N° d'ordre : 4184

THÈSE

présentée à

l'Université Louis Pasteur de Strasbourg

par

PHILIPPE GERNER

pour obtenir le grade de

Docteur de l'Université Louis Pasteur
Mention SCIENCES
Spécialité INFORMATIQUE

La sémantique des directives au compilateur : application au parallélisme de données

soutenue le 10 décembre 2002 devant la commission d'examen composée de :

M. Pascal Gribomont, Rapporteur,
Professeur à l'Université de Liège

M. Gaétan Hains, Rapporteur,
Professeur à l'Université d'Orléans

M. Jean-François Dufourd, Rapporteur interne,
Professeur à l'Université Louis Pasteur de Strasbourg

M. Éric Violard, Membre invité,
Maître de Conférences à l'Université Louis Pasteur de Strasbourg

M. Guy-René Perrin, Directeur de Thèse,
Professeur à l'Université Louis Pasteur de Strasbourg

Résumé

Les ordinateurs parallèles, ou multiprocesseurs, permettent de diminuer le temps d'exécution des programmes, car ils sont capables d'exécuter plusieurs instructions simultanément. Leur programmation n'est pas facile, en raison de la complexité du schéma d'exécution d'un programme. Deux approches sont utilisées en pratique :

- La programmation parallèle dite «de bas niveau», comme avec MPI : le programmeur prend en charge la distribution des calculs sur les processeurs et décrit explicitement les communications entre les processeurs.
- La programmation parallèle dite «de haut niveau» : un langage de haut niveau comme Fortran est utilisé, comme avec OpenMP ou HPF (High Performance Fortran), et la distribution des calculs et la gestion des communications sont laissées à la charge du compilateur. Le programmeur peut alors aider ce dernier, en rajoutant à son code des indications utiles, sous la forme de directives.

Le problème avec la première approche est la difficulté de la programmation. Le problème avec la seconde approche est qu'actuellement, les compilateurs ne sont pas tenus de prendre en compte les directives : en conséquence il est difficile au programmeur d'estimer l'efficacité de ses directives.

La présente thèse s'intéresse à ce dernier problème. La solution proposée est une rédaction claire, dans le cadre de la documentation du langage, d'un contrat entre le programmeur et le compilateur concernant la sémantique du langage. Pour cela, nous distinguons entre une sémantique officielle et une sémantique effective. La méthode de rédaction du contrat est exposée, de façon formelle, à l'aide de la logique de réécriture. La sémantique des directives est donnée par une notion de préférence entre modèles. Cette méthode est ensuite appliquée à l'étude des directives de placement de HPF, et pour proposer un formalisme simple pour le parallélisme de données. Ce formalisme est ensuite utilisé pour expliquer la sémantique du langage PEI.

Mots clés

Programmation parallèle — directives au compilateur — sémantique des langages de programmation — normes — logique de réécriture.

Remerciements

Je remercie Pascal Gribomont, professeur à l'université de Liège, et Gaétan Hains, professeur à l'université d'Orléans, pour avoir accepté de rapporter cette thèse.
Je remercie Jean-François Dufourd, professeur à Strasbourg, pour avoir accepté de rapporter cette thèse. C'est lui qui m'a introduit, en licence et de nouveau en DEA, aux spécifications algébriques, qui m'ont suffisamment intriguées pour que j'aie envie d'expérimenter davantage avec elles.

Merci à Michel Eytan, professeur à l'université Marc Bloch de Strasbourg, pour m'avoir introduit dans son cours de DEA à la théorie des catégories.

Je remercie Guy-René Perrin pour m'avoir permis d'effectuer ce travail. Lorsque mes méthodes d'analyse effectives différaient de mes méthodes officielles, il a su m'amener à le remarquer. Son attention à mon discours de sémantique et de contexte m'a encouragé à poursuivre dans la voie qui m'intéressait.

Je ne saurais trop remercier Éric Violard, mon encadrant de tous les jours. Il a su me laisser choisir une voie de recherche qui me passionne. Je le remercie en particulier pour sa constante insistance pour que je produise des exemples concrets. Il m'a donné l'exemple, au cours de ces années, de ce que devait être une démarche scientifique sérieuse.

Pascal Schreck a été disponible pour des discussions au sujet des spécifications algébriques. Il a également lu des versions préliminaires de bon nombre de mes travaux, et ses commentaires ont été très précieux. Il a su m'aider à croire en la valeur de mon approche : merci Pascal !

Je ne saurais trop remercier tous les membres du laboratoire LSIIT-ICPS, pour l'ambiance chaleureuse. Merci à Michel Salomon, pour nos discussions sur les directives de HPF. Merci également à Mme Broger, pour les biscuits !

Je remercie tous mes amis, qui m'ont encouragé au cours de ce travail.

Ce document a été préparé avec le logiciel LaTeX. Mes remerciements donc au Professeur Donald Knuth. Merci également à Paul Taylor pour son package `diagrams`, qui a été utilisé pour produire les figures.

Table des matières

Introduction

Les ordinateurs parallèles, ou multiprocesseurs, mettent en oeuvre plusieurs processeurs, de façon à pouvoir exécuter plusieurs instructions simultanément. Ils sont utilisés dans :

- Le calcul dit *scientifique*, tel que les calculs balistiques, les simulations d'explosions nucléaires, l'étude de la résistance des matériaux, les simulations d'écoulement d'air ou de fluides, l'analyse structurelle de molécules complexes, le séquençage du génome ;
- Le *traitement massif de données*, à travers des bases de données importantes.

La difficulté de la programmation des ordinateurs parallèles est essentiellement, à notre sens, d'ordre cognitif. En effet, l'exécution des programmes sur les machines parallèles diffère de l'exécution conventionnelle, l'exécution séquentielle, plus facile pour nous à appréhender, en supposant de concevoir l'exécution simultanée de plusieurs dizaines ou milleirs d'instructions interagissant. En conséquence, un langage de programmation de haut niveau destiné à ces machines doit s'efforcer de minimiser le degré de parallélisme que doit gérer le programmeur. Mais il reste ensuite à traduire un programme ainsi écrit avec un parallélisme minimal, dans sa version bas-niveau, où le parallélisme doit être complètement explicité : c'est le travail du compilateur. Or celui qui conçoit et implante le compilateur est, en tant qu'être humain, soumis au mêmes limites cognitives que le programmeur : il n'est donc pas possible de lui laisser imaginer tout le parallélisme du programme. En conséquence, il s'agit actuellement, pour disposer d'un langage parallèle de haut niveau, que celui-ci établisse un bon compromis entre la facilité d'écriture pour le programmeur, et la facilité d'implantation pour l'implanteur du compilateur.

La pratique actuelle de la programmation parallèle montre l'état d'avancement dans la recherche de ce compromis entre aisance à la conception et à la programmation, et efficacité à l'exécution :

- La majorité des programmes parallèles sont écrits en MPI [43], ce qui équivaut à du bas-niveau. C'est-à-dire qu'il est jugé plus simple de donner le maximum à faire au programmeur.
- Le reste des programmes est écrit dans des langages séquentiels, adaptés et auxquels est adjoint un langage de *directives au compilateur*, pour que le programmeur puisse indiquer au compilateur des façons possibles de traduire son programme pour une machine parallèle. Ce sont le langage HPF [55, 62], et les directives d'OpenMP [1].

Ce dernier type de programmation (les langages à directives) n'est pas encore très utilisé, faute de savoir préciser comment obtenir des programmes vraiment efficaces à partir de ces langages. Pourtant, la pratique actuelle de la programmation parallèle concerne très majoritairement la parallélisation de codes séquentiels existants, et cette activité est nettement plus facile en OpenMP ou HPF qu'en MPI. Il existe par ailleurs des *compilateurs paralléliseurs* [22], qui parallélisent de façon automatique des programmes séquentiels. Mais ces traducteurs automatiques sont limités par ce qu'exprime le code séquentiel : ils ne peuvent avoir la même «vue globale» du programme qu'a pu avoir son rédacteur, ou que peut recouvrer le programmeur chargé de la parallélisation de ce programme.

Dans la présente thèse, nous nous intéressons à la programmation parallèle de haut niveau, et en particulier à la sémantique des directives au compilateur. En effet, d'une sémantique claire pour ces directives dépend leur utilisation efficace, et de leur utilisation efficace dépend le succès des langages parallèles de haut niveau actuels comme HPF, face à des langages de bas-niveau comme MPI. La sémantique des directives sera éclairée sous l'angle de la «coopération programmeur-compilateur» sur laquelle elle repose.

Dans une première partie, nous présentons ce que nous appelons la «coopération entre programmeur et compilateur» dans les langages actuels. Un cadre où cette coopération est importante est le parallélisme de données : ce mode de programmation est ici décrit par le biais de cette coopération. Puis, comme le chapitre suivant utilise une méthode formelle pour la description d'idées sémantiques, un état de l'art des méthodes formelles de description de la sémantique de langages de programmation est donné, afin de situer notre approche.

Dans la deuxième partie, nous montrons comment formuler le contrat entre le programmeur et le compilateur sur la signification du langage de programmation utilisé. La sémantique du langage de programmation telle qu'elle est appréhendée par le programmeur (sémantique dite «officielle»), et la sémantique «effective» que constitue l'exécution du programme, sont distinguées : cela permet d'expliciter la relation entre ces deux sémantiques, et donc de rédiger les termes du contrat. La *logique de réécriture*, c'est-à-dire le cadre des spécifications algébriques auxquelles est rajoutée la notion de transition, est utilisée pour faire la description des sémantiques officielle et effective du langage.

Dans une troisième partie, nous appliquons les notions introduites à la formulation de la sémantique des directives, avec pour cas d'étude les directives de placement dans le parallélisme de données. Tout d'abord, nous formulons la sémantique des directives pour le langage HPF. Puis nous proposons un formalisme simple pour le parallélisme de données, que nous appliquons à une explication et une discussion de la sémantique du langage académique à parallélisme de données Pei.

Chapitre 1

La coopération programmeur-compilateur

Cette thèse traite de la structure que l'on peut donner à la sémantique des langages de haut niveau lorsqu'est pris en compte le fait que le programmeur intègre la connaissance qu'il a du compilateur utilisé. À notre connaissance, ce problème n'a pas encore été étudié attentivement. Cela pourrait être dû à l'existence de deux communautés distinctes travaillant sur les langages de programmation : une communauté travaillant sur la sémantique des langages de programmation, et l'autre sur les techniques de compilation.

Dans le présent chapitre nous précisons tout d'abord ce que nous entendons par «coopération programmeur-compilateur», et présentons l'état de cette coopération dans la pratique actuelle de la programmation.

Puis suivent deux sections qui servent d'état de l'art pour les chapitres suivants. Tout d'abord, le chapitre 3 traitant de la coopération programmeur-compilateur dans le parallélisme de données, nous proposons un état de l'art sur le parallélisme de données. Puis, utilisant dans les chapitres 2 et 3 une méthode formelle pour exposer la sémantique d'un langage, nous présentons un état de l'art sur les méthodes formelles qui sont utilisées pour la description de langages de programmation.

1.1 La coopération programmeur-compilateur

1.1.1 L'efficacité d'un programme

La coopération entre programmeur et compilateur définit la contribution de chacun à l'écriture d'un programme dit «efficace».

Qu'est-ce qu'un programme efficace ? La notion d'efficacité est utilisée ici au sens du programmeur, c'est-à-dire en termes d'utilisation des ressources de la machine qui l'exécute (volume de données utiles, temps d'utilisation du processeur, ...). L'utilisateur peut en effet considérer que le programme qui lui est fourni est *satisfaisant*, sans que le programmeur considère ce programme comme *efficace*, en termes d'utilisation des ressources de la machine. Cette non-efficacité du programme est relative : un programme ne peut pas être considéré comme efficace ou non-efficace dans l'absolu. Il ne l'est que compte tenu d'une comparaison (même implicite) avec un autre programme qui résoudrait le même problème. C'est-à-dire que le jugement de l'efficacité repose sur la connaissance qu'a le programmeur de ce qui est possible en termes d'implantation.

La *théorie de la complexité* fournit un moyen d'estimer l'efficacité en termes de mémoire ou de nombre d'instructions requis à l'exécution d'un algorithme. Mais ce type de calcul suppose la prise en compte des facteurs pertinents, en termes de coût mémoire ou en temps, dans le langage de programmation (ou pseudo-langage) utilisé. La connaissance des facteurs pertinents suppose la connaissance de la façon dont le code est compilé et exécuté—i.e., la relation entre le code et le mécanisme qui se déroulera au niveau du matériel. En outre, la théorie de la complexité permet d'étudier l'efficacité d'algorithmes, tandis que l'efficacité de programmes complets dépend de paramètres très variés : efficacité des accès aux périphériques ; gestion de la structuration (appel de sous-routines ou de méthodes) par le compilateur, dé-récursivation automatique ou non, ... Le compilateur et l'environnement d'exécution prennent donc une part importante dans l'efficacité du code.

Or, que sait le programmeur de la façon dont un programme écrit dans un langage L est compilé puis exécuté ?

- Le programmeur a pu être initié au langage en tant qu'étudiant, ou dans un séminaire : la connaissance de la relation entre le langage et l'exécution est alors transmise de façon implicite.
- La documentation du langage (les manuels) donne des indications sur la façon dont le code écrit en L est exécuté. Cependant, cette documentation est insuffisamment précise ou, comme nous le verrons plus loin (1.3.2), ambiguë.
- La *culture* du programmeur lui permet de «deviner» avec plus ou moins de précision l'exécution correspondant à son programme. Par exemple, un code récursif évoquera ou non en lui, selon son expérience de la programmation récursive, un coût à l'exécution lié à la gestion de l'appel des fonctions ; ou encore, il saura ou non identifier si cette récursion est terminale, et si elle l'est évaluer la probalilité que le compilateur l'identifie également et dé-récursive automatiquement la procédure.
- Les exécutions successives de plusieurs versions compilées du programme fournissent au programmeur une indication concrète de l'efficacité relative des différentes versions du programme. Il peut en inférer le travail effectué par le compilateur.
- Il est possible d'utiliser un «profiler» pour étudier attentivement comment s'exécute le programme compilé.
- Les discussions avec d'autres programmeurs, ou la lecture des listes sur Internet,

permet au programmeur de renforcer ou de corriger sa compréhension du langage.

1.1.2 Comment écrire un code efficace ?

On peut écrire «directement» un code efficace. Cela signifie que le programmeur effectue directement, avant d'écrire, un choix entre plusieurs codes possibles pour une même fonctionnalité, choisissant le code qu'il juge le plus efficace.

L'autre procédure consiste à écrire d'abord un code qui satisfait à la fonctionnalité requise, et ensuite à le transformer pour le rendre plus efficace. Il s'agit alors de l'optimisation, «à la main», d'un code existant. Parmi les optimisations à la main, il est possible de distinguer deux types d'optimisation :

- l'optimisation faite par la modification du texte du programme, dans son aspect algorithmique, au niveau général ou sur des détails de quelques lignes.
- L'ajout au texte du programme d'indications au compilateur. C'est-à-dire que le programmeur charge le compilateur d'effectuer les optimisations, et lui donne simplement des indications. Ces indications sont fournies au moyen d'expressions que permet d'écrire le langage. Il s'agit d'aspects du langage qui ne concernent donc pas la programmation proprement dite, mais la compilation, et plus exactement, la partie «optimisation» dans le travail du compilateur. C'est-à-dire que le programmeur optimise à la main «au niveau de la compilation», plutôt qu'au niveau du texte lui-même.

En termes de coopération entre le programmeur et le compilateur, ces deux façons d'optimiser peuvent être distinguées de la façon suivante :

- Dans le premier cas, il s'établit une coopération que nous appelons **implicite**. En effet, pour évaluer la qualité des «améliorations» qu'il effectue sur le programme, le programmeur se repose sur la connaissance qu'il a de la façon dont est compilé et exécuté le programme. Cette prise en compte du compilateur et du mode d'exécution n'est pas explicitée dans le texte du programme—c'est-à-dire qu'elle peut être observée, mais par quelqu'un qui comprend la raison implicite des choix exhibés par le code.
- Dans le second cas (l'ajout au code d'indications au compilateur), il est clair qu'il y a coopération entre le programmeur et le compilateur, et nous appelons **explicite** ce type de coopération.

Coopération et complexité théorique Pour ce qui est de la complexité d'un algorithme formulé dans le langage de programmation, rappelons qu'elle est calculée à partir de présupposés sur la façon dont le programme est compilé. Alors deux cas sont possibles :

1. ces présupposés sont corrects : alors l'ordre de complexité de l'algorithme ne change pas.

2. ces assomptions sont fausses : c'est le cas si le compilateur dispose de stratégies de compilation qui rendent caduques les dites assomptions, et alors c'est *a priori* «en bien», c'est-à-dire que la complexité de l'algorithme «réel» est au moins inférieure à la complexité théorique. Cependant, dans le cas où l'efficacité de l'algorithme n'est pas lié à la seule complexité «en calculs», le «mieux» n'est pas garanti. Pour exemple, considérons l'importance, pour l'efficacité d'un programme parallèle, du schéma de communications inter-processeurs. Le compilateur, en effectuant une optimisation locale pour les calculs ou pour le schéma de communications, peut facilement rompre un équilibre global calculs-communications qu'aura trouvé le programmeur, et augmenter en fait le temps d'exécution du programme.

—**Coopération avec le compilateur ?** Nous parlons de coopération entre le programmeur et le compilateur, mais le compilateur n'est qu'un programme qu'exécute le programmeur : nous justifions donc ici l'usage du vocable «coopération».

Le terme de *langage* de programmation lui-même est abusif, dans le sens où il n'y a pas communication entre le programmeur et la machine à programmer, au moyen de ce langage : le texte du programme constitue simplement l'expression des tâches que devra effectuer la machine. Comme le fait remarquer Neville Holmes [54], l'expression «code de programmation» serait plus appropriée.

Cependant, il est possible de considérer que le programmeur «dialogue», en fait, avec l'implanteur du compilateur. En effet, c'est ce dernier qui détermine comment seront interprétés les termes du programme en termes d'opérations que devra exécuter la machine : c'est donc lui qui «comprend» les programmes écrits par le programmeur. La coopération décrite ci-dessous est donc une coopération entre le programmeur et l'implanteur du compilateur, mais pour faire vite, nous parlerons dans la suite de coopération programmeur-compilateur.

Le cas du parallélisme

La coopération programmeur-compilateur est particulièrement poussée dans le cadre de la programmation parallèle, car, comme nous l'avons vu dans l'introduction, la raison même du parallélisme est la recherche de l'efficacité : il est donc naturel que dans la programmation parallèle, cette recherche ne s'arrête pas au fait que l'on programme parallèle : il s'agit de tirer parti au maximum de la machine, et cela suppose une collaboration efficace entre le programmeur et le compilateur.

1.1.3 Exemples de coopération

L'écriture de code «propre» Une façon simple d'établir une bonne coopération avec le compilateur est d'écrire un code «propre», i.e. où même en essayant d'optimiser, le programmeur conserve la lisibilité du code. C'est-à-dire qu'il évite de faire des optimisations mécaniques que pourrait faire le compilateur, en se concentrant sur des optimisations «intelligentes».

Fortran Un exemple fameux de coopération implicite programmeur-compilateur avec Fortran est la prise en compte, dans le traitement de tableaux à deux dimensions ou plus, du mode de stockage de ces tableaux colonne par colonne en mémoire.

Rappelons, pour la petite histoire, que Fortran a connu (avec FORTRAN I) au moins une instruction d'indication au compilateur : l'instruction `FREQUENCY`, qui permettait au programmeur d'indiquer au compilateur la probabilité que le test d'une conditionnelle s'évalue à «vrai». Cette instruction a été supprimée par la suite, après qu'elle eut été implantée «à l'envers» dans FORTRAN II sans que personne ne le remarque ! Ce qui indiquait que cette instruction n'était en fait pas utilisée par les programmeurs ...

Le langage C On trouve dans le langage C la même possibilité d'optimisation en exploitant le fait que les tableaux sont implantés ligne par ligne. Cela pose d'ailleurs un problème si l'on souhaite faire coopérer des codes Fortran et C sur des mêmes tableaux de données, et que les deux codes ont été optimisés par rapport aux accès aux tableaux !

Un autre exemple de coopération implicite en C est la dé-récursivation «à la main» d'un algorithme écrit, pour sa première version, sous forme récursive. L'expression récursive d'un algorithme et sa version itérative sont deux expressions différentes d'un même algorithme. Souvent, l'expression récursive est la plus simple : pourtant, le programmeur prendra la peine de donner la version itérative. Il est motivé pour cela par sa connaissance du mécanisme d'exécution de la version récursive : il sait que la récursion sera implantée par le compilateur par des appels de fonctions successifs, avec une pile pour stocker les valeurs des variables de l'appel précédent. Cette implantation est coûteuse en temps à cause du coût de l'appel de fonction, et en temps d'accès mémoire, lorsque la pile ne peut pas être stockée entièrement sur le processeur. Le travail de dé-récursivation est traité, par exemple, dans [94].

Pour la coopération explicite, est disponible en C le mot-clé `register` : la déclaration «`register int x`» indique au compilateur que la variable x sera abondamment utilisée, et que ce serait une bonne idée de la stocker dans un registre lors de l'exécution (mais le compilateur n'y est pas tenu).

Notons également le statut particulier de la déclaration «`volatile`» : il s'agit d'une directive qui demande au compilateur, au contraire, de ne pas optimiser sur la variable `x`. C'est-à-dire que le programmeur prend en compte consciemment la partie «optimisation» du travail effectué par le compilateur, est obligé de «désamorcer» l'optimisation sur `x` pour conserver le programme qu'il souhaite.

Quant à l'écriture d'un code propre, on notera l'intérêt pour le compilateur de l'usage, par le programmeur, du mot-clé `const`, qui indique que la valeur de la variable déclarée en «`const`» ne sera pas modifiée. En effet, le compilateur, sachant que la valeur de la variable déclarée `const` ne varie pas, peut la placer dans un segment mémoire accessible en lecture seule.

C++ Un exemple de coopération explicite en C++ est l'instruction `inline`, destinée au compilateur. Elle indique au compilateur d'essayer de remplacer tout appel à la méthode déclarée en `inline`, par une copie du code de la fonction (cette copie pouvant également être optimisée en fonction du contexte de l'appel). Le code d'une méthode déclarée en `inline` est en général court, et donc le coût mémoire de la recopie du code est censé être négligeable comparé à l'avantage de la suppression de l'appel de la méthode.

Certaines implantations de C++ proposent encore d'autres raffinements dans la coopération explicite. Par exemple, les implantations KAI, Cray, SGi, Intel, et gcc reconnaissent l'instruction `restrict`, qui permet au programmeur d'indiquer qu'un certain pointeur ou une référence n'a aucun alias : cela permet au compilateur d'être libre dans ses optimisations par rapport à ce pointeur ou cette référence.

Prolog La face la plus «officielle» de Prolog [17] est son aspect de «langage de programmation logique». C'est-à-dire qu'il est possible d'écrire des programmes Prolog en ne se souciant de rien d'autre que de l'aspect logique de ce que l'on écrit. Cependant, si l'on souhaite obtenir des programmes efficaces, il faut en venir à une coopération implicite avec l'interpréteur, qui consiste à prendre en compte le mécanisme de résolution des règles de Prolog au moment où l'on rédige son programme Prolog.

Prolog permet également une coopération explicite avec son interpréteur, avec la directive «cut» («`!`») [17, pp. 69-92]. En introduisant des «`!`» dans ses règles, le programmeur, qui a bien compris le mécanisme de résolution, indique à l'interpréteur quelles branches de l'arbre d'exploration il est inutile d'explorer. Il est à noter cependant que ce cas de coopération explicite est particulier, car l'ajout d'une coupure (un «cut») dans le texte du programme donne un programme qui peut être considéré comme différent du précédent, bien que cette différence ne soit pas exprimable en termes de logique, mais uniquement en termes d'interprétation du programme. En effet, la version avec «cut» n'est pas logiquement équivalente à la précédente, et l'aspect «logique» d'un programme Prolog disparaît au fur et à mesure de l'introduction de coupures. Les directives mentionnées dans les exemples précédents (et suivants), par contraste, ne modifient pas le paradigme d'expression utilisé par le programme.

Perl Mentionnons également, comme un exemple de coopération explicite, une coopération qui ne vise pas à obtenir un programme plus efficace, mais à développer plus rapidement le programme : en Perl [113], la possibilité de déclarer, sous le mode «`$`», un numéro pour la ligne suivante dans le script. Ce numéro sera mentionné par le compilateur dans les messages d'avertissements ('warnings'). Cette directive est similaire à la directive «`#line`» du langage C.

OpenMP OpenMP est un mode de programmation parallèle qui consiste à ajouter à un code Fortran ou C, des directives au compilateur qui indiquent les points où le parallélisme est possible et les points où il n'y en a pas. Cette expression du parallélisme s'appuie sur la notion de «threads» (processus légers). Il s'agit donc d'une coopération explicite avec le compilateur : le programmeur indique les points où des threads peuvent être créés et utilisés, ou détruits.

1.2 Le parallélisme de données

La programmation par parallélisme de données est choisie comme objet d'étude au chapitre 3. Nous expliquons ce choix ci-dessous, puis nous donnons un état de l'art du parallélisme de données.

1.2.1 Motivation pour l'étude du parallélisme de données

Le choix du parallélisme de données comme objet d'étude Nous avons choisi le parallélisme de données comme objet d'étude parce que, la coopération programmeur-compilateur ayant pour but l'efficacité des programmes, elle est particulièrement importante dans le cadre de la programmation parallèle. L'état actuel du savoir-faire en matière de programmation et de compilation pour les machines parallèles justifie par ailleurs de préciser l'état de la coopération programmeur-compilateur dans ce domaine.

Les deux grands modes de programmation parallèle sont le parallélisme de données et le parallélisme de tâches. En termes de langages de haut niveau ils sont illustrés, par exemple, par les langages HPF et OpenMP. HPF représente la voie du parallélisme de données, tandis qu'OpenMP est fondé sur la notion de processus légers concurrents accédant à un espace mémoire vituellement partagé. Ces deux langages reposent sur l'utilisation de directives explicites au compilateur. L'expression des directives en HPF, et plus généralement dans les langages à parallélisme de données, est plus riche : elle concerne à la fois l'exécution des instructions et la distribution des données. Nous l'avons donc choisi comme objet d'étude.

La coopération programmeur-compilateur dans le parallélisme de données La programmation par parallélisme de données repose fortement sur la coopération programmeur-compilateur, implicite et explicite. Le fondement de la programmation par parallélisme de données est le fait que lorsqu'une même opération est appliquée à tous les éléments d'une collection de données, ces applications peuvent être effectuées en parallèle. En conséquence, l'écriture d'un programme à parallélisme de données suppose que le programmeur exprime ses algorithmes autant que faire se peut, comme des opérations globales sur des collections de données. Il s'agit là d'une forte coopération implicite avec le compilateur, car c'est le compilateur qui va générer du parallélisme à partir de ces opérations.

Mais l'obtention de programmes à parallélisme de données vraiment efficaces suppose également que le programmeur aide le compilateur dans la tâche de *placement des données* sur les processeurs. Cela se fait par des directives au compilateur : il s'agit d'une coopération explicite. Elle est quasiment de règle, par exemple, dans l'utilisation du langage HPF.

1.2.2 Le parallélisme de données hier et aujourd'hui

Le principe du parallélisme de données

L'idée sous-jacente à la programmation parallèle de haut niveau est que le programmeur doit pouvoir imaginer ce que peut faire son programme à l'exécution tout en réduisant, autant que possible, l'expression explicite de la concurrence que cela implique.

Par rapport à cette idée, la programmation à parallélisme de données est une solution intéressante. En effet, dans ce modèle, le parallélisme est réduit à la possibilité d'exécuter en parallèle des opérations globales sur des collections de données ; il n'est donc pas nécessaire au programmeur de penser à chaque élément de la collection individuellement. Ainsi, c'est au niveau de la collection que se situe l'expression du parallélisme, et l'appréhension du parallélisme du programme par le programmeur peut s'arrêter au fait que chaque opération globale peut donner lieu à une implantation parallèle.

Quant au programme dans son ensemble, il sera composé d'une séquence d'instructions globales sur des collections de données. La programmation séquentielle étant cognitivement relativement aisée, cet aspect de la programmation par parallélisme de données consiste en fait à se rapporter à un mode de programmation connu et relativement aisé.

Le principe de la séquence d'instructions parallèles a été exposé par Luc Bougé, dans [11]. Il pose le contraste entre le «`PAR` of `SEQ`» du parallélisme de tâches et le «`SEQ` of `PAR`» du parallélisme de données. Nous pourrions préciser que ce qui fait la différence entre les deux modes de programmation, au-delà de cette inversion dans la composition du «`PAR`» et du «`SEQ`», c'est l'implication de cette inversion au niveau de la charge cognitive pour le programmeur. En effet, le «`PAR`» du parallélisme de données est un `PAR` d'instructions simples et toutes identiques, tandis que le `PAR` du parallélisme de tâches est un `PAR` de tâches (une tâche, séquence d'instructions, étant plus complexe qu'une simple instruction), ces tâches étant en outre susceptibles d'interagir entre elles.

L'évolution du parallélisme de données

Le parallélisme de données, comme mode de programmation, a été utilisé pour la première fois pour la programmation de la Connection Machine, introduite en 1985 [51]. Pour la programmation de cette machine, le langage Lisp a été adapté, le résultat étant le langage *Lisp [100]. Paralation Lisp a également été implanté sur la Connection Machine, ainsi qu'une adaptation du langage C, C* [99]. Toutes ces adaptations introduisaient des instructions globales sur des listes (Lisp) ou des tableaux (C), l'intention étant que ces opérations soient effectuées avec un parallélisme massif par les nombreux processeurs de la Connection Machine.

Par la suite, peu de machines à parallélisme massif ont été produites, en-dehors de la série des Connection Machine's (CM-I, CM-II, CM-200, CM-5), mais le mode de programmation inauguré avec les Connection Machines a été conservé. En particulier, étant particulièrement adapté au traitement de tableaux, il a pu être exploité par le calcul scientifique. Les langages à parallélisme de données suivants furent des adaptations de langages utilisés dans le calcul scientifique, en particulier C et Fortran. Le premier langage à parallélisme de données largement disponible fut Fortran 90. Le standard actuel, HPF, succède à Fortran 90.

Les directives de placement

L'état de l'art en techniques de compilation ne permet pas d'espérer que les compilateurs des langages à parallélisme de données puissent bientôt mettre ces langages en concurrence directe avec des langages comme MPI pour la programmation d'applications parallèles. Cependant le langage HPF fournit au programmeur le moyen d'aider le compilateur dans la tâche du placement des données. En effet, le modèle implicite de la programmation par parallélisme de données est celui d'une machine à mémoire distribuée, par conséquent avec des coûts de communications de données entre processeurs. Il est donc judicieux de définir un bon placement des données, qui consiste à minimiser le nombre, ou la distance, de ces communications. L'importance de la notion de minimisation des communications dans le parallélisme de données est telle que la notion de placement de données a pu être considérée comme l'essence du parallélisme de données (voir, par exemple, [107]).

Le problème cependant, avec les directives de placement de HPF, est que rien n'oblige le compilateur à en tenir compte. Il en résulte l'impossibilité, pour le programmeur, d'affiner ou de corriger ses indications, puisque rien ne lui permet d'établir un lien causal simple entre ses directives et les performances du programme observées à l'exécution.

En comparaison, l'utilisation de MPI, si elle ne permet pas forcément au programmeur de prédire avec précision la performance de son programme, lui permet d'identifier facilement, en cas de résultats décevants, les points critiques concernés, de telle sorte qu'il peut tâcher de remédier au problème.

1.2.3 Les langages à parallélisme de données académiques

Dans la présente section nous présentons le parallélisme de données tel qu'il est proposé dans des langages moins répandus que HPF, proposés par des équipes de recherches académiques. Par «langage», nous entendons ici des langages implantés, tels que tout programme écrit dans un de ces langages puisse être exécuté sur une machine parallèle (après compilation sur une machine séquentielle ou parallèle).

DPHaskell

Une approche pragmatique consiste à étendre la programmation fonctionnelle au cas du parallélisme de données. En effet, dans la programmation fonctionnelle, les algorithmes s'expriment souvent naturellement, par des opérations sur des collections de données (typiquement des listes). Il «suffit» donc de pouvoir indiquer que ces opérations globales peuvent être effectuées en parallèle.

Dans le langage fonctionnel DPHaskell (Data-Parallel Haskell), extension du langage fonctionnel Haskell [24], l'approche consiste à permettre au programmeur d'indiquer dans son programme, par l'insertion appropriée d'annotations (les combinateurs «`par`» et «`seq`»), si l'exécution doit s'effectuer en parallèle ou en séquentiel. Il s'agit donc de directives explicites au compilateur ou à l'interpréteur, à la façon de HPF ou d'OpenMP. Cependant, le modèle d'exécution associé, qui utilise des processus légers, est plus proche de celui d'OpenMP que du modèle d'exécution de HPF.

NESL

NESL [9] est un langage à parallélisme de données fondé sur la notion de suite («sequence», en anglais). Les opérations globales sont appliquées sur de telles suites de données. Le langage ressemble à Standard ML [88], mais sans les fonctions d'ordre supérieur (il n'est pas possible de passer des fonctions en paramètres d'autres fonctions).

Il est possible d'avoir en NESL des suites imbriquées (des suites de suites, impliquant à l'exécution des opérations globales un «parallélisme imbriqué»), d'où le nom de «NESL» (Nested Language). L'imbrication de suites est utile pour l'écriture d'algorithmes de type «diviser pour régner».

Comme la programmation en NESL ressemble à de la programmation fonctionnelle, les programmes NESL sont élégants, et NESL a pu être utilisé pour enseigner le parallélisme.

La bibliothèque `BSMLlib`

Frédéric Loulergue a proposé une bibliothèque de fonctions Caml permettant d'écrire dans le langage fonctionnel Caml [68] des algorithmes data-parallèles qui suivent le modèle d'exécution BSP (Bulk Synchronous Parallelism, [102]). Cette bibliothèque, présentée par F. Loulergue dans sa thèse [70], est appelée `BSMLlib`, et ses primitives suivent le modèle BSλ, proposé par F. Loulergue dans sa thèse pour concilier programmation fonctionnelle et parallélisme de données. Ce modèle est présenté plus avant ci-dessous (Sec. 1.2.4).

1.2.4 Les modèles sémantiques et leur notation

La programmation par parallélisme de données, exprimant une certaine forme de coopération programmeur-compilateur, doit pouvoir s'appuyer sur une compréhension claire de principes fondateurs : nous présentons ces principes de la programmation par parallélisme de données, d'ordre plus «sémantique», à travers quelques études académiques.

Comme notre étude, au chapitre 3, traite principalement des directives au compilateur pour le placement des données, nous partageons la présentation ci-dessous en deux parties : la première traite des modèles qui incluent la notion de placement des données sur des processeurs, et la deuxième, des modèles qui n'incluent pas cette notion. Ces derniers modèles sont cependant pertinents dans le cadre de cette thèse car, du fait même qu'ils s'affranchissent de la notion de placement, ils traitent de façon plus claire des principes du parallélisme de données.

Modèles avec une notion de placement

Le langage $\mathcal{L}$ Le langage $\mathcal{L}$, comme notation pour étudier la programmation par parallélisme de données, a été proposé par Luc Bougé [13]. Dans ce langage à parallélisme de données, les collections de données sont des tableaux (au sens de C ou de Fortran). Les indices de tout tableau, outre leur fonction d'identification d'une case mémoire contenant une donnée, sont l'identifiant d'un processeur. Ainsi, toute donnée du tableau est mise en correspondance avec un processeur (et cela avec un processeur par donnée), et une opération globale sur le tableau est alors censée être exécutée en parallèle par tous les processeurs virtuels qui sont associés au tableau (par ses indices).

Les structures concrètes distribuées Gaétan Hains, Frédéric Loulergue et John Mullins ont introduit dans [47] le modèle des *structures concrètes distribuées* pour la programmation parallèle. Ce modèle est une variante des *structures de données concrètes généralisées* (voir [47] également), inspirées des *structures de données concrètes* de Brookes et Geva [14].

Le principe des structures concrètes distribuées consiste à associer à une structure de données concrète généralisée, une fonction totale Λ de l'ensemble des *cellules* vers l'ensemble des indices des processeurs considérés. Les cellules des structures de données concrètes sont les lieux où peuvent se trouver des valeurs : la fonction Λ distribue ces cellules sur les processeurs, distribuant du même coup les données qui doivent s'y trouver. Cette notion de distribution sur les processeurs via une fonction se retrouve dans le modèle suivant, Pei.

La programmation qui utilise directement le modèle des structures concrètes distribuées n'est cependant pas facile, car la programmation se fait au niveau des (indices des) processeurs (des exemples sont donnés dans l'annexe de l'article [47]). Mais le modèle peut, comme le fait remarquer Gaétan Hains [48], servir pour exprimer la sémantique de langages comme Pei.

BSλ Frédéric Loulergue a proposé dans sa thèse [70] une extension du λ-calcul avec des opérations BSP, qu'il a appelée BSλ. (Pour un court rappel sur le λ-calcul, voir la section 1.3.8 ci-dessous.) Comme dans le modèle des structures concrètes distribuées présenté ci-dessus, les valeurs d'un objet global sont explicitement distribuées sur des processeurs : cela pour que le modèle puisse, en suivant le modèle d'exécution BSP, se prêter aux calculs de coûts que permet le modèle BSP. Cependant, contrairement à la programmation directe suivant les structures concrètes distribuées, le modèle de programmation associé à BSλ, est relativement aisé. Il est implanté par la bibliothèque de primitives `BSMLlib` (voir ci-dessus), et F. Loulergue mentionne que «des étudiants de maîtrise familiers avec Caml ont pu écrire des algorithmes parallèles simples très rapidement [en utilisant cette bibliothèque]» [70, p. 132].

PEI Le langage PEI est le second des deux langages à parallélisme de données que nous allons étudier dans le chapitre 3 (l'autre langage étant HPF). C'est pourquoi la description qui suit est plus conséquente que les trois précédentes.

- **historique et motivation**

Le langage PEI [110, 106], à l'origine une notation, a été développé au laboratoire Image et Calcul Parallèle Scientifique (ICPS), par Éric Violard, Stéphane Genaud et Guy-René Perrin.

Le thème majeur de l'ICPS dans les années 1990 était la parallélisation automatique avec le «modèle polyédrique» [81, 22]. La motivation initiale pour le projet PEI fut de se libérer de la contrainte des dépendances affines du modèle polyédrique en se plaçant dans un nouvau cadre formel permettant de définir un système de transformation de programmes par l'utilisation de règles, dans l'esprit de Morgan [82], et une notion d'un «placement-ordonnancement» («placement» seul dans les derniers articles) associé à toute collection de données.

- **Le modèle sémantique**

PEI s'est développé comme une notation pour exprimer, avec un modèle sémantique précis, des collections de données placées.

Les objets de base du langage PEI sont des collections de données auxquelles sont systématiquement associées un placement sur des processeurs. Ces objets sont appelés *champs de données* mis en forme ('shaped data fields'). Les opérations sur les champs de données affectent également leur placement. En conséquence, la formulation de la sémantique dénotationnelle de PEI n'est pas triviale ; sa dernière formulation est [108]. (Sur la sémantique dénotationnelle en général, voir ci-dessous, la section 1.3.4.) Nous donnons ci-dessous un aperçu de la version exposée dans la littérature [109]. Cette sémantique sera exposée plus avant dans la section 3.4.

Un *champ de données mis en forme* est modélisé par un couple de fonctions (v, σ), où $v : I \rightharpoonup V$ est une fonction partielle d'un ensemble I d'indices vers un ensemble V de valeurs possibles, qui représente la collection de données ; et où $\sigma : I \rightharpoonup I$ représente le placement des données, via le placement des indices sur des processeurs virtuels, dont les indices sont eux aussi sont pris dans I.

Les opérations du langage s'effectuent sur des champs de données mis en forme. Une opération appliquée sur un champ `X` de sémantique (v, σ), pour donner en résultat un champ

`Y` de sémantique (v', σ'), correspond à une opération globale sur la collection de données de `X` qui est modélisée par v, ainsi qu'à l'expression d'une contrainte sur les placements respectifs des données de `X` et de `Y` sur les processeurs. Cette contrainte est exprimée par une relation entre les fonctions σ et σ'.

La sémantique opérationnelle de PEI a longtemps été considérée comme «évidente»; ainsi, des traducteurs de PEI vers HPF [26] et Caml [28] ont pu être écrits en 1997, à partir de la sémantique opérationnelle «intuitive» de PEI. La formalisation de cette sémantique a été donnée tardivement, en 2000 [27]. (Sur la sémantique opérationnelle en général, voir ci-dessous, la section 1.3.6.)

Modèles sans notion de placement

8.1/2 8.1/2 [76] est un langage à parallélisme de données qui permet d'exprimer des algorithmes sur des structures régulières. Une partie du langage utilise une structure de groupe sur les indices des données [29], de telle sorte que les dépendances exprimant l'algorithme peuvent être exprimées avec la loi de composition.

Le langage Alpha Alpha [114] est une notation pour les programmes qui peuvent s'écrire sous la forme d'équations récurrentes. Les travaux sur la compilation de programmes Alpha sont donc voisins de ceux de la parallélisation automatique, ou moins automatique mais utilisant tout de même les notions du «modèle polyédrique» : ordonnancement, alignement, et allocation physique – voir, par exemple, [93]. L'apport de ce modèle pour la coopération programmeur-compilateur se trouve, via des travaux comme celui cité ci-dessus, dans l'expression simple qu'il permet de notions importantes dans la compilation de langages à parallélisme de données.

Le formalisme de Lisper Björn Lisper a proposé un formalisme pour le modèle de programmation par parallélisme de données [69]. Il a repris la notation du lambda-calcul typé (sur le lamda-calcul, voir la section 1.3.8 ci-dessous), et attribue un sens de parallélisme de données aux fonctions, et à certaines opérations sur les fonctions. Par exemple :

- une fonction f d'un ensemble d'indices vers un ensemble de valeurs représente une collection de données; f est appelée un *champ de données* (data field) (cette notion est similaire à celles des champs de données PEI (voir Sec. 1.2.4) sans le placement).
- l'opération globale est la composition à gauche : si g est de type $\gamma_1 \times \cdots \times \gamma_n \to \tau$, et si les f_i, $1 \leq i \leq n$ sont des champs de données de type $\sigma \to \gamma_i$, alors $\lambda x.g(f_1(x), \ldots, f_n(x))$ est l'application de l'opération g à $f_1, \ldots, f_n$.
- les dépendances entre champs sont exprimées, comme en Fortran, par des fonctions sur les indices; cela est codé par la composition à droite : si $f : \tau \to \sigma$ est un champ de données, et g une fonction de type $\gamma \to \tau$, alors $\lambda x.f(g(x))$ est le «get» de f à partir

de g. Le vocable «get» est ici employé car dans le cadre du parallélisme de données, une dépendance implique, par défaut, une communication pour obtenir la donnée à partir du processeur où elle se trouve, vers le processeur où elle va être utilisée dans un calcul (pour l'application locale d'une opération globale).

Ainsi, un algorithme à parallélisme de données peut être codé par un lambda-terme. L'exécution de cet algorithme est alors donnée par une ré-interprétation de la β-réduction des lambda-termes : la β-réduction représente encore une évaluation, mais si l'argument est une fonction f notant un champ de données, cette fonction elle-même devra d'abord, la première fois qu'elle est utilisée en argument, être évaluée *sur tout son domaine de définition*, i.e. pour tout x tel que $f(x) \neq \perp$; le résultat de cette évaluation intrinsèquement parallèle sera placé dans une table ; et toute utilisation suivante de ce champ de données, pour accéder à une de ses valeurs, pourra effectuer de simples "look-up" dans cette table. Ainsi, on reconstruit des tableaux «à la Fortran», mais avec le λ-calcul comme notation, et la sémantique dénotationnelle du λ-calcul typé fournit une sémantique dénotationnelle à ce formalisme. Les champs de données de Lisper correspondent, en cela, aux fonctions «v» des champs de données de PEI 1.2.4.

L'apport du modèle de Lisper dans notre contexte de coopération programmeur-compilateur est l'éclairage qu'il apporte sur les idées qu'il partage avec le langage PEI, qui sera étudié au chapitre 3.

Le formalisme de Bird et Meertens Le formalisme de Bird et Meertens ('Bird-Meertens Formalism', BMF) [5] est une notation qui a pour but de permettre de dériver sur un mode calculatoire des algorithmes efficaces. Les objets de base considérés par BMF sont des structures de données «conteneurs» comme les listes. Plus précisément, il s'agit des types de données qui peuvent se définir comme des foncteurs polynomiaux—pour la notion de foncteur, voir la section 2.2. Par exemple, les listes peuvent être définies comme la somme du foncteur qui construit les listes à un élément, et du foncteur qui concatène les listes.) Les opérations sur les structures de données de BMF sont définies à partir de la structure même (i.e., à partir du foncteur polynomial.

David Skillicorn a montré comment BMF pouvait être vu comme un langage à parallélisme de données [95]. En effet, les types de données utilisés étant des «types conteneurs», les opérations globales sur les objets de base de BMF peuvent être effectuées en parallèle.

La limite du modèle est dans le fait qu'il est limité aux structures qui peuvent se définir comme des foncteur polynomiaux. Cette limite est surmontée avec la théorie des formes, présentée dans le paragraphe suivant.

La théorie des formes (shape theory) Barry Jay a donné un modèle de la notion de structure de données (conteneur) qui met l'accent sur la relation entre les *positions* auxquelles se trouvent les données dans la structure, et la collection des données elles-mêmes : c'est la *théorie des formes* («shape theory») [57].

La théorie des formes a pour but de distinguer, dans les opérations sur des structures de données, celles qui n'affectent que les données de la structure (indépendamment de sa «forme») de celles qui prennent également en compte la forme de la structure—d'où l'intérêt de définir la notion de «forme», et le nom de la théorie. La théorie utilise, comme BMF, la notion de foncteur.

L'apport des «shapes» au parallélisme de données a été exposé par Barry Jay, par exemple dans [58].

Pour nous, nous retiendrons la mise en avant de la notion de position pour traiter des collections de données : cette notion sera particulièrement utilisée au chapitre 3.

1.3 La sémantique formelle des langages de programmation

Dans les deux chapitres suivants, nous utilisons une méthode formelle, la *logique de réécriture* (présentée dans la section 1.3.6 ci-dessous, et décrite plus avant dans la section 2.2.2) pour décrire la sémantique de langages de programmation. L'avantage qu'il y a à utiliser une méthode formelle est expliqué ci-dessous. Pour situer notre approche, nous présentons ensuite l'état de l'art en matière d'utilisation de méthodes formelles pour décrire la sémantique d'un langage.

1.3.1 Préambule

Les méthodes formelles pour la description de la sémantique des langages de programmation sont très variées, et il eût été suffisant de situer notre approche par rapport aux méthodes les plus proches. Cependant, nous choisissons de présenter un état de l'art sous la forme d'un survol de l'ensemble des approches existantes, en essayant de situer celles-ci entre elles. Nous croyons qu'un tel exposé peut être utile, car nous n'en avons pas trouvé dans la littérature.

La multiplicité des méthodes est illustrée par le volume B du *Handbook of Theoretical Computer Science* [103]. Dans cet ouvrage, chacune des principales méthodes existantes est exposée dans un chapitre qui lui est consacré : mais l'ouvrage est constitué de la concaténation de ces chapitres, sans qu'il y ait de chapitre de synthèse. En outre, les chapitres du *Handbook*, reflétant en cela la littérature sur le domaine, utilisent une grande variété de techniques mathématiques : celui qui souhaite utiliser une méthode formelle et consulte la littérature ou le *Handbook* pour tâcher de déterminer, parmi les méthodes existantes, la plus appropriée à son besoin, aura à réviser (ou apprendre) un minimum, au passage, toutes ces techniques mathématiques. Ainsi, il devra considérer : la théorie des domaines [3] pour la sémantique dénotationnelle ; des notions de topologie, pour une meilleure compréhension de la théorie des domaines, et parce que des travaux comme [34] utilisent directement des notions de topologie ; la théorie des modèles [92, 36] pour les spécifications algébriques ; la théorie des catégories [64], car les résultats en théorie des domaines et en spécifications algébriques sont compris plus facilement dans leur exposition sous forme catégorique ; et parce que de nombreux autres résultats sont, pour la même raison, exposés dans un cadre catégorique, voir par exemple [73] ; la théorie des preuves (un texte d'introduction est [31]), pour comprendre comment des avancées comme la logique linéaire [30, 2] sont pertinentes dans le cadre de la sémantique des langages de programmation.

Ainsi, acquérir une vue d'ensemble du sujet dans le seul but de pouvoir choisir la ou les méthodes les plus appropriées à son propre besoin, et ce en consultant directement la littérature, requiert un investissement non négligeable. C'est pourquoi nous proposons le survol qui est donné ci-dessous.

1.3.2 Introduction

Motivation pour les méthodes formelles

Rappelons pourquoi on cherche à décrire le sens d'un langage de programmation. Cela provient de la nécessité de s'entendre entre le programmeur et l'implanteur du langage, sur le sens de ce langage : il faut en effet que tout programme compilé effectue, à l'exécution, «ce que le programmeur a imaginé que le programme doit faire».

Mais où le «sens» du langage est-il défini ? La construction du sens du langage par le programmeur passe beaucoup par sa pratique et plus précisément, par l'interaction avec le compilateur. Il faut donc qu'un document décrive, à l'intention des implanteurs, le sens du langage : c'est le rôle d'une **norme** et des manuels de référence pour le langage. Ces normes et manuels sont également étudiés par le programmeur lorsqu'il souhaite s'assurer de la signification du langage sans dépendre pour cela du compilateur.

Ces manuels de référence sont écrits, dans la très grande majorité des cas, en «langage naturel». L'expression courante «langage naturel» est utilisée par opposition à l'expression «langage formel», un langage formel utilisant des objets et notations mathématiques. L'intérêt d'un langage formel est qu'il s'appuie sur la forte convention qui s'est établie sur la signification de ces notations.

Bien sûr, cette convention ne suffit pas : lorsque la sémantique d'un langage a été décrite au moyen d'objets mathématiques, il faut que le lecteur sache quelle est la signification codée par ces objets mathématiques. Dans la pratique, cela est le plus souvent non énoncé, i.e., laissé implicite dans la description, ce qui est codé étant censé être clair d'après le contexte et la motivation de la description. Cela montre que le rôle d'une description formelle n'est pas tant de définir d'une façon absolue la sémantique du langage, que de la *préciser* autant qu'il est jugé nécessaire. De même, lorsque des explications sur un codage formel sont tout de même données (en langage naturel), il s'agit en grande partie d'explications «d'aide au décodage», i.e., ce n'est pas tant la sémantique qui est discutée, que son codage. Cela est illustré, par exemple, par les «annotations pragmatiques» de la définition du langage

Algol 68 [104] : chacune de ces annotations suit un paragraphe de description formelle, et a pour but—cela est expliqué au début du manuel— : «to help the reader to understand the intentions and implications of the definitions and to find corresponding sections or rules.» [104, p. 18]. Un autre exemple est la définition du langage SML : un manuel de référence [79] décrit la sémantique du langage, par une suite de formules dont la signification est censée être évidente ; mais un autre ouvrage, le *Commentary on Standard ML* [78], l'accompagne, afin d'éclairer certains points.

Pratique des sémantiques formelles

Sémantique formelle et langages «réels» Les manuels de référence des langages de programmation pour les langages «réels» (i.e., non académiques) sont presque toujours rédigés en langage naturel (à l'exception de la donnée de la grammaire du langage).

En fait, le seul exemple connu de nous d'un langage utilisé dont le manuel utilise une méthode formelle, est le langage SML. Mais ce langage, un langage fonctionnel, n'est pas aussi utilisé que C ou C++, et est encore considéré comme un langage «académique», malgré son utilisation dans certaines entreprises.

En fait, le fait qu'une description formelle de la sémantique de SML soit disponible est sans doute fortement lié au fait que la conception même du langage SML soit allée de pair avec sa description : R. Milner mentionne que cela a été un facteur de cohérence et de simplicité du langage. En effet, le fait qu'un langage doive pouvoir être décrit de façon simple dans le cadre d'une méthode formelle est *de fait* une contrainte de simplicité et de cohérence dans la conception de ce langage.

Pour les autres langages de programmation «réels», les descriptions formelles de leur sémantique ont été tentées après la conception (et l'implantation) du langage. Ces descriptions après-coup cherchent davantage à montrer l'applicabilité de la méthode de description utilisée, qu'à fournir un texte de référence. A l'heure actuelle, la recherche pour des méthodes de descriptions adéquates continue. Une approche prometteuse semble être la «notation d'actions» (*Action Notation*, également appelée *Action Semantics* [85] ; voir la section 1.3.8 ci-dessous) : cette approche se distingue de celles qui l'ont précédée en ce qu'elle est plus pragmatique et cherche en premier lieu à fournir une notation convenable, les questions de sémantique «pure» étant reléguées au second plan.

La sémantique formelle des langages académiques Les langages académiques ont le plus souvent pour visée de permettre de tester la pertinence des idées sémantiques qui les fondent, plutôt que de fournir un langage utile dans un contexte industriel. La définition des fondements formels des langages académiques vient d'ailleurs généralement *avant* leur conception même.

Quelques paramètres socio-historiques concernant la recherche en sémantique formelle En informatique théorique, la communauté des sémanticiens a une intersection importante avec celle qui s'occupe de méthodes de dérivation formelle de logiciel. Or cette dernière communauté a mis pendant longtemps un fort accent sur la notion de *preuve de programme*. Les méthodes de description de sémantique en sont donc naturellement venues à devoir permettre de préférence d'effectuer des preuves de programme, ce qui est une contrainte très forte, alors que la réalité passée et actuelle du génie logiciel exhibe des contingences bien différentes. Cette situation était déjà évoquée en 1979, dans [77].

De même, une place importante a été accordée à la notion de calcul, et les langages académiques on mis l'accent sur cette notion, provoquant une prise en compte importante de la notion de calcul dans les méthodes de description formelle. Un résultat est que la notion d'entrées-sorties a été reléguée à un statut inférieur à celle de calcul. Nombre de travaux se sont par exemple ensuite attachés à montrer comment inclure les entrées-sorties dans des méthodes de description sémantique destinées en premier lieu à modéliser le calcul. Un exemple bien connu est le cas des langages fonctionnels, qui héritent du λ-calcul.

De manière générale, on peut peut-être considérer que les chercheurs en sémantique des langages de programmation ont plus de goût pour les problèmes liés à la sémantique elle-même que pour ceux liés à la communication d'une sémantique. C'est pourquoi, alors, une méthode comme la «notation d'actions» mentionnée plus haut apporte du nouveau, bien que n'apportant rien de révolutionnaire sur un plan purement sémantique : P. Mosses prend soin de préciser cela : «The only real originality of action semantics lies in the design of action notation» [85, p. 5].

1.3.3 La classification standard des méthodes formelles

Nous rappelons dans cette section la classification usuelle des méthodes formelles de description de sémantique, en trois grandes approches :
- l'approche dite **dénotationnelle**,
- l'approche dite **axiomatique**,
- et l'approche dite **opérationnelle**.

D'autres classifications sont possibles, mais comme cet exposé a pour but de donner une vue globale du domaine de la sémantique formelle dans de but de permettre de s'orienter dans la littérature, nous préférons reprendre la classification ci-dessus, qui est celle à laquelle se réfère la littérature. Cette classification a également une relation à l'histoire du domaine qui nous permet de structurer notre exposé avec des points de repère historiques.

Cependant, nous tâchons tout d'abord de justifier cette classification d'un point de vue logique—i.e., sans référence historique. Cette discussion nous permettra également d'évoquer dans la conclusion de cette section les autres classifications possibles.

Classification : le rapport au calcul Ainsi que nous l'avons mentionné ci-dessus, la notion de calcul a longtemps été, et est encore, une question centrale en sémantique des langages de programmation. Nous choisissons de justifier la classification standard à partir de la notion de calcul.

Définissons d'abord de façon générale ce qu'est un calcul, ou plus exactement, un calcul automatisé. Nous distinguons trois étapes dans tout calcul :

1. la *préparation* du calcul, par la personne qui souhaite que ce calcul se fasse. Dans le cas du calcul numérique, il s'agit de disposer d'une machine, de la programmer à cet effet, et de lancer le calcul. Les autres modes de calcul actuellement étudiés sont le calcul «moléculaire» (utilisation de solutions et de réactions chimiques), et le calcul quantique.
2. Le calcul s'effectue.
3. la «fin du calcul» est repérée (ou décidée), et la configuration physique résultante est interprétée par la personne qui voulait ce calcul.

Les méthodes formelles existantes pour la description de la sémantique d'un langage s'attachent à définir le calcul lui-même, c'est-à-dire la deuxième étape. Elles peuvent être classées selon deux grandes voies concernant la description du calcul :

- la première voie consiste à décrire uniquement le rapport entre les entrées et les résultats du calcul. Ainsi, on évite d'avoir à dire comment sont faits les calculs. Cette voie se partage en deux approches :
  - l'approche **dénotationnelle**, qui consiste à modéliser la relation entre les données en entrées et les données en sortie sous la forme d'une fonction (mathématique).
  - l'approche **axiomatique**, qui consiste à décrire le calcul par un couple de prédicats : le premier décrit l'état de certaines variables «avant le calcul», et le second l'état des mêmes variables «après le calcul». La sémantique d'une expression d'un langage de programmation est donc définie par un tel couple de prédicats.
- la deuxième voie consiste, au contraire, à essayer de modéliser ce qu'est le calcul, en tant que mécanisme. Cette voie correspond à l'approche dite **opérationnelle**.

Nous exposons à présent plus en détail les trois approches mentionnées ci-dessus, dans le même ordre : dénotationnelle, axiomatique, puis opérationnelle, car il s'agit de leur ordre historique d'apparition dans la littérature, et suivre cet ordre facilite l'exposé. Puis, après avoir traité le cas de la programmation logique, nous exposerons une autre approche très utilisée en pratique : la **traduction**.

1.3.4 La sémantique dénotationnelle

L'approche dénotationnelle, la première à promettre de permettre de donner une sémantique bien définie à un langage de programmation, a nourri la recherche en sémantique dans toutes les années 1970.

L'idée de la sémantique dénotationnelle consiste à affecter une **dénotation** (au sens du logicien Gottlob Frege) à un texte de programme, sous la forme d'une fonction mathématique. Cette méthode est présentée en détail dans [83].

C'est le logicien Dana S. Scott [44], qui a introduit cette idée de modélisation. Scott a également conservé l'idée de **compositionalité** de Frege, selon laquelle la dénotation d'un terme complexe devrait pouvoir être donnée en fonction de la dénotation de ses composants. (Par la suite, on a pu donner des dénotations à des programmes sans forcément utiliser une compositionalité stricte.)

La sémantique dénotationnelle a pu donner un fondement théorique aux langages fonctionnels [98]. Cela est naturel, ces langages étant fondés sur la notion de fonction et d'application de fonction à des arguments.

Mais, en-dehors de ce succès, et ainsi qu'il est indiqué par Peter Mosses [83, p. 626], la sémantique dénotationnelle, dans l'état actuel des outils techniques qui ont été développés autour de cette idée, ne permet pas de donner la sémantique d'un langage «réel». Peter Mosses cite Samson Abramsky sur cette question, dans [85] : «... once languages with features beyond the purely functional are considered, the appropriateness of modelling programs by functions is increasingly open to question. Neither concurrency nor 'advanced' imperative features have been captured denotationally in a fully convincing fashion.»

1.3.5 La sémantique axiomatique

La sémantique axiomatique consiste à utiliser des prédicats pour décrire les états d'avant et d'après le calcul. Elle est surtout connue sous le nom de *logique de Hoare*, depuis l'article [52] de C. A. R. Hoare.

Le principe est le suivant. Soit un texte P (un programme) qui décrit un calcul, et qui utilise des variables `x`, `y`, ... : la définition de ce qui est effectué par ce calcul est donnée par un couple de prédicats $(pre, post)$, *pre* et *post* associant chacun une valeur à chacune de ces variables. Les valeurs décrites par *pre* sont celles «avant» le calcul, et celles données par *post* sont les valeurs des mêmes variables «après» le calcul. Cela est couramment noté : $\{pre\}P\{post\}$.

Les deux grands champs d'application de la sémantique axiomatique, et par lesquels elle est principalement connue, sont :

1. la preuve de programmes. Une présentation de cette utilisation de l'approche axiomatique peut être trouvée dans [19].
2. La construction de programmes. L'utilisation de la sémantique axiomatique pour la construction de programmes est présentée dans [23]. Il s'est avéré que cette méthode n'est pas applicable à la programmation de grands systèmes ; cependant, ses principes de bases ont pu et peuvent servir à des programmeurs prêts à les utiliser. Par exemple, le consultant Robert L. Baber rapporte dans [4] combien les invariants de boucles lui ont rendu service.

En ce qui concerne la description de la sémantique de langages, cependant, l'approche axiomatique n'est pas très répandue. Il faut mentionner toutefois son application à la définition du langage Pascal [53]. Un exemple contemporain d'application de la sémantique axiomatique à la définition d'un langage est la définition du langage parallèle Alpha (voir ci-dessous, en section 1.2.4) [12].

1.3.6 La sémantique opérationnelle

L'approche opérationnelle tente de décrire le mécanisme du calcul, c'est-à-dire en quoi consiste le calcul, «concrètement», plutôt que de décrire simplement la relation entrées-sorties qui résulte du calcul.

Comme on peut s'y attendre, cela n'est pas tâche facile, car tandis que les deux approches précédentes ont pu s'appuyer sur l'essence même des mathématiques du XX$^{\text{ième}}$ siècle, à savoir la théorie des ensembles et la logique du premier ordre, ces mêmes mathématiques n'ont pas abordé la notion de déroulement d'un processus. Les physiciens théoriciens ont pu élaborer des modèles espaces-temps en utilisant sur plusieurs dimensions le corps des réels ; mais l'informatique doit considérer des modèles discrets, et doit trouver comment intégrer la notion d'algorithme.

Dans la pratique, jusqu'à présent, la description du «comment» d'un calcul a toujours consisté à définir le calcul comme une suite d'*états* d'un même "quelque chose" (par exemple une machine), représentant l'évolution de ce quelque-chose dans le temps ; deux états successifs représentant alors, de par leur différence, l'effet d'une opération. C'est-à-dire que le changement d'un état dans le temps est codé par le changement de certains de ses attributs.

Le premier modèle du calcul a été les machines de Turing. Un état y est un état de la bande. Puis, la description d'états successifs s'est faite par des méthodes dites *syntaxiques*, c'est-à-dire où chaque état est représentés par un terme syntaxique. Dans les années 80, les méthodes syntaxiques étaient ainsi associées à l'approche opérationnelle, et comme l'approche dénotationnelle utilisait un modèle mathématique, l'opposition entre méthode mathématique et méthode syntaxique en est venue à être synonyme de l'opposition entre approches dénotationnelle et opérationnelle. Cependant, cette situation a évolué, et des méthodes opérationnelles à modèle mathématique ont commencé à apparaître.

Ci-dessous, nous présentons les méthodes existantes pour la description de la sémantique opérationnelle des langages de programmation : l'approche par la syntaxe du langage, l'approche par modèle mathématique, et l'approche par traduction vers la syntaxe d'un autre langage ou vers une notation connue. Nous devons signaler que tous les modèles du calcul ne sont pas mentionnés : sont inclus, en effet, seuls ceux qui ont effectivement servi à décrire la sémantique opérationnelle d'un langage de programmation. Ainsi, nous n'incluons pas les machines de Turing. Certes les machines de Turing peuvent être vues comme servant à décrire la sémantique du langage de Turing (le «langage machine» des machines de Turing), mais à notre connaissance elles n'ont pas servi à décrire la sémantique d'un autre langage de programmation.

Sémantique opérationnelle par la syntaxe

Dans l'approche opérationnelle par la syntaxe, chaque état de la «chose» qui évolue au cours du calcul est représenté directement par un terme syntaxique ; et un changement d'état est alors simplement représenté par une modification (syntaxique) de ce terme.

La signification de chacun de ces termes syntaxiques, c'est-à-dire l'état représenté par le terme, est soit suffisamment évidente d'après le contexte pour qu'elle n'ait pas à être explicitée, soit est explicitée en langage naturel, dans le texte qui décrit la sémantique du langage. De même pour les règles qui décrivent les modifications syntaxiques : seules les parties «non évidentes» de la signification de ces règles sont explicitées, en langage naturel. Ainsi, l'approche opérationnelle syntaxique est simple, puisque le «sens» qu'elle décrit est fourni automatiquement, de façon «extérieure à la description», par le lecteur de cette description, à partir de son interprétation des termes syntaxiques, guidées éventuellement par des remarques en langage naturel dans le texte.

La forme la plus connue de l'approche par la syntaxe est la **sémantique opérationnelle structurée** [91]. Les règles sont écrites, graphiquement, sous la forme de fractions, ce qui facilite leur lecture : au-dessus de la barre de fraction est le «terme de départ», et le dénominateur est le terme d'arrivée. Une variante de la sémantique opérationnelle structurée est la «**sémantique naturelle**» (*natural semantics*) , ainsi nommée car elle est inspirée du système logique appelé *déduction naturelle* et introduit en 1965 par Dag Prawitz. La sémantique naturelle a par exemple été utilisée pour décrire la sémantique du langage Standard ML [79].

Sémantique opérationnelle par modèle mathématique

Nous présentons à présent les méthodes qui permettent de décrire la sémantique opérationnelle d'un langage en la codant sous la forme d'un objet mathématique.

La différence avec le principe de l'approche par la syntaxe exposée ci-dessus, est qu'un *niveau intermédiaire* est ajouté entre le terme syntaxique et sa signification. En effet, d'abord le terme syntaxique est traduit en une structure mathématique : puis c'est à cette structure mathématique qu'est attribuée un sens. Comme pour l'approche syntaxique directe, soit la signification du codage par l'objet mathématique est suffisamment claire d'après le contexte, soit celle-ci est explicitée, pour ses parties non évidentes, en langage naturel dans le texte.

L'avantage de passer par l'intermédiaire d'une structure mathématique est que cela permet de préciser, justement, une structure dans la signification. Pour cela, la forte convention sur laquelle reposent les objets mathématiques «standard» (ensemble, fonctions) est utilisée. C'est-à-dire qu'une structure est explicitée à moindre frais : au lieu de la décrire en langage naturel, il est fait appel à un objet mathématique de type standard qui exhibe cette structure.

Les méthodes opérationnelles par modèle mathématique sont plus récentes que les précédentes : elles sont apparues à partir du milieu des années 1980, et se sont développées dans les années 1990.

La construction du point fixe dans la sémantique dénotationnelle Nous commençons notre exposition en mentionnant un modèle qui est une exception à la chronologie annoncée ci-dessus, car il date d'avant les années 1980 ; mais il n'était pas suffisamment satisfaisant pour être utilisé pour une sémantique opérationnelle. En outre, sa nature opérationnelle ne se formulait pas facilement au regard du cadre dans lequel elle s'insérait : en effet, il s'agit de la construction par point fixe de la dénotation (au sens de Scott) d'une expression donnée (voir [44]). Cette construction peut être considérée comme «opérationnelle» si l'ordre des éléments de la suite croissante qui part de l'élément «bottom» ($\perp$) du domaine, et allant jusqu'au plus proche point fixe, est interprété comme un ordre temporel discret : alors la suite fournit l'équivalent de la trace de l'exécution d'un algorithme. Observer une trace d'un algorithme permet de donner une idée dont procède l'algorithme en question, mais généralement pas une idée suffisamment précise pour «retrouver» le mécanisme opérationnel de l'algorithme ; autrement dit, cette trace ne peut pas faire fonction de sémantique opérationnelle pour cet algorithme. Cependant, ces suites ont pendant longtemps représenté la seule forme mathématique existante du mécanisme opérationnel d'un algorithme.

Les Abstract State Machines Les «*machines à états abstraits*» ('*abstract state machines*') ont été introduites par Yuri Gurevich [45].

Cette méthode de description de la sémantique opérationnelle de langages adopte une approche pragmatique, en cherchant à coller à la réalité du mode d'exécution de la plupart des langages actuels, et propose un modèle mathématique général pour ces exécutions, à instancier suivant le langage que l'on cherche à décrire.

Le modèle général prend directement en compte les caractéristiques de l'exécution d'un programme : les notions d'état, de succession d'états et de mise à jour (updating). Il permet de définir des *machines abstraites*, qui fonctionnent comme des machines dédiées chacune à l'exécution de programmes écrits dans un certain langage. Une description de la sémantique d'un langage consiste donc à définir une machine abstraite pour exécuter les programmes écrits dans ce langage.

Cette méthode a pu être utilisée, par exemple, pour donner la sémantique du langage C [46].

Les spécifications algébriques et la réécriture Les spécifications algébriques [115] ont été introduites à la fin des années 1970. Elles étaient motivées par le désir de donner une sémantique aux structures de données [39]. Les spécifications algébriques ont plus tard été utilisées pour décrire la sémantique de langages de programmation, comme dans [37].

Le principe des spécifications algébriques est celui de la *théorie des modèles* [92]. C'est-à-dire qu'une structure de données est décrite par la déclaration des opérations sur cette structure qui lui sont typiques, et la donnée d'équations qui définissent les propriétés de ces opérations. Autrement dit, la structure est définie par les opérations : les spécifications algébriques suivent par là le principe d'encapsulation proposé au début de la même décennie (1972) par David Parnas [87]. La «sémantique» de la structure est alors donnée comme étant l'ensemble des modèles de la *théorie* ainsi décrite (où un *modèle* est une algèbre qui interprète chacune des opérations par une fonction, de telle sorte que les fonctions satisfont les propriétés énoncées par les équations de la théorie). Comme aucun modèle n'est privilégié parmi tous les modèles, cette méthode revient effectivement à décrire une structure «abstraite», au lieu de décrire une instance de cette structure.

Pour décrire la sémantique d'un langage, l'approche utilisée par Joseph Goguen et Grant Malcolm dans [37] consiste à spécifier les expressions du langage, un modèle de mémoire avec des opérations de mise à jour associées, et une opération qui représente l'évaluation des programmes. C'est un langage impératif qui est ainsi décrit. Le fonctionnement de l'évaluation est décrit par les équations, mais cette description du mécanisme d'évaluation est perdu lors du passage au modèle mathématique, puisque l'interprétation de l'opération d'évaluation n'est qu'une fonction qui satisfait aux équations.

Cependant, l'idée de Goguen et Malcolm est d'utiliser, pour représenter la sémantique opérationnelle, le système de réécriture associé à la spécification. En effet, dans la description d'une spécification, les axiomes sont écrits comme des équations, «de droite à gauche», avec un signe «=» entre les deux membres de l'équation. L'ordre de ces membres n'influe pas sur les modèles, mais est choisi le plus souvent en fonction de la signification intuitive de l'équation. Par exemple, l'équation

```
top (push p n) = n .
```

qui définit l'opération «`top`» sur les piles d'entiers, est écrite comme si les opérations étaient exécutées, retournant un résultat qui serait défini par le membre droit de l'équation : cela est plus intuitif que l'équation de même sens mathématique

```
n = top (push p n) .
```

En conséquence, lorsqu'un système de réécriture est associé à la spécification, ce système considère les équations, en les lisant de gauche à droite, comme des règles de réécriture syntaxique. Cela permet alors, par exemple, de réduire le terme «`top (push empty 3)`» en «`3`». Avec ce principe, les axiomes qui définissent l'opération d'évaluation des programmes écrits dans le langage spécifié, peuvent être lus comme signifiant le mécanisme d'évaluation. Ainsi, la sémantique opérationnelle est donnée par le système de réécriture. L'inconvénient est que tout axiome ne correspond pas forcément à la description d'un mécanisme d'évaluation : l'interprétation à donner aux règles de réécriture figurées par ces axiomes pose donc problème.

La logique de réécriture La logique de réécriture (*rewriting logic*, RWL), introduite par José Meseguer [75], apporte une solution au problème soulevé ci-dessus, en donnant un cadre logique combinant axiomes et règles de réécriture (appelées *transitions*). Ainsi, il n'est plus besoin de lire des axiomes comme des règles de réécriture, ou des règles de réécriture comme des axiomes.

Le rapport de recherche [71] présente (entre autres) des applications de la logique de réécriture à la formulation de la sémantique de langages, sur l'exemple d'un mini-ML, mais en utilisant seulement des transitions. L'exemple qui présente le codage du lambda-calcul utilise en revanche la distinction équation/transition, en ne donnant que la règle de β-réduction comme une transition, ce qui correspond à la compréhension «implicite» de la signification du λ-calcul (sur le λ-calcul, voir l'exposition brève qui en est donnée dans la section suivante).

La différence entre équations et transitions est en revanche exploitée à plein dans [112], pour donner une sémantique au π-calcul. Ce travail utilise également le fait que les équations, bien qu'étant à présent distinguées des règles de transitions, peuvent encore être utilisées comme règles de réécriture dans un système de réécriture [111]. Ce principe sera également exploité dans les chapitres suivants de cette thèse.

Le modèle mathématique associé à la logique de réécriture est une adaptation de celui des spécifications algébriques : un modèle d'une théorie, au lieu d'être une famille d'ensembles et des fonctions qui interprètent les opérations, est une famille de catégories, et les opérations sont interprétées par des foncteurs. L'idée est que les objets de la catégorie représentent des états, et les flèches des transitions entre états. La déclaration d'une transition dans la théorie correspond à la déclaration d'une transformation naturelle : ce sont les flèches associées à cette transformation naturelle qui codent, au niveau du modèle mathématique, les transitions entre états.

Un avantage de la logique de réécriture est qu'elle est reliée à l'héritage des *systèmes de réécriture* [20]. En effet, les transitions entre états sont décrites sous la forme de règles de réécriture de termes syntaxiques—la différence avec les systèmes de réécritures est que la logique de réécriture permet d'interpréter ces termes dans des modèles. Cette correspondance permet aux transitions d'une théorie de la logique de réécriture d'être réalisées effectivement par une implantation (par un système de réécriture). Les implantations actuelles de la logique de réécriture avec un système de réécriture associé sont Maude [16], Elan [10], et CafeOBJ [86].

Mentionnons enfin que la logique de réécriture à été présentée, dans [75], comme un modèle de la concurrence. Cet aspect sera exposé plus bas, dans la section 2.2.3.

1.3.7 Le cas de la programmation logique

Mentionnons rapidement le cas de la description formelle de la sémantique des langages de programmation logique.

Un exemple d'un tel langage est Prolog, mentionné plus haut, en section 1.1.3. La sémantique de la partie «pure» d'un tel langage peut être décrite en termes de logique formelle. Cependant, ainsi que nous l'avons indiqué, le «cut» coupe également au travers de cette sémantique. Il en est de même pour la formulation des entrées-sorties : la signification logique d'une entrée ou d'une sortie est nulle, ce qui ne reflète évidemment pas la sémantique réelle d'une entrée ou d'une sortie.

Ainsi, selon que le manuel de programmation sera orienté vers la théorie ou vers la pratique, l'aspect logique y sera plus ou moins présent. D'une part l'aspect logique pur d'une règle n'est qu'une partie de la signification de cette règle au coeur d'un programme, puisque le programmeur a le plus souvent en tête, pour ses choix de conception, le mécanisme de résolution. D'autre part, les éléments «pragmatiques» du langage comme les entrées-sorties et le «cut» ne peuvent être négligés. Ainsi par exemple, dans le manuel de programmation *Programming in Prolog* de Clocksin [17], l'algorithme de résolution est expliqué en détail et de façon pédagogique, en langage naturel, dans les tout premiers chapitres (chapitre 1 : 'Tutorial Introduction', et chapitre 2 : 'A Closer Look'), tandis que les chapitres suivants sont consacrés aux aspects moins «logiques» du langage.

1.3.8 Sémantique par traduction

Une autre façon de décrire la sémantique d'un langage de programmation consiste à décrire comment traduire les programmes écrits dans ce langage, dans un autre langage, dont la sémantique est considérée comme plus claire.

Dans ce sens, un compilateur fournit une sémantique opérationnelle au langage qu'il accepte en entrée. En effet, il traduit les programmes en code objet ; et un code objet a une signification opérationnelle relativement bien connue : il s'agit des instructions acceptées par le processeur. Nous ne pouvons pas dire que le langage du processeur a une sémantique opérationnelle parfaitement claire, en raison des bogues possibles, des architectures complexes comme le pipeline, et la micro-programmation : cependant un niveau d'assurance sur ce qui est effectué est atteint, que n'a pas un langage de haut niveau dont la sémantique n'est pas définie en détail et clairement.

Mais choisir, pour décrire le sens d'un langage de haut niveau, d'expliquer comment il est traduit par un compilateur particulier, et renvoyer le programmeur à la documentation concernant le langage assembleur cible n'est bien sûr pas une bonne solution : dans le manuel de référence du langage, il faut une description plus concise.

Ainsi, une approche par traduction consiste, pour décrire la sémantique d'un langage L, à choisir un langage *simple* L' (simple au sens où la traduction d'un programme dans son entier est appréhendable par le programmeur) et à la sémantique relativement claire, et à décrire comment les programmes écrits en langage L sont traduits en L'. Cette description elle-même doit être la moins ambiguë possible : un support formel pourra pour cela être utilisé.

La sémantique du langage L' est définie, soit par la convention parmi les membres du lectorat visé, soit par une des trois approches décrites ci-dessus (axiomatique, dénotationnelle, opérationnelle), soit par une traduction vers un langage $L >>$ de sémantique connue. Ainsi la sémantique par traduction peut donner lieu la remarque selon laquelle : «one person's syntax is another person's semantics», que nous reprenons de [34, p. 1], où elle est formulée dans une discussion sur la sémantique des formalismes pour les systèmes concurrents.

Lorsque le lecteur est susceptible d'être un programmeur ou l'implanteur d'un compilateur, se pose réellement le problème du choix du langage L' : il faut que L' serve à expliquer L, donc que L' soit suffisamment connu, ou sinon que la documentation le concernant soit accessible pour ce lectorat, tant sur le plan physique qu'intellectuel.

C'est pourquoi, dans la pratique, l'approche par traduction a surtout été utilisée lorsque le lectorat est constitué de chercheurs en informatique théorique, tout langage ou toute notation suffisamment connue dans la communauté pouvant être utilisée dans le rôle du langage L'.

Ci-dessous, nous présentons la notation dite «action notation», à notre connaissance la seule notation qui soit conçue justement pour servir de «langage cible» (L') dans une description par traduction de la sémantique d'un langage réel. Cette notation est conçue pour être accessible à des lecteurs non informaticiens-théoriciens.

Puis nous présenterons trois notations liées à des modèles de calcul, que l'on rencontre fréquemment dans la littérature sur la sémantique des langages, et qui sont utilisées comme langages cible en recherche : le λ-calcul, les monades, et la théorie des preuves.

Action Semantics «*Action semantics*» (aussi appelée «*action notation*») est une notation proposée par Peter Mosses [85] pour permettre de décrire la sémantique d'un langage «réel».

L'idée de Mosses est, en réaction à l'échec relatif de la méthode de la sémantique dénotationnelle (comme il l'explique lui-même dans [85]), d'adopter une démarche pragmatique, et de considérer en premier lieu la notation qu'il serait commode de pouvoir utiliser pour décrire la sémantique de langages de programmation. Toujours de façon pragmatique, la sémantique associée à cette notation est décrite par plusieurs «vues» sémantiques. Une des vues correspond à la sémantique opérationnelle ; celle-ci ci est donnée à l'aide de la «sémantique naturelle», mentionnée ci-dessus (page 24).

La notation d'action fournit des «primitives» pour décrire la sémantique d'un langage de programmation. Et la sémantique opérationnelle d'un langage décrit avec la notation d'action est ainsi, «à la fin», donnée par la méthode de la sémantique naturelle, sans que le rédacteur de cette description ait besoin d'écrire des règles de sémantique naturelle : il (ré-)utilise la sémantique de la notation d'actions.

Le λ-calcul Le **λ-calcul** est souvent utilisé comme notation cible pour donner la sémantique de petits langages, ou de langages fonctionnels.

Le λ-calcul, introduit dans les années 1930 par Alonso Church, a été le premier modèle opérationnel du calcul [8, 7]. Le λ-calcul fonctionne comme modèle du calcul de la façon suivante : un terme syntaxique est construit, qui contient le codage des données en entrées, ainsi que le codage des calculs à effectuer sur ces données. Puis, ce terme initial est réduit (réécrit) pas à pas, en utilisant les règles de réduction (de réécriture) du lambda-calcul, jusqu'à obtention d'un terme que l'on ne puisse plus réduire avec ces règles. Les données «résultat» se trouvent alors codées dans le terme final ainsi obtenu. Une des règles de réécriture, appelée la *β-réduction*, est particulière, en ce que le terme réécrit représente un nouvel état (contrairement à ce qui se passe avec les autres règles) : c'est là que le déroulement d'un calcul est codé.

La définition de cette β-réduction se lit comme (la notation de) l'application d'une fonction à ses arguments. C'est pourquoi le codage du λ-calcul est intuitif : il peut être compris en termes de fonctions (les termes) et d'application de fonctions (la β-réduction). La sémantique proprement dite du lambda-calcul lui est donc alors attribuée par son utilisateur.

Le λ-calcul a beaucoup été utilisé comme langage cible dans les années 1960, la sémantique «évidente» exposée ci-dessus satisfaisant les chercheurs. Puis il a été évincé par la sémantique dénotationnelle, qui présentait sur lui l'avantage d'exhiber, comme sémantique, un objet mathématique (même si la notion de calcul était perdue).

Ce n'est qu'à la fin des années 1970, (c'est-à-dire très tardivement dans l'histoire du λ-calcul) qu'un modèle en termes de fonctions a pu effectivement être donnée pour le λ-calcul [7]. Il s'agit justement d'une sémantique dénotationnelle.

Depuis qu'une sémantique dénotationnelle a été donnée au λ-calcul, celui-ci est à nouveau utilisé comme une notation commode pour décrire les aspects fonctionnels de la sémantique de langages de programmation, la sémantique associée à cette notation pouvant être explicitée si besoin est. Cependant, il est passé, entre les années 1960 et son retour dans les années 1980, du statut de notation cible donnant une sémantique opérationnelle, à celui de notation cible donnant une sémantique dénotationnelle.

Les monades de Eugénio Moggi Une notation «à la λ-calcul» a été proposée par Eugenio Moggi [80] pour noter un modèle de calcul qui part de la méthode de la sémantique dénotationnelle. Dans la sémantique dénotationnelle classique, toute dénotation représente une **valeur**. L'idée d'E. Moggi est de distinguer, parmi les objets mathématiques qu'il utilise, ceux qui doivent représenter des valeurs, de ceux qui doivent représenter des **calculs**. Ces derniers ont alors pour fonction de **produire** des valeurs.

La cadre technique dans lequel cette sémantique est formulée est celui de la théorie des catégories [64], et la notion de *monade* est utilisée : le foncteur de la monade sert à «marquer» la différence entre un calcul et une valeur. C'est-à-dire qu'étant donnée une catégorie de types (c'est-à-dire une catégorie où les objets représentent des types), si A est un objet de cette catégorie, et T l'endofoncteur considéré, alors l'objet TA est pris pour représenter le type «calcul qui retourne une valeur de type A». Les transformations naturelles de la monade permettent ensuite de représenter naturellement les notions d'*évaluation* d'un calcul (pour obtenir une valeur), et de *composition* de calculs (codant ainsi les séquences de calcul).

La théorie des preuves La théorie des preuves est pertinente dans le cadre de la sémantique opérationnelle, en raison de l'**isomorphisme de Curry-Howard** [31, pp. 14-21], qui permet de considérer une preuve comme un calcul. Ainsi, une preuve développée avec le système d'aide à la preuve Coq [97] peut être traduite en un programme fonctionnel.

Comme modèle de calcul, un système d'inférence peut être considéré comme un modèle de calcul *syntaxique* : les formules sont des termes syntaxiques, qui peuvent être considérées comme codant des états ; et les règles logiques (règles d'inférence) données par le système peuvent être considérées comme fournissant les règles de réécriture qui décrivent les changements d'état possibles.

Une version récente de la théorie des preuves qui est particulièrement pertinente pour l'informatique est la **logique linéaire** [30], qui prend en compte la notion d'états d'un système. Cela est réalisé en permettant aux formules logiques de représenter des états ; une transition d'un état A vers un état B correspond à l'implication linéaire $A \multimap B$, différente de l'implication classique $A \Rightarrow B$. En particulier, les règles $A \wedge B \Rightarrow A$ (affaiblissement) et $A \Rightarrow A \wedge A$ (contraction) de la logique classique ne se transposent pas à l'implication linéaire, car l'affaiblissement signifierait la disparition magique de n'importe quel «matériel» d'un état, tandis que la contraction signifierait que n'importe quel «matériel» peut se démultiplier à volonté. Un article qui traite de la façon dont la logique linéaire permet de donner une sémantique opérationnelle à la programmation fonctionnelle (via le λ-calcul) est [101].

1.3.9 Conclusion

Opérationnalité

Nous avons présenté ci-dessous les méthodes formelles de description de langages de programmation, en suivant la classification standard sémantique axiomatique/sémantique dénotationnelle/sémantique opérationnelle.

Le problème que pose cette classification est que, si l'on se réfère à l'appellation des trois approches, elles ne s'opposent pas sur un même critère. En effet, «dénotationnel» réfère à l'*esprit* de l'approche (coder la sémantique associée à un terme syntaxique par un objet mathématique) ; «axiomatique» réfère à la *technique* employée (un codage sous la forme de formules logiques) ; tandis qu'«opérationnel» réfère directement au *résultat recherché* de l'approche (coder le mécanisme d'exécution). Cette non-communauté de critères explique que dans la pratique, de nombreuses techniques peuvent être vues comme appartenant à deux de ces approches, voire aux trois. L'exemple le plus évident a été donné ci-dessus, avec les approches opérationnelles par modèle mathématique, qui peuvent toutes être qualifiées d'approches dénotationnelles. La convention est faible sur ce point : de nombreux chercheurs accorderont le qualificatif de «dénotationnel» uniquement aux approches compositionnelles, ou encore uniquement aux approches où la dénotation est une fonction. L'étendue de l'approche dénotationnelle, entre la proposition initiale de Scott, et la multiplication récente des autres modèles mathématiques, est donc floue.

Or, comme nous l'avons montré, les trois approches peuvent également s'aborder à partir de la façon dont elles modélisent la notion de calcul. Nous précisons cette vue dans cette section, en montrant comment les trois approches peuvent être qualifiées d'opérationnelles, différant alors par :
– le degré de détail qu'elles donnent au mécanisme d'exécution ;
– la pragmatique de leur énonciation de l'opérationnalité.

La pragmatique des trois approches Le vocable «pragmatique» est pris ici dans l'acception qui lui est commune dans le champ de la linguistique. C'est-à-dire que nous examinons la convention à partir de laquelle le codage utilisé par chacune des trois approches «fait sens» pour celui qui l'utilise.

Nous montrons d'abord que la notion d'opérationnalité, au sens où elle fait référence au déroulement d'un calcul dans le temps, est bien présente dans chacune des approches axiomatique, dénotationnelle, et «opérationnelle». Elle y est simplement plus ou moins explicite.

– Dans le cas de l'approche axiomatique, c'est le couple ordonné d'états (état avant, état après) qui code, par l'ordre du couple, la notion d'«évolution dans le temps». [1] La notion d'évolution d'*un* état est implicite : il est supposé que l'«état après» n'est pas indépendant de la conjonction de l'«état avant» et du calcul (sinon, l'énonciation du couple (état avant, état après) ne serait pas très intéressante). Ainsi est donc bien codée l'évolution de l'état d'un *même* système dans le temps (sans qu'aucun modèle de ce système ne soit explicité).

– Dans l'approche dénotationnelle, la notion d'ordre dans le temps est codée par l'asymétrie, pour une fonction, entre le rôle de son domaine et celui de son codomaine. (C'est l'équivalent de l'ordre du couple de formules dans l'approche axiomatique.) C'est pourquoi on parle de «données» («input») et de «sorties» («output») de la fonction : implicitement, on a l'ordre suivant dans le temps : au début, le couple (données en entrées, fonction) ; puis l'application de la fonction à la donnée ; puis le résultat de cette application. Mathématiquement, l'application de la fonction à la donnée et le résultat ont même dénotation, et sont donc «égaux» ; mais bien évidemment, le *sens* réel de l'application est de *produire* (par un «calcul» non explicité) le résultat.

[1] Ce codage est «évident» pour un lecteur occidental ; mais des études récentes ont montré le fort lien qui lie le sens de lecture et l'ordre temporel : ce codage sera donc peut-être moins évident, par exemple, pour des lecteurs arabes.

- C'est dans l'approche dite «opérationnelle» que la notion de temps apparaît de la façon la plus évidente. Dans la sémantique opérationnelle, il est considéré que c'est la succession linéaire des termes syntaxiques qui code le déroulement, linéaire, du temps. Cette manière de modéliser le déroulement dans le temps suit la convention du récit linéaire, qui est si habituelle qu'elle ne pose effectivement aucun problème de compréhension. [2]

Niveaux de détail dans l'opérationnalité Nous pouvons à présent ordonner les approches selon le niveau de détail qu'elles permettent de donner au calcul :

- l'approche dénotationnelle décrit un calcul par une seule fonction. Aucun détail n'est donné sur la façon dont se fait ce calcul. Bien sûr, dans la pratique, cette fonction est définie à partir d'une séquence de calculs : mais le codage consiste à coder ces calculs par des fonctions, et leur succession dans le temps par une composition de fonctions. Or les fonctions n'ont pas été conçues, historiquement, pour coder des calculs : une conséquence est que leur composition n'implique nullement un ordre dans le temps : la séquence est donc perdue par la composition. En effet, la composée $g \circ f$ de deux fonctions g et h n'est pas liée à g et h : elle est en général égale à une infinité d'autres compositions. Il n'est donc pas possible de retrouver les calculs intermédiaires à partir de la fonction qui code tout le calcul. C'est pour cela que dans le cas des langages fonctionnels, la sémantique dénotationnelle est accompagnée de la sémantique opérationnelle, qui a essentiellement pour but de coder la même séquence de calculs, *mais sans perdre la séquence au cours du codage* ; l'exercice classique consiste ensuite à montrer que le rapport entre entrées et sorties du calcul entier qui est codé par la sémantique dénotationnelle sous forme d'une fonction, correspond bien au rapport entre entrées et sorties qui résulte de l'application de la succession des opérations définie par la sémantique opérationnelle. Un exemple de cet exercice est donné dans [98].
- L'approche axiomatique permet de moduler le détail des calculs, en introduisant des formules logiques intermédiaires pour coder les propriétés des états intermédiaires. Cette approche permet de conserver la notion de séquence dans le temps, car contrairement à la sémantique dénotationnelle, les dénotations y restent liées (par convention) à leur notation. C'est-à-dire que lorsque la sémantique de la séquence est définie par le fait que de $\{pre\}P\{mid\}$ et $\{mid\}P'\{post\}$ on peut déduire $\{pre\}P\,;P'\{post\}$, on s'autorise à «récupérer» P et P' comme étant des étapes intermédiaires de $P\,;P'$, alors que rien n'oblige *a priori* le comportement du programme $P\,;P'$ à dépendre du comportement de P et de celui de P'.

[2]—Pour des lecteurs occidentaux, en tout cas. En effet, des études anthropologiques ont montré que dans d'autres cultures, l'organisation temporelle des récits n'est pas la même. Par exemple, la séquence `a = 1 ; b = a + 2 ;` pourra être lue par défaut comme deux affectations pouvant se faire en parallèle.

– L'approche opérationnelle, enfin, a pour objet de détailler les étapes du calcul. Contrairement à l'approche axiomatique, où l'on n'est pas obligé de raffiner les détails d'un programme, dans l'approche opérationnelle les calculs de base sont définis en premier, et les programmes construits à partir de ceux-ci. Le mécanisme opératoire du programme entier est donc «par construction» disponible. (Bien sûr, rien n'empêche de définir des calculs de base très abstraits et qui font beaucoup de choses ; en cela on retrouve la notion de «choix du degré d'abstraction» de la sémantique axiomatique.) Ici, l'opérationnalité n'est pas perdue par le codage, parce qu'aucun modèle mathématique n'est là pour faire disparaitre toute notion d'opérationnalité. En effet, comme pour la sémantique axiomatique mais à un degré plus important, la signification des termes syntaxiques est attribuée directement à ceux-ci par le lecteur de la description, au lieu d'être d'être attribuée à des objets mathématiques qui coderaient la «signification» de ces termes.

Descriptions mixtes

La sémantique d'un langage de programmation, au sens de la compréhension du langage par le programmeur, est un objet très complexe. Par exemple, le langage n'est pas compris de façon abstraite, mais via sa pratique dans des projets logiciels. Et l'on touche alors à la façon dont un programmeur appréhende un programme, ce qui est un domaine de recherche à part entière («*program understanding*»). Ce qui est codé avec une méthode formelle ne peut donc prétendre être «la sémantique du langage», mais seulement une description de certains aspects logiques du langage, description qui servira davantage de référence que de médium par lequel le langage sera appris.

Dans cette fonction de référence, comme chaque méthode formelle se prête plus facilement à décrire tel ou tel aspect de la sémantique, il est raisonnable de documenter un langage à l'aide de plusieurs méthodes. C'est ce qui est fait, par exemple, pour la sémantique de la notation d'actions [85].

Chapitre 2

La sémantique de référence

2.1 Améliorer la coopération programmeur-compilateur

Cette thèse consiste à montrer comment préciser le rapport entre la sémantique du langage «pour le programmeur» et la sémantique du langage qui est donnée par le compilateur. Ce rapport sera précisé par la documentation du langage. Nous exposons ci-dessous le rôle que nous allons donner à la documentation du langage.

Dans le chapitre suivant, la méthode exposée dans ce chapitre sera utilisée : le présent chapitre constitue donc également une introduction au chapitre suivant.

2.1.1 La documentation du langage : sa forme habituelle

La coopération programmeur-compilateur repose, ainsi que nous l'avons exposé au chapitre précédent, sur une connaissance partielle par le programmeur de ce que fait le compilateur. En conséquence, la description de la sémantique du langage doit décrire la façon dont le langage va être compilé et/ou exécuté. Cette constatation n'est pas triviale, car il est fréquent que deux «modèles d'exécution» cohabitent dans la sémantique du langage : un modèle «officiel», qui correspond au paradigme sur lequel repose le langage, et le modèle «officieux» qui est fournit par le compilateur, et qui correspond au modèle d'exécution **effectif** du langage. Par exemple, en Caml, le modèle officiel repose sur les appels récursifs de fonctions, tandis que le compilateur va tâcher au maximum de dé-récursiver. Le programmeur est au courant de la sémantique officieuse : comme il sait que les appels récursifs sont coûteux, à l'exécution, en mémoire et en temps, sa coopération implicite avec le compilateur va consister à écrire de préférence des récursions que celui-ci sera susceptible de savoir dé-récursiver.

Or, dans les manuels de référence actuels des langages de programmation, cette distinction entre sémantique officielle et sémantique effective n'est pas énoncée de façon claire. Dans la pratique, le manuel décrira la sémantique officielle, avec pour règle implicite que «par défaut, la sémantique effective est identique», cette sémantique officieuse étant ensuite corrigée par le programmeur lorsqu'il interagit avec les compilateurs du langage, et/ou consulte la documentation de ces compilateurs, et constate qu'il n'y a en fait pas identité.

Cette règle de l'identité par défaut entre les deux sémantiques pose problème, car elle repose fortement sur des contrats implicites entre le programmeur et le compilateur, où la question de savoir ce qui est censé être fait par le compilateur peut surgir, pour le programmeur comme pour l'implanteur.

Un exemple est donné avec l'alignement «par lignes» des tableaux en C. Tout programmeur C apprend à prendre en compte cet alignement pour écrire des programmes efficaces. La norme [60] dit bien : «an array type describes a contiguously allocated nonempty set of objects with a particular member object types, called the *element type*» [60, p. 37]. Le stockage par lignes dans cet espace mémoire est décrit quant à lui *via* la façon dont les accès aux éléments d'un tableau sont traduits en termes de pointeurs [60, p. 71]. Or la norme précise bien : «The semantic descriptions in this International Standard describes the behaviour of an abstract machine in which issues of optimizations are irrelevant.» [60, p. 15]. Cela signifie que les compilateurs ne sont pas tenus de reprendre exactement le fonctionnement de cette machine abstraite, et donc qu'ils n'ont pas obligation de satisfaire le stockage par lignes. Cela peut être une bonne chose au vu des dernières avancées en termes de stockage de tableaux—voir par exemple [15]—où le compilateur peut trouver une meilleure allocation, qui n'est pas forcément l'allocation par lignes. Mais ce qui a toujours été implicite, c'est que le compilateur C satisfaisait *réellement* le stockage par lignes, de telle sorte que le programmeur pouvait prendre la peine d'écrire ses programmes en prenant en compte ce paramètre.

2.1.2 Une structure alternative pour la documentation du langage

Au regard des problèmes posés par le fait de s'appuyer uniquement sur des conventions implicites pour savoir ce que fera le compilateur, nous proposons dans cette thèse d'établir une distinction nette entre la sémantique du langage «pour le programmeur», et celle donnée par le compilateur. Cette distinction devra être reflétée dans la documentation de langage. Celle-ci aura pour rôle, afin de permettre la coopération entre le programmeur et le compilateur, d'établir les rapports entre les deux sémantiques. C'est-à-dire qu'elle sera structurée en deux parties : la description de la **sémantique officielle**, celle que le programmeur prend naturellement en compte pour concevoir son programme ; puis l'énonciation de ce que nous appelons le *contrat* entre le programmeur et le compilateur, c'est-à-dire une description des schémas d'exécution qui devront être effectivement suivis par les compilateurs du langage. Les termes de ce contrat constituent ce que nous appelons la **sémantique effective** du langage. Les directives pourront alors être décrites comme relevant de la sémantique effective.

Nous appelons **sémantique de référence** la sémantique du langage, telle qu'elle doit être donnée par notre méthode. La sémantique de référence se définit de la façon suivante :

$$\boxed{\textbf{sémantique de référence} = \textbf{sémantique officielle} + \textbf{sémantique effective}}$$

La partie «sémantique officielle» expose l'usage du langage (par le programmeur), tandis que la partie «sémantique effective» définit ce que doit réaliser un compilateur pour le langage. Ainsi, la description de la sémantique de référence correspond à la définition d'une *norme* pour le langage.

Ainsi, la stratégie que nous proposons doit permettre de *standardiser* plus facilement le langage. En effet, «a standard is often described as 'a contract between the programmer and the implementer.' It describes not only what is 'legal' source text, but also what a programmer can rely on in general and what behavior is implementation-dependent» [96, p. 133]. La sémantique effective décrit un comportement qui n'est pas dépendant d'une implantation particulière, mais doit être produit par toutes les implantations. La sémantique officielle décrit un comportement qui n'est pas dépendant d'une implantation particulière, et auquel les implantations ne sont pas liées par contrat.

Apports corollaires du contrat programmeur-compilateur

La formulation claire du contrat entre le programmeur et le compilateur permet au compilateur de savoir avec précision ce qu'il est censé faire, et au programmeur de savoir ce qu'il peut attendre du compilateur. Au-delà du confort psychologique qui en résulte (et qui est un point important), le respect du contrat simplifie la programmation et la compilation. Cela est très bien illustré par le cas des directives au compilateur. En effet, si la façon dont le compilateur prend en compte les directives est connue du programmeur, alors une amélioration des directives par rétro-action («feedback») avec l'exécution est possible. Du côté du compilateur, si la façon de prendre en compte les directives est fixée par le contrat, le compilateur n'est pas tenu d'ajouter des exceptions pour les cas où le programmeur aurait été maladroit ou incohérent dans l'écriture de ses directives ; c'est-à-dire qu'il peut se dispenser de faire des analyses coûteuses du programme à la seule fin de détecter ces éventuelles maladresses. I.e., le compilateur laisse au programmeur la responsabilité de ses choix. Un exemple simple est donné par l'indication «`restrict`» en C++ (voir la section 1.1.3) : si le programmeur ajoute à tort cette directive (i.e., l'objet est en fait pointé également par une autre variable), le programme compilé optimisé sera bogué, mais par l'unique faute du programmeur. Bien sûr, dans le cas de directives pour lesquelles les analyses de vérification ne sont pas trop coûteuses, elles peuvent être effectuées tout de même par le compilateur, comme un «service». Un exemple est donné par la vérification, par les compilateurs C++, du bien fondé des déclarations «`const`» par le programmeur.

Une exposition claire de la coopération entre programmeur et compilateur fournit un autre avantage : l'**enseignement** sur les langages de programmation est rendu plus clair— ou du moins, le lien entre un cours sur un paradigme de programmation et ce qu'observe l'étudiant lorsqu'il programme, en Travaux Pratiques, avec un langage relevant de ce paradigme. De même, les échanges entre programmeurs sur des problèmes techniques pourront être identifiés par eux comme relevant uniquement de la sémantique officielle, ou concernant la sémantique effective du langage.

2.1.3 Le mode de présentation de la méthode

Nous utilisons un support formel pour exposer notre méthode : la logique de réécriture. Il nous permet de parler clairement des sémantiques officielle, effective, de référence, et de leurs rapports.

Afin de rendre concrète la présentation de la méthode, nous nous appuyons sur un exemple simple de langage : nous définissons dans ce chapitre la sémantique de référence pour un petit langage impératif, que nous appelons «ATL» (*A Toy Language*).

Pourquoi utiliser la logique de réécriture ?

Le choix de la logique de réécriture comme support formel est motivé par les facteurs suivants :

- La logique de réécriture permet de *spécifier*. Comme notre méthode doit aider à rédiger le manuel de référence, nous spécifions des concepts familiers au programmeur et à implanteur du compilateur : les notions de *machine* (ordinateur), d'*interprétation* d'un programme sur une machine, et de *graphe d'exécution* correspondant à un programme.

- Les machines à états abstraits ou la «notation d'action» sont des méthodes proches de celle que nous utilisons, mais sont conçues pour définir *une* sémantique pour un langage. Notre préoccupation principale, par contraste, est d'étudier comment définir conjointement *deux* sémantiques (officielle et effective) pour un même langage, l'une (la sémantique effective) étant exprimée à partir de l'autre. Or la logique de réécriture hérite des spécifications algébriques des constructions sémantiquement précises pour la *modularisation* des spécifications [36]. Ainsi par exemple, les sémantiques officielle et effective pourront être définies dans des modules distincts, et cependant être combinées lorsque la sémantique effective doit faire référence à la sémantique officielle.

- La notion de modèle, intrinsèque à la logique de réécriture, sera extrêmement utile au chapitre suivant, lorsqu'il s'agira d'articuler la sémantique de directives au compilateur.

- La logique de réécriture est implantée dans des systèmes comme **CafeOBJ** [21] ou **Maude** [16]. Cela permet de tester les descriptions formelles, pour vérifier leur cohérence. D'un point de vue méthodologique, cela permet de repérer rapidement des erreurs dans les descriptions. Toutes les descriptions formelles données dans la suite de cette thèse ont été ainsi testées.

– Enfin, mais ce n'est pas le point le moins important, les modèles de la logique de réécriture ont un aspect *graphique* naturel : nous l'utilisons pour figurer les graphes d'exécution. Ce point est discuté ci-après.

Deux sémantiques ... Il nous faut mentionner le fait que pour deux langages il est souvent donné une sémantique dénotationnelle *et* un sémantique opérationnelle. C'est surtout le cas en ce qui concerne les langages fonctionnels. Comme nous l'avons mentionné au chapitre précédent (section 1.3.9), l'exercice consiste alors montrer une *cohérence* entre les deux sémantiques. Par contraste, dans le cadre qui nous intéresse, les deux sémantiques que nous souhaitons décrire ont un *statut* différent. Le rapport à décrire entre ces deux sémantiques ne consiste donc pas à montrer une telle cohérence, puisqu'elles n'ont tout simplement pas le même propos ; mais à indiquer le rapport entre leurs fonctions respectives. Une motivation supplémentaire pour notre choix de la logique de réécriture a été notre volonté de pouvoir décrire de préférence ces deux sémantiques avec une *même* technique de description, afin que les rapports de l'une avec l'autre puissent être aisément exprimés.

Graphes d'exécution

Les graphes d'exécution seront décrits, dans les modèles catégoriques, par des flèches entre des objets-états machine. Ainsi, la sémantique opérationnelle d'un programme peut être naturellement visualisée. Cela contraste par exemple avec l'approche opérationnelle structurée (voir la section 1.3.6), qui décrit la sémantique opérationnelle par des règles d'inférence : les règles d'inférence donnent une vision *locale* du mécanisme opérationnel, tandis que lorsque l'exécution d'un programme est décrite par les flèches d'une catégorie, une vision *globale* du comportement d'un programme est donnée. Bien sûr il est possible, à partir d'une description par des règles d'inférence, d'inférer à partir de ces règles un graphe d'exécution pour un programme, mais cela réclame une construction supplémentaire, la notion de graphe n'étant pas comprise de manière intrinsèque dans la méthode.

Utiliser un graphe d'exécution pour représenter la sémantique opérationnelle d'un programme, et donner ainsi une «forme graphique» à la sémantique du programme, touche au domaine de recherche «program understanding» (ou «program conprehension»), déjà mentionné en section 1.3.9. Sur la représentation visuelle des programmes, voir par exemple [90]. Un domaine qui touche également à la représentation visuelle de programmes est le problème de la conception de «profilers», voir par exemple [65]. Cependant, à la différence de ces travaux lorsqu'il abordent la notion de graphe d'exécution, nous n'essayons pas de représenter le graphe d'exécution «en dur», sur un écran d'ordinateur et/ou dans la mémoire de l'ordinateur : nous nous intéressons davantage à la façon dont ce graphe peut être visualisé mentalement, comme support à la compréhension d'une description textuelle (le manuel de référence) de la sémantique d'un langage de programmation. Bien entendu, des dessins de parties de graphes d'exécution pourront apparaître dans le manuel de référence pour faciliter l'exposition de la sémantique du langage.

Nous devons signaler que la notion d'aspect graphique d'un objet mathématique est un sujet délicat. Muriel Lefebvre montre dans [66] comment les mathématiciens sont circonspects sur la question de l'aspect graphique des objets mathématiques qu'ils manipulent. Nous espérons que la notion d'aspect graphique d'une catégorie ne prête cependant pas trop à controverse. Par exemple, une notion de base de la théorie des catégories est celle de diagramme commutatif. Un diagramme commutatif sert à énoncer l'égalité de plusieurs *chemins* joignants deux objets. L'aspect graphique des diagrammes en fait un outil de choix dans les preuves utilisant les catégories, et est par là un facteur important de l'efficacité de l'outil catégorique.

Les méthodes formelles, la documentation, et le programmeur

Il nous faut préciser le statut d'une description formelle avec la logique de réécriture, par rapport à la documentation du langage. En effet le bagage mathématique des programmeurs et implanteurs de langages ne leur permet pas *a priori* d'appréhender facilement une description rédigée avec une méthode formelle. La logique de réécriture, par exemple, suppose comme pré-requis la théorie des catégories et la théorie des modèles, de sorte qu'une description en logique de réécriture serait de peu d'utilité au programmeur ou à l'implanteur du compilateur.

Nous avons mentionné la documentation de ML [79, 78], rédigée avec la sémantique opérationnelle structurée : ce mode d'exposition, bien que formel, ne suppose aucun pré-requis mathématique en-dehors de la notion de règle d'inférence, de sorte que cette documentation est a priori accessible à tout programmeur motivé. Un autre exemple de méthode formelle accessible est la Notation d'Actions ('Action Notation'), mentionnée dans la section précédente. La sémantique opérationnelle de la Notation d'Actions est donnée elle-même au moyen de la sémantique opérationnelle structurée, dans le livre [84]. Or il est peu probable que les programmeurs achèteront ce document afin de comprendre la description du langage : l'intérêt même de la notation d'actions est en effet d'être suffisamment intuitive pour que la description en notation d'actions décrive efficacement la sémantique du langage.

Ainsi, si nous voulions utiliser la logique de réécriture pour la documentation d'un langage, il faudrait procéder comme pour la notation d'actions, c'est-à-dire distinguer notation accessible et sémantique «pure», et concevoir une notation «interface» aux descriptions en logique de réécriture : ce serait alors la traduction de la description en logique de réécriture en une version accessible qui serait présentée au programmeur.

Cette idée est rentrée en compte dans notre choix de la logique de réécriture : l'aspect graphique des modèles de la logique de réécriture permettrait à une notation d'inferfaçage de faire appel à des métaphores d'ordre visuel.

2.2 Rappel sur la théorie des catégories et la logique de réécriture

Dans la littérature sur la logique de réécriture, l'accent est davantage mis sur l'aspect de réécriture que sur les modèles. Or nous ne faisons pas appel à la réécriture (à part, «en passant», dans les sections 2.7.2 et 3.1.11 ci-dessous), et mettons en revanche l'accent sur les modèles. Aussi présentons-nous ci-dessous la logique de réécriture directement à partir de la notion de modèle. Comme ces modèles utilisent des notions de la théorie des catégories, nous faisons d'abord un rappel sur ces notions.

Cependant, ce chapitre et le suivant peuvent être compris sans une appréhension parfaite des notions catégoriques : la notion de modèle y est plus importante que celle de catégorie, et les noms d'opération choisis dans les spécifications, ainsi que les diagrammes explicatifs produits, doivent permettre de comprendre les idées véhiculées sans passer forcément par une vision parfaite des modèles catégoriques.

2.2.1 La théorie des catégories

Les modèles de la logique de réécriture, sont des familles de *catégories*. Les catégories, avec les notions qui s'y rattachent (dont celles de foncteur et de transformations naturelle), fournissent un langage mathématique distinct de la théorie des ensembles : tandis qu'un ensemble est défini par les objets qu'ils contient, un objet dans une catégorie est défini par ses relations aux autres objets de la catégorie.

La théorie des catégorie est aujourd'hui de plus en plus utilisée en informatique théorique ; un aperçu en est donné dans [89], et une motivation générale par [33]. Le texte de référence pour la théorie des catégories est [64]. Le livre [41] fournit une introduction progressive aux catégories, à partir des ensembles.

Les notions de catégories, foncteurs, et transformation naturelle qui sont utilisées par les modèles de la logique de réécriture sont des notions de *structure*. Aussi, elles peuvent être données en quelques définitions et axiomes. Cependant, afin de motiver ces structures abstraites, nous avons rajouté quelques exemples.

Catégories

Une **catégorie** C est constituée de :

1. une collection d'**objets**.
2. une collection de **flèches**. Chaque flèche f a une **source** et un **but**, qui sont chacun un objet de C. On note $f : a \to b$ lorsque la source de f est a, et son but b.
3. Si $f : a \to b$ et $g : b \to c$, alors il existe dans C une flèche $f; g : a \to c$, appelée la **composée** de f et g.

4. Pour tout objet a de C, il existe une flèche $1_a : a \to a$, appelée la flèche **identité** de a.

En outre, les propriétés suivantes sont satisfaites par la catégorie C :

$$1_a; f = f; 1_b = f \quad \text{pour toute flèche } f : a \to b$$

et

$$f; (g; h) = (f; g); h \quad \text{si } f : a \to b, g : b \to c \text{ et } h : c \to d \ .$$

Quelques exemples de catégories sont :

- **Set**. Ses objets sont les ensembles, et ses flèches les fonctions entre ces ensembles.
- **Vct** a pour objets les espaces vectoriels, et pour flèches les transformations linéaires entre eux.
- **Top** a pour objets les espaces topologiques, et pour flèches les fonctions continues entre eux.
- **Grp** a pour objets les groupes, et pour flèches les morphismes de groupes.

Foncteurs

Un **foncteur** est un morphisme de catégories. Il s'applique sur les objets et les flèches de la catégorie de départ, et préserve la structure de cette catégorie dans la transformation. Plus précisément, un foncteur F de la catégorie C vers la catégorie D (ce qui est noté : $F : C \to D$) associe à tout objet a de C un objet Fa dans D, et à toute flèche $f : a \to b$ dans C une flèche $Ff : Fa \to Fb$ dans D (c'est-à-dire que la source et le but sont préservés dans la transformation). Les flèches identité sont préservées : pour tout objet a de C, on a $F1_a = 1_{Fa}$. Et la composition des flèches est également préservée : $F(f; g) = Ff; Fg$.

Des exemples de foncteurs sont :

- Le foncteur «puissance» $\mathcal{P}$: **Set** $\to$ **Set**, qui à tout ensemble X associe l'ensemble de ses parties $\mathcal{P}X$. A toute fonction $f : X \to Y$ il fait correspondre la fonction $\mathcal{P}f : \mathcal{P}X \to \mathcal{P}Y$, qui associe à tout sous-ensemble $S \subseteq X$ son image $fS \subseteq Y$.
- Beaucoup de foncteurs consistent à «oublier» une structure. Ils sont appelés «foncteurs oubli». Un exemple est le foncteur oubli U : **Grp** $\to$ **Set**, qui associe à tout groupe G l'ensemble de ses éléments («oubliant» la multiplication entre éléments, et donc la structure de groupe), et qui associe à tout morphisme de groupe $f : G \to G'$ la même fonction f (qui devient simplement une fonction entre ensembles).

Un autre exemple intéressant de foncteur est donné par les *hom-foncteurs*. Dans la théorie des catégories, un objet n'a pas de caractéristiques «en lui-même» : ce sont les flèches qui en partent (vers d'autres objets de la catégorie) qui définissent la particularité de cet objet par rapport aux autres objets de la catégorie. Aussi, il est courant d'utiliser l'équivalence, étant donnée une catégorie C, entre tout objet a de C et son *hom-foncteur* $\mathrm{hom}(a, -) : C \to \mathbf{Set}$, qui à tout objet b de C associe l'ensemble $\mathrm{hom}(a, b)$ des flèches de a vers b. Ce foncteur fait correspondre à toute flèche $k : b \to b'$ de C la fonction $\mathrm{hom}(a, k) : \mathrm{hom}(a, b) \to \mathrm{hom}(a, b')$, qui envoie toute flèche $f \in \mathrm{hom}(a, b)$ sur la flèche $f; k \in \mathrm{hom}(a, b')$. L'équivalence entre le foncteur $\mathrm{hom}(a, -)$ et a lui-même est formulée de façon rigoureuse par le fameux *lemme de Yoneda* [64, p. 61]. En termes informatiques, ce lemme signifie que si les flèches partant de a sont vues comme des *observations* de a par les autres objets, alors l'objet a peut être défini (à isomorphisme près) à partir uniquement de ces observations. C'est pourquoi la théorie des catégories a pu être appelée «les mathématiques orientées objet».

Composition de foncteurs Les foncteurs peuvent être composés. Si $F : C \to D$ et $G : D \to E$ sont deux foncteurs, alors on a le foncteur $F; G : C \to E$, qui a tout objet a de C associe l'objet $G(Fa)$ de E, et à toute flèche f de C associe la flèche $G(Ff)$ de E. La composition de foncteurs est associative. Cela nous permet de donner un nouvel exemple de catégorie : si l'on considère pour toute catégorie C son foncteur identité $I_C : C \to C$, on peut considérer la catégorie **Cat** de toutes les (petites) catégories, avec les foncteurs pour flèches. (Ici, «petites» sert à éviter des problèmes de fondations : **Cat** n'est pas considérée comme une petite catégorie, et ne peut donc pas se contenir elle-même. Pour les aspects fondationnels de la théorie des catégories, voir [64, ch. I, sec. 6].)

Transformations naturelles

Une **transformation naturelle** entre deux foncteurs est une transformation qui préserve la «structure de foncteur». Plus précisément, si F et G sont deux foncteurs de C vers D, alors une transformation naturelle $\eta : F \dot{\to} G$ de F vers G est une fonction qui à tout objet a de C associe une flèche $\eta_a : Fa \to Ga$ dans D, de telle sorte que pour toute flèche $f : a \to b$ de C, le diagramme suivant commute :

$$\begin{tikzcd} Fa \arrow[r, "Ff"] \arrow[d, "\eta_a"'] & Fb \arrow[d, "\eta_b"] \\ Ga \arrow[r, "Gf"'] & Gb \end{tikzcd}$$

c'est-à-dire que l'on ait : $Ff; \eta_b = \eta_a; Gf$. Pour prendre une image, si F et G sont vus comme des dessins différents de C dans D, alors η «fait glisser» le dessin F vers le dessin G.

Pour un exemple de transformation naturelle, considérons à nouveau les hom-foncteurs que nous avons définis plus haut. Considérons une catégorie C, et dans cette catégorie une flèche $f : a \to b$. Nous avons dit que les foncteurs $\mathrm{hom}(a, -)$ et $\mathrm{hom}(b, -)$ pouvaient être considérés comme équivalents à a et b eux-mêmes. Alors quel est l'équivalent de f au niveau des hom-foncteurs ? Il s'agit de la transformation naturelle $\mathrm{hom}(f, -) : \mathrm{hom}(b, -) \stackrel{.}{\to} \mathrm{hom}(a, -)$, qui à tout objet c de C associe la fonction $\mathrm{hom}(f, c) : \mathrm{hom}(b, c) \to \mathrm{hom}(a, c)$, laquelle envoie toute flèche $k : b \to c$ de $\mathrm{hom}(b, c)$ sur la flèche $f; k$ de $\mathrm{hom}(a, c)$. En d'autres termes, $\mathrm{hom}(f, -)$ fait glisser l'hom-foncteur de a vers celui de b, *suivant* f.

Produits de catégories

Il existe de nombreuses constructions utiles de nouvelles catégories à partir de catégories. Nous présentons ici une de ces constructions, qui est utilisée dans les modèles de la logique de réécriture : le produit de deux catégories.

À partir de deux catégories C et D, il est possible de construire la catégorie *produit* de C et D, qui est notée $C \times D$. Les objets de $C \times D$ sont des couples $\langle c, d\rangle$, où c est un objet de C et d un objet de D. Une flèche d'un objet $\langle c, d\rangle$ vers un autre objet $\langle c', d'\rangle$ est un couple $\langle f, g\rangle$, avec $f : c \to c'$ une flèche de C et $g : d \to d'$ une flèche de D. La composition de deux flèches $\langle f, g\rangle : \langle c, d\rangle \to \langle c', d'\rangle$ et $\langle f', g'\rangle : \langle c', d'\rangle \to \langle c >>, d >>\rangle$ est définie comme étant la flèche $\langle f; f', g; g'\rangle$.

2.2.2 La logique de réécriture et ses modèles

À présent que nous avons présenté la théorie des catégories, nous pouvons préciser ce qu'est une théorie de réécriture, en décrivant ses modèles catégoriques.

La présentation des spécifications Les spécifications par lesquelles nous décrivons ci-dessous les sémantiques officielle et effective du langage ATL sont données avec la notation **CafeOBJ** [21]. Cette notation est proche de celle du fameux système OBJ [40], et est associée au système «de qualité industrielle» CafeOBJ [86]. Or, de même que dans un cadre précis d'applications, l'utilisation d'un langage spécialisé permet d'écrire les programmes de façon plus concise qu'avec un langage général, il serait possible, pour définir la sémantique de vrais langages, de concevoir une notation spécialisée pour notre utilisation de la logique de réécriture. Nous conservons cependant ici la notation générale CafeOBJ afin de pouvoir mieux expliquer, en étant plus proche de la forme «pure» d'une spécification RWL et en utilisant une notation connue, la sémantique de nos descriptions.

Exemple : les Lyumines

Afin de rendre l'exposition plus digeste, nous utilisons un exemple de théorie de réécriture. Nous spécifions les *Lyumines* dans la notation CafeOBJ.

Pour aider à la lecture du texte de la théorie, nous indiquons ce que sont censés être les Lyumines, car les modèles eux-mêmes ne disent pas ce que sont les Lyumines : ils précisent seulement une structure concernant l'univers des Lyumines. Les Lyumines sont imaginés par l'auteur du présent exemple comme de petits êtres vivant dans les forêts de la Sibérie orientale. Leur nom est dérivé des *Moumines* ('Moomins' en anglais) de l'écrivain finlandais Tove Jansson. Le préfixe «Lyu» a été choisi pour donner au nom une consonance slave. Contrairement aux Schtroumpfs de Peyo, les Lyumines n'ont pas une couleur constante : ils peuvent changer de couleur. Nous avons ajouté une opération `couleurPréférée` qui indique quelle est la couleur préférée d'un Lyumine. Celle-ci varie avec la couleur du Lyumine.

```
module LYUMINE {
  [ Lyumine ]
  [ NomDeLyumine ] [ Couleur ]

  ops Clara Igor          : -> NomDeLyumine
  ops magenta jaune vert  : -> Couleur

  op <__> : NomDeLyumine Couleur -> Lyumine

  var unNom : NomDeLyumine
  trans < unNom jaune   > => < unNom magenta > .
  trans < unNom magenta > => < unNom vert    > .

  op couleurPréférée_ : Lyumine -> Couleur
  eq couleurPréférée < unNom jaune   > = magenta .
  eq couleurPréférée < unNom magenta > = vert .
  eq couleurPréférée < unNom vert    > = jaune .
}
```

Nous commentons brièvement ce texte CafeOBJ. La signification des mots que nous écrivons en italique est précisée dans la section ci-après. Cette théorie déclare trois *sortes* : `Lyumine`, `NomDeLyumine`, et `Couleur`. Une déclaration de sorte s'écrit en CafeOBJ entre crochets ; par exemple, «`[ Couleur ]`». Sept *opérations* sont déclarées en tout : `Clara`, `Igor`, `jaune`, `vert`, `magenta`, `<__>`, et `couleurPréférée`. Les deux premières servent à définir des noms de Lyumines, et les trois suivantes à déclarer des couleurs. L'opération `<__>` prend deux arguments : un nom de Lyumine et une couleur. Elle nous sert à identifier un Lyumine : le nom de Lyumine est celui attribué à ce Lyumine, et la couleur représente la couleur de ce Lyumine.

Deux *transitions* sont déclarées (avec le mot-clé `trans`). La première déclare que tout Lyumine jaune peut devenir magenta, la seconde que tout Lyumine magenta peut devenir vert. Enfin, trois équations (introduites avec le mot-clé `eq`) indiquent la couleur préférée des Lyumines suivant leur couleur.

Les modèles

La sémantique d'une théorie de réécriture est donnée par l'ensemble de ses modèles. Nous présentons ici seulement la sémantique pour les théories sans équations ou transitions conditionnelles. Les modèles pour les cas avec conditionnelles sont exposés dans [75] : l'exposé est technique et nous préférons ne pas le reproduire ici.

Pour présenter les modèles, nous exposons la relation entre une théorie de réécriture et un modèle (quelconque) de cette théorie. Ce modèle *satisfait* les contraintes énoncées par la théorie. Soit $[\![.]\!]$ la fonction qui définit l'interprétation des contraintes de la théorie par ce modèle.

Une théorie de réécriture est constituée de :

- *sortes.* Une sorte s est un nom, et dans le modèle cela correspond à une *catégorie* $[\![s]\!]$. Dans un modèle de la théorie `LYUMINE`, on aura donc au moins les trois catégories $[\![\texttt{Lyumine}]\!]$, $[\![\texttt{NomDeLyumine}]\!]$ et $[\![\texttt{Couleur}]\!]$.

- Des *opérations.* Une opération σ est un nom, et a une *arité* $s_1 \dots s_n, s$, où s_1, …, s_n et s sont des sortes de la théorie. L'opération correspond, dans le modèle, à un *foncteur* de la catégorie produit $[\![s_1]\!] \times \cdots \times [\![s_n]\!]$ vers la catégorie $[\![s]\!]$. Une opération σ avec une seule sorte pour arité, comme l'opération `Igor` de la théorie `LYUMINE`, correspond à un foncteur de la catégorie **1** à un objet et une flèche (l'identité de cet objet) vers la catégorie $[\![s]\!]$: cette opération identifie donc un objet de $[\![s]\!]$. Pour simplifier, nous considérons dans la suite qu'une opération σ avec une seule sorte s pour arité identifie directement un objet de $[\![s]\!]$. Par exemple, $[\![\texttt{Igor}]\!]$ est un objet de $[\![\texttt{NomDeLyumine}]\!]$.

- Un ensemble $X = \{x_1, \dots, x_n\}$ de *variables*, qui vont servir dans les équations et les transitions. Une variable est un nom, et une sorte lui est associée. Dans notre exemple, nous avons $X = \{\texttt{unNom}\}$, où `unNom` est de sorte `NomDeLyumine`.

  C'est à partir des opérations et de noms de variables que peuvent être construits des *termes.* Par exemple, «`couleurPréférée < Igor jaune >`» est un terme construit à partir des opérations `couleurPréférée`, `<__>`, `Igor`, et `jaune`. Dans le modèle, il correspond à l'objet $([\![\texttt{<\_\_>}]\!]; [\![\texttt{couleurPréférée}]\!])\langle[\![\texttt{Igor}]\!], [\![\texttt{jaune}]\!]\rangle$ de la catégorie `Couleur`. Les termes peuvent être paramétrés par des variables. Un exemple de terme avec une variable est «`couleurPréférée < unNom jaune >`». L'interprétation de ce terme est un foncteur $([\![\texttt{<\_\_>}]\!]; [\![\texttt{couleurPréférée}]\!])\langle -, [\![\texttt{jaune}]\!]\rangle : [\![\texttt{NomDeLyumine}]\!] \to [\![\texttt{Couleur}]\!]$.

- Des *équations.* Une équation a pour forme $t = t'$, où t et t' sont deux termes construits à partir d'opérations et de noms de variables (pris dans X). Ils ont tous deux la même sorte s. Pour simplifier l'exposition, on considère que t et t' sont chacun paramétrés par toutes les variables de X. Alors, pour le modèle, $[\![t]\!]$ et $[\![t']\!]$ sont tous deux des foncteurs $[\![s_1]\!] \times \cdots \times [\![s_n]\!] \to [\![s]\!]$, où s_1, …, s_n sont les sortes des variables x_1, …, x_n.

Et l'équation signifie que les foncteurs $[\![t]\!]$ et $[\![t']\!]$ sont égaux.
Pour exemple, prenons l'équation

```
eq couleurPréférée < unNom jaune > = magenta .
```

Le premier terme est interprété comme le foncteur
$[\![\texttt{couleurPréférée < unNom jaune >}]\!] = ([\![\texttt{<\_\_>}]\!];[\![\texttt{couleurPréférée}]\!])\langle -, [\![\texttt{jaune}]\!]\rangle :$ $[\![\texttt{NomDeLyumine}]\!] \to [\![\texttt{Couleur}]\!]$. Le deuxième terme est interprété comme le foncteur constant $\overline{[\![\texttt{magenta}]\!]} : [\![\texttt{NomDeLyumine}]\!] \to [\![\texttt{Couleur}]\!]$, qui envoie tout nom de Lyumine sur le même objet $[\![\texttt{magenta}]\!]$. En effet, ce terme est paramétré, comme le premier, par la variable `unNom`, et donc correspond à un foncteur qui a $[\![\texttt{NomDeLyumine}]\!]$ comme domaine : le fait que «`unNom`» n'apparaisse pas dans le terme signifie que le résultat de l'application de ce foncteur à un objet $[\![\texttt{unNom}]\!]$ ne dépend pas de la valeur prise par $[\![\texttt{unNom}]\!]$. Ainsi cette équation établit que dans tout modèle, si un Lyumine est jaune, alors sa couleur préférée est forcément magenta—et ce quel que soit son nom.

– des *transitions*. Une transition a pour forme $r : t \to t'$, où r est un *label*, et où t et t' sont, comme pour les équations, deux termes de même sorte construits sur les variables de X. Comme pour les équations, $[\![t]\!]$ et $[\![t']\!]$ sont tous deux considérés comme des foncteurs $[\![s_1]\!]\times\cdots\times[\![s_n]\!] \to [\![s]\!]$. La transition déclare une *transformation naturelle* $[\![\texttt{r}]\!]$, dans le modèle, de $[\![t]\!]$ vers $[\![t']\!]$. Cela implique l'existence dans $[\![s]\!]$, pour tout objet $\langle y_1, \ldots, y_n\rangle$ de $[\![s_1]\!] \times \cdots \times [\![s_n]\!]$, d'une flèche
$$[\![\texttt{r}]\!]_{\langle y_1,\ldots,y_n\rangle} : [\![t]\!]\langle y_1, \ldots, y_n\rangle \to [\![t']\!]\langle y_1, \ldots, y_n\rangle.$$
Pour exemple, la transition

```
trans < unNom jaune > => < unNom magenta > .
```

(sans label) déclare une transformation naturelle
$$\alpha : [\![\texttt{<\_\_>}]\!]\langle -, [\![\texttt{jaune}]\!]\rangle \stackrel{\cdot}{\to} [\![\texttt{<\_\_>}]\!]\langle -, [\![\texttt{magenta}]\!]\rangle.$$
En conséquence, il existe dans $[\![\texttt{Lyumine}]\!]$, pour tout objet y de $[\![\texttt{NomDeLyumine}]\!]$, une flèche $\alpha_y : [\![\texttt{<\_\_>}]\!]\langle y, [\![\texttt{jaune}]\!]\rangle \to [\![\texttt{<\_\_>}]\!]\langle y, [\![\texttt{magenta}]\!]\rangle$.
L'intention de la spécification `LYUMINE` est de signifier par là que quel que soit le nom d'un Lyumine, si ce Lyumine est jaune alors il peut se transformer («en suivant la flèche») en un Lyumine magenta de même nom.

Modules et théories Le texte CafeOBJ qui décrit les Lyumines est appelé un *module* CafeOBJ. Un module porte un nom (ici, «`LYUMINE`»), et décrit (entre les crochets) une théorie. C'est-à-dire qu'un module CafeOBJ *note* une théorie (qui est un objet mathématique). Dans la suite de cette thèse, nous utiliserons, comme ci-dessus, les vocables «module» ou «théorie» suivant que l'accent est mis sur le texte CafeOBJ ou la théorie elle-même ; mais nous nous permettrons de faire «comme si» un module *était* une théorie, afin de ne pas alourdir le texte avec des formules comme «la théorie qui est notée par le module CafeOBJ de nom `X`». Nous parlerons par exemple de «la théorie `X`».
Une différence notable entre modules et théories est que nous pouvons discuter d'un mo-

dule, comme ci-dessus, à partir de l'ordre, dans le texte du module, des déclarations des opérations, équations, et transitions ; tandis que celles-ci n'ont pas d'ordre dans une théorie, qui est constituée d'un ensemble d'opérations, d'un ensemble d'équations, et d'un ensemble de transitions.

Modèles initiaux

Parmi tous les modèles d'une théorie de réécriture, il est possible de distinguer les modèles dits **initiaux**. Dit simplement, ils contiennent «uniquement ce qui est déclaré par la théorie», et n'interprètent deux termes différents par le même objet que si cela est contraint par les équations. Nous utiliserons principalement ces modèles dans la présente thèse.
Plus précisément, Un modèle M d'une théorie de réécriture $\mathcal{R}$ est dit *initial* si :

- chacune de ses catégories est l'interprétation d'une sorte déclarée dans $\mathcal{R}$;
- chacun de ses foncteurs est l'interprétation d'une opération déclarée dans $\mathcal{R}$;
- dans une catégorie $\llbracket s \rrbracket$, tout objet est l'image par un foncteur d'un objet d'une autre catégorie, ou l'interprétation d'une opération d'arité 1 (comme `Clara` dans `LYUMINE`) ;
- dans une catégorie $\llbracket s \rrbracket$, toute flèche est de type $\llbracket r \rrbracket_a : \llbracket t \rrbracket a \rightarrow \llbracket t' \rrbracket a$, où $r : t \rightarrow t'$ est une transition de $\mathcal{R}$.
- lorsqu'aucune équation ne contraint l'égalité de deux objets, alors ils sont distincts.

Les quatre premières conditions définissent un modèle *engendré*. La dernière fait de ce modèle un modèle *typique*. Finalement, ce modèle étant engendré et typique, il est initial.

Par exemple un modèle initial de `LYUMINE` consistera en :

- trois catégories : $\llbracket \texttt{Lyumine} \rrbracket$, $\llbracket \texttt{NomDeLyumine} \rrbracket$, et $\llbracket \texttt{Couleur} \rrbracket$.
- Deux foncteurs : $\llbracket \texttt{<\_\_>} \rrbracket : \llbracket \texttt{NomDeLyumine} \rrbracket \times \llbracket \texttt{Couleur} \rrbracket \rightarrow \llbracket \texttt{Lyumine} \rrbracket$
et $\llbracket \texttt{couleurPréférée} \rrbracket : \llbracket \texttt{Lyumine} \rrbracket \rightarrow \llbracket \texttt{Couleur} \rrbracket$.
- Dans $\llbracket \texttt{NomDeLyumine} \rrbracket$, deux objets : $\llbracket \texttt{Clara} \rrbracket$ et $\llbracket \texttt{Igor} \rrbracket$.
- Dans $\llbracket \texttt{Couleur} \rrbracket$, trois objets : $\llbracket \texttt{magenta} \rrbracket$, $\llbracket \texttt{jaune} \rrbracket$, et $\llbracket \texttt{vert} \rrbracket$.
- Dans $\llbracket \texttt{Lyumine} \rrbracket$, six objets :
$\llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket, \llbracket \texttt{magenta} \rrbracket \rangle$, $\llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket, \llbracket \texttt{jaune} \rrbracket \rangle$, $\llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket, \llbracket \texttt{vert} \rrbracket \rangle$,
$\llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket, \llbracket \texttt{magenta} \rrbracket \rangle$, $\llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket, \llbracket \texttt{jaune} \rrbracket \rangle$, et $\llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket, \llbracket \texttt{vert} \rrbracket \rangle$.
- Dans $\llbracket \texttt{Lyumine} \rrbracket$, quatre flèches :

$$
\begin{array}{lclcl}
\alpha_{\llbracket \texttt{Clara} \rrbracket} & : & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket \ , \llbracket \texttt{jaune} \rrbracket \rangle & \rightarrow & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket, \llbracket \texttt{magenta} \rrbracket \rangle, \\
\alpha_{\llbracket \texttt{Igor} \rrbracket} & : & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket, \llbracket \texttt{jaune} \rrbracket \rangle & \rightarrow & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket, \llbracket \texttt{magenta} \rrbracket \rangle, \\
\beta_{\llbracket \texttt{Clara} \rrbracket} & : & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket \ , \llbracket \texttt{magenta} \rrbracket \rangle & \rightarrow & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Clara} \rrbracket, \llbracket \texttt{vert} \rrbracket \rangle, \\
\beta_{\llbracket \texttt{Igor} \rrbracket} & : & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket \ , \llbracket \texttt{magenta} \rrbracket \rangle & \rightarrow & \llbracket \texttt{<\_\_>} \rrbracket \langle \llbracket \texttt{Igor} \rrbracket, \llbracket \texttt{vert} \rrbracket \rangle \ .
\end{array}
$$

Ainsi, dans ce modèle, comme les seules flèches sont celles qui sont déclarées, il n'y a pas de flèche permettant à un Lyumine vert de changer de couleur. Donc, même si sa couleur préférée est alors le jaune et que, suppose-t-on, il aimerait redevenir jaune, aucune flèche ne le lui permet.

Tous les modèles initiaux d'une théorie sont isomorphes (au sens des spécifications algébriques, mais aussi au sens catégorique). En conséquence, dans la suite de cette thèse, nous parlerons souvent, «du» modèle initial d'une théorie, sachant que celui-ci est défini à isomorphisme près.

CafeOBJ permet de signaler lorsque la sémantique initiale est requise pour une théorie : le module est déclaré par le mot-clé «`module !`»—avec un point d'exclamation à la fin. Ainsi, pour signaler que nous requérons la sémantique initiale pour la théorie `LYUMINE`, nous aurions écrit : «`module! LYUMINE { ... }`».
De même, lorsque la sémantique requise n'est pas la sémantique initiale (elle est alors dite **lâche**), cela peut être déclaré par le mot-clé «`module*`». Dans les spécifications ci-dessous, nous signalerons lorsque cela sera approprié si la sémantique requise est initiale ou lâche—et justifierons dans le cas d'une sémantique lâche.

Structuration des spécifications

Nous structurons la spécification qui va suivre en plusieurs modules, qui correspondent à plusieurs théories de réécriture. Ces modules sont agencés entre eux par des *imports* de modules. Ceux-ci peuvent être de trois types : «`protecting`», «`extending`», ou «`using`», ce qui se déclare comme suit : dans le cas d'un module `B` important un module `A` :

```
module B {
  protecting (A)
  ...
}

module B {
  extending (A)
  ...
}
```

ou

```
module B {
  using (A)
  ...
}
```

Au niveau des modèles, cela signifie :

- pour «`protecting`» : que les modèles de `A` deviennent partie des modèles de `B`.
- pour «`extending`» : comme pour 'protecting', cela correspond à une importation de modèles : mais ici les modèles de `A` peuvent être *étendus* par la spécification `B` : i.e., celle-ci peut déclarer de nouveaux objets pour les sortes de la théorie `A`, ou de nouvelles opérations sur ces objets.
- pour «`using`» : que les modèles de `A` sont «importés», mais, de même qu'ils peuvent être étendus, il se peut que certains objets des sortes de `A` soient «fusionnés» (avec des équations dans le module `B`). Donc en fait, il ne s'agit dans ce cas pas tant d'une importation des modèles de `A` qu'une réutilisation du texte du module `A`—une sorte de «`#include`».

Lors de hiérarchies d'importation—par exemple si un module `C` importe également le module `A` ci-dessus, et qu'un module `D` importe `B` et `C`—les modules communs et leurs modèles associés sont partagés, si toutefois les dits modèles sont identiques. Par exemple, si `B` importe `A` par `protecting` tandis que `C` l'importe par `using`, il y aura dans `D` deux «versions» de `A`, au lieu d'une.

Modèles et réécriture

Il est possible d'utiliser la réécriture de termes syntaxiques pour tester les spécifications RWL, lorsque celles-ci ont une certaine forme. C'est-à-dire que les équations doivent pouvoir correspondre, en étant lues de gauche à droite, comme des règles de réécriture de termes syntaxiques paramétrées par des variables instanciées au moment de la réécriture.

Toutes les théories de réécriture ne sont pas exécutables dans ce sens. La théorie `Lyumines` l'est, comme la plupart de celles données dans la suite de ce chapitre et dans le chapitre suivant. Nous ne détaillons pas ici les rapports entre les modèles et la réécriture, car le sujet est technique [111], et nous nous intéressons davantage aux modèles qu'à la réécriture.

Cependant, comme nous ferons tout de même appel à la réécriture dans les sections 2.7.2 et 3.1.11, nous donnons un exemple de réécriture, avec un test du module `Lyumines` utilisant le système CafeOBJ :

```
open LYUMINE
reduce couleurPréférée < Igor jaune > .
execute < Igor jaune > .
execute couleurPréférée < Igor jaune > .
close
```

Exécuter ce test avec CafeOBJ produit le résultat suivant :

```
processing input : ./test-lyumine.mod
-- opening module LYUMINE.. done.
-- reduce in %LYUMINE : couleurPréférée < Igor jaune >
```

```
magenta : Couleur
(0.000 sec for parse, 1 rewrites(0.040 sec), 1 matches)
-- execute in %LYUMINE : < Igor jaune >
< Igor vert > : Lyumine
(0.000 sec for parse, 2 rewrites(0.000 sec), 7 matches)
-- execute in %LYUMINE : couleurPréférée < Igor jaune >
jaune : Couleur
(0.000 sec for parse, 3 rewrites(0.000 sec), 10 matches)
CafeOBJ>
```

La commande «`reduce`» demande au système d'exécuter en utilisant seulement les équations, tandis que «`execute`» fait utiliser également les transitions. C'est pourquoi la dernière requête diffère de la première : le résultat de la première indique la couleur préférée d'Igor jaune, tandis que la dernière fait d'abord transiter Igor de jaune à magenta puis de magenta à vert, avant d'utiliser l'équation qui indique que la couleur préférée d'un Lyumine vert est le jaune. Le fait que les transitions soient utilisées avant l'équation provient de ce que CafeOBJ utilise par défaut une stratégie *affairée* : ici, l'argument de «`couleurPréférée`» est évalué avant que le terme entier `couleurPréférée < Igor jaune >` le soit : «`couleurPréférée`» est donc en fait appliqué sur l'argument `< Igor vert >`. Cette stratégie par défaut sera utilisée dans les sections 2.7.2 et 3.1.11.

Associativité et élément neutre pour une opération

Un foncteur qui a pour domaine un produit de catégories est appelé un **bifoncteur**. Un exemple est donné, pour les Lyumines, par l'opération `et` dans l'extension suivante de la spécification des Lyumines, qui définit les groupes de Lyumines :

```
module LYUMINES {
  protecting (LYUMINE)
  [ Lyumines ]
  [ Lyumine < Lyumines ]
  op groupeVide : -> Lyumines
  op (_et_) : Lyumines Lyumines -> Lyumines  { assoc id: groupeVide comm }
}
```

`Lyumines` définit la sorte des groupes de Lyumines. `Lyumine < Lyumines` déclare tout Lyumine comme étant déjà un groupe (à 1 individu). En effet, cette déclaration d'un ordre entre les sortes `Lyumine` et `Lyumines` signifie au niveau des modèles, que $[\![\texttt{Lyumine}]\!]$ est une sous-catégorie de $[\![\texttt{Lyumines}]\!]$. C'est-à-dire que tout objet a de $[\![\texttt{Lyumine}]\!]$ est un objet de $[\![\texttt{Lyumines}]\!]$, et que toute flèche $f : a \to b$ de $[\![\texttt{Lyumine}]\!]$ est également une flèche $f : a \to b$ de $[\![\texttt{Lyumines}]\!]$.

Le foncteur $[\![\texttt{et}]\!] : [\![\texttt{Lyumines}]\!] \times [\![\texttt{Lyumines}]\!] \to [\![\texttt{Lyumines}]\!]$ est un bifoncteur. Il sert à construire des groupes de Lyumines, comme la paire «`< Clara magenta > et < Igor jaune >`». Les attributs «`assoc`» et «`id`» donnés dans la spécification pour l'opération «`et`» signifient que le bifoncteur $[\![\texttt{et}]\!]$ est associatif et a pour élément neutre le groupe vide de lyumines, `groupeVide`. La

commutativité («`comm`») déclare la commutativité de ce bifoncteur : il sert à préciser que pour les groupes de Lyumines, l'ordre (donné par le couple d'arguments de $\llbracket \texttt{et} \rrbracket$) ne doit pas compter :
$\llbracket \texttt{< Clara magenta > et < Igor jaune >} \rrbracket$ et
$\llbracket \texttt{< Igor jaune > et < Clara magenta >} \rrbracket$ doivent être le même objet. De même, l'attribut d'associativité sert à indiquer qu'un groupe de Lyumines ne dépend pas de la façon dont il peut être décomposé en paires de groupes : l'attribut déclare que pour tous objets a, b, c de $\llbracket \texttt{Lyumine} \rrbracket$, on a : $\llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{et} \rrbracket \langle a, b \rangle, c \rangle = \llbracket \texttt{et} \rrbracket \langle a, \llbracket \texttt{et} \rrbracket \langle b, c \rangle \rangle$. Cela permet d'écrire sans ambiguïté en CafeOBJ des termes comme «`< Clara magenta > et < Igor jaune > et < Alexander vert >`» (si `Alexander` est une opération d'arité `NomDeLyumine`).

2.2.3 La concurrence dans la logique de réécriture

Nous avons indiqué dans la section 1.3.6 que la logique de réécriture a été présentée comme permettant de modéliser la notion de concurrence. Nous expliquons ici pourquoi nous n'utilisons pas dans la suite cet aspect de la logique de réécriture, alors que nous aurons à modéliser la notion de concurrence.

Transitions concurrentes

Pour exposer tout d'abord brièvement ce qu'est la concurrence dans la logique de réécriture, nous reprenons l'exemple des Lyumines, avec la théorie `LYUMINES` ci-dessus.

La concurrence y concerne la transformation de couleur des Lyumines dans un groupe. Le fait que, dans le groupe $\llbracket \texttt{< Clara magenta > et < Igor jaune >} \rrbracket$, Clara puisse devenir verte en même temps qu'Igor devient lui-même magenta est représenté par la flèche
$\llbracket \texttt{et} \rrbracket \langle \beta_{\llbracket \texttt{Clara} \rrbracket}, \alpha_{\llbracket \texttt{Igor} \rrbracket} \rangle$:
$\llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{< Clara magenta >} \rrbracket, \llbracket \texttt{< Igor jaune >} \rrbracket \rangle \to \llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{< Clara vert >} \rrbracket, \llbracket \texttt{< Igor magenta >} \rrbracket \rangle$
dans $\llbracket \texttt{Lyumines} \rrbracket$.

Par contraste, le changement de couleur de Clara seule est représenté par la flèche
$\llbracket \texttt{et} \rrbracket \langle \beta_{\llbracket \texttt{Clara} \rrbracket}, 1_{\llbracket \texttt{< Igor jaune >} \rrbracket} \rangle$:
$\llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{< Clara magenta >} \rrbracket, \llbracket \texttt{< Igor jaune >} \rrbracket \rangle \to \llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{< Clara vert >} \rrbracket, \llbracket \texttt{< Igor jaune >} \rrbracket \rangle$.

Le changement de couleur d'Igor quant Clara est devenue verte est donné par la flèche
$\llbracket \texttt{et} \rrbracket \langle 1_{\llbracket \texttt{< Clara vert >} \rrbracket}, \alpha_{\llbracket \texttt{Igor} \rrbracket} \rangle$:
$\llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{< Clara vert >} \rrbracket, \llbracket \texttt{< Igor jaune >} \rrbracket \rangle \to \llbracket \texttt{et} \rrbracket \langle \llbracket \texttt{< Clara vert >} \rrbracket, \llbracket \texttt{< Igor magenta >} \rrbracket \rangle$.

La troisième possibilité est qu'Igor change d'abord de couleur, en devenant magenta ; et qu'ensuite seulement Clara change de couleur, en devenant verte. Les trois possibilités peuvent être récapitulées avec le diagramme suivant, de la catégorie $\llbracket \texttt{Lyumines} \rrbracket$:

$$\begin{tikzcd}
 & z_0 \arrow[dl, "{\llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ jaune >} \rrbracket} \rangle}"'] \arrow[dd, "{\llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle}" description] \arrow[dr, "{\llbracket \mathtt{et} \rrbracket \langle 1_{\llbracket \mathtt{< Clara\ magenta >} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle}"] & \\
z_1 \arrow[dr, "{\llbracket \mathtt{et} \rrbracket \langle 1_{\llbracket \mathtt{< Clara\ vert >} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle}"'] & & z_2 \arrow[dl, "{\llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ magenta >} \rrbracket} \rangle}"] \\
 & z_3 &
\end{tikzcd}$$

où :

$z_0 = \llbracket$`< Clara magenta > et < Igor jaune >`$\rrbracket$,
$z_1 = \llbracket$`< Clara vert > et < Igor jaune >`$\rrbracket$,
$z_2 = \llbracket$`< Clara magenta > et < Igor magenta >`$\rrbracket$, et
$z_3 = \llbracket$`< Clara vert > et < Igor magenta >`$\rrbracket$.

La littérature sur la logique de réécriture indique que le rôle des modèles est «[to make] mathematically precise the idea [...] that the models of rewriting logic theories are concurrent systems» [72]. Pour cela, la logique de réécriture interprète les objets comme des états, et les flèches comme des transitions entre états. La composition de flèches est quant à elle interprétée comme codant une séquence dans le temps.

Au niveau des modèles, de par la fonctorialité de $\llbracket \mathtt{et} \rrbracket$, le diagramme ci-dessus *commute*. C'est-à-dire que l'on a :

$$\begin{array}{ll}
 & \llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ jaune >} \rrbracket} \rangle ; \llbracket \mathtt{et} \rrbracket \langle 1_{\llbracket \mathtt{< Clara\ vert >} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle \\
= & \llbracket \mathtt{et} \rrbracket \langle 1_{\llbracket \mathtt{< Clara\ magenta >} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle ; \llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ magenta >} \rrbracket} \rangle \\
= & \llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle
\end{array}$$

En effet :

$$\begin{array}{lll}
 & \llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ jaune >} \rrbracket} \rangle ; \llbracket \mathtt{et} \rrbracket \langle 1_{\llbracket \mathtt{< Clara\ vert >} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle & \\
= & \llbracket \mathtt{et} \rrbracket (\langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ jaune >} \rrbracket} \rangle ; \langle 1_{\llbracket \mathtt{< Clara\ vert >} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle) & \text{(par la fonctorialité de } \llbracket \mathtt{et} \rrbracket \text{)} \\
= & \llbracket \mathtt{et} \rrbracket (\langle \beta_{\llbracket \mathtt{Clara} \rrbracket} ; 1_{\llbracket \mathtt{< Clara\ vert >} \rrbracket}, 1_{\llbracket \mathtt{< Igor\ jaune >} \rrbracket} ; \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle & \text{(d'après la définition} \\
 & & \text{des catégories produits)} \\
= & \llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle & \text{(propriétés} \\
 & & \text{des flèches identité)}
\end{array}$$

et de même pour l'égalité de la composition des flèches du chemin de droite avec $\llbracket \mathtt{et} \rrbracket \langle \beta_{\llbracket \mathtt{Clara} \rrbracket}, \alpha_{\llbracket \mathtt{Igor} \rrbracket} \rangle$.

Cela signifie que les deux transitions séquentielles et la transition concurrente sont en fait représentées par une seule flèche. La littérature explique que cela permet de s'abstraire de l'ordre dans lequel les transitions de parties d'un état sont effectuées, de façon à obtenir la notion recherchée de «true concurrency» [75].

Discussion

Or la composition de flèches était censée coder la notion de séquence. Le fait que les flèches composées soient strictement égales revient donc à nier les différences entre les différentes séquences possibles, et donc à nier la notion même de séquence que ces flèches étaient censées représenter. Mais si la notion de séquence n'est pas représentée dans la catégorie, quelle est la notion de concurrence représentée par la catégorie ? En effet, il est difficile de concevoir la notion de concurrence en l'absence de celle de séquence. Ainsi, la signification opérationnelle de la «true concurrency» n'est pas claire.

En conséquence, dans la suite de cette thèse, lorsque nous souhaiterons coder une transition concurrente, nous le ferons avec une seule flèche, sans introduire de chemins séquentiels «alternatifs», puisque la sémantique de la logique de réécriture a pour objet même, par la notion de «true concurrency», de nier la différence entre les chemins alternatifs.

Un avantage direct du fait d'éviter de définir des chemins alternatifs est que la composition peut vraiment être interprétée comme représentant la séquence dans le temps des transitions représentées par les flèches qui la composent. Cela va nous être utile puisque nous souhaitons spécifier des graphes d'exécution : nous avons besoin que les flèches qui se suivent dans un diagramme représentent vraiment des transitions qui se suivent dans le temps.

2.3 Le langage ATL

Comme annoncé plus haut, nous décrivons la sémantique de référence pour le petit langage ATL. Le produit final des sections à suivre sera la théorie `ATL-REFERENCE-SEM` :

```
module ATL-REFERENCE-SEM {
  protecting (ATL-OFFICIAL-SEM)
  protecting (ATL-ACTUAL-SEM)
}
```

Cette théorie sera expliquée dans la section 2.6 ci-dessous. Les deux théories `ATL-OFFICIAL-SEM` et `ATL-ACTUAL-SEM` auxquelles elle fait appel sont d'abord décrites, respectivement dans les sections 2.4 et 2.5 ci-dessous.

Le langage ATL n'est pas un langage au sens habituel du terme, c'est-à-dire défini d'abord par sa syntaxe. Nous suivons la proposition de Bjarne Stroustrup, à propos de «the syntax being the user interface of a language,» étant donné que «the only really important constant [...] is the basic semantics of the language» [96, p. 208]. Ainsi, pour nous, une syntaxe pour ATL devra être définie comme une interface à ATL lui-même. Nous décrivons uniquement, ci-dessous, la sémantique. Celle-ci est décrite par la théorie de réécriture `ATL-REFERENCE-SEM`. ATL définit un ensemble d'algorithmes possibles. Un programme ATL (non interfacé) est une théorie qui identifie un de ces algorithmes.

Ainsi, le programme suivant est un programme ATL :

```
module ALGO1-REFERENCE-SEM {
  protecting (ALGO1)
  using      (ATL-REFERENCE-SEM)
}
```

où `ALGO1`, donné ci-dessous, définit l'algorithme dont il est question :

```
module! ALGO1 {
  using (ATL-CONSTRUCTS)
  op  algo1       : -> Algo

  var x : String
  eq input-var  x =  x == "a" or x == "b" .
  eq output-var x =  x == "f" .

  eq  algo1 = ("c" := "a" + "b") ; ("d" := "c" + "c") ;
            ("e" := "c" + "a") ; ("f" := "d" + "e") .
}
```

Les opérations «`input-var`» et «`output-var`» sont des prédicats qui définissent le nom des variables d'*entrée* et de *sortie* du programme. Toutes les opérations qui sont utilisées dans cette théorie sont définies par la théorie suivante :

```
module! ATL-CONSTRUCTS {
  protecting (STRING)
  [ Assignt ]
    op _:=_+_ : String String String    -> Assignt

  [ Algo ]
  [ Assignt < Algo ]
    op noAlgo :                         -> Algo
    op _;_    : Algo Algo               -> Algo
                { assoc id: noAlgo }

  op input-var_  : String -> Bool
  op output-var_ : String -> Bool
}
```

Les opérations définies dans ce module (en-dehors de `input-var` et `output-var`) jouent le rôle de "syntaxe abstraite". C'est-à-dire qu'elles définissent la *structure* des textes décrivant un algorithme, et aucunement une syntaxe proprement dite, ou «concrète». Bien sûr, les noms des opérations sont choisis de façon à évoquer leur rôle. Il est possible de considérer les déclarations d'opérations ci-dessus comme de règles de grammaire sous forme BNF (où les membres gauche et droit de la règle sont inversés).

Pour les noms de variables, le module prédéfini dans CafeOBJ `STRING` est importé. Il définit la sorte `String` des chaînes de caractères : «`ceci`» est une chaîne de caractères. Le point d'exclamation (« `!`») accolé à «`module`» signale que ce sont les modèles initiaux qui sont considérés. Cela implique que deux chaînes de caractères différentes correspondent bien à deux objets différents dans les modèles. Bien entendu, la «syntaxe» des chaînes de caractères ne saurait être préservée au niveau des modèles, qui ont pour principale caractéristique d'être indépendants de la syntaxe utilisée dans les théories. Ainsi, `ceci` sert uniquement à identifier un objet. C'est commode, puisque nous n'avons ainsi pas besoin de déclarer (comme une opération d'arité 1) tous les noms de variables qui apparaissent dans un programme : ils «existent déjà», par la sorte `String`.

La sémantique initiale permet également de s'assurer que les structures déclarées par `ATL-CONSTRUCTS` préservent la non-identité : on n'aura pas, par exemple, dans un modèle, $[\![$`"c" := "a" + "b"`$]\!] = [\![$`"d" := "c" + "c"`$]\!]$. En outre, cette même sémantique initiale implique que le prédicat «`==`» sur des chaînes de caractères, tel qu'utilisé par exemple dans le module `ALGO1` (comme dans «`x == "a"`»), correspond exactement au prédicat «`string=`», spécifié dans `STRING`, qui décrit l'égalité de chaînes. Le prédicat «`==`» renvoie `true` si les deux arguments, en tant que termes syntaxiques CafeOBJ, se réduisent en un même terme syntaxique par les équations de la théorie. La sémantique étant initiale, `true` n'est renvoyé pour deux chaînes de caractères que si celles-ci sont effectivement identiques. Nous avons préféré «`==`» à «`string=`», dans `ALGO1` comme dans les spécifications qui vont suivre, uniquement pour que ces spécifications soient plus lisibles.

Un exemple d'interface au programme `ALGO1-REFERENCE-SEM` est donné par le texte suivant, avec une syntaxe «à la C» :

```
int algo1(int a, int b)
{
  int c, d, e, f;
  c = a + b;
  d = c + c;
  e = c + a;
  f = d + e;
  return f;
}
```

Opérations et syntaxe La différence entre l'approche montrée par cet exemple, et l'utilisation usuelle des spécifications algébriques pour décrire un langage de programmation, suivie, par exemple par [37], est que nous ne tentons pas de décrire la syntaxe du langage par la syntaxe choisie pour des opérations de la théorie.

Notre choix provient de ce qu'il ne nous paraît en fait pas possible de spécifier la syntaxe du langage. En effet, il est possible de désigner comme unique modèle pertinent le modèle des termes, mais cela reviendrait à rejeter l'essence même des spécifications algébriques, à savoir l'indépendance par rapport à la syntaxe. Or, si l'on considère uniquement l'ensemble des modèles, sans en désigner un en particulier, alors la syntaxe donnée par les opérations n'est pas préservée au niveau des modèles. Un exemple du type de confusion qu'apporte la tentative de spécifier la syntaxe est donné par la construction, dans [37], qui consiste, pour pouvoir réutiliser le module `INTEGER`, à déclarer que les entiers sont déjà des expressions du langage : «`[ Int < Expr ]`». Cette construction se comprend au niveau des modèles, puisque l'algèbre des entiers ($\mathbb{Z}$ muni de sa structure d'anneau habituelle), est isomorphe à toute algèbre remplaçant $\mathbb{Z}$ par un ensemble de représentations des entiers (représentation décimale signée, par exemple) : il n'y a pas lieu, au niveau des modèles, de distinguer entre les entiers et leur représentation. Mais les auteurs utilisent ensuite l'équation «`eq [[ I ]] = I .`», pour déclarer que la valeur d'un entier (pris comme expression du langage) est ce même entier (pris cette fois comme valeur). Au niveau des modèles, au vu de ce que nous venons d'expliquer, cela est correct, bien que redondant. Mais si la distinction entre expressions et valeurs, dans la spécification, est destinée à exposer clairement la sémantique du langage, l'équation sus-citée devrait au contraire rendre le lecteur perplexe.

Dans notre spécification de ATL, comme nous ne parlons pas de syntaxe du langage au niveau des théories, nous sommes libres d'utiliser le module `INTEGER` des entiers (dans les modules `ATL-OFFICIAL-SEM` et `ATL-EFFECTIVE-SEM` ci-dessous), sans avoir à parler de la désignation syntaxique des entiers.

2.4 Sémantique officielle

La première étape dans notre méthode est la définition de sa sémantique officielle. Celle-ci est l'objet de la présente section.

2.4.1 La machine

La sémantique de ATL est définie par son interprétation par une machine (abstraite). Les transitions d'états de cette machine lorsqu'elle interprète un programme ATL vont décrire le graphe d'exécution pour ce programme. Cette machine est décrite par la théorie suivante :

```
module* MACHINE {
  [ MachineName ]
  op M : -> MachineName

  [ Inst ] [ Trace ]
  [ Inst < Trace ]
  op noInst : -> Inst
  op (_;_)  : Trace Trace -> Trace  { assoc id: noInst }

  [ MachineState ]
```

```
  [ MAlgo ]

  op (_wTrace:_wAlgo:_) :
      MachineName Trace MAlgo       -> MachineState
  op (_wTrace:_toDo:_wAlgo:_) :
      MachineName Trace Inst MAlgo  -> MachineState
  op (_wTrace:_done:_wAlgo:_) :
      MachineName Trace Inst MAlgo  -> MachineState

  var tr : Trace   var inst : Inst    var algo : MAlgo
  trans M wTrace: tr toDo: inst wAlgo: algo
     => M wTrace: tr done: inst wAlgo: algo .   ** [ e ]

  eq    M wTrace: tr done: inst wAlgo: algo
     =  M wTrace: (tr ; inst) wAlgo: algo .

  protecting (STRING)
  protecting (INT)
  op (_.val_) : MachineState String -> Int

  var x : String  var algo' : MAlgo
  eq (M wTrace: noInst wAlgo: algo ) .val x
   = (M wTrace: noInst wAlgo: algo') .val x .
}
```

La sorte `MachineName` déclare les noms de machine. L'opération `op M : -> MachineName` définit le nom de notre machine («`M`»). Bien sûr, la syntaxe de ce nom ne se retrouve pas dans les modèles. En fait, d'un point de vue purement technique, il n'est même pas nécessaire de nommer la machine, puisque nous n'utilisons qu'une seule machine. Ce nom sert simplement à rendre les spécifications qui vont suivre plus facile à lire, car il est plus naturel de nommer la machine dont on discute les états et changements d'état.

`Inst` est la sorte des *instructions* que peut exécuter la machine. `Trace` est la sorte des *traces* de ce qu'a exécuté la machine. Une trace est constituée d'une *séquence* d'instructions : cela provient de l'opération «`_ ;_`», qui construit les séquences de traces, et du fait qu'une instruction est automatiquement promue au rang de trace, par «`Inst < Trace`».

`MachineState` est la sorte des états de la machine. Un état de la machine est décrit par l'algorithme à exécuter, de sorte `MAlgo` ('algorithme pour la machine') ainsi qu'une trace courante : c'est ce qui est déclaré par l'opération

`_wTrace:_wAlgo:_ : MachineName Trace MAlgo -> MachineState` .
(où «`wTrace`» est pour «with trace», et «`wAlgo`» pour «with algorithm».)

Un état machine peut aussi avoir à effectuer une instruction, ou en avoir effectué une «à l'instant». C'est le motif des deux autres façons de se référer à un état machine :
`_wTrace:_toDo:_doing:_wAlgo:_`, et `_wTrace:_done:_doing:_wAlgo:_`.

Puis, une transition déclare la transition spontanée de tout état ayant une instruction `toDo` vers l'état ayant cette instruction `done`. Cette transition a pour label `e` (pour «execution») ; **il s'agit de la seule transition dans notre spécification de la sémantique officielle**.

L'équation qui suit «range» l'instruction effectuée dans la trace de ce qu'a exécuté la machine.

Enfin, la machine, lorsqu'elle exécute un algorithme, associe des valeurs (entières) aux variables du langage ATL : c'est ce que signifie l'opération `.val`.

La théorie est déclarée, par «`module*`», comme requérant une sémantique lâche, ce qui est nécessaire car aucun objet n'est défini pour les sortes `Inst` et `MAlgo`. (La théorie est alors dite *non sensible* [115, p.681].) Ceci parce que nous cherchons à définir une machine générique, qui a pour seule fonction d'exécuter des instructions et de stocker une trace de l'exécution. La théorie suivante utilise `MACHINE` et définira ce qui constitue des algorithmes et des instructions pour la machine.

Déclarer la machine avec une sémantique lâche est également nécessaire en raison de l'opération `.val`. En effet, celle-ci retourne un entier ; donc si nous voulons préserver les modèles initiaux de la sorte `Int` importée, alors nous ne pouvons utiliser un sémantique initiale. En effet, nous n'explicitons pas dans la suite de la spécification une valeur précise pour chacune des chaînes de caractères existante : donc si la sémantique initiale était utilisée, un terme `S .val x`, avec `S` de sorte `MachineState` et `x` une chaîne de caractères, pourrait générer «un nouvel entier» pour la sorte `Int`, ce qui n'est pas ce que nous voulons exprimer.

En conséquence, aucun des modules qui vont suivre ne sera déclaré avec «`module !`», car cela impliquerait alors la non-protection de la sorte des entiers. Cependant, c'est la sémantique initiale dont nous souhaitons parler *pour tout le reste de la spécification* : nous n'introduisons une sémantique lâche *que pour l'opération* `.val`.

C'est-à-dire que nous utilisons une sémantique où certaines opérations doivent être comprises avec une sémantique initiale, et d'autres avec une sémantique lâche. Il n'existe pas à notre connaissance de cadre théorique se rapportant à ce problème, c'est pourquoi nous exprimons cela ici de façon informelle (hors du texte CafeOBJ). Ce schéma se présentera à nouveau pour une opération `nbProc`, au chapitre suivant, à laquelle nous devrons également attacher une sémantique lâche.

États initiaux et finaux La dernière équation du module est liée à la notion d'*état initial*. Un *état initial* de la machine est un état de la forme : `M wTrace: noInst wAlgo: algo`, i.e., un état avec une trace nulle, ce qui signifie que la machine n'a encore rien exécuté. De façon symétrique, nous appelons *état final* de la machine tout état de la forme : `M wTrace: tr wAlgo: noAlgo`, i.e. un état où l'algorithme à exécuter est l'algorithme nul, ce qui signifie que la machine n'a plus rien à exécuter.

La dernière équation de la théorie `MACHINE` déclare que dans un modèle donné de la théorie `MACHINE`, les valeurs des variables sont les mêmes pour tous les états initiaux. (Pour des valeurs différentes des variables dans un état initial, il suffit de se placer dans un autre modèle.) La motivation pour cette égalité des valeurs dans les états initiaux est une simplification des spécifications qui vont suivre, qui sera expliquée au moment où il sera fait appel à cette équation.

Les graphes d'exécution

Ainsi que le laissait présager la syntaxe que nous avons choisie pour les opérations du module ATL-LANGUAGE, ATL est un langage impératif, avec des séquences d'affectations. La sémantique officielle pour ATL est la suivante : les affectations sont exécutées les unes après les autres, dans l'ordre donné par le texte (abstrait) de l'algorithme. L'effet de ces affectations sera décrit par le module ATL-OFFICIAL-SEM ci-dessous (en page 63). Celui-ci importe le module suivant ATL-OFFICIAL-SEM0, qui décrit comment est exécuté l'algorithme.

```
module ATL-OFFICIAL-SEM0 {
  using      (MACHINE)
  protecting (ATL-CONSTRUCTS)

  [ Algo < MAlgo ]

  op <_> : Assignt -> Inst

  var tr : Trace    var algo : Algo   var assgnt : Assignt

  eq M wTrace: tr wAlgo: (assgnt ; algo)
   = M wTrace: tr toDo: < assgnt > wAlgo: algo .
}
```

Le texte de ce module commence par importer les théories MACHINE et ATL-CONSTRUCTS. Puis la déclaration «Algo < MAlgo» signifie que les algorithmes ATL sont des algorithmes pour la machine. L'opération <_> permet quant à elle de faire des affectations ATL des instructions pour la machine.

Puis, l'interprétation de la séquence ATL en une séquence sur des états machines est décrite, par une seule équation. Celle-ci consiste à identifier les états sans instruction «toDo» ou «done», et s'il reste au moins une affectation dans l'algorithme ATL en train d'être exécuté, de la donner en instruction «toDo».

Pourquoi cette équation génère-t-elle une succession d'états machine ? Cela est expliqué ci-dessous, à partir du programme ALGO1. Considérons la sémantique officielle de ALGO1, décrite par le module suivant :

```
module ALGO1-OFFICIAL-SEM {
  using      (ATL-OFFICIAL-SEM)
  protecting (ALGO1)
}
```

L'exécution de l'algorithme est décrite par le diagramme suivant dans la catégorie des états machines :

$$
\begin{tikzcd}
z_0 \arrow[d, "\{\mathtt{<a_1>}\}"] \\
z_1 \arrow[d, "\{\mathtt{<a_2>}\}"] \\
z_2 \arrow[d, "\{\mathtt{<a_3>}\}"] \\
z_3 \arrow[d, "\{\mathtt{<a_4>}\}"] \\
z_4
\end{tikzcd}
$$

où z_0 est l'état initial ⟦`M wTrace: noInst wAlgo: algo1`⟧,
et «$\mathtt{a}_1$,» «$\mathtt{a}_2$,» «$\mathtt{a}_3$» et «$\mathtt{a}_4$» sont des noms pour les termes suivants de sorte `Assignt` :

$\mathtt{a}_1 =$ `"c" := "a" + "b"`,
$\mathtt{a}_2 =$ `"d" := "c" + "c"`,
$\mathtt{a}_3 =$ `"e" := "c" + "a"`,
$\mathtt{a}_4 =$ `"f" := "d" + "e"`;

et où le nom de flèche «$\{\mathtt{a}_i\}$» indique que la flèche ainsi nommée, de la catégorie ⟦`MachineState`⟧, correspond à la transition d'un état $\mathtt{M\ wTrace:\ tr\ toDo:\ <a_i>\ wAlgo:\ algo}$ (où `tr` est une trace et `algo` un algorithme) vers l'état $\mathtt{M\ wTrace:\ tr\ done:\ <a_i>\ wAlgo:\ algo}$

La raison pour laquelle le diagramme ci-dessus découle de la théorie `ALGO1-OFFICIAL-SEM` est explicitée par les égalités et transitions suivantes, menant de l'état z_0 à l'état z_4. La première égalité provient de la définition de `algo1` donnée dans le module `ALGO1`, la deuxième de la règle d'interprétation donnée dans le module `ATL-OFFICIAL-SEM0`, et la troisième de l'équation «de rangement» de la théorie `MACHINE`. Les transitions, quant à elles, proviennent uniquement de la transition déclarée dans le module `MACHINE`.

$$
\begin{array}{lll}
 & [\![\texttt{M wTrace:noInst wAlgo:algo1}]\!] & = z_0 \\
= & [\![\texttt{M wTrace:noInst wAlgo:}\mathtt{a}_1\texttt{ ; }\mathtt{a}_2\texttt{ ; }\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & \\
= & [\![\texttt{M wTrace:noInst toDo:<}\mathtt{a}_1\texttt{> wAlgo:}\mathtt{a}_2\texttt{ ; }\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & \\
\xrightarrow{\{\mathtt{a}_1\}} & [\![\texttt{M wTrace:noInst done:<}\mathtt{a}_1\texttt{> wAlgo:}\mathtt{a}_2\texttt{ ; }\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & = z_1 \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> wAlgo:}\mathtt{a}_2\texttt{ ; }\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> toDo:<}\mathtt{a}_2\texttt{> wAlgo:}\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & \\
\xrightarrow{\{\mathtt{a}_2\}} & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> done:<}\mathtt{a}_2\texttt{> wAlgo:}\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & = z_2 \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> wAlgo:}\mathtt{a}_3\texttt{ ; }\mathtt{a}_4]\!] & \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> toDo:<}\mathtt{a}_3\texttt{> wAlgo:}\mathtt{a}_4]\!] & \\
\xrightarrow{\{\mathtt{a}_3\}} & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> done:<}\mathtt{a}_3\texttt{> wAlgo:}\mathtt{a}_4]\!] & = z_3 \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> ; <}\mathtt{a}_3\texttt{> wAlgo:}\mathtt{a}_4]\!] & \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> ; <}\mathtt{a}_3\texttt{> toDo:<}\mathtt{a}_4\texttt{> wAlgo:noAlgo}]\!] & \\
\xrightarrow{\{\mathtt{a}_4\}} & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> ; <}\mathtt{a}_3\texttt{> done:<}\mathtt{a}_4\texttt{> wAlgo:noAlgo}]\!] & = z_4 \\
= & [\![\texttt{M wTrace:<}\mathtt{a}_1\texttt{> ; <}\mathtt{a}_2\texttt{> ; <}\mathtt{a}_3\texttt{> ; <}\mathtt{a}_4\texttt{> wAlgo:noAlgo}]\!] &
\end{array}
$$

Ainsi donc, l'exécution de la première affectation `a` de l'état courant du texte de l'algorithme correspond à :

1. attendre que l'instruction précédente ait été exécutée et rangée dans la trace ;
2. faire une instruction `<a>` de l'affectation `a`, et mettre cette instruction en «`toDo`» ;
3. attendre que la machine exécute cette instruction, la transformant en «`done`» ;
4. ranger l'instruction `<a>` ainsi exécutée dans la trace.

Le codage par les diagrammes Nous explicitons ici le codage utilisé pour figurer le graphe d'exécution pour un algorithme, à l'aide de l'exemple ci-dessus. La sémantique de `ALGO1-OFFICIAL-SEM` est donnée par l'ensemble de ses modèles. Mais plus exactement, dans un modèle, on considère, dans la catégorie ⟦`MachineState`⟧, l'état initial de la machine qui contient `algo1` comme algorithme, et les flèches qui partent de cet état, pour mener à un état final. Chaque flèche est interprétée comme l'exécution d'une instruction. Dans le chemin (au sens de la théorie des graphes) qui mène de l'état initial à l'état final, la succession de deux flèches code la succession dans le temps des transitions qu'elles codent respectivement. C'est ainsi qu'est codée la séquentialité de l'algorithme.

Une flèche *composée* de la catégorie comme $\{\mathtt{a}_1\};\{\mathtt{a}_2\}$ code l'exécution «$\mathtt{a}_1$, puis $\mathtt{a}_2$». Cela est cohérent, car nous n'utilisons pas la possibilité de transitions concurrentes et la notion associée de «true concurrency».

Égalités et transitions Nous avons utilisé une égalité pour exprimer comment un algorithme est interprété par la machine. Ceci est un moyen commode pour distinguer la règle d'interprétation, de l'exécution de l'algorithme elle-même, qui est représentée quant à elle par des transitions. C'est-à-dire que ce que nous voulons exprimer est le graphe d'exécution : l'équation n'est qu'un moyen d'expression du graphe, et ne correspond aucunement, au niveau des modèles, à une règle d'interprétation.

2.4.2 Effet des affectations

Nous avons décrit la réalisation d'une séquence d'affectations, mais nous n'avons pas encore décrit l'effet de l'exécution d'une affectation par la machine. Cela est réalisé par le module `ATL-OFFICIAL-SEM`, qui importe le module précédent et complète la spécification de la sémantique officielle de ATL. Des équations conditionnelles y sont utilisées. En CafeOBJ, une équation conditionelle est déclarée avec le mot-clé «`ceq`» (pour 'conditional equation').

```
module ATL-OFFICIAL-SEM {
  using (ATL-OFFICIAL-SEMO)

  var tr : Trace   var algo : Algo   vars x y z w : String

  ceq (M wTrace: (tr ; < z := x + y >) wAlgo: algo) .val w
    =   ((M wTrace: tr wAlgo: algo) .val x)
      + ((M wTrace: tr wAlgo: algo) .val y)
      if w ==  z .

  ceq (M wTrace: (tr ; < z := x + y >) wAlgo: algo) .val w
    = (M wTrace: tr wAlgo: algo) .val w
      if w =/= z .
}
```

Le module `ATL-OFFICIAL-SEMO` est importé avec le mot-clé «`using`» pour signifier que tous les modèles ne sont pas préservés par cet import. En effet, `ATL-OFFICIAL-SEM` consiste à rajouter des contraintes sur le foncteur ⟦`.val`⟧.

Les deux équations définissent les valeurs des variables sur l'état décrit au membre gauche, en fonction des valeurs sur l'état décrit au membre droit, *qui n'est pas celui qui précède dans le graphe d'exécution*. En effet, l'état machine qui précéderait l'état

⟦`M wTrace: (tr ; < z := x + y >) wAlgo: algo`⟧

du membre gauche serait :

⟦`M wTrace: tr wAlgo: z := x + y ; algo`⟧ ,

et non pas l'état

⟦`M wTrace: tr wAlgo: algo`⟧

auquel il est fait référence dans le membre droit. Cependant, nous considérons que les valeurs des variables sur un état *ne doit pas dépendre de l'algorithme qui reste à exécuter*, et donc les deux états précédents doivent porter les mêmes valeurs de variables. Or, comme ils ont même trace, ils proviennent, respectivement, des états initiaux respectifs

⟦`M wTrace: noInst wAlgo: algo' ; z := x + y ; algo`⟧

et

⟦`M wTrace: noInst wAlgo: algo' ; algo`⟧

où `algo'` est un algorithme ATL dont l'exécution a produit la trace `tr`. Or, l'égalité des valeurs sur ces deux états initiaux est assurée par la dernière équation du module `MACHINE`. Par suite, comme nous décrivons des calculs *déterministes*, les valeurs sur nos deux états

`⟦M wTrace: tr wAlgo: z := x + y ; algo⟧`

et

`⟦M wTrace: tr wAlgo: algo⟧`

sont bien les mêmes.

Nous préférons nous référer ainsi à des états équivalents plutôt que de définir l'effet d'une instruction en «remontant» l'interprétation de l'algorithme. En effet, nous ne voulons pas faire dépendre la méthodologie de la possibilité d'inférer de la trace ce qui a été exécuté, une telle inférence n'étant possible que pour des exécutions simples comme celle de programmes ATL. Il serait possible cependant de stocker dans la trace les parties d'algorithmes correspondantes, afin de pouvoir légitimement «remonter» l'exécution : mais cela aurait alourdi la présente exposition.

Remarque Dans les équations ci-dessus, nous nous référons, en membre gauche et en membre droit, à une même variable `algo`. L'explication que nous venons de donner indique que cela n'est pas nécessaire, et qu'une équation plus générale aurait également convenu :

```
var tr : Trace   vars algo algo' algo'' : Algo   vars x y z w : String

ceq (M wTrace: (tr ; < z := x + y >) wAlgo: algo) .val w
 =   ((M wTrace: tr wAlgo: algo' ) .val x)
   + ((M wTrace: tr wAlgo: algo'') .val y)
   if w ==  z .
```

Si nous avons préféré conserver la variable `algo` dans le membre droit dans notre spécification, c'est afin que cette spécification soit exécutable—sachant que “restreindre” ainsi l'équation ne change pas l'effet sur la définition des valeurs sur les états. Or nous souhaitons obtenir une spécification si possible exécutable, pour l'utiliser, dans la section 2.7.2, pour effectuer une preuve semi-automatique. Ceci est un exemple de la façon dont le voeu d'obtention d'une spécification exécutable peut influer sur le style de la spécification.

2.5 Sémantique effective

Nous définissons à présent la sémantique effective de ATL, au sens de la norme. C'est-à-dire que nous formulons la sémantique telle qu'elle doit être réalisée par le compilateur.

Afin de pouvoir comparer aisément à la sémantique officielle, nous reprenons la distinction entre machine, graphe d'exécution, et effet des transitions, qui a été utilisée pour la sémantique officielle.

2.5.1 La machine

Pour la machine, nous pouvons en fait réutiliser le modèle de machine défini pour la sémantique officielle. Afin de créer une nouvelle machine, nous renommons quelques sortes et opérations :

```
module* ACTUAL-SEM-MACHINE {
  protecting (MACHINE * {
        sort Inst         -> ActInst,
        sort Trace        -> ActTrace,
        sort MachineState -> ActMachineState,
        op   noInst       -> noActInst,
        op   M            -> AM
        })
}
```

Le nom de sorte «`MachineName`» est conservé ; en revanche, le nom de la machine est changé, en «`AM`» (pour «actual machine»). Les constituants de la trace sont renommés, ainsi que la sorte des états machine. L'opération «`.val`» n'est pas renommée.

Il est important de noter que le renommage crée bien un nouveau module, même si les modèles sont les mêmes. La distinction entre les deux modules nous permettra dans les sections 2.6 et 2.7.2 d'inclure les deux machines dans une même spécification, sans qu'elles soient confondues.

Comme aucun objet n'est déclaré pour la sorte `ActInst`, la nouvelle théorie de machine est, comme la précédente, déclarée avec «`module*`» comme requérant une sémantique lâche.

2.5.2 Le graphe d'exécution

Le graphe d'exécution défini correspond à la contrainte minimale sur le compilateur :

```
module ATL-ACTUAL-SEM0 {
  using (ACTUAL-SEM-MACHINE)
  protecting (ATL-CONSTRUCTS)

  [ Algo < MAlgo ]
  op <<_>> : Algo -> ActInst


  var algo : Algo
  ceq AM wTrace: noActInst wAlgo: algo
    = AM wTrace: noActInst toDo: << algo >> wAlgo: noAlgo
      if algo =/= noAlgo .
}
```

C'est-à-dire que les algorithmes eux-mêmes sont pris comme instructions, et que l'on demande à la machine d'exécuter ces instructions. C'est-à-dire que depuis un état initial avec un algorithme à exécuter `algo`, on donne cet algorithme comme instruction à exécuter à la machine. Le fonctionnement de la machine est le même que pour la sémantique officielle, donc la machine va exécuter *en une transition* cette instruction et la ranger dans la trace. Comme aucun algorithme ne reste alors à exécuter, il s'agit d'un état final. Ce type d'exécution correspond à ce qui serait exprimé par une sémantique dénotationnelle (au sens classique—voir les sections 1.3.4 et 1.3.9) pour le langage ATL.

Pour exemple, nous reprenons le programme `ALGO1`, et mettons côte à côte les graphes d'exécution officiel et effectif :

$$
\begin{tikzcd}
z_0 \arrow[d, swap, "\{\texttt{<a}_1\texttt{>}\}"] & & y_0 \arrow[dddd, "\{\texttt{<<a}_1\texttt{ ; a}_2\texttt{ ; a}_3\texttt{ ; a}_4\texttt{>>}\}"] \\
z_1 \arrow[d, swap, "\{\texttt{<a}_2\texttt{>}\}"] & & \\
z_2 \arrow[d, swap, "\{\texttt{<a}_3\texttt{>}\}"] & & \\
z_3 \arrow[d, swap, "\{\texttt{<a}_4\texttt{>}\}"] & & \\
z_4 & & y_1
\end{tikzcd}
$$

—avec les mêmes conventions qu'auparavant quant aux états z_i et aux flèches $\{\texttt{<a}_i\texttt{>}\}$, et avec
$y_0 =$ ⟦`AM wTrace: noActInst wAlgo: algo1`⟧
et $y_1 =$ ⟦`AM wTrace: << algo1 >> wAlgo: noAlgo`⟧,
et $\{\texttt{<<a}_1\texttt{ ; a}_2\texttt{ ; a}_3\texttt{ ; a}_4\texttt{>>}\}$ un nom pour la transition de
$(y_0 =)$ ⟦`AM wTrace: noActInst toDo: << algo1 >> wAlgo: algo1`⟧
vers ⟦`AM wTrace: noActInst done: << algo1 >> wAlgo: algo1`⟧ $(= y_1)$.

2.5.3 Effet de l'exécution de l'algorithme

Il reste à définir l'effet d'une telle l'exécution «en un coup» de l'algorithme.

Ici, nous reproduisons le type de formulation de contrat utilisé, par exemple, pour la norme du langage C [60]. Cette norme déclare que «The semantic descriptions in this International Standard describes the behaviour of an abstract machine in which issues of optimizations are irrelevant.» [60, p. 15]. Cela correspond à ce que nous avons appelé sémantique officielle. Quant à la contrainte sur le compilateur, elle est minimale : «At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.» [60, p. 15]. (Quelques contraintes supplémentaires sont données par la norme, qui seront discutées dans la section 2.7 ci-dessous.) C'est-à-dire que la sémantique effective est décrite par référence à la sémantique officielle : c'est le schéma que nous reprenons ci-dessous, en le formalisant :

```
module ATL-ACTUAL-SEM {
  protecting (ATL-OFFICIAL-SEM)
```

```
  using      (ATL-ACTUAL-SEMO)

  var actTr : ActTrace   var tr : Trace
  var algo  : Algo       var x  : String

  ceq (AM wTrace: noActInst wAlgo: algo) .val x
    = (M  wTrace: noInst    wAlgo: algo) .val x
      if input-var x .

  ceq (AM wTrace: actTr wAlgo: noAlgo) .val x
    = (M  wTrace: tr    wAlgo: noAlgo) .val x
       if (AM wTrace: noActInst wAlgo: algo) ==> (AM wTrace: actTr wAlgo: noAlgo)
      and (M  wTrace: noInst    wAlgo: algo) ==> (M  wTrace: tr    wAlgo: noAlgo)
      and  output-var x .
}
```

Le mot-clé «`protecting`» utilisé pour l'importation de `ATL-OFFICIAL-SEM` sert à signifier que les modèles de la sémantique officielle ne sont pas affectés par les équations qui vont suivre : ils servent bien seulement de référence.

La première équation déclare que les valeurs des variables en entrée doivent être les mêmes pour les états initiaux de la machine effective que pour ceux de la machine officielle.

La seconde équation déclare que les valeurs des variables en sortie, sur un état final de la machine effective, sont les mêmes que sur l'état final de la machine officielle ayant exécuté le même algorithme. Le prédicat «`==>`» utilisé dans la condition correspond à l'internalisation, dans CafeOBJ, de la notion de transition. C'est-à-dire que dans la première de ses deux utilisations ci-dessus, le prédicat «`==>`» renvoie `true` si et seulement s'il existe, dans la catégorie ⟦`ActMachineState`⟧, une succession de flèches partant de l'objet-état initial ⟦`AM wTrace: noActInst wAlgo: algo`⟧ et arrivant à l'objet-état final ⟦`AM wTrace: actTr wAlgo: noAlgo`⟧. De même, dans sa deuxième utilisation, le prédicat renvoie `true` si et seulement s'il existe, dans la catégorie ⟦`MachineState`⟧, une succession de flèches partant de l'objet-état initial ⟦`M wTrace: noInst wAlgo: algo`⟧ et arrivant à l'objet-état final ⟦`M wTrace: tr wAlgo: noAlgo`⟧.

2.5.4 Sémantique effective du programme exemple

La sémantique effective du programme `ALGO1` est donc décrite par la théorie suivante :

```
module ALGO1-ACTUAL-SEM {
  using      (ATL-ACTUAL-SEM)
  protecting (ALGO1)
}
```

2.6 Sémantique de référence

Nous pouvons à présent formuler la sémantique de référence pour le langage ATL :

```
module ATL-REFERENCE-SEM {
  protecting (ATL-OFFICIAL-SEM)
  protecting (ATL-ACTUAL-SEM)
}
```

C'est-à-dire que la sémantique de référence est définie comme la *somme* (au sens intuitif, mais aussi au sens catégorique) de la sémantique officielle et de la sémantique effective. Le mot-clé «`protecting`» indique que ces deux sémantiques sont importées ici telles quelles, leurs modules communs étant partagés, comme `ATL-CONSTRUCTS`.

(En fait, comme `ATL-ACTUAL-SEM` importe `ATL-OFFICIAL-SEM` pour pouvoir y faire référence, la somme ci-dessus est égale à la théorie `ATL-ACTUAL-SEM` seule. Mais l'identité entre sémantique effective et de référence n'est vraie que si la sémantique officielle est importée dans son entier par la sémantique effective, ce qui est le cas pour ATL mais n'est pas forcément le cas en général.)

Dans la spécification `ATL-REFERENCE-SEM`, nous avons donc *deux* catégories d'états machine : ⟦`MachineState`⟧ et ⟦`ActMachineState`⟧. Et les valeurs sur les états de la deuxième catégorie sont définies à partir des valeurs sur les états de la première. Par exemple, pour le programme `ALGO1`, on a dans la première catégorie le graphe d'exécution officiel, et dans la deuxième le graphe d'exécution effectif, tels qu'ils sont montrés dans le diagramme précédent.

La différence entre ces deux sémantiques est dans leur **statut** : tandis que la sémantique officielle n'engage le compilateur à rien, la sémantique effective **définit** le comportement de celui-ci. Dans le cas d'ATL, la définition de ce que doit réaliser le compilateur n'est guère contraignante. Une sémantique effective plus contraignante sera utilisée dans le chapitre suivant.

Pour résumer, nous avons le schéma suivant (où la sémantique à laquelle est contrainte le compilateur est soulignée) :

$$\begin{tikzcd}
\text{sém. officielle} \arrow[dr] & {} \arrow[d, phantom, "+"] & \underline{\text{sém. effective}} \arrow[ll, "\text{fait référence à}"'] \arrow[dl] \\
& \text{sem. de référence} &
\end{tikzcd}$$

Ainsi, tandis que la sémantique effective indique au programmeur comment son programme sera effectivement réalisé, la sémantique officielle indique au compilateur les valeurs qui doivent être effectivement produites.

2.6.1 Sémantique de référence du programme exemple

La sémantique de référence du programme ALGO1 est donnée par la théorie suivante (qui a été citée à la page 55) :

```
module ALGO1-REFERENCE-SEM {
  protecting (ALGO1)
  using      (ATL-REFERENCE-SEM)
}
```

Ainsi, nous avons explicité le fait que la sémantique d'un programme inclut une sémantique officielle *et* une sémantique effective.

2.7 Optimisations et raffinement

Dans la littérature sur la sémantique des langages de programmation, lorsqu'il est parlé de deux sémantiques pour un même programme, c'est d'ordinaire dans le cadre des *optimisations* à la compilation. Un exemple est [63], qui donne une méthode pour prouver la correction de certaines optimisations typiques. Dans la présente section nous situons le sujet des optimisations par rapport à notre structure pour la sémantique des langages de programmation. En particulier nous nous intéressons au rapport entre les notions d'optimisation et de raffinement.

Raffinement

La norme du langage C déclare : «an implementation might define a one-to-one correspondence between abstract and actual semantics» [60, p. 16]. En effet, la sémantique effective laisse le compilateur libre de faire ce qu'il veut pourvu que la relation entre données en entrée et données en sortie soit satisfaite. Mais l'implanteur du compilateur doit trouver une façon constructive de faire en sorte que cette relation soit satisfaite. La plus simple est de copier la sémantique officielle.

Du point de vue du rapport du compilateur à la norme, cette opération correspond à *raffiner* la sémantique effective donnée par la norme, laquelle sémantique effective, pour C comme pour ATL, consiste en une relation entre données en entrée et données en sortie.

Dans notre cadre, nous donnons un sens précis à la notion de raffinement. Informellement, le raffinement d'une sémantique y consiste en l'ajout, dans les graphes d'exécution, d'états intermédiaires. L'expression formelle est la suivante. Soient deux sémantiques S et S' pour un langage L, décrites avec la méthode présentée dans les sections précédentes. Pour un programme `P` écrit dans L, appelons `P-S` et `P-S'` les théories qui définissent la sémantique «S» et la sémantique «S'» de `P`, respectivement. Alors la sémantique S' est un *raffinement* de S si, pour tout programme `P`, et pour tout modèle M de `P-S`, il existe parmi les modèles de `P-S'` un modèle M' tel que le graphe d'exécution dans M' ait le lien suivant au graphe d'exécution dans M. Soit une flèche $f : s_1 \rightarrow s_2$ sur le chemin d'exécution dans M. Alors le chemin d'exécution dans M' comporte un sous-chemin commençant avec un état s'_1, et finissant avec un état s'_2, tels que s'_1 ait les mêmes valeurs de variables que s_1, et que s'_2 ait les mêmes valeurs de variables que s_2.

Ainsi, dans le cas de ATL et de C, copier la sémantique officielle permet effectivement d'obtenir un raffinement de la sémantique effective. En effet, pour un modèle dans la sémantique effective, on choisi le modèle dans (la copie de) la sémantique officielle qui a exactement les mêmes valeurs pour les variables sur l'état initial—donc a fortiori les mêmes valeurs pour les variables en entrée du programme. De même pour l'état final, et les variables en sortie du programme. La garantie du fait qu'un tel modèle puisse être trouvé dans la sémantique officielle provient du fait que la sémantique effective est justement définie à partir des états initiaux et finaux dans la sémantique officielle.

Remarque Dans la définition formelle du raffinement donnée ci-dessus, la flèche f dans M a pour pendant, dans M', la composée des flèches du sous-chemin dont il est question. Donc, si la notion d'homomorphisme est étendue au modèles catégoriques de la logique de réécriture, alors la condition de raffinement revient à dire qu'il existe un homomorphisme de M vers M' avec l'identité pour la sorte des valeurs.

Optimisation

Copier la sémantique officielle en guise de raffinement est la stratégie de base de tous les compilateurs C. Mais comme cette sémantique n'est pas «optimisée», le travail des compilateurs consiste ensuite à «optimiser» cette sémantique. Prévoyant cette stratégie de compilation, la norme de C fournit elle-même quelques avertissements, comme au sujet du réarrangement d'expressions flottantes par rapport à la sémantique officielle, ou au sujet de la commutativité de l'addition sur les entiers, dont il faut en fait se méfier à cause de la taille limitée des entiers en mémoire [60, p. 17].

Dans notre cadre, l'*optimisation* concerne des raffinements de la sémantique initiale. Une optimisation d'un raffinement S, donnant un raffinement S', est n'importe quelle transformation. En effet, la notion de performance n'est pas définie dans notre cadre, alors que la motivation d'une optimisation est d'améliorer la performance.

En pratique, le compilateur part d'une sémantique S, qui est raffinement de la sémantique effective, et effectue une optimisation, menant à une sémantique S' : le but est alors de montrer que S' est bien également un raffinement de la sémantique effective.

Dans la littérature, la notion d'optimisation n'est pas placée dans un cadre sémantique officielle/sémantique effective. L'optimisation s'applique donc directement sur la sémantique officielle, et il s'agit de prouver que l'optimisation est «correcte». La plupart du temps, ce qui définit cette correction est laissé implicite. Par exemple, dans [63], les preuves pour la correction des optimisations consistent à montrer que la relation entrées-sorties du programme est préservée par l'optimisation : il est donc implicite que c'est la relation entrée-sortie que le programme a pour but de définir—ce qui est explicité avec notre notion de sémantique effective.

Expliciter les contraintes concernant la sémantique effective permet donc d'*expliciter* ce en quoi consiste la tâche du compilateur, au lieu de définir ce qu'est un compilateur par l'exhibition des techniques de compilation.

2.7.1 Un raffinement de la sémantique effective de ATL

Afin d'illustrer la discussion ci-dessus au sujet du raffinement et de l'optimisation, nous donnons un raffinement de la sémantique effective de ATL, qui consiste en une copie optimisée de la sémantique officielle.

La machine

La sémantique qui va servir de raffinement à la sémantique effective est définie comme une sémantique à part entière, comme le sont la sémantique officielle et la sémantique effective. Nous reprenons le découpage utilisé précédemment : machine, description du graphe d'exécution et description de l'effet des transitions.

Pour la machine, nous réutilisons à nouveau la théorie de la machine officielle, en changeant quelques noms.

```
module* REFIND-ACTUAL-SEM-MACHINE {
  extending (MACHINE * {
      sort MachineState -> RActMachineState,
      sort Inst         -> RActInst,
      sort Trace        -> RActTrace,
      op noInst         -> noRActInst,
      op   M            -> RAM })

  op (_||_)       : RActInst RActInst   -> RActInst   { assoc id: noRActInst }
}
```

La sorte des instructions est enrichie avec la possibilité de composer deux instructions pour en former une troisième. Le symbole «`||`» est choisi car cette composition sera utilisée ci-dessous pour exprimer la possibilité d'une exécution de deux instructions en parallèle.

Le graphe d'exécution

L'exécution de cette sémantique effective raffinée est inspirée de la sémantique officielle. Mais par rapport à cette dernière une optimisation est introduite : lorsque deux affectations successives sont indépendantes l'une de l'autre (dans le programme), alors elles peuvent être exécutées de façon concurrente. Un exemple de deux affectations successives indépendantes l'une de l'autre est donné par les deuxième et troisième affectations de l'algorithme `algo1` : `(d := c + c) ; (e := c + a)` .

Le graphe d'exécution est défini comme suit :

```
module ATL-REFIND-ACTUAL-SEMO {
  protecting (ATL-CONSTRUCTS)
  using (REFIND-ACTUAL-SEM-MACHINE)
```

```
  [ Algo < MAlgo ]
  op <_> : Assignt -> RActInst

  var tr : RActTrace  var algo : Algo
  vars x x' y y' z z' w : String

  ** sequence of two independent assignment instructions:
  ceq RAM wTrace: tr wAlgo: ((z := x + y)  ;   (z' := x' + y') ; algo)
    = RAM wTrace: tr toDo: (< z := x + y > || < z' := x' + y' >)
                     wAlgo: algo
      if not (z' == z or x' == z or y' == z or x == z' or y == z') .

  ** sequence of two non-independent assignment instructions:
  ceq RAM wTrace: tr wAlgo: ((z := x + y) ; (z' := x' + y') ; algo)
    = RAM wTrace: tr toDo:  < z := x + y >
                     wAlgo:   ((z' := x' + y') ; algo)
      if      z' == z or x' == z or y' == z or x == z' or y == z' .

  ** when only one assignment instruction is left:
  eq RAM wTrace: tr wAlgo: (z := x + y)
   = RAM wTrace: tr toDo: < z := x + y > wAlgo: noAlgo .
}
```

Les deux premières équations indiquent que si deux affectations successives dans l'algorithme sont indépendantes l'une de l'autre, elles peuvent être données comme une seule instruction, composée des deux instructions correspondantes. (Pour simplifier l'exposé, nous n'avons pas spécifié la recherche, dans l'algorithme, de la plus longue séquence d'affectations indépendantes entre elles.)

L'exécution de l'instruction composée se fait en une transition : cela représente le fait que le compilateur n'est pas tenu d'exécuter les deux instructions en séquence.

Par exemple, pour le programme `ALGO1`, les graphes d'exécution officiel et «raffiné» sont, pour l'algorithme

```
algo1 = ("c" := "a" + "b") ; ("d" := "c" + "c") ;
        ("e" := "c" + "a") ; ("f" := "d" + "e") :
```

$$\begin{tikzcd}
z_0 \arrow[d, "\{\texttt{<a}_1\texttt{>}\}"'] & & r_0 \arrow[d, "\{\texttt{<a}_1\texttt{>}\}"] \\
z_1 \arrow[d, "\{\texttt{<a}_2\texttt{>}\}"'] & & r_1 \arrow[dd, "\{\texttt{<a}_2\texttt{> || <a}_3\texttt{>}\}"] \\
z_2 \arrow[d, "\{\texttt{<a}_3\texttt{>}\}"'] & & \\
z_3 \arrow[d, "\{\texttt{<a}_4\texttt{>}\}"'] & & r_2 \arrow[d, "\{\texttt{<a}_4\texttt{>}\}"] \\
z_4 & & r_3
\end{tikzcd}$$

À droite est le graphe d'exécution «raffiné», avec
$r_0 = [\![\texttt{RAM wTrace: noRActInst wAlgo: algo1}]\!]$,
$r_1 = [\![\texttt{RAM wTrace: <a}_1\texttt{> wAlgo: a}_2\,;\texttt{a}_3\,;\texttt{a}_4]\!]$,
$r_2 = [\![\texttt{RAM wTrace: <a}_1\texttt{> ; (<a}_2\texttt{> || <a}_3\texttt{>) wAlgo: a}_4]\!]$,
$r_3 = [\![\texttt{RAM wTrace: <a}_1\texttt{> ; (<a}_2\texttt{> || <a}_3\texttt{>) ; a}_4\texttt{ wAlgo: noAlgo}]\!]$,

et où $\{\texttt{<a}_2\texttt{> || <a}_3\texttt{>}\}$ est un nom pour la transition de l'état
$(r_1 =)$ $[\![\texttt{RAM wTrace: <a}_1\texttt{> toDo: (<a}_2\texttt{> || <a}_3\texttt{>) wAlgo: a}_4]\!]$
vers l'état $[\![\texttt{RAM wTrace: <a}_1\texttt{> toDo: (<a}_2\texttt{> || <a}_3\texttt{>) wAlgo: a}_4]\!]$ $(= r_2)$.

Analyse de dépendance Les règles d'interprétation ci-dessus utilisent une *analyse de dépendance*. Cette notion est familière dans le champ de la *compilation parallèle*, voir par exemple [6]. Elle nous permet ici de mieux situer les rapports entre sémantique officielle, sémantique effective, et optimisation. Le plus souvent, le compilateur *raffine* la sémantique effective, en s'aidant de la sémantique officielle. Mais pour des questions d'optimisation, il commence par effectuer une analyse du texte du programme, de façon à en obtenir un graphe de dépendances, ce qui correspond à ce que l'on peut qualifier d'un *dé-raffinement* de la sémantique officielle du programme. En effet, la majeure partie des ordonnancements, et toutes les réutilisations de variables du programme peuvent être vues comme des raffinements par le programmeur d'un graphe de dépendances. Après cette étape de dé-raffinage, le compilateur, à partir du graphe de dépendances «retrouvé», produit son propre raffinement. Dans le cas d'ATL ci-dessus, après détection, par analyse de dépendance, de l'indépendance de deux affectations, le raffinement choisi consiste à «ne pas raffiner», i.e., déléguer au compilateur le raffinement de l'exécution des deux affectations—ce qui l'autorisera à les effectuer en parallèle s'il le souhaite. De sorte que la sémantique d'ATL donnée ci-dessus est un dé-raffinement de la sémantique officielle, et cependant encore un raffinement de la sémantique effective.

L'effet des affectations

La théorie suivante indique que pour les effets, les instructions lancées «en parallèle» sont indépendantes entre elles : l'effet d'une des instructions ne dépend pas de l'effet des autres :

```
module ATL-REFIND-ACTUAL-SEM {
  using (ATL-REFIND-ACTUAL-SEM0)

  var rTr      : RActTrace   vars x y z w : String
  var rActInst : RActInst    var algo : Algo

  ceq (RAM wTrace: (rTr ; (rActInst || < z := x + y > )) wAlgo: algo) .val w
    =   ((RAM wTrace: rTr wAlgo: algo) .val x)
      + ((RAM wTrace: rTr wAlgo: algo) .val y)
      if w ==  z .

  ceq (RAM wTrace: (rTr ; (rActInst || < z := x + y > )) wAlgo: algo) .val w
    = (RAM wTrace: (rTr ; rActInst) wAlgo: algo) .val w
      if w =/= z .
}
```

Les deux équations comprennent le cas des instructions non composées, car pour toute affectation `assgnt` on a : $[\![$`< assgnt >`$]\!] = [\![$`noRActInst`$]\!][\![$`||`$]\!][\![$`< assgnt >`$]\!]$, car `noRActInst` est l'identité pour l'opération `||` (ce qui est déclaré dans le module `REFIND-ACTUAL-SEM-MACHINE` ci-dessus).

2.7.2 La preuve du raffinement

Stratégie de preuve Nous souhaitons montrer que `ATL-REFIND-ACTUAL-SEM` est bien un raffinement de la sémantique effective du langage.

Il nous faut donc montrer que `ATL-REFIND-ACTUAL-SEM` produit les mêmes rapports entrées-sorties, pour tous les algorithmes, que la sémantique effective.

Pour cela, nous utilisons le fait que `ATL-REFIND-ACTUAL-SEM` est une modification de la sémantique officielle, et que cette dernière est un raffinement de la sémantique effective (par la définition même de cette dernière). Nous montrons donc que `ATL-REFIND-ACTUAL-SEM` produit le même rapport entrées-sorties que la sémantique officielle.

Remarquons que nous ne cherchons pas à montrer que la sémantique raffinée «satisfait» d'une quelconque façon la sémantique officielle : la satisfaction est par rapport à la sémantique effective.

Preuve semi-automatique La sémantique raffinée ne diffère de la sémantique officielle que sur les affectations successives *indépendantes* entre elles. Pour comparer les rapports entrées-sorties des deux sémantiques, nous n'avons donc qu'à montrer que lorsqu'il s'agit de l'exécution de telles affectations, les deux sémantiques sont bien équivalentes.

Pour cela, nous utilisons le système CafeOBJ, conçu entre autres pour permettre de faire des preuves semi-automatiques. CafeOBJ s'appuie pour cela sur la réécriture associée aux équations et transitions des théories. La relation entre les modèles et les réécritures est délicate : une présentation précise demanderait une section de rappel. Mais notre propos dans cette thèse n'est pas de faire des preuves de correction : la preuve qui suit sert seulement à indiquer l'intérêt, si l'on souhaite faire des preuves de correction, de pouvoir manipuler conjointement les deux sémantiques que l'on souhaite comparer. Nous commenterons donc «en direct» le rôle de la réécriture dans la preuve automatique.

À la vue des équations définissant les valeurs des variables sur les états (dans les modules respectifs `ATL-OFFICIAL-SEM` et `ATL-REFIND-ACTUAL-SEM`), nous observons que dans les deux cas, les seules valeurs susceptibles d'être affectées par l'exécution de la partie d'algorithme `(z := x + y) ; (z' := x' + y')` sont celles des variables `z` et `z'`. Nous pouvons donc concevoir le test suivant, avec CafeOBJ (commenté ci-après) :

```
open (ATL-REFIND-ACTUAL-SEM + ATL-OFFICIAL-SEM) .

var w : String .
eq (RAM wTrace: noRActInst wAlgo: noAlgo) .val w
 = (M   wTrace: noInst     wAlgo: noAlgo) .val w .


**> automatic proof:

exec ((RAM wTrace: noRActInst wAlgo: ("z" := "x" + "y") ; ("z'" := "x'" + "y'"))
                        .val "z")
      ==
     ((M   wTrace: noInst     wAlgo: ("z" := "x" + "y") ; ("z'" := "x'" + "y'"))
                        .val "z")
     and
     ((RAM wTrace: noRActInst wAlgo: ("z" := "x" + "y") ; ("z'" := "x'" + "y'"))
                        .val "z'")
      ==
     ((M   wTrace: noInst     wAlgo: ("z" := "x" + "y") ; ("z'" := "x'" + "y'"))
                        .val "z'") .
**> should be: true

**> seeing closely:
exec (RAM wTrace: noRActInst wAlgo: ("z" := "x" + "y") ; ("z'" := "x'" + "y'"))
                        .val "z" .
**> should be equal to:
exec (M   wTrace: noInst     wAlgo: ("z" := "x" + "y") ; ("z'" := "x'" + "y'"))
                        .val "z" .

close
```

Par la commande «`open (ATL-REFIND-ACTUAL-SEM + ATL-OFFICIAL-SEM)`» nous nous plaçons dans l'environnement des sémantiques effective optimisée et officielle.
Pour simplifier la preuve, nous limitons l'algorithme, dans l'état initial, à «`("z" := "x" + "y") ; ("z'" := "x'" + "y'")`», le cas à examiner. Ici, les chaînes de caractères `"x"`, ..., `"z'"` tiennent lieu de variables. Le fait qu'elles soient toutes différentes implique que l'indépendance des deux affectations sera assurée, ce qui nous place dans le cas que nous voulons examiner.
Comme nous partons de l'état initial, la première équation du test déclare l'égalité des valeurs de toutes les variables sur les états initiaux de la machine effective raffinée (`RAM`) et officielle (`M`), comme il avait été fait pour les variables en entrée entre la machine effective et la machine officielle dans `ATL-ACTUAL-SEM`. L'équation est donnée pour des algorithmes vides : comme la spécification des machines indique l'égalité des valeurs sur tous les états initiaux, la déclaration remplit par inférence l'office voulu.
La première commande «`exec`» effectue la preuve voulue. Comme indiqué dans la section 2.2.2, «`exec`» va d'abord ré-écrire les quatre termes `... wTrace: ... wAlgo: ...`, les amenant à des des états finaux, avant d'évaluer la valeur des variables `"z"` ou `"z'"` sur ces états finaux.
Les deux dernière commandes «`exec`» ne prouvent rien de nouveau, elles servent seulement à mieux voir les états finaux (pour le cas `"z"`) et pourquoi ils sont égaux.

Le résultat de l'exécution de ce test avec CafeOBJ est donné ci-dessous (les termes à réduire sont indiqués par le système : nous ne les répétons pas, afin que le texte soit plus lisible) :

```
-- opening module ATL-REFIND-ACTUAL-SEM + ATL-OFFICIAL-SEM.. done._
**> automatic proof:*
-- execute in %ATL-REFIND-ACTUAL-SEM + ATL-OFFICIAL-SEM :
      ... (the term for the first execution)
true : Bool
(0.000 sec for parse, 69 rewrites(0.000 sec), 326 matches)
**> should be: true
**> seeing closely:
-- execute in %ATL-REFIND-ACTUAL-SEM + ATL-OFFICIAL-SEM :
      ... (the term for the second execution)
((M wTrace: noInst wAlgo: noAlgo) .val "x") + ((M wTrace: noInst
                                              wAlgo: noAlgo)
                                              .val "y") : Int
(0.000 sec for parse, 21 rewrites(0.000 sec), 99 matches)
**> should be equal to:
-- execute in %ATL-REFIND-ACTUAL-SEM + ATL-OFFICIAL-SEM :
      ... (the term for the third execution)
((M wTrace: noInst wAlgo: noAlgo) .val "x") + ((M wTrace: noInst
                                              wAlgo: noAlgo)
                                              .val "y") : Int
(0.000 sec for parse, 12 rewrites(0.000 sec), 63 matches)
CafeOBJ>
```

Ainsi la réponse «`true`», qui indique que les deux sémantiques affectent les mêmes valeurs aux variables `"z"` et `"z'"`, est obtenue automatiquement à partir des équations et transitions des théories `ATL-REFIND-ACTUAL-SEM` et `ATL-OFFICIAL-SEM`. □

2.8 Sur le style des spécifications

Dans les spécifications ci-dessus, nous avons utilisé la logique de réécriture : mais le style utilisé dans ces spécifications est plus proche des spécifications dites **cachées** [38] que celui que l'on trouve d'ordinaire dans les spécifications RWL. La présente section explique ce choix.

2.8.1 Observation des valeurs

Comme indiqué plus haut, l'accent a été mis dès l'introduction de la logique de réécriture [75] sur le fait que cela permettait de spécifier de façon naturelle des transitions concurrentes. Par conséquent, la RWL est utilisée le plus souvent pour cet aspect.

Or, spécifier de façon naturelle avec des transitions concurrentes consiste, comme nous l'avons exposé avec l'exemple des Lyumines, à spécifier des *états* comme étant constitués de parties indépendantes, qui peuvent alors «transiter» de façon indépendante vers d'autres formes. L'utilisation la plus simple de ce style consiste à placer toutes les informations concernant l'état, dans le terme syntaxique qui décrit cet état. Par exemple, pour l'état d'une machine comme ci-dessus, les valeurs associées aux variables seraient alors données dans le terme syntaxique, par exemple par un couple (nom de variable, valeur) pour chaque variable ayant une valeur associée. La description d'une affectation par une transition décrirait alors directement l'effet sur les valeurs des variables.

Les raisons, en plus de celle expliquée en section 2.2.3, pour lesquelles nous n'avons pas suivi ce style naturel aux spécifications de logique de réécriture sont les suivantes :

1. Nous souhaitions mettre l'accent sur les graphes d'exécution : il était donc adéquat de séparer l'aspect dynamique de l'exécution (le graphe) de l'effet des transitions sur les valeurs. En outre, le fait que les valeurs ne soient pas stockées directement dans le terme syntaxique décrivant l'état de la machine, mais accessibles via une fonction d'observation simplifie les termes décrivant les états machines, rendant la spécification plus lisible en ce sens.
2. Dans le chapitre suivant, l'architecture de la machine sera raffinée : conformément à notre approche, au lieu de modifier en conséquence les termes syntaxiques décrivant la machine, ces informations additionnelles seront accessibles via des fonctions d'observation. Ainsi, raffiner une machine n'implique pas de ré-écrire les spécifications. (C'est la méthodologie «orientée objet» appliquée à la spécification d'une machine et de son utilisation.)
3. Disposer les valeurs dans le terme syntaxique ne permet pas d'effectuer des transitions concurrentes avec *lecture concurrente* de valeurs, comme il le faudrait pour des affectations s'effectuant en parallèle. Une solution est proposée dans [74] : elle repose sur la notion d'égalités de termes, avec des équations qui doivent être rajoutées à la spécification. Par contraste, dans notre approche, comme la fonction d'observation `.val` a un statut de description des valeurs

sur un état donné, son appel ne suppose pas de notion de concurrence, donc le problème de l'accès concurrent à des valeurs ne se pose pas.

2.8.2 L'alternative des spécifications cachées

En conclusion, nous utilisons RWL pour pouvoir décrire des transitions, mais évitons de décrire des transitions concurrentes. En outre, nous nous fondons au maximum sur la notion d'accès à des caractéristiques d'un état machine via des fonctions d'observation (une seule dans ce chapitre, pour les valeurs des variables, mais d'autres au chapitre suivant) : cela est la caractéristique de l'approche des spécifications cachées.

Équivalence comportementale

Un des concepts fondamentaux des spécifications cachées est celui d'*équivalence comportementale* de deux états : un état machine s, par exemple, est comportementalement équivalent à un autre état machine s' si aucune «expérience» sur ces états ne permet de les distinguer. Pour définir cette notion d'«expérience», les spécifications cachées distinguent entre des sortes «cachées» et des sortes «visibles» : une sorte visible correspond au nom d'une *classe*, si l'on emploie le vocabulaire de la programmation orientée-objets ; tandis qu'une sorte visible correspond au nom d'un type de donnée élémentaire (comme la sorte `Int` des entiers). Ensuite, sont distinguées les opérations appelées «observations», qui comportent une sorte cachées parmi les sortes en arguments, et qui retournent une sorte visible ; et les «méthodes», qui comportent une sorte cachée parmi les arguments, et retournent une sorte cachée. Une «expérience» sur un état (de sorte cachée) est alors une succession de méthodes appliquée sur cet état, suivie d'une observation pour observer le résultat.

Dans nos spécifications ci-dessus, la sorte cachée serait `MachineState` ; la seule fonction d'observation (un foncteur en fait, dans le cadre de la RWL) est alors $[\![$`.val`$]\!]$, et nous n'avons aucune «méthode» pour cette sorte : donc deux états machine sont «comportementalement équivalents» si les variables y ont les mêmes valeurs. (Pour avoir une spécification «pure», il faudrait accéder aux algorithmes, instructions et traces de nos états machines également via des fonctions d'observation.)

Nous n'avons pas utilisé la notion d'observation comportementale, mais nous aurions pu, pour une partie des équations. Ainsi l'équation de la théorie `MACHINE` qui spécifie l'égalité des valeurs des variables sur les états initiaux s'écrirait :

```
vars algo algo' : Algo
beq (M wTrace: noInst wAlgo: algo)
  = (M wTrace: noInst wAlgo: algo') .
```

où le mot-clé CafeOBJ «`beq`» est pour «*behavioural equation*», signifiant que les termes de gauche et de droite notent des objets comportementalement équivalents.

Notons qu'égalités comportementales et transitions ne sont pas incompatibles. Nous n'avons pas utilisé d'égalités comportementales ci-dessus pour ne pas rendre la description plus complexe que nécessaire.

Méthodes

Les spécifications cachées usuelles ne comportent pas de transitions : les transitions entre états sont uniquement exprimées par des «méthodes».

Donc, nous aurions pu exprimer les transitions à partir d'une méthode «`.executeInst`» (avec un état machine pour seul argument). Le prédicat «`==>`» utilisé dans `ATL-EFFECTIVE-SEM`, aurait alors été traduit par une méthode «`.executeAlgo`», définie à partir de `.executeInst`. La raison pour laquelle nous ne l'avons pas suivi cette approche est que nous voulons mettre l'accent sur la notion de graphe d'exécution. En effet, les flèches d'une catégorie (dans la version RWL utilisée) forment naturellement des graphes, tandis que le graphe «dessiné» par une fonction $[\![\texttt{.executeInst}]\!]$ doit être défini comme suit : ses noeuds sont les états machine, et une flèche existe d'un état s vers un état s' si $[\![\texttt{.executeInst}]\!](s) = s'$. Le graphe dessiné, dans la version RWL, par la catégorie des états machine (sans les flèches composées) est isomorphe au graphe défini ci-dessus ; mais tandis que le graphe dans la catégorie est visualisé de façon naturelle comme un graphe, visualiser le graphe résultant de la construction ci-dessus à partir de la fonction $[\![\texttt{.executeInst}]\!]$ demande, présumons-nous, un effort d'imagination qui n'est pas demandé par la version RWL avec ses modèles catégoriques.

Notons que la discussion ci-dessus présuppose que les modèles doivent exprimer la sémantique opérationnelle du langage. Il est important ici de mentionner comment l'approche de Goguen et Malcolm [37], déjà mentionnée ci-dessus (2.3) se distingue de la nôtre. Dans [37], la sémantique d'un langage est donnée avec des spécifications algébriques, un peu à notre façon. Mais les modèles n'y expriment pas du tout la sémantique opérationnelle du langage : celle-ci est donnée par *la réécriture* correspondant à la lecture des équations (de gauche à droite) comme des transitions. En effet, comme il n'y a pas de transitions dans ces spécifications, aucune transition entre états n'existe au niveau des modèles : ceux-ci sont donc ignorés pour ce qui est de la sémantique opérationnelle. Et comme une sémantique opérationnelle suppose des transitions, les équations sont lues (de gauche à droite) comme des règles de réécriture. Or, la plupart de ces équations n'ont pas une signification «opérationnelle» claire. C'est le cas, par exemple, de l'équation «`eq [[ I ]] = I .`» déjà citée. C'est pour éviter ce genre de problème que nous préférons nous référer aux modèles : mais cela implique bien sûr de définir comment ceux-ci représentent la sémantique opérationnelle du langage, ce dont nous traitons ci-dessus avec des méthodes comme `.executeInst` et `.executeAlgo`.

Spécifications cachées et preuves

Nous pouvons également mentionner que comme une version «cachée» de nos spécifications ne contiendrait que des équations (comportementales et non-comportementales), sans transitions, il serait plus aisé de faire des preuves à l'aide de ces spécifications, puisque nous n'aurions pas à distinguer entre transitions «vraies» et équations-transitions. Mais comme nous l'avons indiqué, le propos de nos spécifications n'est pas de permettre de faire facilement des preuves de correction de programmes, mais d'exprimer le plus clairement possible les sémantiques officielle et effective (et les deux objectifs ne sont pas forcément compatibles sur tous les aspects des spécifications).

Chapitre 3

La sémantique des directives : application au parallélisme de données

Dans ce chapitre nous appliquons la distinction sémantique officielle/sémantique effective à l'analyse de deux langages data-parallèles, dans le but de clarifier leur sémantique.

Le premier langage est le langage industriel HPF. La notion de directive de placement sera étudiée, et la notion de *préférence* quant à la prise en compte d'une directive par le compilateur sera introduite.

Le second langage est le langage académique PEI, dont le propos est de lier considérations de haut niveau et considérations de placement de données (sans directives au sens de HPF). La sémantique de PEI, telle qu'elle est donnée dans la littérature, n'est pas facile à appréhender : nous montrerons comment la notion de *directive automatique* permet de rendre compte plus facilement de la sémantique de PEI, et comment elle permet d'énoncer les problèmes que peut rencontrer le programmeur PEI.

3.1 Sémantique des directives de HPF

3.1.1 Le problème des directives de HPF

Les directives et le programmeur

Le langage HPF est fondé sur la notion de directive au compilateur : ces directives concernent le placement des données sur les mémoires associées aux processeurs d'une machine parallèle.

Or, on peut relever deux problèmes avec ces directives :

1. Le compilateur n'est pas obligé d'en tenir compte. En conséquence, nous nous trouvons dans le cas où le programmeur doit apprendre son compilateur, ce qui, comme nous l'avons montré au chapitre 2, n'est pas une bonne chose pour la portabilité de la performance du programme. Or, le problème est grandement accentué ici du fait que l'intérêt du langage HPF est dans ces directives mêmes.

2. En outre, le modèle HPF pour la notion d'*alignement* des données n'est pas aussi simple qu'il le pourrait (ce que nous argumentons en section 3.2.3). Ce problème est accentué par le fait que la plupart des utilisateurs de HPF ne sont pas des informaticiens, mais des ingénieurs ou des scientifiques qui ont besoin de faire du calcul scientifique.

Dans le cas des directives HPF, les compilateurs existants ne signalent pas, à notre connaissance, la façon dont ils prennent en compte les directives. En conséquence, s'il souhaite le savoir, le programmeur peut tout au plus procéder par induction «avec probabilités», en examinant l'exécution de son programme, avec un outil comme `pgprof`. `pgprof` est un profiler, et permet par exemple de compter les communications générées par une ligne de code. Cependant, il ne détaille pas le placement respectif des données : celui-ci peut donc seulement être inféré à partir du nombre de communications. Si ce placement est très semblable à celui indiqué par les directives, le programmeur peut supposer que cela provient de l'utilisation des directives. L'inconvénient de cet état de fait est que comme le compilateur ne produit pas de feedback quant à son utilisation des directives, le programmeur ne peut pas facilement optimiser ses directives. En outre, écrire une directive demande un investissement de la part du programmeur : il est donc fâcheux que celui-ci n'ait pas un retour explicite sur l'utilité de son investissement.

Ce problème n'en serait pas un s'il était connu que les compilateurs HPF prennent en compte et respectent systématiquement chacune des directives de placement. Or, cela n'est pas le cas. En effet, la conception du langage HPF a pris de l'avance sur l'état de l'art en matière de compilation, de sorte que les compilateurs actuels ne sont effectivement pas en état de garantir que chacune des directives est respectée. (Sur les problèmes de compilation liés à HPF, voir par exemple [18]).

Ce qui qualifie un compilateur

Notre thèse est que le problème des directives provient de la pauvreté des nuances de contrainte, dans la documentation d'un langage comme HPF, au sujet des devoirs du compilateur.

C'est-à-dire que les descriptions des devoirs d'un compilateur ont toujours suivi le schéma d'une *norme de qualification* : «many regulations, including many embodied in legislation, may be viewed as *qualification norms*, which specify the conditions under which some entity x counts as an entity of a particular type F» [59]. Dans le cas des langages de programmation, le manuel de référence définit, au sens légal, les conditions que doit remplir un logiciel pour être un compilateur pour le langage. Un exemple est donné par la sémantique effective, dans le chapitre précédent : un logiciel est un compilateur pour le langage ATL s'il produit un code objet dont l'exécution est un modèle de la théorie qui *définit* la sémantique effective du programme.

Dans un schéma de qualification, soit une condition de qualification est satisfaite, soit elle ne l'est pas : il n'y a pas de nuance intermédiaire. Or, la signification des directives est dans une nuance intermédiaire. En effet, un compilateur qui ignore systématiquement les directives HPF, comme le ferait n'importe quel compilateur pour Fortran 95, est automatiquement qualifié, d'après la documentation HPF, comme un compilateur HPF. Or, il est clair que le programmeur quant à lui ne considérera pas sérieusement un tel compilateur comme un compilateur HPF.

Or, lorsque la notion de directive est introduite dans un langage de programmation, deux possibilités sont offertes au rédacteur de la norme du langage :

1. Déclarer que les directives ont le statut de commentaire (c'est la position de la documentation HPF). Leur traitement dépend alors du compilateur.
2. Déclarer qu'un logiciel est un compilateur de son langage seulement s'il satisfait toute directive présente dans un programme écrit dans ce langage.

Pour le programmeur, comme nous l'avons observé, la première solution n'est pas idéale. Cependant, le texte de la norme peut être affiné afin de mieux rendre compte du rôle des directives. Ce sera le sujet des prochaines sections. La deuxième solution, avec les directives toujours respectées, sera étudiée ensuite, dans la section 3.2.2.

Le corollaire de la description formelle utilisée pour affiner la sémantique des directives est qu'un modèle simple de la notion d'alignement y est donnée, qui est utilisé dans la discussion de la section 3.2.3.

3.1.2 Rappel sur les directives de HPF

Nous donnons d'abord un rappel sur les directives de HPF. La version de HPF discutée est la version 2, la plus récente [25]. Cette version de HPF est une extension du langage Fortran 95 [61], tandis que la première version était une extension du langage Fortran 90 [56].

Pour expliquer les directives de HPF, nous choisissons de les expliquer sur un exemple simple de sous-routine HPF :

```
      SUBROUTINE ALGO1 (I, J, A, B, C)
      INTEGER I, J
      INTEGER, DIMENSION(4) :: A, B, C

      INTEGER K, L
      INTEGER, DIMENSION(4) :: D

!HPF$ TEMPLATE T(4)
!HPF$ ALIGN A(:) with T(:)
!HPF$ ALIGN B(:) with T(:)
!HPF$ ALIGN D(:) with T(:)

!HPF$ PROCESSORS, DIMENSION(2) :: MY_PROCS
!HPF$ DISTRIBUTE (BLOCK) ONTO MY_PROCS :: T

      K = I + J
      L = K + K
      D = A + B
      C = D + L

      END
```

Dans ce texte HPF, les directives de compilation sont les lignes qui commencent par le mot-clé « `!HPF$`». Le caractère « `!`» en début de ligne signale un *commentaire* Fortran 95. Les directives concernent le placement de données sur les processeurs (voir ci-dessous). La documentation HPF déclare : «Data mapping is specified using directives that can aid the compiler in optimizing parallel performance, but have no effect on the semantics of the program» [25, p. 5]. C'est-à-dire que la norme HPF, dans son état actuel, ne considère pas que le placement des données fait partie de la «sémantique» d'un programme HPF.

Affectations de tableaux

Dans le programme ci-dessus, deux instructions sont des instructions sur des tableaux : «`D = A + B`», et «`C = D + L`». Ce sont des instructions Fortran 95. La signification de la première est que pour tous les indices du tableau `D`, i.e. pour tout `I` de 1 à 4 (la taille du tableau), la case d'indice `I` prend la valeur `A(I) + B(I)`. Dans la deuxième instruction, le deuxième argument de l'addition est un scalaire. Cela signifie que pour tout `I` de de 1 à 4, la case d'indice `I` de `C` prend la valeur `D(I) + L`.

Placement des données

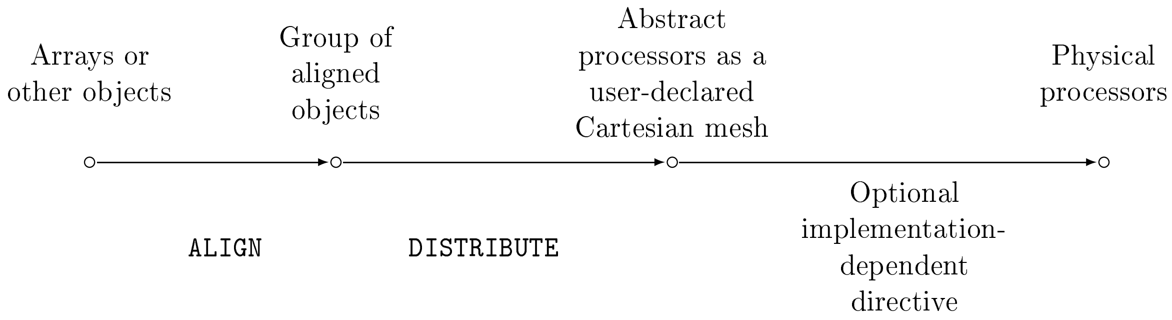
Pourquoi des directives de placement ? À propos de ces instructions de tableaux, la documentation de Fortran 90, le prédécesseur de Fortran 95, déclare : «these features can significantly facilitate optimization of array operations on many computer architectures» [56]. Si une machine parallèle est utilisée, l'optimisation peut consister à effectuer des affectations de valeurs en parallèle. En effet, étant donné que, dans chacune de ces instructions, les affectations des cases sont indépendantes les unes des autres, celles-ci peuvent être effectuées en parallèle sur une machine à plusieurs processeurs.

Sur une machine parallèle la mémoire peut être *partagée* entre les processeurs. Les valeurs des tableaux sont alors *distribuées* parmi les mémoires associées aux processeurs. Par suite, lorsque ces valeurs sont requises pour des calculs, des communications inter-processeurs sont nécessaires, lorsqu'un calcul s'effectuant sur un processeur a besoin de données qui se trouvent sur les mémoires associées à d'autres processeurs.

Or, les communications inter-processeurs sont *coûteuses* en temps. Comme l'objectif dans l'utilisation d'un ordinateur parallèle est de réduire le temps d'exécution d'un programme, il est important que les communications inter-processeurs ne prennent pas une part trop importante du temps d'exécution du programme, sans quoi on perd l'avantage d'utiliser un machine parallèle, jusqu'à éventuellement obtenir une performance pire qu'avec un seul processeur.

Il s'agit donc pour le compilateur de décider d'un placement des données sur les processeurs tel que le nombre de communications inter-processeurs à l'exécution soit minimisé. Or le problème qui consiste à trouver le meilleur compromis entre un ordonnancement optimal des instructions et un placement optimal pour la minimisation des communications est connu comme étant NP-complet. Le compilateur n'a donc pas une tâche facile. D'où l'idée de permettre au programmeur de donner des indications au compilateur quant à un placement possible, en spécifiant soit directement le placement d'une donnée sur un processeur, soit le placement relatif de deux données. C'est ce à quoi servent les directives HPF.

Le modèle de placement de données de HPF Nous allons expliquer le modèle de placement de HPF à partir de l'exemple de programme donné ci-dessus. Pour ses directives, HPF propose le modèle suivant pour permettre au programmeur d'exprimer un placement des données qui minimise des communications. Le diagramme ci-dessous est repris de [25].



Liberté du compilateur Nous expliquons les directives du programme, mais devons préciser, comme il est fait dans la documentation HPF, que «the explanations given do not necessarily reflect the actual compilation process» [25, p. 6]. En effet, d'après la documentation, «HPF directives appear as structured comments that suggest implementation strategies [...] about a program to the compiler.» [25, p. 4], et rien de plus—c'est-à-dire qu'aucune contrainte sur le compilateur n'est introduite. Par suite, l'explication donnée ci-dessus n'a de signification que dans le cas où le compilateur utilise effectivement les informations données dans les directives. Nous la donnons en supposant donc que c'est le cas.

Localité des données L'exemple donné dans le programme ci-dessus consiste à faire en sorte que, étant donnés 2 processeurs, la valeur `A(I)` se trouve sur le même processeur que la case de tableau `D(I)` ; et de même entre `B(I)` et `D(I)`. Ce qui est implicite est que la règle dite '*Owner Computes Rule*' s'applique : i.e., chaque affectation est effectuée par le processeur qui stocke la case où doit être mis le résultat, ici `D(I)`. C'est pourquoi, si la case `A(I)` était sur un processeur p' différent du processeur p sur lequel on trouve la case de tableau `D(I)`, une communication de la valeur `A(I)` du processeur p' vers le processeur p serait nécessaire.

La réalisation de cette localité des données entre les valeurs des tableaux `A`, `B` et `D` est réalisée de la façon suivante. Les tableaux concernés sont d'abord *alignés* sur un *template* `T`, avec des directives `ALIGN` ; puis ce template est *distribué*, avec la directive `DISTRIBUTE`, sur des processeurs abstraits déclarés avec la directive `PROCESSORS`. La distribution est faite par blocs («`(BLOCK)`»). C'est-à-dire que le template est découpé en deux blocs—un pour chaque processeur (puisqu'un tableau de deux processeurs a été déclaré). Le premier bloc est constitué des cases `T(1)` et `T(2)` du template ; et le deuxième bloc des cases `T(3)` et `T(4)`.

Au sujet des templates, la documentation HPF dit : «A template is simply an abstract space of indexed positions ; it can be considered as an 'array of nothing' (as compared to an 'array of integers', say).» [25, p. 40]. La notion de template n'est pas triviale, à cause justement de la «ressemblance» entre un template et un tableau. Dans ce qui suit, nous allons nous attacher à préciser cette notion.

Intuitivement, on peut cependant comprendre que les directives `ALIGN` spécifient que, pour chaque index `I`, la valeur `A(I)` se trouve sur le même processeur que la «non-valeur» `T(I)` ; et de même pour les tableaux `B` et `D`. Par transitivité, cela implique effectivement que la valeur `A(I)` se trouve sur le même processeur que la valeur `D(I)` ; et de même pour la valeur `B(I)`.

Quant au placement du template sur les processeurs (on parle alors plutôt de la *distribution* du template), il implique le placement des valeurs alignées sur ce template. Dans l'exemple, le type de distribution du template sur les processeurs n'influe pas sur le nombre de communications générées par l'affectation `D = A + B`, qui est nul si les directives `ALIGN` sont respectées par le compilateur. Notons que comme le tableau `C` n'a été aligné avec aucun autre tableau ou template, rien n'est dit par ce code sur les communications générées par l'instruction `C = D + L`.

3.1.3 Le langage MINI-HPF

Nous souhaitons clarifier le statut sémantique des directives. L'idée est que cela permettra de mieux concevoir la rédaction d'un texte de norme concernant des directives, comme pour HPF. Concrètement, le programmeur sait que du point de vue de la norme les directives sont de simples commentaires. Mais d'un autre côté, l'implanteur du compilateur sait que le programmeur attend une certaine **qualité** de compilation de la part du compilateur—ce qui amène celui-ci à ne pas simplement ignorer les directives comme de simples commentaires. C'est-à-dire qu'est introduite la notion de *préférence* de la part de l'utilisateur du compilateur.

Notre étude de la sémantique des directives utilise, comme annoncé, une expression formelle pour garantir un modèle clair. Nous donnons pour cela une sémantique formelle au sous-ensemble du langage HPF qui est utilisé dans le programme exemple ci-dessus. Nous appelons «MINI-HPF» ce langage restreint. Nous utilisons, pour donner la sémantique de MINI-HPF, la méthode exposée au chapitre précédent. C'est-à-dire que nous décrivons la sémantique de référence de MINI-HPF, comme la somme d'une sémantique officielle et d'une sémantique effective.

Pour le langage MINI-HPF, comme pour le langage ATL au chapitre précédent, nous nous concentrons davantage sur la sémantique que sur la syntaxe concrète. Nous définissons donc la syntaxe abstraite de `MINI-HPF-CONSTRUCTS`, par la théorie suivante :

```
module MINI-HPF-CONSTRUCTS {
  protecting (STRING)
  [ SAssignt ]      ** sort of the simple assignments
     op _:=s_+_ : String String String   -> SAssignt
                    ** scalar + scalar

  protecting (INT)
     op length_   : String -> Int
                    ** array length

  [ ArrayAssignt ]  ** sort of the array assignments
```

```
  [ String ]
     op _:=as_+_  : String String String -> ArrayAssignt
                    ** array + scalar
     op _:=aa_+_  : String String String -> ArrayAssignt
                    ** array + array

  [ SAssignt ArrayAssignt < Assignt < Algo ]
     op noAlgo  :              -> Algo
     op _;_     : Algo Algo    -> Algo
                 { assoc id: noAlgo }

  op input-var_  : String -> Bool
  op output-var_ : String -> Bool
}
```

Cette théorie ne contient pas de syntaxe abstraite destinée à formuler les directives. La raison est que nous ne traitons pas les *ré-alignements* et *re-distribution* (voir plus bas) : donc les alignements et distributions valent «pour tout le programme», et il n'y a pas besoin d'introduire des «marqueurs syntaxiques» dans le texte d'un algorithme pour indiquer des changements d'alignement ou de distribution. D'autre part les directives sont suffisamment simples pour être exprimées directement à partir d'opérations simple concernant la machine effective (comme il sera fait dans la section 3.1.5 ci-dessous) : aucune syntaxe abstraite n'est donc nécessaire pour les directives.

Les chaînes de caractère (`String`) sont utilisées comme identifiants pour les variables scalaires comme pour les tableaux. Cela signifie qu'un même identifiant peut être utilisé, dans un programme, pour référer à la fois une variable scalaire et une variable de type tableau. Cela n'est pas possible en HPF : cependant, si l'on considère les théories correspondant à des programmes HPF comme étant générées automatiquement à partir de ces programmes, cette possibilité de surcharge ('overloading') des noms de variables n'est tout simplement pas utilisée.

Avec les opérations de `MINI-HPF-CONSTRUCTS`, le texte de la sous-routine HPF `ALGO1` ci-dessus peut être considéré (directives non comprises) comme une expression syntaxique concrète pour la théorie suivante, de même nom :

```
module! ALGO1 {
  using (MINI-HPF-CONSTRUCTS)
  op algo1 : -> Algo

  var x : String
  eq  input-var x =  x == "I" or x == "J" or x == "A" or x == "B" .
  eq output-var x =  x == "C" .

  eq length "A" = 4 .  eq length "B" = 4 .
```

```
  eq length "C" = 4 .  eq length "D" = 4 .

  eq algo1 =  ("K" :=s  "I" + "J") ; ("L" :=s  "K" + "K") ;
              ("D" :=aa "A" + "B") ; ("C" :=as "D" + "K") .
}
```

Si l'on considère cette théorie comme étant générée automatiquement à partir du texte HPF, alors les variables en entrée et en sortie de la sous-routine sont déterminées à partir des arguments de la sous-routine et d'une analyse simple du code. En effet, les entrées et sorties apparaissent forcément dans les arguments de la sous-routine ; et parmi ces variables les entrées sont les variables qui n'apparaissent pas en membre gauche d'une instruction.

3.1.4 Sémantique officielle de MINI-HPF

La documentation HPF déclare : «Data mapping is specified using directives that can aid the compiler in optimizing parallel performance, but have no effect on the semantics of the program» [25, p. 5]. Autrement dit, dans le vocabulaire que nous avons développé au chapitre précédent, HPF est un langage qui utilise la sémantique officielle de Fortran 95 comme sémantique officielle, et dont la sémantique effective est un *raffinement* de la sémantique effective de Fortran 95, lequel consiste à prendre *peut-être* en compte la notion de placement des données (voir la section suite sur la formalisation du «peut-être»).

Ainsi, la sémantique officielle des directives HPF est nulle (ce sont des commentaires Fortran 95), mais leur sémantique effective ne l'est «pas forcément».

La machine

Pour la description de la sémantique officielle, comme pour celle de la sémantique effective ci-après, nous utilisons comme au chapitre précédent une structuration en machines, graphes d'exécutions, et spécification des effets des transitions entre états machine.

La machine pour la sémantique officielle de MINI-HPF est une extension de celle que nous avons utilisée pour la sémantique officielle de ATL, au chapitre précédent. La seule différence est l'ajout d'une opération `_.val__` permettant d'observer les valeurs contenues dans les tableaux, en sus des valeurs des variables scalaires. Nous renommons `BASIC-MACHINE` la machine officielle du chapitre précédent, afin de pouvoir réutiliser le simple nom `MACHINE` dans les spécifications de ce chapitre.

```
module* MACHINE {
  extending (BASIC-MACHINE)

  op (_.val__) : MachineState String Int  -> Int

  var x : String  var i : Int   vars algo algo' : MAlgo
```

```
  eq (M wTrace: noInst wAlgo: algo ) .val x i
   = (M wTrace: noInst wAlgo: algo') .val x i .
}
```

Notons que pour cette opération, nous requérons le même schéma que pour l'opération `_.val_` du chapitre précédent, à savoir que nous requérons une sémantique lâche pour cette opération, et une sémantique initiale pour le reste des opérations dans les modules qui vont suivre.

Le graphe d'exécution

Comme pour ATL, la sémantique officielle de MINI-HPF exécute les instructions de l'algorithme les unes après les autres.

```
module* MINI-HPF-OFFICIAL-SEMO {
  using (MACHINE)
  protecting (MINI-HPF-CONSTRUCTS)

  [ Algo < MAlgo ]

  op <_> : Assignt -> Inst

  var tr : Trace    var algo : Algo   var assgnt : Assignt

  eq M wTrace: tr wAlgo: (assgnt ; algo)
   = M wTrace: tr toDo: < assgnt > wAlgo: algo .
}
```

Ainsi, pour le programme `algo1`, le graphe d'exécution est :

$$\begin{tikzcd} z_0 \arrow[d, "\{<\mathsf{a}_1>\}"] \\ z_1 \arrow[d, "\{<\mathsf{a}_2>\}"] \\ z_2 \arrow[d, "\{<\mathsf{a}_3>\}"] \\ z_3 \arrow[d, "\{<\mathsf{a}_4>\}"] \\ z_4 \end{tikzcd}$$

où z_0 est l'état initial ⟦M wTrace: noInst wAlgo: algo1⟧,
et «$\mathtt{a}_1$,» «$\mathtt{a}_2$,» «$\mathtt{a}_3$» et «$\mathtt{a}_4$» sont des noms pour les termes suivants de sorte `Assignt` :

$\mathtt{a}_1$ = `"K" :=s  "I" + "J"`,
$\mathtt{a}_2$ = `"L" :=s  "K" + "K"`,
$\mathtt{a}_3$ = `"D" :=aa "A" + "B"`,
$\mathtt{a}_4$ = `"C" :=as "D" + "K"`;

et où le nom de flèche «$\{\mathtt{a}_i\}$» indique que la flèche ainsi nommée, de la catégorie ⟦MachineState⟧, correspond à la transition d'un état `M wTrace: tr toDo:` <$\mathtt{a}_i$> `wAlgo: algo` (où `tr` est une trace et `algo` un algorithme) vers l'état `M wTrace: tr done:` <$\mathtt{a}_i$> `wAlgo: algo`.

L'effet des affectations

La définition de l'effet des affectations est similaire à celle utilisée pour ATL. La théorie, `MINI-HPF-OFFICIAL-SEM`, est plus longue que pour ATL car l'effet de chacun des deux types d'affectations de tableaux doit être défini. Cette théorie complète la définition de la sémantique officielle de MINI-HPF. Elle peut être trouvée en annexe A, p. 143.

Sémantique officielle du programme exemple

La sémantique officielle du programme HPF peut être donc être exprimée comme suit :

```
module ALGO1-OFFICIAL-SEM {
  using      (MINI-HPF-OFFICIAL-SEM)
  protecting (ALGO1)
}
```

C'est-à-dire que conformément au fait que les directives n'affectent pas la sémantique du programme «lu comme un programme Fortran 95», celles-ci, en tant que commentaires Fortran 95, ne sont pas représentées au niveau de la sémantique officielle du programme.

3.1.5 Sémantique effective de MINI-HPF

La sémantique effective MINI-HPF suit celle de HPF. Or la norme HPF donne le statut de commentaire aux directives. Donc un logiciel, pour être un compilateur HPF, doit satisfaire la sémantique effective de Fortran 95. Il peut ne tenir aucun compte des directives. Ainsi un compilateur Fortran 95 *est* un compilateur HPF.

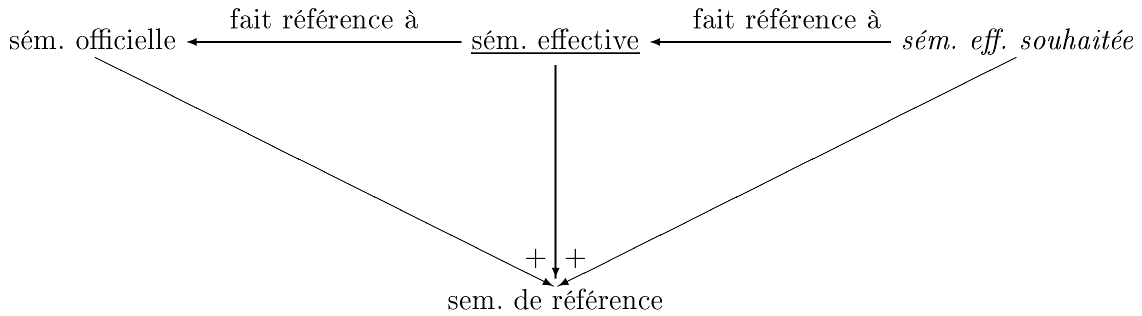
Cependant, la première chose qu'un programmeur attend d'un compilateur HPF, sur une machine parallèle, est qu'il tire parti des opérations globales comme les opérations de tableau pour effectuer des calculs en parallèle. En outre, le programmeur préfère a priori que les directives qu'il prend la peine d'écrire soient alors utilisées pour optimiser la localité des données. Cela revient donc à une définition par incitation de la sémantique de HPF : l'implanteur du compilateur sait que pour que son compilateur soit utilisé/vendu, il vaut mieux qu'il prenne en compte les possibilités de parallélisme et les directives.

Nous structurons donc la sémantique effective de MINI-HPF en deux parties :

1. la sémantique effective «obligatoire» de MINI-HPF est décrite : il s'agit de la sémantique effective de MINI-HPF au sens que nous avons donné à ce terme au chapitre précédent. Il s'agit donc en fait de Fortran 95 réduit au langage MINI-HPF.
2. Puis la sémantique effective *souhaitée* par le programmeur est décrite comme un *raffinement* de la sémantique effective obligatoire, et un ordre de préférence sur les implantations possibles de cette sémantique souhaitée est donné, suivant la satisfaction des directives.

L'intérêt d'expliciter le souhait de l'utilisateur est que le statut de la description du langage de directives par la norme est donné, comme pour le reste de la sémantique, par la notion de contrat entre le programmeur et le compilateur. La différence avec les contrats habituels, c'est que pour les directives le contrat n'a pas de termes d'*obligations* : il énonce seulement explicitement ce qui est souhaité par le programmeur.

La structure qui en résulte pour la sémantique de référence est illustrée avec le schéma suivant, où la sémantique à laquelle est contrainte le compilateur est soulignée, et la sémantique effective souhaitée mise en italique.



Outre une amélioration de la définition des directives pour une norme donnée, un apport de l'énonciation de la sémantique pour les directives est qu'il est ensuite possible d'articuler la façon dont on peut modifier une norme par rapport au statut des directives—c'est-à-dire rendre obligatoire une directive non obligatoire, ou l'inverse. Ce sujet sera abordé dans la section 3.2.2 ci-dessous.

Dans la présente section, la sémantique effective obligatoire de MINI-HPF est décrite. La sémantique effective souhaitée est donnée dans la section suivante, et la préférence entre les implantations satisfaisant la sémantique souhaitée dans la section 3.1.8.

La machine

Comme pour ATL, nous réutilisons la description de la machine officielle pour définir la machine de la sémantique effective. Des sortes et opérations sont renommées, de telle sorte que l'on obtienne une *copie* de la machine officielle. Ce renommage est identique à celui utilisé au chapitre précédent.

```
module* ACTUAL-SEM-MACHINE {
  protecting (MACHINE * {
        sort Inst         -> ActInst,
        sort Trace        -> ActTrace,
        sort MachineState -> ActMachineState,
        op   noInst       -> noActInst,
        op   M            -> AM
        })
}
```

Le graphe d'exécution

Le graphe d'exécution effectif pour Fortran95 est spécifié comme pour le langage C ou le petit langage ATL. La norme Fortran 95 déclare—les italiques sont les notres— : «With the following exceptions, the *effect of execution* is *as if* the executable constructs are executed in the order in which they appear in the main program or subprogram until a STOP, RETURN, or END statement is executed. The exceptions are the following :

(1) Execution of a branching statement [...]
(2) CASE constructs, DO constructs and IF constructs [...]
(3) END=, ERR=, EOR= specifyers [...]
(4) Alternate results may result in a branch.»[61, p. 14]

MINI-HPF est simple au point de ne contenir ni branchements ni structures de contrôle : nous n'avons donc pas d'exceptions à prendre en compte. Cette pauvreté de MINI-HPF n'est pas préjudiciable pour le présent exposé, celui-ci ayant pour propos principal la sémantique des directives : or les affectations de tableaux sont suffisantes pour traiter le cas des directives.

Donc la spécification de la sémantique effective de MINI-HPF suit le schéma exposé déjà pour ATL : la sémantique effective est exprimée en fonction de l'ordre d'exécution de la sémantique officielle.

```
module MINI-HPF-ACTUAL-SEMO {
  using (ACTUAL-SEM-MACHINE)
  protecting (MINI-HPF-CONSTRUCTS)

  [ Algo < MAlgo ]

  op <<_>>   : Algo -> ActInst

  var algo : Algo
  ceq AM wTrace: noActInst wAlgo: algo
    = AM wTrace: noActInst toDo: << algo >> wAlgo: noAlgo
      if algo =/= noAlgo .
}
```

Le graphe d'exécution pour l'algorithme `algo1` est alors le suivant :

$$\begin{tikzcd} y_0 \arrow[d, "\{\texttt{<<a}_1\texttt{ ; a}_2\texttt{ ; a}_3\texttt{ ; a}_4\texttt{>>}\}"] \\ y_1 \end{tikzcd}$$

—avec les mêmes conventions qu'auparavant quant aux états z_i et aux flèches $\{\texttt{<a}_i\texttt{>}\}$, et avec
$y_0 = [\![$`AM wTrace: noActInst wAlgo: algo1`$]\!]$
et $y_1 = [\![$`AM wTrace: << algo1 >> wAlgo: noAlgo`$]\!]$,
et $\{\texttt{<<a}_1\texttt{ ; a}_2\texttt{ ; a}_3\texttt{ ; a}_4\texttt{>>}\}$ un nom pour la transition de
$(y_0 =)$ $[\![$`AM wTrace: noActInst toDo: << algo1 >> wAlgo: algo1`$]\!]$
vers $[\![$`AM wTrace: noActInst done: << algo1 >> wAlgo: algo1`$]\!]$ $(= y_1)$.

Effet de l'exécution

L'effet de l'exécution est défini de la même façon que pour la sémantique effective du langage ATL, dans la section 2.5.3 du chapitre précédent. Le module décrivant cela, `MINI-HPF-ACTUAL-SEM`, complète la définition de la sémantique effective de MINI-HPF. Ce module et son commentaire sont donnés en annexe A, page 144.

La sémantique effective du programme exemple

La sémantique effective du programme HPF donné en exemple ci-dessus peut à présent être donnée, par la théorie suivante :

```
module ALGO1-ACTUAL-SEM {
  using      (MINI-HPF-ACTUAL-SEM)
  protecting (ALGO1)
}
```

3.1.6 Sémantique effective souhaitée de MINI-HPF

Nous pouvons à présent décrire la sémantique souhaitée par le programmeur, laquelle est sous-entendue par les directives.

D'un point de vue formel, la sémantique souhaitée va être codée comme un *raffinement* de la sémantique effective, au sens du chapitre précédent. Ce raffinement a un statut particulier : c'est «le raffinement souhaité par l'utilisateur» : c'est en cette qualité qu'il a place dans la description de la sémantique du langage MINI-HPF. Parmi les modèles de cette sémantique souhaitée, vont ensuite être préférés ceux qui prennent en compte les directives de placement de la façon la plus satisfaisante pour l'utilisateur.

La machine souhaitée

Nous définissons tout d'abord le raffinement de la sémantique effective que présuppose la prise en compte des directives. Comme le modèle d'exécution représenté par cette nouvelle sémantique est celui que l'utilisateur HPF souhaite que le compilateur réalise, nous l'appelons *sémantique effective souhaitée.*

Pour la définition de cette sémantique, nous utilisons le même schéma que précédemment : machine, graphe d'exécution, et effet des transitions.

Nous effectuons tout d'abord une copie de la machine effective (`ACTUAL-SEM-MACHINE`), en renommant certaines sortes et opérations, comme il a été fait pour réutiliser la machine officielle ; et nous enrichissons la machine obtenue avec des sortes, opérations et équations qui codent le *placement de données* pour les valeurs de tableaux et le nombre de *communications de données* entre les processeurs :

```
module* WISHED-FOR-ACTUAL-SEM-MACHINE {
  protecting (ACTUAL-SEM-MACHINE * {
          sort ActInst         -> WActInst,
          sort ActTrace        -> WActTrace,
          sort ActMachineState -> WActMachineState,
          op   noActInst       -> noWActInst,
          op   AM              -> WAM
          })

  [ Proc ]
  op nbProcs :      -> Int
  op noProc  :      -> Proc
  op P_      : Int -> Proc
  var i : Int
  ceq P i = noProc if i <= 0 or i > nbProcs .

  op (_.loc__)  : WActMachineState String Int -> Proc

  op (_.nbComm) : WActMachineState -> Int
    ** number of communications since the beginning of
    ** the execution of the program

  var algo : MAlgo
  eq (WAM wTrace: noWActInst wAlgo: algo) .nbComm = 0 .

  [ VProc ]
  op [_,_]      : String Int                -> VProc
  op _.map_     : WActMachineState VProc     -> Proc
  var S : WActMachineState   var x : String
  eq S .loc x i = S .map [ x , i ] .
```

```
}
```

La signification de ces nouvelles sortes, opérations et équations est à présent détaillée.

Processeurs La sorte `Proc` représente l'ensemble des **processeurs** de la machine souhaitée. C'est-à-dire que c'est une machine parallèle qui est souhaitée. L'entier `nbProcs` est le nombre de processeurs. Chaque processeur est identifié par un entier, avec l'opération `P_` ; l'entier qui identifie le processeur est compris entre 1 et $[\![$`nbProcs`$]\!]$ inclus : c'est ce qui est impliqué par l'équation «`P i = noProc if i <= 0 or i > nbProcs`», qui signifie qu'en-dehors de cet intervalle, `P i` ne référence aucun processeur—où «`noProc`» code «aucun processeur».

Remarquons que nous ne pouvons pas utiliser une sémantique initiale pour l'opération `nbProcs`, car cela amènerait l'élément $[\![$`nbProcs`$]\!]$ à être «un faux entier», c'est-à-dire un objet de la catégorie $[\![$`Int`$]\!]$ distinct des autres objets $[\![1]\!]$, $[\![2]\!]$, $[\![3]\!]$, …, ce qui n'est pas notre intention. Donc nous devons utiliser une sémantique lâche pour que $[\![$`nbProcs`$]\!]$ puisse être un vrai entier. Cependant nous requérons une sémantique initiale pour les autres opérations. Ce schéma est similaire à celui auquel nous avons dû faire appel l'opération `.val` ci-dessus et dans le chapitre précédent.

De par la sémantique initiale attachée à l'opération `P_`, la catégorie $[\![$`Proc`$]\!]$ comporte exactement $[\![$`nbProcs`$]\!]+1$ objets : les $[\![$`nbProcs`$]\!]$ objets $[\![$`P`$]\!]i$, avec $1 \leq i \leq [\![$`nbProcs`$]\!]$ (et i un objet de la catégorie $[\![$`Int`$]\!]$) ; et l'objet $[\![$`noProc`$]\!]$.
Le rôle de la sorte des processeurs dans la présente sémantique effective est celui d'*interface* «entre le physique et l'abstrait». En effet :

- la sorte des processeurs est utilisée ci-dessous pour exprimer le placement des données et les communications inter-processeurs ; et
- nous déclarons ici la convention suivante : **à tout objet de la catégorie $[\![$`Procs`$]\!]$ doit correspondre un processeur physique** (sauf pour l'objet $[\![$`noProc`$]\!]$!).

Ce schéma est en accord avec le fait que par la sorte `Proc` nous cherchons à *abstraire* la notion de processeur physique. Nous reflétons par là le modèle HPF, où sont distingués les *processeurs abstraits* ('abstract processors'), et les processeurs physiques, avec la déclaration suivante au sujet des processeurs abstraits : «The implementation then uses the same number, or perhaps a smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is implementation-dependant.» [25, p. 19] C'est-à-dire que le nombre de processeurs physiques réellement utilisés pourra en fait être inférieur à `nbProcs`.

Remarquons que le compilateur **pghpf** (de Portland Group) refusera en fait d'exécuter un programme si le nombre de processeurs qui lui est alloué est inférieur au nombre de processeurs abstraits (au sens de HPF) demandés par le programmeur avec la directive `PROCESSORS`. On constate alors l'ambiguïté du texte de la norme dans la déclaration ci-dessus, car il est difficile de savoir si le compilateur pghpf satisfait ou non la norme sur ce point du nombre de processeurs. En effet, en théorie, les directives étant des commentaires, le compilateur ne devrait pas refuser l'exécution pour des questions liées aux directives. Peut-être devons-nous alors considérer que pghpf se considère en fait comme compilateur pour la sémantique souhaitée de HPF que nous sommes en train de décrire, et pas seulement comme compilateur de la sémantique effective de Fortran 95.

Placement des données sur les processeurs L'opération «`.loc`» sert à coder le fait que la machine souhaitée est une machine à mémoire distribuée, et que les éléments de tableaux pour un programme en MINI-HPF sont stockés sur les mémoires associées aux processeurs. C'est-à-dire que la valeur $[\![\texttt{.val}]\!]\langle s, x, i\rangle$, associée sur un état machine s à l'identifiant de tableau x et à l'indice i, est stockée sur la mémoire associée au processeur $[\![\texttt{.loc}]\!]\langle s, x, i\rangle$. Cela est toujours le cas, que le tableau x soit concerné par une directive ou non, conformément au modèle énoncé dans la documentation HPF : «An object is considered to be *explicitely mapped* if it appears in an HPF mapping directive [...]; otherwise it is *implicitely mapped*» [25, p. 21] (où «mapped» réfère au placement des données sur les processeurs).

Remarque : HPF permet en fait de placer une valeur sur *plusieurs* processeurs. Nous pourrions représenter cela en donnant comme sorte d'arrivée pour `.loc` non pas la sorte des processeurs, mais une sorte d'ensemble de processeurs. Une telle extension n'est pas complexe, mais rallonge les spécifications, aussi dans le cadre de la présente exposition avons-nous préféré ne pas prendre en compte la réplication des valeurs. Celle-ci est prise en compte, cependant, dans la section suivante 3.3.

Communications inter-processeurs Le fait que le placement des données sur des processeurs génère potentiellement des communications inter-processeurs au cours de l'exécution d'un algorithme est représenté par l'opération `.nbComm`, qui donne, pour un état machine, le nombre de communications généré par l'exécution de l'algorithme depuis l'état initial jusqu'au présent état. Le comptage des communications sera spécifié dans le module `MINI-HPF-PREFERED-ACTUAL-SEM` ci-dessous. L'équation qui suit la déclaration de l'opération spécifie que dans un état initial aucune communication n'a encore eu lieu.

Remarque : Il pourrait être tentant de définir également le nombre de communications générées à l'exécution du programme *entre deux processeurs*. Cela correspondrait à une opération `_.nbComm__ : WActMachineState Proc Proc -> Int`, et permettrait de préciser le schéma de communications, en permettant de l'observer au niveau des échanges entre tout couple de processeurs. Cependant cela n'est possible que dans le contexte que la simplification que nous avons adoptée ci-dessus, qui consiste à ne placer une valeur que sur un processeur. En effet, dans le cas général où une valeur peut être placée sur plusieurs processeurs $p_1, p_2, \ldots, p_n$, pour une requête de cette valeur par un processeur p', la norme HPF ne précise pas sur quel processeur, parmi $p_1, p_2, \ldots, p_n$, doit être cherchée la valeur. Préciser cela est du ressort du compilateur. Comme nous ne souhaitons pas définir un schéma de compilation particulier, ni non plus lier notre spécification au cas particulier où chaque valeur est placée sur un seul processeur, nous ne spécifions pas d'opération `_.nbComm__ : WActMachineState Proc Proc -> Int`.

Processeurs virtuels La sorte `VProc` définit des **processeurs virtuels**. Ceux-ci nous servent à expliquer les directives de placement de HPF. La notion de processeur virtuel (dans le sens que nous allons définir) ne figure pas dans la documentation HPF : nous l'introduisons car elle nous permet d'expliquer de façon simple les directives de placement. En effet, l'opération `[_,_]` déclare que tout couple (chaîne de caractères, entier) identifie un processeur virtuel. Or un tel couple peut également identifier une «case de tableau». La différence avec une case de tableau est qu'un processeur virtuel ne stocke pas forcément une valeur : cela nous permet de rendre compte, dans la suite, des templates.

Les processeurs virtuels sont utilisés pour modéliser la notion d'alignement de tableaux. En effet, il est possible, pour un tableau x, de considérer que la valeur $[\![\texttt{.val}]\!]\langle s, x, i\rangle$, associée sur un état machine s à l'identifiant de tableau x et à l'indice i, est stockée sur la mémoire associée au processeur virtuel $[\![\texttt{[\_,\_]}]\!]\langle x, i\rangle$. Les processeurs virtuels sont reliés aux «vrais» processeurs de la sorte `Proc` par l'opération `.map`. Ainsi, si une valeur est associée à un processeur virtuel $[\![\texttt{[\_,\_]}]\!]\langle x, i\rangle$, elle est considérée comme se trouvant, pour un état machine s, sur le processeur $[\![\texttt{.map}]\!]\langle s, [\![\texttt{[\_,\_]}]\!]\langle x, i\rangle\rangle$. Cela est explicité par l'équation

```
S .loc x i = S .map [ x , i ] .
```

Le graphe d'exécution souhaité

Le propos des directives HPF est de minimiser le nombre de communications requises pour certaines instructions sur des tableaux. Il n'y aurait donc pas de sens de satisfaire cette directive sans exécuter les instructions correspondantes. La sémantique effective simple n'impose pas l'exécution des instructions sur des tableaux : le codage de la sémantique souhaitée doit donc expliciter ces instructions.

D'un autre côté, une directive de placement HPF concerne seulement quelques-unes des instructions de tableaux, et donc l'ensemble des directives ne concerne *a priori* qu'une sous-partie de toutes les instructions de tableaux d'un programme.

Cependant l'idée fondamentale de la coopération programmeur-compilateur dans le parallélisme de données est, rappelons-le, d'exhiber des instructions sur des tableaux car celles-ci exhibent un parallélisme inhérent, dont l'exploitation amène un temps d'exécution réduit. La prise en compte de *toutes* les instructions de tableaux dans la formulation de la sémantique souhaitée fait donc sens. C'est ce qui est codé par la spécification `MINI-HPF-WISHED-FOR-ACTUAL-SEMO` ci-dessous.

(Pour le cas où deux affectations de tableaux seraient indépendantes l'une de l'autre, elles devraient pouvoir être faites dans n'importe quel ordre. Afin de ne pas rallonger la spécification ci-dessous, nous ignorons ce cas, et déclarons l'exécution des affectations parallèles dans l'ordre qui est donné par le texte du programme. En effet, un traitement adéquat de la question des instructions indépendantes requiérerait une spécification de la notion de choix entre plusieurs ordres possibles, ou plus simplement de la notion de graphe de dépendances, ce qui nous amènerait trop loin. Nous n'aurions cependant pas forcément à spécifier l'analyse de dépendance elle-même, celle-ci pouvant être considérée comme étant effectuée par l'outil automatique qui génère la spécification correspondant à un programme.)

```
module MINI-HPF-WISHED-FOR-ACTUAL-SEMO {
  using (WISHED-FOR-ACTUAL-SEM-MACHINE)
  protecting (MINI-HPF-CONSTRUCTS)

  [ Algo < MAlgo ]
```

```
  op <<<_>>> : Algo -> WActInst

  op simple_ : Algo -> Bool

  var wactTr : WActTrace   vars algo algo' : Algo   var ar-assgnt : ArrayAssignt

  ceq WAM wTrace: wactTr                        wAlgo: (algo ; ar-assgnt ; algo')
    = WAM wTrace: wactTr toDo: <<< algo >>>     wAlgo: (ar-assgnt ; algo')
          if (simple algo) and algo =/= noAlgo .

  eq WAM wTrace: wactTr                         wAlgo: (ar-assgnt ; algo)
   = WAM wTrace: wactTr toDo: <<< ar-assgnt >>> wAlgo: algo .

  ceq WAM wTrace: wactTr                        wAlgo: algo
    = WAM wTrace: wactTr toDo: <<< algo >>>     wAlgo: noAlgo
          if (simple algo) and algo =/= noAlgo .

  ** def. of "simple":
  var s-assgnt : SAssignt
  eq simple  noAlgo           = true .
  eq simple (s-assgnt  ; algo) = simple algo .
  eq simple (ar-assgnt ; algo) = false .
}
```

La spécification utilise une opération `simple`, qui teste si un algorithme ne contient pas d'affectations de tableaux—c'est-à-dire s'il ne contient que des affectations de sorte `SAssignt`.

La première équation fait en sorte que toutes les instructions de sorte `SAssignt` qui précèdent une affectation de tableau soient exécutées en une transition, dont l'effet n'est pas spécifié dans le module suivant `MINI-HPF-WISHED-FOR-ACTUAL-SEM1` car nous ne sommes intéressés que par les affectations de tableaux.
La seconde équation donne une affection de tableau à exécuter à la machine.
La troisième équation traite le cas où seules des instructions de sorte `SAssignt` sont contenues dans l'algorithme restant à effectuer.

La définition du comportement de «`simple`» correspond à ce que nous avons dit de cette opération.

Effet sur les valeurs Une transition entre états, pour la sémantique effective souhaitée, a des effets qui peuvent être observés sur les valeurs *et* sur le nombre de communications inter-processeurs effectuées. Nous commençons par exposer l'effet sur les valeurs. Celui-ci fait de la sémantique effective souhaitée un raffinement de la sémantique effective. En effet, les entrées et sorties du programme sont les mêmes : seul change le graphe d'exécution, qui introduit des états intermédiaires que ne précisait pas la sémantique effective simple.

L'effet des transitions est décrit uniquement pour les affectations de tableaux :

```
ceq (WAM wTrace: wactTr ; <<< z :=as x + y >>> wAlgo: noAlgo) .val w i
  =   ((WAM wTrace: wactTr wAlgo: noAlgo) .val x i)
+ ((WAM wTrace: wactTr wAlgo: noAlgo) .val y)
      if w == z and 1 <= i and i <= (length w) .

ceq (WAM wTrace: wactTr ; <<< z :=a  x + y >>> wAlgo: noAlgo) .val w i
  =   ((WAM wTrace: wactTr wAlgo: noAlgo) .val x i)
+ ((WAM wTrace: wactTr wAlgo: noAlgo) .val y i)
      if w == z and 1 <= i and i <= (length w) .
```

L'effet de l'exécution de ces deux instructions sur les autres valeurs que celles du tableau affecté n'est pas précisé. La raison est que nous souhaitons seulement décrire une sémantique qui motive les directives d'alignement. Or ces dernières ne sont motivées que par les communications inter-processeurs générées par des calculs requis pour une affectation de tableau : donc nous spécifions seulement que ces calculs-là ont bien lieu. Nous ne spécifions pas encore, cependant, le fait que la règle 'Owner Computes Rule' s'applique : cela sera fait dans la spécification ci-dessous, de façon indirecte.

Le module dans son entier, `MINI-HPF-WISHED-FOR-ACTUAL-SEM1`, est donnée en annexe A, page 145.

Quant au parallélisme, son codage est similaire à celui utilisé dans le raffinement du chapitre précédent : la concurrence est *attribuée* par le lecteur de la spécification, à la transition. Les deux équations ci-dessus établissent le fait que dans l'exécution d'une affectation de tableau, le résultat du calcul pour un indice ne dépend pas du résultat du calcul pour un autre indice, ce qui signifie que ces calculs peuvent être effectués en parallèle.

Effet sur le nombre de communications Les directives de placement ont pour but de *minimiser le nombre de communications inter-processeurs* générées par l'exécution d'un algorithme. Le fait que le nombre de communications doit être de préférence minimisé est spécifié dans la section suivante. Dans la présente section le *nombre de communications* lui-même est spécifié, par la théorie suivante :

```
module MINI-HPF-WISHED-FOR-ACTUAL-SEM {
  using      (MINI-HPF-WISHED-FOR-ACTUAL-SEM1)

  ** nbComm involved in the execution of an algorithm:

  op  diff-l_____ : WActMachineState String WActMachineState String Int -> Int

  var wactTr  : WActTrace   var algo : Algo
  vars x y z : String       var i    : Int
```

```
  ceq (WAM wTrace: wactTr ; <<< algo >>>         wAlgo: noAlgo) .nbComm
    =    (WAM wTrace: wactTr wAlgo: noAlgo) .nbComm
           if (simple algo) .

  eq  (WAM wTrace: wactTr ; <<< z :=as x + y >>> wAlgo: noAlgo) .nbComm
   =    ((WAM wTrace: wactTr wAlgo: noAlgo) .nbComm)
      + (diff-l (WAM wTrace: wactTr ; <<< z :=as x + y >>> wAlgo: noAlgo) z
                (WAM wTrace: wactTr                        wAlgo: noAlgo) x
               (length z)) .

  eq  (WAM wTrace: wactTr ; <<< z :=aa x + y >>> wAlgo: noAlgo) .nbComm
   =    ((WAM wTrace: wactTr wAlgo: noAlgo) .nbComm)
      + (diff-l (WAM wTrace: wactTr ; <<< z :=as x + y >>> wAlgo: noAlgo) z
                (WAM wTrace: wactTr                        wAlgo: noAlgo) x
               (length z))
      + (diff-l (WAM wTrace: wactTr ; <<< z :=as x + y >>> wAlgo: noAlgo) z
                (WAM wTrace: wactTr                        wAlgo: noAlgo) y
               (length z)) .

  ** definition of "diff-l":
  vars S S' : WActMachineState
  op  diff-s_____ : WActMachineState String WActMachineState String Int -> Int
  eq  diff-l S' z S x 0 = 0 .
  ceq diff-l S' z S x i =  (diff-s S' z S x i) + (diff-l S' z S x (i - 1))
                             if (i >= 1) .
  ceq diff-s S' z S x i = 1  if (S' .loc z i) =/= (S .loc x i) .
  ceq diff-s S' z S x i = 0  if (S' .loc z i)  == (S .loc x i) .

  ** static location of the values:
  vars S S' : WActMachineState   var x : String   var i : Int
  eq S .loc x i = S' .loc x i .
}
```

La première équation spécifie qu'une transition ayant consisté à exécuter un ensemble d'affectations ne concernant pas des tableaux n'aura pas modifié le nombre de communications générées par l'algorithme. Cela parce que l'on ne s'intéresse qu'aux communications générées par des instructions agissant sur des tableaux, puisque ce sont les seules communications dont il est possible d'influencer le nombre par des directives de placement, ces dernières ne concernant que des tableaux. *C'est-à-dire que l'on ne compte que les communications inter-processeurs générées par des opérations sur des tableaux.*

Les deux équations suivantes comptent le nombre de communications inter-processeurs générées par une affectation de tableau—la première lorsque l'addition porte sur un tableau et un scalaire, la seconde lorsque deux tableaux sont additionnés. Dans les deux cas, l'expression

«`(AM wTrace: actTr wAlgo: noAlgo) .nbComm`» dans le membre de droite correspond au nombre de communications comptabilisées sur l'état précédent la transition. À ce nombre est rajouté le nombre de communications générées par l'exécution de l'affectation de tableau : le total est le nombre de communications comptabilisé par l'état apparaissant dans le membre de gauche, qui correspond au nombre total de communications générées depuis le début de l'exécution de l'algorithme.

Le comptage du nombre de communications générées par l'exécution de l'affectation de tableau est fait en considérant que la règle 'Owner-Computes-Rule' s'applique pour les calculs. C'est-à-dire que si l'exécution de l'affectation de tableau requiert, par exemple, pour le calcul de la valeur à l'indice i du tableau z, la valeur à l'indice i du tableau x, alors les faits suivants sont considérés :

- si l'état d'arrivée est s', alors la valeur calculée, $[\![\texttt{.val}]\!]\langle s', z, i\rangle$, est stockée sur le processeur $[\![\texttt{.loc}]\!]\langle s', z, i\rangle$. Cela implique, par la 'Owner-Computes Rule', que le calcul s'est effectué sur le dit processeur $[\![\texttt{.loc}]\!]\langle s', z, i\rangle$.
- Quant à la valeur requise, de valeur $[\![\texttt{.val}]\!]\langle s, x, i\rangle$ sur l'état de départ s, elle se trouve avant calcul sur le processeur $[\![\texttt{.loc}]\!]\langle s, x, i\rangle$. Cela suppose donc qu'elle doit être transférée, pour que le calcul puisse se faire, du processeur $[\![\texttt{.loc}]\!]\langle s, x, i\rangle$ vers le processeur processeur $[\![\texttt{.loc}]\!]\langle s', z, i\rangle$, si ces deux processeurs ne se trouvent pas être les mêmes.

Ceci s'applique pour tous les indices du tableau z. Aussi la spécification utilise-t-elle, avec l'opération `diff-l` (pour «différence sur toute la longueur») un comptage du nombre de différences de locations sur toute la longueur du tableau z. La définition de l'opération `diff-l` est simple : pour tous les indices sur la longueur considérée, la localisation $[\![\texttt{.loc}]\!]\langle s', z, i\rangle$ est comparée avec la localisation $[\![\texttt{.loc}]\!]\langle s, x, i\rangle$: il est compté 1 communication en cas de différence de localisation, et 0 sinon.

Remarque : Dans le cas où nous souhaitions spécifier le placement possible d'une valeur sur plusieurs processeurs, le comptage consisterait à compter, parmi l'ensemble de processeurs $[\![\texttt{.loc}]\!]\langle s', z, i\rangle$ sur lequel doit être stocké la valeur $[\![\texttt{.val}]\!]\langle s', z, i\rangle$, le nombre de processeurs qui n'appartiennent pas à l'ensemble de processeurs $[\![\texttt{.loc}]\!]\langle s, x, i\rangle$ sur lequel la valeur $[\![\texttt{.val}]\!]\langle s, x, i\rangle$ pourrait alors être trouvée. Cela est pris en compte dans [27], ainsi que dans la section suivante 3.3.

Enfin, le mode d'évolution des localisations est spécifié. L'équation «`S .loc x i = S' .loc x i`» indique que la location d'une valeur ne varie pas avec les états machines. C'est le cas par défaut dans HPF. Cependant, dans HPF, les directives `REALIGN` et `REDISTRIBUTE` permettent de faire varier la localisation des données au cours de l'algorithme : étendre MINI-HPF avec ces directives amènerait simplement à remplacer l'équation ci-dessus par des équations prenant en compte le changement possible de distribution et d'alignement au cours de l'exécution de l'algorithme.

La même équation ci-dessus implique, de par la dernière équation du module `ACTUAL-SEM-MACHINE`, qui relie `.map` à `.loc`, la satisfaction de l'équation suivante :

```
eq S .map [ x , i ] = S' .map [ x , i ] .
```

C'est-à-dire qu'il est impliqué que le placement des processeurs virtuels sur les «vrais» processeurs de la sorte `Proc` ne varie pas non plus avec les états machine.

La sémantique souhaitée du programme exemple

Nous pouvons à présent donner la sémantique souhaitée du programme exemple. Le nombre de processeurs donné par la directive «PROCESSORS» y est pris en compte, suivant en cela l'interprétation faite par le compilateur pghpf de cette directive (voir page 95 ci-dessus). L'aspect «identification des processeurs par des entiers» de cette directive est traduit dans la section suivante, qui traite également les autres directives. Pour résumer, la sémantique souhaitée du programme, que nous donnons ici est «brute», c'est-à-dire introduit le fait que le programme s'exécute sur une machine parallèle à mémoire distribuée avec un nombre donné de processeurs (ici, 2 processeurs—peut-être moins sur la machine «physique») : les directives (autres que «PROCESSORS») sont prises en compte dans la section 3.1.8 ci-dessous.

```
module ALGO1-WISHED-FOR-ACTUAL-SEM {
  using      (MINI-HPF-WISHED-FOR-ACTUAL-SEM)
  protecting (ALGO1)

  eq nbProcs = 2 .
}
```

3.1.7 La notion de préférence : aperçu

Dans la suite nous utilisons la notion de *préférence* pour énoncer la sémantique des directives. Nous modélisons la notion de préférence par une relation d'ordre partiel sur les modèles d'une théorie. Cette façon de modéliser la préférence est inspirée par la *logique déontique* [50]. La logique déontique traite la notion d'*obligation*, au sens «légal» en quelque sorte, c'est-à-dire au sens où la caractéristique première d'une obligation est qu'elle est susceptible de ne pas être respectée !—d'où le parallèle avec les directives d'HPF. Pour modéliser l'obligation, la logique déontique introduit une préférence entre plusieurs alternatives. Ces alternatives sont les «mondes possibles» ('possible worlds') traditionnellement utilisés comme modèles pour les logiques modales comme la logique déontique. La préférence est alors modélisée comme une relation d'ordre sur l'ensemble des mondes possibles.

Nous utilisons le lien évident entre les modèles d'une théorie et la notion de «monde possible», et dans notre cadre algébrique modélisons la préférence par une relation d'ordre sur l'ensemble des modèles d'une théorie.

Ainsi, soit une théorie de réécriture T, ainsi que l'ensemble de ses modèles, $\mathrm{MOD}(T)$. Une préférence entre des modèles T est codée par une relation d'ordre $\preccurlyeq$ sur ces modèles : pour deux modèles M et M', $M \preccurlyeq M'$ signifie : «M' est préféré à M».

Pour modéliser, dans la section suivante, la notion de directive, nous faisons en sorte de coder une directive comme une *équation* (comme nous l'avons fait pour la directive PROCESSORS) qui s'ajoute à une spécification, mais n'a pas obligation d'être satisfaite : la notion de satisfaction est remplacée par celle de préférence entre modèles. Ainsi, à la théorie T nous pouvons adjoindre un ensemble D d'équations utilisant les sortes et opérations de T ; puis il suffit de faire dépendre le relation de préférence $\preccurlyeq$ de la satisfaction ou non-satisfaction de ces équations par les modèles des T.

Un exemple simple de prédicat définissant une telle relation d'ordre est celui qui établit que si un modèle M de T satisfait exactement n équations de D, et un autre modèle M' satisfait, parmi les équations de D, *au moins* ces n équations, alors M' est préféré à M. Nous notons cette relation «$\preccurlyeq_D$». (Pour simplifier la présente discussion, nous disons que M' est préféré à M lorsque $M \preccurlyeq M'$, même si l'on a également $M' \preccurlyeq_D M$.) Cette relation de préférence est définie comme suit :

$$M \preccurlyeq_D M' \iff (\forall d \in D \ : \ M \models d \implies M' \models d) \ .$$

Intuitivement, cette relation de préférence suit la relation d'inclusion sur les sous-ensembles de D. En effet, notons $M {\models!}_D U$ dans le cas où M satisfait toutes les équations du sous-ensemble $U \subseteq D$ mais aucune de $D \setminus U$. Alors, si $M {\models!}_D U$ et $M' {\models!}_D V$, on a l'équivalence suivante :

$$U \subseteq V \iff M \preccurlyeq_D M' \ .$$

Travaux sur la préférence dans les spécifications algébriques La notion de préférence sur les modèles, dans le domaine des spécifications algébriques, n'a pas été modélisée auparavant, à notre connaissance.

La notion la plus proche que nous connaissons est celle des *semiotic morphisms* de Joseph Goguen [35]. Dans cet article J. Goguen utilise les spécifications algébriques ; les algèbres y représentent des *systèmes de signes* ('sign systems'), et les «morphismes sémiotiques» sont des morphismes particuliers, qui en particulier peuvent ne pas préserver des axiomes. Cette dernière particularité donne lieu à une relation d'ordre *sur les morphismes*, $\preccurlyeq$, avec $M \preccurlyeq M'$, où M et M' sont des morphismes sémiotiques, si M' préserve tout axiome que M préserve. Cette relation d'ordre est similaire à notre relation $\preccurlyeq_D$ sur les modèles. Elle est une des composantes permettant de comparer la *qualité* respective de deux morphismes sémiotiques, notion qui s'apparente à la «préférence». La différence entre cette notion d'ordre sur des morphismes et notre notion de préférence entre des modèles n'est pas tant technique que méthodologique : J. Goguen part d'un système de signe donné, et évalue les diverses images (qui sont d'autres systèmes de signes) qu'il est possible d'en faire en évaluant les morphismes correspondants. Dans notre application, aucun modèle particulier n'est donné, que nous mettrions en relation avec d'autres modèles.

Notons enfin que les 'partial algebras' [115] satisfont tous les axiomes d'une spécification ; ce sont les fonctions qui interprètent les opérations qui peuvent y être partielles.

Inversement, nous n'avons pas connaissance de travaux en logique modale utilisant les spécifications algébriques. Pourtant, celles-ci ont un intérêt certain pour parler de «mondes possibles», puisque les notions de modèle et de satisfaction y sont déjà définies.

Travaux sur la préférence et la programmation Les seuls travaux dont nous ayons connaissance liant formellement préférence et programmation sont ceux concernant la *programmation logique par préférence* ('preference logic programming' (PLP)) [42]. La notion de préférence y est également modélisée avec une relation d'ordre sur les mondes possibles. Cependant le codage n'est pas exactement le même que celui que nous utilisons : dans la PLP, un monde possible w

peut satisfaire ou non un prédicat exprimant la préférence (noté «$\mathcal{P}_f F$», où F est un prédicat du premier ordre), et la relation d'ordre indique que tout monde possible v satisfaisant F est «at least as good as w». La relation d'ordre sur les mondes possibles est alors en fait définie (contrainte) par ces prédicats de préférence. Dans notre modélisation, par contraste, un modèle ne satisfait pas une préférence : la préférence est uniquement dans la relation d'ordre, que nous définissons «de façon extérieure aux modèles», même si c'est en prenant en compte la satisfaction ou non d'un prédicat (en fait, d'une équation). L'avantage de cette approche est qu'elle nous permet de considérer plusieurs relations de préférence sur un même ensemble de modèles, ce qui sera utilisé dans la section 3.2 ci-dessous.

3.1.8 Préférence du programmeur

Pour rendre compte du fait que le compilateur HPF doit, si possible, prendre en compte les directives données par le programmeur, nous introduisons une notion de **préférence** entre les modèles de la théorie qui code un programme : cette préférence est celle du programmeur entre les différents modèles possibles (implantations) de son programme.

Les directives (autres que la directive `PROCESSORS`) sont pour cela codées comme des équations utilisant les sortes et opérations de la théorie `MINI-HPF-WISHED-FOR-ACTUAL-SEM`.

Ces directives sont attachées au programme pour lequel elles sont conçues : aussi les équations qui les traduisent sont-elles attachées à la théorie qui code le programme sans les directives. Cela est illustré ci-dessous avec la théorie `ALGO1-WISHED-FOR-ACTUAL-SEM`, qui modélise la sémantique souhaitée «brute» du programme HPF `ALGO1`. Nous considérons un ensemble d'«équations-directives» accompagnant cette théorie (et utilisant les sortes et opérations de cette théorie). Celles-ci n'ont pas le statut habituel de satisfaction obligatoire.

```
module ALGO1-WISHED-FOR-ACTUAL-SEM-WITH-DIRECTIVES {
  using      (ALGO1-WISHED-FOR-ACTUAL-SEM)

  var S : WActMachineState   var i : Int

  eq S .map [ "A" , i ] = S .map [ "T" , i ] .
  eq S .map [ "B" , i ] = S .map [ "T" , i ] .
  eq S .map [ "D" , i ] = S .map [ "T" , i ] .

  eq S .map [ "T" , i ] = P (((i - 1) quo nbProcs) + 1) .
}
```

Ces quatre équations traduisent respectivement les trois directives `ALIGN` du programme HPF de la page 83, et la directive `DISTRIBUTE`. Soit dir-ALGO1 cet ensemble d'équations. Nous lui adjoignons pour sémantique un ordre de préférence $\preccurlyeq_{\text{dir-ALGO1}}$ sur les modèles de `ALGO1-WISHED-FOR-ACTUAL-SEM`, au sens de la section précédente, c'est-à-dire que si un modèle satisfait exactement n équations parmi les équations de dir-ALGO1, alors tout modèle qui satisfait, parmi ces mêmes équations de dir-ALGO1, au moins ces n équations, lui est préféré. C'est-à-dire que le programmeur préfère la satisfaction de ses directives à leur non-satisfaction. Cette relation de préférence correspond à celle énoncée par la documentation (les italiques sont les notres) : «if two objects are mapped (via the two-level mapping of alignement and distribution) to the same abstract processor, it is *a strong recommendation* to the implementation that they *ought* to reside in the same physical processor.» [25, p. 20]. Des relations d'ordres alternatives seront considérées dans la section 3.2.1 ci-dessous.

La signification de ces quatre équations est à présent précisée. Il s'agit à chaque fois d'une précision sur la fonction $[\![\texttt{.map}]\!]$. ($[\![\texttt{.map}]\!]$ est bien sûr un *foncteur* avec pour domaine la catégorie produit $[\![\texttt{WActMachineState}]\!] \times [\![\texttt{VProc}]\!]$, mais l'action de $[\![\texttt{.map}]\!]$ sur les flèches de cette catégorie ne représente rien de pertinent dans le présent contexte, et donc nous simplifions en considérons uniquement l'action sur les objets.)

Alignement Les trois premières équations servent à préciser que $[\![\texttt{.map}]\!]$ fait correspondre à certains processeurs virtuels un même processeur «réel».

La première équation, par exemple, établit que pour tout i, et donc par implication pour tous les indices i du tableau de nom $[\![\texttt{A}]\!]$, le processeur virtuel associé $[\![\texttt{[\_,\_]}]\!]\langle[\![\texttt{A}]\!], i\rangle$ est placé sur le même processeur réel que le processeur virtuel $[\![\texttt{[\_,\_]}]\!]\langle[\![\texttt{T}]\!], i\rangle$. Pour le même indice i, et pour tout état s de la machine effective souhaitée, la troisième équation établit que le processeur virtuel $[\![\texttt{[\_,\_]}]\!]\langle[\![\texttt{D}]\!], i\rangle$ est placé sur le même processeur réel que $[\![\texttt{[\_,\_]}]\!]\langle[\![\texttt{T}]\!], i\rangle$—ce qui amène, par transitivité, l'égalité de placement des processeurs virtuels $[\![\texttt{[\_,\_]}]\!]\langle[\![\texttt{A}]\!], i\rangle$ et $[\![\texttt{[\_,\_]}]\!]\langle[\![\texttt{D}]\!], i\rangle$. On a donc, par la dernière équation du module ACTUAL-SEM-MACHINE, l'égalité suivante, pour tout état s de la machine :

$$[\![\texttt{.loc}]\!]\langle s, [\![\texttt{D}]\!], i\rangle = [\![\texttt{.loc}]\!]\langle s, [\![\texttt{A}]\!], i\rangle$$

—ce qui implique, par la convention sur la signification de l'opération .loc, que les valeurs $[\![\texttt{.val}]\!]\langle s, [\![\texttt{D}]\!], i\rangle$ et $[\![\texttt{.val}]\!]\langle s, [\![\texttt{A}]\!], i\rangle$ se trouvent sur le même processeur.

Notons la fonction remplie par le *template* T : cet identifiant sert simplement à introduire des processeurs virtuels intermédiaires de façon à énoncer une égalité de placement entre valeurs de tableaux via une transitivité sur le placement impliquant ces processeurs virtuels.

Le rapport exact de cette modélisation de la notion d'alignement des données à la notion d'alignement du modèle HPF est discuté en section 3.2.3 ci-dessous.

Placement La dernière équation définit le placement de certaines valeurs sur les processeurs, en précisant le comportement du foncteur $[\![\texttt{.map}]\!]$ pour les processeurs virtuels associés à la chaîne de caractères T. C'est à dire que le placement du «template» T est précisé. Comme les processeurs virtuels associés aux tableaux A, B et D sont «alignés» sur ceux du template T, cela définit par implication le placement de ces tableaux.

Dans le programme HPF (de la page 83), la directive DISTRIBUTE décrit un placement particulier du template, dit dans le vocabulaire HPF, «distribution par blocs». Ainsi, par exemple, les deux premières valeurs du tableau A sont placées sur le processeur $[\![\texttt{P 1}]\!]$, et les deux dernières sur le processeur ⟦P 2⟧. Le placement est défini en utilisant l'entier qui référence le processeur réel : cela constitue la «deuxième partie» de la sémantique de la directive `PROCESSORS` ; la première partie est la déclaration du nombre de processeurs, exprimée plus haut.

3.1.9 Préférence et compilation

La relation de préférence code la préférence du programmeur, et l'idée est que cela indique au compilateur quel choix faire entre deux modèles (implantations) possibles. Dans la norme actuelle, cette préférence n'est pas explicitée. En conséquence, c'est le compilateur qui choisit la sienne. Décrire la préférence précise ce à quoi doit tendre le compilateur, et par là ce à quoi peut s'attendre le programmeur. Normer la préférence revient donc bien à normer le comportement du compilateur.

Cependant, il faut être conscient que l'énonciation de la préférence n'est pas une énonciation de *comment* satisfaire au mieux le programmeur, mais une énonciation de *ce qui* satisfait au mieux le programmeur. Le «comment» concerne les techniques de compilation, qui ne sont pas du ressort de la norme.

Mais il nous faut tout de même considérer ce à quoi revient cette notion de préférence pour le compilateur, en pratique. La sémantique de la notion de préférence mentionne l'ensemble des modèles (implantations possibles) pour un programme. Or, en pratique, le compilateur ne peut analyser l'ensemble des modèles pour y trouver un modèle maximal. La situation est celle d'un problème d'optimisation avec un espace de recherche infini : quelle que soit la technique utilisée par le compilateur, ce dernier ne considérera jamais dans sa recherche qu'un sous-ensemble des tous les modèles existants. Il doit explorer l'espace des solutions, et rien ne garantit que le meilleur modèle qu'il saura construire est un élément maximal.

Le programmeur doit être conscient de cela. Or, c'est ce que sous-entend l'utilisation d'une relation de préférence car s'il était systématiquement possible de construire un élément maximal, il n'y aurait pas de préférence, mais la description par la norme d'une obligation par rapport aux directives.

3.1.10 Sémantique de référence pour MINI-HPF

Nous pouvons à présent énoncer la sémantique de référence de MINI-HPF, comme la théorie suivante :

```
module MINI-HPF-REFERENCE-SEM {
  protecting (MINI-HPF-OFFICIAL-SEM)
  protecting (MINI-HPF-ACTUAL-SEM)
  protecting (MINI-HPF-WISHED-FOR-ACTUAL-SEM)
}
```

accompagnée, pour tout programme, d'une relation de préférence sur les modèles de la sémantique effective souhaitée de ce programme, qui prend en paramètre un ensemble d'équations concernant cette sémantique effective souhaitée.

Ainsi la sémantique de référence du programme HPF ALGO1 est donnée par le module suivant :

```
module ALGO1-REFERENCE-SEM {
  using      (MINI-HPF-REFERENCE-SEM)
  protecting (ALGO1)

  eq nbProcs = 2 .
}
```

—accompagné de l'ordre de préférence $\preccurlyeq_{\textsf{dir-ALGO1}}$ sur les modèles de ALGO1-REFERENCE-SEM, qui concerne la satisfaction par la partie «sémantique effective souhaitée» des équations-directives suivantes :

```
  var S : WActMachineState   var i : Int

  eq S .map [ "A" , i ] = S .map [ "T" , i ] .
  eq S .map [ "B" , i ] = S .map [ "T" , i ] .
  eq S .map [ "D" , i ] = S .map [ "T" , i ] .

  eq S .map [ "T" , i ] = P (((i - 1) quo nbProcs) + 1) .
```

3.1.11 Les spécifications algébriques et les tests

Nous avons mentionné plus haut que toutes nos spécifications sont testées avec le système de réécriture CafeOBJ. La possibilité de tester les spécifications est un aspect non négligeable de l'approche par spécifications algébriques : les tests permettent d'une part de se rassurer sur la correction des spécifications, et d'autre part, par l'interactivité qu'ils permettent avec les spécifications, de mieux appréhender celles-ci. Nous donnons comme exemple le comptage des communications générées par le programme ALGO1, avec et sans directives : cela fournit un test de la correction de la spécification de la sémantique effective souhaitée.

Rappelons tout d'abord l'algorithme du programme ALGO1 :

```
eq length "A" = 4 .  eq length "B" = 4 .
eq length "C" = 4 .  eq length "D" = 4 .

eq algo1 = ("K" :=s  "I" + "J") ;
           ("L" :=s  "K" + "K") ;
           ("D" :=a  "A" + "B") ;
           ("C" :=as "D" + "K") .
```

Sans directives d'alignement Le test suivant compte le nombre de communications générées par `ALGO1` si les directives d'alignement ne sont pas données. C'est la machine de la sémantique effective souhaitée, `WAM`, qui est concernée. Rappelons que la sémantique effective souhaitée inclut le nombre de processeurs déclaré par la directive `PROCESSORS`, soit 2 processeurs pour `ALGO1`.

Les seules communications comptabilisées sont celles qui sont générées par des affectations de tableau. Seules deux telles affectations apparaissent dans l'algorithme de `ALGO1` : ce sont les deux dernières instructions. La première de ces instructions additionne deux tableaux. La dimension de ces tableaux est de 4 : pour chaque indice `I` de `D` (de 1 à 4), le processeur sur lequel doit être stockée la valeur correspondante requiert les valeurs de `A` et de `B` à l'indice `I`. Cela requiert au plus : $4 \times 2 = 8$ communications inter-processeurs.

Pour l'instruction suivante, au plus 4 communications sont nécessaires. Le total du nombre de communications générées par `ALGO1` doit donc être au plus de $8 + 4 = 12$ communications.

Le test suivant vérifie cela. La sémantique n'indique pas l'égalité de placement pour des valeurs de `A`, `B` et `D` : le système de réécriture de CafeOBJ comptera une communication inter-processeurs pour chaque dépendance entre données, *à moins d'une égalité explicite de placement*. Or de telles explicitations n'apparaissent pas sans les directives de placement. Le total comptabilisé doit donc correspondre au nombre de communications «au plus», soit 12.

Le mécanisme est le même qu'au chapitre précédent : l'argument de l'opération `.nbComm` va d'abord être évalué, puis l'opération `.nbComm` appliquée sur le résultat. Ici, l'argument est l'état initial pour `algo1`. L'évaluation de cet argument va mener à un état final, sur lequel le nombre de communications total généré par l'algorithme sera comptabilisé par `.nbComm`.

```
CafeOBJ> open ALGO1-REFERENCE-SEM
-- opening module ALGO1-REFERENCE-SEM.. done.
%ALGO1-REFERENCE-SEM> execute (WAM wTrace: noWActInst wAlgo: algo1 ) .nbComm .
-- execute in %ALGO1-REFERENCE-SEM : (WAM wTrace: noWActInst
                                      wAlgo: algo1) .nbComm
12 : NzNat
(0.000 sec for parse, 329 rewrites(0.090 sec), 848 matches)
%ALGO1-REFERENCE-SEM> close
CafeOBJ>
```

Nous obtenons 12 communications comme prévu.

Avec directives d'alignement Nous rajoutons à présent les équations-directives d'alignement et de placement. Elles sont données ici comme de vraies équations, donc ce sont les modèles qui les satisfont toutes qui sont considérés, et pour lesquels le nombre de communications générées est compté.

```
open ALGO1-REFERENCE-SEM
-- opening module ALGO1-REFERENCE-SEM.. done.
%ALGO1-REFERENCE-SEM> var S : WActMachineState
%ALGO1-REFERENCE-SEM> var i : Int
```

```
%ALGO1-REFERENCE-SEM> eq S .map [ "A" , i ] =  S .map [ "T" , i ] .
%ALGO1-REFERENCE-SEM> eq S .map [ "B" , i ] =  S .map [ "T" , i ] .
%ALGO1-REFERENCE-SEM> eq S .map [ "D" , i ] =  S .map [ "T" , i ] .
%ALGO1-REFERENCE-SEM> eq S .map [ "T" , i ] =  P (((i - 1) quo nbProcs) + 1) .
[Warning]: axiom : S .map [ "T" , i ] = P ((i - 1) quo nbProcs
                                          + 1)
    contains error operators.
%ALGO1-REFERENCE-SEM> execute (WAM wTrace: noWActInst wAlgo: algo1 ) .nbComm .
*
-- execute in %ALGO1-REFERENCE-SEM : (WAM wTrace: noWActInst
                                      wAlgo: algo1) .nbComm
4 : NzNat
(0.000 sec for parse, 809 rewrites(0.110 sec), 1697 matches)
%ALGO1-REFERENCE-SEM> close
CafeOBJ>
```

Nous observons que le total descend à 4 communications. Cela est cohérent : l'alignement de `D`, `A` et `B` fait en sorte que l'affectation `"D" :=aa "A" + "B"` ne génère aucune communication : il ne reste donc plus que les 4 communications de l'instruction suivante.

Remarquons que la distribution par blocs sur les processeurs ne diminue pas le nombre de communications pour la dernière instruction : cela est également cohérent, car au pire les données des tableaux `C` et `D` de la dernière affectation sont placées exactement sur le mauvais processeur les unes par rapport aux autres : par exemple pour l'indice 0, la valeur de `C` sur le premier processeur et la valeur de `D` sur le second. C'est ce cas pire qui est considéré par le système CafeOBJ car, comme pour l'alignement, en cas de non spécification explicite d'une égalité de placement, il suppose par défaut des placements différents.

Réécriture et modèles Pour effectuer les tests ci-dessus, il est en fait nécessaire de changer le mode d'importation du module `MINI-HPF-CONSTRUCTS`, dans le module `ALGO1` (en page 87) de `using` vers `protecting`. Cela parce que le système CafeOBJ a besoin, pour la réécriture, d'être sûr que le modèle de `MINI-HPF-CONSTRUCTS` importé par `ALGO1` est le même (à isomorphisme près) que celui importé par le module définissant la machine. Cela place le test dans un modèle sémantique différent de celui que nous avons défini, mais équivalent pour ce qui est des réécritures faites pour le test.

3.2 Discussion sur les directives de HPF

Dans cette section nous mettons à profit le cadre algébrique donné ci-dessus, et en particulier la notion de modèle, pour discuter des directives de HPF.

3.2.1 Préférences possibles

Nous avons choisi ci-dessus une notion de préférence simple ($\preccurlyeq_D$), qui dit au compilateur : «si tu arrives à respecter une directive de plus, le programmeur sera encore plus content». Nous examinons dans cette section quelques préférences alternatives.

Nombre de directives satisfaites

Une relation de préférence plus simple est celle selon laquelle le modèle M est préféré au modèle M' s'il satisfait *plus* d'équations-directives que M'. Si D est l'ensemble des équations-directives, cette relation de préférence se définit par :

$$\begin{gathered} M \preccurlyeq M' \\ \Longleftrightarrow \\ \mathrm{card}(\{d \in D \mid M \models d\}) \leq \mathrm{card}(\{d \in D \mid M' \models d\}) \end{gathered}$$

Si nous n'avons pas considéré cette relation de préférence auparavant, c'est parce que la relation de préférence doit refléter la préférence du programmeur, et qu'en pratique pour le programmeur toutes les directives ne sont pas «égales» : certaines sont plus importantes que d'autres pour minimiser l'ensemble des communications. C'est pour cela que nous avons préféré la relation $\preccurlyeq_D$, qui elle n'établit pas l'égalité de deux directives pour ce qui est de la préférence de leur satisfaction.

Minimisation des communications

Nous avons pris en compte, dans la relation $\preccurlyeq_D$, la satisfaction des directives, mais pas le nombre de communications, dont la minimisation est pourtant la motivation pour l'écriture des directives. Nous examinons donc à présent la possibilité d'une préférence prenant en compte la minimisation du nombre de communications.

Nous pouvons formuler une telle préférence par la relation suivante. Étant donné un programme `ALGO` et son ensemble de directives dir-ALGO, nous définissons la relation d'ordre $\preccurlyeq_{\text{min-com}}$ comme suit.

Soit $[\![$`WAM wTrace: wactTr wAlgo: noAlgo`$]\!]_M$ l'état final de la machine souhaitée atteint par l'exécution, dans M, de l'algorithme de `ALGO` ; et $[\![$`WAM wTrace: wactTr wAlgo: noAlgo`$]\!]_{M'}$ l'état final correspondant dans M'. Alors on définit :

$$\begin{gathered} M \preccurlyeq_{\text{min-com}} M' \\ \Longleftrightarrow \\ [\![\mathsf{final}\ \texttt{.nb-comm}]\!]_M \geq [\![\mathsf{final}\ \texttt{.nb-comm}]\!]_{M'} \end{gathered}$$

où «final» est une abréviation pour «`WAM wTrace: wactTr wAlgo: noAlgo`».

Relation avec le nombre de processeurs Si le programmeur ne fournit pas de directive `PROCESSORS`, alors le nombre de processeurs est non déterminé, et l'ensemble des modèles comprend les modèles pour tout nombre de processeurs possible. Or le nombre de communication n'est plus alors un critère suffisant pour la préférence, car sinon les modèles où $[\![$`nb-procs`$]\!] = 1$, où aucune communication inter-processeurs n'est nécessaire, seront préférés, alors que le parallélisme y est inexistant, et que donc ces modèles ne sont en fait sûrement pas préférés par le programmeur.

Or, l'absence de directive `PROCESSORS` est plutôt une exception dans la pratique. En outre, également dans la pratique, lorsqu'aucune directive `PROCESSORS` n'est donnée, le nombre de processeurs sera donné par le programmeur à l'exécution. Pour simplifier la présentation, nous considérons donc que le nombre de processeurs est fixé, même en l'absence d'une directive `PROCESSORS`.

Relation avec les directives Or, quelle est la relation entre cette préférence et la préférence sur les directives $\preccurlyeq_{\textsf{dir-ALGO}}$? Il est visible, d'après la définition de $\preccurlyeq_{\text{min-com}}$, que cette dernière relation de préférence ne fait aucune allusion aux directives.

Aurait-on

$$M \preccurlyeq_{\textsf{dir-ALGO}} M' \implies M \preccurlyeq_{\text{min-com}} M' \quad ?$$

Non, car il est possible d'avoir des modèles où la non-satisfaction d'un certain nombre de directives permet en fait d'avoir moins de communications. C'est-à-dire qu'il est possible d'avoir deux modèles M et M' avec $M \preccurlyeq_{\textsf{dir-ALGO}} M'$ mais $M' \preccurlyeq_{\text{min-com}} M$ (et $\neg(M \preccurlyeq_{\text{min-com}} M')$—c'est-à-dire que le nombre de communications de M est strictement supérieur à celui de M'). Cela peut alors signifier deux choses au sujet des directives non satisfaites par M mais satisfaites par M' :

1. soit la différence en nombre de communications est due au fait que M effectue un meilleur placement des données que M', sans que cela dépende vraiment de la satisfaction ou non de ces directives ;
2. soit la différence est vraiment due à la non-satisfaction de ces directives.

On se trouve dans le premier cas s'il est possible de trouver un modèle M'' qui satisfasse ces directives et parvienne à un score en nombre de communications au moins aussi bon que celui de M :

$$\exists M'' \mid [M' \preccurlyeq_{\textsf{dir-ALGO}} M'' \ \wedge\ M'' \preccurlyeq_{\text{min-com}} M']$$

Dans le cas contraire, il est possible de certifier que la différence en nombre de communications est bien dûe à la différence en termes de satisfaction des directives.

La qualité des directives Remarquons que la relation $\preccurlyeq_{\text{min-com}}$ est un pré-ordre total sur les modèles. En particulier, il existe au moins un élément maximal B par rapport à ce pré-ordre (l'optimalité en nombre de communications étant borné par 0) : $\forall M, M \preccurlyeq_{\text{min-com}} B$. Cela permet d'évaluer la qualité d'un ensemble de directives pour la minimisation des communications. Un sous-ensemble U de l'ensemble D des directives est dit *remarquable* si $B \models U$, c'est-à-dire si le meilleur placement possible satisfait chacune des directives de U.

Préférence conditionnelle

Nous avons mentionné à la fin du chapitre précédent la différence entre l'ensemble des modèles, et ce qu'en voyait le compilateur. Or il est clair que le compilateur ne sait pas trouver seul un élément maximal comme B ci-dessus. C'est pourquoi le programmeur fournit des directives. Donc la relation de préférence $\preccurlyeq_{\text{min-com}}$ n'est pas celle considérée par le programmeur—elle ignore les directives !

Donc, pour rendre compte de la sémantique des directives, il faut que la satisfaction de celles-ci soit souhaitée par le programmeur, d'une façon ou d'une autre. Un compromis possible entre ce souhait du programmeur et son souhait de minimisation des directives est de considérer que la satisfaction de la directive est souhaitée par le programmeur *seulement si cela minimise les communications*. Cette préférence peut être modélisée par la relation suivante sur les modèles (pour un programme $ALGO$ et son ensemble d'équations-directives dir-ALGO) : $\preccurlyeq_{\text{opt}}$, définie par :

$$M \preccurlyeq_{\text{opt}} M' \\ \iff \\ M \preccurlyeq_{\textsf{dir-ALGO}} M' \;\wedge\; M \preccurlyeq_{\text{min-com}} M'$$

C'est-à-dire qu'ayant à comparer deux modèles M et M', avec M satisfaisant plus de directives que M', le compilateur doit évaluer M comme «préférable» seulement si M génère également moins de communications que M'.

Conclusion La relation $\preccurlyeq_{\text{opt}}$, destinée à indiquer au compilateur quel choix faire entre deux modèles, suppose que celui-ci sait évaluer le nombre de communications générées par chacun de ces modèles. Or, déterminer statiquement ce nombre n'est pas évident. Tout au plus une estimation de la *différence* (positive ou négative) entre le nombre de communications pour chacun des modèles peut-elle être faite par le compilateur. La relation $\preccurlyeq_{\text{opt}}$ reste alors valide en tant qu'indication de la préférence du programmeur, mais ce que cette relation communique comme information utile au compilateur devient plus flou, s'il devient difficile à celui-ci d'évaluer si un modèle est préféré à un autre d'après cette préférence.

Si, en revanche, nous préférons ne pas faire confiance aux capacités d'estimation du compilateur, alors la meilleure relation de préférence à lui indiquer reste la relation $\preccurlyeq_{\textsf{dir-ALGO}}$.

Cela nous amène à nous demander si la satisfaction des directives ne devrait pas être tout simplement obligatoire. Ce point est traité dans la section ci-dessous.

3.2.2 Pour ou contre des directives contraignantes ?

Nous avons introduit une notion de préférence pour rendre compte de la sémantique des directives et améliorer la coopération programmeur-compilateur. Un problème qui reste à résoudre est le manque de rétro-action ('feedback') du compilateur au programmeur : bien que le programmeur puisse à présent vraiment s'attendre à ce que ses directives soient prises en compte, il ne sait pas exactement lesquelles sont prises en compte.

Une solution naturelle à ce problème peut consister à rendre obligatoire la satisfaction des directives par le compilateur. Cela signifie que dans les cas où le compilateur ne sait pas satisfaire une directive, le processus de compilation serait arrêté. Le compilateur avertirait alors le programmeur du problème, et ce serait à ce dernier de décider comment poursuivre : supprimer la directive en cause, la modifier, ou modifier l'algorithme. En fait, plus le compilateur fournira d'informations sur les problèmes qu'il rencontre, plus le programmeur sera en mesure d'améliorer l'efficacité de son programme. De façon générale, la notion importante ici est celle de *transparence* du compilateur, quant au traitement des directives.

Concernant la définition d'une norme, rendre les directives obligatoires implique de rendre obligatoire la «sémantique effective souhaitée». La question qui se pose est alors celle de la pérennité des programmes HPF. Par exemple, les compilateurs pour le langage C ne sont pas contraints par la norme de tenir compte de la directive «`register`», et cela est appréciable car les derniers avancements en optimisation de code font que l'obligation de respecter cette directive obligerait le compilateur à fournir un code moins performant que ce qu'il est capable de fournir.

Mais dans le cadre d'un langage à directives comme HPF, qui se présente comme une extension de Fortran 95, cela est un faux problème : le langage de directives de HPF a été conçu pour indiquer au compilateur comment minimiser le nombre de communications inter-processeurs, dans le cadre de l'utilisation de machines parallèles à mémoire distribuée. Aussi, si ce sont des machines à mémoire partagée qui sont visées, ou peut-être dans l'avenir des machines où le coût des communications est un facteur négligeable dans la performance de l'application, alors HPF ne sera tout simplement plus le langage de directives pertinent : un autre langage de directives pourra alors être choisi. La mise à jour du code sera alors facilitée par le fait que le programme de base, en Fortran 95, pourra être conservé intact : seules les directives seront changées.

En conclusion, du point de vue du génie logiciel, rien ne s'oppose, et au contraire, à ce que les directives soient toutes obligatoires. Bien sûr une solution «hybride» pour la norme et par suite pour le compilateur, consisterait à permettre au programmeur de choisir le *mode* de compilation souhaité : soit le mode «toutes directives obligatoires» soit le mode «de préférence, prendre en compte les directives».

3.2.3 De l'expression de l'alignement

Nous montrons ici comment la spécification de MINI-HPF fournit une clarification de la notion d'alignement des données.

Le modèle d'alignement de HPF est le suivant (voir aussi le schéma p. 85 ci-dessus) : «The basic concept is that every array (indeed, every object) is created with *some* alignment to an entity, which in turn has *some* distribution onto *some* abstract processors.» [25, p. 20]. Et, plus loin : «Every object is created as if according to some complete set of specification directives ; if the program does not include complete specifications for the mapping of some object, the compiler provides defaults. By default an object is not aligned with any other object ; it is ultimately aligned with itself.» [25, p. 21]

Par contraste, dans notre modèle, l'alignement est une indication du programmeur qui précise la fonction $[\![$`.map`$]\!]$. Cette fonction place tous les processeurs virtuels, et par là tous les objets auxquels ils sont associés. En gros, elle fournit un placement absolu, tandis que l'alignement d'un objet avec un autre implique un placement relatif des objets.

Donc, dans notre modèle, par défaut, aucune directive n'est donnée concernant le placement d'un objet ou son alignement avec un autre objet. Si aucune directive n'indique l'alignement entre deux objets, alors dans certains modèles ces deux objets sont alignés, dans d'autres à moitié alignés, dans d'autres pas du tout. En particulier nous n'avons pas besoin de préciser que par défaut un objet est «aligné avec lui-même».

Ces considérations sont pertinentes dans le cadre d'une discussion sur la préférence liée au directives, parce que le langage HPF ne permet pas d'aligner/placer un tableau ou template plus d'une fois sur un autre tableau ou template. Tout schéma d'alignement entre objets doit donc prendre la forme d'un *arbre*, les fils étant alignés sur les pères. Or, la norme HPF déclare : «if an object `A` is aligned with an object `B`, which in turn is already aligned to an object `C`, this is regarded as an alignment of `A` with `C` directly, with `B` serving only as an intermediary at the time of the specification. We say that `A` is *immediately aligned* with `B` but *ultimately aligned* with `C`. [...] The alignement relationships form a tree with everything ultimately aligned to the object at the root of the tree ; however, the tree is always immediately 'collapsed' so that every object is related directly to the root. » [25, p. 20].

Toutes ces précisions de la part de la norme ont peut-être pour but de rendre absolu le placement, à partir de placements relatifs. Mais elles apportent avec elles des questions. En effet, les directives ne sont pas obligatoires ou seulement «préférées». Alors quel est le statut de l'alignement de `A` avec `C`, par exemple, si l'alignement de `A` avec `B` n'est pas satisfait ? `A` doit-il être aligné avec `C` parce que l'alignement avec `B` est seulement un «intermédiaire» ? Ou la relation de transition prend-elle en compte la notion de satisfaction/non-satisfaction ? En pratique, c'est probablement le cas, car si le compilateur ne satisfait pas l'alignement de `A` avec `B`, c'est probablement parce qu'il ne sait pas le faire, plutôt que parce qu'il a trouvé mieux. Or s'il ne sait pas satisfaire cet alignement, il est probable qu'il ne saura pas aligner `A` avec `C` non plus. Ainsi, si l'alignement est vu comme un arbre «aplati» ('collapsed'), alors on se rend compte que la satisfaction ou non d'une des branches a une incidence sur la satisfaction ou non d'autres branches : les branches ne sont donc pas indépendantes en terme de satisfaction.

Par contraste, le caractère absolu de la fonction $[\![\texttt{.map}]\!]$, et le statut que nous attribuons aux directives n'implique pas de dépendances entre directives, de telle sorte que la satisfaction ou non d'une directive, dans ce cadre, ne remet pas en cause la satisfaction ou non des autres directives.

3.3 Un formalisme simple pour le parallélisme de données

Nous donnons à présent un formalisme simple pour le parallélisme de données, en reprenant les notions définies dans cette thèse. Il permet de mettre en perspective le langage HPF ainsi que le formalisme Pei, étudié ci-dessous. Les principes utilisés sont ceux de la distinction entre sémantique officielle, sémantique effective, et sémantique effective préférée, et la notion de préférence sur les modèles de la sémantique effective préférée. Pour que le modèle soit effectivement simple, et pour l'appliquer plus aisément à l'étude de Pei, nous n'utilisons pas ici la logique de réécriture. Le lecteur constatera sans peine que le formalisme s'inspire en grande partie des idées qui ont permis de définir la sémantique de MINI-HPF.

3.3.1 Exemple : la transformation de Shanks

Pour motiver la suite, nous donnons un exemple d'application où le parallélisme de données est un mode de programmation approprié.

La transformation de Shanks

La transformation de Shanks [49, pp. 205–207] donne un moyen, dans le cadre du calcul numérique, d'améliorer la vitesse de convergence d'une série. Si la série $(S_n)_{n\in\mathbb{N}}$ converge vers une limite S, alors on peut appliquer aux termes de la série la transformation suivante :

$$T(S_n) = \frac{S_{n+2}S_n - (S_{n+1})^2}{S_{n+2} - 2S_{n+1} + S_n} \ .$$

Cette transformation permet d'extrapoler la limite à partir des termes de la série (S_n). En appliquant la même transformation sur la série $(T(S_n))_{n\in\mathbb{N}}$, on obtient une série qui converge encore plus rapidement, et ainsi de suite ....

Par exemple, prenons la série de Leibnitz, définie par

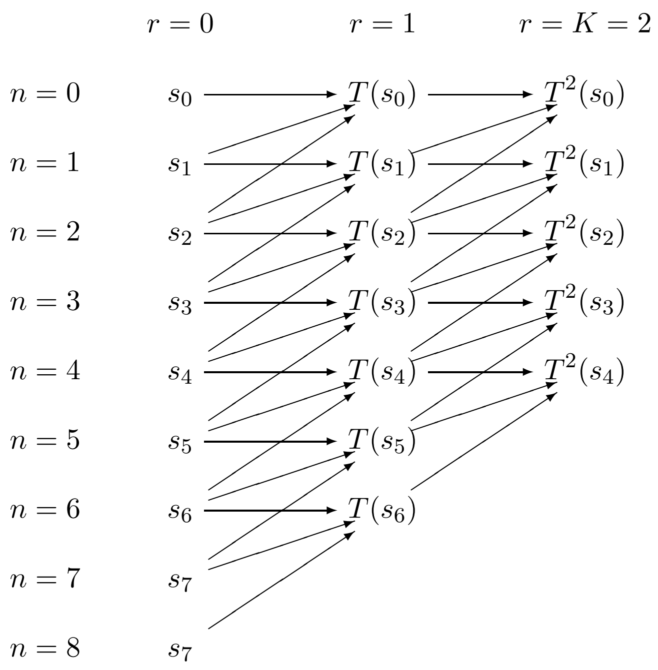
$$A_n = \sum_{i=0}^{n} ((-1)^i \times \frac{4}{2i+1}) \ .$$

Elle converge vers π, mais très lentement. Nous donnons ci-dessous un tableau montrant l'accélération obtenue après deux transformations :

indice	A	$T^2(A)$
0	4.0	3.1421052631579203
1	2.666666666666667	3.1414502164502891
2	3.4666666666666668	3.1416433239963153
3	2.8952380952380956	3.1415712902015125
4	3.3396825396825403	3.1416028416032189
5	2.9760461760461765	
6	3.2837384837384844	
7	3.0170718170718178	
8	3.2523659347188767	

Les 7 premières décimales de π sont 1415927 ; on peut voir l'efficacité de la transformation effectuée sur la série de Leibnitz.

La différence entre le nombre de données dans la série en entrée et la série résultat provient des dépendances correspondant à la transformation de Shanks. Pour l'exemple ci-dessus, les dépendances suivent le schéma suivant :



Le programme en Fortran95

Si nous voulons écrire un programme calculant, à partir des $N+1$ premiers termes d'une série $(A_n)_{n\in\mathbb{N}}$, la série «accélérée» K fois ($K \geq 1$), $(T^K(A_n))_{n\in\mathbb{N}}$, alors nous pourrions nous y prendre comme suit en Fortran 95 :

```
...
INTEGER N,K

...

DOUBLE PRECISION A(0:N), B(0:N-(2*K))
DOUBLE PRECISION T(0:K,0:N)

FORALL (J=0:N)
  T(0,J) = A(J)
END FORALL

DO I=1, K
  FORALL (J=0:N-(2 * I))
    T(I,J) = SHANKS_OP(T(I-1,J), T(I-1,J+1), T(I-1,J+2))
  END FORALL
```

```
END DO

FORALL (J=0:N-(2*K))
  B(J) = T(K,J)
END FORALL

...
```

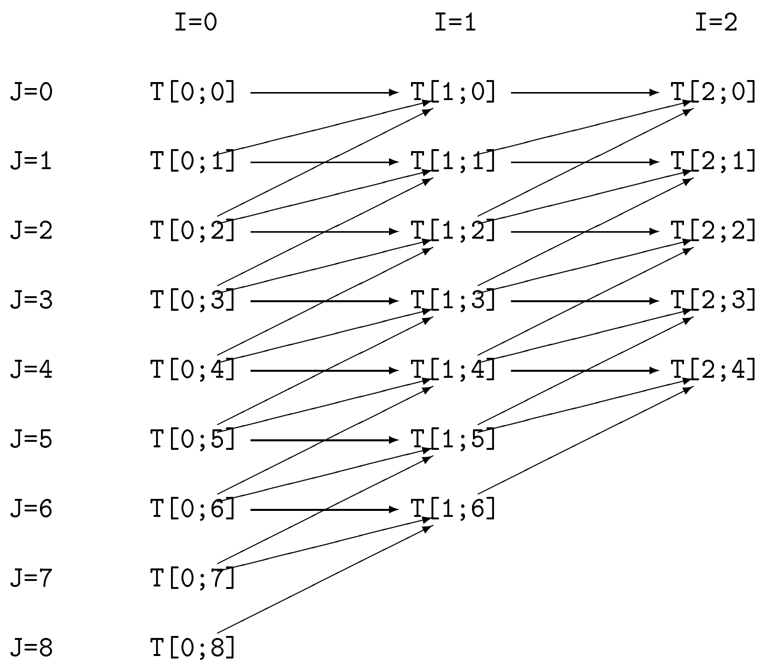
où la fonction `SHANKS_OP` est définie par :

```
PURE FUNCTION SHANKS_OP(V0, V1, V2)
...
SHANKS_OP = ((V2 * V0) - (V1 * V1)) / ((V2 - (2 * V1)) + V0)
END FUNCTION
```

Dans ce programme, les $N+1$ premiers termes de la série $(A_n)_{n\in\mathbb{N}}$ sont stockés dans le tableau en entrée `A`. Les valeurs de la série résultat $(T^K(A_n))_{n\in\mathbb{N}}$ sont stockées dans le tableau `B`.

Le programme opère en calculant les séries intermédiaires $(T^1(A_n))_{n\in\mathbb{N}}$, $(T^2(A_n))_{n\in\mathbb{N}}$, ... Toutes les séries sont placées dans un tableau `T` à deux dimensions : la première dimension correspond au rang de la transformation (de 0 à K), et la deuxième dimension sert à stocker, pour chaque rang de la transformation, les termes de la série. Ainsi, les valeurs de `A` fournissent la série de rang 0 dans `T`, et la série de rang K dans `T` fournit la série recherchée, stockée dans le tableau B.

Les dépendances entre les cases du tableau `T` qui sont exprimées par la boucle principale du programme suivent le schéma de dépendance montré ci-dessus. Ainsi, pour $N = 8$ et $K = 2$, les dépendances sont les suivantes :



Assignation unique, stockage par lignes, ... le programme Fortran 95 `SHANKS` ci-dessus est *à assignation unique*, c'est-à-dire qu'une case de tableau n'est jamais affectée deux fois.

Du point de vue de la sémantique effective de Fortran 95, c'est-à-dire celle indiquée par la norme, cela n'a aucune signification particulière. Du point de vue des sémantiques données par les compilateurs Fortran 95, en revanche, c'est un choix discutable. En effet, en pratique les compilateurs s'inspirent de la sémantique officielle. Si un compilateur applique «telle quelle» la sémantique officielle pour le programme Shanks, cela l'amènera à stocker jusqu'à la fin de l'exécution du programme toutes les valeurs du tableau `T`. Alors que stocker une seule ligne à la fois serait suffisant, au vu des dépendances entre les lignes du tableau. C'est pourquoi certains programmeurs, au vu de ces dépendances, auraient préféré écrire le programme suivant :

```
...
INTEGER N, K

...

DOUBLE PRECISION A(0:N), B(0:N-(2*K))
DOUBLE PRECISION T(0:N)

FORALL (I=0:N)
  T(I) = A(I)
END FORALL

DO I=1, K
  FORALL (J=0:N-(2 * I))
    T(J) = SHANKS_OP(T(J), T(J+1), T(J+2))
  END FORALL
END DO

FORALL (I=0:N-(2*K))
  B(I) = T(I)
END FORALL

...
```

Ainsi, il se seraient assurés d'une utilisation économique de la mémoire. (Notons que l'intruction exécutée en «`FORALL`» est correcte, malgré les dépendances. L'instruction `FORALL` fait partie de Fortran 95, et sur une machine séquentielle par exemple, le compilateur introduira la mémoire temporaire nécessaire pour que l'affectation s'effectue bien «comme en parallèle».)

Quant au stockage par lignes, peu de programmeurs Fortran chevronnés l'auraient choisi, puisque la norme indique qu'il existe un ordre sur les éléments de tableaux, que cet ordre est «par colonnes», et, deux cent pages plus loin, que «a nonpointer array of intrinsic type or derived sequence type occupies a sequence of contiguous storage sequences, one for each array element, in array element order» [61, p. 286].

Cependant, c'est volontairement que nous ne nous sommes pas préoccupés d'optimisation dans le programme `SHANKS` : nous avons préféré préserver la clarté du codage et reporter le problème de la gestion efficace de la mémoire à la phase de compilation, qui peut utiliser par exemple, pour l'optimisation de la mémoire, des techniques comme celles exposées dans [67] ou [15].

Nous pouvons remarquer qu'il semble, au vu de la citation de la norme ci-dessus, que ces optimisations ne sont en fait pas autorisées par la norme Fortran 95. Ce problème correspondrait au fait que cette norme donne dans le cadre de sa sémantique effective des précisions qui deviennent pénalisantes, de par leur statut d'obligations, lorsque les techniques de compilation deviennent capables de «faire mieux». Mais il est probable que sur ce point les implanteurs des compilateurs Fortran s'autoriseront à ne pas respecter la norme.

3.3.2 Le formalisme $\mathcal{D}$

Nous donnons un formalisme, que nous appelons $\mathcal{D}$ (pour «data-parallèle»). Le formalisme $\mathcal{D}$ est utilisé pour formuler des *énoncés*. En outre, $\mathcal{D}$ reprend la distinction entre théories et modèles, de telle sorte qu'un énoncé $\mathcal{D}$ est considéré comme faisant référence à une ensemble de modèles le satisfaisant. Plusieurs syntaxes peuvent être associées au formalisme $\mathcal{D}$, de sorte qu'un programme HPF peut être vu comme une syntaxe particulière associée à un énoncé $\mathcal{D}$. Par exemple, le texte Fortran 95 `SHANKS` ci-dessus peut servir de syntaxe à l'énoncé Shanks exprimé avec le formalisme $\mathcal{D}$.

$$\text{programme} \xleftarrow{\text{syntaxe}} \text{énoncé} \xrightarrow{\text{satisfaction}} \text{modèles}$$

3.3.3 Sémantique officielle de $\mathcal{D}$

La principale différence entre $\mathcal{D}$ et un langage comme Fortran 95 est que dans la sémantique officielle de $\mathcal{D}$, aucune indication n'est donnée concernant la façon dont doit se dérouler l'exécution pour passer des données au résultat : la sémantique officielle énonce seulement une relation entre les données et le résultat. Nous sommes inspirés en cela par le langage Alpha [114], mentionné au chapitre 1, où un programme note un système d'équations récurrentes affines. En particulier, cela signifie que nous n'avons pas dans la sémantique officielle de $\mathcal{D}$ la notion d'*affectation* d'une valeur à une case mémoire.

Champs de valeurs

De façon à nous abstraire de la notion de tableau, et permettre d'appliquer le modèle $\mathcal{D}$ à tout type de données «conteneur de valeurs», comme les listes, nous utilisons la notion de *champ de valeurs*. Cette dénomination est choisie par analogie avec les *champs de données* ('data fields') du formalisme de Björn Lisper (introduit en page 15) ou encore du formalisme PEI (page 14). L'idée est de réserver la notion de «donnée» à l'implantation sur une machine, donc à la sémantique effective.

En effet, chaque champ de valeurs mentionné dans la sémantique officielle de l'énoncé donnera lieu, à l'exécution, à un *champ de données* lui correspondant. C'est-à-dire que chaque valeur est codée par une ou plusieurs données en mémoire. Chacune de ces données est un représentant de cette valeur. Distinguer ainsi entre les valeurs et leurs représentants facilite l'exposition du lien entre la sémantique officielle et la sémantique effective du formalisme $\mathcal{D}$.

Lorsqu'une valeur a plusieurs représentants, c'est sur plusieurs processeurs différents. L'intérêt d'une telle duplication est qu'il est parfois plus rentable, en termes de performance, de calculer plusieurs fois cette valeur que de la calculer une seule fois, puis de la communiquer aux processeurs qui en ont besoin.

Un champ de valeurs est essentiellement un muti-ensemble de valeurs. Pour pouvoir distinguer plusieurs champs, chaque champ a un nom. Nous utilisons pour les nommer des chaînes de caractères parmi l'ensemble de toutes les chaînes de caractères, que nous notons *String*. À l'intérieur des champs de valeurs, il faut pouvoir identifier chaque valeur, aussi utilisons-nous un ensemble d'indices *Index*. Les valeurs quant à elles sont prises dans un ensemble *Val* de valeurs. (Pour ce formalisme simple, nous utilisons un seul «type» de valeurs.)

En $\mathcal{D}$, un énoncé P est construit à partir d'un ensemble *VarName* de noms pour des champs de valeurs : $\mathit{VarName} \subseteq \mathit{String}$. Les valeurs de ces champs sont alors données par une fonction

$$\boxed{\mathrm{val} : \mathit{VarName} \times \mathit{Index} \to \mathit{Val}^*_\perp},$$

où $\mathit{Val}^*_\perp = \mathit{Val} \cup \{\perp, *\}$, où $\perp$ sert à coder l'absence de valeur, et où $\{*\}$ code une valeur «erreur»— dont la signification est expliquée ci-dessous.

Relations entre champs de valeurs

Un énoncé P décrit des relations entre les valeurs des champs, qui doivent être satisfaites par la fonction val. Des *champs en entrée* sont identifiés pour cet énoncé, par un ensemble $\mathit{Input} \subseteq \mathit{VarName}$. De même, des *champs en sortie* sont identifiés par un ensemble $\mathit{Output} \subseteq \mathit{VarName}$.

Nous présentons pour exemple la forme que prend une relation entre trois champs : les autres formes sont similaires. Une telle relation est énoncée par un septuplet $(A, Z, X, g_1, Y, g_2, op)$, où :

- Z, X, Y désignent des noms de champs : $X, Y, Z \in \mathit{VarName}$; en outre, Z ne peut pas être le nom d'un champ en entrée : $Z \notin \mathit{Input}$.
- A est un sous-ensemble fini de *Index* : $A \subseteq \mathit{Index}$;
- op est une fonction de type $\mathit{Val} \times \mathit{Val} \to \mathit{Val}^*$;
- g_1 et g_2 sont deux fonctions de type $\mathit{Index} \to \mathit{Index}_\perp$ (avec $\mathit{Index}_\perp = \mathit{Index} \cup \{\perp\}$), dont le domaine de définition inclut à chaque fois l'ensemble A : $\forall i \in A,\ g_1(i) \neq \perp$ et $g_2(i) \neq \perp$.

Une telle relation est *satisfaite* par une fonction val : $\mathit{VarName} \times \mathit{Index} \to \mathit{Val}^*_\perp$ si cette fonction vérifie la propriété suivante :

$$\forall i \in A, \mathrm{val}(Z, i) = \begin{cases} op^*(\mathrm{val}(X, g_1(i)), \mathrm{val}(Y, g_2(i))), & \text{si } \mathrm{val}(X, g_1(i)) \neq \perp \\ & \text{et } \mathrm{val}(Y, g_2(i)) \neq \perp \\ \perp, & \text{sinon,} \end{cases}$$

où $op^* : Val^* \times Val^* \rightarrow Val^*$ est l'extension stricte de op à l'élément $*$, c'est-à-dire :

$$\forall (v_1, v_2) \in Val^* \times Val^*,$$
$$op^*(v_1, v_2) = \begin{cases} op(v_1, v_2), & \text{si } v_1 \neq * \text{ et } v_2 \neq *, \\ *, & \text{sinon.} \end{cases}$$

Ainsi, un énoncé est donné par un ensemble R de telles relations. C'est-à-dire qu'un énoncé P est un sextuplet $(Val, Index, VarName, Input, Output, R)$.

Modèles La sémantique officielle pour un énoncé $P = (Val, Index, VarName, Input, Output, R)$ est donnée par l'ensemble de ses modèles. Un tel modèle est un sextuplet $(Val, Index, VarName, Input, Output, \mathrm{val})$, où $\mathrm{val} : VarName \times Index \rightarrow Val^*_\perp$ satisfait chacune des relations de R.

Signification des éléments $\perp$ et $*$ Nous décrivons une sémantique officielle, mais sans oublier qu'une sémantique effective souhaitée y sera associée. L'énonciation d'une relation comme celle ci-dessus sera interprétée par la sémantique effective comme un calcul, avec les champs de données X et Y en arguments, et le champ Z en résultat. L'élément $*$ sert à coder la «non-définition» du résultat d'un calcul, comme par exemple, le résultat d'une division d'un réel par zéro.

Par contraste, l'élément $\perp$ code l'absence de résultat induite par l'absence d'arguments pour ce calcul. En d'autres termes, $\perp$ et $*$ codent respectivement «le résultat d'un calcul dont l'un des arguments est non défini» et «les arguments sont définis et le calcul retourne une valeur erreur».

Programmes bien formés

Nous retenons comme énoncés *bien formés* pour le formalisme $\mathcal{D}$, ceux pour lesquels une sémantique effective est définie. Un énoncé bien formé satisfait la condition suivante : si on a dans R deux relations $r = (A, Z, \dots)$ et $r' = (A', Z', \dots)$ avec $Z = Z'$, alors $A \cap A' = \emptyset$. Cela correspond à la condition d'assignation unique, ce qui permet d'énoncer facilement une sémantique effective.

Notons que la définition précédente est suffisante pour la propriété d'assignation unique, mais non absolument nécessaire. En effet, on pourrait admettre $A \cap A' \neq \emptyset$, à condition que les deux relations définissent les mêmes valeurs pour le champ (de nom) Z, sur le domaine $A \cap A'$. Cependant, pour simplifier, nous avons préféré exclure les énoncés présentant de tels cas.

Exemple

Le programme HPF Shanks précédent, dans sa version à assignation unique, peut être formulé par l'énoncé $\mathcal{D}$ bien formé suivant :

- $Val = \mathbb{R}$, $Index = \bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$ (dans la suite nous appelons $\mathcal{Z}$ cet ensemble d'indices).
- $VarName = \{\mathtt{A}, \mathtt{T}, \mathtt{B}\}$, $Input = \{\mathtt{A}\}$, $Output = \{\mathtt{B}\}$.

- Paramètres : entiers N, K.
- $R = \{(\textit{domain-input-T}, \mathtt{T}, \mathtt{A}, \textit{get-input}, id_{\mathbb{R}}),$
  $(\textit{domain-comp}, \mathtt{T}, \mathtt{T}, \textit{get1}, \mathtt{T}, \textit{get2}, \mathtt{T}, \textit{get3}, \textit{shanks-op}),$
  $(\textit{domain-B}, \mathtt{B}, \mathtt{T}, \textit{get-output}, id_{\mathbb{R}})\},$
  avec :

$$\begin{array}{lcl} \textit{domain-input-T} & = & \{(i,j) \in \mathbb{N}^2 \mid i = 0 \ \wedge \ 0 \leq j \leq N\} \\ \textit{domain-comp} & = & \{(i,j) \in \mathbb{N}^2 \mid 1 \leq i \leq K \ \wedge \ 0 \leq j \leq N - (2 \times i)\} \\ \textit{domain-B} & = & \{i \in \mathbb{N} \mid 0 \leq i \leq N - (2 \times K)\} \end{array} ,$$

$$\begin{array}{lcl} \textit{get-input} & : & ((i,j) \in \mathbb{N}^2) \mapsto j \ \ \text{(non définie ailleurs)} \\ \textit{get1} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i-1, j) \ \ \text{(non définie ailleurs)} \\ \textit{get2} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i-1, j+1) \ \ \text{(non définie ailleurs)} \\ \textit{get3} & : & ((i,j) \in \mathbb{N}^2) \mapsto (i-1, j+2) \ \ \text{(non définie ailleurs)} \\ \textit{get-output} & : & (i \in \mathbb{N}) \mapsto (K, i) \ \ \text{(non définie ailleurs)} \end{array} ,$$

$$\begin{array}{lcl} id_{\mathbb{R}} & : & \mathbb{R} \to \mathbb{R}^*, \quad v \mapsto v \\ \textit{shanks-op} & : & \mathbb{R}^3 \to \mathbb{R}^*, \quad (v_0, v_1, v_2) \mapsto ((v2 \times v0) - (v1 \times v1))/((v2 - (2 \times v1)) + v0)) \end{array}$$

Syntaxe Une syntaxe possible pour cet énoncé est donnée par le texte Fortran 95 `SHANKS` donné en début de section, si les signes «`=`» sont compris comme signifiant une égalité plutôt qu'une affectation. C'est parce que ce programme Fortran est à assignation unique que cette interprétation du texte est possible.

3.3.4 Sémantique effective de $\mathcal{D}$

Comme le formalisme $\mathcal{D}$ se veut un modèle du parallélisme de données, nous ne distinguons pas la sémantique effective de la sémantique effective souhaitée. La sémantique effective que nous décrivons à présent est donc à comparer à la sémantique effective *souhaitée* de MINI-HPF, plutôt qu'à sa sémantique effective (celle de Fortran 95).

La sémantique effective pour $\mathcal{D}$ est une extension de la sémantique officielle. C'est-à-dire que celle-ci est préservée, et que des informations supplémentaires sont ajoutées. Cette extension est définie ci-dessous.

Dans la suite, nous nous plaçons dans le contexte d'un énoncé $P = (\textit{Val}, \textit{Index}, \textit{VarName}, \textit{Input}, \textit{Output}, R)$ et nous définissons la sémantique effective qui lui est associée.

Champs de données

La sémantique effective commence par énoncer, comme pour MINI-HPF, qu'une machine parallèle à mémoire distribuée est utilisée. Soit *Proc*, l'ensemble des processeurs de la machine.

La notion de placement des représentants d'une valeur sur les mémoires des processeurs est codée par la fonction suivante :

$$\boxed{\text{loc} : (\textit{VarName} \cup \textit{TemplateName}) \times \textit{Index} \to \mathcal{P}(\textit{Proc})} ,$$

où $\mathcal{P}(Proc)$ est l'ensemble des parties de l'ensemble $Proc$, $\mathrm{loc}(X,i)$ est l'ensemble des processeurs qui auront à comporter en mémoire un représentant pour $\mathrm{val}(X,i)$, et où $\mathit{TemplateName} \subseteq \mathit{String}$ est un ensemble de *noms de templates*, et où l'on suppose $\mathit{VarName} \cap \mathit{TemplateName} = \emptyset$. La notion de template est expliquée dans le paragraphe suivant.

L'implication suivante doit être satisfaite par la fonction loc :

$$\boxed{\forall (X,i) \in \mathit{VarName} \times \mathit{Index},\ \mathrm{val}(X,i) \neq \perp \implies \mathrm{loc}(X,i) \neq \emptyset} \ .$$

C'est-à-dire qu'une valeur d'un champ X, indicée i dans ce champ, doit avoir au moins un représentant dans le champ de données correspondant à X, si elle est définie.

Templates Nous fournissons la notion de *template* de HPF, car celle-ci est commode dans un certain nombre de situations pour énoncer des alignements. La fonction d'un template est de fournir un ensemble de *processeurs virtuels*, au sens de MINI-HPF.

Affectations

Dans la sémantique effective chacune des relations entre champs de valeurs induit des affectations et des calculs. Ceux-ci sont exprimés par des transitions d'états. Nous reprenons ici le codage utilisé dans [27]. Un état est codé par un ensemble de fonctions

$$\boxed{\mathrm{val}_p : \mathit{VarName} \times \mathit{Index} \rightarrow \mathit{Val}^*_\perp} \ ,$$

où les indices p sont pris dans l'ensemble des processeurs de la machine, $Proc$. Intuitivement, une fonction val_p code l'état de la mémoire sur un processeur.

État initial

Un *état initial* est défini comme contenant des valeurs uniquement dans les champs en entrée :

$$\forall p \in Proc, \forall X \in (\mathit{VarName} \setminus \mathit{Input}), \forall i \in \mathit{Index},\ \mathrm{val}_p(X,i) = \perp \ .$$

(Cela ne signifie pas que les champs en entrée ont forcément des valeurs définies pour tout indice. Dans un programme réaliste, les valeurs seront définies, pour une variable en entrée, sur un sous-ensemble fini d'indices.)

Transitions

À partir d'un état initial, des transitions s'appliquent, définies à partir des relations de R. Nous exposons le cas d'une relation $(A, Z, X, g_1, Y, g_2, op)$. Une telle relation induit les transitions suivantes : pour tout $i \in A$, pour tous $p, p', p'' \in Proc$:

$$\frac{\begin{array}{l} \mathrm{val}_p(Z,i) = \perp, p \in \mathrm{loc}(Z,i) \\ \mathrm{val}_{p'}(X, g_1(i)) \neq \perp, p' \in \mathrm{loc}(X, g_1(i)) \\ \mathrm{val}_{p''}(Y, g_2(i)) \neq \perp, p'' \in \mathrm{loc}(Y, g_2(i)) \\ p \in \mathrm{loc}(X, g_1(i)) \implies p = p' \\ p \in \mathrm{loc}(Y, g_2(i)) \implies p = p'' \end{array}}{\mathrm{val}_p \longrightarrow \mathrm{val}_p[(Z,i) \leftarrow op^*(\mathrm{val}_{p'}(X, g_1(i)), \mathrm{val}_{p''}(Y, g_2(i)))]} \tag{3.1}$$

où $\text{val}_p[(X,i) \leftarrow v]$ est l'état val_p où la valeur $\text{val}_p(X,i)$ a été remplacée par la valeur v.

Calculs : Chaque transition signifie que la valeur $op^*(\text{val}_{p'}(X, g_1(i)), \text{val}_{p''}(Y, g_2(i)))$ est *calculée* par le processeur p, sur lequel elle est ensuite stockée selon la 'Owner-Computes Rule'. Remarquons que lorsque $\text{loc}(Z,i)$ contient plus d'un processeur, cela implique que le calcul sera fait sur chacun des processeurs. Nous rappelons que cette duplication des calculs sert, dans certains cas, à éviter des communications.

Communications : Le calcul ci-dessus requiert les valeurs $\text{val}_{p'}(X, g_1(i))$ et $\text{val}_{p''}(Y, g_2(i))$, qui sont stockées, respectivement, sur les processeurs p' et p''. Lorsque $p \neq p'$, alors une *communication* inter-processeurs est requise. La condition de transition «$p \in \text{loc}(X, g_1(i)) \implies p = p'$» exprime que si le processeur p doit accueillir un représentant pour la valeur à l'indice $g_1(i)$ du champ de valeurs X, alors le calcul s'effectue en utilisant ce représentant et pas un autre : cela évite une communication. De même si $p \in \text{loc}(Y, g_2(i))$.

Ainsi, le nombre de communications finalement générées pour le calcul de $\text{val}(Z,i)$ est au plus : $\text{card}(\text{loc}(Z,i) \setminus \text{loc}(X, g_1(i))) + \text{card}(\text{loc}(Z,i) \setminus \text{loc}(Y, g_2(i)))$.

État final Un *état final* est un état tel qu'aucune transition ne peut plus s'opérer. On a alors la relation suivante entre la sémantique effective et la sémantique officielle : l'état final fournit les représentants pour les champs de valeurs d'un modèle de la sémantique officielle.

Cela s'exprime plus précisément comme suit.
Soit val : $\mathit{VarName} \times \mathit{Index} \to \mathit{Val}^*_\perp$ la fonction définie à partir de l'état final de la façon suivante :

$$\forall (X,i) \in \mathit{VarName} \times \mathit{Index}, \forall p \in \mathit{Proc} \ : \ p \in \text{loc}(X,i) \implies \text{val}(X,i) = \text{val}_p(X,i) \ .$$

Cette fonction val est un modèle de la sémantique officielle.

Une preuve de cette correspondance ferait appel à la notion de construction du plus petit point fixe, dans le cadre de la théorie des domaines de Scott [44]. En effet, les transitions ci-dessus correspondent en fait à une version distribuée et asynchrone d'une telle construction. Ce type de construction est connu, voir par exemple [32, pp. 256–258] : nous ne détaillons donc pas cette preuve, mais en indiquons seulement le principe. L'énoncé est vu comme une équation $\text{val} = \psi(\text{val})$, où la fonction ψ est définie à partir de l'ensemble des relations de l'énoncé, ce qui est possible grâce à la propriété d'«assignation unique» de l'énoncé. Le domaine des fonctions val considéré est l'ensemble des fonctions qui comportent pour les champs en entrée les mêmes valeurs que l'état initial choisi val_0, lorsque ces valeurs sont définies. Les transitions ci-dessus correspondent alors à une version désynchronisée des applications successives de la fonction ψ à partir de l'état initial : $\psi(\text{val}_0), \psi^2(\text{val}_0), \psi^3(\text{val}_0), \ldots$. Ainsi, lorsque plus aucune transition n'est possible, cela signifie qu'un point fixe a été atteint. Remarquons qu'il est possible qu'aucun point fixe ne soit jamais atteint—dans le cas où les domaines d'application des affectations (les ensembles «A») sont infinis. Dans ce cas, le domaine considéré pour ψ n'est pas un ordre partiel complet, et rien ne garantit donc que l'on arrive à un point fixe—et dans le cas où des dépendances font «s'étendre» continuellement des champs de données, aucun point fixe ne sera effectivement jamais atteint. Cependant, nous nous intéressons ici à la programmation «réelle», où un domaine fini est toujours précisé pour les applications des affectations.

Directives de placement

Des directives de placement peuvent accompagner l'énoncé. En $\mathcal{D}$, une directive est formulée par une relation entre le placement de deux templates ou champs de valeurs. Elle a la forme suivante : (A, X, Y, g), avec :

- $X, Y \in (\mathit{VarName} \cup \mathit{TemplateName})$;
- $g : \mathit{Index} \rightarrow \mathit{Index}_\perp$.

Une telle relation est *satisfaite* par la fonction loc si :

$$\forall i \in A,\ \mathrm{loc}(X, i) = \mathrm{loc}(Y, g(i)) \ .$$

Intuitivement, la satisfaction d'une telle relation implique que le nombre de communications nécessaires aux calculs diminue.

Préférence

Le statut des directives ajoutées à un énoncé $\mathcal{D}$ est donné par une notion de préférence sur des modèles.

Nous supposons pour simplifier que l'ensemble de processeurs *Proc* est fixé. L'ensemble des modèles considérés au niveau de la sémantique effective pour un énoncé $P = (\mathit{Val}, \mathit{Index}, \mathit{VarName}, \mathit{Input}, \mathit{Output}, R)$, est l'ensemble des septuplets $(\mathit{Val}, \mathit{Index}, \mathit{VarName}, \mathit{Input}, \mathit{Output}, \mathrm{val}, \mathrm{loc})$ tels que $(\mathit{Val}, \mathit{Index}, \mathit{VarName}, \mathit{Input}, \mathit{Output}, \mathrm{val})$ est un modèle de la sémantique officielle.

Si D est l'ensemble des directives rajoutées à l'énoncé, alors cet ensemble de modèles est muni de la relation de préférence $\preccurlyeq_D$ exposée en section 3.1.7. C'est-à-dire que si le compilateur peut construire un modèle dont la fonction loc satisfait une directive de plus, cela est préféré par le programmeur, l'idéal étant donc la satisfaction par loc de chacune des directives.

Exemple 1

L'énoncé $\mathcal{D}$ pour la transformation de Shanks ci-dessus peut être complété, par exemple, par les directives suivantes :

- $\mathit{TemplateName} = \emptyset$.
- Directives :
  $\{(\mathit{domain\text{-}input\text{-}T}, \mathtt{T}, \mathtt{A}, \mathit{get\text{-}input}),$
  $(\mathit{domain\text{-}comp}, \mathtt{T}, \mathtt{T}, \mathit{previous}),$
  $(\mathit{domain\text{-}B}, \mathtt{B}, \mathtt{T}, \mathit{get\text{-}output})\}$,

avec *previous* : $((i, j) \in \mathbb{N}^2) \mapsto (i-1, j)$ (non définie ailleurs).

La première directive d'alignement implique qu'aucune communication ne sera nécessaire pour placer les données de `A` dans le champ `T`. La dernière directive implique qu'aucune communication ne sera nécessaire pour extraire de `T` les données en sortie et les placer dans `B`. La deuxième directive implique que la dépendance entre val($\mathtt{T}, (i,j)$) et val($\mathtt{T}, (i-1,j)$) ne générera pas de communications.

Une syntaxe «à la HPF» pour ces trois directives serait :

```
!HPF$  ALIGN T(0,I)    WITH A(I)
!HPF$  ALIGN T(1:K,I)  WITH T(0:(K-1),I)
!HPF$  ALIGN B(I)      WITH T(K,I)
```

En fait, ces directives ne sont pas valides en HPF car le tableau `T` est aligné deux fois. En outre, il n'est pas possible en HPF d'aligner seulement une sous-partie d'un tableau. Quant à la version :

```
!HPF$  ALIGN A(I)      WITH T(0, I)
!HPF$  ALIGN T(I,J)    WITH T(I-1,J)
!HPF$  ALIGN B(I)      WITH T(K,I)
```

elle semble valide d'après la norme, mais le compilateur pghpf la refuse car elle contient un alignement circulaire. C'est la formulation de l'exemple suivant qui devra donc être utilisée dans un programme HPF.

Exemple 2

Nous pouvons exprimer le même alignement en passant par un template :

- *TemplateName* = $\{\mathtt{LINE}\}$.
- $\{(\textit{domain-A}, \mathtt{A}, \mathtt{LINE}, \textit{id}),$
  $(\textit{domain-T}, \mathtt{T}, \mathtt{LINE}, \textit{project}),$
  $(\textit{domain-B}, \mathtt{A}, \mathtt{LINE}, \textit{id})\}$,

avec :
$\textit{domain-A} = \textit{domain-B} = \{i \in \mathbb{N} \mid 0 \leq i \leq N\}$,
$\textit{domain-T} = \{(i,j) \in \mathbb{N}^2 \mid 0 \leq i \leq K \ \wedge\ 0 \leq j \leq N - (2 \times K)\}$,
$\textit{project} : (i,j) \mapsto j$ (non définie ailleurs).

Les indices du template `LINE` qui sont concernés sont ceux de l'ensemble *domain-A*. Notons que si val(A, i) est défini sur tout l'ensemble *domain-A*, cela implique, de par la définition de l'alignement et la règle de cohérence qui requiert un représentant pour chaque valeur, que loc($\mathtt{LINE}, i$) $\neq \emptyset$ pour tous les indices de *domain-A*.

En HPF, cet alignement serait exprimé de la façon suivante :

```
!HPF$  TEMPLATE LINE(0:N)
!HPF$  ALIGN A(I)   WITH LINE(I)
!HPF$  ALIGN B(I)   WITH LINE(I)
!HPF$  ALIGN T(*,J) WITH LINE(J)
```

Remarquons que l'alignement des colonnes de `T` entre elles est quelque chose que n'aurait pas eu besoin de faire le programmeur qui aurait optimisé son programme «à la main», comme montré plus haut, en faisant de `T` un tableau à une seule dimension et en réutilisant la mémoire dans l'algorithme. Dans la version à assignation unique, on a donc ici un exemple où le programmeur fait attention à ne pas embrouiller le code avec ses optimisations, mais où le compilateur doit ensuite retrouver une version efficace, à moins que le programmeur ne l'y aide par une directive. C'est-à-dire que l'optimisation du programmeur est déplacée depuis une réécriture du code vers l'écriture d'une directive ajoutée à un code non optimisé.

3.4 Étude du langage PEI

Nous traduisons le langage PEI dans le formalisme $\mathcal{D}$ afin d'éclaircir la sémantique de PEI et d'en expliquer les limitations.

3.4.1 Comparaison entre PEI et $\mathcal{D}$

La sémantique d'un programme PEI peut être exprimée dans le formalisme $\mathcal{D}$. C'est-à-dire que l'on retrouve dans PEI (exprimées différemment) les notions de champ de valeurs, de champ de données, et d'alignement des données.

Les deux principales différences sont :
- PEI inclut une syntaxe, et
- en PEI, des directives d'alignement sont *automatiquement* générées pour chaque relation entre champs de valeurs.

Les autres différences sont deux points sur lesquels PEI est plus contraint :
- l'ensemble des processeurs est un sous-ensemble de $\mathcal{Z}$, l'ensemble des n-uplets d'entiers. Cet ensemble joue le rôle d'un ensemble de processeurs virtuels (au sens des processeurs virtuels de MINI-HPF) plutôt que de vrais processeurs.
- Les directives ont un statut d'obligation pour le compilateur.

3.4.2 Sémantique de PEI

La sémantique de PEI, dans la littérature, se partage entre sa sémantique *dénotationnelle* (dont la dernière version est donnée dans le rapport de recherche [108]), et sa sémantique *opérationnelle*, définie dans [27]. Nous situons ces deux sémantiques de PEI par rapport aux sémantiques officielle et effective de $\mathcal{D}$.

Sémantique dénotationnelle de PEI

Pour la sémantique dénotationnelle, nous en utilisons la présentation qui est connue dans la littérature, comme par exemple dans [109].

Dans la sémantique dénotationnelle de PEI, la notion fondamentale est le *champ de données mis en forme* ('shaped data field'). Cette notion est codée par un objet mathématique, conformément au principe de la sémantique dénotationnelle. Cet objet est un couple de fonctions (v, σ), où v est une fonction partielle avec pour domaine $\mathcal{Z}$, et pour codomaine un ensemble de valeurs ; et où σ est une fonction partielle $\mathcal{Z} \rightharpoonup \mathcal{Z}$.

La fonction v correspond à la notion de champ de valeurs de $\mathcal{D}$—cela sera détaillé ci-dessous. La fonction v se rattache donc à la sémantique officielle de $\mathcal{D}$. La fonction σ correspond quant à elle au placement d'un champ de données, et donc à ce qui est exprimé dans la sémantique effective de $\mathcal{D}$. Ainsi, l'objet de base de la sémantique dénotationnelle de Pei, le champ de données mis en forme, comporte les deux aspects : les valeurs, et le placement des représentants des valeurs.

Conformément au principe de la sémantique dénotationnelle, la sémantique dénotationnelle de Pei est *compositionnelle*. Par contraste, la sémantique de $\mathcal{D}$ n'est pas compositionnelle. Cela amène deux différences entre Pei et $\mathcal{D}$:

- comme l'objet de base est le champ de données mis en forme, on a en Pei des expressions qui ne sont pas des champs nommés, et dont la sémantique est définie à partir des champs nommés apparaissant comme sous-expressions. Par contraste, en $\mathcal{D}$, seuls les noms de l'ensemble *VarName* correspondent à des champs de valeurs.
- Alors que l'objet mathématique de base de Pei comporte l'aspect des valeurs et l'aspect du placement, un énoncé $\mathcal{D}$ a deux aspects séparés : l'expression des valeurs des champs, avec la fonction val, et l'expression du placement des données, avec une fonction loc.

Sémantique opérationnelle de Pei

La sémantique opérationnelle de Pei peut être vue comme identique à la sémantique effective de $\mathcal{D}$, après que nous aurons explicité ci-dessous la traduction du langage Pei vers le formalisme $\mathcal{D}$.

Remarquons qu'alors que la notion d'alignement concerne la sémantique effective, et ne trouve donc sa justification que dans la sémantique opérationnelle, qui introduit la notion de communication inter-processeurs, cette notion est énoncée en Pei au niveau de la sémantique dénotationnelle, avec la fonction σ, indépendamment de considérations opérationnelles. Historiquement, cela a rendu difficile la justification du modèle Pei à partir de sa seule sémantique dénotationnelle.

3.4.3 Le programme Shanks en Pei

Nous donnons ci-dessus un équivalent en langage Pei du programme Shanks :

```
SHANKS : A -> B
{
domain-input-T :: T = get-input :: A
domain-comp    :: T = shanks-op (T <| get1) (T <| get2) (T <| get3)
                  B = get-output :: T
}

domain-input-T(i,j) = (i = 0, 0 <= j <= N) . (i, j)
get-input(i)        = (0 <= i <= N) . (0, i)
```

```
domain-comp-T(i,j)  = (1 <= i <= K, 0 <= j <= N - (2 * K)) . (i  ,j)
get1(i,j)           = (1 <= i <= K, 0 <= j <= N - (2 * K)) . (i-1,j)
get2(i,j)           = (1 <= i <= K, 0 <= j <= N - (2 * K)) . (i-1,j+1)
get3(i,j)           = (1 <= i <= K, 0 <= j <= N - (2 * K)) . (i-1,j+2)

get-output(i,j)     = (i = K, 0 <= j <= N - (2*K)) . i

shanks-op(v0, v1, v2) = (v2*v0 - v1*v1) / (v2 - 2*v1 + v0))
```

La sémantique de PEI est telle qu'un ensemble de directives est généré à partir de ce programme : ces directives sont explicitées ci-dessous.

3.4.4 L'alignement des données en PEI

Nous expliquons ici la notion d'alignement de PEI, en la traduisant en $\mathcal{D}$: cette traduction nous permet d'apporter un éclairage sur la sémantique de PEI, et d'autre part d'exposer les limites de ce langage.

Le placement en PEI

Nous indiquons comment la notion de champ de données mis en forme se traduit en $\mathcal{D}$.

L'ensemble des indices, *Index*, est $\mathcal{Z}$. L'ensemble des valeurs correspond à l'ensemble *Val*. *VarName* est l'ensemble des variables du programmes.

Pour une variable $X \in \mathit{VarName}$ de sémantique (v, σ) en PEI, la fonction v correspond en $\mathcal{D}$ à la fonction $\widehat{\mathrm{val}}(X) : \mathcal{Z} \rightarrow \mathit{Val}^*_\perp$, où $\widehat{\mathrm{val}} : \mathit{VarName} \rightarrow (\mathit{Index} \rightarrow \mathit{Val}^*_\perp)$ est la forme curryfiée de val.

La non-définition de $v(i)$, en PEI, pour un indice $i \in \mathcal{Z}$, correspond en $\mathcal{D}$ à : $\widehat{\mathrm{val}}(X)(i) = \perp$. Il n'y a pas l'équivalent de l'élément $*$ en PEI, car un programme PEI est supposé ne contenir que des calculs corrects.

Quant à la fonction σ, son rôle dans PEI est d'indiquer la location des valeurs de la façon suivante : pour un champ de valeurs v pour le nom $X \in \mathit{VarName}$, la valeur $v(i)$ est localisée aux emplacements $\sigma^{-1}(i)$. La traduction en $\mathcal{D}$ est donc la suivante : $\mathit{Proc} = \mathcal{Z}$, et σ correspond à la fonction $\widehat{\mathrm{loc}}(X) : \mathcal{Z} \rightarrow (\mathcal{P}(\mathcal{Z}))_\perp$, définie par : $\widehat{loc}(X)(i) = \sigma^{-1}(i)$ (où $\widehat{loc} : (\mathit{VarName} \cup \mathit{TemplateName}) \rightarrow (\mathit{Index} \rightarrow \mathcal{P}(\mathit{Proc}))$ est la forme curryfiée de loc). La sémantique de PEI requiert que le domaine de définition de la fonction partielle v soit inclus dans l'image de la fonction partielle σ, ce qui correspond à notre contrainte (page 123).

En résumé, la sémantique d'un champ de données mis en forme de nom $X \in \mathit{VarName}$ correspond en $\mathcal{D}$ au couple $(\widehat{val}(X), \widehat{\mathrm{loc}}(X))$.

Observons qu'en raison de la définition du placement par l'inverse d'une fonction, un champ de valeurs comportant n valeurs (i.e., défini en n points de $\mathcal{Z}$) verra en PEI ses valeurs placées sur au moins n points de $\mathcal{Z}$. Cela interdit en PEI d'aligner un tableau avec lui-même comme nous avons pu le faire en $\mathcal{D}$ pour le champ `T` du programme Shanks, dans l'exemple de la page 125. Ceci est à notre avis une des limitations principales du modèle PEI.

Ce point explique également pourquoi l'ensemble de locations $\mathcal{Z}$ est un ensemble de processeurs virtuels : si une matrice 1000×1000 est représentée en PEI par un champ mis en forme, ses valeurs sont placées sur un minimum de 10^9 locations, qui ne peuvent être vues comme correspondant aux processeurs d'une vraie machine. Le placement «final» devra donc supposer un placement de ces 10^9 processeurs virtuels sur le nombre limité de processeurs d'une architecture parallèle.

Sémantique dénotationnelle et placement

Dans la sémantique du langage PEI, compositionnelle, toute sous-expression bien formée a pour sémantique un champ de données mis en forme. Pour rendre compte de cela dans le cadre du formalisme $\mathcal{D}$, nous considérons l'ensemble $\mathit{Expr} \subseteq \mathit{String}$ des expressions du programme qui notent un champ de données mis en forme, et les fonctions : $\mathrm{val}_e : \mathit{Expr} \times \mathit{Index} \to \mathit{Val}_\perp^*$ ('e' pour «expression»), et $\mathrm{loc}_e : \mathit{Expr} \times \mathit{Index} \to \mathit{Proc}_\perp$. Avec, bien sur :

- $\forall X \in \mathit{VarName}, \forall i \in \mathit{Index},\ \mathrm{val}_e(X,i) = \mathrm{val}(X,i)$, et
- $\forall X \in \mathit{VarName}, \forall i \in \mathit{Index},\ \mathrm{loc}_e(X,i) = \mathrm{loc}(X,i)$.

Ainsi, nous allons pouvoir «suivre» la sémantique compositionnelle de PEI et en rendre compte en $\mathcal{D}$. En $\mathcal{D}$, il n'y a pas de sémantique compositionnelle parce qu'il n'y a pas de syntaxe. Les fonctions val_e et loc_e servent ici d'intermédiaires pour définir les fonctions val et loc : cela reflète le fait que dans la sémantique de PEI, bien que ce soit la sémantique associée aux identificateurs de champs (les éléments de *VarName*) dont on veut rendre compte, la sémantique compositionnelle amène à considérer les autres expressions comme notant également des champs de données mis en forme, et à faire transiter l'information de placement des données par ces expressions, comme nous allons le voir.

3.4.5 Directives induites

En PEI, si une expression $E \in \mathit{Expr}$ désigne un champ de données mis en forme, et $\mathtt{g} \in \mathit{String}$ une fonction $[\![\mathtt{g}]\!]$ de type $\mathcal{Z} \rightharpoonup \mathcal{Z}$ (comme par exemple `domain-input-T` ci-dessus), alors «$\mathtt{g} :: E$» et «E `<|` `g`» sont toutes deux des expressions notant un champ de données mis en forme.

Réindiçage L'utilisation du symbole «`::`» indique un «réindiçage» du champ mis en forme E par la fonction $[\![\mathtt{g}]\!]$. C'est-à-dire que les indices d'accès aux valeurs sont changés, mais pas la localisation de ces valeurs (ou des données qui les représentent plus exactement, dans le vocabulaire $\mathcal{D}$). En PEI, cette opération est appelée *changement de base*, parce que c'est comme si le repère dans lequel sont indicées les valeurs changeait, sans que les représentants des valeurs sur les locations soient affectés.

Le réindiçage est codé de la façon suivante en PEI. Si la sémantique de E est le couple (v, σ), alors la sémantique de $\mathtt{g} :: E$ est le champ placé $(v \circ [\![\mathtt{g}]\!]^{-1}, [\![\mathtt{g}]\!] \circ \sigma)$.

En $\mathcal{D}$, cela s'exprime de la façon suivante, $[\![\mathsf{g}]\!]$ étant considérée de type $\mathcal{Z} \to \mathcal{Z}_\perp$. Pour tout $i \in \mathit{Index}$:

- $\mathrm{val}_e(\mathsf{g} :: E, [\![\mathsf{g}]\!](i)) = \mathrm{val}_e(E, i)$ si $[\![\mathsf{g}]\!](i) \neq \perp$, $\perp$ sinon.
- $\mathrm{loc}_e(\mathsf{g} :: E, [\![\mathsf{g}]\!](i)) = \mathrm{loc}_e(E, i)$ si $[\![\mathsf{g}]\!](i) \neq \perp$, $\emptyset$ sinon.

Transfert de données L'utilisation du symbole «`<|`» indique en un certain sens le contraire du réindiçage : c'est-à-dire que la fonction d'accès aux valeurs est changée, mais également la localisation des représentants de ces valeurs. C'est pour cela que nous appelons cela un «transfert de données». En PEI, cette opération est appelée *opération géométrique*, parce que la localisation des données est changée : comme les locations sont prises dans $\mathcal{Z}$, si on considère les éléments de $\mathcal{Z}$ plongés dans les espaces euclidiens discrets $\mathbb{Z}^n$, alors c'est effectivement la géométrie du champ de données qui change.

Cette opération est codée comme suit en PEI : si la sémantique de `X` est le couple (v, σ), alors la sémantique de `X <| f` est le champ placé $(v \circ [\![\mathsf{g}]\!], \sigma)$.

La traduction $\mathcal{D}$ est la suivante, en considérant de nouveau $[\![\mathsf{g}]\!]$ comme étant de type $\mathcal{Z} \to \mathcal{Z}_\perp$. Pour tout indice $i \in \mathit{Index}$:

- $\mathrm{val}_e(E \mathrel{<|} \mathsf{g}, i) = \mathrm{val}_e(E, [\![\mathsf{g}]\!](i))$ si $[\![\mathsf{g}]\!](i) \neq \perp$, $\perp$ sinon.
- $\mathrm{loc}_e(E \mathrel{<|} \mathsf{g}, i) = \mathrm{loc}_e(E, i) \quad ,$

La contrainte de PEI selon laquelle le domaine de définition de v doit être compris dans l'image de σ correspond ici à la contrainte $\mathcal{D}$ selon laquelle :

$$\mathrm{val}_e(E \mathrel{<|} \mathsf{g}, i) \neq \perp \implies \mathrm{loc}_e(E \mathrel{<|} \mathsf{g}, i) \neq \emptyset \ .$$

Nous avons donc :

$$\mathrm{val}_e(E \mathrel{<|} \mathsf{g}, i) \neq \perp \implies \mathrm{loc}_e(E, i) \neq \emptyset \ .$$

Cela signifie que la fonction de localisation associée à E, $\widehat{\mathrm{loc}_e}(E)$, doit «prévoir» une localisation pour les champs mis en forme dérivés de E. En effet, on peut avoir $\mathrm{val}_e(E, i) = \perp$ mais $\mathrm{val}_e(E \mathrel{<|} \mathsf{g}, i) \neq \perp$, ce qui oblige $\mathrm{loc}_e(E, i) \neq \emptyset$ bien que E n'ait pas de représentant à placer sur cet ensemble non vide de locations.

Cela indique la nature «artificielle» du couple (v, σ) dans le cadre de la sémantique compositionnelle. Dans le cas de l'opération de transfert de données, la fonction σ de E doit contenir le placement du champ E, et être prête à prendre en compte le placement d'autres champs : ceux qui se situent au niveau des expressions contenant E. Cet artifice sert donc à s'abstraire de la contrainte compositionnelle. L'inconvénient est bien sûr que cette sémantique n'est pas très intuitive, car le fait que la dénotation d'une expression ne dépend pas seulement de la dénotation de ses sous-expressions nous fait sortir de la logique compositionnelle.

Dans la dernière version de PEI [108], la présentation de la sémantique a été modifiée : un champ de données y est à présent une fonction qui prend en argument une forme (une fonction σ), de sorte qu'une fonction v n'est pas attachée à «un» σ. La formulation générale ressort encore de la technique compositionnelle, mais la définition du placement (ou de σ) quant à elle n'est plus purement compositionnelle.

Application de fonction Pour l'application de fonction, nous traitons le cas d'une opération binaire, les autres cas étant similaires. En PEI, une expression $(E\ op\ E') \in \mathit{Expr}$ représente l'application de l'opération $[\![op]\!]$ sur les valeurs des champs notés par E et E', de la façon suivante. Si E et E' notent respectivement des couples (v, σ) et (v', σ'), alors cela implique $\sigma = \sigma'$. Puis, $(E\ op\ E')$ note le couple $(v >>, \sigma)$, où $v >>= i \mapsto [\![op]\!](v(i), v'(i))$.

La traduction en $\mathcal{D}$ est la suivante. Pour tout $i \in \mathit{Index}$:

- $(E\ op\ E') \in \mathit{Expr} \implies \mathrm{val}_e(E\ op\ E', i) = [\![op]\!]^*(\mathrm{val}_e(E, i), \mathrm{val}_e(E', i))$,
  si $\mathrm{val}_e(E, i) \neq \perp$ et $\mathrm{val}_e(E', i) \neq \perp$; $\perp$, sinon.
- $(E\ op\ E') \in \mathit{Expr} \implies \mathrm{loc}_e(E, i) = \mathrm{loc}_e(E', i) = \mathrm{loc}_e((E\ E'\ op), i)$.

On voit comment la sémantique dénotationnelle est utilisée pour introduire des contraintes d'alignement : ici, l'alignement des trois champs impliqués est codé par le fait qu'ils sont placés exactement de la même manière.

Équations Une équation PEI E = E' relie deux expressions de champs mis en forme, de la façon suivante : si E est un champ (v, σ) et E' un champ placé (v', σ'), alors l'équation définit : $v = v'$ et $\sigma = \sigma'$.

La traduction en $\mathcal{D}$ est la suivante. Pour tout $i \in \mathit{Index}$:

- $\mathrm{val}_e(E, i) = \mathrm{val}_e(E', i)$.
- $\mathrm{loc}_e(E, i) = \mathrm{loc}_e(E', i)$.

Ainsi, le placement pour les expressions de gauche et de droite est la même : cela se répercute sur les sous-expressions, dont les placements se trouvent ainsi reliés. Or un champ nommé qui n'est pas un champ en entrée ou en sortie apparaît typiquement dans plusieurs équations : cela propage le placement d'équation en équation. Ainsi la sémantique de PEI construit un placement «absolu» des données à partir de placements «relatifs».

Exemple 1

Nous montrons dans cet exemple et les suivants la façon dont PEI met en relation les valeurs et leur placement.

Nous examinons dans ce premier exemple ce qu'induit la première équation du programme Shanks ci-dessus, à savoir :

```
domain-input-T :: T = get-input :: A
```

Pour les valeurs Nous analysons d'abord la signification de cette équation pour les valeurs respectives des champs T et A. La définition du réindiçage amène :

- $val_e(\texttt{domain-input-T :: T}, (i,j)) = val_e(\texttt{T}, (i,j))$ si $i = 0 \wedge 0 \leq j \leq N$,
  $val_e(\texttt{domain-input-T :: T}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$.
- $val_e(\texttt{get-input :: A}, (i,j)) = val_e(\texttt{A}, j))$ si $i = 0 \wedge 0 \leq j \leq N$,
  $val_e(\texttt{get-input :: A}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$.

Ce qui implique :

- $\forall (i,j) \in \mathbb{Z}^2 \ : \ i = 0 \wedge 0 \leq j \leq N \implies val_e(\texttt{T}, (i,j)) = val_e(\texttt{domain-input-T :: T}, (i,j))$.
- $\forall (i,j) \in \mathbb{Z}^2 \ : \ i = 0 \wedge 0 \leq j \leq N \implies val_e(\texttt{A}, j)) = val_e(\texttt{get-input :: A}, (i,j))$.

Or, l'équation quant à elle implique :

$$\forall z \in \mathcal{Z},\ val_e(\texttt{domain-input-T :: T}, z) = val_e(\texttt{get-input :: A}, z).$$

D'où : $\forall (i,j) \in \mathbb{Z}^2 \ : \ i = 0 \wedge 0 \leq j \leq N \implies val_e(\texttt{T}, (i,j)) = val_e(\texttt{A}, j)$.

Si l'on s'abstrait à présent de la fonction intermédiaire val_e qui a servi à établir cette égalité, nous avons : $\forall (i,j) \in \mathbb{Z}^2 \ : \ i = 0 \wedge 0 \leq j \leq N \implies val(\texttt{T}, (i,j)) = val(\texttt{A}, j)$, qui est ce que nous voulions exprimer en écrivant cette équation, à savoir l'égalité entre les valeurs du tableau T sur sa ligne 0 et les valeurs du tableau en entrée A. Pour la sémantique opérationnelle, cette équation impliquera une copie des valeurs de A dans la première ligne de T.

Pour le placement Pour le placement, le raisonnement est le même (voir la définition du réindiçage ci-dessus), et l'on a : $\forall (i,j) \in \mathbb{Z}^2 \ : \ i = 0 \wedge 0 \leq j \leq N \implies loc_e(\texttt{T}, (i,j)) = loc_e(\texttt{A}, j)$, d'où l'on tire, de même : $\forall (i,j) \in \mathbb{Z}^2 \ : \ i = 0 \wedge 0 \leq j \leq N \implies loc(\texttt{T}, (i,j)) = loc(\texttt{A}, j)$.

Nous observons donc ici le principe de Pei : les équations induisent automatiquement un placement respectif des champs nommés. Nous pouvons constater à présent que la première équation du programme Shanks en Pei a été écrite de telle sorte que la première ligne du tableau T soit alignée avec le tableau A, le but étant que dans la sémantique opérationnelle aucune communication ne soit générée par l'affectation correspondant à l'équation.

Exemple 2

Nous procédons à une analyse similaire pour la deuxième équation du programme Pei Shanks ci-dessus, soit l'équation :

```
domain-comp :: T = shanks-op (T <| get1) (T <| get2) (T <| get3)
```

Pour les valeurs Nous établissons la relation donnée par cette équation entre les valeurs du champ T.

- $val_e(\texttt{domain-comp-T :: T}, (i,j)) = val_e(\texttt{T}, (i,j))$ si $1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K)$,
  $val_e(\texttt{domain-comp-T :: T}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$.
- $val_e(\texttt{T <| get1}, (i,j)) = val_e(\texttt{T}, (i-1,j))$ si $1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K)$,
  $val_e(\texttt{T <| get1}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$
  (suivant la définition de l'opération `<|` donnée plus haut).
- $val_e(\texttt{T <| get2}, (i,j)) = val_e(\texttt{T}, (i-1,j+1))$ si $1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K)$,
  $val_e(\texttt{T <| get2}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$.
- $val_e(\texttt{T <| get3}, (i,j)) = val_e(\texttt{T}, (i-1,j+2))$ si $1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K)$,
  $val_e(\texttt{T <| get3}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$.
- $val_e(\texttt{shanks-op (T <| get1) (T <| get2) (T <| get3)}, (i,j)) =$
  $[\![\texttt{shanks-op}]\!]^*(\mathrm{val}_e(\texttt{T <| get1}, (i-1,j)), \mathrm{val}_e(\texttt{T <| get2}, (i-1,j+1)),$
  $\mathrm{val}_e(\texttt{T <| get3}, (i-1,j+2)))$ si $1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K)$,
  $\perp$ pour tous les autres indices de $\mathcal{Z}$.

L'équation implique quant à elle :
$\forall z \in \mathcal{Z},\ val_e(\texttt{domain-comp-T :: T}, (i,j)) =$
$val_e(\texttt{shanks-op (T <| get1) (T <| get2) (T <| get3)}, (i,j))$.

Il vient donc : $\forall (i,j) \in \mathbb{Z}^2\ :\ 1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K) \implies$
$\mathrm{val}_e(\texttt{T}, (i,j)) = [\![\texttt{shanks-op}]\!]^*(\mathrm{val}_e(\texttt{T}, (i-1,j)), \mathrm{val}_e(\texttt{T}, (i-1,j+1)), \mathrm{val}_e(\texttt{T}, (i-1,j+2)))$.

C'est-à-dire, pour la fonction val : $\forall (i,j) \in \mathbb{Z}^2\ :\ 1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K) \implies$
$\mathrm{val}(\texttt{T}, (i,j)) = [\![\texttt{shanks-op}]\!]^*(\mathrm{val}(\texttt{T}, (i-1,j)), \mathrm{val}(\texttt{T}, (i-1,j+1)), \mathrm{val}(\texttt{T}, (i-1,j+2)))$.

Ce qui correspond bien à ce que nous voulions exprimer. Pour la sémantique opérationnelle, le calcul des valeurs pour cette équation récursive se fera suivant les dépendances, donc les représentants des valeurs de `T` en $(i-1,j)$, $(i-1,j+1)$ et $(i-1,j+2)$ devront être produits avant que les représentants de la valeur en (i,j) puissent l'être.

Pour le placement Nous étudions à présent ce qu'énonce cette équation quant au placement du seul champ nommé concerné, `T`.

- **Pour le membre gauche :**

$loc_e(\texttt{domain-comp-T :: T}, (i,j)) = loc_e(\texttt{T}, (i,j))$ si $1 \leq i \leq N \ \wedge \ 0 \leq j \leq N - (2 \times K)$,
$loc_e(\texttt{domain-comp-T :: T}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$.

- **Pour le membre droit :**

Pour tout indice $z \in \mathit{Index}$:

- $loc_e(\texttt{T <| get1}, z) = loc_e(\texttt{T}, z)$
  (suivant la définition de l'opération `<|` donnée plus haut).
- $loc_e(\texttt{T <| get2}, z) = loc_e(\texttt{T}, z)$.
- $loc_e(\texttt{T <| get3}, z) = loc_e(\texttt{T}, z)$.
- $loc_e(\texttt{shanks-op (T <| get1) (T <| get2) (T <| get3)}, z)$
  $= \mathrm{loc}_e(\texttt{T <| get1}, z) = \mathrm{loc}_e(\texttt{T <| get2}, z) = \mathrm{loc}_e(\texttt{T <| get3}, z)$.

- **Pour l'équation :**

Par l'égalité entre membres gauche et droit, il vient : $val_e(\texttt{T}, z) = val_e(\texttt{T}, z)$, pour tout $z \in Index$. D'où, pour tout $z \in Index$: $val(\texttt{T}, z) = val(\texttt{T}, z)$. C'est-à-dire qu'aucune information d'alignement n'est apportée par cette équation.

Cela est en fait conforme à ce que nous avions fait remarqué plus haut, à savoir qu'il n'est pas possible en PEI de toute façon d'induire des alignements d'un champ avec lui-même.

Exemple 3

Nous pouvons cependant observer à quoi correspondrait le fait d'aligner un champ avec lui-même.

Si nous voulions formuler l'équivalent de l'alignement énoncé avec $\mathcal{D}$ par la directive

$$(domain\text{-}comp, \texttt{T}, \texttt{T}, previous)$$

(voir la page 125), nous écririons en PEI :

```
domain-comp :: T = shanks-op (get1b :: T) (T <| get2) (T <| get3) ,
```

avec `get1b (i,j) = (0 <= i < K, 0 <= j <= N - (2 * K)) . (i+1,j)`,
signifiant que nous souhaitons aligner, pour `T`, les représentants pour une valeur en (i, j), avec les représentants pour une valeur en $(i + 1, j)$.

Cette équation a la même signification, quant aux valeurs (val), que l'équation précédente. Pour le placement, nous aurions obtenu, entre autres :

- $loc_e(\texttt{get1 :: T}, (i + 1, j)) = loc_e(\texttt{T}, (i, j))$ si $0 \leq i < K \ \wedge\ 0 \leq j \leq N - (2 \times K)$,
  $val_e(\texttt{get1b :: T}, z) = \perp$ pour tous les autres indices de $\mathcal{Z}$
  (suivant la définition de l'opération `<|` donnée plus haut).
- $\forall z \in Index\ :\ loc_e(\texttt{T <| get2}, z) = loc_e(\texttt{T}, z)$.
- Par l'application de l'opération `shanks-op` :
  $\forall z \in Index\ :\ loc_e(\texttt{get1b :: T}, z) = loc_e(\texttt{T <| get2}, z)$.

D'où il vient :

$$\forall (i, j) \in \mathbb{Z}^2\ :\ 1 \leq i \leq N \ \wedge\ 0 \leq j \leq N - (2 \times K) \implies loc_e(\texttt{T}, (i - 1, j)) = loc_e(\texttt{T}, (i, j)),$$

qui est bien ce que nous souhaitions.

Bien sûr, de par la définition de la fonction de localisation comme l'inverse d'une fonction (σ^{-1}), cette équation PEI n'est pas correcte. Mais la contrainte de PEI suivant laquelle $\widehat{\mathrm{loc}}(X)$ est l'inverse d'un fonction a pour but de faire en sorte que les directives induites permettent de construire un placement. C'est-à-dire que la garantie de constructivité amène certaines contraintes. Cela est nécessaire car en PEI les directives sont obligatoires, et pour que cela ait un sens il faut que le compilateur soit dans tous les cas en mesure de satisfaire ces directives.

Exemple 4

Le premier et le troisième exemple a montré que le symbole «`::`» introduit un alignement, et le deuxième exemple que le symbole «`<|`» correspond à l'absence d'alignement. Il n'y a pas d'alternative entre ces symboles, et dans le cadre de la sémantique de PEI, cela introduit des contraintes subtiles, qui apparaissent rapidement lorsque le nombre de champs nommés mis en relations augmente.

Par exemple, supposons que nous voulions donner en sortie du programme Shanks en PEI, au lieu de la série transformée, la différence entre la série d'origine et la série transformée. Appelons `D` ce champ «différence».

Il nous suffirait a priori d'indiquer $Output = \{\mathtt{D}\}$ au lieu de $Output = \{\mathtt{B}\}$, et de rajouter l'équation suivante :

```
D = B - A
```

La sémantique du placement indique , du fait de l'application de l'opération «`-`» : $\forall z \in \mathcal{Z}\ :\ \mathrm{loc}_e(\mathtt{A}, z) = \mathrm{loc}_e(\mathtt{B}, z)$, d'où nous avons : $\forall z \in \mathcal{Z}\ :\ \mathrm{loc}(\mathtt{A}, z) = \mathrm{loc}(\mathtt{B}, z)$.

Or, cette dernière équation implique, de par les première et troisième équations du programme Shanks, que les représentants pour les indices $\{(i,j) \in \mathbb{Z}^2 \mid i = 0 \wedge 0 \leq j \leq N-(2\times K)\}$ du champ `T` sont alignées avec les représentants pour les indices $\{(i,j) \in \mathbb{Z}^2 \mid i = K\ \wedge\ 0 \leq j \leq N-(2\times K)\}$ du même champ, ce qui est, comme nous l'avons vu, impossible d'après la sémantique de PEI.

Donc cette équation PEI n'est pas valide au regard des équations précédentes. C'est ce type de subtilité que permet de déceler le *typechecker* de PEI [105].

En conclusion, il n'est pas possible d'aligner les tableaux `A` et `B` avec le tableau `D` sans modifier les trois premières équations, car ces équations empêchent d'aligner `A` et `B`.

Exemple 5

Il est possible d'aligner `A` et `B` en modifiant les équations précédentes. Une solution possible est de modifier le placement de `A` et `B` par rapport à `T` :

```
SHANKS : A -> D
{
domain-input-T :: T = A <| get-inputb
domain-comp    :: T = shanks-op (T <| get1) (T <| get2) (T <| get3)
                  B = T <| get-outputb
```

```
                    D = B - A
}
```

avec `get-inputb(i,j) = (i = 0, 0 <= j <= N) . j`
et `get-outputb(i) = (0 <= i <= N - (2*K)) . (K, i).`

Dans ce programme, le tableau `B` n'est plus aligné avec la dernière ligne de `T`. Pour le placement, l'équation `B = T <| get-outputb` implique :

– loc_e(`T <| get-outputb`) $= \mathrm{loc}_e$(`T`) ;
– loc_e(`B`) $= \mathrm{loc}_e$(`T <| get-outputb`),

D'où : loc(`B`) = loc(`T`). Donc, comme loc(`B`) est l'inverse d'un fonction, que les indices des valeurs de `B` sont dans $\mathbb{Z}$ et ceux de `T` dans $\mathbb{Z}^2$, les valeurs de ces deux tableaux ne sont pas alignées.

De même pour le rapport de `A` à `T` : loc(`A`) = loc(`T`).

Il en découle : loc(`A`) = loc(`B`). En outre, de par l'équation `D = A - B`, on a également : loc(`D`) = loc(`A`) = loc(`B`), soit l'alignement que nous souhaitions obtenir.

Remarquons que `A`, `B` et `T` se retrouvent alignés sans qu'il ait été fait usage du symbole «`::`» pour les aligner ainsi. La raison en est qu'en Pei, si deux champs X et Y utilisent, pour indicer une valeur, un même indice $z \in Index$, et que la chaîne des relations, dans le programme, menant de X à Y, ne comporte pas de symbole «`::`» comportant une autre fonction que l'identité restreinte à un domaine, alors le placement associé à cet indice est la même pour ces deux champs.

3.4.6 Statut de la localisation en Pei

Les fonctions σ associées à chaque champ mis en forme d'un programme Pei *définissent* ce qu'est Pei, en tant que modèle du parallélisme de données. Si la sémantique des opérations est interprétée, comme nous l'avons fait, comme induisant des directives de placement de façon automatique, alors ces alignements doivent être considérés comme obligatoires, puisque participant de la définition de ce qu'est Pei.

Donc ces directives de placement ne sont pas des directives au sens de HPF (ou de $\mathcal{D}$) : un compilateur implantant Pei a obligation de satisfaire toutes les directives induites par le programme. En ce sens, le traducteur **`pei2hpf`** écrit par Stéphane Genaud [26] ne satisfait pas la sémantique de Pei, puisque les directives de Pei, une fois traduites en HPF, deviennent non obligatoires.

Or, le fait de ne pas satisfaire une des directives de Pei pose un problème théorique similaire à celui que nous avons relevé pour HPF. En effet, dans la cadre de la sémantique compositionnelle de Pei, les placements respectifs des champs mis en forme sont déduits les uns des autres, via la fonction σ. Donc si l'une des inductions provenant de la sémantique compositionnelle n'est pas satisfaite, le schéma général de placement est brisé, et les directives qui dépendaient de cette induction n'ont en fait plus lieu d'être satisfaites.

Une solution simple à ce problème consiste à calculer le placement général «comme si toutes les directives étaient obligatoires», et ensuite seulement à s'accorder à ne pas satisfaire certains des placements. C'est que qui est fait implicitement avec le traducteur **`pei2hpf`** : il utilise l'outil d'inférence des placements respectifs de Pei, le «typechecker», pour déterminer le schéma général de placement, traduit ce schéma en directives HPF, et ensuite HPF peut ne pas satisfaire certaines des directives.

3.4.7 Conclusion

Nous avons vu au travers des exemples comment PEI construit un placement absolu pour les champs de données, à partir de placements relatifs des champs entre eux. Ce placement absolu est entièrement contraint, contrairement à $\mathcal{D}$ où la fonction loc est contrainte par morceaux par des directives.

L'avantage du schéma de placement de PEI est que le placement respectif des champs étant entièrement déterminé, le compilateur construit facilement ce placement. L'inconvénient est qu'il n'est pas aisé de déterminer, à la vue d'un programme PEI, si deux champs sont alignés entre eux, comme pour les champs `A` et `B` ci-dessus dans les exemples ci-dessus.

Par contraste, en $\mathcal{D}$, si aucune directive n'est énoncée quant à l'alignement de deux champs, alors des valeurs de ces champs peuvent être soit non alignées, soit alignées tout de même. Le programmeur n'a donc pas d'inférence à faire à partir des directives qu'il a énoncées. En revanche, le compilateur doit placer lui-même de façon appropriée les champs au sujet desquels le programmeur n'a rien dit.

Nous avons interprété la notion de placement de PEI comme étant construite à partir de «directives automatiques». En particulier nous avons vu comment le symbole «`::`» signifie un alignement. Or il serait possible de reprendre cette idée dans le cadre de directives non obligatoires. Par exemple, en $\mathcal{D}$, cela pourrait se modéliser par un argument supplémentaire *optionnel* (pour éviter les problèmes de PEI exhibés plus haut) à côté des champs arguments. Une directive type serait alors «`noComm`», qui requiérerait que le placement respectif du champ argument concerné et du champ résultat soient tels que la relation en question n'induise aucune communication au niveau de la sémantique effective. Par exemple, la relation pour la première équation de Shanks pourrait alors être simplement :

$$(\textit{domain-input-T}, \mathtt{T}, \mathtt{A}, \textit{get-input}, \mathtt{noComm}, \textit{id}_{\mathbb{R}}) \ ,$$

qui induirait automatiquement sur la fonction val la contrainte

$$(\textit{domain-input-T}, \mathtt{T}, \mathtt{A}, \textit{get-input})$$

que nous avions dû écrire explicitement dans notre modèle.

Dans une syntaxe «à la HPF», cela pourrait s'écrire :

```
      FORALL (J=0:N)
        T(0,J) = A(J)
!HPF$       [ NOCOMM ]
      END FORALL
```

Bien sûr, il ne serait pas question de rajouter «`noComm`» systématiquement car cela n'aiderait pas le compilateur qui serait incapable de satisfaire toutes ces directives. Il s'agit donc de réserver ce type de directive aux cas «importants».

Conclusion

Dans le cadre de la programmation parallèle, le compilateur peut avoir du mal à produire un code parallèle efficace. Actuellement, cela motive l'utilisation de la bibliothèque de communications MPI, où c'est le programmeur lui-même qui prend en charge la gestion de la répartition des tâches sur les processeurs et des communications de données entre processeurs. Dans le domaine des langages parallèles de plus haut niveau, l'approche actuelle consiste, pour espérer concurrencer la performance de MPI, à permettre au programmeur d'ajouter dans son programme des directives au compilateur, afin d'aider celui-ci à produire un code parallèle efficace.

Nous nous sommes intéressés au cas de la programmation à parallélisme de données. Le langage data-parallèle industriel actuel, HPF, comporte un langage de directives satisfaisant, mais le statut de ces directives n'est pas clair. La norme HPF les définit comme de simples commentaires, tout en indiquant leur signification pour le cas où elles seraient en fait davantage que des commentaires (ce qu'elles sont). La conséquence est que la coopération programmeur-compilateur concernant ces directives n'est pas optimale : pour déterminer l'effet de ses directives, le programmeur doit utiliser des outils de déboguage. En outre, l'efficacité du programme qui est obtenue après ce «déboguage» ne bénéficie quant à elle d'aucune garantie de portabilité.

Dans cette thèse, nous avons montré comment la notion de directive au compilateur pouvait s'articuler dans le cadre d'une norme. Afin d'obtenir un modèle mathématiquement clair, nous nous sommes appuyés sur la logique de réécriture. Nous avons énoncé dans ce cadre les notions de sémantique officielle, sémantique effective, et sémantique effective souhaitée par le programmeur.

Cela pose des liens entre les méthodes existantes pour descrire formellement la sémantique d'un langage, et le domaine de la compilation. En effet, les descriptions formelles de sémantique tendent à ignorer la phase de compilation ; tandis que la phase de compilation est généralement décrite par les techniques qu'elle met en jeu, plutôt que par son rapport à la sémantique du langage qu'elle compile. Or, le rapport entre la sémantique du langage et la compilation est important lorsque la notion de *performance* rentre en compte, ce qui est le cas pour une grande partie des utilisations d'un langage de programmation, et plus encore lorsqu'il s'agit d'exploiter une machine parallèle.

Le cadre algébrique apporté par la logique de réécriture et ses ensembles de modèles nous ont permis de modéliser la notion de *préférence* par le programmeur de la satisfaction d'une directive par le compilateur, montrant ainsi que la notion de directive pouvait être intégrée de façon précise dans une norme, sans que la satisfaction des directives par le compilateur soit rendue obligatoire.

Nous avons exposé avec ce modèle la sémantique des directives de HPF, ce qui nous a permis de discuter avec précision du modèle HPF de placement de données.

Puis nous avons proposé un formalisme simple pour le parallélisme de données, qui intègre la notion de directive conformément à la sémantique que nous en avons donnée. La relation de ce formalisme au modèle HPF est décrite.

Puis ce formalisme a été utilisé pour expliquer la sémantique du langage académique PEI. En particulier, nous avons montré que la sémantique de PEI induit des contraintes sur ce qu'il est possible d'écrire, mais de telle sorte que l'effort demandé au programmeur pour écrire un programme correct rend ensuite la tâche de placement facile pour le compilateur. Nous avons également pu retenir de PEI l'idée de directive induite par symbole-clé, idée qui n'apparaît pas dans HPF et qui peut simplifier l'expression de directives pour un code.

La notion de directive, mise au premier plan dans les modèles OpenMP et HPF, présente l'avantage méthodologique de séparer les préocupations de définition des calculs à effectuer, des préocupations quant au mode d'exécution correspondant. Du point de vue du génie logiciel, cela ne peut avoir que de bonnes répercutions sur la pérennité des programmes écrits dans ce style. En effet, lorsque les directives perdent leur pertinence à cause d'une évolution des architectures des ordinateurs, des environnements d'exécution, ou des techniques de compilation, il est possible de mettre à jour la partie «directives» du programme sans avoir à modifier la partie «fonctionnelle» du code. Si l'approche HPF est suivie, où les directives visent un type particulier d'architecture, alors la mise à jour consistera à utiliser un langage de directives successeur, et donc à remplacer les directives HPF par les nouvelles directives, sans avoir à toucher au code Fortran 95.

Du point de vue du génie logiciel, c'est donc une philosophie proche de celle de la *réutilisabilité* de logiciel, avec pour caractéristique que si l'effort initial de programmation est plus important (il faut écrire des directives), à moyen et long terme cet investissement devient rentable. Elle est pertinente dans le cadre de la complexification croissante des environnements d'exécution—grappes d'ordinateurs, grille de calculs, systèmes embarqués, ...— où la durée de vie d'un code dédié à un environnement particulier est appelée à diminuer toujours davantage. Il est donc important de séparer considérations de «ce qui est calculé» et «comment cela est calculé»—distinction traditionnellement exprimée par l'opposition entre sémantique dénotationnelle et sémantique opérationnelle.

Au laboratoire LSIIT-ICPS, un projet est en cours qui vise à adapter des programmes pour l'environnement d'exécution de la grille computationnelle. L'objectif est à terme d'automatiser des techniques d'adaptation, l'adaptation se faisant actuellement à la main. Or, les contraintes de la grille impliquent que des informations sur le programme soient connues : cela implique que les outils d'adaptation prennent en entrée de telles informations qu'ils ne sauraient inférer statiquement à partir du code du programme, et que le programmmeur devra indiquer. Cela amène la notion de directive, ici aux outils d'adaptation.

Un travail théorique reste à mener concernant le rapport du langage des directives à la sémantique officielle, lorsque celui-ci est fort. Par exemple, dans HPF, les directives s'ancrent sur les indices des tableaux, et l'alignement est exprimé par des fonctions sur ces indices entiers, utilisant par là le fait que les indices se trouvent être des n-uplets d'éléments d'une structure d'anneau standard ($\mathbb{Z}$), ce qui permet une expression simple de relation entre indices de tableaux. C'est suivant ce principe que nous avons considéré dans le formalisme $\mathcal{D}$ des fonctions d'indice à indice pour exprimer les directives, mais il reste à étudier comment sont exprimées ces fonctions dans le cas d'arbres ou de listes, ou d'autres structures de données.

Annexe A

La spécification de la sémantique de MINI-HPF

A.1 La sémantique officielle de MINI-HPF

A.1.1 L'effet sur les valeurs

La définition du la sémantique officielle est complétée par le module suivant, qui définit l'effet des affectations :

```
module* MINI-HPF-OFFICIAL-SEM {
  using (MINI-HPF-OFFICIAL-SEMO)

  var tr       : Trace    var algo : Algo
  vars x y z w : String   var i    : Int

  ceq (M wTrace: (tr ; < z :=s  x + y >) wAlgo: algo) .val w
   =   ((M wTrace: tr wAlgo: algo) .val x)
     + ((M wTrace: tr wAlgo: algo) .val y)
     if w == z .

  ceq (M wTrace: (tr ; < z :=as x + y >) wAlgo: algo) .val w i
    =   ((M wTrace: tr wAlgo: algo) .val x i)
      + ((M wTrace: tr wAlgo: algo) .val y)
      if w == z and 1 <= i and i <= (length w) .

  ceq (M wTrace: (tr ; < z :=aa x + y >) wAlgo: algo) .val w i
    =   ((M wTrace: tr wAlgo: algo) .val x i)
      + ((M wTrace: tr wAlgo: algo) .val y i)
      if w == z and 1 <= i and i <= (length w) .
```

```
** negative cases:

  ceq (M wTrace: (tr ; < z :=s  x + y >) wAlgo: algo) .val w
   =  (M wTrace: tr wAlgo: algo) .val w
      if not (w == z) .

  ceq (M wTrace: (tr ; < z :=as x + y >) wAlgo: algo) .val w i
    = (M wTrace: tr wAlgo: algo) .val w i
      if not (w == z) .

  ceq (M wTrace: (tr ; < z :=aa  x + y >) wAlgo: algo) .val w i
    = (M wTrace: tr wAlgo: algo) .val w i
      if not (w == z) .

  var ar-assgnt : ArrayAssignt
  eq  (M wTrace: (tr ; < ar-assgnt >)    wAlgo: algo) .val w
   =  (M wTrace: tr wAlgo: algo) .val w .

  var s-assgnt : SAssignt
  eq  (M wTrace: (tr ; < s-assgnt >)     wAlgo: algo) .val w i
    = (M wTrace: tr wAlgo: algo) .val w i .
}
```

A.2 La sémantique effective de MINI-HPF

A.2.1 L'effet sur les valeurs

La définition du la sémantique effective est complétée par le module suivant, qui définit l'effet de l'exécution d'un algorithme. Les deux premières équations établissent l'égalité entre les valeurs des variables d'entrée sur les états initiaux de la machine officielle et les états initiaux de la machine effective. Les deux équations suivantes établissent l'égalité entre les variables en sortie sur les états finaux de la machine officielle et les états finaux de la machine effective.

```
module MINI-HPF-ACTUAL-SEM {
  protecting (MINI-HPF-OFFICIAL-SEM)
  using      (MINI-HPF-ACTUAL-SEM0)

  ** input and output values identical for official and actual semantics:

  var actTr : ActTrace    var tr : Trace   var algo : Algo
  vars x y z w : String   var i : Int
```

```
  ceq (AM wTrace: noActInst wAlgo: algo  ) .val x
    = (M  wTrace: noInst    wAlgo: algo  ) .val x
          if input-var x .

  ceq (AM wTrace: noActInst wAlgo: algo  ) .val x i
    = (M  wTrace: noInst    wAlgo: algo  ) .val x i
          if input-var x .

  ceq (AM wTrace: actTr     wAlgo: noAlgo) .val x
    = (M  wTrace: tr        wAlgo: noAlgo) .val x
    if (AM wTrace: noActInst wAlgo: algo) ==> (AM wTrace: actTr wAlgo: noAlgo)
   and (M  wTrace: noInst    wAlgo: algo) ==> (M  wTrace: tr    wAlgo: noAlgo)
   and  output-var x .

  ceq (AM wTrace: actTr     wAlgo: noAlgo) .val x i
    = (M  wTrace: tr        wAlgo: noAlgo) .val x i
    if (AM wTrace: noActInst wAlgo: algo) ==> (AM wTrace: actTr wAlgo: noAlgo)
   and (M  wTrace: noInst    wAlgo: algo) ==> (M  wTrace: tr    wAlgo: noAlgo)
   and  output-var x .
}
```

A.3 La sémantique effective souhaitée de MINI-HPF

A.3.1 L'effet sur les valeurs

Les quatres premières équations sont similaires à celles qui constituent le module `MINI-HPF-ACTUAL-SEM` (p. 144). Elles établissent, pour les deux premières, l'égalité des valeurs pour les variables en entrée pour les états initiaux sur la machine effective et sur la machine effective souhaitée. Les deux équations suivantes établissent l'égalité des valeurs pour les variables en sortie pour les états finaux sur les deux machines.

Les deux dernières équations du modules sont celles qui sont commentées dans la section 3.1.6.

```
module MINI-HPF-WISHED-FOR-ACTUAL-SEM1 {
  protecting (MINI-HPF-ACTUAL-SEM)
  using      (MINI-HPF-WISHED-FOR-ACTUAL-SEM0)

  ** satisfaction of the effective semantics:

  vars x y z w : String   var i : Int
  var actTr  : ActTrace   var algo : Algo
  var wactTr : WActTrace
```

```
  ceq (WAM wTrace: noWActInst wAlgo: algo) .val x =
      ( AM wTrace: noActInst  wAlgo: algo) .val x
      if input-var x .

  ceq (WAM wTrace: noWActInst wAlgo: algo) .val x i =
      ( AM wTrace: noActInst  wAlgo: algo) .val x i
      if input-var x .

  ceq (WAM wTrace: wactTr     wAlgo: noAlgo) .val x
    = (AM  wTrace: actTr      wAlgo: noAlgo) .val x
        if     (WAM wTrace: noWActInst wAlgo: algo)
           ==> (WAM wTrace: wactTr wAlgo: noAlgo)
       and     ( AM wTrace: noActInst  wAlgo: algo)
           ==> ( AM wTrace: actTr  wAlgo: noAlgo)
       and  output-var x .

  ceq (WAM wTrace: wactTr     wAlgo: noAlgo) .val x i
    = (AM  wTrace: actTr      wAlgo: noAlgo) .val x i
        if     (WAM wTrace: noWActInst wAlgo: algo)
           ==> (WAM wTrace: wactTr wAlgo: noAlgo)
       and     ( AM wTrace: noActInst  wAlgo: algo)
           ==> ( AM wTrace: actTr  wAlgo: noAlgo)
       and  output-var x .


  ** the effect of array assignments on values :

  ceq (WAM wTrace: wactTr ; <<< z :=as x + y >>> wAlgo: noAlgo) .val w i
    =   ((WAM wTrace: wactTr wAlgo: noAlgo) .val x i)
      + ((WAM wTrace: wactTr wAlgo: noAlgo) .val y)
             if w == z and 1 <= i and i <= (length w) .

  ceq (WAM wTrace: wactTr ; <<< z :=aa  x + y >>> wAlgo: noAlgo) .val w i
    =   ((WAM wTrace: wactTr wAlgo: noAlgo) .val x i)
      + ((WAM wTrace: wactTr wAlgo: noAlgo) .val y i)
             if w == z and 1 <= i and i <= (length w) .
}
```

Bibliographie

[1] *OpenMP Fortran Application Program Interface*, November 2000. Version 2.0.

[2] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, (111) :3–57, 1993.

[3] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 633–674. Oxford University Press, 1992.

[4] Robert L. Baber. The prolonged metamorphosis of a software engineer. In Robert L. Glass, editor, *In the Beginning — Recollections of Software Pionneers*, pages 174–208. IEEE Computer Society, Los Alamitos, California, 1998.

[5] Roland Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[6] Utpal Banerjee. *Loop Transformation for Restructuring Compilers – The Foundations*. Kluwer Academic Publishers, 1993.

[7] Henk Barendregt. *The lambda calculus, its syntax and semantics*. North-Holland, 1984.

[8] Henk Barendregt. The impact of the lambda calculus. *Bulletin of Symbolic Logic*, 3(2) :181–215, 1997.

[9] Guy E. Blelloch. NESL : A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, September 1995.

[10] Peter Borovansky, Horatiu Cirtea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. *ELAN User Manual*. LORIA, December 1998. ELAN V3.3.

[11] Luc Bougé. The data parallel programming model : A semantic perspective. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, pages 4–26. Springer, 1996.

[12] Luc Bougé and David Cachera. A logical framework to prove properties of alpha programs (revised version). Technical Report 97-16, École Normale Supérieure de Lyon, France, May 1997.

[13] Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages : Semantics and implementation. *Future Generation Computer Systems*, 8 :363–378, 1992.

[14] S. Brookes and S. Geva. Continuous functions and parallel algorithms on concrete data structures. In *Mathematical Foundations of Programming Semantics, Proceedings of the Seventh International Conference*, pages 326–349, Pittsburgh, 1992. Springer-Verlag, LNCS 598.

[15] Philippe Clauss and Benoît Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. 4th Annual Workshop in Interaction between Compilers and Computer Architectures, INTERACT-4, Toulouse, January 2000.

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *Maude : Specification and Programming in Rewriting Logic*. SRI International, March 1999.

[17] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, third edition, 1987.

[18] Fabien Coelho, Cécile Germain, and Jean-Louis Pazat. *State of the Art in Compiling HPF*, volume LNCS 1132, pages 104–133. Springer-Verlag, May 1996.

[19] Patrick Cousot. Methods and logics for proving programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 841–993. Elsevier Science Publishers, 1990.

[20] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, pages 243–320. Elsevier Science Publishers, 1990.

[21] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. Submitted to *Theoretical Computer Science*, 2000.

[22] Claude D. Diderich. *Automatic Data Distribution for Massively Parallel Distributed Memory Computers*. PhD thesis, École polytechnique fédérale de Lausanne, 1998.

[23] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[24] Simon Peyton Jones [editor] and John Hugues [editor] et al. *The Haskell 98 Report — Haskell 98 : A Non-Strict, Purely Functional Language*, February 1999. Également disponible en ligne : `http://www.haskell.org/onlinereport` (page consultée le 21 mars 2002).

[25] HPF Forum. *High Performance Fortran Language Specification*, 2.0 edition, January 1997.

[26] Stéphane Genaud. *Transformations de programmes* PEI *: application au parallélisme de données*. PhD thesis, Université Louis Pasteur, Strasbourg, January 1997.

[27] Ph. Gerner and E. Violard. A theoretical framework of data parallelism and its operational semantics. In Arndt Bode, Thomas Ludwig, and Roland Wismüller, editors, *Proceedings of Euro-Par 2000 – Parallel Processing,* volume 1900 des *LNCS*, pages 668–677, Munich, September 2000. Springer Verlag.

[28] Philippe Gerner. Interprétation fonctionnelle de programmes PEI. Mémoire pour le Diplôme d'Études Approfondies, Université Louis Pasteur, June 1997.

[29] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In *Parallel Symbolic Languages Systems (International Workshop PSLS'95), LNCS 1068*, pages 209–215, I. Takayasu and R. H. Jr. Halstead and C. Queinnec, 1995. Springer Verlag.

[30] Jean-Yves Girard. Linear logic, its syntax and semantics. In Girard, Lafont, and Regnier, editors, *Advances in Linear Logic*. Cambridge University Press, 1995. London Mathematical Society Lecture Notes Series 222.

[31] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Computer Science. Cambridge University Press, 1989. CTCS 7.

[32] Paul Gochet and Pascal Gribomont. *Logique*, volume 2 : méthodes formelles pour l'étude des programmes. Hermès, 1994.

[33] Joseph Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1) :49–67, March 1991.

[34] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, pages 159–191, 1992.

[35] Joseph Goguen. An introduction to algebraic semiotics, with application to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphor, Analogy and Agents,* LNAI, Vol. 1562, pages 242–291. Springer Verlag, 1999.

[36] Joseph Goguen and Rod Burstall. Institutions : Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39 :95–146, 1992.

[37] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[38] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1) :55–101, August 2000. Special issue on Algebraic Engineering, edited by Chrystopher Nehaniv and Masami Ito.

[39] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24 :68–95, January 1977.

[40] Joseph Goguen, Timothy Winkler, José Meseguer, Prof. Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Grant Malcolm, editor, *Software Engineering with OBJ : algebraic specification in action*. Kluwer, 2000.

[41] R. Goldblatt. *Topoi : the categorial analysis of logic*, volume 98 of *Studies in Logic*. North-Holland, revised, 1984 edition, 1979.

[42] Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Preference logic programming. In *International Conference on Logic Programming*, pages 731–745, 1995.

[43] William Groppa, Ewing Lusk, and Anthony Skjellum. *Using MPI*. Scientific and Engineering Computation. MIT Press, 2nd edition, November 1999.

[44] C. A. Gunter and D.S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. Elsevier Science Publishers, 1990.

[45] Y. Gurevich. *Evolving Algebras 1993 : Lipari Guide*, pages 9–36. Oxford University Press, 1995.

[46] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Selected papers from CSL'92 (Computer Science Logic)*, volume LNCS 702, pages 274–308. Springer Verlag, 1992.

[47] G. Hains, F. Loulergue, and J. Mullins. Concrete data structures and functional parallel programming. *Theoretical Computer Science*, 258(1–2) :233–267, April 2001.

[48] Gaétan Hains and John Mullins. A categorical model of array domains. Technical Report 94-43, École Normale Supérieure de Lyon, France, December 1994.

[49] Richard W. Hamming. *Numerical Methods for Scientists and Engineers*. McGraw-Hill, New York, 1962.

[50] Sven Ove Hansson. *The Structure of Values and Norms*. Cambridge University Press, 2001.

[51] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[53] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 4(2) :335–355, 1973.

[54] Neville Holmes. Some comments on the coding of programs. *Computer*, 33(11) :128 & 126–127, November 2000.

[55] HPF Forum. *High Performance Fortran Language Specification*, 1.0 edition, 1993.

[56] ISO. *Fortran 90*. May 1991. ISO/IEC 1539 : 1991 (E).

[57] C. Barry Jay. Data categories. In *Proceedings of CATS'96 (Computing : The Australian Theory Symposium)*, January 1996.

[58] C. Barry Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, 37 :207–224, 2000.

[59] Andrew J.I. Jones and Marek Sergot. *On the Characterisation of Law and Computer Systems : The Normative Systems Perspective*, chapter 12. John Wiley and Sons, 1993.

[60] ISO/IEC JTC1/SC22/WG14. Programming languages – C – Committee draft. January 1999.

[61] ISO/IEC JTC1/SC22/WG5. X3j3/96-007 – Fortran 95 – Committee draft. March 1996.

[62] Charles H. Koelbel, David B. Loveman, Robert S. Shreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation series. MIT Press, 1994.

[63] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Symposium on Principles of Programming Languages*, pages 283–294, 2002.

[64] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 2nd edition, 1998.

[65] James R. Larus. Whole program paths. In *Proceedings of SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99)*, Atlanta, Georgia, May 1999.

[66] Muriel Lefebvre. *Images, écritures et espace de médiation — Étude anthropologique des pratiques graphiques dans une communauté de mathématiciens*. PhD thesis, 2001.

[67] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. Technical Report 1997/8, PRiSM, Versailles University, France, March 1997.

[68] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.04 : Documentation and user's manual*, December 2001. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[69] Björn Lisper. *Data Parallelism and Functional Programming*, volume LNCS 1132, pages 220–251. Springer-Verlag, May 1996.

[70] Frédéric Loulergue. *Conception de langages fonctionnels pour la programmation massivement parallèle*. PhD thesis, Janvier 2000.

[71] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, 1993. To appear in *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[72] Narciso Martí-Oliet and José Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds : From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.

[73] Guerino Mazzola. Objective C via category theory. Draft Document, 1996.

[74] J. Meseguer. Rewriting logic as a semantic framework for concurrency : a progress report. In *Proc. CONCUR'96*, number LNCS 1119, pages 331–372. Springer, August 1996.

[75] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96 :73–155, 1992.

[76] Olivier Michel. *Représentations dynamiques de l'espace dans un langage déclaratif de simulation*. PhD thesis, Université de Paris-Sud, U.F.R. scientifique d'Orsay, December 1996.

[77] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5) :271–280, May 1979.

[78] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, MA, 1991.

[79] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[80] Eugenio Moggi. Notions of computation and monads. *Information And Computation*, 1(93), 1991.

[81] C. Mongenet. Data compiling for systems of uniform recurrence equations. *Parallel Processing Letters*, pages 245–257, 1994.

[82] Caroll Morgan. *Programming from Specifications*. Prentice Hall International, 1990.

[83] Peter D. Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 575–631. Elsevier Science Publishers, 1990.

[84] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992. CTCS 26.

[85] Peter D. Mosses. Theory and practice of action semantics. Technical Report BRICS RS-96-53, Basic Research in Computer Science (BRICS), University of Aarhus, December 1996.

[86] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ User's Manual – ver.1.4–*.

[87] David L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5) :330–336, May 1972.

[88] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[89] Benjamin C. Pierce. A taste of category theory for computer scientists. Technical Report CMU-CS-90-113R, Carnegie Mellon University, September 1990.

[90] Damien Ploix. *Élaboration, réalisation et évaluation d'un environnement de programmation analogique*. PhD thesis, Université Paris 8, 1999.

[91] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[92] Bruno Poizat. *A Course in model theory – an introduction to contemporary mathematical logic*. Springer-Verlag edition, 2000.

[93] Patrice Quinton, Sanjay Radjopadhye, and Doran Wilde. On deriving data parallel code from a functional program. In *Proceedings of the 9th International Symposium on Parallel Processing, IPPS'95*, pages 766–773, Santa Barbara, CA, April 1995. IEEE Computer Society Press.

[94] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Publishing Company, Inc., 1990.

[95] David B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *NATO ARW "Software for Parallel Computation"*. Springer-Verlag, 1993.

[96] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, January 1994.

[97] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, May 2002. Version 7.3.

[98] R. D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.

[99] Thinking Machines Corp. *C* Programming Guide*, November 1990.

[100] Thinking Machines Corp., 245 First Street, Cambridge, Massachussetts 02142. *Connection Machine : Programming in *Lisp*, 1991.

[101] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*. Special issue on linear logic, to appear.

[102] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103, August 1990.

[103] Jan van Leewen, editor. *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics. Elsevier Science Publishers, 1990.

[104] A. van Wijngaarden et al., editor. *Revised Report on the Algorithmic Language ALGOL 68*. Springer-Verlag, 1976. Published in 1971 under title : Report on the algorithmic language ALGOL 68.

[105] Eric Violard. Typechecking of PEI programs. In *LNCS 1300, Proceedings of EUROPAR'97*, pages 521–529. Springer-Verlag, August 1997.

[106] Eric Violard. A formal semantics of data-parallel languages. Technical Report ICPS RR 98-04, Université Louis Pasteur, France, December 1998. URL : `http ://icps/pub-98/pub-98-04.ps.gz`.

[107] Eric Violard. What really is data parallel programming ? Technical Report ICPS RR 00-01, Université Louis Pasteur, France, January 2000.

[108] Eric Violard. A semantic framework to address data locality in data parallel languages. Technical Report ICPS RR 01-05, Université Louis Pasteur, France, June 2001.

[109] Eric Violard, Stéphane Genaud, and Guy-René Perrin. Refinement of data-parallel programs. In R. Bird and L. Meertens, editors, *Proceedings of the IFIP Working Conference on Algorithmic Language and Calculi*, pages 107–131. Chapman & Hall, February 1997.

[110] Eric Violard and Guy-René Perrin. Pei : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18 :1167–1184, 1992.

[111] Patrick Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Dipartimento di Informatica, Università di Pisa, December 1995.

[112] Patrick Viry. A rewriting implementation of pi-calculus. Technical Report R-96-30, Dipartimento di Informatica, Università di Pisa, March 1996.

[113] Larry Wall, Tom Christiansen, and John Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

[114] Doran K. Wilde. The alpha language. Technical Report 827, IRISA, Rennes University, France, May 1994.

[115] Martin Wirsing. Algebraic specification. In Jan van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, pages 675–788. Elsevier Science Publishers, 1990.