définie puisque, d'après la proposition ci-dessus, il n'existe pas de chemin du GD inférieur au GD supérieur ou vice-versa qui ne passe pas par un attribut de u.

La figure B.6 représente les GD inférieur et supérieur du nœud 6 dans l'arbre de la figure B.1. Notons qu'ils sont obtenus très facilement en "coupant" le GDC à ce nœud.

Définition 10 (Graphes de dépendance synthétisés et hérités) *Soit $GA = (G, A, F)$ une grammaire attribuée, t un arbre défini par G et u un nœud dans t.*

Le graphe défini comme suit :

– *son ensemble de sommets est $A(N(u))$;*

FIG. B.7 –. *Projections des GD supérieur et inférieur de la figure B.6 sur le nœud 6*

– *il y a un arc de a à b dans ce graphe si et seulement si il y a un chemin non vide de $u.a$ à $u.b$ dans $D(t/u)$ (resp. $D(t \setminus u)$) qui ne passe par aucun autre attribut de u ;*

est un *des* graphes de dépendance synthétisés (resp. hérités) *de* $N(u)$. *L'ensemble complet des graphes de dépendance synthétisés et des graphes de dépendance hérités de tous les non-terminaux de GA est défini en appliquant cette construction à tous les nœuds de tous les arbres définis par GA.*

Il est très important de noter que ces graphes représentent *réellement* des dépendances qui peuvent apparaître dans un arbre réel. En outre, avec nos GA en forme normale, les arcs d'un GDS vont forcément d'un attribut hérité à un attribut synthétisé, et vice-versa dans un GDH.

La figure B.7 représente le GDH et le GDS de L qui résultent de l'application de la définition ci-dessus au nœud 6 dans l'arbre de la figure B.1 ; les GD supérieur et inférieur correspondants ont été présentés dans la figure B.6.

Propriété 3 *Pour une grammaire attribuée donnée il n'existe qu'un nombre fini de GDS et GDH pour chaque non-terminal.*

La preuve est évidente puisque le nombre d'attributs de chaque non-terminal est fini, et donc tel est aussi le nombre de relations binaires que l'on peut construire sur ces attributs.

Théorème 3 [Knu68, *corr.*] *Les ensembles complets de graphes de dépendance synthétisés et hérités de tous les non-terminaux d'une grammaire attribuée peuvent être calculés statiquement.*

La construction des ensembles complets de GDS est effectuée par l'algorithme suivant :

Algorithme 1 [Knu68, *corr.*]

Donnée *Une grammaire attribuée*

Résultat *L'ensemble complet des graphes de dépendance synthétisés $GDS(X)$ pour chaque non-terminal X*

Méthode *Exécuter la procédure de la figure B.8*

En effet il est facile de voir que, de même que dans le théorème 1, pour tout arbre t et tout nœud $u \in t$ avec $P(u) = p$, on a :

$$D(t/u) = D(u) \cup \bigcup_{1 \leq i \leq n_p} D(t/u.i)$$

Puisque, par définition, la projection de $D(t/u.i)$ sur $u.i$ est un des GDS de X_i, la construction exprimée dans la formule ci-dessus est la même que la construction du graphe de dépendance

1. $\forall X \in N,\ GDS(X) \leftarrow \emptyset$;
2. **répéter** convergence $\leftarrow$ vrai ;
3.   **pour chaque** $X \in N$ **faire**
4.   **pour chaque** $p \in P$ avec LHS$(p) = X$, $p\ :\ X \rightarrow X_1 X_2 \cdots X_{n_p}$ **faire**
5.   **pour chaque** $\gamma_i \in GDS(X_i), 1 \leq i \leq n_p$ **faire**
6. $\Gamma \leftarrow D(p)[\emptyset, \gamma_1, \ldots, \gamma_{n_p}]$;
7. $\gamma \leftarrow \Gamma|_0$;
8.   **si** $\gamma \notin GDS(X)$ **alors**
9. $GDS(X) \leftarrow GDS(X) \cup \{\gamma\}$;
10.   convergence $\leftarrow$ faux

                finsi

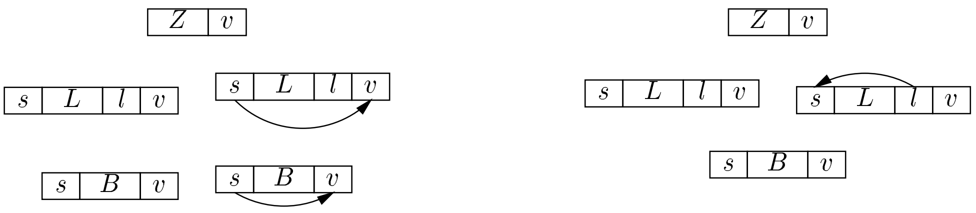
            finpour

        finpour

    finpour

11. **jusqu'à** convergence.

FIG. B.8 –. *Procédure pour construire l'ensemble des graphes de dépendance synthétisés $GDS(X)$ de tous les non-terminaux X*



(a) Graphes de dépendance synthétisés   (b) Graphes de dépendance hérités

FIG. B.9 –. *Graphes de dépendance synthétisés et hérités pour la GA de la figure A.1*

---

augmenté à la ligne 6 de l'algorithme. La preuve par induction sur la hauteur de t/u est alors évidente.

La construction de l'ensemble complet des graphes de dépendance hérités, une fois que l'ensemble des GD synthétisés est connu, est effectuée par un algorithme similaire à celui ci-dessus, sauf qu'il fonctionne "du haut vers le bas" dans la grammaire, plutôt que de manière ascendante, et que le pas de base est

1. la construction du graphe de dépendance augmenté suivant :

$$\Gamma = D(p)[\eta_0, \gamma_1, \ldots, \gamma_{i-1}, \emptyset, \gamma_{i+1}, \ldots, \gamma_{n_p}]$$

avec $\eta_0 \in GDH(X_0)$ et $\gamma_j \in GDS(X_j), 1 \leq j \leq n_p$

2. sa projection sur X_i pour obtenir un (nouveau) graphe de dépendance hérité pour X_i.

Notons donc que les dépendances synthétisés et héritées ne sont pas tout à fait symétriques les unes des autres.

L'ensemble complet des graphes de dépendance synthétisés et hérités pour notre GA exemple est présenté dans la figure B.9.

Il est maintenant clair qu'un unique graphe de dépendance synthétisé (resp. hérité) *résume* les dépendances réelles dans tout un ensemble de graphes de dépendance inférieurs (resp. supérieurs). Ceci nous donne deux nouveaux couples à ajouter à notre table des correspondances statiques/dynamiques :

vue statique	*vue dynamique*
plusieurs	beaucoup
graphe de dépendance synthétisé	graphe de dépendance inférieur
graphe de dépendance hérité	graphe de dépendance supérieur

Les graphes de dépendance synthétisés et hérités ont de nombreux usages [FrJ91], mais concentrons-nous sur le test de non-circularité.

B.2.4 Un test de non-circularité statique

Considérons un graphe de dépendance complet circulaire et un cycle particulier dans ce graphe. Ce cycle est soit contenu tout entier dans un unique graphe de dépendance de production, soit réparti sur plusieurs productions. Le premier cas est facilement détecté sur la GA elle-même et constitue une erreur évidente ; notons que ceci ne peut pas arriver dans une GA en forme normale. Dans l'autre cas, d'après le théorème 1, il y a toujours un nœud dans l'arbre qui est le "pic" de ce cycle et qui est étiqueté par une production qui contient un arc d'un certain attribut synthétisé d'un certain non-terminal en partie droite à un certain attribut hérité d'un autre (ou du même) non-terminal en partie droite, tel que le cycle est brisé si l'on en enlève cette production et les dépendances associées.

Maintenant, le reste du cycle peut être résumé par un certain graphe de dépendance synthétisé pour chacun des non-terminaux dans la partie droite de cette production particulière. Ainsi, si l'on construit le graphe de cette production augmenté par ces graphes de dépendance synthétisés, le cycle réapparaîtra inévitablement.

On en déduit donc qu'une GA donnée est non circulaire si et seulement si tout graphe augmenté construit au cours de l'algorithme ci-dessus est non circulaire. Ceci peut être vérifié facilement en insérant le test suivant entre les lignes 6 et 7 :

$6'$. **si** Γ est circulaire **alors** la GA est circulaire **finsi** ;

Propriété 4 [Knu68, *corr.*] *L'algorithme 1 est exponentiel en temps par rapport à la "taille" de la grammaire attribuée d'entrée, où la taille d'une GA est définie comme n'importe quelle combinaison polynomiale des nombres de non-terminaux, de productions, de symboles en partie droite des productions et d'attributs.*

Théorème 4 [JOR75] *Le problème de déterminer si une GA donnée est circulaire a une complexité en temps qui est intrinsèquement exponentielle en la taille de la GA.*

Bien que l'algorithme 1 soit théoriquement intrinsèquement exponentiel, il est tout à fait réalisable en pratique si on l'implante avec assez de soin (cf. la section 2.5).

Le lecteur intéressé vérifiera facilement que notre GA exemple est effectivement non circulaire.

B.2.5 Conclusion

On a utilisé la propriété que tous les GDS peuvent être calculés statiquement pour vérifier statiquement une propriété dynamique d'une GA. Beaucoup d'autres propriétés peuvent être vérifiées statiquement en construisant d'autres sortes de graphes de dépendance de production augmentés, c'est-à-dire des graphes de dépendance de production augmentés par d'autres sortes de “graphes de dépendance de non-terminal”, ou certaines abstractions, plus simples, de ces graphes. La thèse que je soutiens dans ce mémoire est que ceci constitue vraiment le cœur de l'étude des GA : l'analyse statique est omnipotente. En outre, ce genre d'analyse est faisable en pratique et donne des résultats très précis, voire exacts comme dans le test de non-circularité, parce que le formalisme des GA est simple, en tous cas beaucoup plus que la plupart des autres.

Annexe C

English Summary

This summary is presently restricted to the sections in chapter 3 which describe yet unpublished work. For the sequel, the reader is invited to refer to the corresponding publications of mine, which are generally in English.

The structure of this section reflects the structure of the whole document.

C.1 Introduction

C.2 Static Analysis of AGs to Improve Attribute Evaluators

C.3 Static Analysis of AGs to Improve Easiness of Use and Expressive Power

C.3.1 What an application is for Fnc-2

C.3.2 The AG-description language Olga

Generalities

List nodes and plane shift

In Fnc-2, the trees that are the input and output data of the various passes, in particular AGs, are *attributed abstract trees*, described in a special language called `asx`. The most important difference between `asx` and classical BNF is that the former provides for *list operators*, i.e. nodes with a variable number of sons. This feature is very nice to use, but classical AG-manipulation algorithms don't provide for it. So we need a way to associate the list-node notion with some "equivalent" notion in the classical AG framework.

This problem is an instance of what I call the *plane shift* method, illustrated in Fig. 3.3. Let S be the specification of some computation, expressed in a formalism that is close, but not identical, to classical AGs. We want to build as automatically and as easily as possible an implementation I of S. The *plane shift* then consists in defining, on one hand, the *reduction* of S into a normal AG G and, on the other hand, the *interpretation* that allows to build I from the evaluator E constructed from G by the classical techniques, so that, of course, the diagram commutes!

As for list nodes, we have imposed some reasonable constraints on the shape of semantic rules. Then, the reduction step consists in building the corresponding dependency schemes for 0, 1, 2 and 3 sons, and the interpretation step consists in putting a loop around the actions dealing with the middle son in the 3-son production.

Automatic construction of missing semantic rules

C.3.3 Generic AGs

Here begins the discussion of topics on which we are working presently and which have not been published yet. These topics are "heavy" extensions of the notion of AGs, aimed at increasing their easiness of use and expressive power.

Generic AGs are to classical AGs what generic modules are to simple modules, e.g. in Ada: they capture the essence of an interesting algorithm, expressed in an AG as attributes and semantic rules, in such a form that it's easy to reuse this algorithm on other data types, i.e. syntaxes.

A generic AG (GAG) is a normal AG, except that the syntax is reduced to the structures (non-terminals and productions) that play an important "semantic" role in the algorithm. For instance, if the problem at hand is name analysis, these important structures are blocks, declarations of an identifier and uses of an identifier.

Instantiating a generic AG on a "naked" syntax should result in a new AG in which the computations described by the former are "transported" onto the latter. So the user must identify in his syntax which non-terminals and (possibly) productions play the above-described important semantic role in the algorithm and, hence, what are their counterparts in the reduced syntax of the generic AG. These important non-terminals are called *key* non-terminals; with each non-terminal in the GAG is associated a set of key NTs in the syntax to be instantiated. These sets must be disjoint but they're not supposed to cover the whole syntax (there remains non-key NTs).

Given a GAG, a syntax to be instantiated and an association of key NTs in the latter with each NT in the former, the instantiation works as follows. The basic idea is that, with each tree in the syntax, we would like to associate a tree of the GAG, on which the computations in the GAG can be performed and transported back to the original tree. This association is performed by partitioning the original tree into *regions*, i.e. connected parts which contain no key NT except on their border (root and leaves). With each such region is then associated the smallest tree pattern in the GAG with the same border, up to the given key-NT association. If this can be done unambiguously, the computations performed on each GAG pattern can be transported back to the corresponding region quite easily.

But we want to do this statically, whereas the above description is quite dynamic. The idea is to (try to) construct an AG that, from a tree in the syntax, constructs the corresponding tree in the GAG. Then, transporting the computations (semantic rules) back to the original tree reduces to meta-composing this translation AG with the given GAG and identify, in the resulting AG, the output attributes of the GAG, which can be found attached to the key NTs.

We are presently working on the *static* construction of such translation AGs: we have devised a number of algorithms therefore and identified a number of conditions that would make them fail. We intend to publish these results soon and acquire experience with this new notion.

Related work can be found in [FMY92].

C.3.4 Dynamic AGs

Dynamic AGs are another extension to classical AGs that aims at increasing their expressive power. Our motivation here originates in the observation that a visit-sequence-based evaluator is very easy to implement in any programming language that provides recursion. However such an evaluator is far from exploiting the whole power of these languages: no conditional statements that could influence the control flow, only strict recursion (as opposed to recursion on the same input data structure but with different other parameters).

So we'd like to incorporate these two notions — control flow by means of conditions and non-strict recursion — in an AG itself. As shown in Fig. 3.4, this is achieved with two constructs:

1. *semantic conditions* determining the use of one set of semantic rules among others, and
2. multiple *views* of a single physical production as different virtual structures, allowing a dynamic number of passes on the same subtree.

Presently our works on this topic are rather preliminary: we need an implementation and practical experience. As for the former, we expect no problem because dynamic AGs are another example of the notion of *plane shift* (see section 3.2.2). Indeed, each dynamic production gives rise to a finite number of virtual, well-formed, classical productions; for instance the dynamic production of Fig. 3.4 corresponds to the two dependency schemes depicted in Fig. 3.5. This implements the reduction step. The interpretation step is as simple: gather all the visit-sequences built from these virtual productions into a single one, attached to the physical production, by inserting adequate conditional statements and replacing references to virtual nodes (such as W in the above example) by physical nodes. Note that, to make this plane shift correct, *all* the attributes must be stored out of the tree, but this is no problem, as shown in section 2.2.

We believe, although we cannot prove this yet, that dynamic AGs would have the same expressive power as denotational semantics while benefiting from the "software engineering" advantages of AGs. In particular, they subsume circular AGs in a more elegant framework (in our opinion, at least!). Finally, combining generic AGs and dynamic AGs would lead to interesting applications, e.g. a data flow analyzers generator as advocated by Kildall long ago [Kil73].

C.3.5 Future work

Another problem I'd like to find the time to work on is the following. In practical AGs, numerous semantic rules express the *updating* of some data structure with new information (see e.g. the symbol table in Fig. 3.6). The update operations are described in OLGA by side-effect-free functions (see "insert" in Fig. 3.7(a)).

Besides, memory management optimizations, presented in section 2.2, often turn "update" semantic rules of the form $X.a := f(Y.b, \dots)$ into assignments of the form $v := f(v, \dots)$. In such cases, instead of using the applicative function f, we would like to use some equivalent but destructive procedure p (e.g. procedure "insert" in Fig. 3.7(b)) and replace the assignment with $p(v, \dots)$. This would certainly improve the efficiency of the resulting program, as shown by our experience with OLGA.

So I'm interested in automatically deriving such destructive procedures from side-effect free functions, described in OLGA or other languages. It seems that this problem has not attracted much attention, but it's certainly of use for FNC-2. It might be the case that expressing these functions as AGs eases the problem.

C.3.6 Conclusion

C.4 Conclusion

C.A An Example of Stepwise Development of an AG

C.B An Example of Exact Static Analysis: the Circularity Test

Annexe D

Bibliographie

[**AFN82**] AFNOR, “Le langage de programmation Pascal,” norme ISO 7185, Paris, 1982.

[**AkS91**] R. op den Akker & E. Sluiman, “Storage Allocation for Attribute Evaluators Using Stack and Queues,” in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 140–150.

[**Alb91**] H. Alblas, “Attribute Evaluation Methods,” in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 48–113.

[**Boc76**] G. V. Bochmann, “Semantic Evaluation from Left to Right,” *Comm. ACM* **19** (Feb. 1976), 55–62.

[**BDJ83**] P. Boullier, P. Deschamp & M. Jourdan, “Spécification et réalisation d'un traducteur Pascal-Ada,” in *Journées ADA de l'AFCET*, Jacques André & A. Strohmeier, eds., Dec. 1983, 9–29.

[**BDJ84**] ———, “Application d'outils de haut niveau à la réalisation d'un traducteur automatique Pascal-Ada,” in *2ème Coll. AFCET de Génie Logiciel*, June 1984, 375–385.

[**BoJ87**] P. Boullier & M. Jourdan, “A New Error Repair and Recovery Scheme for Lexical and Syntactic Analysis,” *Sci. Comput. Programming* **9** (Dec. 1987), 271–286.

[**CoF82**] B. Courcelle & P. Franchi-Zannettacci, “Attribute Grammars and Recursive Program Schemes,” *Theoret. Comput. Sci.* **17** (1982), 163–191 et 235–257.

[**CoC77**] P. Cousot & R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximate fixpoints,” in *4th ACM Symp. on Principles of Progr. Languages*, Jan. 1977, 239–252.

[**Der84**] P. Deransart, “Validation des grammaires d'attributs,” Univ. de Bordeaux I, thèse d'État, Oct. 1984.

[**DJL88**] P. Deransart, M. Jourdan & B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. #**323**, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[**DJL84**] ———, “Speeding up Circularity Tests for Attribute Grammars,” *Acta Inform.* **21** (Dec. 1984), 375–391.

[**DMR90**] A. Despland, M. Mazaud & R. Rakotozafy, “Pagode: A Back-end Generator using Attributed Abstract Syntaxes and Term Rewritings,” in *Compiler Compilers '90*, Dieter Hammer, ed., Lect. Notes in Comp. Sci. #**477**, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1990, 86–105.

[**DuC90**] G. D. P. Dueck & G. V. Cormack, “Modular Attribute Grammars,” *Comput. J.* **33** (1990), 164–172.

[**EnF82**] J. Engelfriet & G. Filé, “Simple Multi-Visit Attribute Grammars,” *J. Comput. System Sci.* **24** (June 1982), 283–314.

[**EnJ90**] J. Engelfriet & W. de Jong, "Attribute Storage Optimization by Stacks," *Acta Inform.* **27** (1990), 567–581.

[**Fan72**] I. Fang, "FOLDS, a Declarative Formal Language Definition System," Comp. Sc. Dept., Stanford Univ., PhD thesis, report STAN-CS-72-329, Dec. 1972.

[**FMY92**] R. Farrow, T. J. Marlowe & D. M. Yellin, "Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation," in *19th ACM Symp. on Principles of Progr. Languages*, Jan. 1992, 223–234.

[**FaY86**] R. Farrow & D. M. Yellin, "A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators," *Acta Inform.* **23** (1986), 393–427.

[**Fil83**] G. Filé, "Interpretation and Reduction of Attribute Grammars," *Acta Inform.* **19** (1983), 115–150.

[**Fil87**] ———, "Classical and Incremental Attribute Evaluation by Means of Recursive Procedures," *Theoret. Comput. Sci.* **53** (Jan. 1987), 25–65.

[**Fra82**] P. Franchi-Zannettacci, "Attributs sémantiques et schémas de programmes," Univ. de Bordeaux I, thèse d'État, Mar. 1982.

[**FrJ91**] P. Franchi-Zannettacci & M. Jourdan, "Conception des environnements de programmation, première partie : Attribute Grammars," in *Actes de l'École Jeunes Chercheurs du GRECO de Programmation*, Apr. 1991, 71 pages, en anglais.

[**GaG84**] H. Ganzinger & R. Giegerich, "Attribute Coupled Grammars," *ACM SIGPLAN Notices* **19** (June 1984), 157–170.

[**HoO82**] C. M. Hoffmann & M. J. O'Donnell, "Pattern Matching in Trees," *J. ACM* **29** (Jan. 1982), 68–95.

[**JOR75**] M. Jazayeri, W. F. Ogden & W. C. Rounds, "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars," *Comm. ACM* **18** (Dec. 1975), 679–706.

[**JoM81**] N. D. Jones & S. S. Muchnick, eds., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[**Jou84a**] M. Jourdan, "An Optimal-time Recursive Evaluator for Attribute Grammars," in *6th Int. Symp. on Programming*, M. Paul & Bernard Robinet, eds., Lect. Notes in Comp. Sci. #**167**, Springer-Verlag, New York–Heidelberg–Berlin, Apr. 1984, 167–178.

[**Jou84b**] ———, "Strongly Non-Circular Attribute Grammars and Their Recursive Evaluation," *ACM SIGPLAN Notices* **19** (June 1984), 81–93.

[**Jou91**] ———, "A Survey of Parallel Evaluation Methods," in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 234–255.

[**Jou84**] ———, "Les grammaires attribuées: implantation, applications, optimisations," Univ. Paris VII, thèse de docteur-ingénieur, May 1984.

[**JLP90**] M. Jourdan, C. Le Bellec & D. Parigot, "The Olga Attribute Grammar Description Language: Design, Implementation and Evaluation," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 222–237.

[**JoP92**] M. Jourdan & D. Parigot, *The FNC-2 System User's Guide and Reference Manual* version 1.8, INRIA, Rocquencourt, Feb. 1992, Ce manuel est périodiquement remis à jour.

[**JoP91**] ———, "Internals and Externals of the Fnc-2 Attribute Grammar System," in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 485–504.

[**JoP89**] ———, "The FNC-2 System: Advances in Attribute Grammar Technology," in *Procs. of the Soviet-French Symposium Informatika '89*, O. M. Tammepuu, ed., May 1989, 94–118, Voir aussi : rapport RR-834, INRIA, Rocquencourt, Apr. 1988.

[JoP90a] ———, "Techniques for Improving Grammar Flow Analysis," in *European Symp. on Programming (ESOP '90)*, Neil Jones, ed., Lect. Notes in Comp. Sci. #**432**, Springer-Verlag, New York–Heidelberg–Berlin, May 1990, 240–255.

[JoP90b] ———, "Application Development with the FNC-2 Attribute Grammar System," in *Compiler Compilers '90*, Dieter Hammer, ed., Lect. Notes in Comp. Sci. #**477**, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1990.

[JPJ90] M. Jourdan, D. Parigot, C. Julié, O. Durin & C. Le Bellec, "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System," *ACM SIGPLAN Notices* **25** (June 1990), 209–222.

[Jul89] C. Julié, "Optimisation de l'espace mémoire pour l'évaluation des grammaires attribuées," Dépt. d'Informatique, Univ. d'Orléans, thèse, Sept. 1989.

[JuP90] C. Julié & D. Parigot, "Space Optimization in the FNC-2 Attribute Grammar System," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 29–45.

[JuD84] R. K. Jullig & F. De Remer, "Regular Right-part Attribute Grammars," *ACM SIGPLAN Notices* **19** (June 1984), 171–189.

[KLM83] G. Kahn, B. Lang, B. Mélèse & É. Marcos, "Metal: a Formalism to Specify Formalisms," *Sci. Comput. Programming* **3** (1983), 151–188.

[Kas80] U. Kastens, "Ordered Attribute Grammars," *Acta Inform.* **13** (1980), 229–256.

[Kas84] ———, "The GAG-System—A Tool for Compiler Construction," in *Methods and Tools for Compiler Construction*, Bernard Lorho, ed., Cambridge Univ. Press, New York, NY, 1984, 165–182.

[Kas91] ———, "Implementation of Visit-Oriented Attribute Evaluators," in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 114–139.

[Kas87] ———, "Lifetime Analysis for Attributes," *Acta Inform.* **24** (Nov. 1987), 633–652.

[KHZ82] U. Kastens, B. Hutt & E. Zimmermann, *GAG: A Practical Compiler Generator*, Lect. Notes in Comp. Sci. #**141**, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[KeW76] K. Kennedy & S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars," in *3rd ACM Symp. on Principles of Progr. Languages*, Jan. 1976, 32–49.

[Kil73] G. Kildall, "A unified approach to global program optimization," in *1st ACM Symp. on Principles of Progr. Languages*, Jan. 1973, 194–206.

[Knu68] D. E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2** (June 1968), 127–145, *Corrigendum*: *Math. Systems Theory* **5**, 1 (Mar. 1971), 95–96.

[Knu90] ———, "The Genesis of Attribute Grammars," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 1–12.

[Le B89] C. Le Bellec, "Spécification de règles sémantiques manquantes," Dépt. d'Informatique, Univ. d'Orléans, rapport de DEA, Sept. 1989.

[LMW88] P. Lipps, U. Möncke & R. Wilhelm, "OPTRAN: A Language/System for the Specification of Program Transformations—System Overview and Experiences," in *Compiler Compilers and High Speed Compilation*, Dieter Hammer, ed., Lect. Notes in Comp. Sci. #**371**, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1988, 52–65.

[Lor77] B. Lorho, "Semantic Attributes Processing in the System DELTA," in *Methods of Algorithmic Language Implementation*, A. Ershov & Cornelius H. A. Koster., eds., Lect. Notes in Comp. Sci. #**47**, Springer-Verlag, New York–Heidelberg–Berlin, 1977, 21–40.

[LoP75] B. Lorho & C. Pair, "Algorithms for Checking Consistency of Attribute Grammars," in *Proving and Improving Programs*, Gérard Huet & Gilles Kahn, eds., INRIA, Rocquencourt, July 1975, 29–54.

[**Mar90**] B. Marmol, "Évaluateurs d'attributs parallèles sur multi-processeurs à mémoire partagée," Univ. d'Orléans, rapport de DEA, Sept. 1990.

[**MöW91**] U. Möncke & R. Wilhelm, "Grammar Flow Analysis," in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 151–186.

[**Par88**] D. Parigot, "Transformation, évaluation incrémentale et optimisations des grammaires attribuées : le système FNC-2," Univ. de Paris-Sud, thèse, Orsay, May 1988.

[**Rep84**] T. Reps, *Generating Language-based Environments*, MIT Press, Cambridge, MA, 1984.

[**Rii83**] H. Riis-Nielson, "Computation Sequences: A Way to Characterize Subclasses of Attribute Grammars," *Acta Inform.* **19** (1983), 255–268.

[**Rou92**] G. Roussel, "Une transformation de grammaire attribuée pour éliminer les règles de copie inutiles," INRIA, rapport de recherche à paraître, Rocquencourt, 1992.

[**Sou91**] A. Souah, "Contribution à la sémantique déclarative des systèmes de transformation d'arbres attribués," Univ. d'Orléans, thèse, Oct. 1991.

[**SwV91**] S. D. Swierstra & H. Vogt, "Higher Order Attribute Grammars," in *Attribute Grammars, Applications and Systems*, Henk Alblas & Bořivoj Melichar, eds., Lect. Notes in Comp. Sci. #**545**, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 256–296.

[**VoM82**] A. O. Vooglaid & M. B. Mériste, "Abstract Attribute Grammars," *Progr. and Computer Software* **8** (Sept. 1982), 242–251.

Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées

Résumé

Les grammaires attribuées (GA) sont reconnues depuis longtemps comme une méthode de spécification très bien adaptée à tous les problèmes dont les données d'entrée ont une composante syntaxique forte. Cependant, elles ne sont quasiment pas utilisées comme outil de programmation pour développer des applications industrielles. Cela est dû essentiellement à deux raisons :

1. les implantations des GA (évaluateurs d'attributs) ne sont pas encore assez efficaces pour concurrencer les programmes écrits à la main ;

2. le formalisme classique des GA manque quelque peu de facilité d'utilisation, en particulier pour la programmation en équipe, et de puissance d'expression.

Notre thèse est que le seul moyen de résoudre en même temps ces deux problèmes, a priori contradictoires, passe par l'analyse statique de GA. Nous montrerons que le formalisme des GA est en effet suffisamment simple pour qu'une analyse statique relativement peu sophistiquée puisse fournir des résultats exacts sur le comportement dynamique d'un évaluateur.

D'autre part, nous présenterons un certain nombre des travaux sur les GA que nous avons menés ces dernières années et qui accréditent cette thèse en répondant à l'un ou l'autre des points ci-dessus. En ce qui concerne le premier, citons : transformation FNC $\rightarrow$ l-ordonnée améliorée ; optimisations mémoire ; évaluation parallèle ; méta-composition des GA ; optimisations des algorithmes d'analyse. Quant au second, citons : construction modulaire d'une application ; le langage OLGA ; GA génériques ; GA dynamiques. Ces travaux sont le fondement du système FNC-2, qui est un des meilleurs du monde pour la mise en œuvre des GA.

La valeur du système FNC-2 commence d'ailleurs à être reconnue jusque chez les industriels, en particulier nos partenaires du Projet ESPRIT "COMPARE".