Rapport de stage de DEA

Structures de Données Régulières et Analyse de Flot

Albert Cohen[1]

Sous la direction de Jean-François Collard[2]

24 juin 1996

1. Élève de l'École Normale Supérieure de Lyon (ENS-Lyon).
2. Chargé de Recherches CNRS à l'Université de Versailles St-Quentin en Yvelines.

Résumé

L'analyse de flot de données permet, dans un programme généralement écrit dans un langage impératif, de retrouver pour chaque lecture d'une valeur en mémoire, l'instance de l'instruction qui l'a calculée. Pour les langages comme Fortran qui font appel à des nids de boucles et des tableaux, il existe des techniques performantes. L'objet de ce stage est d'étudier les extensions de l'analyse de flot à des structures de données plus complexes et à la récursivité.

Nous introduisons un modèle mathématique pour décrire les structures de données, un nouveau langage pour les manipuler, et nous présentons ensuite des résultats très prometteurs sur l'analyse de flot pour ce langage. Cet exposé s'accompagne d'une comparaison avec d'autres travaux autour de ce domaine, une critique des choix effectués et un aperçu des développements futurs.

Abstract

Dataflow analysis computes for every value read in a memory cell, the very operation which produced it. It is usually applied to imperative languages. For Fortran-like languages with loop nests and arrays, powerful techniques have been developed. This work deals with handling more complex data structures and recursive programs.

We define a theoretical model to describe data structures, a new language to handle them, and further we demonstrate promising results on dataflow analysis for this language. This paper comes along with a comparison to some related work, an evaluation of our design choices and a presentation of future extensions.

Mots clés : Analyse de Flot de Données, Structures de Données, Graphes de Cayley, Programmation entière, Parallélisation Automatique.

Remerciements : Ce stage de DEA a été effectué du 5 mars au 24 juin 1996 au Laboratoire PRiSM de l'Université de Versailles St-Quentin en Yvelines, sous la direction de Jean-François Collard, pour l'obtention du DEA d'Informatique de Lyon. Je le remercie pour sa disponibilité constante et pour la confiance qu'il m'a accordée pour travailler sur ce sujet très enrichissant. Je remercie aussi toute l'equipe PAF, dirigée par Paul Feautrier, pour son accueil.

Mise en page, LaTeX 2ε.

Table des matières

Introduction

Rassembler des informations sur les données manipulées par des langages de programmation est depuis longtemps devenu une tâche indispensable du processus de compilation et d'optimisation, c'est l'analyse statique ([1]). Les programmes gourmands en ressources et en calculs passent le plus cher de leur temps à exécuter des boucles ou des algorithmes récursifs qui opèrent sur des structures de données répétitives de grande taille, et c'est naturellement sur ces programmes que se concentrent les recherches en matière d'analyse statique. Celles-ci peuvent prendre la forme de prospections générales des propriétés d'un programme, ou bien se spécialiser dans la recherche des *dépendances* ou l'analyse de *flot de données*. Elles conduisent à des résultats très satisfaisants en matière d'optimisation de langages impératifs classiques et a permis plus récemment d'améliorer sensiblement les performances à l'exécution de langages plus abstraits ([18, 15]). Ceux-ci autorisent la manipulation directe de structures de données complexes (arborescentes pour les langages fonctionnels), et font généralement appel a la récursivité comme moyen d'expression naturel des algorithmes. Les performances de ces langages reposent sur le typage fort de leurs expressions et structures de données (langages orientés objet ou fonctionnels typés à la ML par exemple) ; un langage comme le C ne permettant pas une analyse statique très fine car les pointeurs sont manipulés trop librement (des résultats intéressants existent néanmoins, [8]).

Cependant, les techniques employées généralement traitent chaque structure de données comme un objet indivisible (scalaire). Au contraire, la parallélisation, la vectorisation, certaines optimisations fines (placement de données, gestion automatique de la mémoire) et la vérification de programmes s'appuient sur l'existence de références indépendantes à différentes parties de ces structures. L'analyse de dépendances consiste à employer des tests basés sur la recherche d'*alias* entre références (principalement pour des indices de tableaux) ; mais les méthodes générales conduisent à des résultats approximatifs ou incapables de distinguer les différents types de dépendance (cf. 1.3). D'autres méthodes plus précises recherchent pour chaque cellule mémoire accédée en lecture (le *puits*), l'opération qui l'a produite (la *source*) : on parle alors d'*analyse de flot de données* (*dataflow analysis*). Dans le cas des tableaux, il existe des techniques récentes très performantes, notamment l'*analyse de flot de données pour les tableaux* (*Array Dataflow Analysis*, ADA, [10, 25]), ou la *Value-Based Dependance Analysis* ([28]).

Des techniques aussi fines que l'ADA perdent malheureusement en généralité ce qu'elles gagnent en efficacité et en précision. Toute possibilité de contrôle dynamique est exclu, et la manipulation de structures de données se limite aux indexations affines dans des tableaux (cf. 4.1). Il n'y a pas non plus de support des appels récursifs de fonctions ou de procédures. Ces restrictions sont parfois acceptables, notamment pour une large catégorie de calculs scientifiques faisant intervenir de l'algèbre linéaire sur des matrices pleines, mais dans la pratique seul un nombre limité d'algorithmes peuvent respecter les conditions de l'ADA. Il apparaît alors nécessaire d'étendre ces analyses, c'est à dire d'étendre les structures de *données* et de *contrôle* considérées. Lorsque (comme pour l'ADA) toute l'exécution est connue statiquement (pas de contrôle dépendant de paramètres dynamique), on parle d'analyse statique exacte. Tout n'est pas perdu néanmoins lorsque le contrôle est dynamique (présence de `if`, de `while`...) : en général il n'est plus possible de trouver statiquement une unique source à un puits. On parle alors d'analyse statique *floue* (*fuzzy*). Des avancées très intéressantes ont lieu dans ce domaine, comme la *Fuzzy Array Dataflow Analysis* (FADA, [3]).

Le problème que nous nous sommes posés est à replacer dans le cadre des recherches sur la généralisation des techniques d'analyse de flot pour les tableaux, et sur de nouvelles techniques adaptées à d'autres classes de programmes. Il s'agit en l'occurrence d'extension à des structures de données plus complexes et à des algorithmes exprimés de façon récursive. Notre contribution s'articule autour de trois directions. Il nous a d'abord semblé indispensable de définir un cadre mathématique clair et puissant pour définir des structures *régulières* plus générales que les tableaux sans compromettre l'exploitation de leurs propriétés statiques. Le modèle retenu repose sur les graphes de Cayley. Dans un deuxième temps nous présentons les caractéristiques originales d'un langage conçu spécialement pour définir des structures de données régulières et écrire des algorithmes qui opèrent sur celles-ci. Nous défendrons les orientations qui ont dirigé l'écriture ce ce langage (validées par l'implémentation partielle d'un compilateur), et les aspects importants de la compilation seront abordés. Enfin, dans le dernier chapitre nous exposerons les résultats concernant l'objet de ce stage proprement dit : l'extension de l'analyse de flot de données à ce langage. Ce dernier chapitre est beaucoup plus technique que les précédents mais il reprend la plupart des concepts développés pour l'ADA. Un premier chapitre sera consacré à la présentation générale du domaine et permettra par la suite de comparer les différentes approches.

Chapitre 1

Présentation

1.1 Une analyse de flot peut en cacher une autre

1.1.1 Vers une classification des analyses de flot de données

Comme on l'a vu dans l'introduction, des problèmes assez distincts relèvent de l'analyse de flot de données. Nous ne parlerons pas ici de l'analyse «scalaire» traditionnelle en optimisation, mais des analyses utilisées en parallélisation automatique.

Le tableau suivant tente de classer les analyses en fonction des algorithmes et des structures de données sur lesquelles ils opèrent. On distinguera instructions d'appel (fonctions ou procédures), instructions conditionnelles et instructions élémentaires. On pourra classer en instructions élémentaires les assignations, mais aussi les appels de fonction ou les instructions conditionnelles qui n'interfèrent ni avec le contrôle ni avec les données sur lesquels portent l'analyse. Enfin, un appel conditionnel sera dit *gardé statiquement* si l'on peut décider à la compilation de la validité de la condition.

contrôle données	instructions élémentaires	boucles for à bornes statiques	appels récursifs gardés statiquement
scalaires	analyses classiques ADA...	ADA...	1
tableaux	sans grand intérêt	ADA...	2
structures complexes (arbres, graphes...)	sans grand intérêt	3	4

1.1.2 Objectifs

Dans le cadre de ce stage de DEA, il s'agit principalement d'étudier l'extension aux structures de données complexes des techniques d'analyse exacte, voire d'en découvrir de nouvelles (catégories 3 et 4). Mais la plupart des algorithmes manipulant ces structures (ce sont des arbres bien souvent) s'appuient sur des parcours récursifs. Dans certains cas (structures de données «creuses») il peut être fait usage de parcours récursifs et de boucles imbriqués. C'est pourquoi il parait difficile de ne pas considérer simultanément les structures de contrôle récursives (catégories 1 et 2), la réciproque n'ayant pas nécessairement été vraie (algorithmes récursifs sur des tableaux).

On note que l'analyse floue peut se greffer sur toutes les catégories ci-dessus ; les concepts proposés par la FADA semblent d'ailleurs suffisamment généraux pour s'étendre à l'étude de la «dynamicité» du contrôle et même aux incertitudes dues au non respect des contraintes imposées par la plupart des analyses exactes envisageables. Cependant on se limitera principalement à des analyses exactes (cas contraire indiqué expressément), tout en gardant à l'esprit les possibilités d'extension future vers la gestion du contrôle dynamique ; on s'efforcera donc lors de chaque introduction de nouveau concept de ne pas réduire artificiellement sa généralité ou son pouvoir d'expression.

1.2 Les structures de données complexes en algorithmique

De nombreux algorithmes s'expriment naturellement en termes de parcours de graphes et d'arbres, c'est à dire souvent des structures reliant des cellules semblables (des *nœuds*).

C'est notamment le cas pour tous les calculs mathématiques (calcul matriciel, calcul formel, simulation numérique) quelque soit la représentation des objets considérés. On retrouve aussi des algorithmes de tri, de gestion de base de donnée, d'imagerie et de traitement du signal, de routage dans des réseaux, de compilation, etc...

La plupart des structures considérées comportent des régularités intrinsèques très fortes : des grilles, des tableaux, des arbres binaires, des arbres rationnels (types récursifs à la ML), des graphes k-réguliers, etc... Elles peuvent aussi posséder des propriétés structurelles plus diffuses : graphes acycliques, tas, arbres rouge-noir, extension limitée par une structure régulière, etc...

Toutes ces propriétés sont intéressantes pour le programmeur et le compilateur, puisqu'elles apportent des informations sur les manipulations que l'on va pouvoir effectuer et leur étendue en mémoire. Elles vont aussi permettre de réutiliser des modules existants voire même d'utiliser des constructions du langage spécialement étudiées (si elles existent).

Bien sûr il est possible (c'est malheureusement la solution la plus largement retenue par les programmeurs) de tout ramener aux tableaux ou d'utiliser des pointeurs explicites ; il est même difficile de faire autrement avec les langages impératifs classiques. Même les langages à objets utilisent des références ou des pointeurs explicites, mais en les encapsulant dans des classes ; ils améliorent ainsi la lisibilité et la sécurité du code mais n'informent pas le compilateur des propriétés structurelles pour autant.

1.3 Alias et dépendances

Une fois les structures de données définies, on va pouvoir écrire des algorithmes qui les parcourent. Par exemple des *nids de boucles* pour indexer des tableaux multi-dimensionnels. Comme on l'a dit précédemment, le problème central de l'analyse statique *fine* est celui de l'*analyse d'alias* (*alias analysis*), c'est la condition préalable à toute recherche des dépendances de données. Le problème de l'analyse de dépendances est indécidable en général. Il suffit en réalité que des paramètres inconnus statiquement apparaissent dans les accès aux structures de données pour rendre toute analyse exacte impossible. C'est pourquoi l'on préférera souvent utiliser la notion de *memory disambiguation* : deux références pour lesquelles on ne peut montrer leur indépendance sont considérées dépendantes. Le problème devient alors de tirer parti au maximum des informations disponibles statiquement à la compilation pour minimiser le nombre de dépendances superflues. C'est pourquoi le langage joue un rôle primordial en permettant ou non de transmettre ces informations au compilateur.

Les dépendances qui découlent de conflit d'accès (alias de références) sont de trois sortes : de flot (producteur → consommateur), de sortie (producteur → producteur), ou anti-dépendance (consommateur → producteur). Malheureusement lorsqu'un grand nombre d'accès à une même cellule mémoire sont effecutés, le nombre de dépendances engendré augmente de façon quadratique. C'est pour cette raison que l'on préfère effectuer des analyses de flot, et donc rechercher les sources des accès en lecture (les puits) (cf. 4). La *dépendance directe* est une notion intermédiaire entre celle de dépendance et celle de source. Il s'agit pour une instruction d'écriture donnée de l'instance qui assure la dépendance de flot la plus*courte* vers l'instruction de lecture. La source de cette instruction de lecture est donc à rechercher parmis les dépendances directes. On appellera *opérations* les instances d'une instruction au cours de l'exécution d'un programme.

1.4 Quelques techniques d'analyse d'alias en présence de structures complexes

1.4.1 En présence de pointeurs

Il est parfois possible de conserver un certain nombres de propriétés sur des références explicites par pointeurs en utilisant la définition des structures parcourues (`struct` en C). Alain Deutsch propose des adaptations des techniques de *treillis* (treillis d'intervalles, de systèmes d'inégalités...) aux chemins d'accès symboliques (du type `X->left->right->value`) du C. On peut alors à l'aide de calculs de points fixes dériver itérativement des équations de flot ([8] propose une bonne bibliographie sur le sujet).

Ces résultats intéressants en termes d'optimisation de code ne sont malheureusement pas suffisamment précis pour être utilisables en parallélisation automatique, dès qu'il y a des boucles ou algorithmes récursifs complexes.

1.4.2 Annotation de code

Pour remédier à la trop grande versatilité des pointeurs, une idée naturelle consiste à guider le compilateur en ajoutant au langage des annotations qui imposent (ou qui simulent) une restriction de l'utilisation de ceux-ci. Par exemple une liste chaînée peut être décrite par la structure suivante :

```
struct list {
  int value;
  struct list *next;
}
```

Mais une telle structure peut aussi bien représenter une liste circulaire, un arbre orienté vers la racine...

Laurie Hendren et Joseph Hummel ont introduit le système *Abstract Description of Data Structures* (ADDS, [17]) qui propose d'ajouter aux champs récursifs de pointeurs (e.g. `struct list *next`) des annotations de *direction* et de *sens* qui peuvent permettre de préciser la structure abstraite d'une structure.

Dans certains cas, ces annotations améliorent sensiblement les résultats des analyses d'alias. Des outils de développement ou de vérification existent qui exploitent ce genre de techniques, mais ils imposent un travail supplémentaire au programmeur et sont susceptibles d'introduire des erreurs difficilement détectables.

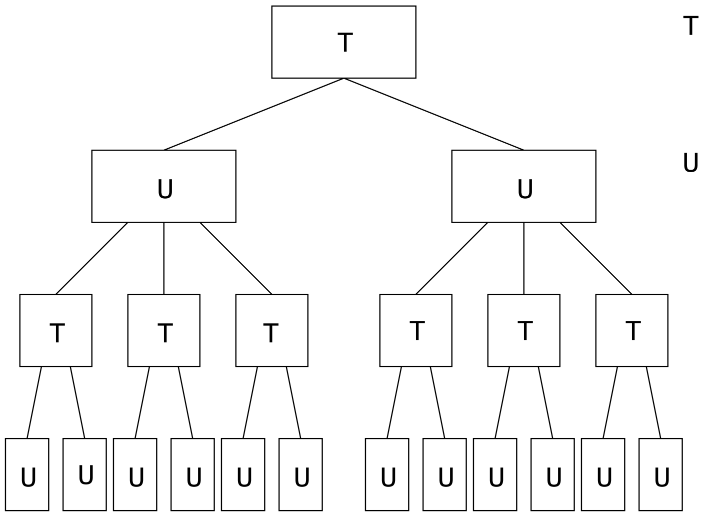
De plus, ce genre de méthodes ne font que répondre partiellement au problème : si elles augmentent la précision des informations statiques accessibles au compilateur, elles ne proposent pas d'algorithmes pour effectuer une analyse de flot proprement dite.

1.4.3 Types récursifs et langages fonctionnels

La notion de type récursif est issue du rapprochement entre grammaires d'arbres et langages fonctionnels. Les types de données récursifs peuvent être vus en tant que cas particuliers de grammaires d'arbres ou les *non-terminaux* sont sont appelés types et les productions sont de la forme :

$$T \quad \rightarrow \quad v(a_1 : T_1, \ldots, a_n : T_n)$$

Où T et les T_i sont des types. v est ainsi déclarée variante du type T contenant les champs $a_1 \ldots a_n$. Les instanciations d'un type de données récursif sont constituées par les arbres de dérivation de la grammaire hors-contexte sous-jacente (fig. 1.1). Ils peuvent être implémentés à l'aide de pointeurs explicites, mais le programmeur (en ML par exemple) n'a jamais accès à cette représentation interne.




```
T -> couple (l : U, r : U)
  | feuille

U -> triplet (l : T, d : T, r : T)
  | feuille
```

FIG. 1.1 - *Exemple de type récursif*

Ces types sont naturellement associés aux langages fonctionnels par leur structure arborescente qui favorise leur exploration naturelle par des algorithmes récursifs. Ils définissent de plus de manière statique toutes les propriétés structurelles des structures de données qu'ils engendrent. C'est à dire que le compilateur ayant typé une expression référençant un nœud d'une telle structure connaît automatiquement tous les chemins issus de ce nœud dans la structure (sans tenir compte de ses bornes) ; c'est exactement ce que requiert une analyse statique. En réalité il existe une logique (dite monadique du second ordre) décidable qui permet d'exprimer toutes les relations d'accessibilité dans les structures ([5]).

Bien entendu il n'est plus possible de décrire des graphes plus généraux que les arbres avec ces types, c'est leur défaut principal. il existe cependant des tentatives de généralisation, en insérant par exemple des *routages* (sous forme d'expression régulières) dans la grammaire qui se superposent aux arbres pour créer des graphes plus complexes (listes circulaires, matrices creuses, arbres avec arêtes vers la racine...) ; il est même possible d'étendre la logique à ces types «graphiques» et de montrer qu'elle reste décidable ([19]).

1.5 Notre approche du problème

A la lumière de ces nombreux travaux dans le domaine de l'analyse de flot nous avons fait le choix de restreindre la généralité des structures que l'on cherche à analyser. Comme pour les types récursifs on recherche une expression directe des caractéristiques structurelles dans un langage adapté. Nous avons donc défini une notion de *régularité* et un langage permettant de définir et de manipuler les structures *régulières*. Cette approche est très semblable à celle proposée dans [2] pour étendre le parallélisme de données à des structures plus générales que les tableaux.

Notre démarche est dirigée par un objectif de généralisation des domaines auxquels s'appliquent l'analyse de flot de données, en se basant sur les points forts des techniques actuelles pour en trouver de nouvelles ou plus simplement pour les adapter à d'autres problèmes. Ceci nous a conduit à nous intéresser aux graphes de Cayley en tant que modèles de régularité structurelle, puisqu'ils autorisent des opérations algébriques analogues aux indexations affines dans les tableaux. Et nous avons aussi tenté d'exploiter des techniques éprouvées (en l'occurrence PIP [13] ou Omega [27]) pour résoudre des systèmes de contraintes, lorsque ceux-ci s'y prêtaient (lorsque les structures «ressemblent» à des tableaux).

Chapitre 2

Régularité

Dans le chapitre précédent nous avons suggéré de partir d'un nouveau modèle de structures capable de généraliser un grand nombre de structures pour lesquelles une analyse de flot efficace semble possible. Nous avons trouvé cette définition de la *régularité* dans les travaux de Jean-Louis Giavitto, Olivier Michel et Jean-Paul Sansonnet sur une extension du langage déclaratif à parallélisme de données $8_{1/2}$ ([14]).

2.1 Modèle mathématique

Notre objectif est de caractériser une large classe de graphes que l'on considérera réguliers. L'idée de [14] consiste à rendre la notion de *voisinage* des nœuds explicite. Cela est réalisé en définissant une *forme* (*shape*) qui définit pour chaque *nœud* l'ensemble de ses voisins dans le graphe (orienté).

2.1.1 Préliminaires

Commençons par quelques rappels de théorie des groupes... Classiquement on utilisera indifféremment $(\mathcal{G}, .)$ ou $\mathcal{G}$ pour désigner un groupe, et ϵ (pas e) représentera l'élément neutre.

Définition 1 (base, générateurs)

- $\{e_1, \ldots, e_n\}$ *est une base d'un groupe* $\mathcal{G}$ *si tout élément de* $\mathcal{G}$ *s'écrit comme composition des* e_i *;*
- $\mathcal{E} = \{e_1, \ldots, e_n\}$ *engendre* $\mathcal{G}$ *si* $\mathcal{E} \cup \mathcal{E}^{-1} = \{e_1, \ldots, e_n, e_1^{-1}, \ldots, e_n^{-1}\}$ *est une base de* $\mathcal{G}$.

Définition 2 (Représentation de groupe) *On dit qu'un groupe* $\mathcal{G}$ *admet une* représentation $< \mathcal{E}, \Re >$ *où* $\mathcal{E} \subset \mathcal{G}$ *et* $\Re$ *est une* congruence[1] *sur le langage* $(\mathcal{E} \cup \mathcal{E}^{-1})^*$ *(ensemble des mots sur* $\mathcal{E} \cup \mathcal{E}^{-1}$*), si* $\mathcal{G} = (\mathcal{E} \cup \mathcal{E}^{-1})^*/\Re$.

Les éléments d'un groupe représenté par $< \mathcal{E}, \Re >$ sont donc des classes de mots sur $\mathcal{E} \cup \mathcal{E}^{-1}$ rassemblant tous les mots en relation par $\Re$ ($\Re$ étant une équivalence, on a bien sûr $g.g^{-1} = \epsilon$).

Définition 3 (groupe de type fini) *On appellera groupe de type fini tout groupe admettant une représentation* $< \mathcal{E}, \Re >$ *finie, c'est à dire telle que* $\mathcal{E}$ *soit fini et* $\Re$ *soit la clôture réflexive, symétrique, transitive et par concaténation d'une relation* finie[2].

1. Une *congruence* est une relation d'équivalence respectant la loi de concaténation (ou de composition pour un groupe).
2. Un ensemble fini de couples de mots en relation.

En effet, la clôture réflexive, symétrique et transitive d'une relation finie donne la plus petite équivalence qui la contient, et la clôture par concaténation (telle que $x\Re y$ implique $x.z\Re y.z$ et $z.x\Re z.y$) donne la plus petite congruence qui contient cette équivalence. On confondra souvent une congruence avec les relations qui l'engendrent par clôture.

2.1.2 Formes régulières

La régularité d'une *forme* est donnée par un *groupe de type fini* à travers la définition suivante :

Définition 4 (graphe de Cayley) *Soit $(\mathcal{G}, .)$ un groupe de type fini, et $< \mathcal{E}, \Re >$ une représentation de $\mathcal{G}$, on définit le graphe $G = (\mathcal{G}, \{(g, g.e) | (g, e) \in \mathcal{G} \times \mathcal{E}\})$. G est le graphe de Cayley du groupe $\mathcal{G}$ représenté par $< \mathcal{E}, \Re >$.*

Le graphe de Cayley d'un groupe représenté par $< \mathcal{E}, \Re >$ est donc le graphe dont les sommets sont les éléments de ce groupe et les arêtes relient les éléments à leurs images par la composition avec les générateurs de $\mathcal{E}$.

Définition 5 (forme) *Une* forme *(*shape*) est le graphe de Cayley d'un groupe de type fini.*

Par la suite on ne considérera que des représentations finies, et donc des graphes de Cayley de groupes de type fini. On utilisera la notation A^B pour désigner l'ensemble des applications de B dans A (notation classique, intuitive si l'on pense aux suites réelles de $\mathbb{R}^{\mathbb{N}}$), et $A^{(B)}$ pour les applications d'une partie finie de B dans A (applications à support fini de B dans A).

2.2 Conventions, notations

Une congruence $\Re$ engendrée par des relations finies sera notée $r_1, \ldots, r_n$ où les r_i sont des éléments en relation avec le mot vide ϵ, qui engendrent $\Re$. Par exemple le groupe $(\mathbb{Z}/n\mathbb{Z}, +)$ admet une représentation finie que l'on note $< u; u^n >$.

Définition 6 *On étend aux formes les propriétés classiques des groupes, et on parlera donc de*

- *formes commutatives (ou abéliennes);*
- *formes libres ($\Re = \emptyset$) ;*
- *formes commutatives libres ($\Re = \{e_i.e_j.(e_i.e_j)^{-1} | (i, j) \in \{1, \ldots, d\}\}$)...*

On utilisera souvent le terme «forme» pour désigner l'ensemble des nœuds de celle-ci ou son groupe associé.

On définit une notation pour définir directement des formes sans passer par leur représentation de groupe sous-jacente.

On note $< e_1, \ldots, e_d; r_1, \ldots, r_n >$ la forme associée au graphe de Cayley du groupe représenté par $< \mathcal{E} = \{e_1, \ldots, e_d\}, \Re >$, où $\Re$ est la clôture réflexive, symétrique, transitive et par concaténation de la relation finie $\{(r_1, \epsilon), \ldots, (r_n, \epsilon)\}$.

On définit une notation allégée pour pour des formes *commutatives*

$$\lhd e_1, \ldots, e_d; r_1, \ldots, r_n \rhd$$

est synonyme de :

$$< e_1, \ldots, e_d; r_1, \ldots, r_n, \{e_i.e_j.(e_i.e_j)^{-1} | i, j \in \{1, \ldots, d\}\} \rhd$$

Parmi les formes commutatives, on aura souvent besoin de considérer des graphes d'interconnexion non orientés (possibilité de se déplacer librement dans des «grilles»). Pour cela on utilise la notation

$$\blacktriangleleft\ e_1, \ldots, e_d; r_1, \ldots, r_n\ \blacktriangleright$$

qui est synonyme de :

$$\lhd e_1, \ldots, e_d, e_1^{-1}, \ldots, e_d^{-1}; r_1, \ldots, r_n \rhd$$

L'orientation des arêtes dans une forme est capitale : elle conditionne les accès qui peuvent être effectués relativement à un nœud donné [3]. Il n'est pas possible par exemple de parler du nœud $g.u^{-1}$ dans la forme $\lhd u \rhd$ qui modélise les listes chaînées, par contre on peut référencer le nœud $g.a.b^{-1}$ dans la forme $\blacktriangleleft\ a, b\ \blacktriangleright$ qui modélise les grilles à deux dimensions (les tableaux entre autres). Le fait qu'un nœud soit défini en tant qu'élément du groupe est indépendant de son accessibilité à partir d'un autre nœud, c'est le graphe qui détermine les interconnections.

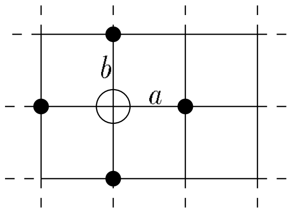
3. Dans la littérature classique sur les graphes de Cayley, on considère généralement des graphes non orientés, seuls les noms des arêtes sont associés à une orientation ; les *formes* que l'on étudie ici se démarquent donc de cette habitude.

2.3 Exemples

Voici quelques exemples de formes simples. D'autres exemples (2.2 et 2.3) sont donnés plus loin.

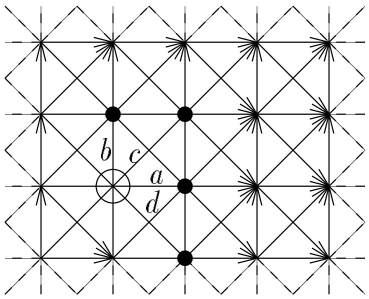
Voisinage de Von-Neuman, grille rectangulaire

$\blacktriangleleft a, b \blacktriangleright$



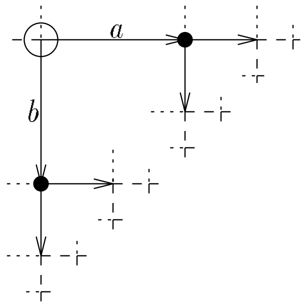
Demi-voisinage de Moore, grille rectangulaire

$\triangleleft a, b, c, d; c.(a.b)^{-1}, d.(a.b^{-1})^{-1} \triangleright$



Arborescence binaire

$< a, b >$



Liste circulaire doublement chaînée

$< u, u^{-1}; u^6 >$

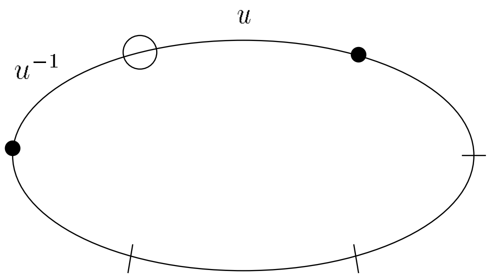


FIG. 2.1 - *Formes classiques, avec leurs groupes associés*

2.4 Problème du mot et formes normales

Les groupes nous apportent simultanément une grande généralité dans le type de formes régulières que l'on peut modéliser et la possibilté d'effectuer directement des compositions, des calculs d'occurrence et des toutes sortes d'opérations algorithmiques basées sur la loi de groupe. Cependant le problème algorithmique principal lors de la manipulation de groupes est celui de la détermination de l'équivalence entre deux écritures d'un même élément. Dans le cas général, ce problème appelé *problème du mot* est indécidable.

Heureusement, dans la pratique, tous les exemples de structures régulières (du produit de matrices pleines à la propagation de la chaleur dans les quasi-cristaux en passant par le tri par tas) que l'on rencontre sont associées à des groupes pour lesquels on peut définir des formes normales, parfois même «canoniques» c'est à dire «naturelles».

On se restreindra donc par la suite à l'étude de formes pour lesquelles sont spécifiées (implicitement ou non) une forme normale. Dans ce cadre le problème du mot est bien entendu décidable, il se réduit au calcul de cette forme normale.

Pour les groupes commutatifs on considérera la forme normale de Smith ([4, 29]) fondée sur l'isomorphisme avec un groupe de la forme

$$\mathbb{Z}^n \times \mathbb{Z}/n_1\mathbb{Z} \times \cdots \times \mathbb{Z}/n_q\mathbb{Z}$$

Où les n_i sont tels que n_i divise n_{i+1}. Il est intéressant de noter que la plupart des techniques actuelles d'analyse de dépendances ou de flot de données, ainsi que de parallélisation automatique rentrent dans le cadre de ces formes commutatives, y compris les *data-fields* de [24], les listes chaînées, les tableaux systoliques, les tores quelconques...

$\blacktriangleleft\ v, h, d; v.h.d^{-1}\ \blacktriangleright$

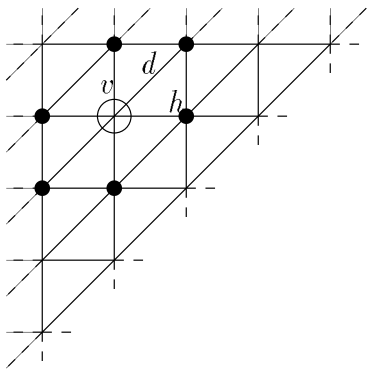


FIG. 2.2 - *Grille triangulaire*

Pour les groupes non commutatifs, le problème est beaucoup plus délicat. Dans le cas extrême des groupes libres, le calcul d'une forme normale se résume à simplifier les expressions de la forme $g.g^{-1}$. Les structures arborescentes sont donc traitées sans problème. Pour les formes non commutatives et non arborescentes, il faudra procéder au cas par cas. Dans l'état actuel, nous ne proposons pas de méthode de description des formes normales qui soit réalisable dans le cadre d'un langage de programmation, mais l'idée de proposer une bibliothèque de formes avec leur formes normales associées pourrait s'avérer utile, dans la mesure ou ces structures non commutatives seraient utiles en pratique (cf. 3.6.3).

Exemple de la structure triangulaire

Si l'on considère l'exemple de la forme triangulaire (fig. 2.2), la forme normale canonique serait une projection sur les coordonnées cartésiennes en écrivant d'abord la composante en h et ensuite la composante en v :

$$v^3.d^2.h^{-1} \rightsquigarrow h.v^5$$

2.5 Structures de données régulières

Définition 7 (type) *Dans ce modèle, l'ensemble des types est le plus petit ensemble contenant :*

- *les types de base ;*
- *les types construits par produit ou somme de types de base ;*
- *les ensembles T^S où T est un type et S une forme, c'est à dire les applications des nœuds du graphe de Cayley S vers T.*

Une structure de données A de type (T, S) est donc un élément de $T^{(S)}$, c'est à dire une partie finie du graphe de Cayley S dans laquelle chaque nœud contient une valeur de type T.

2.6 Substitutions

Une dernière notion mathématique nous sera utile pour manipuler des formes, c'est la substitution sur les mots. Si $X = x_1 \dots x_n$ est un mot sur le langage $\mathcal{L} = \{e_1, \dots, e_d\}$, on note $X\{F_1/e_1, \dots, F_d/e_d\}$ la substitution dans X de toutes les occurrences des lettres e_i par les mots F_i.

Par exemple, $aabacb\{x/a, xy/b, \epsilon/c\} = xxxyxxy$.

Pour rendre les substitutions plus utilisables, on va modifier un peu leur définition habituelle : on les définit comme morphismes d'un monoïde libre (un langage) sur un autre. C'est à dire qu'à *chaque* lettre du premier alphabet est associée un mot du deuxième langage, et que la composition est respectée. Il n'y a pas de lettres substituées et d'autres inchangées, donc pas de substitution partielle. Par conséquent, comme on conserve la même syntaxe, une lettre n'apparaissant pas dans la substitution sera implicitement substituée par ϵ :

$$\{x/a, xy/b\} = \{x/a, xy/b, \epsilon/c\} \text{ (si l'alphabet de départ est } \{a, b, c\})$$

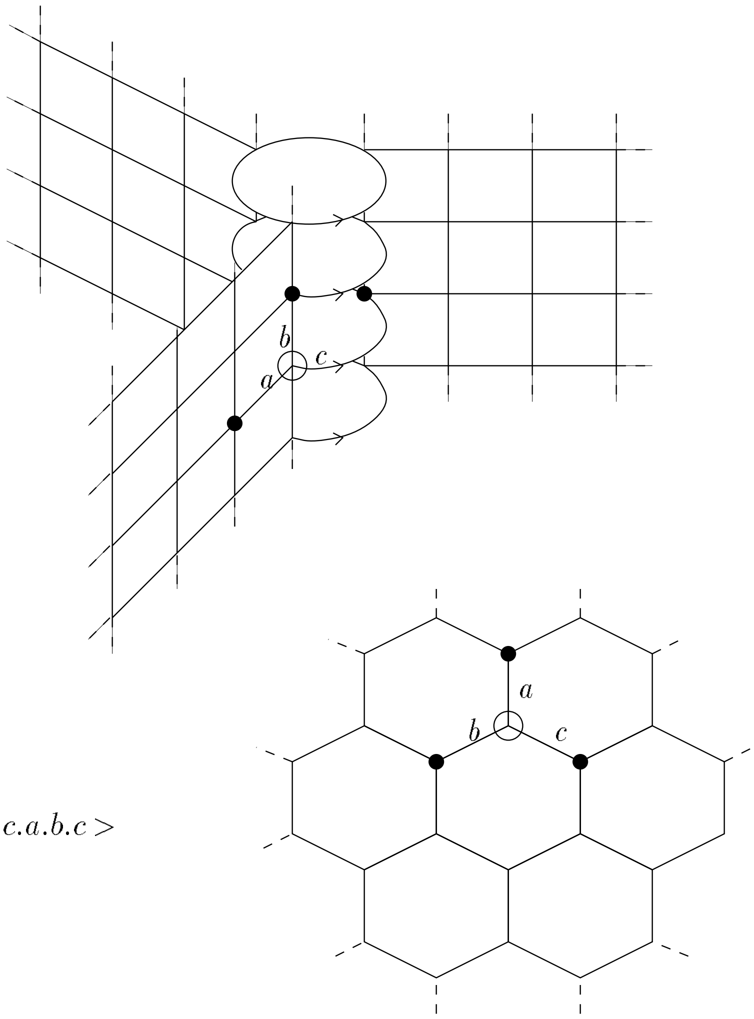
Ce comportement est inhabituel mais c'est le seul compatible avec l'extension aux morphismes entre deux langages différents.

Pour compliquer encore, on utilisera les substitutions en tant qu'applications d'un monoïde libre sur un groupe ou une forme : les mots F_i seront en fait assimilés à des éléments du groupe, et la composition prendra la place de la concaténation. On parlera d'ailleurs fréquemment de *mots* ou de *mots d'accès* au lieu d'éléments d'un groupe, et de *lettres* ou de *directions* au lieux de générateurs.

En considérant par exemple une substitution vers la forme $\lhd u; u^3 \rhd$, c'est à dire le graphe de Cayley de $\mathbb{Z}/3\mathbb{Z}$, on a

$$aabacb\{u/a, \epsilon/b, u^2/c\} = u.u.u.u^2 = u^5 = u^2$$

$\lhd a, b, c, a^{-1}, b^{-1}; c^3 \rhd$



$< a, b, c, a^{-1}, b^{-1}, c^{-1}; a^2, b^2, c^2, a.b.c.a.b.c >$

FIG. 2.3 - *Formes non commutatives*

Chapitre 3

Le langage LEGS

Les concepts de structure régulière nécessitent la définition de nouvelles constructions syntaxiques, décrivant leur construction et leur manipulation à l'aide d'une sémantique propre. Nous avons donc défini le *Language for Exploration of Group-shaped Structures* (LEGS). On ne décrira pas la sémantique exacte du langage en termes mathématiques mais on passera en revue les principales originalités et on les illustrera par des exemples.

3.1 Caractéristiques générales

Ce langage, dérivé du Pascal et du C ne contient pas toutes les constructions indispensables à la réalisation de programmes réels, il s'agit d'un langage «jouet». Il permettra de fixer les idées et la syntaxe des codes à analyser (comme Fortran pour la parallélisation automatique).

A la différence du Pascal, le point-virgule ; est un terminateur d'instruction (comme en C). Pour des raisons artistiques similaires `{` et `}` jouent le rôle de délimiteurs de blocs...

Un bloc est décomposé en partie déclarative et partie opérative, comme en C, ce qui permet de déclarer plus simplement des variables locales (elles ne sont pas subordonnées aux fonctions comme en Pascal). Il n'y a pas d'ordre particulier pour les déclarations.

Les fonctions ont exactement la même syntaxe qu'en Pascal, mais les procédures se déclarent en utilisant le même mot clef `function`. Les variables dont déclarées à la Pascal, mais il n'y a aucune possibilité de passage par référence dans les fonctions. La déclaration de types se réduit à la déclaration de types réguliers (fondés sur des formes), les enregistrements et unions ne sont pas implémentés. Il y a aussi beaucoup de restrictions syntaxiques par rapport au Pascal, toujours dans un souci de simplification. La description grammaticale complète est donnée en appendice (cf. A).

3.2 Un langage pour les structures régulières

Nous allons décrire précisément la syntaxe et surtout la sémantique des constructions originales du langage.

3.2.1 Constructeurs de *formes* et déclaration de types

La syntaxe des constructeurs de formes est inspirée des notations mathématiques du chapitre précédent :

$S =< e_1, \ldots, e_d; r_1, \ldots, r_n >$	`shape` nom_de_forme `= [<lab_1, ..., lab_d> mot_1, ..., mot_n]`
$S = \lhd e_1, \ldots, e_d; r_1, \ldots, r_n \rhd$	`shape` nom_de_forme `= [|lab_1, ..., lab_d| mot_1, ..., mot_n]`
$S = \blacktriangleleft e_1, \ldots, e_d; r_1, \ldots, r_n \blacktriangleright$	`shape` nom_de_forme `= [[lab_1, ..., lab_d] mot_1, ..., mot_n]`

Les `mots` sont associés naturellement aux r_i dans lesquels on a substitué les occurrences des e_j par des `lab_j`.

On peut noter que s'il est possible de définir des formes paramétrées (dépendant de paramètres évalués dynamiquement, dans une procédure par exemple), il n'est pas possible de faire évoluer la forme une fois qu'elle est définie. Toutes les formes sont statiques. Il sera donc possible d'ajouter ou de retirer des nœuds à une structure (faire croître un arbre par exemple) mais pas de transformer une forme de liste circulaire à 14 éléments en liste circulaire à 17 éléments. Cette limitation peut être sévère dans certains cas, mais elle semble difficile à éviter dans la mesure où l'on cherche à effectuer des analyses statiques.

Les types de données réguliers

Les types de données réguliers sont déclarés très simplement :
`type` nom_de_type `=` nom_de_type_existant nom_de_forme

Par exemple, `type matrice_reelle = real matrice` définit un type `matrice_reelle` de la forme `matrice` et dont les nœuds sont de type `real`.

Les types prédéfinis sont les classiques `int`, `real`, et `bool`.

Syntaxe des mots d'accès aux structures

Pour indexer des structures régulières on peut utiliser des variables de type *index de structures* qui contiennent des mots d'accès, ou bien directement ces mots écrits dans une syntaxe particulière.

Les opérateurs sont la concaténation `.` et l'itération `^`. On peut ainsi écrire le mot

$$a.b.a^2.b^{-1}.(a.b.a)^{3(x+y)}$$

sous la forme

```
a.b.a^2.b^-1.(a.b.a)^3*(x+y)
```

où x et y sont des variables entières.

3.2.2 Déclaration des variables et structures

La déclaration de variables scalaires ou structurées est identique :

`var` nom_de_variable : nom_de_type

La déclaration de variables d'index est identique sauf que l'on les qualifie par un nom de forme et non de type :

`var` nom_de_variable : nom_de_forme

Dans le cas des variables structurées il n'y a bien évidement pas d'allocation de la structure en mémoire. Pour que cela soit possible statiquement il faut indiquer au compilateur l'étendue qu'elle peut prendre, dans le cas contraire (ou si cette étendue n'est pas bornée) elle sera gérée dynamiquement lors de l'exécution. On peut aussi chercher à spécifier plus finement (récursivement) le voisinage des nœuds dans une structure (cf. 3.6.2), ou à borner leur étendue.

On utilise pour cela le mot clef `construct` ; la syntaxe la plus simple est :

`construct` nom_de_variable `while` condition`{lab_1: recurse; ... lab_d: recurse;}`

Les `labels` sont des *directions* (des générateurs) de la forme $< \mathcal{E}, \Re >$ associée à nom_de_variable. Cette construction syntaxique déclare inductivement que les seuls nœuds accessibles sont ceux qui vérifient la condition du `while`, dont le mot d'accès appartient au langage $\mathcal{L} = \{\texttt{lab\_1}, \ldots, \texttt{lab\_}d\}^*$, et dont le parent dans ce langage $\mathcal{L}$ est déclaré accessible (cette condition n'est pas nécessaire pour ϵ).

Le mot clef `recurse` rappelle qu'il s'agit d'une déclaration inductive, mais il n'y a pas de véritable parcours récursif, c'est juste une déclaration qui pourra être ensuite utilisée pour vérifier la validité des accès (statiquement ou dynamiquement).

La notation `#`nom_de_label est une expression de type `int` dont la valeur est le nombre d'occurrences de nom_de_label dans le mot d'accès du nœud. Il est ainsi possible de borner la taille d'une structure. Attention : le nombre d'occurrences n'est pas calculé par rapport aux mots d'accès dans leur *forme normale* (cf. 2.4), mais par rapport au mot d'accès «brut» : c'est à dire au mot qui étiquette le nœud dans le langage $\mathcal{L}$. De plus, par convention, lorsque une direction `d` n'est pas bi-directionnelle (son inverse n'apparaît pas dans la définition de la forme et la syntaxe `[[...] ...]` n'est pas utilisée), la contrainte `#d>=0` sera supposée implicitement.

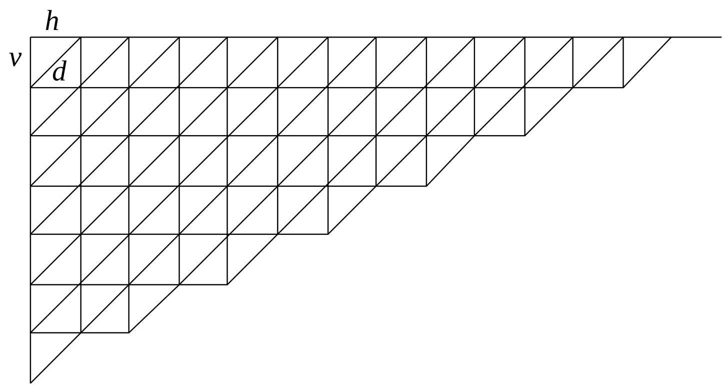
Exemple

Voici un exemple complet de déclaration et de construction d'une structure de forme «grille triangulaire» (cf. 2.2), en supposant que l'on ait défini la forme normale de cette grille de façon naturelle, c'est à dire en se ramenant aux «coordonnées cartésiennes» (cf. 2.4) :

```
shape grille_triangulaire = [[v, h, d] v.h.d^-1];
type grille_triangulaire_entiere = int grille_triangulaire;
var grille : grille_triangulaire_entiere;

construct grille while #h>=0 and #v>=0 and #h+2*#v<15 {
  h: recurse;
  v: recurse;
}
```

On déclare ainsi la structure `grille`, restreinte au domaine triangulaire suivant :



On notera que `d` n'apparaît pas dans la déclaration `construct`. Les bornes `#h>0 and #v>0` sont indispensables étant donné que la grille est bi-directionnelle : les déplacements h^{-1}, v^{-1} et d^{-1} sont autorisés.

Expression générale de la syntaxe `construct`

On peut étendre cette syntaxe en affinant les bornes à l'aide de gardes appliquées récursivement sur chaque *direction* :

label: `when` condition `recurse;`

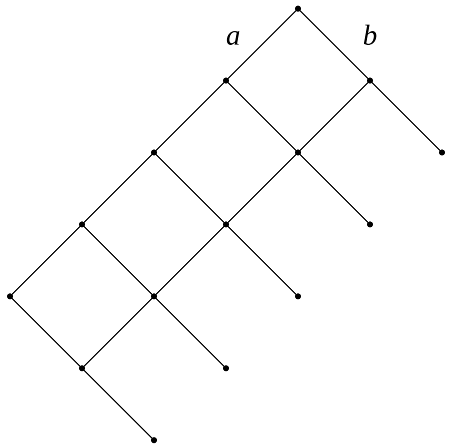
Cette fois, l'accessibilité est déclarée par induction (d'où le mot clef `recurse`). Pour un nœud donné déclaré valide, chaque voisin indiqué par un label dont la garde (éventuelle) est vérifiée sera *examiné*. L'examen d'un nœud consiste à vérifier pour celui-ci la validité de la condition du `while`.

L'exemple suivant donne une construction inductive, son analogue algorithmique et une figure représentative :

```
construct echelle while #a<4 and #b<3 {
  a: when #b<2 recurse;
  b: recurse;
}
```

```
fonction construit (a : entier, b : entier) {
    si a<=4 et b<=3 alors {
        si b<=2 alors construit (a+1, b);
        construit (a, b+1);
    }
}

{
    construit (1, 1);
}
```



Il est très important de noter que la garde introduite par `when` est vérifiée dans le contexte du nœud courant et non pas du nœud qui va être crée après la lecture du mot clef `recurse` (si la garde du while le permet). Le nombre d'occurrences du label sur lequel porte ce `when` est donc inférieur de 1 au nombre d'occurrences de ce même label dans la condition du `while` du fils. L'exemple suivant est plus lumineux :

```
construct liste while #u>=3 {
  u: when #u<14 recurse;
}
```

Définit une liste de 12 éléments indexés de 3 à 14 ! En effet `#u` vaut 13 lors de la dernière validation de la garde, et donc `#u` vaut 14 lors de la dernière validation d'un nœud, après récursion.

Bien entendu il ne s'agit que d'une déclaration inductive, aucune allocation effective ni aucun calcul récursif n'est effectué lors de la compilation. Par contre lors de la vérification à l'exécution (éventuellement remplacée par une analyse d'étendue statique pour améliorer les performances) une vérification itérative sera effectuée à partir de cette déclaration inductive lors de chaque accès à un nœud inexploré (cf. 3.5).

3.2.3 Structure de contrôle adaptée au parcours de formes

Les auteurs de [14] ont pris le parti de n'effectuer que des opérations globales sur leurs structures régulières. C'est à dire des opérations géométriques, et des définitions de «champs» de valeurs construits itérativement à partir de déclarations particulières.

Nous avons préféré une approche plus algorithmique pour rester dans le cadre traditionnel de l'analyse de flot, et pour des raisons d'efficacité (les langages déclaratifs sont moins efficaces dans pratiquement toutes les situations). De plus l'objet de cette étude est d'analyser des langages impératifs, et non pas des langages déclaratifs (où les dépendances sont écrites explicitement) ou fonctionnels purs (où les seules dépendances sont des relations source-puits).

Nous avons cependant introduit une construction syntaxique nouvelle qui exprime des algorithmes récursifs de façon plus «contrôlable». Celle-ci est introduite par une structure de la forme :

```
explore (label_1, ..., label_n) while condition {
   ...
}
```

L'idée de départ consiste à récupérer les avantages des boucles `for` ou `while` tout en permettant l'expression directe de parcours récursifs.

La syntaxe est très proche de celle du `construct`, sauf que le nom_de_variable est devenu une liste de `labels`, et que c'est ceux qui détermineront les instructions du bloc. L'autre différence est qu'il est possible de mélanger instructions classiques (assignations, appels de fonctions...) et instructions d'appel récursif (éventuellement gardées par `when`) avec un `label`. On peut imbriquer des `explore` et les tous les labels déclarés dans le nid d'`explore`s englobant sont accessibles (pour des indexations ou des dénombrements d'occurrences introduits par `#`)

L'ordre des instructions est fondamental, c'est lui qui détermine les priorités du parcours récursif (par exemple préfixe, infixe ou postfixe pour un arbre binaire).

On peut noter qu'il n'est pas possible d'exprimer directement un parcours en largeur d'un arbre avec la syntaxe `explore`. C'est normal puisqu'il ne s'agit pas véritablement d'une exploration récursive. Pour réaliser un tel parcours en respectant les voisinages, il faut changer de structure de données : utiliser un arbre ou les nœuds situés à une même profondeur sont reliés par une liste chaînée. On ne peut malheureusement pas exprimer une telle structure en LEGS, ses voisinages ne sont pas représentables par un graphe de Cayley ; à moins de ne représenter justement que ces listes chaînées de nœuds de même profondeur et de perdre les liens parentaux : c'est la représentation habituelle des arbres dans des tableaux où les fils d'un nœud i sont indexés par $2i$ et $2i+1$.

L'analogue algorithmique est similaire à celui du `construct` mis à part que le mot sur lequel sont calculées les occurrences des `labels` n'est pas un mot d'accès à une structure mais le mot étiquetant le sommet actuel dans l'arbre d'appel récursif, c'est à dire un mot de

$$\{\texttt{label\_1}, \ldots, \texttt{label\_n}\}^*$$

On utilisera souvent la dénomination *mot de contrôle* par opposition à *mot d'accès* lorsque l'on parlera du chemin dans l'arbre d'appel.

On donnera un exemple avec son analogue algorithmique dans la section suivante, et on verra que les boucles `for` ne sont en réalité que des cas particuliers d'utilisation de la syntaxe `explore`.

3.2.4 Indexation des structures

On utilise une syntaxe proche de celle des accès aux tableaux en Pascal :

structure[mot]

On peut bien entendu assigner des indexations comme de simples variables.

L'idée de remplacer une fonction récursive par une construction du type `explore` ne serait pas très intéressante si elle ne permettait pas de poursuivre plus avant l'analogie avec les boucles `for` : on aimerait bien indexer directement les structures à partir du chemin dans l'arbre d'appel. Celui-ci jouant le même rôle que les tuples d'entiers dans les nids de boucles.

Les substitutions constituent en fait la généralisation aux groupes des applications affines pour les espaces vectoriels (les $\mathbb{Z}$-modules en réalité puisque les indices de boucles sont entiers).

Au lieu d'écrire $[3i - j + 2, j]$ pour une matrice représentée par la forme $\blacktriangleleft$ *ligne*, *colonne* $\blacktriangleright$, explorée avec les labels `i` et `j`, on écrira :

$$\{\texttt{ligne}^3/\texttt{i}, \texttt{ligne}^{-1}.\texttt{colonne}/\texttt{j}\}.\texttt{ligne}^2$$

En effet, le résultat de cette substitution sur un mot de $\{i, j\}^*$ est :

$$(\texttt{ligne}^3)^{\#\texttt{i}}.(\texttt{ligne}^{-1}.\texttt{colonne})^{\#\texttt{j}}.\texttt{ligne}^2$$

Malgré la complexité syntaxique apparente, on prend vite l'habitude de raisonner en termes de substitutions. La puissance de celles-ci prend tout son sens pour indexer des formes non commutatives. Les parcours d'arbres peuvent ainsi s'écrire directement sans utiliser de variable d'index. On évite ainsi toute variable dans les accès aux structures, seul le l'arbre d'appel compte. C'est l'un des points forts de LEGS : exprimer statiquement des parcours récursifs.

Syntaxe des substitutions en LEGS

la syntaxe du langage est très proche de la notation mathématique :

$\{x/a, xy/b\}$ est écrit `{x/a,xy/b}`

On rappelle que les lettres n'apparaissant pas dans la substitution sont supposées implicitement substituées par ϵ (contrairement à l'habitude de les laisser invariantes qui n'a pas de sens dans ce cadre d'utilisation des substitutions).

Les substitutions LEGS ont le type «mot» et peuvent donc être concaténées avec d'autres mots, itérées, etc...

On peut également définir des constantes de type substitution afin d'alléger les notations :

`substitution` nom `to` nom_de_forme `=` mot_1`/lab_1`, ..., mot_n`/lab_n`

Les labels substitués (`lab_i`) doivent être définis dans le nid d'`explore`s englobant ; et le nom_de_forme est indispensable pour définir les substituants.

Il existe une syntaxe particulière pour tester l'existence d'un nœud dans une structure :

?nom_de_structure[mot]

Cette expression booléenne est vérifiée lorsque le nœud correspondant au mot d'accès est alloué dans la structure. L'utilisation de cette syntaxe rend à priori le contrôle dynamique, sauf si une analyse préalable (fondée sur les parcours qui précèdent) permet de circonscrire exactement l'étendue de la structure de données.

Exemple d'exploration, avec son analogue algorithmique

On poursuit l'exemple de l'échelle (ou du peigne plutôt...) (cf. 3.2.2).

```
explore (i, j) while #i<3 and #j<3 {
  j: recurse;
  echelle[{b/i,a/j}.b] := echelle[{a/i}];
  i: when #j<2 recurse;
}
```

(on considère ici qu'`echelle` est un tableau C).

fonction explore (i : entier, j : entier) {
    si i<=3 **et** j<=3 **alors** {
        construit (a, b+1);
        echelle[i+1][j] ← echelle[i][1];
        si b<=2 **alors** construit (a+1, b);
    }
}

{
    explore (1, 1);
}

3.3 Exemples complets

Un exemple très simple, l'échange récursif des valeurs des fils dans un arbre :

```
shape BT = [<l, r>];
type IBT = int BT;
var T : IBT;
var N : int;

{
  explore (left, right) while ?T[{l/left, r/right}]
                          and ?T[{r/left, l/right} {
    T[{l/left, r/right}] := T[{r/left, l/right}];
    left: recurse;
    right: recurse;
  }
}
```

Un exemple complexe d'utilisation de la syntaxe `explore` :

```
shape BT = [<l, r>];
type IBT = int BT;
var image : IBT;
```

```
shape A1D = [|u|];
type IA1D = int A1D;
var out : IA1D;

var image_0 : int;

function compute (x: int, y : int) : int {
  return x + y;
}

function haar (p : int) {
  var i : int;

  i := 0;
  explore (depth) while #depth < p {
    explore (left, right) while #left + #right < p - #depth {
      substitution sigma to BT = l/left, r/right;

      out [i] := compute (image [{sigma}.l], image [{sigma}.r]);
      i := i + 1;
      image [{sigma}] := 0;
      left: recurse;
      right: recurse;
    }
    out [i] := compute (image_0, image []);
    image_0 := 0;
    depth: recurse;
  }
}

{
  haar (4);
}
```

La fonction `haar` implémente un algorithme classique en traitement d'image (sans les calculs sur les scalaires). La version Pascal de cette fonction ressemblerait à :

```
for depth=1 to p begin
  mid := 2^p/2^depth; (* ^ abrège un calcul de puissance *)
  for l=0 to mid-1 begin
    out[mid+l] := compute (image[2*l], image[2*l+1]);
    image[l] := 0;
  end
end
out[0] := image[0];
```

La première boucle for est conservée en LEGS (transcrite dans la syntaxe `explore`), et la seconde implémente en réalité un parcours d'arbre en largeur. Mais le calcul est identique avec un parcours préfixe en profondeur.

3.4 Implémentation

Pour être en mesure de vérifier la validité des choix syntaxiques et sémantiques qui sous-tendent ce langage, une ébauche de compilateur à été écrit en *Caml light* ([21, 22]). Il pourra servir de base à un futur analyseur de flot, puis à un paralléliseur, un analyseur d'étendue, etc... (cf 4.5).

Le langage Caml à été retenu pour sa souplesse de langage fonctionnel, ses propriétés de typage fort et automatique qui simplifient grandement le développement et le débuggage, et enfin pour son mécanisme de programmation par cas gestion des *flots* (*streams*) particulièrement adapté à l'écriture de compilateurs. Un passage à Objective Caml (la version actuellement la plus performante du dialecte de ML, introduisant une extension orientée objet et une structure plus modulaire) est probable pour l'écriture d'un prototype d'analyseur.

3.5 Aspects théoriques et pratiques, compilation de LEGS

Dans cette section nous évoquons quelques problèmes de compilation posés par les structures de données LEGS ainsi que leur résolution.

3.5.1 Nécessité d'une analyse d'étendue

Pour que les performances à l'exécution de LEGS soient correctes il est impératif de récupérer le maximum d'informations sur l'étendue des structures de données. C'est à dire que pour chaque structure de contrôle `construct` ou `explore`, les accès aux structures doivent être analysés et une approximation de leur localisation doit être calculée.

Dans le cas où une telle analyse ne donne pas de résultats intéressants (accès trop complexes), le compilateur devra se contenter d'une gestion dynamique de la mémoire, beaucoup plus simple mais généralement moins efficace.

Tous les langages proposant des structures de données allouées automatiquement sont confrontés à ce problème et de nombreuses solutions existent.

3.5.2 Choix de la représentation des données

L'analyse d'étendue est susceptible d'apporter suffisamment de précisions sur les accès à une structure pour permettre de représenter celle-ci statiquement à l'aide d'un tableau. On gagnera ainsi énormément en performances, surtout lorsque les accès à cette structure sont dispersés (c'est rarement le cas pour les algorithmes exprimés en LEGS qui utilisent les substitutions et la structure de contrôle `explore`).

Dans le cadre par exemple de calculs d'algèbre linéaire, on considère généralement des matrices de taille constante. Il est alors très simple de spécifier cette propriété à l'aide de la déclaration `construct`:

```
construct matrix while #ligne>0 and #ligne<=N and #colonne>0 and #colonne<=M {
  ligne: recurse;
  colonne: recurse;
}
```

L'effet en termes d'information transmise au compilateur est comparable à la déclaration C `int matrix[N][M]`.

3.5.3 Vérifications dynamiques

Si l'analyse statique ne suffit pas, quelques opérations doivent être reportées à l'exécution :

1. L'allocation de la mémoire étant complètement automatique, les parties du code opérant sur des structures représentées par des pointeurs doivent vérifier l'existence des nœuds et le cas échéant les construire.

2. Les structures déclarées à l'aide du mot clef `construct` nécessitent une vérification de la validité des accès. Celle-ci peut être réalisée en ajoutant un drapeau de validité à la représentation interne des nœuds, et dès qu'un nouveau nœud est alloué la vérification est effectuée (itérativement à partir du nœud existant le plus proche) et le drapeau peut être placé.

3.6 Comparaison avec d'autres approches

3.6.1 Langages orientés objet ou proposant des abstractions de données

Le problème de l'abstraction des structures de données n'est pas nouveau, et depuis les premiers langages qui proposent une gestion directe des structures de données complexes ([23]), comme les langages orientés objet par exemple, des progrès considérables ont été effectués. Mais abstraire l'implémentation des types ne suffit pas à rendre ceux-ci analysables statiquement. Encore faut il que le compilateur puisse en tirer quelque-chose, à savoir une connaissance des voisinages et des interconnections dans les structures. Les types récursifs ML permettent cela, ainsi que les *formes* utilisées ici ; c'est la régularité qui importe pour déterminer statiquement les alias, le typage ne suffit pas a résoudre les ambiguïtés de référence.

Cependant les types abstraits ont en commun avec les types réguliers LEGS de proposer des fonctions de création et de manipulation spécifiques. Cette abstraction de l'implémentation est indispensable pour réaliser une analyse de flot de données.

3.6.2 Types récursifs, structures de données récursives

Le modèle de régularité que nous proposons n'est malheureusement pas parfaitement adapté à la description des types récursifs. En effet s'il permet de caractériser des arbres binaires, ternaires, sans problème il n'est pas possible d'exprimer directement des propriétés classiques des arbres infinis rationnels comme l'alternance du degré des nœuds Dans l'exemple de la figure 1.1, page 8, on peut définir une forme d'arbre ternaire et ensuite utiliser une construction récursive d'un arbre de degré alternativement 2 et 3 respectant cette forme ; mais comme on le verra dans le chapitre suivant, l'analyse de flot devient alors beaucoup plus difficile.

```
shape TT = [<l, d, r>];
type ITT = int TT;
var tree : ITT;

construct tree {
  l: recurse;
  d: when (#l+#d+#r)%2 = 1 recurse;
  r: recurse;
}
```

3.6.3 Critique du modèle fondé sur les graphes de Cayley

On vient de voir que ce modèle n'est pas assez expressif pour pour décrire parfaitement toutes les structures sur lesquelles on souhaite travailler, mais à l'inverse il semble aussi beaucoup trop général. Certaines formes semblent n'avoir aucune application pratique, notamment la plupart des formes non commutatives qui ne sont pas libres (elles sont d'ailleurs difficile à manipuler, n'ayant pas de forme normale en général).

En réalité les deux faits principaux qui incitent à utiliser ce modèle de régularité sont : d'une part le regroupement sous un même formalisme des structures arborescentes, toriques et vectorielles ; et d'autre part la prise en compte directe de la notion de voisinage et de chemin d'accès à l'aide d'une loi algorithmique simple, la composition. Il s'agit par conséquent d'un cadre de travail intéressant qu'il faudra néanmoins affiner dans le futur.

Chapitre 4

Extension de l'analyse de flot aux structures régulières

Avant d'introduire l'analyse de flot de données pour structures régulières (*Group-shaped Structures Dataflow Analysis*, GSDA) nous commençons par un rappel des techniques employées par l'ADA.

4.1 ADA, l'approche programmation entière

Le but de cette section est de présenter succinctement l'analyse de flot de données pour les algorithmes sur des tableaux, c'est à dire l'ADA. On considère donc des nids de boucles et des accès à des tableaux. Une exposition détaillée peut être trouvée dans [10].

4.1.1 Notations

Si $\vec{x}$ est un vecteur entier, on note $\vec{x}[k]$ sa k^{e} composante. $\vec{x}[k..l]$ représente la restriction du vecteur $\vec{x}$ aux composantes k à l. On notera $\ll$ l'ordre lexicographique sur les vecteurs. Une instruction S peut avoir plusieurs instances dans le déroulement d'une boucle ; si $\vec{x}$ est le vecteur d'itération[1] de S, l'instance correspondante est étiquetée par $\langle S, \vec{x} \rangle$. La *profondeur* d'une instance est le nombre de boucles englobantes, donc le compteur d'une boucle à la profondeur p est la $p+1^{\text{e}}$ composante du vecteur d'itération. On note enfin $\lhd$ l'ordre local défini par : $S \lhd R$ si et seulement si S précède R dans le texte du programme. Enfin, on appellera *opération* une instance d'instruction.

L'ordre d'exécution séquentielle, noté $\prec$ est défini par :

$$\begin{aligned} \langle S, \vec{x} \rangle \prec \langle R, \vec{x} \rangle \quad \equiv \quad & \vec{x}[1..N_{SR}] \ll \vec{y}[1..N_{SR}] \vee \\ & \big(\vec{x}[1..N_{SR}] = \vec{y}[1..N_{SR}] \wedge (S \lhd R)\big) \end{aligned} \tag{4.1}$$

Où N_{SR} est le nombre de boucles qui englobent simultanément S et R. Cet ordre séquentiel peut être développé en fonction de la profondeur :

$$\langle S, \vec{x} \rangle \prec \langle R, \vec{y} \rangle \equiv \bigvee_{p=0}^{N_{SR}} \langle S, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle \tag{4.2}$$

1. Le vecteur formé par les compteurs des boucles englobantes

Avec pour $p \in \{0, \ldots, N_{RS} - 1\}$:

$$\langle S, \vec{x} \rangle \prec_p \langle R, \vec{y} \rangle \equiv (\vec{x}[1..p] = \vec{y}[1..p]) \wedge (\vec{x}[p+1] < \vec{y}[p+1])$$

Et :

$$\langle S, \vec{x} \rangle \prec_{N_{SR}} \langle S, \vec{y} \rangle \equiv (\vec{x}[1..N_{SR}] = \vec{y}[1..N_{SR}]) \wedge (S \lhd R)$$

Dans cette section, max (respectivement min) représente le maximum (réciproquement minimum) soit lexicographiqe ($\ll$) soit séquentiel ($\prec$).

4.1.2 Modèle de programme

Le langage analysé peut être vu comme un Pascal simplifié où du Fortran «rationalisé». En réalité c'est surtout l'aspect statique qui importe, c'est pourquoi des hypothèses supplémentaires définissent un sous-ensemble de ce langage de base :

- Les données sont des scalaires et des tableaux multi-dimensionnels de scalaires ;
- Les seules instructions élémentaires considérées sont l'assignation de scalaires et d'éléments de tableaux ;
- La seule structure de contrôle est la boucle `for` ; on autorisera néanmoins des expressions conditionnelles ne faisant pas apparaître d'éléments de tableaux, celles-ci peuvent être nécessaires pour le calcul proprement dit ;
- On supposera qu'aucune assignation des compteurs de boucles, ni d'indexations illégales (en dehors des bornes) des tableaux ne sont effectuées.

On supposera que le programme est à contrôle statique. Dans le cadre de l'ADA, ça veut dire qu'en plus des restrictions imposées précédemment, les seuls paramètres pouvant apparaître dans les bornes des compteurs de boucles sont des paramètres de structure (définis en dehors de la partie de programme considérée), des constantes, et les compteurs des boucles englobantes. On introduira une définition plus générale mais analogue pour la GSDA. On appellera $l_p(\vec{x}[1..p])$ et $u_p(\vec{x}[1..p])$ les bornes de $\vec{x}[p+1]$, et on notera $D(S)$ l'ensemble des vecteurs d'itérations $\vec{x}$ pour lesquels l'opération $\langle S, \vec{x} \rangle$ est définie.

4.1.3 Caractérisation de la source

Le problème de l'analyse de flot consiste à retrouver la source S d'une opération effectuant un accès en lecture R. On note $\sigma(\langle R, \vec{y} \rangle)$ la source de l'opération $\langle R, \vec{y} \rangle$, c'est dire de l'instance de l'instruction R pour le vecteur d'itération $\vec{y}$. Une source S candidate pour un accès dans l'instruction R est de la forme :

```
S: a(f(x)) := ...
...
R: ... := ... a(g(y))
```

Trois conditions doivent être vérifiées pour que $\langle S, \vec{x} \rangle$ puisse être une source potentielle (c'est à dire à l'origine d'une dépendance de flot vers $\langle R, \vec{y} \rangle$:

Prédicat d'existence : $\langle S, \vec{x} \rangle$ est une opération valide :

$$\vec{x} \in \mathbf{D}(S) \tag{4.3}$$

Prédicat de séquencement : $\langle S, \vec{x}\rangle$ est exécuté avant $\langle R, \vec{y}\rangle$:

$$\langle S, \vec{x}\rangle \prec \langle R, \vec{y}\rangle \tag{4.4}$$

Équation de Conflit : Les deux opérations accèdent la même cellule mémoire :

$$f(\vec{x}) = g(\vec{y}) \tag{4.5}$$

On ajoute aussi une contrainte «d'environnement» qui garantit que le puits $\langle R, \vec{y}\rangle$ est une opération valide :

$$\vec{y} \in \mathbf{D}(R)$$

L'ensemble des sources candidates est alors

$$\mathbf{Q}_{SR}(\vec{y}) = \{\vec{x} \mid \vec{x} \in \mathbf{D}(S), f(\vec{x}) = g(\vec{y}), \langle S, \vec{x}\rangle \prec \langle S, \vec{y}\rangle\}$$

On peut le le développer en fonction de la profondeur :

$$\mathbf{Q}^p_{SR}(\vec{y}) = \{\vec{x} \mid \vec{x} \in \mathbf{D}(S), f(\vec{x}) = g(\vec{y}), \langle S, \vec{x}\rangle \prec_p \langle S, \vec{y}\rangle\} \tag{4.6}$$

4.1.4 Détermination de la source

On suppose à présent que les bornes des boucles sont des fonctions affines ainsi que les fonctions d'accès aux tableaux (f, g...). Comme chaque relation $\prec_p$ est alors affine, $\mathbf{Q}^p_{SR}$ est un polyèdre.

On appelle dépendance *directe* de S sur R toute instance de S maximale pour l'ordre séquentiel. C'est à dire ici le maximum de $\mathbf{Q}^p_{SR}$ pour l'ordre lexicographique, puisque celui-ci est total (en cas d'analyse floue, il peut y avoir plusieurs dépendances directes possibles, le choix du maximum n'est pas réalisable statiquement).

$$\mathbf{K}^p_{SR}(\vec{y}) = \max \mathbf{Q}^p_{SR}(\vec{y}) \tag{4.7}$$

La puissance de l'ADA consiste à exprimer ce maximum en termes de programmation entière. Le programme PIP ([13]) développé à partir de l'algorithme de Programmation Entière Paramétrique de Paul Feautrier permet de calculer $\mathbf{K}^p_{SR}$, et l'opération correspondante est notée

$$\mathcal{S}^p_{SR}(\vec{y}) = \langle S, \mathbf{K}^p_{SR}(\vec{y})\rangle$$

PIP est exponentiel en le nombre de contraintes affines qui définissent le polyèdre convexe, mais en général ce nombre étant du même ordre que le nombre d'imbrications de boucles (rarement supérieur à 4) la complexité reste raisonnable.

En général, il y a plusieurs instructions $S_1, \ldots, S_m$ qui écrivent dans le même tableau, et donc autant de sources candidates. On peut à présent écrire la source de l'opération :

$$\sigma(\langle R, \vec{y}\rangle) = \max \{\mathcal{S}^p_{S_k R}(\vec{y}) \mid 1 \leqslant k \leqslant m, 0 \leqslant p \leqslant N_{S_k R}\} \tag{4.8}$$

Ce maximum est calculé à l'aide de règles de simplification ([10]), et le résultat est un *arbre de sélection quasi-affine* (*Quasi-Affine Selection Tree*, quast), c'est à dire une expression conditionnelle à plusieurs niveaux telle que :

- Les prédicats sont des tests sur le signe d'expressions affines ;

– Les «feuilles» sont soit des opérations dont les vecteurs d'itération sont affines, soit $\bot$ (une opération exécutée avant la portion de code considérée).

Nous essaierons autant que possible de conserver des notations proches de celles-ci dans la suite de ce chapitre.

4.2 Exemple introductif

L'exemple suivant, bien que simple en apparence, mérite une attention particulière car il introduit les problèmes de façon simple et laisse entrevoir des développements prometteurs.

```
shape A1D = [[u]];
type IA1D = int A1D;
var A : IA1D;
var N : int;

{
  explore (lft, rgt) while #lft + #rgt < N {
    lft: recurse;
    A[{u/rgt}] := A[{u/rgt}.u^-1];
    rgt: recurse;
  }
}
```

Sans entrer dans les détails, on peut reprendre les prédicats de l'ADA et les écrire pour cet exemple. On notera S l'unique instruction d'affectation ; l'ordre local donne $\mathtt{lft} \lhd S \lhd \mathtt{rgt}$ (parcours récursif infixe).

Si W est le mot de contrôle étiquetant l'opération $\langle S, W\rangle$ on note $\#\mathtt{rgt}(W)$ et $\#\mathtt{lft}(W)$ le nombre d'occurrences des labels. Le mot de contrôle du puits sera noté W et celui des sources potentielles W'.

L'existence s'écrit

$$\#\mathtt{lft}(W') \geqslant 0 \wedge \#\mathtt{rgt}(W') \geqslant 0 \wedge \#\mathtt{lft}(W') + \#\mathtt{rgt}(W') < N \tag{4.9}$$

Le séquencement : $W'.S \ll W.S$ (cf. 4.3.2 pour les détails de la définition du prédicat de séquencement). C'est à dire $W.S = U.X.V$, $W'.S = U.X'.V'$, U, V et V' sont des mots, X et X' peuvent prendre les valeurs $\mathtt{lft}$, $\mathtt{rgt}$ ou S et vérifient $X' \lhd X$. Lorsque $X = S$, $V = \epsilon$, de même pour X' et V'. Et lorsque $X = \mathtt{lft}$ ou $\mathtt{rgt}$, $V \in \{\mathtt{lft}, \mathtt{rgt}\}^*.S$ (la dernière lettre de V est S).

L'équation de conflit s'écrit

$$\#\mathtt{rgt}(W') = \#\mathtt{rgt}(W) - 1 \tag{4.10}$$

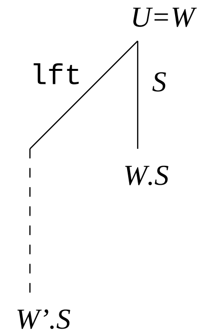
car seul le nombre d'occurrences de $\mathtt{rgt}$ importe.

Il y a une différence majeure avec les nids de boucles : le prédicat de séquencement est défini par une relation d'ordre lexicographique sur des mots et non pas des vecteurs. Il en résulte une profondeur non bornée (à part par le prédicat d'existence) de la différenciation entre W et W'. On ne peut donc pas réduire la résolution de ces trois contraintes à un problème de programmation entière.

On va voir qu'il est néanmoins possible d'écrire l'ensemble des source potentielles, et même de trouver la source, sans que le résultat soit aussi simple qu'un quast malheureusement.

On peut distinguer 3 cas, suivant que $X' = \mathtt{lft}$ et $X = S$ (a), $X' = \mathtt{lft}$ et $X = \mathtt{rgt}$ (b), et $X' = S$ et $X = \mathtt{rgt}$ (c).

(a)



On a $W.S = U.X$, donc $W = U$, et donc le prédicat de séquencement implique que

$$\begin{aligned} W'.S &= U.\mathtt{lft}.V' \\ &= W.\mathtt{lft}.V' \end{aligned}$$

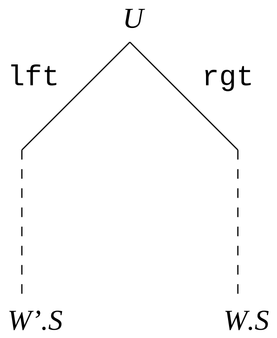
(la dernière lettre de V' est S)
En «projetant» sur la direction $\mathtt{rgt}$:

$$\begin{aligned} \#\mathtt{rgt}(W') &= \#\mathtt{rgt}(W) \\ &\quad +\#\mathtt{rgt}(V') \end{aligned}$$

En reportant cette condition dans l'équation 4.10 :

$$\#\mathtt{rgt}(V') = -1$$

(b)



On a $W.S = U.\mathtt{rgt}.V$, donc le prédicat de séquencement implique que

$$\begin{aligned} W'.S &= U.\mathtt{lft}.V' \\ &= W.S.(\mathtt{rgt}.V)^{-1}.\mathtt{lft}.V' \end{aligned}$$

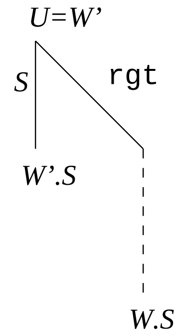
(la dernière lettre de V' est S)
En «projetant» sur la direction $\mathtt{rgt}$:

$$\begin{aligned} \#\mathtt{rgt}(W') &= \#\mathtt{rgt}(W) \\ &\quad -\#\mathtt{rgt}(V) \\ &\quad -1 + \#\mathtt{rgt}(V') \end{aligned}$$

En reportant cette condition dans l'équation 4.10 :

$$\#\mathtt{rgt}(V') = \#\mathtt{rgt}(V)$$

(c)



On a $W.S = U.\mathtt{rgt}.V$, donc le prédicat de séquencement implique que

$$\begin{aligned} W'.S &= W.S.(\mathtt{rgt}.V)^{-1}.S \\ &= U.S \end{aligned}$$

Et donc $W' = U$. (la dernière lettre de V est S).
En «projetant» sur la direction $\mathtt{rgt}$:

$$\begin{aligned} \#\mathtt{rgt}(W') &= \#\mathtt{rgt}(W) \\ &\quad -\#\mathtt{rgt}(V) - 1 \end{aligned}$$

En reportant cette condition dans l'équation 4.10 :

$$\#\mathtt{rgt}(V) = 0$$

Dans le cas (a), on ne peut avoir un nombre d'occurrences négatif dans un mot de contrôle, il n'y a donc pas de source potentielle de ce type :

$$\mathbf{Q}_{(a)}(W) = \emptyset$$

Dans le cas (c), W doit être de la forme $U.\mathtt{rgt}.\mathtt{lft}^n, n \geqslant 0$ pour qu'il y ait des sources potentielles. Si c'est le cas l'existence de $\langle S, W' \rangle$ est bien vérifiée, car W' est un ancêtre de W dans l'arbre d'appel. L'ensemble des sources potentielles de ce type est donc un singleton :

$$\mathbf{Q}_{(c)}(W) = \{U\} = \{W.(\mathtt{rgt}.\mathtt{lft}^n)^{-1}\}$$

Où $\mathtt{lft}^n$ est le plus grand suffixe constitué de $\mathtt{lft}$ dans W ($n \geqslant 0$).

Dans le cas (b), la condition obtenue donne une infinité de solutions, mais seul un nombre fini vérifie le prédicat d'existence. Pour exprimer ces solutions, on va devoir introduire un nouveau type de langages formels : ce que nous appelons *langages à occurrences comptées* (*Numbered Occurrences Languages*, NOL). Nous ouvrons donc une parenthèse dans l'analyse de cet exemple.

Définition 8 *Le* pompage *d'un alphabet* $\Sigma = \{e_1, \dots, e_d\}$ *par une application de* Σ *dans* $\mathbb{Z}$ *qui à* e_j *associe* i_j*, noté* $[e_1^{i_1}, \dots, e_d^{i_d}]$*, est le langage formé de tous les mots dont le nombre d'occurrences de chaque* e_j *est exactement* i_j*.*

Par exemple :

$$[a^2, b^3] = \{aabbb, ababb, abbab, abbba, baabb, babab, babba, bbaab, bbaba, bbbaa\}$$

$$a.[a^2, b^2].b = \{aaabbb, aababb, aabbab, ababab, abbaab\}$$

On utilisera par la suite des pompages *paramétrés* par des indices variables (mais toujours positifs ou nuls). On définit aussi une syntaxe particulière : si M est un mot et P un pompage, $M.P^{-1}$ est l'ensemble des préfixes de M tels que le suffixe correspondant soit un mot de P. Cet ensemble est soit vide, soit réduit à un singleton. Par exemple :

$$abbabba.[a^2, b^3]^{-1} = \{ab\}, abbabba.[a^2, b^1]^{-1} = \emptyset$$

Définition 9 *On considère une liste de mots (éventuellement vides) $M_1, \dots, M_p$ sur un alphabet Σ, un ensemble d'*indices *$\mathcal{I}$, une liste de pompages de Σ et d'inverses de pompages $P_1, \dots, P_{p-1}$ paramétrés par $\mathcal{I}$, et un ensemble $\mathcal{A}$ d'inégalités ou d'égalités affines.*

Le langage

$$\mathcal{L} = \bigcup_{\mathcal{I} \text{ vérifiant } \mathcal{A}} M_1.P_1.\cdots.M_{p-1}.P_{p-1}.M_p$$

$$\textit{noté } \mathcal{L} = \frac{M_1.P_1.\cdots.M_{p-1}.P_{p-1}.M_p}{\mathcal{A}}$$

est un langage à occurrences comptées (NOL). De plus seuls les langages définis ainsi sont des NOLs.

Par exemple :

$$\frac{[a^i, b^i].c}{i \geqslant 1} = \{abc, bac, aabbc, ababc, abbac, baabc, babac, bbaac, aaabbbc, \dots\}$$

$$\frac{abbabba.[a^2, b^j]^{-1}.[a^j]}{j \leqslant 3} = \{abb\underbrace{aa}_{j=2}, ab\underbrace{aaa}_{j=3}\}$$

Lorsque l'ensemble $\mathcal{A}$ est vide, on simplifie la notation en écrivant simplement

$$\mathcal{L} = M_1.P_1.\cdots.M_{p-1}.P_{p-1}.M_p$$

Attention, cela ne veut pas dire qu'il n'y a pas de contraintes sur les occurrences : lorsque deux lettres ont le même indice, leur nombre d'occurrences doit bien entendu être le même :

$$bbbabbabba.[a^2, b^j]^{-1}.[a^j] = \{bbbabb\underbrace{aa}_{j=2}, bbbab\underbrace{aaa}_{j=3}, bbba\underbrace{aaaa}_{j=4}\}$$

Une étude théorique de ces langages (propriétés, reconnaissance par automates, grammaires attribuées...) serait sans doute très intéressante, notament pour résoudre le problème qui va nous importer ici : calculer le maximum lexicographique dans un NOL. Mais on verra a la section 4.4 que la programmation entière nous rend déjà de précieux services.

Nous pouvons à présent reprendre la description de l'ensemble des sources de type (b). Si l'on ne prend pas en compte le prédicat d'existence, l'équation de conflit et le prédicat de séquencement nous donnent la contrainte suivante :

$$W' \in W.([\mathtt{rgt}^i, \mathtt{lft}^j].\mathtt{rgt})^{-1}.\mathtt{lft}.[\mathtt{rgt}^i, \mathtt{lft}^k]$$

W vérifie le prédicat d'existence, donc

$$\#\mathtt{lft}(W) + \#\mathtt{rgt}(W) < N$$

Et comme W' vérifie aussi ce prédicat, on a $\#\mathtt{lft}(W) + \#\mathtt{rgt}(W) - i - j - 1 + 1 + i + k < N$, donc

$$\mathbf{Q}_{(b)}(W) = \frac{W.(\mathtt{rgt}.[\mathtt{rgt}^i, \mathtt{lft}^j])^{-1}.\mathtt{lft}.[\mathtt{rgt}^i, \mathtt{lft}^k]}{k - j < N - \#\mathtt{lft}(W) - \#\mathtt{rgt}(W)}$$

On peut remarquer que lorsque $\mathbf{Q}_{(c)}(W)$ est non vide, la solution W' qu'il contient est supérieure lexicoraphiquement à tous les éléments de $\mathbf{Q}_{(b)}(W)$.

Le calcul de $\mathbf{K}(W)$ consiste à calculer le maximum lexicographique $\mathbf{K}_{(i)}(W)$ de chaque $\mathbf{Q}_{(i)}(W)$, puis à prendre de nouveau le maximum des $\mathbf{K}_{(i)}(W)$. La solution est visiblement calculable pour chaque W mais on ne voit pas très bien comment exprimer simplement la solution générale. Deux classes de programmes seront étudiées dans la suite de ce chapitre, où il est possible de calculer la forme générale de $\mathbf{K}(W)$.

Dans cet exemple nous pouvons calculer des dépendances directes «potentielles» :

$$\mathbf{K}_{(a)}(W) = \bot$$

$$\mathbf{K}_{(c)}(W) = \left(\mathtt{if}\ \ W = U.\mathtt{rgt}.\mathtt{lft}^n\ \ \mathtt{then}\ \ U\ \ \mathtt{else}\ \ \bot\right)$$

Mais pour $\mathbf{K}_{(b)}(W)$ c'est plus difficile, on donnera une solution à la section 4.4.1.

4.3 Extension aux structures régulières, GSDA

L'ADA peut s'adapter simplement à des programmes LEGS opérant sur des tableaux avec des nids d'`explore` «linéaires», c'est à dire avec un seul label chacun ; puisque ceux-ci sont équivalents à des boucles `for`. On propose cette fois d'étudier les explorations récursives de structures régulières.

4.3.1 Modèle de programme

On se restreint à un seul niveau d'imbrication d'`explore`, et on suppose que les appels récursifs ne sont pas gardés (par `when`). Ce n'est donc pas une généralisation des nids de boucles (remarquons tout de même qu'un seul explore avec des gardes bien choisies peut simuler n'importe quel nid de boucles) ; cependant cette limitation est surtout présente pour des raisons simplificatrices et l'extension à des nids d'`explore` ne devrait pas poser beaucoup de problèmes. La structure de contrôle est donc celle d'un parcours récursif. Les labels LABELS $= \{C_1, \ldots, C_n\}$ ne sont pas gardés (pas de `when`), on isole deux instructions qui accèdent la même structure de données `A`, R en lecture et S en écriture. Cette structure a une forme quasi-libre.

```
shape SHAPE = [<e_1,...,e_d> r_1,...,r_m];
type TYPE = ... SHAPE;
var A : TYPE;

explore (C_1,...,C_n) while polyèdre_convexe {
  C_1 : recurse;
```

```
  ...
  A[σ_S.γ_S] := ...;              (* S *)
  ...
  ... := ... A[σ_R.δ_R] ...;      (* R *)
  ...
  C_n : recurse;
}
```

γ_S et δ_R sont des mots sur $\mathcal{E} = \{e_1, \ldots, e_d\}$ et σ_S, σ_R des substitutions de LABELS sur $\mathcal{E}$:

$$\sigma_S = \{\sigma_S(C_i)/C_i\} \ \texttt{et} \ \sigma_R = \{\sigma_R(C_i)/C_i\}$$

L'ordre local de ce pseudo-code est :

$$C_1 \lhd \cdots \lhd S \lhd \cdots \lhd R \lhd \cdots \lhd C_n$$

Le polyèdre_convexe est une condition booléenne formée par la conjonction d'inégalités et d'égalités affines sur les occurrences des C_i (et les paramètres de structure).

On le représente par une matrice à n colonnes A et un vecteur de dimension n, B, qui expriment la condition booléenne par la formule

$$AM(W) \leqslant B \tag{4.11}$$

Où $M(W)$ est le vecteur $(\#C_1(W), \ldots \#C_n(W))$ et W est le mot de contrôle sur LABELS* étiquetant le chemin dans l'arbre d'appel. L'ordre $\leqslant$ est l'ordre «produit» sur $\mathbb{Z}^n$ (un vecteur est inférieur à un autre si et seulement si toutes ses composantes le sont).

4.3.2 Caractérisation de la source

On reprend les trois conditions de l'ADA qui caractérisent le fait que $\langle S, W'\rangle$ soit une source potentielle de $\langle R, W\rangle$, et on les écrit dans ce cadre nouveau.

Prédicat d'existence : Il s'écrit simplement

$$AM(W') \leqslant B \tag{4.12}$$

On remarquera que si $\langle S, W'\rangle$ existe et que W'' est un préfixe de W', alors $\langle S, W''\rangle$ existe aussi. C'est la base du parcours récursif.

Prédicat de séquencement : C'est lui qui pose le plus de problèmes, comme on l'a vu dans l'exemple introductif

$$\begin{aligned} \langle S, W'\rangle \prec \langle R, W\rangle \quad &\Longleftrightarrow \quad W'.S \ll W.R \qquad (4.13)\\ &\Longleftrightarrow \quad \exists U \in \text{LABELS}^*,\\ &\qquad \big(\exists X \in \text{LABELS et } \exists V \in \text{LABELS}^*.R\big) \text{ ou } \big(X = R \text{ et } V = \epsilon\big),\\ &\qquad \big(\exists X' \in \text{LABELS et } \exists V' \in \text{LABELS}^*.S\big) \text{ ou } \big(X = S \text{ et } V' = \epsilon\big),\\ &\qquad W.R = U.X.V,\ W'.S = U.X'.V', \quad \text{et } X' \lhd X \end{aligned}$$

Équation de conflit : $\langle S, W'\rangle$ et $\langle R, W\rangle$ accèdent la même cellule mémoire si et seulement si

$$\sigma_S(W').\gamma_S = \sigma_R(W).\delta_R \tag{4.14}$$

L'expression de l'équation de conflit est équivalente au problème du mot. Mais nous avons supposé à la section 2.4 que toutes les formes considérées seraient caractérisées par l'existence de formes normales, canoniques ou non. Pourtant si le problème est décidable dans ces conditions il n'en reste pas moins difficile... Au cours de ce stage de DEA nous avons limité notre étude aux tableaux et aux arbres. Mais les résultats obtenus permettent envisager une extension relativement aisée à des formes plus complexes (notamment les formes associées à des groupes dont la représentation ne met en relation que des éléments égaux à une commutation près).

4.4 Calcul de la source

Pour les tableaux et les arbres, tous les mots d'accès à un même nœud (ils représentent tous le même élément du groupe) ont la même longueur. On note $|g|$ la longueur de g. On notera aussi $W[x..y]$ le mot constitué de la x^{e} à la y^{e} lettre de W, par analogie avec les vecteurs.

Enfin on pose $p = |W|$, $g = \sigma_R(W)$ et $g' = \sigma_S(W')$, et $\ell = |\delta_R| - |\gamma_S|$.

On peut d'ores et déjà remarquer que si $g.\delta_R.\gamma_S^{-1}$ n'est pas un mot sur $\mathcal{E}^*$, l'ensemble des sources potentielles est vide. Supposons que ce n'est pas le cas, on pose $\mathbf{C}_{SR}(W)$ l'ensemble des mots W' tels que $g' = g.\delta_R.\gamma_S^{-1}$. On a alors

$$\mathbf{Q}_{SR}(W) = \big\{W' \in \mathbf{C}_{SR}(W) \mid \langle S, W'\rangle \prec \langle R, W\rangle \wedge AM(W') \leqslant B\big\} \tag{4.15}$$

Pour des formes complexes le calcul de $\mathbf{Q}_{SR}$ ne peut probablement se faire qu'à l'aide d'outils performants, comme des systèmes de réécriture, et de calcul sur les groupes. Les groupes automatiques ([9]) y jouent certainement un rôle fondamental. Mais pour les tableaux et les arbres la tâche est plus aisée.

4.4.1 Cas des tableaux

Recherche des sources potentielles, Q

L'expression de $\mathbf{C}_{SR}(W)$ est assez simple dans ce cas, c'est l'ensemble des solutions d'un système d'équations affines. En projetant g (ou g') sur les générateurs e_j (les mots associés aux éléments de groupes sont supposés être la représentation de ceux-ci dans leur forme normale) :

$$\#e_j(g) = \sum_{k=1}^{p} \#e_j(\sigma_R(W[k])) = \sum_{i=1}^{n} \#e_j(\sigma_R(C_i))\#C_i(W)$$

Le résultat étant analogue pour g', l'équation de conflit s'écrit :

$$\forall 1 \leqslant j \leqslant d, \sum_{i=1}^{n} \#e_j\big(\sigma_S(C_i)\big)\#C_i(W') + \#e_j(\gamma_S) = \sum_{i=1}^{n} \#e_j\big(\sigma_R(C_i)\big)\#C_i(W) + \#e_j(\delta_R) \tag{4.16}$$

Donc :

$$\mathbf{C}_{SR}(W) = \frac{[C_1^{c_1}, \ldots, C_n^{c_n}]}{\forall 1 \leqslant j \leqslant d, \sum_{i=1}^{n} \#e_j\big(\sigma_S(C_i)\big)c_i + \#e_j(\gamma_S) = \sum_{i=1}^{n} \#e_j\big(\sigma_R(C_i)\big)\#C_i(W) + \#e_j(\delta_R)}$$

On cherche W' sous la forme $W[1..q].X'.V'$. Pour que cela ait un sens on doit avoir $q < \min(p, p+\ell)$, et pour que le prédicat de séquencement soit vérifié $X' \lhd (W.R)[q+1]$ ($X' \in$ LABELS $\cup \{S\}$).

$$\#C_i(W') = \#C_i(W[1..q].X') + \#C_i(V')$$

Donc le prédicat d'existence devient

$$AM(V') \leqslant B - AM(W[1..q].X') \tag{4.17}$$

On note $\mathbf{Q}_{SR}^{q,X'}(W)$ l'ensemble des sources potentielles de la forme $W[1..q].X'.V'$ avec $X' \lhd (W.R)[q+1]$. On note que $V' = \emptyset$ lorsque $X' = S$. Tous les calculs qui vont suivre pour déterminer les nombres d'occurrences des C_i dans V' n'ont donc pas lieu d'être dans ce cas.

V' est l'ensemble des mots tels que $W[1..q].X'.V'$ vérifie l'équation de conflit 4.16 et la condition d'existence 4.17. Donc $\mathbf{Q}_{SR}^{q,X'}(W)$ est un NOL ! On notera t_i les nombres d'occurences des C_i dans V'.

Lemme 1

$$\textit{Alors } \mathbf{Q}_{SR}^{q,X'}(W) = \frac{W[1..q].X'.[C_1^{t_1}, \dots, C_n^{t_n}]}{\begin{array}{l} \forall 1 \leqslant j \leqslant d, \sum_{i=1}^{n} \#e_j\big(\sigma_S(C_i)\big)t_i = \\ \qquad \sum_{i=1}^{n} \#e_j\big(\sigma_R(C_i)\big)\#C_i(W) + \#e_j(\delta_R) \\ \qquad - \sum_{i=1}^{n} \#e_j\big(\sigma_S(C_i)\big)\#C_i(W[1..q].X') - \#e_j(\gamma_S), \\ A.(t_1, \dots, t_n) \leqslant B - AM(W[1..q].X') \end{array}} \tag{4.18}$$

Preuve : Si W' est élément du NOL décrit en partie droite de l'équation précédente, il vérifie l'équation de conflit et le prédicat d'existence. De plus par construction de X' on a $W[1..q].X' \ll W.R$, donc le prédicat de séqencement est vérifié. □

$$\text{Donc } \mathbf{Q}_{SR}(W) = \bigcup_{\substack{0 \leqslant q \leqslant \min(p, p+\ell) \\ X' \lhd (W.R)[q+1]}} Q_{SR}^{q,X'}(W) \tag{4.19}$$

Recherche des dépendances directes, K

Posons $\mathbf{K}_{SR}^{q,X'}(W) = \max\, \mathbf{Q}_{SR}^{q,X'}(W)$. Pour calculer $\mathbf{K}_{SR}^{q,X'}(W)$, nous pouvons réutiliser l'algorithme central de l'ADA :

On utilise PIP ([13]) pour trouver le maximum lexicographique de $\mathbf{Q}_{SR}^{q,X'}(W)$. En effet, écrit sous la forme 4.18, on recherche le vecteur $(t_1, \dots, t_n)$ lexigographiquement maximal dans le domaine paramétré par les $\#C_i(W)$, les $\#C_i(W[1..q].X')$, et les éventuels paramètres de structure. L'expression de $\mathbf{K}_{SR}^{q,X'}(W)$ est alors un quast classique (cf. 4.1). Fort heureusement, ce calcul n'est effectué qu'une seule fois ! le quast est le même pour tous les $\mathbf{K}_{SR}^{q,X'}(W)$, seuls les paramètres varient.

L'expression de $\mathbf{K}_{SR}^{q,X'}(W)$ est donc :

$$\mathbf{K}_{SR}^{q,X'}(W) = W[1..q].X'.C_1^{t_1}.\cdots.C_n^{t_n} \tag{4.20}$$

On remarque que si $X' = S$, tous les t_i sont nuls.

Soit $\mathbf{K}_{SR}^{q}(W)$ le maximum des $\mathbf{K}_{SR}^{q,X'}(W)$:

$$\mathbf{K}_{SR}^{q}(W) = \mathbf{K}_{SR}^{q,\max\ \{X' \mid X' \lhd (W.R)[q+1]\}}(W)$$

Comme $\mathbf{K}_{SR}^{q,X'}(W)$ n'est différent de $\perp$ que lorsque $X' = S$ ou $X' = C_r$ et $t_r > 0$ (pour $1 \leqslant r \leqslant n$), on a

$$\mathbf{K}_{SR}^{q}(W) = \mathbf{K}_{SR}^{q,\max\ \{X' \ \mid\ X' \lhd (W.R)[q+1] \wedge (X'=S \vee (X'=C_r \wedge t_r>0))\}}(W) \tag{4.21}$$

$$\text{Et } \mathbf{K}_{SR}(W) = \max\ \{\mathbf{K}_{SR}^{q}(W) \mid 1 \leqslant q \leqslant \min(p, p+\ell)\} \tag{4.22}$$

On remarquera que si σ_S et σ_R sont des substitutions différentes, il est fort probable que $\mathbf{K}_{SR}^{q}(W) = \perp$ sauf pour q petit.

En réalité, $\mathbf{K}_{SR}^{q}(W)$ est un quast composé d'une sélection sur $(W.R)[q+1]$ dont les branches sont les quasts associés aux $\mathbf{K}_{SR}^{q,X'}(W)$: en effet si $(W.R)[q+1] = X''$, $\mathbf{K}_{SR}^{q}(W)$ n'est autre que le quast $\mathbf{K}_{SR}^{q,X''}(W)$ dans lequel on a remplacé tous les $\perp$ par le quast $\mathbf{K}_{SR}^{q,\text{prédécesseur}(X'')}$, puis les $\perp$ de $\mathbf{K}_{SR}^{q,\text{prédécesseur}(X'')}$, etc... récursivement jusqu'à $\mathbf{K}_{SR}^{q,C_1}$. Le risque d'obtenir un «gros» quast est inévitable...

Malheureusement $\mathbf{K}_{SR}(W)$ n'est semble-t-il pas un quast dans le cas général. En effet, c'est un maximum sur un nombre potentiellement non borné de $\mathbf{K}_{SR}^{q}(W)$! On peut néanmoins décrire sa valeur par un algorithme de ce type :

```
K_SR(W) = {
    q := min(p, p + ℓ) − 1;
    while (K^q_SR(W) = ⊥ and q >= 0) do q := q - 1;
    K^q_SR(W);
}
```

Dans certains cas particuliers, il est clair qu'il existe toujours une solution à profondeur bornée ($p - q$ borné), mais il n'est pas facile de trouver des conditions suffisantes générales. Inversement, lorsque les substitutions σ_S et σ_R sont très différentes, il est probable que les dépendances directes seront très éloignées du puits dans l'arbre d'appel, et donc q restera petit.

On peut aussi remarquer que de nombreuses simplifications peuvent être faites avant l'appel à PIP : notamment grâce à la relation entre les nombres d'occurrences des lettres de W et $W[1..q]$, au fait que $W[1..q]$ est un préfixe de W, etc... Et ces simplifications ne seront pas de trop, en effet le problème de programmation linéaire risque d'être beaucoup plus complexe que les problèmes intervenant dans la résolution de l'ADA.

Exemple

Reprenons l'exemple introductif, nous allons pouvoir calculer $\mathbf{K}(W)$.

On remplace $W.([\ldots].\texttt{rgt})^{-1}$ par $W[1..q]$. Donc $j = \#\texttt{lft}(W) - \#\texttt{lft}(W[1..q])$. On développe l'expression de $\mathbf{Q}(W)$ suivant q tel que $(W.S)[q+1] = \texttt{rgt}$ (on a déjà vu que $q = p = |W|$ ne donnait aucune source possible : $\mathbf{Q}_{(a)} = \emptyset$) :

Si $X' = \texttt{lft}$:

$$\mathbf{Q}^{q,\texttt{lft}}(W) \quad = \quad \frac{W[1..q].\texttt{lft}.[\texttt{rgt}^i, \texttt{lft}^k]}{\begin{array}{l} k - \#\texttt{lft}(W) + \#\texttt{lft}(W[1..q]) < N - \#\texttt{lft}(W) - \#\texttt{rgt(W)}, \\ i \geqslant 0, k \geqslant 0, i = \#\texttt{rgt(W)} - \#\texttt{rgt(W[1..q])} - 1 \end{array}}$$

Pour simplifier la résolution on reporte la deuxième contrainte dans la première et on note que

$$\#\mathtt{rgt}(W[1..q]) + \#\mathtt{lft}(W[1..q]) = q$$

$$\mathbf{Q}^{q,\mathtt{lft}}(W) = \frac{W[1..q].\mathtt{lft}.[\mathtt{rgt}^i, \mathtt{lft}^k]}{\begin{array}{l} k < N - q - i - 1, i \geqslant 0, k \geqslant 0, \\ i = \#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} - 1 \end{array}} \tag{4.23}$$

On trouve $i = -1$ lorsque $\#\mathtt{rgt(W)} = \#\mathtt{rgt(W[1..q])}$. Dans ce cas $\mathbf{Q}^{q,\mathtt{lft}}(W) = \emptyset$.

$$\text{Et si } X' = S : \mathbf{Q}^{q,S}(W) = \frac{W[1..q]}{\#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} = 1} \tag{4.24}$$

On peut calculer les dépendances directes «potentielles» (*à la main* ici, mais avec PIP en général) :

$$\begin{array}{rcl} \mathbf{K}^{q,\mathtt{lft}}(W) & = & \mathtt{if}\ N - q - \#\mathtt{rgt(W)} + \#\mathtt{rgt(W[1..q])} > 0 \\ & & \quad \wedge\ \#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} \geqslant 1 \\ & & \mathtt{then}\ W[1..q].\mathtt{lft}.\mathtt{rgt}^{\#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} - 1}.\mathtt{lft}^{N - q - \#\mathtt{rgt(W)} + \#\mathtt{rgt(W[1..q])}} \\ & & \mathtt{else}\ \bot \end{array} \tag{4.25}$$

$$\text{Et } \mathbf{K}^{q,S}(W) = \mathtt{if}\ \#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} = 1\ \mathtt{then}\ W[1..q]\ \mathtt{else}\ \bot \tag{4.26}$$

$$\begin{array}{rcl} \text{Donc } \mathbf{K}^q(W) & = & \mathsf{max}\big(\mathbf{K}^{q,S}(W), \mathbf{K}^{q,\mathtt{lft}}(W)\big) \\ & = & \mathtt{if}\ (W.S)[q+1] = \mathtt{rgt}\ \mathtt{then} \\ & & \quad \mathtt{if}\ \#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} = 1\ \mathtt{then}\ W[1..q]\ \mathtt{else} \\ & & \quad\quad \mathbf{K}^{q,\mathtt{lft}}(W) \\ & & \quad \mathtt{else}\ \bot \end{array}$$

On retrouve la remarque de l'exemple introductif : $\mathbf{K}^{q,\mathtt{lft}}(W) \lhd \mathbf{K}^{q,S}(W)$ car $\mathtt{lft} \lhd S$. De surcroît $\#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} = 0$ est la situation où $\mathbf{K}^{q,S}(W) = \bot$, mais dans ce cas on a aussi $\mathbf{K}^{q,\mathtt{lft}}(W) = \bot$. Donc $\mathbf{K}^{q,\mathtt{lft}}(W)$ n'est jamais une dépendance directe. ($(W.S)[q+1] = S$ ne conduit à aucune dépendance directe possible).

$$\text{Donc } \mathbf{K}^q(W) = \big(\mathtt{if}\ \#\mathtt{rgt(W)} - \#\mathtt{rgt(W[1..q])} = 1\ \mathtt{then}\ W[1..q]\ \mathtt{else}\ \bot\big) \tag{4.27}$$

Dans ce cas, tout s'arrange bien, $\mathbf{K}(W)$ est le plus grand préfixe de suivi de `rgt` dans W. C'est bien entendu un cas extrêmement favorable ! Mais dans le cas général on a vu que l'expression de la source est extrêmement complexe.

4.4.2 Cas des arbres

Dans le cas des arbres il semble peu judicieux de tenter de se ramener à un problème de programmation entière, en effet le nombre d'occurrences des labels apporte bien moins d'information que l'ordre dans lequel les appels récursifs sont effectués.

Le problème est plus difficile que pour les tableaux, on s'en doutait ; mais dans un cas particulier très restrictif, mais qui semble assez fréquent dans la pratique, la source s'exprime facilement.

Nous reprenons le même modèle de programme que dans la section précédente mais avec des restrictions supplémentaires :

- La forme considérée est arborescente : `[<`$\mathcal{E} = e_1, \ldots, e_d$`>]` ;
- On suppose que les substitutions ont pour image des e_j et non pas des mots sur les e_j (ni ϵ), et que ces images dans une même substitution sont toutes distinctes. Les e_j n'apparaissent donc au plus qu'une fois dans σ_S et une fois dans σ_R.

σ_S et σ_R sont alors des bijections de LABELS sur $\mathcal{E}$, on peut donc utiliser leurs inverses.
Dans ce cas il n'y a qu'une dépendance directe possible, et elle s'obtient par substitution :
Si δ_R est un préfixe de γ_S ou γ_S est un suffixe de δ_R, et que $\sigma_R(W).\delta_R.\gamma_S^{-1} \in \mathcal{E}^*$, alors

$$\mathbf{K}_{SR}(W) = \mathbf{Q}_{SR}(W) = \sigma_S^{-1}\big(\sigma_R(W)\big).\sigma_S^{-1}(\delta_R.\gamma_S^{-1})$$

Exemple

Pour l'exemple de l'arbre de la section 3.3, la source est exprimée par la substitution

$$W' = W\{\mathtt{l}/\mathtt{r}, \mathtt{r}/\mathtt{l}\}$$

4.4.3 Commentaire sur ces résultats

Pour conclure cette section un peu technique, on peut dire que nous avons réussi à exploiter le modèle de régularité des structures et le langage LEGS de façon efficace pour l'analyse de flot sur les parcours récursifs de tableaux. De manière beaucoup plus simple, ce modèle a aussi permis d'exprimer simplement les résultats d'une analyse (somme toute assez intuitive) sur les explorations d'arbres. Tout ceci nous semble très prometteur et le modèle fondé sur des graphes de Cayley en sort renforcé. Toutefois nous n'obtenons pas de résultats très applicables pour le moment en matière d'expression de la source. Il est d'ailleurs hors de question d'espérer trouver une expression sous forme de quast, en général.

Un point important dans le succès de l'analyse sur les tableaux a été l'utilisation de PIP. Pour travailler avec d'autres structures il nous a certainement manqué un outil pour calculer sur les mots et les groupes. Comme on l'a dit au début de ce chapitre les groupes automatiques ([9]) et les systèmes de réécriture comme uUTRL[2] sont des pistes intéressantes.

4.5 Applications, développements

Manipuler des structures de données régulières

2. Développé à l'université d'Umeå, `ftp.cs.umu.se`

Le langage LEGS ne prétend pas être la panacée pour manipuler les structures régulières. Un rapprochement avec les travaux effectués au LRI autour du langage $8_{1/2}$ (pour lequel à été imaginé le modèle des formes repris par LEGS, [14]) serait sans doute intéressant.

Des outils comme Maple™ ou Pari [3] ([4]) peuvent s'avérer très utiles pour manipuler les groupes commutatifs (dans leur forme normale de Smith).

Ordonnancement, génération de code parallèle

Les analyses de flot de données classiques, comme l'ADA sont à la base des parallélisateurs automatiques (comme le Parallélisateur Automatique pour Fortran, PAF, développé à l'Université de Versailles). Des algorithmes d'ordonnancement performants ont été trouvés depuis le début des années 90. Basés sur la méthode de l'hyper-plan pour les dépendances uniformes ([7]), ou bien sur PIP ou Omega pour les dépendances affines (générées par l'ADA, [12, 11]). Le problème du placement des données sur un réseau de processeurs utilise lui aussi l'analyse de flot de données ([6]) ; les idées développées par Miguel Alabau pour la compilation de langages parallèles opérant sur des graphes (il introduit d'ailleurs des constructions proches de celles de LEGS, [2]) sont très avancées en la matière.

Vérification, analyse d'étendue

La compilation de LEGS est très dépendante d'une analyse statique efficace pour optimiser les performances à l'exécution. Il serait intéressant d'étudier l'extension aux structures régulières des techniques d'analyse d'étendue ([24]), qu'il s'agisse d'analyse floue ou exacte.

De plus les structures déclarées par la syntaxe `construct` en LEGS peuvent être très complexes. Nous nous sommes restreints dans la GSDA à des déclarations inductives non gardées (pas de `when`), mais les gardes introduisent une variété beaucoup plus grande de structures, pour lesquelles il est parfois même difficile de déterminer si elles sont bornées où non (même avec des contraintes affines). Un étude de ces déclarations devrait être menée, le problème se ramenant peut-être à des méthodes de point fixe.

Optimisation de l'occupation en mémoire

Enfin, l'analyse de flot de données est conçue pour éviter l'explosion des dépendances, mais du même coup on perd toutes les dépendances de sortie et les anti-dépendances (cf. 1.3). On contourne ce problème en parallélisation automatique en procédant à une *conversion en assignation unique*. C'est à dire que les objets accédés plusieurs fois en écriture sont dupliqués jusqu'à ce que toutes les cellules mémoires soient écrites au plus une fois. Cette transformation utilise bien sûr les résultats de l'analyse de flot de données. Il en découle une augmentation du degré de parallélisme que l'on peut obtenir, mais aussi une explosion de la place mémoire utilisée (généralement une structure de complexité spatiale n^2 nécessite n^3 cellules mémoire après expansion).

Des solutions récentes ont été trouvées pour résoudre ce problème, elles consistent à réduire au maximum la redondance d'information en ne conservant que le strict minimum en mémoire. Une des plus efficaces fait appel au concept de *Working Set* ([20]), et elle cherche à «neutraliser» les dépendances couvertes par des dépendances de flot, pour réduire le nombre de réplications. La

3. Développé à l'Université de Bordeaux I : `ftp megrez.math.u-bordeaux.fr`

mise en assignation unique des programmes LEGS en vue de leur parallélisation nécessitera une extension de cette méthode aux structures régulières et aux algorithmes récursifs.

Conclusion

L'analyse de flot de données est un problème complexe et vaste. On l'a vu tout au long de ce travail, de nombreux domaines de l'informatique y sont liés et les techniques mises en œuvre requièrent des notions mathématiques variées. L'extension que nous avons décrite introduit un modèle fondé sur la théorie des groupes et des graphes de Cayley qui se montre assez bien adapté à la représentation informatique de structures de données régulières. En effet le langage LEGS qui l'accompagne ne se contente pas de rendre possible des analyses de flot de données dans un cadre élargi, il introduit une vision nouvelle des structures de données fondée sur les voisinages. Ce sont ces voisinages, et leurs propriétés de régularité qui rendent possible l'analyse statique. Nous avons décrit les structures sémantiques originales de LEGS : la déclaration de formes et les syntaxes `construct` et `explore`. Celles-ci privilégient les parcours récursifs comme moyen naturel d'expression des algorithmes ; les boucles `for` ou `while` étant considérées comme des cas particuliers d'exploration récursive. Ce langage a rendu possible la caractérisation de deux nouvelles classes de programmes pour lesquels une analyse de flot exacte est possible : les explorations récursives d'arbres et de tableaux.

En revanche, les graphes de Cayley ne sont pas entièrement satisfaisants, ils ne sont pas assez généraux pour modéliser toutes les structures que l'on qualifie intuitivement de régulières (notamment les types récursifs). Une partie de compilateur LEGS a été implémentée en CAML ; elle a permis de valider les choix syntaxiques et certains choix sémantiques du langage. Les résultats que l'on a obtenu en ce qui concerne l'analyse de flot sont prometteurs et pourraient faire l'objet d'expérimentations dans le cadre d'un prototype d'analyseur pour LEGS ; mais ils ont aussi mis en évidence toute la difficulté du problème, notamment en termes d'expression des ensembles de sources potentielles et du calcul des maximums.

Il semble que le recours à des logiciels de calcul symbolique, des systèmes de réécriture, des algorithmes de simplification d'expressions (arithmétiques et conditionnelles) et même des systèmes de preuve, soit inévitable. De plus, nous avons fait usage à deux reprises d'objets mathématiques apparemment nouveaux pour lesquels une étude approfondie mériterait d'être menée : il s'agit des explorations (et des constructions) récursives gardées et des langages à occurrences comptées. Il faudra probablement abandonner la recherche d'une analyse exacte à tout prix (c'est à dire à chaque fois que c'est possible), non seulement pour généraliser le champ d'applications des analyses de flot, mais aussi pour que les résultats soient «raisonnablement» utilisables. Nous pensons que ce travail ouvre des perspectives intéressantes pour la manipulation des structures de données et pour l'analyse statique, et nous espérons qu'il sera suivi d'applications dans les domaines de la compilation, de la vérification de programmes et du parallélisme.

Annexe A

Description de la syntaxe LEGS

Pour information, voici l'intégralité de la grammaire LEGS en notation pseudo-BNF.

program → statement_list
statement_list → declaration_list instruction_list
declaration_list → declaration ; [declaration_list]
declaration → shape_def | type_def | var_decl
| index_decl | function_def
| substitution_def | construct_def
block → `{` statement_list `}`

Partie declarative :

shape_def → `shape` name `=` `[` `<` name_list `>` word_list `]`
| `shape` name `=` `[` | name_list | word_list `]`
| `shape` name `=` `[` `[` name_list `]` word_list `]`
name_list → name [`,` name_list]
type_def → `type` name `=` name name
var_decl → `var` name : name
construct_def → `construct` name [`while` boolean_expression] `{` direction_list `}`
function_def → `function` name `(` [arg_def_list] `)` [`:` name] block
arg_def_list → name : name [`,` arg_def_list1]
substitution_def → `substitution to` name `=` substitution_list
substitution_list → word `/` name [`,` substitution_list]

Partie operative :

instruction_list → instruction ; [instruction_list]
instruction → block | assign | call | conitional | explore
| `nop` | `return` expression
| direction
direction → name : [`when` boolean_expression] `recurse`
direction_list → direction [direction_list]
explore → `explore` `(` name_list `)` [`while` boolean_expression] block
conditional → `if` expression `then` instruction [`else` instruction]
call → name `(` [arg_list] `)`
arg_list → expression [`,` arg_list]
word_list → word [`,` word_list]
word → monomial [`.` word] | `(` word `)` | [name] `{` [substitution] `}`
substitution → word `/` name [`,` substitution]
monomial → name | `{` substitution `}` | `{` name `}` | word `^` expression
assign → name [`[` word `]`] `:=` expression

Expressions élémentaires:

```
        expression  →  call | name | integer | name [ word ]
                    |  expression + expression
                    |  expression - expression
                    |  expression * expression
                    |  expression / expression
                    |  expression % expression
                    |  boolean_expression | # name | word
boolean_expression  →  expression = expression
                    |  expression <> expression
                    |  expression <= expression
                    |  expression >= expression
                    |  expression < expression
                    |  expression > expression
                    |  boolean_expression and boolean_expression
                    |  boolean_expression or boolean_expression
                    |  not boolean_expression
                    |  ? name [ word ]
                    |  boolean
           integer  →  -?[0-9]+
           boolean  →  true | false
```

Bibliographie

[1] AHO (A. V.), SETHI (R.) et ULLMAN (J. D.). – *Compilers : Principles, Techniques and Tools.* – Reading, Mass., Addison-Wesley, 1986.

[2] ALABAU (M.). – *Une expression des algorithmes massivement parallèles à structures de données irrégulières.* – Thèse de PhD, Université de Bordeaux I, Septembre 1994.

[3] BARTHOU (D.), COLLARD (J.-F.) et FEAUTRIER (P.). – Fuzzy array dataflow analysis. *In : Principles and Practice of Parallel Programming, PPoPP'95.* – ACM.

[4] COHEN (H.). – *A course in computational algebraic number theory.* – Springer-Verlag, 1993, *Graduate Text in Mathematics*, volume 138.

[5] COURCELLE (B.). – The monadic second order logic of graphs. *Information and Computation*, 1990.

[6] DARTE (A.) et ROBERT (Y.). – *Mapping Uniform Loop Nests onto Distributed Memory Architectures.* – Rapport technique n° 93-03, LIP-IMAG, Janvier 1993.

[7] DARTE (A.) et ROBERT (Y.). – *Mapping Uniform Loop Nests onto Distributed Memory Architectures.* – Rapport technique n° 93-03, LIP-IMAG, Janvier 1993.

[8] DEUTSCH (A.). – Interprocedural may-alias analysis for pointers: Beyond k-limiting. *In : Proc. ACM SIGPLAN Conf. PLDI.*

[9] EPSTEIN (D. B. A.). – *Word Processing in Groups.* – Addison Wesley, 1992.

[10] FEAUTRIER (P.). – Dataflow analysis of array and scalar references. *Int. Journal of Parallel Programming*, 1991.

[11] FEAUTRIER (P.). – Some efficient solutions to the affine scheduling problem, multidimensional time. *Int. Journal of Parallel Programming*, 1992.

[12] FEAUTRIER (P.). – Some efficient solutions to the affine scheduling problem, one-dimensional time. *Int. Journal of Parallel Programming*, 1992.

[13] FEAUTRIER (P.) et TAWBI (N.). – *Resolution de Systèmes d'Inéquations Linéaires; mode d'emploi du logiciel PIP.* – Rapport technique n° 90.2, IBP-MASI, Avril 1992.

[14] GIAVITTO (J.-L.), MICHEL (O.) et SANSONNET (J.-P.). – Group based fields. *In : Proc. Parallel Symbolic Languages and Systems (PSLS)*, éd. par HALSTEAD (R. H.), TAKAYASU (I.) et QUEINNEC (C.). – Springer-Verlag.

[15] Harrison III (W. L.). – An efficient approach to computing fixpoints for complex program analysis. *Journal of Programming Languages*, vol. 3, Mars 1995, pp. 31–38.

[16] Hendren (L. J.) et Nicolau (A.). – Parallelizing programs with recursive data structures. *In : Int. Conf. on Parallel Computing.*

[17] Hendren (L. J.) et Nicolau (A.). – Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. *In : Proc. ACM SIGPLAN Conf. PLDI.*

[18] Jones (N. D.) et Muchnick (S. S.). – A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *ACM Press*, 1982.

[19] Klarlund (N.) et Schwartzbah (M. I.). – Graph types. *In : Proc. 20^{th} PoPL.* – ACM.

[20] Lefebvre (V.) et Feautrier (P.). – Storage management in parallel programs. – 1996. To be published.

[21] Leroy (X.) et Weis (P.). – *Le langage Caml.* – InterEditions, 1993.

[22] Leroy (X.) et Weis (P.). – *Manuel de Référence du langage Caml.* – InterEditions, 1993.

[23] Liskov (B.), Snyder (A.), Atkinson (R.) et Schaffert (C.). – Abstraction mechanisms in clu. *Communications of the ACM*, Août 1977.

[24] Lisper (B.) et Collard (J.-F.). – *Extent analysis of data-fields.* – Rapport technique n° 94:03, Royal institute of Technology, Sweden, Janvier 1994.

[25] Maydan (D. E.), Amarasinghe (S. P.) et Lam (M. S.). – Array dataflow analysis and its use in array privatisation. *In : Proc. of the Conf. on Principles of Programming Languages.* – ACM.

[26] Michel (O.). – *Design and implementation of* $8_{1/2}$*, a declarative data-parallel laguage.* – Rapport technique, Laboratoire de Recherche en Informatique (LRI), Dècembre 1995.

[27] Pugh (W.). – A fast and practical integer programming algorithm for dependance analysis. *Communication of the ACM*, vol. 8, Août 1992, pp. 102–114.

[28] Pugh (W.) et Wonnacott (D.). – *An exact method for analysis of value-based data dependances.* – Rapport technique n° 3196, University of Maryland, Décembre 1993.

[29] Sims (C. C.). – *Computation with Finitely Presented Groups.* – Cambridge University Press, 1995.