# Séries génératrices et analyse automatique d'algorithmes

Paul Zimmermann


# THÈSE

présentée à

L'ÉCOLE POLYTECHNIQUE
PALAISEAU

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE
spécialité INFORMATIQUE

par

Paul ZIMMERMANN

—→←—

Sujet : Séries génératrices et analyse automatique d'algorithmes

Soutenue le 6 mars 1991, devant le jury composé de :

| | |
|---|---|
| François BERGERON | Rapporteur |
| Maylis DELEST | |
| Philippe FLAJOLET | |
| Jean-Jacques LÉVY | Rapporteur |
| Michèle SORIA | |
| Jean-Marc STEYAERT | |
| Jean VUILLEMIN | Président |

*A Marie,*
*à mes parents.*

Je remercie tout d'abord les personnes qui me font l'honneur de faire partie de mon jury de thèse. Philippe Flajolet a été mon maître de conférences en première année à l'École Polytechnique ; il m'a enseigné le langage Pascal, mais surtout m'a fait découvrir la beauté d'un programme bien écrit.

En seconde année, Jean Vuillemin a été mon professeur d'informatique ; par le langage Le_Lisp, il nous a présenté les principes fondamentaux de l'informatique théorique. C'est aussi Jean Vuillemin qui m'a initié à la recherche : au sein du projet Vlsi de l'Inria, il m'a appris à combiner théorie et expérimentation, et m'a conseillé dans mes choix d'outils de travail (LaTeX, Emacs). Je lui en suis aujourd'hui particulièrement reconnaissant, et je le remercie d'avoir accepté de présider mon jury de thèse.

Jean-Marc Steyaert m'a suivi tout au long de ma scolarité : il m'a enseigné le calcul formel (avec Marc Giusti et Philippe Flajolet) et le langage Le_Lisp à Polytechnique, puis l'analyse d'algorithmes au magistère d'informatique de l'École Normale Supérieure. Je le remercie également pour les discussions enrichissantes que nous avons eues à l'Inria.

J'ai rencontré Michèle Soria au projet Algo ; par son intérêt sans relâche pour l'entreprise $\Lambda\Upsilon\Omega$ (même lorsque le système permit de retrouver automatiquement ses propres résultats) et sa constante bonne humeur, elle a largement contribué à la bonne ambiance régnant au sein du projet. Je la remercie particulièrement pour le soin avec lequel elle a relu une version préliminaire de mon rapport.

Je dois beaucoup à Jean-Jacques Lévy, qui a toujours su s'intéresser aux travaux des jeunes chercheurs. Lors de ses cours d'informatique graphique au magistère de l'ENS, dans ses comptes-rendus de voyage, ou même à la cafétéria de l'Inria, j'ai apprécié son goût de l'anecdote et du détail historique. Je regrette le temps où, occupant un bureau à coté du sien, je pouvais entendre ses récriminations contre les "dieux de l'informatique".

Maylis Delest, lors de ses passages à Paris ou au cours de colloques, a montré beaucoup d'intérêt pour nos recherches et nous a encouragés, Bruno Salvy et moi, à les poursuivre. C'est sans doute avec Michèle Soria l'utilisatrice la plus assidue (et la plus exigeante) du système $\Lambda\Upsilon\Omega$.

Lors de son séjour en France en 1989, François Bergeron m'a fait découvrir un autre versant de la combinatoire, vue depuis le Québec ; grâce au courrier électronique, nous sommes restés en contact permanent depuis.

Je suis particulièrement reconnaissant envers Jean-Jacques Lévy et François Bergeron, qui ont accepté la lourde tâche de rapporteur, et qui l'ont assumée avec enthousiasme et promptitude. N'oublions pas les heures qu'ils ont consacrées à la lecture de cette thèse !

Je tiens à remercier tout spécialement mes partenaires dans l'entreprise $\Lambda\Upsilon\Omega$ : Philippe Flajolet et Bruno Salvy. Philippe Flajolet a été un directeur de thèse formidable : malgré ses lourdes responsabilités de chef de projet, il a toujours été disponible pour répondre à mes interrogations ou pour m'indiquer d'éventuelles voies de recherche, sans jamais me les imposer. De plus, il a le

tact nécessaire pour ne pas décourager le chercheur qui n'a fait que réinventer la roue. Il a su aussi me confier des responsabilités correspondant à mes goûts au sein du projet Algo. Enfin, en relisant de nombreuses fois mon rapport aux différents stades de son élaboration, il m'a aidé à rendre la lecture de cette thèse plus attrayante.

Plus qu'un collègue de travail, Bruno Salvy a été pour moi un compère dans mes recherches. Il a toujours répondu à mes nombreuses requêtes, parfois impatientes, et a su me freiner au moment opportun dans mes modifications *ad hoc* du système $\Lambda\Upsilon\Omega$. Par son humour sans limite, il a égayé les discussions du projet Algo. Enfin, sans les centaines d'heures qu'il a passées à écrire (et à corriger) des programmes MAPLE, mon travail n'aurait pas le même intérêt.

Ma reconnaissance va aussi aux autres membres du projet Algorithmes de l'Inria : François Morain qui a été constamment disponible à mon égard, et dont j'admire le franc-parler, Luc Albert qui a usé de son peu de temps libre pour m'orienter vers l'analyse d'algorithmes d'unification, Philippe Jacquet qui a su faire profiter autour de lui de ses connaissances en protocoles de communication et en mécanique, Mireille Régnier qui a veillé à ce que mes compétences d'ingénieur système soient suffisamment utilisées, Henry Crapo qui, par ses contacts européens, a promu le système $\Lambda\Upsilon\Omega$, Loïc Merel et Philippe Dumas dont l'humeur est sans cesse au beau fixe. Je n'oublie pas la secrétaire du projet, Virginie Collette, dont l'ardeur et le sourire sans relâche vinrent à bout des tâches les plus fastidieuses.

Un grand merci à Robert Ehrlich et Francis Dupont, qui ont toujours été disponibles pour m'instruire et m'aider dans les problèmes techniques liés à UNIX, à Michel Mauny et Pierre Weis qui m'ont conseillé lors de difficultés liées au langage CAML. En particulier, l'interface entre CAML et MAPLE a été réalisée grâce à Robert Ehrlich.

Je voudrais aussi remercier tout ceux qui ont contribué, de près ou de loin, à ce travail : les enseignants d'informatique de Polytechnique, de Paris VII et de l'ENS, particulièrement Patrick Cousot qui m'a beaucoup apporté par son dynamisme, Christian Queinnec, Guy Cousineau et Gérard Huet ; mon collègue et homonyme allemand Wolf Zimmermann, qui a examiné soigneusement une version préliminaire de mon rapport, bien qu'il soit écrit dans une langue étrangère pour lui ; ceux grâce à qui la mise en page de mon rapport a été grandement facilitée, notamment Donald Knuth et Leslie Lamport qui ont réalisé les systèmes de traitement de texte TEX et LATEX ; les sportifs de l'Inria, en particulier Philippe Robert et Olivier Monga.

# Table des matières

| Notations | |
|---|---|
| $A \backslash B$ | complémentaire de $B$ dans $A$ |
| $Q(f)$ | $\frac{1}{1-f}$ |
| $L(f)$ | $\log \frac{1}{1-f}$ |
| $E(f)$ | $\exp(f)$ |
| $E^{\star}(f)$ | $\frac{\exp(f)-1}{f}$ |
| $L^{\star}(f)$ | $\frac{1}{f} \log \frac{1}{1-f}$ |
| $\|t\|$ | taille de l'objet $t$ |
| $U$ | non-terminal |
| $\mathcal{U}$ | ensemble des objets dérivés par $U$ |
| $\mathcal{U}_n$ | ensemble des objets de taille $n$ de $\mathcal{U}$ |
| $U_n$ ou $u_n$ | card$(\mathcal{U}_n)$ |
| $[z^n]f(z)$ | coefficient de $z^n$ dans $f(z)$ |
| $P(x : X)$ | procédure $P$ dont l'argument est $x$, de type $X$ |
| $P : X$ | procédure $P$ de type $X$ |
| $f(n) = O(g(n))$ | $\|f(n)\| \leq C\|g(n)\|$ pour $n$ assez grand |
| $f(n) = \Omega(g(n))$ | $\|f(n)\| \geq C\|g(n)\|$ pour $n$ assez grand |
| $f(n) = \Theta(g(n))$ | $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$ |
| $x \simeq d_0.d_1 \ldots d_k$ | $\|x - d_0.d_1 \ldots d_k\| \leq 10^{-k}$ |
| $RootOf(F(\_Z, z))$ | solution en $Y$ de $F(Y, z) = 0$ |

# Introduction

*L'ordinateur n'est pas créé* par, *mais* pour *sa fonction. Nuance ! Son énorme capacité de mémoire, sa rapidité opérationnelle ne peuvent faire oublier qu'il est rigoureusement asservi à sa tâche et que, bourré de données, entièrement dues à l'intelligence humaine, il ne fait que raccourcir à l'extrême les calculs et les recherches.*

Hervé Bazin, *Abécédaire*

Ce document décrit l'une des deux composantes d'un procédé général d'analyse d'algorithmes, dont l'autre composante fait l'objet du travail de thèse de Bruno Salvy [Sal91]. En combinant ces deux composantes, on obtient une méthode pour analyser automatiquement des algorithmes, la *méthode des séries génératrices*. Afin de montrer que cette théorie est effective, un système dénommé LAMBDA-UPSILON-OMEGA (ou $\Lambda\Upsilon\Omega$) a été construit sur ce modèle. Le premier prototype de $\Lambda\Upsilon\Omega$ date du début de l'année 1988 [FSZ89b], et en 1989, le système analyse déjà une classe non triviale de programmes [FSZ89a]. Ce travail et celui de B. Salvy détaillent les principes de la méthode des séries génératrices, qui sont succinctement résumés dans [FSZ91].

Le mot "algorithme" provient du nom d'un mathématicien originaire d'Ouzbekistan, Mukhammad ibn Musa al-Khwârizmî, contemporain de Charlemagne et de Louis I$^{\text{er}}$. D'après le Petit Larousse, un algorithme est un "processus de calcul permettant d'arriver à un résultat final déterminé". L'*analyse* d'un algorithme consiste à déterminer le coût du processus (sa durée par exemple) en fonction de l'état de départ (les données du problème). Quant à elle, l'analyse *automatique* d'algorithmes consiste à faire faire cette analyse par un ordinateur lui-même. Cette thèse définit une classe non triviale d'algorithmes pour lesquels l'analyse automatique est possible, et précise les règles de calcul correspondantes.

En général, il existe plusieurs algorithmes pour résoudre un problème donné, c'est-à-dire plusieurs chemins pour atteindre un but fixé (le résultat) à partir d'un certain point de départ (les données du problème). Considérons par exemple le problème du tri : partant d'une liste d'objets en ordre quelconque (par exemple des entiers), il s'agit de ranger ces objets selon un certain critère (en ordre croissant par exemple). Deux algorithmes de tri illustreront notre propos.

**Algorithme** Tri par sélection.

1. sélectionner le plus petit entier de la liste des données,
2. ôter l'entier sélectionné et l'ajouter à droite dans la liste résultat,
3. s'il reste des entiers dans la liste des données, aller en 1.

**Algorithme** Tri par insertion.

Figure 1 : Al-Khwârizmî (env. 780-850)

1. prendre l'entier le plus à gauche de la liste des données et l'en ôter,
2. le placer à droite de la liste résultat,
3. si l'entier à sa gauche est plus grand, les échanger et retourner en 3,
4. s'il reste des entiers dans la liste des données, aller en 1.

L'analyse d'algorithme consiste à étudier le comportement d'un certain critère en fonction des données ; par exemple pour un algorithme de tri, on pourra s'intéresser au nombre de comparaisons effectuées au cours du tri, ou au nombre de déplacements d'éléments. En termes plus informatiques, ce critère est appelé le *coût* de l'algorithme.

Le principe fondamental est que le coût de l'algorithme dépend essentiellement d'un petit nombre de *caractéristiques* des données, aisément calculables. Par exemple, pour un algorithme de tri, on pourra prendre comme caractéristique le nombre d'objets à trier. Lorsqu'il y a une seule caractéristique, on a l'habitude de l'appeler la *taille* des données. Après que les caractéristiques ont été choisies, deux cas de figure peuvent se présenter :

1. la connaissance des caractéristiques suffit à déterminer le coût de l'algorithme,
2. le coût de l'algorithme diffère, même pour des données de caractéristiques identiques.

Le premier cas est le plus favorable, et permet souvent de connaître exactement le coût de l'algorithme. Considérons par exemple l'algorithme de tri par sélection décrit plus haut, avec comme taille le nombre $n$ d'entiers à trier, et comme coût le nombre de comparaisons effectuées. L'étape de sélection (1) nécessite $n-1$ comparaisons pour déterminer le plus petit entier de la liste, puis $n-2$ pour trouver le plus petit parmi ceux qui restent, puis $n-3$ et ainsi de suite jusqu'à ce que la liste des données soit vide. Le nombre total de comparaisons utilisées est donc

$(n-1)+(n-2)+\cdots+1+0 = n(n-1)/2$, et cela quelle que soit la distribution initiale des entiers. Nous sommes donc dans le cas favorable où le coût ne dépend que des caractéristiques (ici $n$), et qui plus est nous avons obtenu une relation simple entre $n$ et le coût de l'algorithme de tri par sélection sur des listes de $n$ éléments : $C_n = n(n-1)/2$.

Mais le plus fréquemment, les caractéristiques ne suffisent pas à déterminer le coût. Pour le tri par insertion par exemple, le nombre de comparaisons ne dépend pas uniquement du nombre d'objets, mais aussi de l'ordre dans lequel ils sont initialement rangés (en particulier, le coût est plus faible s'ils sont presque triés). Pour mieux cerner le comportement de l'algorithme, on a alors recours à trois analyses complémentaires :

- le coût minimal sur toutes les données de même taille (le meilleur cas),
- le coût maximal sur toutes les données de même taille (le pire des cas),
- le coût moyen sur toutes les données de même taille, chaque donnée étant par exemple considérée comme équiprobable (analyse en moyenne).

L'analyse du meilleur et du pire des cas donne un encadrement du coût de l'algorithme. Quelquefois, cet encadrement suffit à départager deux algorithmes. Par exemple, pour le tri par insertion d'une liste de $n$ entiers, le nombre minimal de comparaisons est $n-1$ (lorsque la liste est déjà triée), et le nombre maximal est $n(n-1)/2$ (lorsqu'elle est triée en ordre inverse). Le tri par insertion est par conséquent toujours plus rapide vis-à-vis du nombre de comparaisons que le tri par sélection, qui nécessite lui $n(n-1)/2$ comparaisons, quelle que soit la distribution initiale des entiers.

Mais dans la plupart des cas, les encadrements donnés par l'analyse du meilleur et du pire des cas ne sont pas disjoints, et il faut recourir à une analyse plus fine. Par exemple, il existe un autre algorithme de tri dénommé *QuickSort* (ce mot anglais signifie "tri rapide") dont le nombre minimal de comparaisons est $n \log_2 n$ ($n-1$ pour le tri par insertion), et dont le nombre maximal est $n(n-1)/2$ (idem pour le tri par insertion). Quel algorithme choisir pour trier des listes de 5 entiers ? de 10 entiers ? de 100 entiers ? Seule l'analyse en moyenne permet de répondre à ces questions. Dans ce cas particulier, l'algorithme de tri par insertion nécessite en moyenne $n^2/4$ comparaisons pour trier $n$ entiers, alors que *QuickSort* n'en nécessite que $2n \log n$. Lorsque $n$ devient grand, il est donc préférable d'utiliser *QuickSort*. Plus précisément, *QuickSort* devient plus rapide en moyenne à partir de $n = 20$, comme on le verra au chapitre 5 (section 5.2).

Par opposition à l'analyse dans le cas le pire, l'analyse en moyenne dépend du modèle probabiliste choisi (ou imposé). Le modèle le plus couramment utilisé est celui de la distribution uniforme sur l'ensemble des données de mêmes caractéristiques. Parfois, ce modèle induit par la définition des caractéristiques rend l'analyse en moyenne difficile. Considérons par exemple l'algorithme d'Euclide (300 ans avant JC)[1] qui calcule le plus grand commun diviseur de deux entiers, et prenons comme taille la valeur du plus grand entier. Avec cette notion de taille, le coût dans le cas le pire est connu depuis un siècle et demi (G. Lamé, 1845), alors que le premier terme du coût moyen était encore partiellement inconnu il y a 25 ans (Heilbronn et Dixon, 1970), et les deux termes suivants ont été obtenus tout récemment [Nor90]. Néanmoins pour de nombreux problèmes, comme ceux étudiés dans cette thèse, le modèle uniforme induit par la notion naturelle de taille est bien adapté à l'analyse en moyenne.

Même si le sujet de ce travail est l'analyse en moyenne, nous ne pouvons pas ne pas parler des recherches effectuées pour automatiser l'analyse dans le cas le pire. A notre connaissance, le seul

[1] Il semblerait cependant que cet algorithme ait été connu 200 ans avant Euclide [Knu81, page 318].

système spécialisé dans ce domaine est ACE [Met88]. Ce système, dont le nom est l'acronyme de AUTOMATIC COMPLEXITY EVALUATOR (calculateur automatique de complexité), a été développé par D. Le Métayer en 1985. ACE analyse des programmes écrits dans un langage fonctionnel nommé FP, et détermine une borne supérieure du coût de ces programmes. Nous reviendrons sur ce système à la fin de la section 1.

A présent, nous allons passer en revue les différents procédés qui ont été envisagés pour automatiser l'analyse en moyenne. Nous distinguons les procédés qui calculent seulement le coût moyen, et ceux qui déterminent toute la distribution des coûts, comme celui proposé par Hickey et Cohen, qui est d'ailleurs le seul à notre connaissance.

Dans l'approche de Hickey et Cohen [HC88], les données sont caractérisées par des *fonctions de distribution*, et un programme est une fonction qui associe à une distribution initiale, une distribution finale. Par exemple, si $\mathbf{x}$ représente le vecteur des variables entières du programme, $in(\mathbf{x})$ est la probabilité que le vecteur initial des variables prenne la valeur $\mathbf{x}$, et $out(\mathbf{x})$ est la probabilité qu'à la fin du programme, le vecteur des variables entières égale $\mathbf{x}$. Hickey et Cohen proposent un premier modèle basé sur les travaux de Kozen en sémantique des programmes. Ce modèle s'applique à un langage simple sans procédures (SL), dont les programmes manipulent un nombre fini de variables entières à l'aide de boucles **while**, d'instructions **if-then-else** et d'affectations de variables (y compris par des valeurs aléatoires). Un système a été construit sur ce modèle, THE PERFORMANCE COMPILER. Ce système prend en entrée un programme écrit dans le langage SL et une distribution initiale sous forme symbolique, et produit un système d'équations dont la solution est la distribution finale des variables du programme. Hickey et Cohen considèrent THE PERFORMANCE COMPILER comme la première partie d'un système d'analyse de programmes, qui comprendrait aussi des outils de simplification et de résolution des équations produites :

> *It* [THE PERFORMANCE COMPILER] *is the first part of a program analysis tool that would allow a user to analyze probabilistic programs interactively in five phases, as listed below:*
>
> 1. *Obtain the input distribution function and the program from the user.*
> 2. *Generate the recursive system of equations.*
> 3. *Simplify the system of equations.*
> 4. *Help the user find the general solution to the system of equations.*
> 5. *Help the user find the smallest nonnegative solution to the system of equations.*
>
> *We envision this tool as part of a larger interactive system, including in particular a package for manipulating and solving systems of symbolic equations (e.g.,* MACSYMA*). What becomes apparent from the existing analysis and from the analyses we have undertaken in this paper is that the solution of general difference equations and formal summations plays a critical role in automating program analysis. There have been many papers on automatically solving various sorts of difference equations and formal summations. A good performance compiler would need to incorporate state-of-the-art algorithms for solving these problems.*

Dans une seconde partie de leur article [HC88] est présenté un second modèle plus ambitieux. Les programmes sont écrits dans le langage FP, et correspondent à des applications d'un espace de données $\Omega$ dans lui-même. Les données sont des atomes ou des vecteurs dont les éléments peuvent eux-mêmes être des vecteurs ; elles sont donc équivalentes à des arbres d'arité quelconque. La compilation des programmes en équations est réalisée à l'aide de grammaires probabilistes attribuées (*Attributed Probabilistic Grammars*, ou *APG*). Trois exemples simples d'analyse sont

| date | nom du système | langage de description | type d'analyse |
|---|---|---|---|
| 1975 | METRIC | Lisp | M, W-C |
| 1985 | ACE | FP | W-C† |
| 1988 | THE PERFORMANCE COMPILER | SL | M |
| 1988 | COMPLEXA | STYFL | M, W-C |
| 1988 | $\Lambda\Upsilon\Omega$ | Adl | M |

† : seule une macro-analyse est effectuée ($O(n)$, $O(n^2)$, $O(e^n)$)

Figure 2 : Les systèmes d'analyse automatique d'algorithmes.

donnés : concaténation de deux listes, retournement d'une liste avec ou sans retournement des sous-listes. Cet autre modèle permet également d'obtenir le coût moyen du tri de $n$ objets par insertion dans un arbre binaire de recherche,

$$N_{\text{insert}} = 2(n+1)H_n - 3n \sim 2n \log n,$$

mais il ne s'agit pas à proprement parler d'une analyse automatique ; en effet, cette analyse consiste en 6 pages de calculs comprenant deux lemmes spécialisés dont la preuve semble difficile à automatiser. Ainsi, ce second modèle semble mieux s'adapter à une analyse *semi-automatique*, où l'utilisateur interagit avec le programme pour guider l'analyse. D'ailleurs, nous ne connaissons aucun système réalisé sur ce modèle.

Dans la citation de Cohen et Hickey, il apparaît clairement qu'un système d'analyse de programmes comprend deux phases principales (les phases numérotées 2 et 4) :

- une première phase qui génère des équations de fonctions mathématiques représentant le coût du programme. C'est en quelque sorte une *projection* de l'espace des programmes vers un espace mathématique ;
- une seconde phase qui, à partir des équations produites par la première phase, détermine des informations sur le coût moyen, soit une forme explicite en fonction des caractéristiques, soit une forme asymptotique. C'est une phase d'*extraction*.

Cette séparation avait déjà été faite par Ramshaw dans le résumé de sa thèse intitulée *Formalizing the Analysis of Algorithms* :

> *Consider the average case analyses of particular deterministic algorithms. Typical arguments in this area can be divided into two phases. First, by using knowledge about what it means to execute a program, an analyst characterizes the probability distribution of the performance parameter of interest by means of some mathematical construct, often a recurrence relation. In the second phase, the solution of this recurrence is studied by purely mathematical techniques.*

Par conséquent, les capacités d'un tel système se situent à *l'intersection* des capacités de chacune des phases. Le travail qui est présenté dans cette thèse constitue la première phase d'une méthode que nous appellerons *la méthode des séries génératrices*, dont la seconde phase est décrite dans la thèse de Bruno Salvy [Sal91].

Dès lors, nous considérons l'analyse en moyenne dans le cas où les données sont distinguées par *une seule* caractéristique. Suivant l'usage, nous désignons cette caractéristique sous le nom de *taille*,

nous notons $|d|$ la taille de la donnée $d$, et nous convenons d'utiliser la lettre $n$ pour représenter la taille d'une donnée. On distingue alors deux types de problèmes : les problèmes dits *discrets*, lorsque l'ensemble des données est fini ou dénombrable, et les autres problèmes, dits *continus*. Dans le cas des problèmes discrets, ceux qui nous intéressent plus particulièrement ici, nous pouvons encore distinguer deux classes de méthodes : les méthodes *probabilistes* qui déterminent le coût moyen par estimation directe de probabilités, et les méthodes *combinatoires* qui dénombrent le nombre de données de taille $n$, totalisent le coût du programme sur ces données, et extraient la moyenne par l'égalité :

$$\text{coût moyen sur les données de taille } n = \frac{\text{coût total sur les données de taille } n}{\text{nombre de données de taille } n}. \qquad (1)$$

Les méthodes probabilistes sont surtout utilisées pour montrer qu'une propriété est presque sûrement vraie ou presque sûrement fausse sur les données de taille $n$, lorsque $n$ devient grand (on parle alors de "lois 0-1"). Mais elles permettent aussi l'énumération de structures combinatoires et l'analyse d'algorithmes, même lorsque l'on ne sait pas calculer le dénominateur de (1). Par exemple, dans son livre intitulé *Random Graphs* [Bol85], Bollobás détermine [corollaire 17 p. 52] le nombre $L_r(n)$ de graphes $r$-réguliers (graphes non orientés ayant $n$ nœuds étiquetés, dont partent $r$ arêtes) :

$$L_r(n) \sim \sqrt{2} e^{-(r^2-1)/4} \left( \frac{r^{r/2}}{e^{r/2} r!} \right) n^{rn/2}.$$

Plus loin, il montre que le nombre $\chi_g$ de couleurs utilisées par l'algorithme du "glouton", pour colorier un graphe non orienté de $n$ points dont la probabilité de chaque arête est $1/2$, vérifie en moyenne $\chi_g \log_2 n/n \to 1$ presque sûrement [corollaire 11 p. 265]. Ces deux analyses ne sont pas possibles par les méthodes décrites dans cette thèse. La raison profonde est que les graphes sont essentiellement des structures *non décomposables*, alors que les méthodes combinatoires développées ici s'appliquent essentiellement à des structures *décomposables*.

Cependant, les structures de données utilisées en programmation (listes, ensembles, arbres) sont souvent *décomposables*. Le point de départ des méthodes combinatoires pour l'analyse en moyenne est l'égalité fondamentale (1) lorsque parmi les données de taille $n$, toutes sont équiprobables. Nous dirons alors que la distribution des données de même taille est uniforme. L'analyse en moyenne par un procédé combinatoire se ramène donc à compter d'une part le nombre de données de taille $n$, et d'autre part le coût cumulé de l'algorithme sur toutes ces données. Il existe principalement deux méthodes pour effectuer ces calculs : la *méthode des récurrences* et la *méthode des séries génératrices*.

## 1 La méthode des récurrences

La méthode des récurrences est celle que l'on utilise généralement pour analyser un programme *à la main*. Le premier à concevoir un système d'analyse automatique fut Wegbreit en 1975, et c'est à partir de cette méthode que son système fut bâti [Weg75]. Aussi certains auteurs parlent de la méthode de Wegbreit, notamment dans le contexte de l'analyse automatique [Zim90]. Le système de Wegbreit, METRIC, analyse des programmes simples écrits en Lisp. L'analyse se fait en trois phases. La première phase attribue des coûts aux opérations élémentaires, et calcule le coût des procédures non récursives. La deuxième phase traduit les procédures récursives en récurrences

pour le coût associé aux données de taille $n$. La troisième phase essaie de résoudre ces récurrences pour obtenir une forme exacte du coût moyen. METRIC est donc le premier analyseur automatique de programmes. En ce qui concerne l'automatisation, la première phase ne cause aucun problème puisqu'il suffit de remplacer les appels de procédure par leur coût symbolique, les séquences d'instructions par des sommes, les opérations élémentaires par des coûts donnés. La seconde phase est plus délicate car elle nécessite de déterminer les caractéristiques qui influent sur le coût du programme. Dans METRIC en effet, ces caractéristiques sont déterminées lors de l'analyse, mais les différentes possibilités sont fixées *a priori* : ce sont la longueur des listes (nombre d'éléments, atomes ou sous-listes) et leur taille (nombre d'atomes plus somme des tailles des sous-listes). Les équations de récurrence obtenues à l'issue de la seconde phase sont *conditionnelles* ou *non conditionnelles*, suivant qu'elles comprennent des tests (**if** ... **then** ... **else** ou **when**) ou non. Les équations non conditionnelles admettent des solutions exactes (le coût ne dépend que de $n$). Par contre, pour les équations conditionnelles, Wegbreit effectue d'un côté une analyse du meilleur et du pire des cas, et de l'autre une analyse en moyenne à l'aide de séries génératrices. Cette analyse dépend des probabilités de certaines conditions terminales, comme la probabilité qu'une liste soit vide, ou que son premier élément soit égal à une autre liste.

Les idées de Wegbreit ont été reprises récemment par mon homonyme allemand Wolf Zimmermann, et étendues à l'analyse de complexité d'une classe plus riche de programmes, d'où le nom de son système : COMPLEXA [Zim90]. COMPLEXA analyse des programmes écrits dans le langage STYFL (Simple TYped Functional Language), homologue typé de Lisp, et fournit leur coût en moyenne et dans le cas le pire. Les principaux atouts de COMPLEXA par rapport à METRIC sont

- la capacité de gérer des récurrences conditionnelles dont la condition dépend de $n$,
- l'analyse de dépendance (*dependency analysis*).

L'analyse de dépendance s'intercale entre la seconde et la troisième phase. Elle consiste à combiner de façon linéaire les équations de récurrence conditionnelles afin de faire disparaître les conditions. Ainsi, il est parfois possible de poursuivre l'analyse, comme par exemple dans le cas de l'algorithme *QuickSort*.

Lors de la troisième phase, il est nécessaire de posséder un catalogue d'équations de récurrences, ainsi que les solutions associées. Les capacités du système dépendent directement de la variété de ce catalogue. En outre, cette méthode s'applique bien à des structures de données *linéaires* comme les listes de LISP ; l'extension à d'autres types de structures (arbres, ensembles, graphes) ne semble pas facile. (Du point de vue analytique, des structures de données non linéaires induisent des récurrences elles-mêmes non linéaires, plus difficiles à traiter. Par exemple, la version actuelle du système de calcul formel MAPLE ne sait pas résoudre des équations quadratiques du genre $f_{n+1} = \sum_i f_i f_{n-i}, f_0 = 1$.) Un autre inconvénient majeur de cette méthode est le fait que les probabilités des conditions terminales sont obligatoirement constantes. Or la probabilité qu'une liste soit vide diffère généralement d'un point du programme à l'autre. Par contre, un des intérêts de cette méthode est le fait que les caractéristiques régissant le coût de l'algorithme sont déterminées dynamiquement lors de l'analyse.

Nous revenons ici sur le système ACE, qui est un autre descendant de METRIC, antérieur à COMPLEXA. ACE effectue l'analyse dans le cas le pire de programmes écrits dans le langage fonctionnel FP [Met88]. L'une des particularités d'ACE est que lors de la seconde phase (traduction en récurrences), on reste dans le langage FP : la description de l'algorithme et son analyse se font

dans un seul et même langage. Par exemple, la fonction factorielle s'écrit en FP

$$\text{fact} = \text{eq0} \rightarrow \text{"1"}; {}^{*}o[\text{id}, \text{fact } o \text{ sub1}],$$

et l'équation de sa complexité produite et simplifiée par le système est aussi une fonction FP :

$$\text{Cfact} = \text{eq0} \rightarrow \text{"0"}; \text{plus1 } o \text{ Cfact } o \text{ sub1},$$

qui donne après application du "principe d'induction récursive"

$$\text{Cfact} = \text{id},$$

ce qui signifie que la caractéristique est l'argument *id* de la fonction *fact* (c'est-à-dire l'entier dont on calcule la factorielle), et que la complexité est linéaire. D'autres résultats d'analyses effectuées par ACE sont : la concaténation de deux listes a un coût linéaire en la longueur de la première liste, le tri par sélection a un coût (dans le cas le pire) proportionnel au carré de la longueur de la liste des éléments à trier. Comme ces exemples le montrent, ACE fait une analyse grossière où les constantes de proportionnalité sont négligées.

Cependant, un grand avantage du système ACE est le principe sur lequel il est bâti : une routine principale (comme le moteur d'inférence des systèmes experts) qui utilise des règles bien précises (comme la base de règles d'un système expert). Par exemple, la simplification des programmes FP s'effectue à l'aide de 14 axiomes (SA1 à SA14), des transformations exactes sont faites à l'aide de 3 définitions récursives (D1 à D3), et la majoration des coûts utilise 7 types d'approximations (A1 à A7). Une telle séparation facilite l'extension du système (il suffit d'ajouter de nouvelles règles), ainsi que la preuve de correction du programme, tout en diminuant les risques d'erreur. Cette philosophie de séparation du principe de la méthode et des règles de transformation se retrouve dans cette thèse pour l'analyse en moyenne.

## 2 La méthode des séries génératrices

Cette méthode, dont la formulation dans le cadre de l'analyse d'algorithmes est due à Steyaert et Flajolet [FS81], utilise comme outil les séries génératrices[2]. Le principe de la méthode est le suivant (cf figure 3) : dans une première phase (*analyse algébrique*), on calcule à partir de la définition des structures de données, des procédures et de la mesure de complexité, un ensemble d'équations. Ces équations ont pour inconnues les séries génératrices de dénombrement des données et les séries génératrices de coût des procédures (appelées *descripteurs de complexité*). Dans une seconde phase (*analyse analytique*), on considère les séries génératrices comme des fonctions d'une variable complexe. Les deux sortes de séries sont donc traitées de la même façon. On extrait à partir des équations un développement asymptotique des coefficients des séries génératrices. Ce développement est suivant le cas celui du nombre de données de taille $n$ ou celui du coût total de la procédure sur celles-ci, et par conséquent une simple division donne le coût moyen asymptotique.

Les équations produites par l'analyse algébrique peuvent aussi être utilisées pour calculer le coût moyen du programme pour une valeur précise (par exemple $n = 100$) de la taille des données. Lorsque les équations se résolvent, les coefficients se calculent par simple développement de Taylor

[2]On attribue usuellement à Laplace l'usage systématique des séries génératrices comme outil en combinatoire. Pour une introduction aux séries génératrices, le lecteur pourra consulter [Wil90].

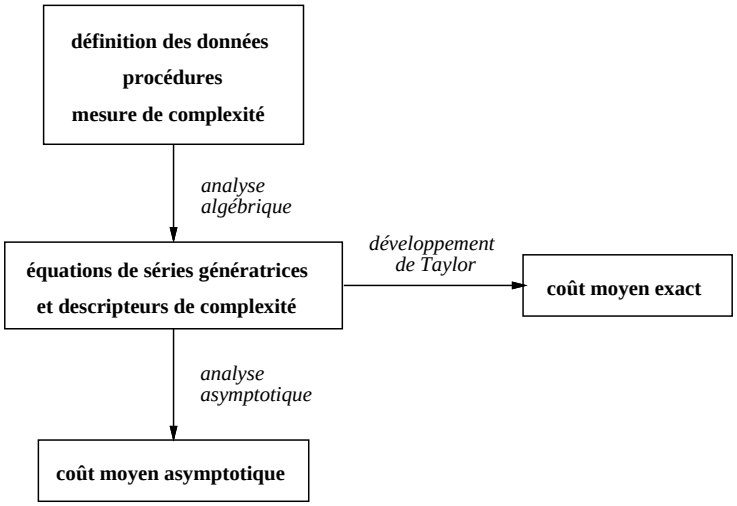



Figure 3 : Le principe de la méthode des séries génératrices.

des formes explicites. Sinon, il faut calculer le développement de Taylor de fonctions définies par une équation (par exemple par la méthode des coefficients indéterminés); cela est toujours possible du fait de l'origine combinatoire des équations.

## 2.1 Analyse algébrique

A chaque type de données est associée une série génératrice comptant le nombre de données de même taille, et à chaque procédure est associée une autre série génératrice, qui elle cumule le coût de la procédure sur les données de même taille. Le principe fondamental de l'analyse algébrique est que la définition des structures de données et des procédures se traduit *directement* en équations fonctionnelles pour ces séries génératrices, *sans passer par des formules de récurrences ou des formules exactes*.

L'intérêt de cette approche réside dans le fait que le coût moyen exact d'un algorithme a une forme souvent compliquée, comportant des sommations, alors que la série génératrice associée s'exprime simplement à l'aide des fonctions usuelles. Par exemple, la série génératrice de coût de l'algorithme *QuickSort* est

$$\tau Q(z) = \frac{2}{(1-z)^2} \log \frac{1}{1-z}. \tag{2}$$

tandis que l'expression exacte du coût moyen est

$$\tau \overline{Q}_n = 2 \sum_{i=1}^{n} \frac{n-i+1}{i}. \tag{3}$$

De plus, l'étude asymptotique s'effectue plus facilement sur la série génératrice (2), et conduit en général à un développement asymptotique simple :

$$\tau \overline{Q}_n = 2n \log n + O(n). \tag{4}$$

L'analyse algébrique (appelée aussi *méthode symbolique*) consiste à transformer la définition d'un algorithme en équations de séries génératrices telles que (2). La méthode symbolique regroupe des

méthodes classiques de manipulation de séries génératrices avec des principes issus de la combinatoire énumérative ; son originalité est de partir de correspondances systématiques entre constructions combinatoires et opérateurs de séries génératrices, et de généraliser ces correspondances aux programmes. Cette généralisation est due à J.-M. Steyaert et Ph. Flajolet [FS81]. Notamment, la thèse d'état de J.-M. Steyaert est intitulée *Structure et complexité des algorithmes* [Ste84]. L'analyse algébrique entretient également des liens étroits avec les travaux sur les séries et les langages formels (Chomsky, Schützenberger), en particulier leur spécialisation aux arbres (Berstel, Reutenauer). Un autre courant très proche est celui des espèces de structures inspiré par Joyal, et le système DARWIN [BC88] qui en est issu (Bergeron, Cartier).

### 2.2 Analyse asymptotique

Les séries génératrices sont donc un outil pratique, permettant de représenter une suite de nombres par une seule expression. L'analyse asymptotique montre que le comportement asymptotique de ces nombres se lit *directement* sur la série, considérée alors comme fonction d'une variable complexe. Par là même, le développement asymptotique (4) est obtenu à partir de (2) sans passer par la formule exacte (3), et par conséquent même lorsqu'il n'existe pas de formule exacte simple.

Plus précisément, soit $f(z) = \sum f_n z^n$ une fonction de la variable complexe $z$. Alors si $\rho$ est le rayon de convergence de $f$, les coefficients $f_n$ croissent exponentiellement suivant $1/\rho$ :

$$f_n \approx \rho^{-n}.$$

Le facteur de croissance exponentielle des coefficients de $f$ est donc directement lié à la position des singularités de plus petit module de $f$ (dites singularités *dominantes*). Mieux encore, l'étude du comportement local de $f$ au voisinage des singularités dominantes fournit les facteurs sous-exponentiels de croissance, par des théorèmes de transfert [FO90]. On distingue ainsi quatre phases dans l'algorithme d'analyse asymptotique [FSZ91] :

1. *Radius* : déterminer le rayon de convergence de $f$,
2. *Directions* : déterminer les directions des singularités dominantes,
3. *Expansion* : développer localement la fonction au voisinage des singularités dominantes,
4. *Transfer* : en déduire un développement asymptotique des coefficients $f_n$.

Une description détaillée de ces quatre phases est faite dans la thèse de B. Salvy [Sal91].

## 3 Contributions de cette thèse

Dans cette thèse, nous établissons l'existence de classes entières de programmes se prêtant à l'analyse algébrique, et nous indiquons des algorithmes efficaces pour effectuer cette analyse. Ainsi il est possible d'écrire un analyseur algébrique à partir des résultats évoqués ici. La démarche à suivre est la suivante : dans un premier temps on vérifie que le programme se trouve bien dans la classe des programmes que l'on sait analyser automatiquement (chapitre 1), et l'on s'assure à l'aide d'algorithmes donnés dans le même chapitre que le programme est bien fondé. Lorsque la vérification se termine de façon positive, on utilise les règles du chapitre 2 pour traduire le programme en équations de séries génératrices. Si le programme à analyser manipule des données étiquetées (par exemple des entiers), on utilise plutôt les règles du chapitre 3. Le chapitre 4 s'avère quant à lui utile

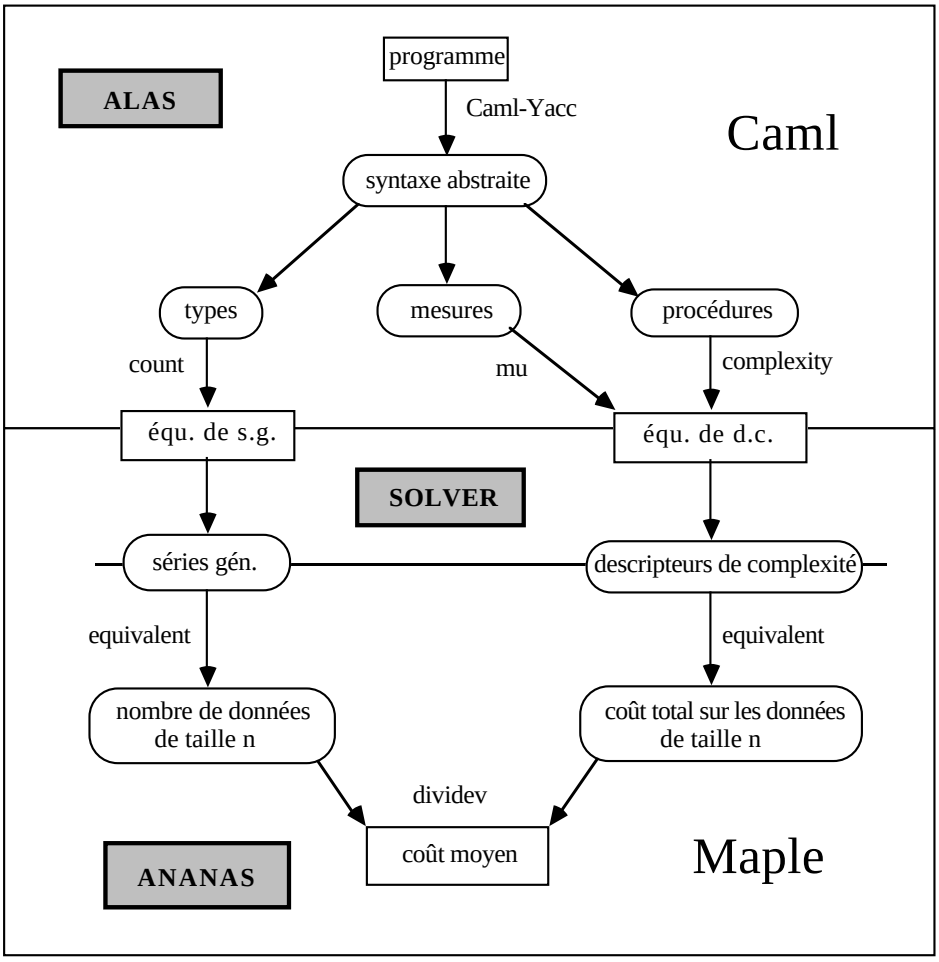



Figure 4 : La structure générale du système $\Lambda\Upsilon\Omega$.

pour l'analyse de programmes comprenant des fonctions à nombre fini de valeurs (par exemple des fonctions booléennes). Chacun de ces quatre chapitres contient de nombreux exemples d'analyse en moyenne, où l'application directe des règles donne le résultat, d'où la dénomination "théorème automatique". Le chapitre 5 est consacré plus spécialement à l'étude de véritables problèmes ; la plupart d'entre eux sont traités de façon *semi*-automatique, et mettent en évidence les limitations (ou les possibilités d'extension) de la méthode des séries génératrices.

Nous avons implanté les méthodes indiquées par cette thèse dans le système d'analyse automatique $\Lambda\Upsilon\Omega$, dont la figure 4 montre le schéma général. L'analyseur algébrique (ALAS), qui comprend à l'heure actuelle 2500 lignes dans le langage CAML, est constitué principalement de deux modules : `count` qui traduit les spécifications de types en équations pour les séries génératrices de dénombrement, et `complexity` qui traduit les définitions de procédures en équations de descripteurs de complexité, à l'aide des mesures de complexité (`mu`). Nous avons également réalisé un module de résolution des équations produites par l'analyse algébrique (SOLVER). Ce module combine les fonctionnalités des fonctions `solve` et `dsolve` de MAPLE pour essayer de déterminer une forme explicite des séries de dénombrement et des descripteurs de complexité, tout en éliminant les solutions ne

correspondant pas à des séries formelles. L'analyseur analytique (ANANAS) du système $\Lambda\Upsilon\Omega$ a été réalisé par B. Salvy ; il comprend à l'heure actuelle 14000 lignes dans le langage MAPLE. Les deux modules principaux ont `equivalent`, qui prend en entrée une série génératrice et produit en sortie un développement asymptotique de ses coefficients, et `dividev` qui effectue la division de développements asymptotiques.

Grâce à ces implantations, nous pourrons énoncer dans cette thèse des *théorèmes automatiques* et des *lemmes automatiques*. Nous convenons de qualifier un théorème (ou un lemme) d'*automatique* lorsque le résultat énoncé est obtenu par application de règles générales, comme celles formulées dans les chapitres 2, 3 et 4 de cette thèse. Le nombre et la diversité de ces théorèmes automatiques prouvent que l'analyse automatique d'algorithmes à l'aide de séries génératrices est un procédé effectif.

# Chapitre 1

# La classe de programmes $\Pi$

*Sapiens nihil affirmat quod non probat*

Considérons le programme suivant, qui détermine le nombre de sommants d'une partition d'entier, où une partition de $n$ est une suite d'entiers $(i_1, i_2, \ldots, i_k)$ telle que $1 \leq i_1 \leq \ldots \leq i_k$ et $i_1 + \cdots + i_k = n$ :

```
type partition = multiset(entier);
     entier = sequence(un, card >= 1);
     un = atom(1);

procedure compte_sommants (p : partition);
begin
   forall e in p do
      count
end;

measure count : 1;
```

Le but de ce chapitre est de donner une signification précise à un tel programme, tandis que le chapitre suivant déterminera son coût moyen. Ici, la partition $(1, 1, 2, 7)$ est représentée par l'objet $\{\!\{(\texttt{un}), (\texttt{un}), (\texttt{un}, \texttt{un}), (\texttt{un}, \texttt{un}, \texttt{un}, \texttt{un}, \texttt{un}, \texttt{un}, \texttt{un})\}\!\}$, et l'évaluation de la procédure `compte_sommants` sur cet objet provoque quatre appels de l'instruction élémentaire `count`. Comme l'objet atomique `un` est défini avec une taille 1, la taille de la représentation d'une partition $(i_1, \ldots, i_k)$ est exactement l'entier $n = i_1 + \cdots + i_k$, ici $n = 11$. D'autre part, comme on a fixé à 1 le coût de l'instruction `count`, le coût de l'évaluation de `compte_sommants` sur la représentation d'une partition est exactement le nombre $k$ de sommants de celle-ci, ici $k = 4$. Par conséquent, le coût moyen de `compte_sommants` sur les représentations de taille $n$ est exactement le nombre moyen de sommants des partitions de l'entier $n$.

Cet exemple met en évidence deux caractéristiques des langages de haut niveau comme Pascal, Lisp ou le langage C. Premièrement, les données d'un programme sont construites de façon *structurée* à l'aide de données élémentaires (entiers, caractères, ici l'atome `un`) et d'un petit nombre de modèles (enregistrements, listes, tableaux). Nous appellerons ces modèles des *constructeurs*. Ici, le constructeur *sequence*[1] fabrique des séquences et le constructeur *multiset* fabrique des multi-

[1] Les noms des constructeurs sont issus de la langue anglaise, et nous convenons d'indiquer les mots étrangers

ensembles (ensemble avec répétition). Deuxièmement, les procédures également sont écrites de manière structurée, à l'aide d'instructions élémentaires (lecture ou écriture d'un caractère, addition de deux mots-machine) et d'un petit nombre de *schémas* de programmation (exécution séquentielle, appel de sous-routine, boucles). Ici le schéma **forall** $e$ **in** $p$ **do** `count` permet d'effectuer l'instruction `count` pour chaque élément du multi-ensemble $p$ (c'est-à-dire chaque sommant de la partition associée).

Pour définir une classe de programmes (ici Π), il nous suffit donc de définir ses constructeurs et ses schémas de programmation. Mais avant d'analyser un programme, il faut d'abord s'assurer que ce programme n'est pas dégénéré, par exemple qu'il ne boucle pas pour certaines données, auquel cas le coût moyen ne serait pas défini. Il faut donc vérifier pour la classe de programmes dans laquelle nous nous plaçons, que le caractère bien fondé (non dégénéré) est décidable. Plutôt que de faire cette vérification uniquement pour la classe Π, nous dégageons des conditions suffisantes sur les constructeurs et les schémas de programmation pour que le caractère bien fondé soit décidable. Ces conditions sont évidemment remplies dans le cas de la classe Π.

Le plan de ce chapitre est le suivant : tout d'abord, la section 1.1 présente deux exemples mettant en évidence le genre de résultats obtenus à l'aide de cette thèse. La section 1.2 rappelle ce qu'est une grammaire, les différentes formes qu'elle peut prendre, et y ajoute une notion de taille. Nous définissons ensuite dans la section 1.3 les constructeurs qui permettront de fabriquer les données des programmes de Π (séquences, ensembles, cycles, etc). La section 1.4 donne des conditions suffisantes sur les constructeurs pour qu'une définition de données soit bien fondée, et fournit des algorithmes pour le décider. Cette section étant relativement technique, on pourra se contenter en première lecture du théorème 1, qui en est le résultat principal. Les schémas de programmation de la classe Π sont définis à la section 1.5, qui établit aussi que le caractère bien fondé et la terminaison des programmes sont décidables dans Π. Enfin, la section 1.6 définit le langage ADL qui permet d'écrire les programmes de Π sous une forme plus lisible (le programme donné en exemple ci-dessus est écrit dans ce langage).

## 1.1 Deux exemples introductifs

Notre but ici est de présenter à l'aide de deux exemples, d'une part le langage dans lequel sont définis les programmes et leurs données, d'autre part le genre d'analyses automatiques obtenues.

### Dérivation formelle

Voici d'abord un programme qui écrit la dérivée d'expressions formées à partir des entiers 0 et 1, de la variable $x$, et des opérateurs d'addition, de multiplication et d'exponentiation.

```
type expression = zero | un | x | product(plus,expression,expression)
                  | product(mult,expression,expression) | product(expo,expression);
     zero, un, x, plus, mult, expo = atom(1);

procedure diff (e : expression);
begin
  case e of
```

en caractères italiques : "*sequence*" est le mot anglais, sans accent, tandis que "séquence" est le mot français, avec accent.

```
      zero       : write(zero);
      un         : write(zero);
      x          : write(un);
      (plus,f,g) : begin write(plus); diff(f); diff(g) end;
      (mult,f,g) : begin write(plus);write(mult);diff(f);copy(g);write(mult);copy(f);diff(g) end;
      (expo,f)   : begin write(mult); write(expo); copy(f); diff(f) end;
    end
  end;

  procedure copy (e : expression);
  begin
    case e of
      zero       : write(zero);
      un         : write(un);
      x          : write(x);
      (plus,f,g) : begin write(plus); copy(f); copy(g) end;
      (mult,f,g) : begin write(mult); copy(f); copy(g) end;
      (expo,f)   : begin write(expo); copy(f) end;
    end
  end;

  measure write : 1;
```

La notation est préfixe : par exemple, l'expression mathématique

$$E = x + e^{(1+x)\cdot x}$$

est représentée par l'objet `plus x expo mult plus un x x`, du type `expression`. La taille de cet objet est 8 (ici la taille est le nombre de symboles ; la notion de taille sera définie plus précisément à la section 1.2).

La procédure `diff` prend en argument une expression, et écrit la représentation préfixe de sa dérivée par rapport à $x$, en appelant la procédure auxiliaire `copy` pour recopier une expression (ce qui est nécessaire pour dériver un produit ou une exponentielle), et l'instruction élémentaire `write` pour écrire chaque mot (`zero`, `un`, `x`, `plus`, `mult`, `expo`). Par exemple, la dérivation de l'expression $E$ ci-dessus produira comme résultat

`plus un mult expo mult plus un x x plus mult plus zero un x mult plus un x un`,

c'est-à-dire

$$E' = 1 + e^{(1+x)\cdot x} \cdot ((0+1)\cdot x + (1+x)\cdot 1).$$

La déclaration `measure write : 1` indique que chaque appel à `write` coûte une unité. Le coût de l'évaluation de `diff(e)` est par suite le nombre d'appels à `write` lors de la dérivation formelle de l'expression `e`, c'est-à-dire le nombre de mots de l'expression dérivée (20 pour notre exemple).

Voici le premier lemme automatique que nous rencontrons : nous dirons qu'un théorème (ou un lemme) est *automatique* lorsque le résultat énoncé est obtenu par application de règles générales comme celles formulées dans les chapitres 2 et 3 de cette thèse.

**Lemme automatique 1.** *Le coût moyen de la procédure* `diff` *sur les expressions de taille $n$ est*

$$\overline{\tau\mathrm{diff}}_n = \frac{[z^n]\tau\mathrm{diff}(z)}{[z^n]\mathrm{expression}(z)},$$

*où*

$$\tau\mathrm{diff}(z) = \frac{(1-2z-12z^2)\sqrt{1-2z-23z^2}-1+3z+34z^2}{4z(1-2z-23z^2)}, \tag{1.1}$$

$$\mathrm{expression}(z) = \frac{1-z-\sqrt{1-2z-23z^2}}{4z}, \tag{1.2}$$

*et $[z^n]f(z)$ est le coefficient de $z^n$ dans le développement de Taylor de $f$ à l'origine. En particulier, les premières valeurs sont $\overline{\tau\mathrm{diff}}_1 = 1$, $\overline{\tau\mathrm{diff}}_2 = 4$, $\overline{\tau\mathrm{diff}}_3 = 38/7$, $\overline{\tau\mathrm{diff}}_4 = 175/19$, $\overline{\tau\mathrm{diff}}_5 = 1237/109$.*

La preuve de ce lemme sera effectuée au chapitre 2, par application de règles énoncées dans le même chapitre, ce qui justifie le caractère "automatique". En outre, à partir des expressions (1.1) et (1.2), l'analyse de singularités permet d'obtenir, toujours automatiquement, un développement asymptotique du coût moyen $\overline{\tau\mathrm{diff}}_n$ :

**Lemme automatique 2.** *Le développement asymptotique suivant est valide :*

$$\frac{[z^n]\tau\mathrm{diff}(z)}{[z^n]\mathrm{expression}(z)} = \frac{(1+\sqrt{6})\sqrt{138\pi(12-\sqrt{6})}}{276}\,n^{3/2} + \frac{11(12-\sqrt{6})}{276}\,n + O(n^{1/2}).$$

La preuve de ce lemme découle des théorèmes énoncés dans la thèse de B. Salvy [Sal91]. La réunion de ces deux lemmes donne le coût moyen asymptotique de la procédure `diff`.

**Théorème automatique 1.** *Le coût moyen de la procédure* `diff` *sur les expressions de taille $n$ est*

$$\overline{\tau\mathrm{diff}}_n = \frac{(1+\sqrt{6})\sqrt{138\pi(12-\sqrt{6})}}{276}\,n^{3/2} + \frac{11(12-\sqrt{6})}{276}\,n + O(n^{1/2}).$$

### Graphes connexes monocycliques

Les graphes non orientés, connexes, et ayant un seul cycle (appelés ici graphes connexes monocycliques) sont des objets d'étude classique en théorie des graphes aléatoires [Bol85, page 113]. Le programme ci-dessous prend en entrée un tel graphe, et détermine la longueur de son unique cycle.

```
type c_u_graph = ucycle(tree);          % connected unicyclic graph %
     tree = product(node,set(tree));
     node = Latom(1);

procedure count_trees (g : c_u_graph);
begin
   forall t in g do
      count;
end;

measure count : 1;
```

**Lemme automatique 3.** *La longueur moyenne de l'unique cycle d'un graphe connexe monocyclique de $n$ nœuds est*

$$\overline{l}_n = \frac{[z^n]\tau\mathrm{CT}(z)}{[z^n]G(z)}$$

*où*

$$G(z) = \frac{1}{2}[\log\frac{1}{1-f(z)} + f(z) + \frac{1}{2}f(z)^2] \quad et \quad \tau\mathrm{CT}(z) = \frac{1}{2}[\frac{f(z)}{1-f(z)} + f(z) + f(z)^2] \tag{1.3}$$

*et $f(z)$ est la série solution de l'équation*

$$f(z) = ze^{f(z)}. \tag{1.4}$$

La preuve de ce lemme sera faite au chapitre 3. L'analyse asymptotique des expressions (1.3) et (1.4) est possible de manière automatique.

**Lemme semi-automatique 4.** *Le développement asymptotique suivant est valide :*

$$\frac{[z^n]\tau\mathrm{CT}(z)}{[z^n]G(z)} = \sqrt{\frac{2}{\pi}}n^{1/2} + O(1).$$

Ce lemme n'est que semi-automatique à cause de certaines insuffisances en l'état actuel du système de calcul formel MAPLE ; par contre, comme on le constatera (page 95), le reste de la preuve découle de manière systématique de la théorie asymptotique développée dans [Sal91].

Le lecteur attentif aura remarqué que dans le programme de dérivation formelle, les données élémentaires sont définies à l'aide du constructeur `atom`, alors que dans le programme sur les graphes, les nœuds sont définis par le constructeur `Latom`. Cette légère différence modifie complètement la nature des structures de données : le mot `Latom` (abréviation de *Labelled atom* en anglais) crée des atomes portant en plus une étiquette. Lorsque les données élémentaires sont définies par le constructeur `atom` (resp. `Latom`), nous dirons que nous sommes en univers non étiqueté (resp. étiqueté). Le présent chapitre définit une classe $\Pi$ de programmes sur des objets non étiquetés, qui s'analysent par les règles du chapitre 2, tandis que le chapitre 3 définit une classe $\hat{\Pi}$ de programmes sur des objets étiquetés, et établit les règles d'analyse associées.

## 1.2 Mots, langages et grammaires

Cette section étend le concept classique de grammaire (sur des données linéaires ou *mots*) à un concept plus général à l'aide de la notion de *constructeur* (section 1.2.1). Puis nous montrons qu'on retrouve les grammaires classiques avec les constructeurs *union* et *produit cartésien* (section 1.2.2). Enfin, nous dotons les grammaires d'une notion de taille (section 1.2.4). Ces grammaires étendues, que nous appelons *spécifications*, constituent le cadre théorique de l'étude menée aux sections 1.3 (définition d'une classe particulière) et 1.4 (vérification du caractère bien fondé).

### 1.2.1 Grammaires, constructeurs et dérivation

Nous adopterons la définition usuelle suivante pour les grammaires.

**Définition 1.** *Une grammaire est un quadruplet* $(\mathcal{T}, \mathcal{N}, S, \mathcal{P})$ *comprenant*

$$\begin{cases} \textit{un ensemble } \mathcal{T} \textit{ de} \text{ terminaux}, \\ \textit{un ensemble } \mathcal{N} \textit{ de} \text{ non-terminaux}, \\ \textit{un symbole d'entrée } S \in \mathcal{N}, \\ \textit{un ensemble } \mathcal{P} \textit{ de productions.} \end{cases}$$

Nous utiliserons généralement des lettres minuscules pour représenter les terminaux (dits aussi *atomes*) et des lettres majuscules pour les non-terminaux. Les productions de la grammaire servent à *construire* des objets complexes à partir d'objets élémentaires (les atomes) : elles sont de la forme

$$U \to a \quad \text{ou} \quad U \to V$$

où $a$ est un terminal, $U$ et $V$ des non-terminaux, conformément à notre convention sur les majuscules, ou bien de la forme

$$U \to \Phi(R_1, \ldots, R_k)$$

où $\Phi$ est un *constructeur* et $R_1$, ..., $R_k$ des non-terminaux.

**Définition 2.** *Un* constructeur *est une règle* $\Phi$ *qui, à partir d'une séquence de* $k$ *objets* $(r_1, \ldots, r_k)$ *construit un nouvel objet, noté* $\Phi(r_1, \ldots, r_k)$*. Si* $\mathcal{R}_1, \ldots, \mathcal{R}_k$ *sont des ensembles d'objets, on note* $\Phi(\mathcal{R}_1, \ldots, \mathcal{R}_k)$ *l'ensemble construit par* $\Phi$*, au sens usuel de l'extension des opérateurs aux ensembles :*

$$\Phi(\mathcal{R}_1, \ldots, \mathcal{R}_k) \stackrel{\text{def}}{=} \bigcup_{r_1 \in \mathcal{R}_1, \ldots, r_k \in \mathcal{R}_k} \{\Phi(r_1, \ldots, r_k)\}.$$

*Lorsque* $k$ *peut prendre plusieurs valeurs, on dit que* $\Phi$ *est un* multi-constructeur *(sinon, c'est un constructeur* d'arité fixe*) et on note alors,* $\mathcal{R}$ *étant un ensemble d'objets,*

$$\Phi(\mathcal{R}) \stackrel{\text{def}}{=} \bigcup_{k \in \mathbb{N}} \Phi(\underbrace{\mathcal{R}, \ldots, \mathcal{R}}_{k}).$$

*Les objets* $r_j$ *sont appelés* composantes directes *de* $\Phi(r_1, \ldots, r_k)$*. Les composantes au sens large d'un objet sont lui-même, ses composantes directes, et les composantes au sens large de celles-ci ; les composantes au sens strict sont les composantes au sens large, excepté l'objet lui-même.*

Nous avons déjà rencontré des constructeurs dans les trois programmes donnés en exemple au début de ce chapitre : le constructeur d'arité fixe *product* formant des produits, que nous avons utilisé pour définir le type `expression` du programme de dérivation formelle, les multi-constructeurs *multiset* formant des multi-ensembles, *sequence* formant des séquences, *ucycle* formant des cycles non orientés, et *set* formant des ensembles.

Le produit cartésien est un constructeur d'arité fixe $k = 2$ : à partir de deux ensembles $\mathcal{A}$ et $\mathcal{B}$, il permet de définir l'ensemble produit $\mathcal{A} \times \mathcal{B}$. Le constructeur *sequence*, que nous définirons précisément plus loin, est un multi-constructeur formant des séquences d'objets : si $k \neq 0$, $\Phi(r_1, \ldots, r_k) = (r_1, \ldots, r_k)$, et si $k = 0$, le constructeur *sequence* engendre la séquence vide (sans

élément), que nous noterons (). Il faut voir un constructeur comme une façon d'assembler des objets. Le produit cartésien les assemble de manière linéaire, en les ordonnant. Le constructeur *ensemble* défini plus loin assemble les objets sans notion d'ordre. Le constructeur *cycle* forme des cycles d'objets. Un multi-constructeur est en quelque sorte la réunion de constructeurs d'arité fixe formant des objets de même espèce (par exemple, le constructeur "séquence" est la réunion des constructeurs "séquence de $k$ objets", pour $k \in \mathbb{N}$).

Un objet $w$ est *dérivé* d'un non-terminal $W$ si

1. les composantes terminales de $w$ sont dans $\mathcal{T}$,
2. il existe une suite de productions de $\mathcal{P}$ qui, partant de $W$, construisent $w$.

L'ensemble des mots dérivés du symbole d'entrée $S$ est le *langage* décrit par la grammaire. (Nous ne faisons ici qu'étendre à des constructeurs quelconques la notion classique de dérivation utilisée pour les langages *context-free* [AU72].) Par exemple, la grammaire[2]

$$\begin{cases} \mathcal{T} & = & \{\,'(', ')', \epsilon\} \\ \mathcal{N} & = & \{S, R\} \\ P_1 & : & S \to \epsilon \\ P_2 & : & S \to RS \\ P_3 & : & R \to (S) \end{cases}$$

(la juxtaposition représente la concaténation des mots) décrit le langage des mots correctement parenthésés ; ainsi, le mot $()(())$ est dérivé de la manière suivante

$$\begin{array}{rcl} S & \xrightarrow{P_2} & RS \\ & \xrightarrow{P_3 \text{ et } P_2} & (S)RS \\ & \xrightarrow{P_1, P_3 \text{ et } P_1} & ()(S) \\ & \xrightarrow{P_2} & ()(RS) \\ & \xrightarrow{P_3 \text{ et } P_1} & ()((S)) \\ & \xrightarrow{P_1} & ()(()). \end{array}$$

Plus le nombre de constructeurs autorisés dans les productions est grand, plus variés sont les objets dérivés, et donc plus nombreux les langages que l'on peut décrire. Par exemple, avec comme seuls constructeurs l'union et le produit cartésien, nous obtenons des langages dits *context-free*, qui font l'objet de la prochaine sous-section.

### Propriété de croissance

Il découle de la définition 2 qu'un constructeur est *croissant* au sens suivant.

**Corollaire 1.** *Si $\Phi$ est un constructeur d'arité fixe $k$, alors*

$$\mathcal{R}_1 \subset \mathcal{T}_1, \ldots, \mathcal{R}_k \subset \mathcal{T}_k \quad \Longrightarrow \quad \Phi(\mathcal{R}_1, \ldots, \mathcal{R}_k) \subset \Phi(\mathcal{T}_1, \ldots, \mathcal{T}_k).$$

*Si $\Phi$ est un multi-constructeur, alors*

$$\mathcal{R} \subset \mathcal{T} \quad \Longrightarrow \quad \Phi(\mathcal{R}) \subset \Phi(\mathcal{T}).$$

[2] Dans la théorie des langages formels, $\epsilon$ représente usuellement le mot vide.

### 1.2.2 Grammaires context-free

Une grammaire est dite *context-free* (indépendante du contexte) lorsque les constructeurs autorisés sont l'union

$$T \to A \mid B$$

et le produit cartésien

$$T \to A\,B,$$

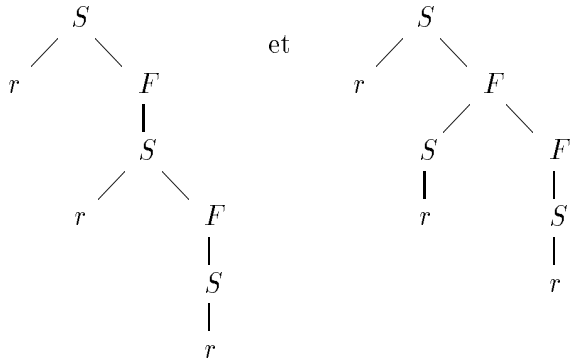
où $A$ et $B$ peuvent être des terminaux ou des non-terminaux. De plus, le nombre de termes peut être quelconque (pas forcément 2). La grammaire ci-dessous, qui engendre tous les mots de la forme $r^n$, pour $n \geq 1$, est *context-free*.

$$(G)\begin{cases} S \to r & (a) \\ S \to rF & (b) \\ F \to S & (c) \\ F \to SF & (d) \end{cases}$$

NOTE : les langages (ensembles de mots) générés par les grammaires *context-free* sont qualifiés également de *context-free*. On utilise aussi le terme *algébrique*, avec la même signification, pour les grammaires et pour les langages ; dans cette thèse, nous choisissons d'employer le terme *context-free* dans les deux cas. Les langages non *context-free* sont donc ceux qui ne sont générés par aucune grammaire *context-free* : par exemple le classique langage $L_3 = \{a^n b^n c^n, n \in \mathbb{N}\}$ [AU77].

#### Grammaires ambiguës

La grammaire $(G)$ définie ci-dessus est *ambiguë* : des arbres de dérivation différents produisent le même mot. Par exemple, les deux arbres de dérivation




engendrent le même mot $rrr$. Dans notre exemple, l'ambiguïté provient de la production $F \to SF$, car $S$ et $F$ dérivent tous deux des mots de la forme $r^n$. Il suffit pour la lever d'entourer les mots dérivés de $F$ (par exemple) par des *marqueurs*. Remplaçons la production $(d)$ par

$$F \to xSFy. \qquad (d')$$

Avec cette nouvelle production, les deux arbres de dérivation ci-dessus engendrent respectivement

$$rrr \qquad \text{et} \qquad rxrry,$$

et l'ambiguïté a disparu. Les programmes que nous définirons opéreront sur les *arbres de dérivation*, et non sur les mots. Ainsi, un objet ayant plusieurs arbres de dérivation comptera pour autant

d'objets différents. La charge incombe à l'utilisateur de vérifier que la distribution des données ainsi définie correspond bien à celle qu'il désire modéliser.

### 1.2.3 Différentes formes d'une grammaire

#### Forme compacte et forme développée

Lorsqu'il existe plusieurs dérivations possibles pour un non-terminal $A$, nous pouvons représenter cela sous forme *compacte* :

$$A \to B \mid C \mid D,$$

ou sous forme *développée* :

$$\begin{aligned} A &\to B \\ A &\to C \\ A &\to D. \end{aligned}$$

Par extension, une grammaire est dite sous forme compacte lorsque chaque non-terminal apparaît dans le membre gauche d'une seule production, et sous forme développée lorsqu'aucun membre droit ne contient le symbole d'union "|".

#### Chomsky Normal Form

Une grammaire est dite en *Chomsky Normal Form (CNF)* lorsque tous les constructeurs d'arité fixe (par exemple le produit cartésien) sont unaires ou binaires (cf définition 2). Par exemple, toute grammaire *context-free* peut se mettre sous cette forme [AU72]. Il suffit pour cela de remplacer les productions

$$A \to B_1 \mid \cdots \mid B_k$$

où $k > 2$ par $k-1$ productions binaires, en introduisant de nouveaux non-terminaux $A_2, \ldots, A_{k-1}$ :

$$\begin{aligned} A &\to B_1 \mid A_2 \\ &\cdots \\ A_i &\to B_i \mid A_{i+1} \\ &\cdots \\ A_{k-1} &\to B_{k-1} \mid B_k \end{aligned}$$

et de faire de même pour les produits cartésiens.

### 1.2.4 Grammaires avec notion de taille

Les grammaires nous permettent de définir des ensembles d'objets. Ces objets seront les données du programme à analyser. Notre objectif est d'obtenir des analyses de complexité en moyenne à un paramètre, comme :

*Le coût moyen de la procédure* `diff` *sur les expressions de taille $n$ est*

$$\overline{\tau\mathrm{diff}}_n = \frac{(1+\sqrt{6})\sqrt{138\pi(12-\sqrt{6})}}{276}\, n^{3/2} + \frac{11(12-\sqrt{6})}{276}\, n + O(n^{1/2}).$$

Par conséquent, il faut définir dans chaque programme, la *taille* des données. Nous utilisons la définition suivante :

**Définition 3.** *Une spécification de structures de données est un quadruplet* $(\mathcal{T}, \mathcal{N}, \mathcal{P}, |.|)$*, telle que* $(\mathcal{T}, \mathcal{N}, S, \mathcal{P})$ *est une grammaire pour un certain non-terminal* $S$*, et* $|.|$ *est une fonction de* $\mathcal{T}$ *dans* $\mathbb{N}$ *dite fonction de taille. Ainsi la taille d'un terminal* $a$ *est l'entier* $|a|$ *donné par la spécification. Si* $d$ *est une structure de donnée composée, sa taille, qui est notée aussi* $|d|$*, est la somme des tailles de ses composantes directes.*

De cette définition découlent immédiatement deux propriétés :

- la taille d'une donnée est un entier positif ou nul,
- la taille est *additive* : pour tout constructeur $\Phi$, $|\Phi(r_1, \ldots, r_k)| = |r_1| + \cdots + |r_k|$.

Ces deux propriétés fondamentales seront indispensables pour l'analyse en moyenne d'algorithmes : le caractère discret permet d'utiliser comme outil de calcul les séries génératrices, et le caractère additif permet d'obtenir des opérateurs simples pour les constructions ensemblistes. Ainsi, les méthodes développées ici ne sont pas utilisables lorsque le paramètre d'analyse (la taille) n'est pas additif.

REMARQUE : La propriété d'additivité étant vraie pour tout $k$, elle est vraie en particulier pour $k = 0$, et s'exprime alors sous la forme $|\Phi()| = 0$. L'objet $\Phi()$, s'il existe, est l'objet sans composante dérivé par $\Phi$ ; par exemple, nous avons déjà vu que le constructeur *sequence* dérive la séquence vide (), et nous verrons plus loin que le constructeur *ensemble* dérive l'ensemble vide $\{\}$. La taille des objets sans composante est par conséquent nulle : $|()| = |\{\}| = 0$.

## 1.3 La classe Ω

Les grammaires classiques (par exemple *context-free*) produisent des données linéaires (ou mots). Dans le langage PASCAL, cela correspond aux types définis par des enregistrements (**record**), avec ou sans variante (**case**). Dans le langage C, ce sont les types définis à l'aide de **struct** et **union**. Or les données manipulées par un programme sont en général plus complexes : par exemple un programme de complétion opère sur des ensembles, un algorithme de fermeture transitive sur des graphes. Ceci nous a amenés à définir dans la précédente section un concept plus général de grammaire, le concept de spécification, basé sur les notions de constructeur et de taille. Nous définissons dans cette section une classe particulière de spécifications, sur laquelle nous définirons par la suite des programmes.

### 1.3.1 Les constructeurs

Nous introduisons ici des constructeurs autres que l'union et le produit cartésien, afin de pouvoir former des séquences, des ensembles ou des cycles. Grâce aux définitions 1 à 3, ces nouveaux constructeurs induisent une classe de spécifications : Ω.

#### Séquences

Le constructeur *sequence* forme des séquences d'objets. Dans une grammaire, nous utilisons comme syntaxe

$$A \to B^*$$

pour indiquer que le non-terminal $A$ engendre toutes les séquences d'éléments dérivés de $B$. Nous appelons *longueur* d'une telle séquence, et nous notons $l(\cdot)$, le nombre d'éléments dérivés de $B$ qu'elle contient. Nous notons les séquences à l'aide de parenthèses, en séparant les éléments par des virgules : $(a, b, a)$ représente la séquence de trois éléments dont le premier est $a$, le second $b$ et le troisième $a$ ; la séquence vide est notée $()$. Cette représentation n'introduit pas d'ambiguïté, à condition de ne pas utiliser les parenthèses ni la virgule comme symboles terminaux.
REMARQUE : Les deux productions

$$\begin{array}{rcl} A & \rightarrow & () \mid (B') \\ B' & \rightarrow & B \mid B\,,\,B' \end{array} \tag{1.5}$$

engendrent exactement les mêmes objets que la production $A \rightarrow B^*$. L'ajout du constructeur *sequence* à des grammaires utilisant déjà les constructeurs union et produit cartésien n'augmente donc pas leur puissance d'expression ; c'est simplement une commodité d'écriture.

Un corollaire de cette remarque est que les langages dérivés par des grammaires utilisant les constructeurs union, produit cartésien, séquence sont *context-free*.

### Ensembles et multi-ensembles

La production

$$A \rightarrow \mathcal{P}(B)$$

signifie que le non-terminal $A$ engendre tous les ensembles finis d'éléments dérivés de $B$. Nous noterons les ensembles à l'aide d'accolades : $\{a, b\}$ représente l'ensemble composé de deux éléments, $a$ et $b$ ; l'ensemble vide est noté $\{\}$. Ici, il n'y a pas de description équivalente sous forme de grammaire *context-free* : le constructeur ensemble enrichit réellement la classe d'objets que l'on peut créer.

Lorsque l'on autorise le même élément à apparaître plusieurs fois dans un ensemble, on définit alors un ensemble avec répétition ou *multi-ensemble*. La production correspondante est

$$A \rightarrow \mathcal{M}(B)$$

et un multi-ensemble est noté par des accolades doubles $\{\!\{$ et $\}\!\}$. Par exemple, $\{\!\{a, b, b, a, b\}\!\}$ représente le multi-ensemble comprenant deux fois la lettre $a$ et trois fois la lettre $b$ ; le même multi-ensemble sera noté aussi $\{\!\{a^2, b^3\}\!\}$ (puisqu'il n'y a pas d'ordre, nous pouvons regrouper les objets identiques). De même, $\{\!\{\}\!\}$ représente le multi-ensemble vide ; il ne faut pas le confondre avec $\{\{\}\}$ qui est l'ensemble ayant comme seul élément l'ensemble vide.

### Cycles

Les cycles sont essentiellement des séquences définies à une permutation circulaire près. Nous considérons ici uniquement des cycles *orientés* représentés par des crochets [ et ]. La notation $[x_1, \ldots, x_k]$ représente le graphe comportant un arc de $x_1$ vers $x_2$, de $x_2$ vers $x_3$, ..., de $x_{k-1}$ vers $x_k$ et de $x_k$ vers $x_1$. Par exemple, $[a, b, c]$, $[b, c, a]$ et $[c, a, b]$ représentent le même cycle orienté, en l'occurrence le cycle $A$ de la figure 1.1. Le cycle orienté $B$ est représenté indifféremment par $[a, c, b]$ ou $[c, b, a]$ ou $[b, a, c]$. Ce sont les deux seuls cycles orientés sur l'ensemble $\{a, b, c\}$. Les cycles orientés sont dérivés par le constructeur $\mathcal{C}$ :

$$A \rightarrow \mathcal{C}(B).$$

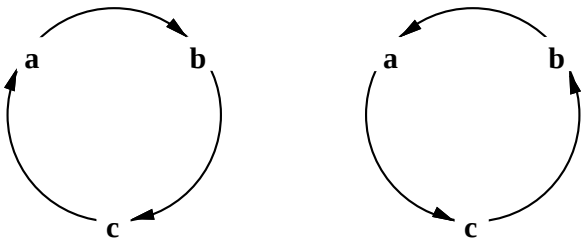



Figure 1.1 : Les deux cycles orientés $A = [a, b, c]$ et $B = [a, c, b]$ sur l'ensemble $\{a, b, c\}$

REMARQUE : il n'existe pas de cycle vide (sans élément). Le plus petit cycle au sens du nombre d'éléments est donc $[a]$.

Le tableau ci-dessous résume les nouvelles constructions et leurs notations.

| constructeur | notations | |
|---|---|---|
| séquence | $(\cdot)^*$ | $(a, b, a)$ |
| ensemble | $\mathcal{P}(\cdot)$ | $\{a, b\}$ |
| multi-ensemble | $\mathcal{M}(\cdot)$ | $\{\!\{a^2, b\}\!\}$ |
| cycle orienté | $\mathcal{C}(\cdot)$ | $[a, b, c]$ |

Il ne nous reste plus qu'à définir la classe des spécifications utilisant ces nouvelles constructions.

**Définition 4.** *La classe $\Omega$ est l'ensemble des spécifications en univers non étiqueté dont les productions utilisent les constructeurs union, produit cartésien, séquence, ensemble, multi-ensemble et cycle orienté.*

Nous avons précisé dans cette définition que l'univers (ensemble des objets que l'on peut créer) est *non étiqueté* ; le chapitre 3 définira une classe analogue à $\Omega$ en univers étiqueté.

Avant de définir des programmes sur des spécifications de $\Omega$, il nous faut vérifier que celles-ci sont bien fondées.

## 1.4 Spécifications bien fondées

Parmi les spécifications que l'on peut définir, certaines sont dégénérées, et ne se prêtent pas à l'analyse en moyenne d'algorithmes. La première phase de l'analyse d'un programme consiste donc à vérifier que la spécification des structures de données est *bien fondée* (non dégénérée). Dans la suite de cette section, nous définissons ce qu'est une spécification bien fondée (définition 5), nous montrons que pour un certain type de constructeurs (constructeur idéal), le caractère bien fondé est décidable, et nous exhibons un algorithme pour le décider (théorème 1).

Considérons la spécification

$$(\Sigma_1)\begin{cases} \mathcal{T} & = & \{a\} \\ \mathcal{N} & = & \{S, R\} \\ \mathcal{P} & : & S \to a \mid R \qquad R \to aS \\ |.| & : & |a| = 0 \end{cases}$$

Cette spécification décrit les mots $a \ldots a$ formés à partir de la seule lettre $a$. Comme la taille de $a$ est nulle, tous les mots sont de taille nulle. Le nombre de données de taille 0 décrites par cette spécification étant infini, nous ne pouvons pas les dénombrer. D'autre part, si le nombre de données de taille $n$ est nul pour tout $n$, le coût moyen n'est jamais défini. Pour éviter de tels cas dégénérés, nous introduisons la définition suivante.

**Définition 5.** *La* valuation *d'un non-terminal est la plus petite taille des objets qu'il dérive :*

$$val(X) \stackrel{\text{def}}{=} inf\{|x|, X \text{ dérive } x\}.$$

*Une spécification* $(\mathcal{T}, \mathcal{N}, \mathcal{P}, |.|)$ *est dite* bien fondée[3] *si pour chaque non-terminal* $X \in \mathcal{N}$ *:*

*(i) la* valuation *de* $X$ *est finie,*

*(ii) pour tout* $n \in \mathbb{N}$*, l'ensemble* $\mathcal{X}_n$ *des objets de taille* $n$ *dérivés de* $X$ *est fini.*

Par convention, lorsque $X$ est un non-terminal, nous utiliserons la lettre majuscule calligraphiée $\mathcal{X}$ pour l'ensemble des objets dérivés de $X$, et $\mathcal{X}_n$ pour l'ensemble des objets de taille $n$ ; lorsque $\mathcal{X}_n$ est fini, son cardinal sera noté $X_n$ ou $x_n$.

**Corollaire 2.** *Pour tout non-terminal* $U$ *d'une spécification bien fondée, il existe une suite d'entiers* $(u_n)$ *tels que* $u_n = card(\mathcal{U}_n) < +\infty$*, où* $\mathcal{U}_n = \{u \in \mathcal{U}, |u| = n\}$.

Par exemple, la production $T \to TT$ ne dérive à elle seule aucun objet de taille finie : une spécification contenant cette production, et aucune autre avec $T$ en membre gauche, n'est pas bien fondée, car la condition $(i)$ n'est pas vérifiée. D'autre part, la spécification $(\Sigma_1)$ définie ci-dessus n'est pas bien fondée, puisque $\mathcal{S}_0 = \{a^n, n \geq 1\}$ est infini : cette fois-ci, c'est la condition $(ii)$ qui est violée. Par contre, la spécification

$$(\Sigma_2) \left\{ \begin{array}{lcll} \mathcal{T} & = & \{a\} & \\ \mathcal{N} & = & \{S, R\} & \\ \mathcal{P} & : & S \to a \mid R \qquad R \to aS & \\ |.| & : & |a| = 1 & \end{array} \right.$$

est bien fondée ($\text{card}(\mathcal{S}_1) = 1$ et $\text{card}(\mathcal{S}_n) = \text{card}(\mathcal{R}_n) = 1$ pour $n \geq 2$). Or la seule différence entre $(\Sigma_1)$ et $(\Sigma_2)$ est la taille de $a$ qui vaut respectivement 0 et 1.

Avant d'analyser un programme, il faudra décider (en un temps fini) si la spécification de ses données est bien fondée ou non. Pour les grammaires *context-free*, on sait déjà décider si le langage dérivé d'un non-terminal est vide ou non [AU72, algorithme 2.7 page 144]. Cela nous permet de savoir si la valuation de ce non-terminal est finie. Pour les grammaires *context-free* avec en plus une notion de taille, D. H. Greene [Gre83, pages 69-70] définit les mots "pseudo vides" (*pseudo empty strings*) comme étant ceux dont la taille est nulle. Pour déterminer les non-terminaux dérivant des mots pseudo vides, il propose d'utiliser les algorithmes de Aho et Ullman, en particulier celui qui détermine si un non-terminal dérive le mot vide $\epsilon$ (c'est-à-dire un mot de taille nulle lorsqu'il y a une notion de taille). Ensuite, pour calculer les $u_n = \text{card}(\mathcal{U}_n)$, il définit une relation d'ordre partiel sur les non-terminaux : $N_i \prec N_j$ s'il existe une production $N_j \to N_i N_k$ (ou $N_j \to N_k N_i$) où $N_k$

[3] Le terme "bien fondé" est utilisé dans d'autres domaines : en mathématiques, un ordre (partiel ou total) est bien fondé s'il ne possède pas de chaîne infinie descendante ; on utilise aussi cette définition pour les ordres de réduction sur les règles de réécriture [JL86].

dérive un mot pseudo vide. Greene ajoute que s'il existe un cycle dans cette relation d'ordre partiel, alors le dénombrement n'est pas bien défini.
NOTE : Le caractère bien fondé d'une spécification s'interprète aussi au niveau des séries génératrices associées, qui seront définies au chapitre suivant. Il implique alors qu'il existe une et une seule solution au système d'équations vérifiées par les séries génératrices. Le caractère bien fondé d'une spécification équivaut donc en quelque sorte aux conditions du théorème des fonctions implicites pour les séries associées. Ce théorème a été énoncé dans le cadre des structures combinatoires par André Joyal [Joy81, théorème 4 page 37].

L'objet de ce qui suit est d'étendre les résultats de Greene à une classe plus vaste de grammaires, contenant les grammaires *context-free*, tout en proposant les algorithmes de décision correspondants. La vérification du caractère bien fondé s'effectue en deux temps : premièrement on s'assure que chaque non-terminal dérive au moins un objet de taille finie (condition $(i)$), puis on vérifie que le nombre d'objets de taille bornée est fini (condition $(ii)$).

### 1.4.1 Constructeur idéal

Dans cette section, nous définissons la notion de constructeur *idéal* ; tous les constructeurs que nous avons vus jusqu'ici (séquence, ensemble, multi-ensemble, cycle) sont idéaux. Le résultat principal est le théorème 1, dont une conséquence est que le caractère bien fondé est décidable dans $\Omega$ (corollaire 3). Cette section étant assez technique, il est conseillé, en première lecture au moins, de passer directement à la section suivante (page 35), qui définit des programmes sur $\Omega$.

**Définition 6.** *Un constructeur* $\Phi$ *est* idéal *si c'est le constructeur* union *ou* produit cartésien, *ou si c'est un multi-constructeur qui vérifie en plus les conditions suivantes :*

1. *il existe une fonction* $f_\Phi : \mathbb{N} \cup \{\infty\} \to \mathbb{N} \cup \{\infty\}$ *telle que* $\forall Y,\ val(\Phi(Y)) = f_\Phi(val(Y))$,
2. *(2a)* $(\forall Y, val(Y) = 0 \Rightarrow card(\Phi(\mathcal{Y})) = \infty)$ *ou* $(2b)(\forall Y, card(\mathcal{Y}) < \infty \Rightarrow card(\Phi(\mathcal{Y})) < \infty)$,
3. *soient* $P(n)$ *et* $Q(n)$ *les deux assertions (exclusives) suivantes,*
   $\begin{cases} P(n) : \textit{tout objet de taille } n \textit{ dérive par } \Phi \textit{ un objet de taille } n, \\ Q(n) : \textit{aucun objet de taille } n \textit{ dérivé par } \Phi \textit{ n'a de composante de taille } n \ ; \end{cases}$
   *pour chaque entier* $n$, *l'une des deux assertions est vraie :*
   $$\forall n \geq 0, (P(n) \textit{ ou } Q(n)),$$
   *et pour* $n$ *assez grand, c'est toujours la même qui est vraie :*
   $$\exists N_\Phi, (\forall n \geq N_\Phi, P(n)) \textit{ ou } (\forall n \geq N_\Phi, Q(n)).$$

REMARQUE 1 : la fonction $f_\Phi$ est nécessairement croissante, à cause de la croissance du constructeur $\Phi$ (corollaire 1) : si $\mathcal{Y} \subset \mathcal{Z}$, alors d'un côté $\mathrm{val}(Z) \leq \mathrm{val}(Y)$ et de l'autre $\Phi(\mathcal{Y}) \subset \Phi(\mathcal{Z})$ donc $\mathrm{val}(\Phi(Z)) = f_\Phi(\mathrm{val}(Z)) \leq \mathrm{val}(\Phi(Y)) = f_\Phi(\mathrm{val}(Y))$.
REMARQUE 2 : les deux assertions (2a) et (2b) sont mutuellement exclusives : il suffit de prendre $\mathcal{Y} = \{a\}$ avec $|a| = 0$. De plus, si (2a) est vraie, et si $\mathrm{val}(Y) = 0$, alors $\Phi(Y)$ dérive une infinité d'objets *de taille* 0. En effet, si $\mathcal{Y}_0$ est l'ensemble des objets de taille 0 dérivés de $Y$, alors $\Phi(\mathcal{Y}_0)$ est infini (2a), ne contient que des objets de taille nulle (additivité des tailles), et est inclus dans $\Phi(\mathcal{Y})$ (croissance du constructeur $\Phi$).

REMARQUE 3 : les constructeurs *union* et *produit cartésien* vérifient une condition voisine de la condition 1 : $f_{\text{union}}(m,n) = \min(m,n)$ et $f_{\text{product}}(m,n) = m+n$ ; ces deux fonctions sont croissantes sur $\mathbb{N}^2$ au sens de $(m \leq m' \text{ et } n \leq n') \Rightarrow f(m,n) \leq f(m',n')$.

Les conditions imposées à un constructeur pour être idéal semblent compliquées, néanmoins elles sont vérifiées par les constructeurs que nous avons introduits jusqu'ici :

**Lemme 1.** *Les constructeurs séquence, ensemble, multi-ensemble et cycle orienté sont idéaux.*

**Démonstration :** CONSTRUCTEUR SÉQUENCE : vérifions les conditions dans l'ordre de la définition :

1. $f_*(n) = f_*(\infty) = 0$ car $Y^*$ dérive toujours la séquence vide, même si $Y$ ne dérive rien,
2. le constructeur *sequence* vérifie (2a) : $\text{val}(Y) = 0 \Rightarrow \text{card}(Y^*) = \infty$,
3. $P(n)$ est vrai pour tout $n$ : $|(y)| = |y|$.

CONSTRUCTEUR MULTI-ENSEMBLE : les propriétés sont les mêmes que pour le constructeur séquence.

CONSTRUCTEUR ENSEMBLE :

1. $f_{\mathcal{P}}(n) = f_{\mathcal{P}}(\infty) = 0$ car $\mathcal{P}(Y)$ dérive toujours l'ensemble vide,
2. l'assertion (2b) est vérifiée : si $\text{card}(Y) = N$, alors $\text{card}(\mathcal{P}(Y)) = 2^N < \infty$,
3. $P(n)$ est vrai pour tout $n$ : $|\{y\}| = |y|$.

CONSTRUCTEUR CYCLE ORIENTÉ :

1. $f_{\mathcal{C}}(n) = n$, $f_{\mathcal{C}}(\infty) = \infty$ car un cycle de taille minimale dérivé de $\mathcal{C}(Y)$ est $[y]$, où $|y| = \text{val}(Y)$,
2. l'assertion (2a) est vérifiée : $\text{val}(Y) = 0 \Rightarrow \text{card}(\mathcal{C}(Y)) = \infty$,
3. $P(n)$ est vrai pour tout $n$ : $|[y]| = |y|$.

■

Nous annonçons tout de suite le théorème fondamental de cette section :

**Théorème 1.** *Soit $\omega$ une classe de spécifications dont les productions n'utilisent que des constructeurs idéaux. Le caractère bien fondé est décidable dans $\omega$.*

**Corollaire 3.** *Le caractère bien fondé est décidable dans $\Omega$.*

Dans le reste de cette section, nous nous appliquons à prouver le théorème ci-dessus.

### 1.4.2 Détermination de la valuation des non-terminaux

**Lemme 2.** *Soit $\omega$ une classe de spécifications dont les productions utilisent uniquement des constructeurs idéaux. Dans $\omega$, la valuation des non terminaux est décidable.*

**Démonstration :** L'algorithme suivant calcule les valuations des non-terminaux.

**Algorithme A :**
Donnée : l'ensemble $N$ des non-terminaux de la spécification.
Résultat : $v(T)$ est la valuation du non-terminal $T$.

**pour** chaque atome $a$, $v_0(a) \leftarrow |a|$
**pour** chaque non-terminal $T$, $v_0(T) \leftarrow \infty$
$j \leftarrow 0$
**répéter**

$j \leftarrow j + 1$
**pour** chaque non-terminal $T$ **faire**
**si** $T \to R_1 \mid \ldots \mid R_k$ **alors**
$v_j(T) \leftarrow \min(v_{j-1}(R_1), \ldots, v_{j-1}(R_k))$
**si** $T \to R_1 \ldots R_k$ **alors**
$v_j(T) \leftarrow v_{j-1}(R_1) + \cdots + v_{j-1}(R_k)$
**si** $T \to \Phi(R)$ **alors**
$v_j(T) \leftarrow f_\Phi(v_{j-1}(R))$
**fin pour**

**jusqu'à** ce que $v_j(T) = v_{j-1}(T)$ pour chaque non-terminal $T$
**pour** chaque non-terminal $T$, $v(T) \leftarrow v_j(T)$

Après $j$ passages dans la boucle **répéter-jusqu'à**, $v_j(T)$ n'est autre que la valuation des mots dérivés de $T$ en au plus $j$ étapes. De plus, pour chaque $j$, $v_j(T) \leq v_{j-1}(T)$ à cause de la croissance des fonctions *min*, + et $f_\Phi$. Or $v_j$ est minoré par le vecteur $(0, \ldots, 0)$ ; par conséquent, la suite $(v_j)_{j \in \mathbb{N}}$ converge, et l'algorithme termine. ■

Essayons l'algorithme $A$ sur les spécifications des programmes donnés au début de ce chapitre.
EXEMPLE 1 : La définition des partitions (page 13) est

**type** partition = *multiset*(entier);
entier = *sequence*(un, *card* >= 1);
un = *atom*(1);

soit, en choisissant la lettre $P$ pour `partition`, $E$ pour `entier` et $u$ pour `un`,

$$\begin{cases} P & \to & \mathcal{M}(E) \\ E & \to & u\,F \\ F & \to & u^* \\ |u| & = & 1. \end{cases}$$

Nous avons remplacé `sequence(un,card>=1)` par la production `product(un,sequence(un))`, et nous avons introduit un nouveau non-terminal $F$ pour mettre la grammaire en *Chomsky Normal Form.*

Appliquons l'algorithme $A$ : les valeurs initiales des valuations sont :

$$v_0(P) = \infty, \quad v_0(E) = \infty, \quad v_0(F) = \infty, \quad v_0(u) = 1.$$

Après un passage dans la boucle **répéter-jusqu'à**, les nouvelles valeurs sont

$$v_1(P) = 0, \quad v_1(E) = \infty, \quad v_1(F) = 0, \quad v_1(u) = 1,$$

puis après un second passage,

$$v_2(P) = 0, \quad v_2(E) = 1, \quad v_2(F) = 0, \quad v_2(u) = 1,$$

et le troisième donne les mêmes valeurs. Les valuations sont donc : val$(P) = 0$ et val$(E) = 1$. □

EXEMPLE 2 : Dans le programme de dérivation formelle, le type `expression` est défini comme suit

**type** expression = zero | un | x | *product*(plus,expression,expression)
| *product*(mult,expression,expression) | *product*(expo,expression);
zero, un, x, plus, mult, expo = *atom*(1);

c'est-à-dire avec des notations évidentes :

$$\begin{cases} E & \to & z \mid u \mid x \mid E_+ \mid E_\times \mid E_e \\ E_+ & \to & p\,F \\ E_\times & \to & t\,F \\ E_e & \to & e\,E \\ F & \to & E\,E \\ |z| & = & |u| = |x| = |p| = |t| = |e| = 1. \end{cases}$$

Les valeurs de $v_j$ pour les non-terminaux sont

$$v_1(E) = 1, \quad v_1(E_+) = \infty, \quad v_1(E_\times) = \infty, \quad v_1(E_e) = \infty, \quad v_1(F) = \infty,$$

$$v_2(E) = 1, \quad v_2(E_+) = \infty, \quad v_2(E_\times) = \infty, \quad v_2(E_e) = 2, \quad v_2(F) = 2,$$

$$v_3(E) = 1, \quad v_3(E_+) = 3, \quad v_3(E_\times) = 3, \quad v_3(E_e) = 2, \quad v_3(F) = 2,$$

et le vecteur $v_4$ est identique à $v_3$, donc la valuation de $E$ est 1. □

EXEMPLE 3 : Même si ce chapitre est consacré à l'univers non étiqueté, les résultats de cette section s'appliquent aussi à l'univers étiqueté, et notamment l'algorithme $A$. En anticipant sur la définition des constructeurs en univers étiqueté, nous pouvons calculer les valuations de la spécification des graphes connexes monocycliques (page 16).

**type** c_u_graph = *ucycle*(tree);
tree = *product*(node,*set*(tree));
node = *Latom*(1);

Cette spécification s'écrit sous forme de grammaire (la lettre $o$ représente le type `node`) :

$$\begin{cases} G & \to & \mathcal{UC}(T) \\ T & \to & o\,F \\ F & \to & \mathcal{P}(T) \\ |o| & = & 1. \end{cases}$$

Le constructeur cycle non orienté vérifie val$(\mathcal{UC}(\mathcal{R}))$ = val$(\mathcal{R})$, et le constructeur ensemble vérifie val$(\mathcal{P}(\mathcal{R})) = 0$. L'algorithme $A$ donne par conséquent

$$v_1(G) = \infty, \quad v_1(T) = \infty, \quad v_1(F) = 0,$$

$$v_2(G) = \infty, \quad v_2(T) = 1, \quad v_2(F) = 0,$$

$$v_3(G) = 1, \quad v_2(T) = 1, \quad v_2(F) = 0,$$

et $v_4 = v_3$, donc $\mathrm{val}(G) = \mathrm{val}(T) = 1$. □

Voici un autre exemple qui nous servira dans la sous-section suivante :
EXEMPLE 4 :

$$(\Sigma_3)\begin{cases} T & = & \{a,b,c\} \\ N & = & \{S,U,V\} \\ P & : & S \to a \mid aUa \qquad U \to a \mid bV \qquad V \to S \mid Uc \\ |.| & : & |a| = 1 \quad |b| = |c| = 0 \end{cases}$$

Au départ ($j = 0$), nous avons $v_0(a) = 1$, $v_0(b) = v_0(c) = 0$ et $v_0(S) = v_0(U) = v_0(V) = \infty$. Les formules donnant $v_j$ en fonction de $v_{j-1}$ sont

$$\begin{aligned} v_j(S) &= \min(1, 2 + v_{j-1}(U)) \\ v_j(U) &= \min(1, v_{j-1}(V)) \\ v_j(V) &= \min(v_{j-1}(S), v_{j-1}(U)) \end{aligned}$$

Après une étape, nous obtenons $v_1(S) = v_1(U) = 1$ et $v_1(V) = \infty$, puis $v_2(S) = v_2(U) = v_2(V) = 1$ et $v_3(S) = v_3(U) = v_3(V) = 1$. Nous en concluons que les trois non-terminaux de $(\Sigma_3)$ ont pour valuation 1. □

Seule la condition 1 de la définition d'un constructeur idéal a été utilisée dans la preuve du lemme 2. Cette condition permet de déterminer les valuations sans avoir à énumérer les objets de taille minimale. Lorsque l'énumération est indispensable, le calcul des valuations devient plus ardu, comme l'exemple suivant le montre.
EXEMPLE 5 : Soit le constructeur $\mathcal{P}_2$ tel que $\mathcal{P}_2(Y)$ dérive les ensembles de cardinalité deux, dont chaque élément est dérivé de $Y$. La valuation de $\mathcal{P}_2(Y)$ ne dépend pas uniquement de celle de $Y$, elle dépend aussi du nombre d'objets distincts de taille $\mathrm{val}(Y)$, comme les trois exemples suivants le montrent :

$$\begin{array}{cccc} Y \to \{a,b\} & |a| = |b| = 1 & \mathcal{P}_2(Y) \to \{\{a,b\}\} & \mathrm{val}(\mathcal{P}_2(Y)) = 2, \\ Y \to \{a,b\} & |a| = 1, |b| = 2 & \mathcal{P}_2(Y) \to \{\{a,b\}\} & \mathrm{val}(\mathcal{P}_2(Y)) = 3, \\ Y \to \{a\} & |a| = 1 & \mathcal{P}_2(Y) \to \emptyset & \mathrm{val}(\mathcal{P}_2(Y)) = \infty. \end{array}$$

La seule relation que l'on peut obtenir *a priori* entre les deux valuations est une inégalité : $\mathrm{val}(\mathcal{P}_2(Y)) \geq 2\mathrm{val}(Y)$. Le premier exemple est un cas d'égalité, l'exemple intermédiaire un cas d'inégalité stricte, et le dernier exemple montre qu'il n'existe même pas de borne supérieure finie pour $\mathrm{val}(\mathcal{P}_2(Y))$ ! □

Ainsi, lorsqu'il n'existe pas de fonction $f_\Phi$, il faut recourir à une méthode *ad hoc* pour déterminer les valuations.

### 1.4.3 Détermination du caractère bien fondé d'une spécification

Nous savons maintenant calculer la valuation d'un non-terminal, donc nous savons vérifier que chaque non-terminal dérive au moins un objet de taille finie (condition $(i)$ de la définition 5). Pour vérifier la condition $(ii)$, il faut nous assurer que pour tout $n$, le nombre d'éléments de taille $n$ est fini. Cette vérification n'est pas toujours évidente. Par exemple, dans la spécification $(\Sigma_3)$, le non-terminal $S$ engendre une infinité de mots de la forme $ab^nac^na$ qui sont tous de taille 3. Cela

est dû aux productions $U \to bV$ et $V \to Uc$ ; mises bout à bout, ces deux productions donnent la chaîne $U \to bV \to bUc$. La dégénérescence provient de ce que $U$ dérive $bUc$, où $b$ et $c$ sont des atomes de taille nulle.

Soit $\Sigma$ une spécification n'utilisant que des constructeurs idéaux. Pour $n \in \mathbb{N}$, nous définissons la relation d'ordre partiel $\prec_n$ entre les non-terminaux de $\Sigma$ par $V \prec_n U$ lorsque :

- soit $U \to V \mid W$,
- soit $U \to V \times W$ et $\mathrm{val}(W) = 0$,
- soit $U \to \Phi(V)$ et $\Phi$ vérifie l'assertion $P(n)$ de la condition 3 (définition 6).

Ceci définit une suite de relations $(\prec_n)_{n\in\mathbb{N}}$. Cette suite est stationnaire car pour chaque multi-constructeur $\Phi$, la suite $P(n)$ l'est, et le nombre de multi-constructeurs de $\Sigma$ est fini.

**Lemme 3.** *Soit une spécification $\Sigma$ utilisant des constructeurs idéaux et telle que les valuations des non-terminaux sont finies, alors $\alpha \Leftrightarrow (\beta_1$ ou $\beta_2)$ :*

*($\alpha$) il existe un non-terminal dérivant une infinité d'objets de même taille,*

*($\beta_1$) il existe une production $X \to \Phi(Y)$ où $\Phi$ vérifie la condition (2a) et val$(Y) = 0$,*

*($\beta_2$) il existe un cycle dans un ordre partiel $\prec_n$, et l'un des non-terminaux du cycle dérive au moins un objet de taille $n$.*

**Démonstration :** Pour cette preuve, nous considérons la grammaire de $\Sigma$ sous forme compacte. $\alpha \Rightarrow (\beta_1$ ou $\beta_2)$ : supposons que $\alpha$ est vraie, $\beta_1$ fausse, et nous allons montrer que $\beta_2$ est nécessairement vraie. Soit $X$ un non-terminal dérivant une infinité d'objets de taille $n$ : $\mathrm{card}(\mathcal{X}_n) = \infty$. Sans restreindre la généralité du problème, nous pouvons imposer à $n$ d'être le plus petit entier pour lequel il existe un tel $X$. La production dont $X$ est le membre gauche est de l'un des types suivants :

$X \to a$ où $a$ est un atome : c'est impossible car $\{a\}$ contient un seul objet,

$X \to Y \mid Z$ : nécessairement $\mathrm{card}(Y_n) = \infty$ ou $\mathrm{card}(Z_n) = \infty$,

$X \to Y \times Z$ : comme $n$ est le plus petit entier adéquat, nécessairement $\mathrm{val}(Y) = 0$ et $\mathrm{card}(Z_n) = \infty$, ou bien $\mathrm{card}(Y_n) = \infty$ et $\mathrm{val}(Z) = 0$,

$X \to \Phi(Y)$ : les objets de taille $n$ dérivés de $X$ sont formés à partir d'objets de taille inférieure ou égale à $n$, dérivés de $Y$ (additivité des tailles). Nous noterons $\mathcal{Y}_{\leq n}$ l'ensemble de ces objets. Deux cas peuvent se présenter. Soit $\Phi$ vérifie (2a) et alors $\mathrm{val}(Y) > 0$ (car $\beta_1$ est fausse) : si $\mathcal{Y}_{\leq n}$ était de cardinal fini $N$, comme $\Phi$ dérive au plus un objet sans composante, on aurait $\mathrm{card}(\mathcal{X}_{\leq n}) \leq 1 + N + N^2 + \cdots + N^n < \infty$. Dans le cas où $\Phi$ vérifie (2b), si $\mathcal{Y}_{\leq n}$ était fini, alors $\Phi(\mathcal{Y}_{\leq n})$ et par suite $\mathcal{X}_{\leq n}$ seraient aussi finis. Dans les deux cas, supposer $\mathcal{Y}_{\leq n}$ fini contredit l'hypothèse $\mathcal{X}_n$ infini : $\mathcal{Y}_{\leq n}$ est donc infini, et $\mathcal{Y}_n$ aussi (hypothèse $n$ minimum).

Si $\Phi$ vérifiait $Q(n)$, les objets de $\mathcal{X}_n$ seraient dérivés d'objets de $\mathcal{Y}_{<n}$, qui est fini (hypothèse $n$ minimum) ; par le même raisonnement que ci-dessus, $\Phi(\mathcal{Y}_{<n})$ serait fini, donc $\mathcal{X}_n$, ce qui serait contradictoire. Par conséquent, $\Phi$ vérifie nécessairement $P(n)$.

Nous avons montré que si $\mathcal{X}_n$ est infini, alors il existe un non-terminal $Y$ tel que $\mathcal{Y}_n$ est infini et soit $X \to Y \mid Z$, soit $X \to YZ$ et $\text{val}(Z) = 0$, soit $X \to \Phi(Y)$ et $\Phi$ vérifie $P(n)$. Ceci s'exprime simplement à l'aide de l'ordre partiel $\prec_n$ : $\text{card}(\mathcal{X}_n) = \infty \Rightarrow \exists Y, (\text{card}(\mathcal{Y}_n) = \infty)$ et $(Y \prec_n X)$. Les non-terminaux étant en nombre fini, l'ordre partiel $\prec_n$ admet forcément un cycle.

($\beta_1$ ou $\beta_2$) $\Rightarrow \alpha$ : lorsque $\beta_1$ est vraie, il est évident que $\alpha$ l'est aussi ($\mathcal{X}_0$ est infini). Supposons maintenant que $\beta_2$ est vraie. Soit $X$ l'un des non-terminaux du cycle, dérivant $x$ de taille $n$. Soit $Y$ le non-terminal suivant $X$ dans le cycle $(X \prec_n Y)$, alors $Y$ dérive un objet de taille $n$ formé à partir de $x$ :

**si** $Y \to X \mid Z$ **:** alors $Y$ dérive $x$,

**si** $Y \to X \times Z$ **:** par définition de $\prec_n$, $\text{val}(Z) = 0$, donc $Z$ dérive un objet $z$ de taille nulle, et $Y$ dérive $(x, z)$, de taille $n$,

**si** $Y \to \Phi(X)$ **:** d'après $P(n)$, $\Phi(\{x\})$ contient au moins un objet de taille $n$. Par additivité des tailles, $x$ est forcément une composante de cet objet.

En continuant ce procédé le long du cycle, nous formons un objet $x'$ de taille $n$ dérivé de $X$, dont $x$ est une composante. Au moins l'une des productions du cycle n'est pas une union, sinon la grammaire serait ambiguë, ce que nous avons exclu dès le départ. Il s'ensuit que $x$ est une composante au sens strict de $x'$ (c'est-à-dire $x \neq x'$). En répétant ce procédé, nous pouvons construire une infinité d'objets différents $x, x', x'', \ldots$ qui sont tous dans $\mathcal{X}_n$. ■

**Lemme 4.** *Soit une spécification $\Sigma$ utilisant des constructeurs idéaux et telle que les valuations des non-terminaux sont finies. Nous supposons en plus que tout non-terminal dérive un nombre fini d'objets de taille strictement inférieure à $n$, et que la condition ($\beta_1$) du lemme 3 n'est pas vérifiée. Alors pour tout non-terminal $X$, l'affirmation "$X$ dérive au moins un objet de taille $n$" est décidable.*

**Démonstration :** Si $n = 0$, l'affirmation est équivalente à "$\text{val}(X) = 0$", qui est décidable (lemme 2). Supposons $n > 0$. Soit $\Re(X)$ l'ensemble des non-terminaux $Y$ tels que $Y \prec_n^* X$, où $\prec_n^*$ est la clôture transitive de $\prec_n$. Alors $X$ dérive un objet de taille $n$ si et seulement si l'un des éléments de $\Re(X)$ dérive un objet de taille $n$. Or les objets de taille $n$ dérivés de $Y \in \Re(X)$ sont de trois types :

1. ceux construits à partir d'objets de taille inférieure à $n$,
2. ceux construits à partir d'un objet $z$ de taille $n$, et éventuellement d'autres objets de taille nulle,
3. les atomes produits directement par $Y$.

Pour les objets du type 2, le non-terminal $Z$ dérivant $z$ est forcément dans $\Re(X)$ par définition de la relation $\prec_n$. Par conséquent, $X$ dérive un objet de taille $n$ si et seulement si l'un des éléments de $\Re(X)$ dérive un objet de type 1 ou 3. Pour le type 3, la vérification est facile. Quant au type 1, il suffit pour chaque $Y \in \Re(X)$ de former tous[4] les objets formés à partir de composantes de taille strictement inférieure à $n$, et de regarder si l'un d'eux est de taille $n$. ■

Nous pouvons maintenant prouver le théorème 1 (énoncé page 27), en donnant un algorithme de décision du caractère bien fondé.

**Démonstration :** (théorème 1)

[4]Ils sont en nombre fini : si $\mathcal{U}_{<n}$ et $\mathcal{V}_{<n}$ sont finis, il en est de même de $\mathcal{U}_{<n} \times \mathcal{V}_{<n}$, et de $\Phi(\mathcal{U}_{<n})$ si $\beta_1$ est fausse.

**Algorithme B:**
Donnée : une spécification $\Sigma$ n'utilisant que des constructeurs idéaux.
Résultat : *vrai* si $\Sigma$ est bien fondée, *faux* sinon.

1. calculer la valuation de chaque non-terminal à l'aide de l'algorithme A,
   si l'une des valuations est infinie, renvoyer *faux* ;
2. pour $\Phi$ vérifiant $(2a)$, s'il existe $X \to \Phi(Y)$ avec $\mathrm{val}(Y) = 0$, renvoyer *faux* ;
   $n \leftarrow 0, \quad N \leftarrow \max\{N_\Phi\}$,
3. si $n \geq N$, aller en 5,
   calculer le graphe de la relation $\prec_n$,
   si ce graphe n'admet pas de cycle, aller en 4,
   si un non-terminal d'un cycle dérive un objet de taille $n$, renvoyer *faux* ;
4. $n \leftarrow n + 1$, aller en 3 ;
5. calculer le graphe de la relation $\prec_N$,
   si ce graphe n'admet pas de cycle, aller en 6,
   si un non-terminal d'un cycle dérive un objet de taille $\geq N$, renvoyer *faux* ;
6. renvoyer *vrai*.

Pour $n \geq N$, comme la relation $\prec_n$ est stationnaire, on peut faire une vérification simultanée. Pour cela, on utilise un raisonnement analogue à celui du lemme 4 pour décider si un non-terminal dérive un objet de taille supérieure ou égale à $N$. ■

EXEMPLE 6 : (Partitions) Nous rappelons la grammaire :

$$\begin{cases} P & \to & \mathcal{M}(E) \\ E & \to & u\,F \\ F & \to & u^* \\ |u| & = & 1. \end{cases}$$

Appliquons l'algorithme $B$ : (1) les valuations calculées par l'algorithme $A$ (exemple 1 page 28) sont $\mathrm{val}(P) = 0$, $\mathrm{val}(E) = 1$ et $\mathrm{val}(F) = 0$ ; (2) les constructeurs multi-ensemble et séquence vérifient la condition $(2a)$, mais $\mathrm{val}(E) > 0$ et $\mathrm{val}(u) > 0$ ; $N_{\mathcal{M}} = N_* = 0$ donc $N = 0$ ; (5) la seule relation entre les non-terminaux est $E \prec P$. Il n'y a pas de cycle, donc la spécification est bien fondée.

En fait, nous aurions pu conclure après l'étape (2). En effet, la spécification des partitions est explicite, c'est-à-dire non récursive, en ce sens qu'il n'y a pas de cycle dans les définitions ($P$ ne dépend que de $E$, qui lui-même ne dépend que de $u$), donc *a fortiori* il ne peut pas y avoir de cycle dans le graphe de la relation $\prec$, qui est un sous-graphe du graphe de dépendance des non-terminaux. □

EXEMPLE 7 : (Dérivation formelle) Cet exemple est plus intéressant car la spécification est récursive : le type `expression` est défini en fonction de lui-même. A partir de la grammaire

$$\begin{cases} E & \to & z \mid u \mid x \mid E_+ \mid E_\times \mid E_e \\ E_+ & \to & p\,F \\ E_\times & \to & t\,F \\ E_e & \to & e\,E \\ F & \to & E\,E \\ |z| & = & |u| = |x| = |p| = |t| = |e| = 1, \end{cases}$$

l'algorithme $A$ nous avait donné $\mathrm{val}(E) = 1$, $\mathrm{val}(E_+) = \mathrm{val}(E_\times) = 3$ et $\mathrm{val}(E_e) = \mathrm{val}(F) = 2$ (exemple 2). L'ordre $\prec$ ne dépend pas de $n$ : les relations sont $E_+ \prec E$, $E_\times \prec E$, $E_e \prec E$, et il n'y a donc pas de cycle. Nous pouvons le constater aussi sur la forme initiale $E \to z \mid u \mid x \mid p\,E\,E \mid t\,E\,E \mid e\,E$ : dans chaque produit cartésien où apparaît $E$ se trouve aussi un atome de taille 1, donc $E \not\prec E$. □

EXEMPLE 8 : (Graphes connexes monocycliques) Voici un autre exemple de spécification récursive :

$$\begin{cases} G & \to & \mathcal{UC}(T) \\ T & \to & n\,F \\ F & \to & \mathcal{P}(T) \\ |n| & = & 1. \end{cases}$$

Les valuations sont $\mathrm{val}(G) = \mathrm{val}(T) = 1$ et $\mathrm{val}(F) = 0$ (exemple 3). A nouveau, l'ordre $\prec$ ne dépend pas de $n$, et les relations sont $T \prec G$ et $T \prec F$ : cette spécification est bien fondée. □

EXEMPLE 9 : Soit la spécification

$$(\Sigma_4)\begin{cases} A & \to & C \mid A \times C \\ B & \to & A \mid b \\ C & \to & \mathcal{C}(B) \\ |b| & = & 1. \end{cases}$$

Le calcul des valuations par l'algorithme A donne $\mathrm{val}(A) = \mathrm{val}(B) = \mathrm{val}(C) = 1$. La relation $\prec_n$ ne dépend pas de $n$ puisque tous les constructeurs vérifient $P(n)$ pour tout $n$, et son graphe comprend un cycle $A \prec_n B \prec_n C \prec_n A$. De plus, le non-terminal $B$ dérive un objet de taille 1, l'atome $b$. La spécification $(\Sigma_4)$ n'est donc pas bien fondée. On constate en effet que $B$ dérive $b$, $[b]$, $[[b]]$, ... qui sont tous de taille 1. □

Tous les constructeurs de $\Omega$ vérifient la propriété $P(n)$ pour tout $n$. Ceci a pour conséquence que la relation $\prec$ ne dépend jamais de $n$ pour les spécifications de $\Omega$, ce qui simplifie considérablement l'algorithme $B$. Il existe néanmoins des constructeurs idéaux qui vérifient $Q(n)$ pour certaines valeurs de $n$, comme le constructeur *diagonal* $\Delta$ : *Le constructeur diagonal (noté $\Delta$) qui construit à partir d'un ensemble $Y$ l'ensemble des paires $(y, y)$, pour $y \in Y$, est idéal :*

1. $f_\Delta(n) = 2n$, $f_\Delta(\infty) = \infty$,
2. $\Delta$ vérifie (2b) : $\mathrm{card}(Y) = N \Rightarrow \mathrm{card}(\Delta(Y)) = N$,
3. $\Delta$ vérifie $P(0)$, et $Q(n)$ pour $n \geq 1$ : $N_\Delta = 1$.

Ainsi, le théorème 1 nous permet d'enrichir à volonté la classe $\Omega$ par des constructeurs idéaux comme $\Delta$, en conservant la décidabilité du caractère bien fondé.

EXEMPLE 10 : Remplaçons dans $(\Sigma_4)$ le constructeur *cycle* par $\Delta$. Nous obtenons une nouvelle spécification

$$(\Sigma_5)\begin{cases} A & \to & C \mid A \times C \\ B & \to & A \mid b \\ C & \to & \Delta(B) \\ |b| & = & 1. \end{cases}$$

Les valuations sont $\mathrm{val}(B) = 1$, $\mathrm{val}(A) = \mathrm{val}(C) = 2$. Pour $n = 0$, le graphe associé comporte un cycle, $A \prec_0 B \prec_0 C \prec_0 A$, mais aucun non-terminal n'engendre d'objet de taille nulle. Pour $n > 0$,

le cycle est "cassé", car $\Delta$ vérifie $Q(n)$ (et non plus $P(n)$). La spécification $(\Sigma_5)$ est donc bien fondée. □

Nous avions déjà remarqué que pour un constructeur non idéal, il pouvait s'avérer nécessaire pour le calcul des valuations d'énumérer les objets dérivés. Il en est de même pour vérifier que chaque non-terminal dérive un nombre fini d'objets de taille bornée, comme le montre l'exemple suivant (le constructeur $\mathcal{P}_2$ engendre les parties à deux éléments, cf exemple 5).

EXEMPLE 11 :

$$(\Sigma_6)\begin{cases} U \to a \mid b \mid \mathcal{P}_2(V) \\ V \to \mathcal{P}_2(U) \\ |a| = |b| = 0. \end{cases}$$

Le calcul des valuations donne $\mathrm{val}(U) = \mathrm{val}(V) = 0$ car $V$ contient l'ensemble $\{a, b\}$. Modifions légèrement la production de $U$ en introduisant un troisième atome :

$$(\Sigma_7)\begin{cases} U \to a \mid b \mid c \mid \mathcal{P}_2(V) \\ V \to \mathcal{P}_2(U) \\ |a| = |b| = |c| = 0. \end{cases}$$

Les valuations sont les mêmes, mais la spécification $(\Sigma_6)$ est bien fondée, alors que $(\Sigma_7)$ ne l'est pas. En effet, dans $(\Sigma_6)$, $V$ dérive un seul objet, l'ensemble $\{a, b\}$, et par suite $U$ ne dérive que les atomes $a$ et $b$.

Par contre, dans $(\Sigma_7)$, $V$ dérive au moins trois objets, les ensembles $\{a, b\}$, $\{b, c\}$ et $\{a, c\}$. En fait, tous les objets de $\mathcal{U}$ et $\mathcal{V}$ sont de taille nulle. Supposons que leur cardinal, respectivement $u_0$ et $v_0$, soit fini. Nous avons alors les égalités $u_0 = 3 + v_0(v_0 - 1)/2$ et $v_0 = u_0(u_0 - 1)/2$. Puisque $u_0 \geq 3$, la seconde égalité donne $v_0 \geq u_0$, ce qui donne en reportant dans la première $u_0 \geq 3 + u_0$. Comme l'hypothèse aboutit à une contradiction, nous concluons que $u_0$ et $v_0$ sont infinis. □

Ce dernier exemple nous montre que pour des constructeurs non idéaux, le calcul des valuations, même lorsqu'il est possible, ne suffit pas pour décider simplement du caractère bien fondé de la spécification.

## 1.5 Des programmes sur Ω

Dans la section précédente, nous avons défini une classe Ω de spécifications, et nous avons montré que le caractère bien fondé est décidable dans Ω. Dans cette section, nous définissons une classe Π de programmes sur Ω. Comme pour les spécifications, nous ne nous intéressons qu'aux programmes bien fondés, et nous montrons que cette propriété est décidable dans Π (théorème 2). En particulier, la terminaison des programmes est décidable dans Π (lemme 6).

### 1.5.1 Procédures et instructions

**Définition 7.** *La définition d'une procédure s'écrit* $P(x : X) := \langle corps \rangle$ *où*

- $P$ *est le nom de la procédure,*
- $x$ *est une variable qui représente l'argument de la procédure,*
- $X$ *est un non-terminal qui représente le type de* $x$*,*
- $\langle corps \rangle$ *est le corps de la procédure.*

*Un programme est un ensemble de définitions de procédures.*

Le corps de la procédure est une suite d'instructions, qui sont exécutées séquentiellement lors de l'évaluation de la procédure. Une instruction est soit une instruction élémentaire de coût constant (par exemple addition de deux entiers de la machine, ou écriture d'un caractère), soit une instruction de *sélection* (appel d'une procédure sur une composante), soit une instruction d'*itération* (appel d'une procédure sur toutes les composantes). Les procédures que nous étudierons ont par conséquent un seul argument (qui peut néanmoins être un produit cartésien) ; il n'y a pas de variables locales ni d'affectations.

Nous noterons aussi $P : X$ pour indiquer que $P$ prend un argument de type $X$, lorsqu'il n'est pas nécessaire de préciser le nom de la variable. Nous étudions à présent les différentes instructions (ou schémas) qui peuvent composer le corps de la procédure.

EXÉCUTION SÉQUENTIELLE : ce schéma permet de regrouper plusieurs opérations dans une même procédure. Nous le retrouvons en Pascal sous la forme des blocs **begin** ... **end**, dans le langage C sous la forme des blocs { ... }, en Lisp sous la forme des **(progn (...))**.

$$P(a : A) := Q(a); R(a) \qquad \text{où } Q : A \text{ et } R : A.$$

L'évaluation de $P(a)$ entraîne celle de $Q(a)$, puis celle de $R(a)$.

INSTRUCTIONS ÉLÉMENTAIRES : comme nous ne nous intéressons qu'au coût du programme, il nous suffit d'un schéma pour modéliser les instructions élémentaires, c'est-à-dire celles dont le coût est constant :

$$P(a : A) := \text{count}(\mu) \qquad \text{où } \mu \in \mathbb{N}.$$

Cette déclaration indique que le coût d'évaluation de la procédure $P$ est $\mu$ pour n'importe quelle donnée. Le fait d'imposer à $\mu$ d'être entier a bien un sens : le temps d'exécution des instructions élémentaires d'un microprocesseur se compte en nombre entier de cycles d'horloge.

Chaque constructeur induit une instruction (nous dirons aussi schéma) de *sélection* et éventuellement un schéma d'*itération*, lorsque les objets comprennent des composantes de même type (multi-objets).

UNION : seul le schéma de sélection existe, puisqu'il n'y a qu'une seule composante.

$$C \to A \mid B$$
$$P(c : C) := c \in A \to Q(c),\ c \in B \to R(c) \qquad \text{où } Q : A \text{ et } R : B.$$

Lorsque la donnée $c$ est dans $\mathcal{A}$, le programme exécute $Q(c)$, sinon $R(c)$. Ce schéma est souvent utilisé dans les langages de programmation : c'est le procédé de filtrage que l'on retrouve dans les instructions **case** ... **end** de Pascal, **switch** ... { ... } du langage C, et plus généralement dans les langages utilisant le *pattern-matching*.

PRODUIT CARTÉSIEN : le schéma de sélection consiste à appeler une procédure avec comme argument l'une des composantes du produit cartésien :

$$C \to A \times B$$
$$P(c : C) := c = (a, b) \to Q(a) \qquad \text{où } Q : A.$$

| schéma de programmation | | notation |
|---|---|---|
| déclaration de procédure | | $P(a:A) := \langle \text{corps} \rangle$ |
| exécution séquentielle | | $Q(a); R(a)$ |
| union | sélection | $c \in A \rightarrow Q(c)$ |
| produit cartésien | sélection | $c = (a,b) \rightarrow Q(a)$ |
| liste, ensemble, | sélection | $c = (a_1, \ldots, a_k) \rightarrow Q(a_{\text{rnd}(1\ldots k)})$ |
| multi-ensemble, cycle | itération | $c = (a_1, \ldots, a_k) \rightarrow Q(a_1); \ldots; Q(a_k)$ |
| test sur la taille | | $\lvert a \rvert \leq k \rightarrow Q(a),\ \lvert a \rvert > k \rightarrow R(a)$ |
| test sur la longueur | | $l(a) \leq k \rightarrow Q(a),\ l(a) > k \rightarrow R(a)$ |

Figure 1.2 : Les schémas de programmation de la classe II

Si $a$ et $b$ sont les composantes du produit cartésien $c$, ce programme appelle la procédure $Q$ sur $a$. Ce schéma de programmation est utilisé par exemple lorsqu'on fait une recherche dans une structure de donnée. Dans ce cas, la recherche peut se limiter à l'une des composantes. Par contre, dans le schéma d'itération, toutes les composantes sont visitées, et il est alors nécessaire qu'elles soient du même type pour évaluer la même procédure dessus. Comme le nombre de composantes d'un produit cartésien est fini, le schéma d'itération se déduit des schémas d'exécution séquentielle et de sélection dans un produit cartésien, et n'est donc pas un schéma de base.

LISTE, ENSEMBLE, MULTI-ENSEMBLE ET CYCLE : le schéma de sélection visite un élément choisi *au hasard*. Par exemple pour un ensemble :

$$C \rightarrow \mathcal{P}(A)$$
$$P(c:C) := c = \{a_1, \ldots, a_k\} \rightarrow Q(a_{\text{rnd}(1\ldots k)}) \qquad \text{où } Q : A,$$

où $\text{rnd}(1\ldots k)$ est un nombre tiré au hasard uniformément dans l'intervalle entier $[1, \ldots, k]$. Le schéma d'itération visite quant à lui toutes les composantes ; par exemple pour un cycle :

$$C \rightarrow \mathcal{C}(A)$$
$$P(c:C) := c = [a_1, \ldots, a_k] \rightarrow Q(a_1); \ldots; Q(a_k) \qquad \text{où } Q : A.$$

Pour le constructeur *sequence*, on désire parfois visiter le premier élément, au lieu d'un élément choisi au hasard. Pour ce faire, il faut définir la séquence à l'aide des constructeurs *union* et *produit cartésien* comme indiqué page 23 (remarque et équation (1.5)). En fait, nous verrons au chapitre suivant que ces deux schémas sont équivalents en termes de complexité.

Après les schémas de sélection et d'itération sur les composantes des constructeurs, nous présentons deux schémas supplémentaires permettant de sélectionner des objets par leur taille, ou par le nombre de leurs composantes pour un objet créé par un multi-constructeur.

TEST SUR LA TAILLE : le schéma est le suivant :

$$P(a:A) := \lvert a \rvert \leq k \rightarrow Q(a),\ \lvert a \rvert > k \rightarrow R(a) \qquad \text{où } Q : A,\ R : A,$$

et où $k$ est une constante entière positive ou nulle. Cette instruction permet de diriger l'exécution du programme en fonction de la taille de l'objet $a$ : si celle-ci est inférieure ou égale à $k$, $Q(a)$ est évalué, sinon c'est $R(a)$ qui est évalué.

TEST SUR LA LONGUEUR : pour un objet $a$ dérivé par un multi-constructeur, sa longueur $l(a)$ est le nombre de composantes directes. Le schéma ci-après permet de modifier l'exécution en fonction de la longueur d'un multi-objet :

$$P(a:A) := l(a) \leq k \rightarrow Q(a),\ l(a) > k \rightarrow R(a) \qquad \text{où } Q : A \text{ et } R : A.$$

Comme pour le test sur la taille, $k$ est une constante entière positive ou nulle ; lorsque la longueur est inférieure ou égale à $k$, on appelle $Q$ sur $a$, sinon on appelle $R$.

**Définition 8.** *La classe* Π *est l'ensemble des programmes dont les structures de données sont définies par une spécification de* Ω*, et dont les procédures sont définies à partir d'instructions élémentaires, des schémas d'exécution séquentielle, de sélection dans une union ou dans un produit cartésien, de sélection et d'itération dans une séquence, un ensemble, un cycle ou un multi-ensemble, et des schémas de test sur la taille et sur la longueur.*

Le tableau de la figure 1.2 résume les schémas de programmation de la classe Π.

Ainsi, les deux premiers exemples de ce chapitre (calcul du nombre de sommants dans une partition et dérivation formelle d'expression) sont dans la classe Π. Nous avons déjà vu plus haut que leurs spécifications sont dans la classe Ω. Il ne nous reste plus qu'à vérifier que les schémas de programmation sont bien ceux de Π. Pour cela, nous montrons qu'il est possible d'écrire ces programmes avec les notations de la figure 1.2. Par exemple, la figure 1.3 indique la formulation du programme de dérivation formelle avec la lettre $E$ représentant le type `expression`, $D$ la procédure `diff` et $C$ la procédure `copy`. Cette formulation est cependant lourde et trop rigide ; nous introduirons dans la prochaine section une autre notation, plus souple et plus proche des véritables langages de programmation.

### 1.5.2 Caractère bien fondé

Nous avons défini ci-dessus une classe Π de programmes, de façon syntaxique. La présente section s'intéresse de plus près à la terminaison des programmes de Π.

**Définition 9.** *Un programme est dit* bien fondé *si*

- *la spécification de ses structures de données est bien fondée,*
- *pour toute procédure $P : A$, et pour toute donnée $a \in \mathcal{A}$, l'évaluation de $P(a)$ termine au bout d'un nombre fini d'appels de procédures.*

*Lorsque la seconde propriété est vraie, nous dirons que le programme termine. Sinon, nous dirons qu'il "boucle".*

Par exemple, le programme suivant n'est pas bien fondé : il ne termine pas.

$$(\pi_1)\begin{cases} X & \rightarrow & a \\ P(x:X) & := & Q(x) \\ Q(x:X) & := & P(x) \end{cases}$$

$$
\begin{array}{rcl}
E & \to & E_0 \mid E_1 \mid E_x \mid E_+ \mid E_\times \mid E_{\exp} \\
E_0 & \to & 0 \\
E_1 & \to & 1 \\
E_x & \to & x \\
E_+ & \to & (+, E, E) \\
E_\times & \to & (\times, E, E) \\
E_{\exp} & \to & (\exp, E) \\
|0| & = & |1| = |x| = |+| = |\times| = |\exp| = 1 \\
 & & \\
D(e:E) & := & e \in E_0 \to 0,\ e \in E_1 \to 0,\ e \in E_x \to 1, \\
 & & e \in E_+ \to D_+(e),\ e \in E_\times \to D_\times(e),\ e \in E_{\exp} \to D_{\exp}(e) \\
D_+(e:E_+) & := & e = (+, f, g) \to (+, D(f), D(g)) \\
D_\times(e:E_\times) & := & e = (\times, f, g) \to (+, (\times, D(f), C(g)), (\times, C(f), D(g))) \\
D_{\exp}(e:E_{\exp}) & := & e = (\exp, f) \to (\times, C(e), D(f)) \\
 & & \\
C(e:E) & := & e \in E_0 \to 0,\ e \in E_1 \to 1,\ e \in E_x \to x, \\
 & & e \in E_+ \to C_+(e),\ e \in E_\times \to C_\times(e),\ e \in E_{\exp} \to C_{\exp}(e) \\
C_+(e:E_+) & := & e = (+, f, g) \to (+, C(f), C(g)) \\
C_\times(e:E_\times) & := & e = (\times, f, g) \to (\times, C(f), C(g)) \\
C_{\exp}(e:E_{\exp}) & := & e = (\exp, f) \to (\exp, C(f))
\end{array}
$$

Figure 1.3 : La définition du programme de dérivation formelle

En effet, pour une donnée quelconque $a$, $P(a)$ appelle $Q(a)$, qui lui-même appelle à nouveau $P(a)$, et ainsi de suite. Annonçons tout de suite l'objectif de cette sous-section.

**Théorème 2.** *Le caractère bien fondé des programmes est décidable dans la classe* Π.

Il ne nous reste plus qu'à fournir la preuve de ce théorème.

Les schémas définis précédemment confèrent aux procédures la propriété fondamentale suivante : les procédures de Π "déconstruisent" les objets créés par les constructeurs de Ω, mais sans les modifier. En d'autres termes, les procédures *descendent* dans l'arbre de dérivation des objets :
PROPRIÉTÉ DE DESCENTE : une procédure $P$ est dite de *descente* (resp. *descente stricte*) si pour toute donnée $a$ de $P$, tous les appels $Q(b)$ produits par l'évaluation de $P(a)$ sont tels que $b$ est une composante au sens large (resp. strict) de $a$.

Par exemple, la procédure définie par $P(x : X) := Q((x, x))$ telle que l'évaluation de $P$ sur l'objet $x$ entraîne l'évaluation de $Q$ sur la paire $(x, x)$ ne vérifie pas la propriété de descente, car $(x, x)$ n'est pas une composante de $x$. Pour peu que $Q(y)$ appelle à son tour $P(y)$, on voit qu'il va y avoir une explosion de la taille des données. La propriété de descente entraîne que tous les appels issus de $P(x)$ se font sur des composantes de $x$, donc sur des objets de taille inférieure. Pour s'assurer de la terminaison, il suffira de vérifier que l'on ne peut pas boucler sur des objets de même taille.

**Lemme 5.** *Si un programme de* Π *boucle, et que ses structures de données sont bien fondées, alors il existe un cycle de procédures* $(P_0, P_1, \ldots, P_k = P_0)$ *ayant toutes le même type, telles que le corps de* $P_i(x)$ *contient un appel* $P_{i+1}(x)$*, pour* $0 \leq i < k$*.*

**Démonstration :** Supposons qu'il existe une procédure $Q_0$ et un objet $y_0$ tels que l'évaluation de $Q_0(y_0)$ boucle. Il existe donc une séquence infinie d'appels $(Q_0(y_0), Q_1(y_1), \ldots, Q_j(y_j), \ldots)$ tels que l'évaluation de $Q_j$ sur $y_j$ entraîne celle de $Q_{j+1}$ sur $y_{j+1}$. A cause de la propriété de descente, $y_{j+1}$ est une composante de $y_j$. De plus, le nombre d'entiers $j$ pour lesquels $y_{j+1}$ est une composante au sens strict de $y_j$ est nécessairement fini, parce que l'objet initial $y_0$ est formé par un nombre fini de constructions. Il s'ensuit qu'à partir d'un certain rang, tous les $y_j$ sont égaux. Ensuite, l'existence d'un cycle découle de la finitude du nombre de procédures. Il existe donc un objet $y$ et un cycle $(P_0, P_1, \ldots, P_k = P_0)$ tel que $P_l(y)$ appelle $P_{l+1}(y)$ pour $0 \leq l < k$.

Soit $X_l$ le type des arguments de $P_l$. Les seuls schémas pour lesquels l'évaluation d'une procédure sur un objet $y$ entraîne celle d'une autre procédure sur le *même* objet sont l'exécution séquentielle, la sélection dans une union ainsi que les tests sur la taille et la longueur. Pour les types, cela correspond respectivement à $X_l \to X_{l+1}$ ou à $X_l \to X_{l+1} \cup Y_{l+1}$. Le caractère bien fondé des structures de données interdit le schéma de sélection dans une union, car on pourrait alors écrire $X_0 \to X_0 \cup Y$. Par conséquent, toutes les procédures ont le même type. ■

Par exemple, la figure 1.4 montre le graphe des appels associé au programme $(\pi_1)$. Il est immédiat

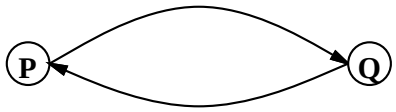



Figure 1.4 : Le graphe associé au programme $(\pi_1)$

de constater que ce graphe est cyclique. Nous avons à présent une condition nécessaire pour qu'un programme boucle. Cette condition n'est malheureusement pas suffisante, à cause des schémas de test sur la taille et sur la longueur. Par exemple, le programme

$$(\pi_2)\begin{cases} A \to a \\ P(a:A) := |a| \leq 5 \to R(a), \quad |a| > 5 \to Q(a) \\ Q(a:A) := |a| \leq 3 \to P(a), \quad |a| > 3 \to R(a) \\ R(a:A) := \mathrm{count}(1) \end{cases}$$

comporte bien un cycle $P(a) \to Q(a) \to P(a)$. Et pourtant ce programme termine, car $P$ appelle $Q$ lorsque $|a| > 5$, mais alors $Q$ appelle $R$, qui termine immédiatement. Le même phénomène advient pour le schéma de sélection en fonction de la longueur. Mais une fois encore, nous sommes sauvés par une condition de finitude.

**Lemme 6.** *La terminaison* (halting problem) *est décidable dans la classe* Π*, pour les programmes dont la définition des structures de données est bien fondée.*

**Démonstration :** Définissons l'ordre partiel $\prec_{n,k}$ entre deux procédures de même type $X$ : si $x$ est de taille $n$ et de longueur $k$, et que l'évaluation de $Q(x)$ entraîne celle de $P(x)$, alors $P \prec_{n,k} Q$.

Une condition nécessaire et suffisante pour qu'un programme boucle est qu'il existe deux entiers $n$ et $k$ et un cycle de procédures $P \prec_{n,k} Q \prec_{n,k} \cdots \prec_{n,k} P$ de même type $X$ tels que $X$ dérive au moins un objet $x$ de taille $n$ et de longueur $k$. L'aspect nécessaire est évident (il suffit de prendre pour $x$ l'objet $y$ de la preuve du lemme précédent). Réciproquement, étant donné un tel objet $x$, on voit immédiatement que $P(x)$ boucle.

Montrons maintenant la décidabilité. Comme pour les données, la suite double $(\prec_{n,k})$ est stationnaire : si $N$ et $K$ sont les plus grands entiers apparaissant dans les tests sur la taille et la longueur respectivement, alors le graphe de $(\prec_{n,k})$ est constant dans les régions $(n > N, k > K)$, ($n$ fixé $\leq N, k > K$), ($n > N, k$ fixé $\leq K$). Il y a au plus $(N+2)(K+2)$ graphes différents à calculer. Pour chaque graphe, il faut chercher les cycles, et pour chaque cycle de procédures de type $X$, déterminer si $X$ dérive un objet de taille $n$ (ou $> N$) et de longueur $k$ (ou $> K$). Ceci est décidable lorsque la spécification des données est bien fondée (lemme 4). ■

Par exemple, pour le programme $(\pi_2)$, la longueur n'influe pas (on peut considérer que $K$ vaut $-1$), et $N$ vaut 5. En fait, il n'y a que 3 régions à étudier : $0 \leq n \leq 3$, $3 < n \leq 5$ et $5 < n$. Pour la première région, lorsque la taille de la donnée $a$ est inférieure ou égale à 3, le graphe de la relation $\prec_{n,k}$ contient un arc de $P$ vers $R$, et un de $Q$ vers $P$. Dans la seconde région, il y a un arc de $P$ vers $R$ et un autre de $Q$ vers $R$. Enfin, dans la troisième région, il y a un arc de $P$ vers $Q$, et un autre de $Q$ vers $R$. Aucun de ces trois graphes n'admettant de cycle, le programme $(\pi_2)$ termine.

Grâce à ce dernier lemme, nous pouvons maintenant prouver le théorème 2.

**Démonstration :** (théorème 2) La preuve est constituée par l'algorithme ci-dessous, qui détermine si un programme est bien fondé.

**Algorithme C** :
Donnée : un programme de Π (une spécification Σ et un ensemble de procédures).
Résultat : *vrai* si le programme est bien fondé, *faux* sinon.

1. vérifier le caractère bien fondé de Σ par l'algorithme $B$,
   si Σ n'est pas bien fondée, renvoyer *faux* ;
2. $N \leftarrow$ plus grand entier apparaissant dans les tests de taille,
   $K \leftarrow$ plus grand entier apparaissant dans les tests de longueur,
3. pour chacune des $(N+2)(K+2)$ régions $(n = 0 \ldots N, n > N; k = 0 \ldots K, k > K)$ :
   pour chaque cycle du graphe de l'ordre partiel $\prec_{n,k}$ défini au lemme 6 :
   soit $X$ le type commun des procédures du cycle,
   si $X$ dérive un objet de taille $n$ et de longueur $k$, renvoyer *faux* ;
4. renvoyer *vrai*.

■

Lorsqu'il n'y a ni test sur la taille, ni test sur la longueur, nous avons $N = K = -1$ à l'étape (2), et il y a une seule région à étudier, donc un seul ordre partiel $\prec$. Les programmes calculant le nombre de sommants d'une partition ou la longueur du cycle d'un graphe connexe monocyclique sont de "mauvais" exemples pour l'algorithme $C$ : il y a une seule procédure, et celle-ci comporte une seule instruction, qui est du genre **forall** $a$ **in** $c$ **do** *count*. Le graphe de la relation $\prec$ ne contient alors qu'un seul point, et aucun arc, donc ces deux programmes sont bien fondés.

Le programme de dérivation formelle est plus intéressant, puisqu'il comporte deux procédures, `diff` et `copy`, qui s'appellent récursivement.

EXEMPLE 12 : (Dérivation formelle) Comme il n'y a aucun test sur la taille ni sur la longueur, il y a une seule région à étudier. Nous rappelons (cf preuve du lemme 6) que $P \prec Q$ si $(i)$ $P$ et $Q$ sont de même type, $(ii)$ l'évaluation de $Q(x)$ entraîne celle de $P(x)$ (un moyen mnémotechnique pour se souvenir du sens de la relation $\prec$ est de voir le signe $\prec$ comme la pointe d'une flèche dirigée de la procédure appelant vers la procédure appelée).

```
procedure diff (e : expression);
begin
  case e of
    zero      : write(zero);
    un        : write(zero);
    x         : write(un);
    (plus,f,g)  : begin write(plus); diff(f); diff(g) end;
    (mult,f,g)  : begin write(plus);write(mult);diff(f);copy(g);write(mult);copy(f);diff(g) end;
    (expo,f)    : begin write(mult); write(expo); copy(f); diff(f) end;
  end
end;

procedure copy (e : expression);
begin
  case e of
    zero      : write(zero);
    un        : write(un);
    x         : write(x);
    (plus,f,g)  : begin write(plus); copy(f); copy(g) end;
    (mult,f,g)  : begin write(mult); copy(f); copy(g) end;
    (expo,f)    : begin write(expo); copy(f) end;
  end
end;
```

Dans le corps de la procédure `diff`, il n'y a aucun appel de procédure avec `e` comme argument, et il en est de même dans celui de `copy`. Par conséquent, le graphe de $\prec$ ne comprenant aucun arc, le programme est bien fondé.

REMARQUE : Pour dériver une exponentielle, à la place de `write(expo); copy(f)`, nous aurions pu écrire tout simplement `copy(e)`. En ce qui concerne le caractère bien fondé, cela ajouterait un arc de `diff` vers `copy`, mais le programme serait toujours bien fondé.

□

EXEMPLE 13 :

$$(\pi_3)\begin{cases} A \rightarrow x \mid B \\ B \rightarrow xA \\ |x| = 1 \\ \\ P(a:A) := Q(a);R(a) \\ Q(a:A) := R(a); a \in B \rightarrow S(a) \\ R(a:A) := a \in B \rightarrow S(a) \\ S(b:B) := b = (x,a) \rightarrow P(a) \end{cases}$$

Le calcul des valuations à l'aide de l'algorithme $A$ donne $\mathrm{val}(A) = 1$ et $\mathrm{val}(B) = 2$. On vérifie ensuite à l'aide du lemme 3 que les structures de données sont bien fondées. Enfin, il ne reste plus qu'à s'assurer que les procédures terminent. Comme ce programme ne comporte aucun test sur la

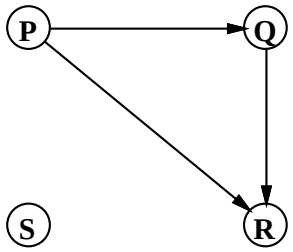



Figure 1.5 : Le graphe associé au programme ($\pi_3$)

taille ni sur la longueur, il y a un seul graphe à étudier. Les relations sont $Q \prec P$, $R \prec P$ et $R \prec Q$ (bien que $S(a)$ apparaisse dans la définition de $Q$, il ne peut y avoir de relation entre ces procédures car elles ont des *types différents*). Le graphe étant acyclique (figure 1.5), le programme ($\pi_3$) est bien fondé, et termine pour toute donnée. □

## 1.6 Adl : un langage pour décrire les algorithmes

Dans cette section, nous introduisons un langage de définition de programmes. Ce langage, dont le nom est Adl (*Algorithm Description Language*, c'est-à-dire langage de description d'algorithmes en anglais) permet d'écrire tous les programmes de la classe II définie ci-dessus. Par construction, ce langage est parfaitement adapté pour l'analyse en moyenne d'algorithmes (cf chapitre suivant).

### 1.6.1 Motivation

Les programmes tels que nous les avons définis dans la section précédente ne ressemblent pas beaucoup aux programmes d'un langage réel tel que Pascal, C ou Lisp. Par exemple, le programme calculant le nombre de sommants d'une partition s'écrit en utilisant les notations de la figure 1.2 :

$$\begin{cases} P & \to & \mathcal{M}(E) \\ E & \to & u\,u^* \\ |u| & = & 1 \\ \\ CS(p:P) & := & p = \{\!\{e_1, \ldots, e_k\}\!\} \to \text{count}, \ldots, \text{count} \end{cases}$$

Le langage Adl permet d'écrire les programmes de la classe II sous une forme plus lisible, plus proche des langages de programmation, et sans utiliser de caractères spéciaux. Par exemple, le programme ci-dessus s'écrit en Adl de la façon suivante :

```
type P = multiset(E);
     E = sequence(u,card>=1);
     u = atom(1);

procedure CS (p : P);
begin
   forall e in p do
      count
```

| constructeur | notation mathématique | syntaxe Adl |
|---|---|---|
| union | $\mid$ | $\mid$ |
| produit cartésien | $\times$ | `product` |
| séquence | $^*$ | `sequence` |
| ensemble | $\mathcal{P}$ | `set` |
| multi-ensemble | $\mathcal{M}$ | `multiset` |
| cycle orienté | $\mathcal{C}$ | `cycle` |
| atome (de taille $k$) | $\mid \cdot \mid = k$ | `atom`$(k)$ |

Figure 1.6 : Les constructeurs de la classe $\Omega$

**end**;

Cet exemple nous montre qu'un programme Adl comporte deux parties : la définition des structures de données, puis la définition du programme proprement dit, c'est-à-dire d'une ou de plusieurs procédures.

### 1.6.2 Définition des structures de données en Adl

La définition des structures de données commence par le mot-clé **type**, puis comporte une suite de définitions correspondant aux productions de la spécification sous-jacente. Chaque définition se termine par un point-virgule. Comme l'exemple ci-dessus le montre, les flèches ($\rightarrow$) de la grammaire sont devenues des égalités ($=$), les unions sont toujours désignées par le symbole "|", le constructeur *produit cartésien* est indiqué par le mot `product`, et les atomes sont mis en évidence à l'aide du mot `atom` qui permet également de préciser leur taille. Le tableau de la figure 1.6 indique les notations Adl correspondant à chacun des constructeurs étudiés jusqu'ici.

### 1.6.3 Définition des procédures en Adl

Une procédure commence par le mot-clé **procedure**, puis vient le nom de la procédure, le nom de son argument et son type, comme en Pascal. Le corps de la procédure est délimité par les mots-clés **begin** et **end**. Les instructions sont séparées par des points-virgules (schéma d'exécution séquentielle). La sélection dans une union s'exprime à l'aide du mot-clé **casetype**, et la sélection dans un produit cartésien à l'aide du mot-clé **case**, qui permet de surcroît de nommer les composantes du produit. La sélection et l'itération dans des séquences, ensembles ou cycles s'expriment respectivement par les mots-clés **forone** et **forall**. Le tableau de la figure 1.7 indique comment formuler en Adl les schémas de programmation de la classe II définis dans la précédente section.

#### De la souplesse dans les définitions

Le programme de dérivation formelle de la page 14 ne vérifie pas la définition stricte du langage Adl donnée ci-dessus. En effet, si nous examinons de plus près le corps de la procédure `diff`,

```
procedure diff (e : expression);
begin
  case e of
```

| schéma | | notation mathématique | syntaxe Adl |
|---|---|---|---|
| déclaration de procédure | | $P(a : A) :=< \text{corps} >$ | **procedure** P(a : A);<br>**begin**<br><instruction><br>**end** |
| exécution séquentielle | | $Q(a); R(a)$ | Q(a);<br>R(a) |
| union | sélection | $c \in A \rightarrow Q(c)$ | **casetype** c **of**<br>A : Q(c)<br>**end** |
| produit cartésien | sélection | $c = (a, b) \rightarrow Q(a)$ | **case** c **of**<br>(a,b) : Q(a)<br>**end** |
| séquence, ensemble, multi-ensemble, cycle | sélection | $c = (a_1, \ldots, a_k) \rightarrow Q(a_{\text{rnd}(1\ldots k)})$ | **forone** a **in** c **do**<br>Q(a) |
| | itération | $c = (a_1, \ldots, a_k) \rightarrow Q(a_1); \ldots; Q(a_k)$ | **forall** a **in** c **do**<br>Q(a) |
| test sur la taille | | $\|a\| \leq k \rightarrow Q(a),\ \|a\| > k \rightarrow R(a)$ | **if** size($a$) <= $k$ **then** $Q(a)$<br>**else** $R(a)$ |
| test sur la longueur | | $l(a) \leq k \rightarrow Q(a),\ l(a) > k \rightarrow R(a)$ | **if** card($a$) <= $k$ **then** $Q(a)$<br>**else** $R(a)$ |

Figure 1.7 : Les schémas de programmation de la classe II

```
    zero       : write(zero);
    un         : write(zero);
    x          : write(un);
    (plus,f,g) : begin write(plus); diff(f); diff(g) end;
    (mult,f,g) : begin write(plus);write(mult);diff(f);copy(g);write(mult);copy(f);diff(g) end;
    (expo,f)   : begin write(mult); write(expo); copy(f); diff(f) end;
  end
end;
```

nous constatons que le schéma **case** $e$ **of** ... **end** est utilisé à la fois comme schéma de sélection dans une union (les différents types d'expressions), et comme schéma de sélection dans un produit cartésien (pour les sommes, produits et exponentielles). Pour définir cette procédure en respectant *strictement* la définition du langage, il faut redéfinir le type `expression` :

```
type expression = zero | un | x | somme | produit | exponentielle;
     somme = plus expression expression;
     produit = mult expression expression;
     exponentielle = expo expression;

procedure diff (e : expression);
begin
  case e of
    zero        : write(zero);
    un          : write(zero);
    x           : write(un);
```

```
        somme       : diff_somme(e);
        produit     : diff_produit(e);
        exponentielle : diff_exponentielle(e);
    end
end;
```

et définir également trois autres procédures `diff_somme`, `diff_produit` et `diff_exponentielle` dérivant respectivement des sommes, des produits et des exponentielles. Cette transformation avait d'ailleurs déjà été faite pour écrire le programme de dérivation avec les notations "mathématiques" de la classe II (figure 1.3 page 39). Dans les chapitres qui suivent, nous étudierons des programmes Adl sous forme non stricte, à condition qu'ils aient une forme stricte évidente comme ci-dessus.

### 1.6.4 Coût des programmes

Il existe plusieurs notions de complexité pour un programme. Les notions les plus couramment utilisées sont la complexité en temps et la complexité en espace. Dans ce travail, nous nous intéressons uniquement à une notion de complexité *additive*, représentant par exemple le temps d'exécution d'un programme sur une machine séquentielle :

$$\text{coût}[\text{calcul}_1 ; \text{calcul}_2] = \text{coût}[\text{calcul}_1] + \text{coût}[\text{calcul}_2].$$

A partir de maintenant, le terme "coût" désigne une notion de complexité additive.

Pour définir le coût d'un programme, il suffit de spécifier le coût des opérations de base, comme l'addition de deux entiers, l'incrémentation d'un compteur, un branchement pour l'appel d'une sous-routine, la comparaison de deux caractères. Par exemple, sur une machine donnée, la routine d'addition `add` utilisera deux cycles d'horloge, l'incrémentation `incr` d'un compteur de boucle en utilisera un seul, un appel de sous-routine `jsr` coûtera cinq cycles, et la comparaison `comp` de caractères en nécessitera trois. Dans le langage Adl, ceci se définira à l'aide du mot-clé **measure** de la façon suivante :

```
measure add : 2; incr : 1; jsr : 5; comp : 3;
```

La définition des coûts des instructions élémentaires s'interprète de deux manières. Soit on désire connaître le temps d'exécution du programme sur une machine réelle, et alors les coûts élémentaires sont les temps d'exécution des instructions de cette machine ; soit on désire étudier une certaine quantité *additive* liée à l'exécution du programme, et alors on placera des instructions élémentaires *fictives* aux endroits stratégiques, et l'on définira de façon adéquate leur coût. Illustrons cette seconde utilisation sur un exemple. Considérons à nouveau le programme de dérivation défini au début de ce chapitre, et supposons que l'on veuille étudier non pas le nombre total de symboles des expressions dérivées, mais seulement le nombre d'occurrences des opérateurs binaires `plus` et `mult`. Alors au lieu de `measure write : 1`, on déclarera

```
measure plus, mult : 1;
```

et chaque fois que l'un des ces symboles apparaîtra au cours de l'évaluation de `diff` ou de `copy`, il sera compté un coût d'une unité.

Ce mode de spécification est très souple, et permet de définir des modèles de complexité très variés (quoique tous additifs), comme le nombre d'appels d'une procédure donnée ou le nombre de passages en un endroit donné du programme.

# Chapitre 2

# Analyse automatique dans Π

*In medias res.*
Horace, Art Poétique, 148

Dans ce chapitre, nous montrons que la classe Π se prête à l'analyse automatique. Plus précisément, tout programme de Π se traduit, par des règles que nous précisons, en un système d'équations pour des fonctions d'une variable complexe ; à partir de ces équations, le coût moyen du programme sur les données de taille $n$ s'obtient par des algorithmes de complexité polynomiale en $n$ (section 2.3). De plus, ces équations permettent d'obtenir ultérieurement un développement asymptotique du coût moyen lorsque $n$ tend vers l'infini [Sal91].

Le programme suivant détermine la *longueur de cheminement* d'un arbre binaire, c'est-à-dire la somme des distances de chaque nœud à la racine de l'arbre. La longueur de cheminement intervient souvent dans le coût d'algorithmes sur des arbres ; par exemple si l'arbre est la représentation préfixe d'un dictionnaire, la longueur de cheminement est la taille qu'aurait le dictionnaire si l'on écrivait les mots l'un à la suite de l'autre.

```
type tree = node | product(node, tree, tree);
     node = atom(1);

procedure size (t : tree);
begin
  case t of
    node          : count;
    (node,u,v)    : begin count; size(u); size(v); end;
  end;
end;

procedure pathlength (t : tree);
begin
  case t of
    node          : nil;
    (node,u,v)    : begin size(u); size(v); pathlength(u); pathlength(v) end;
  end;
end;

measure count : 1;
```

L'analyse algébrique d'un tel programme s'effectue en deux phases. En premier lieu, la spécification des structures de données (**type** ...) se traduit en des équations pour les séries génératrices associées aux différents types. Si $T(z)$ est la série du type `tree`, et $N(z)$ la série du type `node`, il s'ensuit dans le cas de notre exemple :

$$\begin{cases} \text{tree} = \text{node} \mid \mathit{product}(\text{node}, \text{tree}, \text{tree}); \\ \text{node} = \mathit{atom}(1); \end{cases} \quad \Longrightarrow \quad \begin{cases} T(z) = N(z) + N(z)T(z)^2, \\ N(z) = z. \end{cases} \tag{2.1}$$

Les règles qui permettent de passer automatiquement de la première partie du programme Adl (définition des types) au système d'équations (2.1) sont l'objet de la section 2.1.

En second lieu, les définitions de procédures se traduisent en équations pour les séries génératrices de coût associées aux procédures (*descripteurs de complexité*). Si l'on note $\tau S(z)$ le descripteur de complexité de la procédure `size`, et $\tau P(z)$ celui de `pathlength`, la correspondance est la suivante :

$$\begin{cases} \textbf{procedure}\ \text{size}(t : \text{tree}); \\ \textbf{case}\ t\ \textbf{of} \\ \quad \text{node} : \text{count}; \\ \quad (\text{node}, u, v) : \ \textbf{begin}\ \text{count}; \\ \qquad\qquad \text{size}(u); \text{size}(v)\ \textbf{end} \\ \textbf{end} \\ \\ \textbf{procedure}\ \text{pathlength}(t : \text{tree}); \\ \textbf{case}\ t\ \textbf{of} \\ \quad \text{node} : \text{nil}; \\ \quad (\text{node}, u, v) : \ \textbf{begin}\ \text{size}(u); \text{size}(v); \\ \qquad\qquad \text{pathlength}(u); \text{pathlength}(v)\ \textbf{end} \\ \textbf{end} \end{cases} \quad \Longrightarrow \quad \begin{cases} \tau S(z) \\ = \\ N(z) \\ +N(z)T(z)^2 \\ +2N(z)T(z)\tau S(z) \\ \\ \\ \tau P(z) \\ = \\ 0 \\ +2N(z)T(z)\tau S(z) \\ +2N(z)T(z)\tau P(z) \end{cases} \tag{2.2}$$

La section 2.2 indiquera quelles sont les règles qui autorisent l'automatisation de cette traduction.

Pour les programmes de la classe Π, nous verrons qu'il est toujours possible d'obtenir les systèmes (2.1) et (2.2). Ensuite, deux cas peuvent se présenter :

- les équations se résolvent à l'aide des fonctions usuelles ($+$, $-$, $\times$, $/$, $\sqrt{}$, exp, log, ...) : c'est le cas *explicite* ;
- il n'y a pas de forme explicite pour les séries génératrices ou pour les descripteurs de complexité : c'est le cas *implicite*.

Pour le programme calculant la longueur de cheminement dans un arbre binaire, nous sommes dans le cas explicite. Les systèmes (2.1) et (2.2) se résolvent en effet :

$$N(z) = z, \qquad T(z) = \frac{1 - \sqrt{1 - 4z^2}}{2z},$$

$$\tau S(z) = \frac{-1 + 4z^2 + \sqrt{1 - 4z^2}}{2z(1 - 4z^2)}, \qquad \tau P(z) = \frac{1 - 2z^2 - \sqrt{1 - 4z^2}}{z(1 - 4z^2)}.$$

Ces séries génératrices renferment toutes les informations nécessaires sur la complexité en moyenne du programme. Nous pouvons les utiliser de deux façons :

**analyse exacte :** l'extraction des coefficients de $z^n$ dans les séries génératrices donne le coût moyen exact du programme sur les données de taille $n$. Dans le cas explicite, cette extraction se fait sans effort à l'aide d'un bon système de calcul formel effectuant des développements de Taylor (par ex. MAPLE). Ainsi la longueur de cheminement moyenne des arbres binaires de 101 nœuds (tels que définis, les arbres binaires sont tous de taille impaire) est

$$\overline{\tau P}_{101} = \frac{2669615431987142938701754963 30}{24728270721952008170297 1807} \simeq 1079.580316.$$

Dans le cas implicite, l'extraction se fait à partir des équations, mais le nombre d'opérations à effectuer pour déterminer le coût moyen exact demeure polynomial en $n$ (section 2.3).

**analyse asymptotique :** l'analyse des singularités des séries génératrices, considérées alors comme fonctions de la variable complexe $z$, révèle le comportement asymptotique du coefficient de $z^n$ lorsque $n$ tend vers l'infini. Cette analyse peut se faire aussi bien sur les formes explicites que sur les équations. Pour une sous-classe importante de Π, il est donc possible de déterminer automatiquement un développement asymptotique du coût moyen. Par exemple, la longueur de cheminement d'un arbre binaire de $n$ nœuds vaut en moyenne

$$\overline{\tau P}_n = \sqrt{\frac{\pi}{2}} n^{3/2} + O(n).$$

Mais l'analyse asymptotique des fonctions d'une variable complexe constitue à elle seule un vaste sujet de recherche ; aussi nous renvoyons le lecteur à la thèse de B. Salvy [Sal91] pour de plus amples informations.

En résumé, l'analyse algébrique ramène l'analyse en moyenne sur les données de taille $n$ d'un coût exponentiel en $n$ (en évaluant le programme sur toutes les données de taille $n$) à un coût polynomial (calcul du coefficient de $z^n$ dans une série génératrice) ; l'analyse asymptotique permet de passer d'un coût polynomial (pour calculer la complexité moyenne exacte) à un coût constant (pour calculer un équivalent de la complexité moyenne lorsque $n$ tend vers l'infini).

## 2.1 Analyse des structures de données

Les constructeurs de la classe Ω ont une propriété particulière : le nombre d'objets de taille donnée dérivés par une production $X \to \Phi(Y, Z, \ldots)$ dépend *uniquement* du *nombre* d'objets de taille inférieure dérivés par $Y$ et $Z$, et non de la nature de ces objets. Grâce à cette propriété fondamentale, pour compter les objets, il nous suffit de connaître les *suites de dénombrement.*

**Définition 10.** *La* suite de dénombrement *associée à un ensemble $X$ est la suite $(X_n)_{n\in\mathbb{N}}$, où $X_n$ est le nombre d'objets de taille $n$ de $X$. Un constructeur $\Phi$ est dit* admissible *si la suite de dénombrement de*

$$X \to \Phi(Y, Z)$$

*ne dépend que des suites de dénombrement de $Y$ et $Z$.*

Par exemple, si $A \to B \times C$, alors un objet de taille $n$ de $A$ a une composante de taille $k$ dans $B$ et une de taille $n - k$ dans $C$, d'où

$$A_n = \sum_{k=0}^{n} B_k C_{n-k}. \tag{2.3}$$

L'entier $A_n$ dépend uniquement des entiers $B_k$ et $C_k$, pour $0 \leq k \leq n$, donc le constructeur *produit cartésien* est admissible. La relation (2.3) s'écrit de manière plus concise en termes de séries génératrices.

**Définition 11.** *La série génératrice ordinaire*[1] *associée à un ensemble* $\mathcal{A}$ *est*

$$A(z) = \sum_{a \in \mathcal{A}} z^{|a|} = \sum_{n=0}^{\infty} A_n z^n.$$

Étant donné un ensemble $\mathcal{A}$ de structures combinatoires dont on connaît la série génératrice ordinaire $A(z)$, le nombre de structures de taille $n$ est donné par le coefficient de $z^n$ dans $A(z)$, noté usuellement $[z^n]A(z)$. En termes de séries génératrices, l'équation (2.3) devient tout simplement

$$A(z) = B(z)C(z).$$

En résumé, la production $A \rightarrow B \times C$ se traduit dans le domaine des séries génératrices ordinaires par l'équation $A(z) = B(z)C(z)$. Nous noterons cette correspondance par une *règle*, en l'occurrence :

$$\frac{A \rightarrow B \times C}{A(z) = B(z)C(z).}$$

### 2.1.1 Constructions de base

Comme annoncé plus haut, les constructeurs de la classe $\Omega$ sont tous admissibles, et les règles correspondantes sont indiquées ci-dessous. Les règles 1 à 6 sont classiques (voir par exemple [Zim88] ou [VF90]). La règle 7 (cycle orienté) est quant à elle démontrée dans [Rea61], et une preuve combinatoire est donnée dans [FS91].

**Règle 1.** *(Atome)*

$$\frac{\textbf{type } A = atom(k);}{A(z) = z^k}$$

**Démonstration :** $A(z) = z^{|atom(k)|} = z^k$. ∎

**Règle 2.** *(Union)*

$$\frac{\textbf{type } A = B \mid C;}{A(z) = B(z) + C(z)}$$

**Démonstration :** $A(z) = \sum_{a \in \mathcal{A}} z^{|a|} = \sum_{b \in \mathcal{B}} z^{|b|} + \sum_{c \in \mathcal{C}} z^{|c|} = B(z) + C(z)$. ∎

**Règle 3.** *(Produit cartésien)*

$$\frac{\textbf{type } A = product(B, C);}{A(z) = B(z)C(z)}$$

[1] Il existe d'autres sortes de séries génératrices : les séries *exponentielles* utilisées dans le prochain chapitre, les séries de Dirichlet utilisées dans l'annexe A, les séries Eulériennes, les séries doublement exponentielles, les séries chromatiques [Sta78]. Dans ce chapitre, nous dirons simplement série génératrice en parlant de série génératrice ordinaire.

**Démonstration :** $\sum_{a\in\mathcal{A}} z^{|a|} = \sum_{(b,c)} z^{|(b,c)|} = \sum_{(b,c)} z^{|b|+|c|} = (\sum_{b\in\mathcal{B}} z^{|b|})(\sum_{c\in\mathcal{C}} z^{|c|})$. ■

**Règle 4.** *(Séquence)*

$$\frac{\textbf{type } A = sequence(B);}{A(z) = Q(B(z))}$$

*où* $Q(f) = \frac{1}{1-f}$.

**Démonstration :** $A(z) = \sum_{k\geq 0} \sum_{b_1,\ldots,b_k\in\mathcal{B}} z^{|(b_1,\ldots,b_k)|} = \sum_{k\geq 0} \sum_{b_1,\ldots,b_k\in\mathcal{B}} z^{|b_1|+\cdots+|b_k|} = \sum_{k\geq 0} B(z)^k$. ■

**Règle 5.** *(Ensemble)*

$$\frac{\textbf{type } A = set(B);}{A(z) = \Psi_S(B)(z)}$$

*où*

$$\Psi_S(f)(z) = \exp(\frac{f(z)}{1} - \frac{f(z^2)}{2} + \frac{f(z^3)}{3} - \cdots).$$

**Démonstration :** L'ensemble $\mathcal{B}$ étant dénombrable (caractère bien fondé, condition $(ii)$ de la définition 5), il existe une bijection $\varphi : \mathbb{N} \to \mathcal{B}$. Associons alors à chaque élément $E = \{b_1, \ldots, b_k\}$ de $\mathcal{A}$ la séquence infinie $(e_0, e_1, e_2, \ldots)$, où si l'élément $b = \{\varphi(n)\}$ est dans $E$, alors $e_n = \{b\}$, sinon $e_n = \{\}$. Nous définissons ainsi une bijection entre l'ensemble $\mathcal{A}$ et un produit cartésien infini

$$\mathcal{A} \approx \prod_{b\in\mathcal{B}} (\{\} \cup \{b\}), \tag{2.4}$$

et de plus cette bijection conserve la taille. Par les règles 2 et 3, il s'ensuit

$$A(z) = \prod_{b\in\mathcal{B}} (1 + z^{|b|}) = \prod_{n\geq 1} (1 + z^n)^{B_n}.$$

La forme donnée par la règle s'obtient par passage au logarithme, développement en série des logarithmes et interversion des sommes :

$$\log A(z) = \sum_{n\geq 1} B_n \log(1 + z^n) = \sum_{n\geq 1} B_n \sum_{k\geq 1} \frac{(-1)^{k+1}}{k} z^{kn} = \sum_{k\geq 1} \frac{(-1)^{k+1}}{k} B(z^k).$$ ■

**Règle 6.** *(Multi-ensemble)*

$$\frac{\textbf{type } A = multiset(B);}{A(z) = \Psi_M(B)(z)}$$

*où*

$$\Psi_M(f)(z) = \exp(\frac{f(z)}{1} + \frac{f(z^2)}{2} + \frac{f(z^3)}{3} + \cdots).$$

**Démonstration :** La démonstration est similaire à celle ci-dessus. On établit d'abord une bijection conservant la taille :

$$\mathcal{A} \approx \prod_{b\in\mathcal{B}} (\{\} \cup \{b\} \cup \{b, b\} \cup \cdots), \tag{2.5}$$

puis on utilise les règles 2 et 3 pour passer aux séries génératrices :

$$A(z) = \prod_{b \in \mathcal{B}} \frac{1}{1 - z^{|b|}} = \prod_{n} \frac{1}{(1 - z^n)^{B_n}}.$$

Le passage au logarithme donne la forme exponentielle indiquée. ∎

Il faut ici noter l'importance des formules (2.4) et (2.5) : elles montrent que les constructions ensemble et multi-ensemble se déduisent de l'union et du produit cartésien (infinis). Ces bijections nous serviront par la suite, pour déterminer le coût de procédures dont les données sont des ensembles ou des multi-ensembles.

**Règle 7.** *(Cycle orienté)*

$$\frac{\textbf{type } A = \mathit{cycle}(B);}{A(z) = \Psi_C(B)(z)}$$

*où*

$$\Psi_C(f)(z) = \sum_{k \geq 1} \frac{\phi(k)}{k} \log \frac{1}{1 - f(z^k)}$$

*($\phi$ est la fonction d'Euler : $\phi(k)$ est le nombre d'entiers positifs inférieurs à $k$ et premiers avec $k$ : $\phi(1) = 1, \phi(2) = 1, \phi(3) = 2, \phi(4) = 2, \phi(5) = 4, \ldots$).*

**Démonstration :** Voir la section A.2 de l'annexe A. ∎

Ainsi, la classe Ω (spécifications en univers non étiqueté dont les productions utilisent les constructeurs union, produit cartésien, séquence, ensemble, multi-ensemble et cycle orienté) se prête à l'analyse automatique par les règles 1 à 7 :

**Théorème 3.** *Toute spécification de Ω (cf définition 4) se traduit en un système d'équations pour les séries génératrices ordinaires associées aux structures de données. Les équations de ce système sont formées à partir de 1 et z par application des opérateurs +, ×, Q, $\Psi_C$, $\Psi_S$ et $\Psi_M$.*

Nous appellerons $\mathcal{UR}$ l'ensemble des fonctions solutions de tels systèmes, en conformité avec la notation introduite dans l'article [FSZ91] ($\mathcal{UR}$ est l'abréviation de *Unlabelled Recursive*, c'est-à-dire non étiqueté récursif). Les opérateurs $\Psi_C$, $\Psi_S$ et $\Psi_M$ correspondant aux cycles orientés, ensembles et multi-ensembles sont appelés *opérateurs de Pólya*.

**Démonstration :** Le système d'équations évoqué dans l'énoncé s'obtient par application des règles 1 à 7, qui s'écrivent bien à l'aide des opérateurs cités. ∎

EXEMPLE 14 : (Arbres binaires) Nous pouvons maintenant justifier les équations des séries de dénombrement de l'exemple introductif de ce chapitre (longueur de cheminement). La spécification des arbres binaires était :

**type** tree = node | *product*(node, tree, tree);
node = *atom*(1);

Soit $T(z)$ la série génératrice ordinaire du type `tree`, et $N(z)$ celle du type `node` : les règles 1 (atomes), 2 (union) et 3 (produit cartésien) s'appliquent ici, pour donner les équations

$$T(z) = N(z) + N(z)T(z)^2, \qquad N(z) = z.$$

Après substitution de $N(z)$, nous obtenons pour $T(z)$ une équation du second degré ; l'une des solutions de cette équation est infinie en $z = 0$, donc ne peut pas être une série de dénombrement, pour laquelle la valeur en 0 est le nombre d'objets de taille zéro. L'autre solution est

$$T(z) = \frac{1 - \sqrt{1 - 4z^2}}{2z}.$$

□

EXEMPLE 15 : Grâce aux trois premières règles, nous pouvons calculer la série génératrice $E(z)$ associée au type `expression` du programme de dérivation formelle (page 14) :

**type** expression = zero | un | x | *product*(plus,expression,expression)
| *product*(mult,expression,expression) | *product*(expo,expression);
zero, un, x, plus, mult, expo = *atom*(1);

Tous les atomes étant de taille 1, par la règle 1, la série associée est $z$. Convenons de noter $\Xi_a$ la série associée à l'atome `a` :

$$\Xi_{\text{zero}}(z) = \Xi_{\text{one}}(z) = \Xi_{\text{x}}(z) = \Xi_{\text{plus}}(z) = \Xi_{\text{mult}}(z) = \Xi_{\text{expo}}(z) = z,$$

et par exemple la production `product(mult,expression,expression)` se traduit d'après la règle 3 en $\Xi_{\text{mult}}(z)E^2(z)$. En utilisant la règle 2, nous obtenons alors

$$E(z) = \Xi_{\text{zero}}(z) + \Xi_{\text{one}}(z) + \Xi_{\text{x}}(z) + \Xi_{\text{plus}}(z)E^2(z) + \Xi_{\text{mult}}(z)E^2(z) + \Xi_{\text{expo}}(z)E(z).$$

Après substitution et simplification, nous trouvons l'équation

$$E(z) = 3z + zE(z) + 2zE^2(z). \tag{2.6}$$

La résolution de cette équation du second degré permet de déterminer une forme explicite pour la série $E(z)$ :

$$E(z) = \frac{1 - z - \sqrt{1-2z-23z^2}}{4z}. \tag{2.7}$$

□

L'exemple suivant est un cas où il n'y a pas de forme explicite simple à l'aide des fonctions usuelles.

EXEMPLE 16 : (Arbres généraux non planaires) Un arbre est dit *général* lorsque le nombre de ses sous-arbres est variable, et il est dit *non planaire* lorsque les sous-arbres ne sont pas ordonnés. Ainsi, un arbre général non planaire est un nœud auquel est rattaché un ensemble de $k$ sous-arbres ($k \geq 0$) pouvant être identiques, c'est-à-dire en fait un *multi-ensemble*. Lorsque $k = 0$, l'arbre est une feuille :

**type** gentree = node *multiset*(gentree);
node = *atom*(1);

La série ordinaire $G(z)$ des arbres généraux non planaires vérifie donc l'équation

$$G(z) = ze^{G(z)+\frac{G(z^2)}{2}+\frac{G(z^3)}{3}+\cdots}. \tag{2.8}$$

Même si elle ne donne pas une forme explicite pour $G(z)$, cette équation permet de calculer son coefficient de $z^n$, c'est-à-dire le nombre d'arbres généraux de taille $n$, en temps polynomial en $n$

(voir section 2.3). De surcroît, l'analyse asymptotique de $G(z)$ est possible à partir de (2.8) par la méthode générale exposée dans la section 5.5). □

A elle seule, la classe Ω permet de modéliser des problèmes dont la solution est non triviale, comme le lecteur pourra le constater en considérant l'exemple qui suit (problème numéro 18 du *CookBook* 1989 [FSZ89a]).

EXEMPLE 17 : Le problème posé par Steven Bird, de l'université de sciences cognitives d'Edimbourg (Écosse) est le suivant [Bir89] :

```
The recursively defined function:

f(m,n) = f(m,n-1) + f(m-1,n-1) + f(m-1,n) (m,n > 1)
f(m,1) = f(1,n) = 1

is the number of ways of drawing straight lines between a row consisting of
m points and a row of n points so that (i) each point has at least one line
coming from it, and (ii) no lines cross.  The function f(m+1,n+1) is also the
number of ways of extending a partial ordering: a<a1<...<am<b, a<b1<...<bn<b
to a total ordering (allowing for the possibility that ai=bj for some i,j).
It appears that the function g(m) = f(m,n) is a polynomial of degree n-1.
The function f(n,n) is exponential, tending towards (2 + 8^1/2)^n.
Is there a non-recursive definition for f?  Failing that, is there a recursive
definition for f(n,n)?  Here is what f looks like for small values of m & n.

    2       3       4       5       6       7       8       9
2   3
3   5      13
4   7      25      63
5   9      41     129     321
6  11      61     231     681    1683
7  13      85     377    1289    3653    8989
8  15     113     575    2241    7183   19825   48639
9  17     145     833    3649   13073   40081  108545  265729
```

Nous nous proposons de répondre à la question *"Y a-t-il une définition récursive pour $f(n,n)$ ?"* en construisant un type de la classe Ω dont le nombre d'objets de taille $n$ est exactement $f(n,n)$.

Définissons d'abord $g(m,n) = f(m+1,n+1)$. La récurrence vérifiée par $g$ est la même que celle de $f$, seules les conditions initiales sont modifiées :

$$\begin{cases} g(m,n) = g(m,n-1) + g(m-1,n-1) + g(m-1,n) & \text{pour } m,n > 0 \\ g(m,0) = g(0,n) = 1 \end{cases}$$

A la vue de ce système, nous constatons que l'entier $g(m,n)$ est le nombre de façons d'aller du point $(0,0)$ au point $(m,n)$ par un chemin comportant trois sortes de pas : les pas horizontaux (`x`), les pas verticaux (`y`) et les pas diagonaux (`d`). Par exemple, $g(2,1) = 5$, ce qui correspond aux cinq chemins

xxy xd yxx dx xyx

L'entier cherché $g(n,n)$ est par conséquent le nombre de chemins diagonaux (`CD`) pour aller de $(0,0)$ à $(n,n)$. Un tel chemin diagonal se décompose de façon unique en pas diagonaux (`Diag`) où l'on ne quitte pas la diagonale, et en "arches" qui ne touchent la diagonale qu'au début et à la fin. Il y a en outre deux sortes d'arches : les arches qui se trouvent "en-dessous" de la diagonale, commençant par `x` et finissant par `y` (`Arche_basse`), et les arches qui se trouvent "au-dessus", commençant par `y` et finissant par `x` (`Arche_haute`). Enfin, les arches se définissent à l'aide de "ponts", qui sont des séquences d'arches restant du même côté de la diagonale. Tout ceci conduit à la spécification *context-free* suivante :

**type** CD = Diag *sequence*(Arche Diag);
Arche = Arche_basse | Arche_haute;
Arche_basse = x Pont_bas y;
Pont_bas = Diag | Diag x Pont_bas y Pont_bas;
Arche_haute = y Pont_haut x;
Pont_haut = Diag | Diag y Pont_haut x Pont_haut;
Diag = *sequence*(d);

Les mots dérivés du non-terminal CD contiennent autant de fois la lettre `x` que la lettre `y` par construction ; pour savoir en quel point $(n,n)$ ils amènent, il suffit donc d'additionner le nombre d'occurrences des lettres `x` et `d`. En définissant les tailles des atomes par

x,d = *atom*(1); y = *atom*(0);

le nombre de mots de taille $n$ dérivés de CD est exactement $g(n,n)$.

L'analyse algébrique conduit à une forme explicite pour la série génératrice ordinaire des chemins diagonaux :

$$\mathrm{CD}(z) = \frac{1}{\sqrt{1-6z+z^2}}.$$

Nous pouvons énoncer le théorème suivant :

**Théorème automatique 2.** *Le nombre de chemins pour aller du point* $(0,0)$ *au point* $(n,n)$ *à l'aide de pas horizontaux, verticaux ou diagonaux est*

$$g(n,n) = [z^n]\frac{1}{\sqrt{1-6z+z^2}} = C\frac{A^n}{\sqrt{n}} + O(\frac{A^n}{n^{3/2}}),$$

*où* $A = 3+2\sqrt{2} \simeq 5.828$ *et* $C = \sqrt{A/\pi}/2^{5/4} \simeq 0.573$.

(la partie asymptotique du théorème est due au programme `equivalent` de B. Salvy). Nous sommes à présent en mesure de répondre à la question originale :

$$f(n,n) = [z^n]\frac{z}{\sqrt{1-6z+z^2}} = C\frac{A^{n-1}}{\sqrt{n}} + O(\frac{A^n}{n^{3/2}}). \tag{2.9}$$

En effet, $f(n,n) = g(n-1,n-1) = [z^{n-1}]\mathrm{CD}(z) = [z^n]z\mathrm{CD}(z)$. Les premiers termes du développement de $z\mathrm{CD}(z)$ sont

```
> taylor(z/sqrt(1-6*z+z^2),z,10);
          2       3       4       5        6        7         8          9       10
   z + 3 z  + 13 z  + 63 z  + 321 z  + 1683 z  + 8989 z  + 48639 z  + 265729 z  + O(z  )
```

et nous retrouvons les valeurs indiquées dans le message original. La définition (2.9) est récursive, car elle donne pour la série génératrice $f(z) = \sum f(n,n)z^n$ l'équation $f(z)^2(1-6z+z^2)-z^2=0$, qui conduit à une récurrence (non linéaire) pour les coefficients $f(n,n)$. Comme il est montré plus loin (section B.4), les coefficients $f(n,n)$ vérifient également une récurrence *linéaire* à coefficients polynomiaux :

```
> comtet((1-6*z+z^2)*f^2-z^2,f,'proc',0,1);
proc(N)
local n,f;
    f[0] := 0;  f[1] := 1;
    for n from 2 to N do f[n] := -(-9+6*n)/(1-n)*f[n-1]-(-n+2)/(1-n)*f[n-2] od
end
> "(100);
  354133039609265536846415517309219320565185505702928148184024525417873569343
```

Le programme MAPLE ci-dessus, généré par la procédure `comtet`, détermine $f(n,n)$ de manière efficace ($f(100,100)$ a été ainsi obtenu en moins d'une seconde sur un Sun 3/60, alors qu'il faut près d'une minute par développement de Taylor). □

Avant d'analyser de véritables programmes comprenant des procédures, nous étudions une extension possible de la classe Ω, dans laquelle la propriété de traduction automatique en séries génératrices est conservée.

### 2.1.2 Restrictions sur la longueur

Pour les multi-constructeurs (séquence, ensemble, multi-ensemble, cycle), on a la possibilité de restreindre la longueur, c'est-à-dire le nombre de composantes directes. Par exemple, afin que le non-terminal $A$ dérive des cycles formés de 3 objets dérivés par $B$, nous écrirons en Adl

**type** A = *cycle*(B, *card*=3);

Ainsi, chacun des multi-constructeurs introduits jusqu'ici induit une infinité de constructeurs "restreints", pour $k$ variant de zéro à l'infini. Ces constructeurs modifiés demeurent admissibles, et la règle associée se calcule à partir de la règle générale (sans restriction de longueur) en marquant le nombre de composantes à l'aide d'une variable auxiliaire $u$.

**Lemme 7.** *Soit* $\Phi$ *un multi-constructeur (cf définition 2) admissible, dont la règle associée est de la forme*

$$\frac{\textbf{type } \mathcal{A} = \Phi(\mathcal{B});}{A(z) = \Psi(B(z), B(z^2), \ldots, B(z^j), \ldots)}. \tag{2.10}$$

*Supposons que* $\Phi$ *vérifie de plus la condition de* transport *suivante :*

*pour toute bijection* $\varphi : \mathcal{R} \to \mathcal{T}$,

$$t_1 = \varphi(r_1), \ldots, t_k = \varphi(r_k) \quad \Longrightarrow \quad \Phi(t_1, \ldots, t_k) = \Phi(r_1, \ldots, r_k)_{r_1 \leftarrow t_1, \ldots, r_k \leftarrow t_k}$$

*où* $\Phi(r_1, \ldots, r_k)_{r_1 \leftarrow t_1, \ldots, r_k \leftarrow t_k}$ *représente l'objet* $\Phi(r_1, \ldots, r_k)$ *dans lequel on a remplacé* $r_1$ *par* $t_1$, ..., $r_k$ *par* $t_k$.

*Alors la règle pour* $k$ *composantes est*

$$\frac{\textbf{type } \mathcal{A} = \Phi(\mathcal{B}, card = k);}{A(z) = [u^k]\Psi(uB(z), u^2B(z^2), \ldots, u^jB(z^j), \ldots)}.$$

**Démonstration :** Si $\Phi$ vérifie (2.10), alors pour tout ensemble bien fondé $\mathcal{V}$, la série génératrice associée à $\Phi(\mathcal{V} \times \mathcal{B})$ est

$$\Psi(V(z) \cdot B(z), V(z^2) \cdot B(z^2), \ldots).$$

Traduisons les atomes des objets dérivés de $V$ par la variable $u$ au lieu de la variable $z$ : nous obtenons ainsi une série à deux variables,

$$\Psi(V(u) \cdot B(z), V(u^2) \cdot B(z^2), \ldots), \tag{2.11}$$

dont le coefficient de $z^n u^k$ est le nombre d'objets de $\Phi(\mathcal{V} \times \mathcal{B})$ dont la somme des tailles des objets de $\mathcal{B}$ est $n$, et la somme des tailles des objets de $\mathcal{V}$ est $k$.

Fixons maintenant $\mathcal{V} = \{v\}$ où $v$ est un atome de taille 1, de série génératrice $V(z) = z$. La série (2.11) s'écrit à présent

$$A'(z, u) = \Psi(uB(z), u^2 B(z^2), \ldots),$$

et son coefficient de $z^n u^k$ est le nombre d'objets de $\Phi(\{v\} \times \mathcal{B})$ dont la somme des tailles des objets dérivés de $\mathcal{B}$ est $n$, et le nombre d'occurrences de l'atome $v$ est $k$ (puisque $|v| = 1$). Or par construction, le nombre d'occurrences de $v$ égale le nombre d'objets de $\mathcal{B}$. La série génératrice associée à l'ensemble des objets de $\Phi(\{v\} \times \mathcal{B})$ ayant $k$ composantes est par suite $[u^k]A'(z, u)$.

Pour conclure, il suffit de noter que $\mathcal{B}$ et $\{v\} \times \mathcal{B}$ sont en bijection évidente : d'après la condition de transport, cette bijection induit une bijection entre $\Phi(\mathcal{B})$ et $\Phi(\{v\} \times \mathcal{B})$, qui de plus conserve la taille et le nombre de composantes. ■

Pour les multi-constructeurs séquence, ensemble, multi-ensemble et cycle, la condition de transport est vérifiée. Nous pouvons donc utiliser le lemme pour déterminer les règles avec restriction de la longueur.

CONSTRUCTEUR SÉQUENCE : l'opérateur général est $1/(1 - B(z))$, donc l'opérateur restreint à la longueur $k$ est

$$[u^k]\frac{1}{1 - uB(z)} = B^k(z).$$

CONSTRUCTEUR ENSEMBLE : l'opérateur restreint à la longueur $k$ est

$$\Psi_{\mathcal{S},k}(B, z) = [u^k] \exp(u\frac{B(z)}{1} - u^2\frac{B(z^2)}{2} + u^3\frac{B(z^3)}{3} - \cdots).$$

Or dans l'exponentielle, seuls les termes de degré en $u$ inférieur ou égal à $k$ contribuent au coefficient de $u^k$, et par conséquent

$$\Psi_{\mathcal{S},k}(B, z) = [u^k] \exp(u\frac{B(z)}{1} - u^2\frac{B(z^2)}{2} + \cdots + (-1)^{k+1}u^k\frac{B(z^k)}{k}).$$

Pour les premières valeurs de $k$, nous obtenons :

$$\begin{aligned}
\Psi_{\mathcal{S},0}(f, z) &= 1 \\
\Psi_{\mathcal{S},1}(f, z) &= f(z) \\
\Psi_{\mathcal{S},2}(f, z) &= \frac{f^2(z)}{2} - \frac{f(z^2)}{2}
\end{aligned}$$

$$\begin{aligned}
\Psi_{\mathcal{S},3}(f,z) &= \frac{f(z^3)}{3}-\frac{f(z)f(z^2)}{2}+\frac{f(z)^3}{6}\\
\Psi_{\mathcal{S},4}(f,z) &= \frac{f(z)f(z^3)}{3}-\frac{f(z^4)}{4}+\frac{f(z^2)^2}{8}-\frac{f(z^2)f(z)^2}{4}+\frac{f(z)^4}{24}\\
\Psi_{\mathcal{S},5}(f,z) &= \frac{f(z^5)}{5}-\frac{f(z)f(z^4)}{4}-\frac{f(z^3)f(z^2)}{6}+\frac{f(z^3)f(z)^2}{6}+\frac{f(z)f(z^2)^2}{8}-\frac{f(z^2)f(z)^3}{12}+\frac{f(z)^5}{120}.
\end{aligned}$$

CONSTRUCTEUR MULTI-ENSEMBLE : le même raisonnement que pour le constructeur *ensemble* donne

$$\Psi_{\mathcal{M},k}(B,z)=[u^k]\exp(u\frac{B(z)}{1}+u^2\frac{B(z^2)}{2}+\cdots+u^k\frac{B(z^k)}{k}). \tag{2.12}$$

Le calcul montre que $\Psi_{\mathcal{M},k}$ s'obtient à partir de $\Psi_{\mathcal{S},k}$ en remplaçant simplement les signes $-$ par des signes $+$ : ceci est dû aux formules

$$\begin{aligned}
\Psi_{\mathcal{S},k} &= [u^k]\Upsilon(B(z),-B(z^2),B(z^3),-B(z^4),\ldots)\\
\Psi_{\mathcal{M},k} &= [u^k]\Upsilon(B(z),B(z^2),B(z^3),B(z^4),\ldots)
\end{aligned}$$

avec $\Upsilon(x_1,x_2,x_3,\ldots)=\exp(ux_1+u^2x_2/2+u^3x_3/3+\cdots)$. Or le coefficient de $u^k$ dans $\Upsilon(v_1,v_2,v_3,\ldots)$ est aussi $([u^k]\Upsilon)(v_1,v_2,v_3,\ldots)$, donc $\Psi_{\mathcal{S},k}$ et $\Psi_{\mathcal{M},k}$ s'obtiennent par substitution de $x_1$ par $B(z)$, de $x_2$ par $\pm B(z^2)$, ... dans $[u^k]\Upsilon$.

REMARQUE : Cette fonction $\Upsilon$ à une infinité de variables permet donc de dénombrer les ensembles et les multi-ensembles, avec ou sans restriction de longueur. Écrivons le développement de $\Upsilon$ en puissances croissantes de $u$ :

$$\Upsilon(x_1,x_2,x_3,\ldots)=1+x_1u+(x_1^2+x_2)\frac{u^2}{2}+(x_1^3+3x_1x_2+2x_3)\frac{u^3}{6}+O(u^4).$$

Dans le facteur de $u^k$, chaque monôme $cx_1^{\alpha_1}x_2^{\alpha_2}\ldots$ vérifie $\alpha_1+2\alpha_2+\cdots=k$. Cette remarque montre que la variable $u$ est en fait superflue puisque l'information qu'elle porte (la longueur $k$) est déjà contenue dans les variables $x_i$. Lorsque l'on remplace $u$ par 1, on obtient ce qui s'appelle dans la théorie de Pólya [Pól37, Joy81, BLL90] la fonction *indicatrice de cycle* (*Zyklenzeiger* en allemand) de la construction *multi-ensemble*. Nous noterons $Z_\Phi$ la fonction indicatrice de cycle d'un constructeur $\Phi$.

$$\begin{aligned}
Z_{\mathcal{M}}(x_1,x_2,\ldots) &= 1+x_1+\frac{1}{2}(x_1^2+x_2)+\frac{1}{6}(x_1^3+3x_1x_2+2x_3)\\
&+ \frac{1}{24}(x_1^4+6x_1^2x_2+3x_2^2+6x_4+8x_1x_3)\\
&+ \frac{1}{120}(x_1^5+10x_1^3x_2+15x_1x_2^2+20x_1^2x_3+20x_2x_3+30x_1x_4+24x_5)+\cdots
\end{aligned}$$

La fonction indicatrice de cycle permet de retrouver les règles avec restriction de longueur : un monôme $cx_1^{\alpha_1}x_2^{\alpha_2}\ldots$ correspond à des objets comportant $\alpha_1$ éléments indépendants, $\alpha_2$ paires de composantes identiques, ... et se traduit en $cf^{\alpha_1}(z)f^{\alpha_2}(z^2)\ldots$ Il suffit donc de prendre tous les monômes tels que $\alpha_1+2\alpha_2+\cdots=k$ pour obtenir la restriction à $k$ composantes. Dans l'expression de $Z_{\mathcal{M}}$, nous avons justement regroupé ces termes entre parenthèses. Par exemple, pour $k=2$, le terme $(x_1^2+x_2)/2$ se traduit en $(f^2(z)+f(z^2))/2$, et l'on retrouve $\Psi_{\mathcal{M},2}(f,z)$.

CONSTRUCTEUR CYCLE ORIENTÉ : le lemme 7 donne

$$\Psi_{\mathcal{C},k}(B,z) = [u^k] \sum_{j\geq 1} \frac{\phi(j)}{j} \log \frac{1}{1-u^j B(z^j)}.$$

Le terme de plus bas degré en $u$ de $\log \frac{1}{1-u^j B(z^j)}$ étant $u^j B(z^j)$, il suffit de se restreindre à $1 \leq j \leq k$. De plus, si $j$ n'est pas un diviseur de $k$, la contribution est nulle. Un calcul simple aboutit à la règle

$$\frac{\textbf{type } A = cycle(B, card = k);}{A(z) = \frac{1}{k}\sum_{j|k} \phi(j) B^{k/j}(z^j)}.$$

Les premiers termes de la série indicatrice de cycle du constructeur cycle orienté sont par suite

$$Z_{\mathcal{C}}(x_1, x_2, \ldots) = x_1 + \frac{1}{2}(x_1^2 + x_2) + \frac{1}{3}(x_1^3 + 2x_3) + \frac{1}{4}(x_1^4 + x_2^2 + 2x_4) + \frac{1}{5}(x_1^5 + 4x_5) + \cdots$$

Chaque monôme ne contient qu'un seul facteur $x_i^{\alpha_i}$, et la somme des coefficients des monômes correspondant à une même longueur $k$ égale $k$ (car $\sum_{j|k} \phi(j) = k$).

## 2.2 Analyse des programmes

Dans cette section, nous montrons que parallèlement aux règles associées aux constructeurs de la classe $\Omega$, il existe des règles de traduction pour les schémas de programmation sur ces constructeurs (descente, sélection, itération). Ces règles conduisent à des équations pour les *descripteurs de complexité*, qui sont des séries génératrices comptant le coût total des procédures. Ainsi, non seulement tout programme de la classe II se traduit automatiquement en équations de séries génératrices (de dénombrement et de coût), mais nous pouvons caractériser précisément la classe des équations engendrées (théorème 4).

Étant données une procédure $P : A$ et une donnée $a \in \mathcal{A}$, nous noterons $\tau P\{a\}$ le coût de l'évaluation de $P(a)$. Ce coût est par définition la somme des coûts des opérations élémentaires provoquées par l'évaluation de $P(a)$. Ces coûts élémentaires sont spécifiés par la déclaration **measure** du programme Adl, et constituent la *mesure de complexité* du problème (voir chapitre précédent, section 1.6.4).

**Définition 12.** *Le* descripteur de complexité *ordinaire associé à une procédure $P$ et à un ensemble $\mathcal{A}$ est la série génératrice*

$$\tau P^{\mathcal{A}}(z) = \sum_{a\in\mathcal{A}} \tau P\{a\} z^{|a|} = \sum_{n=0}^{\infty} \tau P_n^{\mathcal{A}} z^n.$$

Ainsi, $\tau P_n^{\mathcal{A}}$ représente le coût de l'évaluation de $P$, cumulé sur tous les objets de taille $n$ de $\mathcal{A}$. Lorsqu'il n'y a pas de confusion possible sur l'ensemble $\mathcal{A}$, nous noterons simplement $\tau P$ et $\tau P_n$ au lieu de $\tau P^{\mathcal{A}}$ et $\tau P_n^{\mathcal{A}}$.

**Définition 13.** *Un schéma de programmation* **procedure** $P(a : A)$; **begin** ... **end** *est dit* admissible *si la suite $(\tau P_n^{\mathcal{A}})$ ne dépend que des suites $(\tau Q_n)$ associées aux procédures $Q$ se trouvant dans le corps de $P$, de la suite $(A_n)$ associée à $\mathcal{A}$, et des suites $(C_n)$ associées aux composantes des objets de $\mathcal{A}$.*

REMARQUE : Dans la pratique, l'ensemble $\mathcal{A}$ est fabriqué à l'aide de constructeurs qui sont aussi admissibles, par exemple $\mathcal{A} = \mathit{sequence}(\mathcal{C})$, et la suite $(A_n)$ s'exprime donc à l'aide des suites $(C_n)$ associées aux composantes des objets de $\mathcal{A}$.

Le résultat principal de cette section, que nous annonçons dès à présent, est le suivant :

**Théorème 4** *Le schéma élémentaire, les schémas d'exécution séquentielle, de sélection dans une union ou dans un produit cartésien, de sélection et d'itération dans une séquence, un ensemble, un cycle orienté ou un multi-ensemble sont admissibles.*

### 2.2.1 Les schémas généraux

Comme les constructeurs, les schémas de programmation que nous avons définis au chapitre précédent induisent des opérateurs sur les descripteurs de complexité. Pour plus de lisibilité, nous utilisons la syntaxe Adl dans les règles qui suivent. La forme

$$\frac{< \mathit{spécification}\ \Sigma > \qquad < \mathit{procédure}\ P >}{< \mathit{équation}\ \Theta >}$$

signifie que la spécification $\Sigma$ et la procédure $P$ conduisent à l'équation $\Theta$.

**Règle 8.** *(Instruction élémentaire)*

$$\frac{\textbf{procedure}\ P(a:A);\ \textbf{begin}\ \text{count}\ \textbf{end;} \qquad \textbf{measure}\ \text{count} : \mu;}{\tau P(z) = \mu \cdot A(z)}$$

**Démonstration :** $\tau P(z) = \sum_{a \in \mathcal{A}} \mu\, z^{|a|} = \mu \cdot A(z)$. ∎

**Règle 9.** *(Exécution séquentielle)*

$$\frac{\textbf{procedure}\ P(a:A);\ \textbf{begin}\ Q(a); R(a)\ \textbf{end;}}{\tau P(z) = \tau Q(z) + \tau R(z)}$$

**Démonstration :** $\tau P(z) = \sum_{a \in \mathcal{A}} (\tau Q\{a\} + \tau R\{a\}) z^{|a|} = \tau Q(z) + \tau R(z)$. ∎

Pour une plus grande concision, nous omettons dans les règles qui suivent les mots-clés **begin** et **end** délimitant le corps de la procédure.

**Règle 10.** *(Sélection dans une union)*

$$\frac{\textbf{type}\ A = B \mid C; \qquad \textbf{procedure}\ P(a:A);\ \textbf{casetype}\ a\ \textbf{of}\ B : Q(a);\ C : R(a)\ \textbf{end;}}{\tau P^{\mathcal{A}}(z) = \tau Q^{\mathcal{B}}(z) + \tau R^{\mathcal{C}}(z)}$$

**Démonstration :** $\displaystyle\sum_{a \in \mathcal{B} \cup \mathcal{C}} \tau P\{a\} z^{|a|} = \sum_{a \in \mathcal{B}} \tau Q\{a\} z^{|a|} + \sum_{a \in \mathcal{C}} \tau R\{a\} z^{|a|} = \tau Q^{\mathcal{B}}(z) + \tau R^{\mathcal{C}}(z)$. ∎

**Règle 11.** *(Sélection dans un produit cartésien)*

$$\frac{\textbf{type}\ A = \mathit{product}(B, C); \qquad \textbf{procedure}\ P(a:A);\ \textbf{case}\ a\ \textbf{of}\ (b,c) : Q(b)\ \textbf{end;}}{\tau P^{\mathcal{A}}(z) = \tau Q^{\mathcal{B}}(z) C(z)}$$

**Démonstration :**

$$\tau P^{\mathcal{A}}(z) = \sum_{b\in\mathcal{B},c\in\mathcal{C}} \tau P\{(b,c)\} z^{|(b,c)|} \tag{2.13}$$

$$= \sum_{b\in\mathcal{B},c\in\mathcal{C}} \tau Q\{b\} z^{|b|+|c|} \tag{2.14}$$

$$= \left(\sum_{b\in\mathcal{B}} \tau Q\{b\} z^{|b|}\right) \cdot \left(\sum_{c\in\mathcal{C}} z^{|c|}\right) = \tau Q^{\mathcal{B}}(z) C(z). \tag{2.15}$$

Le point crucial de cette démonstration est le passage de (2.13) à (2.14), qui est possible par la conjonction de deux faits : le coût de l'évaluation de $P$ sur $(b,c)$ ne dépend que de $b$ : $\tau P\{(b,c)\} = \tau Q\{b\}$, et la taille d'un produit cartésien est la somme des tailles des composantes : $|(b,c)| = |b|+|c|$. Ainsi, les sommations se séparent dans (2.14), pour donner le résultat final (2.15). ■

Ces premières règles suffisent pour effectuer l'analyse algébrique du programme de dérivation formelle, et prouver ainsi le lemme automatique 1 du chapitre 1 (page 14).

**Démonstration :** (Lemme automatique 1) Notre mesure de complexité était une unité pour chaque appel de la fonction `write`, c'est-à-dire le nombre de mots (`zero`, `un`, `x`, ...) écrits par la procédure de dérivation. Par exemple, pour l'expression `(mult,f,g)`, la procédure `diff` évalue la séquence d'instructions

**begin** write(plus); write(mult); diff(f); copy(g); write(mult); copy(f); diff(g) **end**

Le coût est par conséquent 3 (une fois `plus` et deux fois `mult`) plus le coût des appels `diff(f)`, `copy(g)`, `copy(f)` et `diff(g)`. Par les règles 8, 9 et 11, le descripteur de complexité associé à cette partie de la procédure `diff` est

$$3zE^2(z) + 2zE(z)\tau D(z) + 2zE(z)\tau C(z)$$

où $\tau D$ et $\tau C$ sont les descripteurs de complexité associés respectivement aux procédures `diff` et `copy`, et $E$ la série génératrice du type `expression`. L'analyse se fait de même pour les autres sortes d'expressions (par souci de lisibilité, nous avons omis les mots-clés **begin** et **end** et nous avons abrégé les noms des procédures) :

| | $\tau D(z) =$ |
|---|---|
| `procedure D (e : expression);` | |
| `case e of` | |
| `zero : write(zero);` | $z$ |
| `un : write(zero);` | $+z$ |
| `x : write(un);` | $+z$ |
| `(plus,f,g) : write(plus); D(f); D(g);` | $+zE^2(z) + 2zE(z)\tau D(z)$ |
| `(mult,f,g) : write(plus); write(mult); D(f); C(g);` | $+2zE(z)^2 + zE(z)\tau D(z) + zE(z)\tau C(z)$ |
| `write(mult); C(f); D(g);` | $+zE(z)^2 + zE(z)\tau C(z) + zE(z)\tau D(z)$ |
| `(expo,f) : write(mult); write(expo); C(f); D(f);` | $+2zE(z) + z\tau C(z) + z\tau D(z)$ |
| `end;` | |

et conduit à une équation linéaire en $\tau D$ :

$$\tau D(z) = 3z + 2zE(z) + 4zE^2(z) + z\tau C(z) + 2zE(z)\tau C(z) + z\tau D(z) + 4zE(z)\tau D(z).$$

Pour la procédure `copy`, il vient également une équation linéaire en $\tau C$ :

$$\tau C(z) = 3z + zE(z) + 2zE^2(z) + z\tau C(z) + 4zE(z)\tau C(z).$$

Ces deux équations, combinées avec l'égalité (2.7) donnant $E(z)$, permettent de calculer une forme explicite pour $\tau D$ et $\tau C$ :

$$\tau D(z) = \frac{(1-2z-12z^2)\sqrt{1-2z-23z^2} - 1 + 3z + 34z^2}{4z(1-2z-23z^2)},$$

$$\tau C(z) = \frac{1-z-\sqrt{1-2z-23z^2}}{4z\sqrt{1-2z-23z^2}}.$$ ■

L'analyse asymptotique de ces deux fonctions complète la preuve du théorème automatique 1, qui donne le comportement du coût moyen de la procédure `diff` lorsque la taille $n$ des données tend vers l'infini :

$$\overline{\tau\mathrm{diff}}_n = \frac{(1+\sqrt{6})\sqrt{138\pi(12-\sqrt{6})}}{276}\, n^{3/2} + \frac{11(12-\sqrt{6})}{276}\, n + O(n^{1/2}).$$

Nous présentons maintenant les règles concernant les schémas de sélection et d'itération dans une séquence. Nous avons vu au chapitre précédent que le constructeur "séquence" peut se définir à l'aide des constructeurs "union" et "produit cartésien". Nous utiliserons cette correspondance pour prouver les deux règles qui suivent, qui sont ainsi en quelque sorte dérivées des règles précédentes.

**Règle 12.** *(Sélection dans une séquence)*

$$\frac{\textbf{type } A = \mathit{sequence}(B); \qquad \textbf{procedure } P(a:A);\ \textbf{forone } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \tau Q(z)/(1-B(z))}$$

**Démonstration :** Utilisons les règles ci-dessus pour démontrer cette règle. Le programme

$$\begin{array}{l}\textbf{type } A = \mathit{sequence}(B);\\ \textbf{procedure } P(a:A);\ \textbf{forone } b \textbf{ in } a \textbf{ do } Q(b);\end{array}$$

est équivalent au programme

$$(\pi_4)\begin{cases}\textbf{type} \quad A = \epsilon \mid C;\\ \qquad\quad C = \mathit{product}(B, A);\\ \textbf{procedure } P(a:A);\ \textbf{casetype } a \textbf{ of } C : R(a) \textbf{ end;}\\ \textbf{procedure } R(c:C);\ \textbf{case } c \textbf{ of } (b,a) : Q(b) \textbf{ end;}\end{cases}$$

en ce sens que tous deux conduisent à la même série génératrice pour le type $A$ et au même descripteur de complexité pour la procédure $P$. En effet, dans le second programme, $\mathcal{A}$ est équivalent à une séquence d'éléments de $\mathcal{B}$ (voir la remarque page 23) et la procédure $P$ appelle $Q$ sur le premier élément (lorsque la séquence est non vide) : ceci revient au même pour le coût moyen que de choisir un élément au hasard, parce que la distribution du premier élément d'une séquence est la même que celle de n'importe quel autre. En vertu des règles ci-dessus, le programme $(\pi_4)$ se traduit en un système de quatre équations :

$$(S)\begin{cases} A(z) &=& 1 + C(z) & \text{(règle 2)}\\ C(z) &=& B(z)A(z) & \text{(règle 3)}\\ \tau P(z) &=& \tau R(z) & \text{(règle 10)}\\ \tau R(z) &=& \tau Q(z)A(z) & \text{(règle 11)}\end{cases}$$

qui donne après résolution $\tau P(z) = \tau Q(z)A(z) = \tau Q(z)/(1-B(z))$. ■

**Règle 13.** *(Itération dans une séquence)*

$$\frac{\textbf{type } A = \mathit{sequence}(B); \qquad \textbf{procedure } P(a:A);\ \textbf{forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \tau Q(z)/(1-B(z))^2}$$

**Démonstration :** La démonstration procède de la même manière que pour la sélection : remplaçons la définition de la procédure $R$ du programme $(\pi_4)$ par

$$\textbf{procedure } R(c:C);\ \textbf{case } c \textbf{ of } (b,a) : Q(b); P(a) \textbf{ end;}$$

qui appelle $Q$ sur le premier élément, puis $P$ sur le reste de la séquence. Le nouveau programme obtenu effectue bien l'itération de $Q$ sur la séquence donnée en argument à $P$. Seule l'équation de $\tau R$ change dans le système $(S)$, et devient

$$\tau R(z) = \tau Q(z)A(z) + B(z)\tau P(z) \qquad \text{(règles 9 et 11)},$$

et comme on a toujours $\tau P = \tau R$, il vient

$$\tau P(z) = \tau Q(z)\frac{A(z)}{1-B(z)} = \frac{\tau Q(z)}{(1-B(z))^2}.$$

■

**Règle 14.** *(Itération dans un ensemble)*

$$\frac{\textbf{type } A = \mathit{set}(B); \qquad \textbf{procedure } P(a:A);\ \textbf{forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \Psi_S(B)(z)\,(\tau Q(z) - \tau Q(z^2) + \tau Q(z^3) - \cdots)}$$

**Démonstration :** La construction $\prod_{b\in\mathcal{B}}(\{\} \cup \{b\} \cup \{v\,b\})$ forme toutes les parties finies de $\mathcal{B}$, où chaque élément apparaît tel quel ou bien "marqué" par la lettre $v$ (voir l'équation (2.4) pour une construction analogue). Si l'on ne garde que les parties où un seul élément est marqué, et qu'on remplace la marque $v$ par $v\tau Q\{b\}$, on obtient la règle d'itération dans un ensemble :

$$\begin{aligned}
\tau P(z) &= [v]\prod_{b\in\mathcal{B}}(1 + z^{|b|} + v\tau Q\{b\}z^{|b|}) \\
&= [v]\prod_{n=1}^{\infty}\left[(1+z^n)^{B_n} + (1+z^n)^{B_n-1}\sum_{\substack{b\in\mathcal{B}\\|b|=n}} v\tau Q\{b\}z^n\right] \\
&= [v]\prod_{n=1}^{\infty}(1+z^n)^{B_n}(1 + v\frac{\tau Q_n z^n}{1+z^n}) \\
&= \Psi_S(B)(z)\sum_{n=1}^{\infty}\frac{\tau Q_n z^n}{1+z^n} = \Psi_S(B)(z)\ (\tau Q(z) - \tau Q(z^2) + \tau Q(z^3) - \cdots).
\end{aligned}$$

Dans la première équation, chaque élément $b$ est soit absent de l'ensemble (terme 1), soit présent sans appel de $Q(b)$ (terme $z^{|b|}$), soit présent avec appel de $Q(b)$ (terme $v\tau Q\{b\}z^{|b|}$). Le coefficient de $v$ de ce produit donne donc le coût de l'itération de $Q$. A la seconde ligne, on a regroupé les facteurs correspondant aux objets $b$ de même taille $n$, et on a éliminé les termes multiples de $v^2$. ■

Puisque $A(z) = \Psi_S(B)(z)$, la règle ci-dessus s'écrit aussi $\tau P(z) = A(z)\tau Q(z) - A(z)\tau Q(z^2) + A(z)\tau Q(z^3) - \cdots$, et s'interprète alors de la façon suivante. Le terme principal est $A(z)\tau Q(z)$,

mais dans ce terme, on compte des ensembles comportant deux fois le même élément. Il faut donc retrancher $A(z)\tau Q(z^2)$ pour corriger. Mais cette correction décompte aussi des ensembles ayant trois fois le même élément, qu'il faut à nouveau rajouter, ce qui explique le terme $A(z)\tau Q(z^3)$, et ainsi de suite.

**Règle 15.** *(Sélection dans un ensemble)*

$$\frac{\textbf{type } A = set(B); \qquad \textbf{procedure } P(a:A); \textbf{ forone } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \sum_{n=1}^{\infty} \tau Q_n z^n \int_0^1 \frac{B^S(z,u)}{1+uz^n} du}$$

*où* $B^{\mathcal{S}}(z,u) = \prod_{b\in B}(1+uz^{|b|}) = \prod_{n\geq 1}(1+uz^n)^{B_n}$.

La série $\tau P$ ne dépendant que des $\tau Q_n$ et des $B_n$, ce schéma est aussi admissible.
**Démonstration :** Pour un ensemble $\{b_1,\ldots,b_k\}$, chaque élément a probabilité $1/k$ d'être choisi, et par conséquent il faut diviser le coût de tous les appels par $k$. Pour cela, nous utilisons une troisième variable $u$ comptant la longueur des ensembles, en partant de la même formule que pour la preuve de la règle d'itération :

$$\begin{aligned}
\tau P(z) &= \sum_{k=1}^{\infty} \frac{1}{k}[u^k]\left[[v]\prod_{b\in\mathcal{B}}(1+uz^{|b|}+uv\tau Q\{b\}z^{|b|})\right] \\
&= \sum_{k=1}^{\infty} \frac{1}{k}[u^k]\left[B^{\mathcal{S}}(z,u)\sum_{n=1}^{\infty}\frac{u\tau Q_n z^n}{1+uz^n}\right] \\
&= \sum_{n=1}^{\infty}\tau Q_n z^n \sum_{k=1}^{\infty}\frac{1}{k}[u^k]\frac{uB^{\mathcal{S}}(z,u)}{1+uz^n} \\
&= \sum_{n=1}^{\infty}\tau Q_n z^n \int_0^1 \frac{B^{\mathcal{S}}(z,u)}{1+uz^n}du.
\end{aligned}$$

Le passage de la première à la seconde ligne procède de la même façon que la preuve précédente. Ensuite, on intervertit les sommations et on utilise le fait que pour une série génératrice $f$, $\sum_{k=1}^{\infty} 1/k[u^k](uf(u)) = \int_0^1 f(u)du$. ∎

**Règle 16.** *(Itération dans un multi-ensemble)*

$$\frac{\textbf{type } A = multiset(B); \qquad \textbf{procedure } P(a:A); \textbf{ forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \Psi_M(B)(z)\ (\tau Q(z) + \tau Q(z^2) + \tau Q(z^3) + \cdots)}$$

**Démonstration :** Comme pour le constructeur ensemble, remplaçons le terme $z^{|b|}$ par $z^{|b|} + v\tau Q\{b\}z^{|b|}$ dans la série de dénombrement de $\mathcal{A}$ :

$$\begin{aligned}
\tau P(z) &= [v]\prod_{b\in\mathcal{B}}\frac{1}{1-(z^{|b|}+v\tau Q\{b\}z^{|b|})} \\
&= [v]\prod_{n=1}^{\infty}\prod_{|b|=n}\frac{1}{1-z^n}\frac{1}{1-v\tau Q\{b\}\frac{z^n}{1-z^n}}
\end{aligned}$$

$$= [v]\prod_{n=1}^{\infty}\frac{1}{(1-z^n)^{B_n}}\prod_{|b|=n}(1+v\tau Q\{b\}\frac{z^n}{1-z^n}) = \Psi_M(B)(z)[v]\prod_{n=1}^{\infty}(1+v\tau Q_n\frac{z^n}{1-z^n})$$

$$= \Psi_M(B)(z)\sum_{n=1}^{\infty}\tau Q_n\frac{z^n}{1-z^n} = \Psi_M(B)(z)\ (\tau Q(z)+\tau Q(z^2)+\tau Q(z^3)+\cdots).$$

De la seconde à la troisième ligne, on a utilisé le fait que $[v]\prod_b \frac{1}{1-vf(b)} = [v]\prod_b(1+vf(b))$. ■

De façon similaire à l'itération dans un ensemble, la règle ci-dessus s'écrit $\tau P(z) = A(z)\tau Q(z) + A(z)\tau Q(z^2)+A(z)\tau Q(z^3)+\cdots$ et admet une interprétation simple. Le terme $A(z)\tau Q(z)$ correspond à l'itération de $Q$ sur chaque élément *différent* du multi-ensemble, le terme $A(z)\tau Q(z^2)$ compte un appel supplémentaire pour les paires d'éléments identiques, $A(z)\tau Q(z^3)$ pour les triplets, et ainsi de suite.

**Règle 17.** *(Sélection dans un multi-ensemble)*

$$\frac{\textbf{type } A = \mathit{multiset}(B); \qquad \textbf{procedure } P(a:A);\ \textbf{forone } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \sum_{n=1}^{\infty}\tau Q_n z^n \int_0^1 \frac{B^{\mathcal{M}}(z,u)}{1-uz^n}du}$$

*où* $B^{\mathcal{M}}(z,u) = \prod_{b\in B}\frac{1}{1-uz^{|b|}} = \prod_{n\geq 1}(1-uz^n)^{-B_n}$.

**Démonstration :** Nous introduisons la variable $u$ pour compter le nombre d'éléments :

$$\begin{aligned}
\tau P(z) &= \sum_{k=1}^{\infty}\frac{1}{k}[u^k]\left[[v]\prod_{b\in\mathcal{B}}\frac{1}{1-uz^{|b|}(1+v\tau Q\{b\})}\right] \\
&= \sum_{k=1}^{\infty}\frac{1}{k}[u^k]\left[B^{\mathcal{M}}(z,u)\sum_{n=1}^{\infty}\tau Q_n\frac{uz^n}{1-uz^n}\right] \\
&= \sum_{n=1}^{\infty}\tau Q_n z^n\sum_{k=1}^{\infty}\frac{1}{k}[u^k]\frac{uB^{\mathcal{M}}(z,u)}{1-uz^n} \\
&= \sum_{n=1}^{\infty}\tau Q_n z^n\int_0^1\frac{B^{\mathcal{M}}(z,u)}{1-uz^n}du
\end{aligned}$$

Nous renvoyons le lecteur désirant plus de détails à la preuve de la règle de sélection dans un ensemble. ■

REMARQUE : L'opérateur associé au schéma de sélection se simplifie lorsqu'il y a restriction sur la longueur du multi-ensemble. En effet, si la longueur vaut $k \geq 1$, alors la seconde ligne de la preuve donne

$$\tau P_{\mathrm{card}=k}(z) = \frac{1}{k}[u^k]B^{\mathcal{M}}(z,u)\sum_{n=1}^{\infty}\tau Q_n\frac{uz^n}{1-uz^n}.$$

Or $B^{\mathcal{M}}(z,u)$ se décompose en fonctions des $\Psi_{\mathcal{M},j}(B,z)$, d'où

$$\tau P_{\mathrm{card}=k}(z) = \frac{1}{k}[u^k]\sum_{j=0}^{\infty}u^j\Psi_{\mathcal{M},j}(B,z)\sum_{n=1}^{\infty}\tau Q_n\frac{uz^n}{1-uz^n}.$$

L'intervalle de sommation pour $j$ se réduit à $[0, k-1]$, et à $j$ fixé, seul le coefficient de $u^{k-j}$ contribue dans la sommation sur $n$ :

$$\tau P_{\mathrm{card}=k}(z) = \frac{1}{k}\sum_{j=0}^{k-1} \Psi_{\mathcal{M},j}(B,z)\tau Q(z^{k-j}). \tag{2.16}$$

D'ailleurs, la construction $multiset(B)$ équivalant à l'union des constructions $multiset(B, \mathtt{card} = k)$, pour $k$ variant de 0 à l'infini, l'égalité (2.16) fournit une forme alternative pour la règle de sélection dans un multi-ensemble :

$$\tau P(z) = \sum_{k=1}^{\infty} \tau Q(z^k)\left[\frac{\Psi_{\mathcal{M},0}(B,z)}{k} + \frac{\Psi_{\mathcal{M},1}(B,z)}{k+1} + \frac{\Psi_{\mathcal{M},2}(B,z)}{k+2} + \cdots\right] \tag{2.17}$$

$$= \sum_{k=1}^{\infty} \tau Q(z^k)\left[\frac{1}{k} + \frac{B(z)}{k+1} + \frac{B(z)^2 + B(z^2)}{2(k+2)} + \cdots\right]. \tag{2.18}$$

**Règle 18.** *(Sélection dans un cycle)*

$$\frac{\textbf{type } C = cycle(A); \qquad \textbf{procedure } P(c:C); \textbf{ forone } a \textbf{ in } C \textbf{ do } Q(a);}{\tau P(z) = \sum_{k\geq 1} \frac{\phi(k)}{k} L^*(A(z^k))\tau Q(z^k)}$$

*où* $L^*(f) = \frac{1}{f}\log\frac{1}{1-f}$.

**Démonstration :** La compréhension de cette preuve sera facilitée par la lecture préalable de la section A.2 (annexe A).

En utilisant la variable $z$ pour compter la taille d'un cycle et $u$ pour compter sa longueur, le descripteur de complexité à deux variables du schéma de sélection est :

$$\tau P^{\mathcal{C}}(z,u) = \sum_{c\in\mathcal{C}} \tau P\{c\} z^{|c|} u^{l(c)}$$

et la série recherchée est $\tau P(z) = \tau P^{\mathcal{C}}(z,1)$. Soit $\mathcal{PC}$ l'ensemble des cycles primitifs de $\mathcal{C}$ (voir la définition 25 page 161). Chaque cycle s'écrit de façon unique $c^k$, où $c$ est sa racine primitive (lemme 17 page 161) ; un tirage aléatoire uniforme sur $c^k$ a la même distribution de probabilité qu'un tirage sur $c$, donc $\tau P\{c^k\} = \tau P\{c\}$ et

$$\tau P^{\mathcal{C}}(z,u) = \sum_{c\in\mathcal{PC}}\sum_{k=1}^{\infty} \tau P\{c\} z^{k|c|} u^{kl(c)} = \sum_{k=1}^{\infty} \tau P^{\mathcal{PC}}(z^k, u^k). \tag{2.19}$$

Or chaque cycle primitif de longueur $k$ représente $k$ *mots* primitifs, et le schéma de sélection a même coût sur un cycle que sur l'une des séquences associées :

$$\tau P^{\mathcal{PC}}(z,u) = \int_0^u \tau P^{\mathcal{PS}}(z,t)\frac{dt}{t} \tag{2.20}$$

où $\mathcal{PS}$ est l'ensemble des mots primitifs sur $\mathcal{A}$. Il ne nous reste plus qu'à exprimer $\tau P^{\mathcal{PS}}$ en fonction de $\tau P^{\mathcal{S}}$, $\mathcal{S}$ désignant l'ensemble des séquences sur $\mathcal{A}$ :

$$\tau P^{\mathcal{S}}(z,u) = \sum_{k=1}^{\infty} \tau P^{\mathcal{PS}}(z^k, u^k). \tag{2.21}$$

D'autre part, $\tau P^{\mathcal{S}}(z,u)$ est donné par la règle 12 :

$$\tau P^{\mathcal{S}}(z,u) = \frac{u\tau Q(z)}{1-uA(z)}. \tag{2.22}$$

La formule d'inversion de Möbius (lemme 16 page 159) appliquée à l'équation (2.21), combinée avec (2.22), donne

$$\tau P^{\mathcal{PS}}(z,u) = \sum_{k\geq 1}\mu(k)\tau P^{\mathcal{S}}(z^k,u^k) = \sum_{k\geq 1}\mu(k)\frac{u^k\tau Q(z^k)}{1-u^kA(z^k)}.$$

Puis nous substituons dans (2.20) :

$$\tau P^{\mathcal{PC}}(z,u) = \sum_{k\geq 1}\frac{\mu(k)\tau Q(z^k)}{k}\frac{1}{A(z^k)}\log\frac{1}{1-u^kA(z^k)}.$$

En remplaçant cette formule dans l'équation (2.19), et à l'aide de l'identité $\sum_{p|k}\mu(p)/p = \phi(k)/k$ (équation (A.3) page 160), nous obtenons enfin :

$$\tau P^{\mathcal{C}}(z,u) = \sum_{k\geq 1}\frac{\phi(k)}{k}\frac{1}{A(z^k)}\log\frac{1}{1-u^kA(z^k)}\tau Q(z^k)$$

ce qui, après substitution de $u$ par 1, donne le résultat annoncé. ■

NOTE : Le terme $A(z^k)$ du dénominateur est compensé par les termes du développement du logarithme qui sont tous multiples de $A(z^k)$.

**Règle 19.** *(Itération dans un cycle)*

$$\frac{\textbf{type } C = \mathit{cycle}(A); \qquad \textbf{procedure } P(c:C);\ \textbf{forall } a \textbf{ in } c \textbf{ do } Q(a);}{\tau P(z) = \sum_{k\geq 1}\phi(k)\frac{1}{1-A(z^k)}\tau Q(z^k)}$$

**Démonstration :** Pour le schéma d'itération, le coût sur $c^k$ vaut $k$ fois le coût sur $c$ : $\tau P\{c^k\} = k\tau P\{c\}$ ; l'équation (2.19) devient par conséquent

$$\tau P^{\mathcal{C}}(z,u) = \sum_{c\in\mathcal{PC}}\sum_{k=1}^{\infty} k\tau P\{c\}z^{k|c|}u^{kl(c)} = \sum_{k=1}^{\infty} k\tau P^{\mathcal{PC}}(z^k,u^k). \tag{2.23}$$

L'équation (2.20) est inchangée, et l'équation (2.21) devient

$$\tau P^{\mathcal{S}}(z,u) = \sum_{k=1}^{\infty} k\tau P^{\mathcal{PS}}(z^k,u^k), \tag{2.24}$$

tandis que le descripteur de complexité du schéma d'itération sur $\mathcal{S}$ devient par la règle 13

$$\tau P^{\mathcal{S}}(z,u) = \frac{u\tau Q(z)}{(1-uA(z))^2}.$$

Le corollaire de la formule d'inversion de Möbius (page 160) appliqué sur l'équation (2.24) donne

$$\tau P^{\mathcal{PS}}(z,u) = \sum_{k\geq 1} k\mu(k)\frac{u^k\tau Q(z^k)}{(1-u^kA(z^k))^2},$$

| schéma | | syntaxe Adl | traduction |
|---|---|---|---|
| exécution séquentielle | | `Q(a); R(a);` | $\tau Q(z) + \tau R(z)$ |
| union | sélection | **casetype** $a$ **of** $B : Q(a)$ **end;** | $\tau Q^{\mathcal{B}}(z)$ |
| produit cartésien | sélection | **case** $a$ **of** $(b, c) : Q(b)$ **end;** | $\tau Q^{\mathcal{B}}(z) C(z)$ |
| séquence | sélection | **forone** $b$ **in** $a$ **do** $Q(b)$; | $A(z)\, \tau Q(z)$ |
| | itération | **forall** $b$ **in** $a$ **do** $Q(b)$; | $A(z)^2\, \tau Q(z)$ |
| ensemble | sélection | **forone** $b$ **in** $a$ **do** $Q(b)$; | $\sum_{n=1}^{\infty} \tau Q_n z^n \int_0^1 \frac{B^{\mathcal{S}}(z,u)}{1+uz^n} du$ |
| | itération | **forall** $b$ **in** $a$ **do** $Q(b)$; | $A(z)\,(\tau Q(z) - \tau Q(z^2) + \cdots)$ |
| multi-ensemble | sélection | **forone** $b$ **in** $a$ **do** $Q(b)$; | $\sum_{n=1}^{\infty} \tau Q_n z^n \int_0^1 \frac{B^{\mathcal{M}}(z,u)}{1-uz^n} du$ |
| | itération | **forall** $b$ **in** $a$ **do** $Q(b)$; | $A(z)\,(\tau Q(z) + \tau Q(z^2) + \cdots)$ |
| cycle orienté | sélection | **forone** $b$ **in** $a$ **do** $Q(b)$; | $\sum_{k\geq 1} \frac{\phi(k)}{k} \frac{1}{B(z^k)} \log \frac{1}{1-B(z^k)} \tau Q(z^k)$ |
| | itération | **forall** $b$ **in** $a$ **do** $Q(b)$; | $\sum_{k\geq 1} \phi(k) \frac{1}{1-B(z^k)} \tau Q(z^k)$ |

Figure 2.1 : La traduction des schémas de la classe Π

d'où par intégration

$$\tau P^{\mathcal{PC}}(z,u) = \sum_{k\geq 1} \mu(k) \frac{u^k \tau Q(z^k)}{1 - u^k A(z^k)}.$$

En remplaçant cette formule dans (2.23), et en utilisant l'identité $\sum_{d|n} n/d\ \mu(d) = \phi(n)$, nous obtenons finalement

$$\tau P^{\mathcal{C}}(z,u) = \sum_{k\geq 1} \phi(k) \frac{u^k}{1 - u^k A(z^k)} \tau Q(z^k).$$

En faisant $u = 1$, on obtient bien le résultat annoncé. ∎

Nous sommes maintenant en mesure d'énoncer le théorème principal de ce chapitre, qui fonde l'analyse automatique dans Π.

**Théorème 4.** *Tout programme dont la spécification des données est dans* $\Omega$*, et dont les procédures sont définies à partir d'instructions élémentaires, des schémas d'exécution séquentielle, de sélection dans une union ou dans un produit cartésien, de sélection et d'itération dans une séquence, un ensemble, un cycle ou un multi-ensemble se traduit en un système d'équations pour les descripteurs de complexité associés aux procédures.*

*Ces équations sont de la forme* $\tau = f$, $\tau = \tau' + \tau''$, $\tau = f\tau'$, $\tau = \Theta(\tau', f)$ *où* $\tau$, $\tau'$ *et* $\tau''$ *sont des descripteurs de complexité,* $f$ *une fonction de la classe* $\mathcal{UR}$ *(cf théorème 3) et* $\Theta$ *l'un des opérateurs*

$$\Psi_S^i \quad : \quad (\tau, f) \to f \cdot (\tau(z) - \tau(z^2) + \tau(z^3) - \cdots)$$

$$\begin{aligned}
\Psi^s_S &: (\tau, f) \to \sum_{n=1}^{\infty} \tau_n z^n \int_0^1 \frac{\prod_{i\geq 1}(1+uz^i)^{f_i}}{1+uz^n} du \\
\Psi^i_M &: (\tau, f) \to f \cdot (\tau(z) + \tau(z^2) + \tau(z^3) + \cdots) \\
\Psi^s_M &: (\tau, f) \to \sum_{n=1}^{\infty} \tau_n z^n \int_0^1 \frac{\prod_{i\geq 1}(1-uz^i)^{-f_i}}{1-uz^n} du \\
\Psi^s_C &: (\tau, f) \to \sum_{k\geq 1} \frac{\phi(k)}{k} \frac{1}{f(z^k)} \log \frac{1}{1-f(z^k)} \tau(z^k) \\
\Psi^i_C &: (\tau, f) \to \sum_{k\geq 1} \phi(k) \frac{1}{1-f(z^k)} \tau(z^k).
\end{aligned}$$

**Démonstration :** La première affirmation découle des règles 8 à 19. Pour montrer la seconde assertion, il suffit de vérifier que le membre droit de chaque règle est de l'une des formes susmentionnées. La règle 8 (coût constant) donne une équation de la forme $\tau = f$. Les règles 9 et 10 donnent $\tau = \tau' + \tau''$, et la règle 11 donne $\tau = f\tau'$. Les règles 12 et 13 conduisent à $\tau = \tau'/(1-f)$ et $\tau = \tau'/(1-f)^2$ avec $f \in \mathcal{UR}$ : or $1/(1-f) = Q(f)$ et $1/(1-f)^2 = Q(f) \times Q(f)$ sont aussi dans $\mathcal{UR}$. Les six règles de sélection ou d'itération sur un ensemble, un cycle ou un multi-ensemble conduisent à une équation de la forme $\tau = \Theta(\tau', f)$ où $\Theta$ est l'un des six opérateurs $\Psi^i_S$, $\Psi^i_M$, $\Psi^s_S$, $\Psi^s_M$, $\Psi^s_C$ et $\Psi^i_C$. ∎

Les équations des descripteurs de complexité sont *linéaires* par rapport à ceux-ci, en ce sens que les expressions des coefficients sont de la forme

$$\tau_n = \sum \tau'_i f_j g_k \ldots$$

où $f_j$, $g_k$, ... sont des coefficients de séries génératrices. Cette propriété de linéarité provient en fait de l'additivité de la notion de coût (en termes physiques, les séries génératrices de dénombrement n'ont pas de dimension, alors que les descripteurs de complexité ont la dimension d'un coût, par exemple un temps).

### Tests sur la taille et la longueur

Deux règles supplémentaires permettent d'analyser les procédures effectuant une sélection par la taille ou la longueur des objets, et complètent la preuve du caractère systématique de l'analyse des programmes de II. Ces deux règles ont été volontairement séparées des autres car elles font apparaître d'autres objets mathématiques : le test sur la taille introduit des séries tronquées et des polynômes, tandis que le test sur la longueur introduit des séries à deux variables.

**Règle 20.** *(Test sur la taille)*

$$\frac{\textbf{procedure } P(a:A);\ \textbf{if } size(a) <= k \textbf{ then } Q(a) \textbf{ else } R(a);}{\tau P(z) = \tau Q_{\leq k}(z) + \tau R_{>k}(z)}$$

*où* $\tau Q_{\leq k}(z) = \sum_{j\leq k} \tau Q_j z^j$ *et* $\tau R_{>k}(z) = \sum_{j>k} \tau R_j z^j$.

**Démonstration :** $\tau P(z) = \sum_{|a|\leq k} \tau Q\{a\} z^{|a|} + \sum_{|a|>k} \tau R\{a\} z^{|a|} = \tau Q_{\leq k}(z) + \tau R_{>k}(z)$. ∎

**Règle 21.** *(Test sur la longueur)*

$$\frac{\textbf{procedure } P(a:A);\ \textbf{if } card(a) <= k \textbf{ then } Q(a) \textbf{ else } R(a);}{\tau P(z) = \tau R(z) + \sum_{j=0}^{k} [u^j](\tau Q(z,u) - \tau R(z,u))}$$

**Démonstration :** Ici, il faut déterminer les descripteurs de complexité à deux variables $\tau Q(z,u)$ et $\tau R(z,u)$ définis par

$$\tau Q(z,u) = \sum_{a \in \mathcal{A}} \tau Q\{a\} z^{|a|} u^{l(a)}.$$ ■

Ce dernier schéma n'est pas admissible au sens strict où nous l'avons défini, car il n'est pas possible de calculer $\tau P_n$ à partir des seuls coefficients $A_n$, $\tau Q_n$ et $\tau R_n$. Néanmoins, l'introduction d'une variable auxiliaire $u$ comptant la longueur permet d'exprimer $\tau P_n$ en fonction des coefficients $\tau Q_{n,k} = [z^n u^k]\tau Q(z,u)$ et $\tau R_{n,k} = [z^n u^k]\tau R(z,u)$.

EXEMPLE 18 : Considérons un algorithme prenant en entrée un arbre binaire non planaire (les branches ne sont pas ordonnées), et descendant de façon aléatoire à chaque niveau dans l'une des deux branches, jusqu'à atteindre une feuille. Supposons que l'on désire étudier le nombre moyen de nœuds parcourus par cet algorithme. Pour cela on commence par définir les arbres binaires non planaires.

```
type B = x | product(x,multiset(B, card=2));
     x = atom(1);
```

Un tel arbre est soit une feuille, soit un nœud auquel est attaché un multi-ensemble de deux arbres. Puis on définit une procédure sur ces arbres, qui ajoute un coût d'une unité pour chaque nœud parcouru.

```
procedure H (b : B);
begin
   count1;
   case b of
      x : count0;
      (x,m) : forone c in m do H(c)
   end
end;

measure count1 : 1;  count0 : 0;
```

A l'aide de l'équation (2.16) pour la traduction de l'instruction `forone c in m do H(c)`, l'analyse de ce programme conduit aux équations

$$B(z) = z + \frac{z}{2}(B(z^2) + B^2(z)), \qquad \tau H(z) = B(z) + \frac{z}{2}[B(z)\tau H(z) + \tau H(z^2)].$$

□

EXEMPLE 19 : (Nombre de sommants d'une partition) Nous pouvons à présent analyser le programme donné comme exemple introductif tout au début du chapitre précédent. Les partitions sont définies comme suit :

```
type partition = multiset(entier);
     entier = sequence(un, card >= 1);
```

un $=atom(1)$;

Par application des règles 1, 4 et 6, les séries génératrices $P(z)$ du type `partition` et $E(z)$ du type `entier` sont

$$P(z) = \exp(E(z) + \frac{E(z^2)}{2} + \frac{E(z^3)}{3} + \cdots) \qquad \text{et} \qquad E(z) = \frac{z}{1-z}.$$

Quant au programme lui-même, il peut s'écrire

```
procedure compte_sommants (p : partition);
begin
   forall e in p do
      compte(e)
end;

procedure compte (e : entier);
begin
   count
end;

measure count : 1;
```

Soit $\tau CS(z)$ le descripteur de complexité de `compte_sommants` et $\tau C$ celui de `compte`. Les règles 16 et 8 donnent

$$\tau CS(z) = P(z) \cdot (\tau C(z) + \tau C(z^2) + \tau C(z^3) + \cdots) \qquad \text{et} \qquad \tau C(z) = E(z),$$

ce qui prouve le théorème suivant.

**Théorème automatique 3.** *Le nombre moyen de sommants dans les partitions de l'entier $n$ est*

$$\overline{s}_n = \frac{[z^n]P(z)D(z)}{[z^n]P(z)} \tag{2.25}$$

*où* $P(z) = \exp(\frac{z}{1-z} + \frac{z^2}{2(1-z^2)} + \frac{z^3}{3(1-z^3)} + \cdots)$, *et* $D(z) = \frac{z}{1-z} + \frac{z^2}{1-z^2} + \frac{z^3}{1-z^3} + \cdots$.

En ce qui concerne l'étude asymptotique de $\overline{s}_n$, nous indiquons ici uniquement le principe de l'analyse. On utilise la "méthode de col", qui s'applique aux fonctions à forte croissance au voisinage de leur singularité dominante. En effet, la fonction $P(z)$ est à croissance exponentielle au voisinage de $z = 1$ :

$$P(z) \approx \tilde{P}(z) = \exp\left(\frac{\pi^2}{6}\frac{1}{1-z}\right).$$

Au contraire, la fonction $D(z)$ est à croissance moins violente :

$$D(z) \sim \tilde{D}(z) = \frac{1}{1-z}\log\frac{1}{1-z}.$$

Les fonctions $\tilde{P}(z)$ et $\tilde{P}(z)\tilde{D}(z)$ sont de la forme

$$\frac{1}{(1-z)^\alpha}\log^\beta\frac{1}{1-z}\exp(\frac{\gamma}{1-z}).$$

Macintyre et Wilson [MW54] ont montré que les fonctions de cette classe vérifient les conditions de la méthode de col. Par conséquent, nous pouvons appliquer la méthode de col à $\tilde{P}(z)$ et $\tilde{P}(z)\tilde{D}(z)$.

L'étape suivante consiste à dire que la fonction $\tilde{D}(z)$ variant beaucoup moins violemment que $\tilde{P}(z)$, elle peut être considérée comme constante dans le voisinage du point col donnant la contribution principale de l'intégrale de col. Il s'ensuit que le point col de $\tilde{P}(z)\tilde{D}(z)$ est le même que celui de $\tilde{P}(z)$, c'est-à-dire $\zeta \sim 1 - \sqrt{\frac{\pi^2}{6n}}$, et que

$$\frac{[z^n]\tilde{P}(z)\tilde{D}(z)}{[z^n]\tilde{P}(z)} \sim \tilde{D}(\zeta) \sim \sqrt{\frac{3n}{2\pi^2}}\log n. \tag{2.26}$$

Pour appliquer la méthode à $P(z)$ et $D(z)$, la difficulté provient du fait que le cercle $|z| = 1$ est "frontière", c'est-à-dire que les singularités de $P(z)$ et de $D(z)$ ont une répartition dense le long de ce cercle. Néanmoins le développement asymptotique ci-dessus doit rester valable.

Nous avons comparé le nombre moyen exact $\overline{s}_n$ de sommants des partitions de $n$ à l'équivalent $\delta_n = \sqrt{\frac{3n}{2\pi^2}}\log n$ donné par (2.26) : $\overline{s}_{50}/\delta_{50} \simeq 1.27$, $\overline{s}_{100}/\delta_{100} \simeq 1.21$, $\overline{s}_{200}/\delta_{200} \simeq 1.17$. □

### 2.2.2 Sélection, itération et dérivation

Nous présentons dans ce paragraphe une méthode à la fois plus élégante et plus générale pour obtenir les opérateurs correspondant aux schémas d'itération et de sélection.

Soit $\Phi$ un multi-constructeur. Par définition, l'ensemble produit par $\Phi$ à partir d'un ensemble $\mathcal{B}$ s'écrit

$$\Phi(\mathcal{B}) = \bigcup_{(b_1,\ldots,b_l)\in I} \Phi(b_1,\ldots,b_l). \tag{2.27}$$

où $I$ rassemble les séquences de composantes des objets de $\Phi(\mathcal{B})$.

**Lemme 8.** *Soit $\Phi$ un multi-constructeur idéal et admissible, tel que $A = \Phi(B)$ se traduise en $A(z) = \Psi(B(z), B(z^2), \ldots)$. Alors le descripteur de complexité du schéma d'itération d'une procédure $Q$ sur $\mathcal{A}$ est*

$$\tau P_{\rm it}(z) = \tau Q(z)\frac{\partial\Psi}{\partial x_1}(B(z), B(z^2), \ldots) + 2\tau Q(z^2)\frac{\partial\Psi}{\partial x_2}(B(z), B(z^2), \ldots) + \cdots \tag{2.28}$$

*et le descripteur de complexité du schéma de sélection est*

$$\tau P_{\rm sel}(z) = \int_0^1 \left[\tau Q(z)\frac{\partial\Psi}{\partial x_1} + 2u\tau Q(z^2)\frac{\partial\Psi}{\partial x_2} + 3u^2\tau Q(z^3)\frac{\partial\Psi}{\partial x_3} + \cdots\right]_{x_j = u^j B(z^j)} du. \tag{2.29}$$

**Démonstration :** L'équation (2.27) donne pour les séries génératrices :

$$\Psi(B(z), B(z^2), \ldots) = \sum_{(b_1,\ldots,b_l)\in I} z^{|b_1|}\ldots z^{|b_l|}.$$

Si dans le membre droit, nous remplaçons chaque $z^{|b_j|}$ par $(1 + v\tau Q\{b_j\})z^{|b_j|}$, nous allons obtenir dans le membre gauche $\Psi(B_1(z,v), B_2(z,v), \ldots)$ où $B_\alpha(z,v) = \sum_{b\in\mathcal{B}}(1 + v\tau Q\{b\})^\alpha z^{\alpha|b|}$, soit

$$\Psi(B_1(z,v), B_2(z,v), \ldots) = \sum_{(b_1,\ldots,b_l)\in I} (1 + v\tau Q\{b_1\})z^{|b_1|}\ldots(1 + v\tau Q\{b_l\})z^{|b_l|}. \tag{2.30}$$

En dérivant cette dernière équation par rapport à $v$, puis en remplaçant $v$ par 0, il vient :

$$\left.\frac{\partial\Psi(B_1(z,v),B_2(z,v),\ldots)}{\partial v}\right|_{v=0} = \tau P_{\rm it}(z). \tag{2.31}$$

Or la dérivée partielle de $B_\alpha(z,v)$ par rapport à $v$, prise en $v=0$, vaut $\alpha\tau Q(z^\alpha)$, ce qui donne la formule annoncée pour le schéma d'itération.

Pour le schéma de sélection, nous rajoutons une troisième variable $u$ qui compte le nombre de composantes. Soit $B_\alpha(z,u,v) = \sum_{b\in\mathcal{B}}(1+v\tau Q\{b\})^\alpha u^\alpha z^{\alpha|b|}$, alors l'équation (2.30) devient

$$\Psi(B_1(z,u,v),B_2(z,u,v),\ldots) = \sum_{(b_1,\ldots,b_l)\in I} u^l(1+v\tau Q\{b_1\})z^{|b_1|}\ldots(1+v\tau Q\{b_l\})z^{|b_l|},$$

et

$$\left.\frac{\partial\Psi(B_1(z,u,v),B_2(z,u,v),\ldots)}{\partial v}\right|_{v=0} = \sum_{(b_1,\ldots,b_l)\in I} u^l(\tau Q\{b_1\}+\cdots+\tau Q\{b_l\})z^{|b_1|}\ldots z^{|b_l|},$$

donc le descripteur de complexité du schéma de sélection est

$$\tau P_{\rm sel}(z) = \int_0^1 \frac{1}{u}\left.\frac{\partial\Psi(B_1(z,u,v),B_2(z,u,v),\ldots)}{\partial v}\right|_{v=0} du.$$

Or la dérivée partielle de $B_\alpha(z,u,v)$ par rapport à $v$ vaut $\alpha u^\alpha\tau Q(z^\alpha)$ en $v=0$. ■

Pour le constructeur séquence par exemple, l'opérateur associé est $\Psi(x_1,x_2,\ldots) = 1/(1-x_1)$, qui ne dépend que de $x_1$. La dérivée partielle par rapport à $x_1$ vaut $1/(1-x_1)^2$ : l'équation (2.28) donne $\tau Q(z)/(1-B(z))^2$ pour l'itération. Pour la sélection, nous devons calculer $\int_0^1 1/(1-ux_1)^2du$, qui vaut $1/(1-x_1)$, ce qui donne comme opérateur $\tau Q(z)/(1-B(z))$.

Pour le constructeur multi-ensemble, nous avons $\Psi(x_1,x_2,x_3,\ldots) = \exp(x_1+x_2/2+x_3/3+\cdots)$. Comme $\frac{\partial\Psi}{\partial x_i}$ vaut $\Psi(x_1,x_2,\ldots)/i$, et que l'on multiplie à nouveau par $i$ dans (2.28), l'opérateur associé à l'itération est finalement $\Psi(x_1,x_2,\ldots)(\tau Q(z)+\tau Q(z^2)+\cdots)$. Pour la sélection, l'équation (2.29) s'écrit ici

$$\tau P(z) = \int_0^1 [\tau Q(z)+u\tau Q(z^2)+u^2\tau Q(z^3)+\cdots]\exp(uB(z)+u^2B(z^2)/2+\cdots)du.$$

Cette expression semble différer de celle donnée par la règle 17, mais nous retrouvons facilement la forme alternative (2.17) en décomposant $\exp(uB(z)+u^2B(z^2)/2+\cdots)$ en $\sum_{k\geq0}u^k\Psi_{\mathcal{M},k}(B,z)$.

Pour le constructeur cycle orienté, l'opérateur de dénombrement est $\sum_{k\geq1}\phi(k)/k\log(1/(1-x_k))$. Sa dérivée partielle par rapport à $x_k$ vaut $\phi(k)/k\ 1/(1-x_k)$, et par conséquent l'opérateur d'itération est $\sum_{k\geq1}\phi(k)\tau Q(z^k)/(1-B(z^k))$. Quant à la sélection, l'équation (2.29) donne après inversion de la sommation sur $k$ et de l'intégrale

$$\sum_{k\geq1}\phi(k)\tau Q(z^k)\int_0^1\frac{u^{k-1}du}{1-u^kB(z^k)},$$

or l'intégrale ci-dessus vaut $1/(kB(z^k))\log(1/(1-B(z^k)))$, et l'on retrouve le résultat de la règle 18.

## 2.3 Complexité du calcul des coefficients

Nous avons montré que les programmes de la classe Π se traduisent en équations pour les séries génératrices de dénombrement et de coût cumulé (théorème 4). Ce résultat serait d'utilité restreinte si le calcul du coût moyen à partir de ces équations était de même ordre de complexité que la méthode consistant à évaluer le programme. En réalité, le calcul du coût moyen exact d'un programme de Π à partir des équations produites par l'analyse algébrique est de complexité polynomiale (théorème 6).

Cette section est divisée en deux : nous nous intéressons aux coefficients des séries de dénombrement d'abord, puis à ceux des descripteurs de complexité.

### 2.3.1 Dénombrement des structures de données

Nous montrons ici que le dénombrement des structures de données de taille $n$ s'effectue essentiellement en $O(n^2)$ opérations par des méthodes classiques.

**Théorème 5.** *Soit $\Sigma$ une spécification bien fondée de $\Omega$. Le calcul des coefficients des séries génératrices ordinaires des non-terminaux de $\Sigma$ jusqu'à l'ordre $n$ coûte*

$$O(|\Sigma| n^2)$$

*opérations arithmétiques sur les coefficients (additions, soustractions, multiplications, divisions), où $|\Sigma|$ est le nombre de non-terminaux dans la forme normale de Chomsky de $\Sigma$ (cf section 1.2.3).*

**Démonstration :** Nous mettons d'abord $\Sigma$ en *Chomsky Normal Form*, puis nous utilisons l'algorithme suivant.

**Algorithme Calcul des coefficients.**
Donnée : une spécification $\Sigma$ en *Chomsky Normal Form* et un entier $n$.
Résultat : les coefficients $A_k$ pour tout non-terminal $A$ et $0 \leq k \leq n$.

**pour** $k := 0$ **jusqu'à** $n$ **faire**
  $\mathcal{E} \leftarrow \mathcal{N}$ (ensemble des non-terminaux)
  **tant que** $\mathcal{E} \neq \{\}$ **faire**
    choisir $A$ minimal dans $\mathcal{E}$ : $\nexists B \in \mathcal{E}, B \prec_k A$ (cf section 1.4.3)
    calculer $A_k$ (à l'aide des formules ci-dessous)
    $\mathcal{E} \leftarrow \mathcal{E} - \{A\}$
  **fin tant que**
**fin pour**

L'existence d'un $A$ minimal à chaque passage dans la boucle **while** est garantie par le fait que l'ordre partiel $\prec_k$ n'admet pas de cycle, car $\Sigma$ est bien fondée. Il nous reste à savoir calculer $A_k$ en fonction des $B_i$ pour $i < k$ et des $C_k$ pour $C \leq_k A$.

**atomes :** si $A \rightarrow a$, alors $A_k = 1$ si $|a| = k$, et 0 sinon. Le coût (nombre d'opérations arithmétiques) est $O(1)$.

**union :** si $A \rightarrow B \mid C$, alors $A_k = B_k + C_k$. Le coût est aussi $O(1)$.

**produit :** si $A \rightarrow B \times C$, alors $A_k = \sum_{i=0}^{k} B_i C_{k-i}$. Le coût de la convolution est $O(k)$.

**séquence :** si $A \to B^*$, alors $A(z) = 1/(1 - B(z))$ s'écrit aussi $A(z) = 1 + B(z)A(z)$, d'où $A_0 = 1$ et pour $k > 0$, $A_k = \sum_{i=1}^{k} B_i A_{k-i}$. Le coût est ici aussi celui d'une convolution, c'est-à-dire $O(k)$.

**multi-ensemble :** si $A \to \mathcal{M}(B)$, alors $A(z) = \exp(B(z) + B(z^2)/2 + \cdots)$. Soit $F(z) = B(z) + B(z^2)/2 + \cdots$ : $A = \exp(F)$ donc $A' = AF'$, et $A_k = 1/k[z^{k-1}]AF'$. Supposons que l'on ait stocké les coefficients de $F$ jusqu'à l'ordre $k - 1$. Pour calculer $A_k$, nous calculons d'abord $F_k = [z^k](B(z) + B(z^2)/2 + \cdots + B(z^k)/k) = B_k + B_{k/2}/2 + \cdots + B_1/k$, avec la convention $B_{k/i} = 0$ lorsque $k/i$ n'est pas entier. Ensuite nous stockons $F_k$ pour les calculs futurs, et nous déterminons $A_k$ par la convolution ci-dessus. Les calculs de $F_k$ et de $A_k$ un coût total en $O(k)$.

**ensemble :** la méthode est la même que pour un multi-ensemble, à la seule différence que $F(z) = B(z) - B(z^2)/2 + \cdots$.

**cycle :** si $A \to \mathcal{C}(B)$, alors $A(z) = \log(1 - B(z))^{-1} + 1/2 \log(1 - B(z^2))^{-1} + \cdots + \phi(k)/k \log(1 - B(z^k))^{-1} + \cdots$. Comme pour le constructeur multi-ensemble, nous stockons les coefficients de $G(z) = \log(1 - B(z))^{-1}$. Le calcul d'un nouveau coefficient de $G$ s'effectue en utilisant le fait que $G' = B'/(1 - B)$, qui s'écrit aussi $G' = B' + BG'$. Cette dernière formule donne $G_k = 1/k[z^{k-1}](B' + BG')$. Puis $A(z) = G(z) + 1/2\ G(z^2) + \cdots + \phi(k)/k\ G(z^k) + \cdots$, donc $A_k = G_k + 1/2\ G_{k/2} + \cdots + \phi(k)/k\ G_1$ avec la même convention que ci-dessus. Les calculs de $G_k$ et $A_k$ coûtent $O(k)$ opérations (on suppose les $\phi(k)$ connus).

Quel que soit le constructeur, le calcul d'un coefficient $A_k$ nécessite au plus $O(k)$ opérations. Une étape de la boucle principale de l'algorithme **Calcul des coefficients**, calculant $A_k$ pour chaque non-terminal de la spécification en *Chomsky Normal Form*, nécessite $O(|\Sigma|k)$ opérations. Le coût du calcul de tous les coefficients jusqu'à l'ordre $n$ est donc $O(|\Sigma|n^2)$. ■

Ce théorème signifie que les règles de dénombrement de la section 2.1 sont effectivement utilisables, puisqu'elles permettent de dénombrer les objets en temps polynomial.

Le nombre de registres utilisés par l'algorithme **Calcul des coefficients** pour stocker les coefficients est $O(|\Sigma|n)$. En effet, pour chaque non-terminal $A$ de la spécification en *Chomsky Normal Form*, il faut stocker les $n + 1$ coefficients $A_k$, $0 \leq k \leq n$, plus éventuellement ceux de $F$ (ensemble ou multi-ensemble), ou de $G$ (cycle).

REMARQUE : Le coût $O(|\Sigma|n^2)$ donné par le théorème est relatif aux opérations élémentaires sur les *coefficients*. Or l'analyse asymptotique montre que ceux-ci croissent en $A^n$, où $A \geq 1$. L'espace mémoire utilisé par un coefficient est donc $O(B_\Sigma\, n)$, et le coût du calcul des coefficients en nombre d'opérations élémentaires sur des mots-machine est

$$O(C_\Sigma\, n^2 M(n))$$

où $M(n)$ est le coût des opérations élémentaires sur des nombres de $n$ mots-machine, et $C_\Sigma$ une constante dépendant de $|\Sigma|$ et du plus grand facteur $A$ de croissance exponentielle des suites de dénombrement de $\Sigma$.

Pour diminuer le nombre d'opérations, on peut utiliser des méthodes de calcul plus évoluées comme la transformée de Fourier rapide (FFT), à la fois pour le calcul des suites de dénombrement, et pour les opérations entre coefficients. Avec ces méthodes associées à la méthode de Newton sur les séries, on obtient à chaque étape deux fois plus de termes exacts dans les suites de dénombrement,

au lieu d'un seul pour l'algorithme donné ci-dessus. Nous obtenons ainsi un nouvel algorithme de calcul des coefficients, coûtant $O(|\Sigma| n \log n)$ opérations sur les coefficients, au lieu de $O(|\Sigma| n^2)$. Pour plus de détails, nous renvoyons le lecteur à l'annexe B, où nous montrerons aussi que la complexité est linéaire dans le cas des familles simples d'arbres (par exemple le type `expression` du programme de dérivation).

### 2.3.2 Calcul du coût des procédures

Nous supposons à présent que les suites de dénombrement sont connues jusqu'à l'ordre $n$, et nous évaluons ici le coût additionnel pour calculer les coefficients des descripteurs de complexité, toujours jusqu'à l'ordre $n$.

Nous nous limitons ici aux programmes n'utilisant pas le schéma de test sur la longueur, qui introduirait des séries bivariées. Étant donné un programme bien fondé, en vertu du lemme 6, il existe pour chaque taille $n$ un ordre partiel $\prec_n$ sur les procédures (avec le test sur la longueur, il y aurait plusieurs ordres partiels $\prec_{n,k}$ éventuellement différents, et par conséquent il faudrait calculer séparément les parties correspondantes des descripteurs de complexité bivariés).

De nombreuses règles induisent des formules analogues à celles obtenues pour les types. Pour ces règles, nous connaissons déjà le coût du calcul d'un nouveau coefficient $\tau P_k$ : les règles 8 (coût constant), 9 (séquence d'instructions) et 10 (sélection dans une union) nécessitent $O(1)$ opérations ; la règle 11 (sélection dans un produit cartésien) et la règle 20 (test sur la taille) en nécessitent $O(k)$. Il nous reste à étudier les autres règles :

**sélection et itération dans une séquence (règles 12 et 13) :** pour la sélection, $\tau P = \tau Q/(1-B)$ s'écrit aussi $A\tau Q$ puisque $A = 1/(1-B)$ : calculer $\tau P_k$ coûte $O(k)$ opérations. Pour l'itération, $\tau P = \tau Q/(1-B)^2$ s'écrit $F\tau Q$, où $F = A^2$. Si l'on stocke les coefficients de $F$ au fur et à mesure, le calcul de $\tau P_k$ coûte deux convolutions, c'est-à-dire $O(k)$.

**itération dans un ensemble et dans un multi-ensemble (règles 14 et 16) :** $\tau P(z)$ vaut $A(z)F(z)$, où $F(z) = \tau Q(z) \pm \tau Q(z^2) + \tau Q(z^3) \pm \cdots$. Si l'on stocke les coefficients de $F(z)$, le calcul d'un nouveau coefficient coûte $O(k)$.

**sélection dans un ensemble (règle 15) :** une méthode possible consiste à calculer dans un premier temps les coefficients de la série à deux variables $B^{\mathcal{S}}(z,u)$ :

$$B_{n,k} = [z^n u^k] \prod_{j \geq 1} (1 + uz^j)^{B_j},$$

puis à calculer les coefficients de $\tau P(z)$ en fonction des $B_{n,k}$ (voir la section B.5 de l'annexe B pour plus de détails). Les calculs sont faits au fur et à mesure, en stockant les $B_{n,k}$. Le calcul d'un nouveau coefficient $\tau P_k$ par cette méthode nécessite $O(k^2 \log k)$ opérations sur les coefficients.

**sélection dans un multi-ensemble (règle 17) :** la méthode est similaire à celle utilisée pour le constructeur ensemble (voir l'annexe) ; le coût est le même aussi.

**itération dans un cycle (règle 19) :** le descripteur de complexité est

$$\tau P(z) = \sum_{j \geq 1} \phi(j) \frac{\tau Q(z^j)}{1 - A(z^j)},$$

ce qui s'écrit également $\tau P(z) = \sum_{j\geq 1} \phi(j) F(z^j)$ en posant $F = \tau Q/(1-A)$. Comme $F(0) = 0$,

$$\tau P_k = \sum_{j=1}^{k} \phi(j)[z^k]F(z^j) = F_k + F_{k/2} + 2F_{k/3} + \cdots + \phi(j)F_{k/j} + \cdots + \phi(k)F_1, \quad (2.32)$$

avec la convention habituelle que $F_{k/j}$ est nul lorsque $j$ ne divise pas $k$. Les coefficients de $F$ sont calculés au fur et à mesure par la formule $F_k = [z^k](\tau Q + AF)$ (coût $O(k)$), puis on détermine $\tau P_k$ par l'égalité (2.32), ce qui coûte aussi $O(k)$ opérations.

**sélection dans un cycle (règle 18) :** ici, nous avons

$$\tau P(z) = \sum_{j\geq 1} \frac{\phi(j)}{j} F(z^j) \text{ avec } F(z) = \frac{\tau Q(z)}{A(z)} \log \frac{1}{1-A(z)}.$$

Le coefficient $\tau P_k$ est à nouveau donné par l'équation (2.32), sauf que $F$ n'est plus la même série ; $F_k$ s'obtient par exemple comme $[z^k]G(z)\tau Q(z)$, avec $G(z) = \frac{1}{A(z)} \log \frac{1}{1-A(z)}$ vérifiant $(AG)' = A'/(1-A)$. Si l'on stocke les coefficients de $1/(1-A)$ et de $G$, le calcul d'un nouveau coefficient $F_k$ nécessite $O(k)$ opérations.

Cette étude des différents schémas possibles nous permet d'énoncer le théorème suivant.

**Théorème 6.** *Le calcul des descripteurs de complexité d'un programme $\pi$ de la classe* Π *jusqu'à l'ordre $n$ coûte*

$$O(|\pi| n^3 \log n)$$

*opérations sur les coefficients des séries de dénombrement et des descripteurs de complexité, où $|\pi|$ est une constante dépendant du programme. Lorsque $\pi$ n'utilise pas les schémas de sélection dans un ensemble ou un multi-ensemble, $O(|\pi| n^2)$ opérations suffisent.*

En conclusion, pour les programmes de Π n'utilisant pas les schémas de sélection dans un ensemble ou un multi-ensemble, le calcul du coût moyen exact sur les données de taille $n$ prend un temps (mesuré en opérations sur les bits, *bit complexity* en anglais) $O(n^4)$ avec des algorithmes élémentaires, et $O(n^2 \log^2 n)$ avec des algorithmes rapides En pratique, cela signifie que l'on peut calculer le coût moyen *exact* jusqu'à des valeurs de $n$ de l'ordre de la centaine (algorithmes élémentaires) ou du millier (algorithmes rapides).

On peut aussi calculer les suites de dénombrement en précision fixe, et on obtient alors un coût moyen *approché* en temps $O(n^2)$ et $O(n \log n)$ respectivement, ce qui permet d'atteindre des valeurs de $n$ beaucoup plus élevées.

EXEMPLE 20 : (Calcul du coût moyen du programme de dérivation formelle) Le tableau ci-dessous indique pour différentes valeurs de $n$, le temps de calcul en secondes du coût moyen exact du programme de dérivation formelle par l'égalité

$$\overline{\tau \text{diff}}_n = \frac{\tau \text{diff}_n}{E_n} = \frac{[z^n]\tau \text{diff}(z)}{[z^n]\text{expression}(z)}$$

à l'aide de MAPLE et sur un Sun 3/60. Pour calculer $E_n$, on a utilisé le fait que la série $E(z) = \sum E_n z^n$ vérifie une équation polynomiale de degré 2, qui se transforme grâce à une remarque de Comtet (voir à ce propos la section B.4) en une récurrence linéaire pour les coefficients.

| $n$ | 10 | 20 | 50 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|---|
| nombre d'expressions (ordre de grandeur) | $10^6$ | $10^{13}$ | $10^{36}$ | $10^{74}$ | $10^{150}$ | $10^{381}$ | $10^{766}$ |
| temps de calcul du coût moyen | $0.1s$ | $0.2s$ | $0.6s$ | $1.3s$ | $2.9s$ | $13s$ | $42s$ |

Par exemple, le coût moyen de la procédure `diff` sur les expressions de taille 100 est :

$$\overline{\tau\mathrm{diff}}_{100} = \frac{5394852102222784230979085204746031784198845689609126324750014786507374368823 7}{6359356154818995683623500594592300823007470008359125866692903795004909954 5}.$$

Si l'on envisageait de calculer ce coût moyen par évaluation de la procédure, il faudrait générer toutes les expressions de taille 100, soit plus de $10^{73}$ expressions. Or par les séries génératrices, en moins d'une minute, nous avons même calculé $\overline{\tau\mathrm{diff}}_{1000}$, le coût moyen exact pour une taille dix fois plus grande, grâce au théorème 6 qui garantit une complexité polynomiale pour le calcul des coefficients.

Nous pouvons aussi comparer le coût moyen exact avec les premiers termes du développement obtenu par analyse asymptotique. Toujours pour le programme de dérivation, le tableau suivant indique la différence relative entre le coût moyen exact et les $k$ premiers termes du développement asymptotique (ligne $\delta_k$).

| $n$ | 10 | 20 | 50 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|---|
| $\delta_1$ | 0.2 | 0.1 | 0.08 | 0.05 | 0.04 | 0.02 | 0.02 |
| $\delta_2$ | 0.05 | 0.03 | 0.01 | 0.007 | 0.004 | 0.002 | 0.0008 |
| $\delta_3$ | 0.02 | 0.007 | 0.002 | 0.0007 | 0.0003 | 0.00006 | 0.00002 |
| $\delta_4$ | 0.004 | 0.0003 | 0.00007 | 0.00002 | 0.000004 | 0.0000007 | 0.0000002 |
| $\delta_5$ | 0.005 | 0.0001 | 0.000002 | 0.0000004 | 0.00000008 | 0.000000008 | 0.000000001 |
| $\delta_6$ | 0.005 | 0.00008 | 0.0000001 | 0.00000002 | 0.000000002 | 0.0000000001 | 0.00000000002 |

Pour cette comparaison, nous avons calculé les premiers termes du développement asymptotique de $\overline{\tau\mathrm{diff}}_n$ (grâce au programme `equivalent`) avec 20 chiffres significatifs :

$$\overline{\tau\mathrm{diff}}_n \simeq 0.80421754408588654\,9\, n^{3/2} + 0.38063627836733710\,5\, n + 0.66472324010206045\,7\, n^{1/2} - 0.58051519414966478\,1 - 0.15189351161001618\,7\,\tfrac{1}{n^{1/2}} + 0.03508383746173822\,8\,\tfrac{1}{n} + O(\tfrac{1}{n^{3/2}}).$$

En regardant les colonnes du tableau, on voit bien que chaque terme fait gagner un facteur de $\sqrt{n}$ en précision, soit par exemple une décimale pour $n = 100$. D'autre part, en considérant les lignes, on constate qu'à nombre de termes fixé, l'erreur relative diminue lorsque $n$ augmente : par exemple, pour $n$ supérieur à 1000, le premier terme donne une approximation à moins de 2% près. □

# Chapitre 3

# Univers étiqueté

*Le mieux qu'un homme puisse faire de sa vie, c'est de transformer en conscience une expérience aussi vaste que possible.*

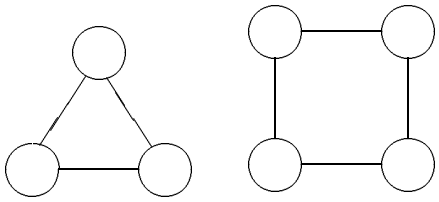
Camus

Dans les chapitres précédents, nous avons considéré des programmes opérant sur une certaine classe $\Omega$ de structures de données. Or cette classe convient mal pour modéliser certains problèmes, notamment lorsqu'il existe une relation d'ordre sous-jacente aux données. On introduit alors d'autres types de données, dites *étiquetées*, pour résoudre ce genre de problèmes. Par opposition, nous dirons que la classe $\Omega$ se trouve dans un univers non étiqueté. Les similarités sont grandes entre l'analyse en univers étiqueté et en univers non étiqueté, comme le font remarquer Bender et Goldman en 1970 [BG71] :

> *While studying the enumerative uses of generating functions we have come to the conclusion that a great deal of unity underlies the enumeration of "completely labeled" and "completely unlabeled" objects.*

Nous présentons cependant à part l'analyse algébrique en univers étiqueté car, même si les différences sont faibles en ce qui concerne les séries génératrices obtenues, la définition des structures de données nécessite une étude séparée. L'objet de ce chapitre est de montrer que l'analyse automatique est possible également en univers étiqueté, d'énoncer et de prouver les règles correspondantes.

Considérons des graphes non orientés dont chaque nœud est relié à exactement deux autres nœuds. Ces graphes particuliers sont dits 2-réguliers. Voici un graphe 2-régulier ayant 7 nœuds :



(G)

Dans le modèle étiqueté, chaque atome porte une information supplémentaire, appelée *étiquette*, qui permet de le comparer aux autres atomes. Une incarnation possible de ce modèle est de choisir les entiers de 1 à $n$ pour étiqueter les atomes d'un objet de taille $n$. Ainsi, en répartissant les entiers de 1 à 7 sur les nœuds du graphe *non étiqueté* (G), nous obtenons *plusieurs* graphes étiquetés, toujours 2-réguliers, dont par exemple les graphes (A) et (B) :


(A) (B)

Ces deux graphes sont différents car il n'est pas possible de superposer simultanément leur structure non étiquetée sous-jacente et les étiquettes. En effet, dans le graphe (A), l'étiquette 1 se trouve dans le cycle de longueur trois, alors que dans (B), elle se trouve dans celui de longueur quatre. Mais par contre, le graphe (B) et le graphe (C) ci-dessous sont identiques : il suffit de tourner le cycle de longueur trois de (B) d'un angle de $-2\pi/3$, et de faire faire une symétrie d'axe 3-5 au cycle de longueur quatre pour obtenir (C).


(C) (D)

Les graphes (A), (B) et (C) ont la même structure non étiquetée sous-jacente, en l'occurrence le graphe (G), alors que le graphe (D) est structurellement différent (il ne comporte pas de cycle de longueur quatre).

La première section de ce chapitre introduira des constructeurs sur les objets étiquetés, et définira la classe $\hat{\Omega}$ des spécifications utilisant ces constructeurs. Par exemple, les graphes 2-réguliers étiquetés seront définis par

**type** tworegg = *set*(component);
component = *ucycle*(node, *card*>=3);
node = *Latom*(1);

Nous verrons ensuite (section 3.2) les règles associées aux constructeurs de la classe $\hat{\Omega}$. A l'aide de ces règles, la spécification ci-dessus se traduira en

$$\hat{T}(z) = \exp(\hat{C}(z)), \quad \hat{C}(z) = \frac{1}{2}\log\frac{1}{1-z} - \frac{z}{2} - \frac{z^2}{4},$$

où $\hat{T}(z)$ et $\hat{C}(z)$ sont les séries génératrices *exponentielles* associées aux types `tworegg` et `component`. Nous définirons à la section 3.3 une classe $\hat{\Pi}$ de programmes sur $\hat{\Omega}$. Les schémas de programmation de $\hat{\Pi}$, comme en univers non étiqueté, seront essentiellement la sélection et l'itération sur les constructeurs de la classe $\hat{\Omega}$. Nous établirons dans cette même section les règles de traduction correspondant à ces schémas. Par exemple, pour compter le nombre de composantes connexes d'un graphe 2-régulier, on pourra écrire le programme suivant :

**procedure** visit (t : tworegg);
**begin**

```
    forall c in t do
        count
end;

measure count : 1;
```

dont l'analyse algébrique produira l'équation

$$\widehat{\tau V}(z) = \exp(\hat{C}(z))\,\hat{C}(z),$$

qui, par analyse asymptotique, conduira au résultat suivant :

**Théorème automatique 4.** *Le nombre moyen de composantes connexes d'un graphe 2-régulier étiqueté ayant $n$ nœuds est*

$$\overline{\tau V}_n = \frac{[z^n]\exp(\hat{C}(z))\,\hat{C}(z)}{[z^n]\exp(\hat{C}(z))},$$

*où* $\hat{C}(z) = \frac{1}{2}\log\frac{1}{1-z} - \frac{z}{2} - \frac{z^2}{4}$, *et vérifie asymptotiquement*

$$\overline{\tau V}_n = \frac{1}{2}\log n + O(1).$$

Les trois premières sections de ce chapitre sont par conséquent le pendant des chapitres 1 et 2, en univers étiqueté. Au passage, nous montrons une analyse automatique effectuée sur ordinateur (page 94). La dernière section (3.4) décrit quant à elle des constructeurs et des schémas spécifiques au modèle étiqueté. En résumé, nous pouvons avec ces nouveaux constructeurs imposer des conditions sur les étiquettes, tandis que les schémas associés utilisent l'ordre induit par les étiquettes pour diriger l'exécution du programme. Cette extension est particulièrement intéressante, dans la mesure où elle conduit à des équations *différentielles* sur les séries génératrices, type d'équations qu'il est impossible de rencontrer en univers non étiqueté (cf théorème 4). Des problèmes que l'on peut ainsi traiter sont par exemple la taille de l'intersection d'arbres tournois (section 3.4.5) ou le nombre de comparaisons de l'algorithme de tri *QuickSort* (section 5.2).

## 3.1 Objets étiquetés et constructeurs

Dans cette section, nous définissons une classe $\hat{\Omega}$ de spécifications en univers étiqueté. Pour cela, nous définissons les constructeurs de cette classe. Nous verrons que chaque constructeur de la classe $\Omega$ (univers non étiqueté) à un équivalent en univers étiqueté. Cette correspondance a pour conséquence que la détermination du caractère bien fondé des spécifications est décidable dans $\hat{\Omega}$ également.

ATOME ÉTIQUETÉ : les atomes étiquetés sont définis à l'aide du mot réservé *Latom* (pour *Labelled atom* en anglais) :

$$\textbf{type } a = \mathit{Latom}(1).$$

Cette déclaration définit $a$ comme un atome étiqueté de taille 1. Le constructeur *Latom* est *contagieux* : tout ensemble construit à partir d'atomes étiquetés doit être entièrement étiqueté. Cela signifie en particulier que tous les constructeurs utilisés dans la construction doivent être des

constructeurs de la classe $\hat{\Omega}$. En conséquence, un seul des deux mots réservés *atom* et *Latom* doit apparaître dans une spécification. Cette propriété de séparation des univers étiqueté et non étiqueté nous autorise à utiliser les mêmes symboles pour les constructeurs similaires de ces deux univers.

CONSTRUCTEUR UNION : les ensembles $\mathcal{B}$ et $\mathcal{C}$ contenant des objets étiquetés, l'union $\mathcal{B} \cup \mathcal{C}$ contient l'union des objets de $\mathcal{B}$ et de $\mathcal{C}$. Comme en univers non étiqueté, cette union doit s'interpréter au niveau des arbres de dérivation des objets. En Adl, nous noterons cette construction

$$\textbf{type } A = B \mid C.$$

REMARQUE : Dans une même spécification, nous ne mélangeons pas objets étiquetés et non étiquetés. Par conséquent, il n'est pas nécessaire de choisir des symboles différents pour les constructions étiquetées : la définition des atomes indique à elle seule dans quel univers on se place.

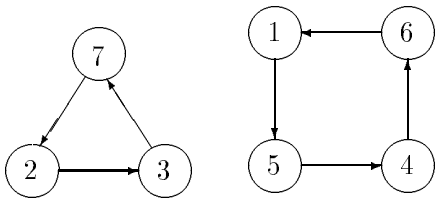
CONSTRUCTEUR PRODUIT PARTITIONNEL : si la notion naturelle d'union est analogue à celle définie en univers non étiqueté, il n'en est pas de même pour la notion de produit. Par exemple, considérons les deux objets étiquetés (E) et (F) ci-dessous :


(E) (F)

Si nous effectuons leur "produit cartésien", c'est-à-dire que nous les regroupons tels quels pour faire un seul objet, nous constatons que ce nouvel objet aura deux étiquettes 1, deux étiquettes 2, et deux étiquettes 3. Or pour en faire un véritable objet étiqueté, il faudrait que les étiquettes varient de 1 à $3+4=7$. Pour cela, il faut redistribuer les étiquettes, mais cette redistribution doit respecter l'ordre initial : par exemple, dans l'objet (E), le nœud ayant la plus petite étiquette (initialement 1) devra pointer vers le nœud ayant l'étiquette intermédiaire (initialement 2), qui devra lui-même pointer vers le nœud ayant la plus grande étiquette (initialement 3) ; dans (F), les deux étiquettes les plus petites devront être à distance 2. Cette redistribution s'appelle un *réétiquetage*, et permet de définir la notion naturelle de produit sur des objets étiquetés, le *produit partitionnel* :

**Définition 14.** *Soient $\alpha$ et $\beta$ deux objets étiquetés. Un* réétiquetage *de $(\alpha, \beta)$ est une affectation d'étiquettes (distinctes et totalement ordonnées) sur l'ensemble des atomes de $\alpha$ et $\beta$. Un réétiquetage est dit* compatible *s'il respecte les ordres initiaux sur $\alpha$ et $\beta$. Le* produit partitionnel *de $\{\alpha\}$ par $\{\beta\}$, noté $\{\alpha\} \times_p \{\beta\}$, est l'ensemble des réétiquetages compatibles de $(\alpha, \beta)$.*

Par exemple, dès que l'on a choisi les étiquettes de (E) parmi les entiers de 1 à 7, il y a une seule façon de respecter les ordres initiaux sur (E) et (F). Si l'on choisit 2, 3 et 7, le seul réétiquetage compatible est :




L'un des $\binom{7}{3} = 35$ objets de $\{E\} \times_p \{F\}$.

Le nombre de réétiquetages compatibles d'une paire $(\alpha, \beta)$ est donc $\frac{(i+j)!}{i!\,j!}$ où $i$ est la taille de $\alpha$ et $j$ celle de $\beta$. Le produit partitionnel de deux ensembles est la réunion des produits partitionnels élément à élément :

$$\mathcal{A} \times_p \mathcal{B} = \bigcup_{a \in \mathcal{A}, b \in \mathcal{B}} \{a\} \times_p \{b\}$$

et nous le noterons $product(A, B)$ (nous pouvons utiliser le même symbole que pour le produit cartésien car on se place toujours dans un seul univers à la fois, cf la remarque ci-dessus).

CONSTRUCTEUR COMPLEXE PARTITIONNEL : le constructeur *complexe partitionnel* permet de former des suites d'éléments de même type, comme le constructeur "séquence" en univers non étiqueté. Il se définit également par union et produit partitionnel : d'abord, le produit partitionnel construit des $k$-uplets d'objets

$$\mathcal{A}^{<k>} = \underbrace{\mathcal{A} \times_p \cdots \times_p \mathcal{A}}_{k},$$

puis le constructeur *complexe partitionnel* des séquences de longueur quelconque

$$\mathcal{A}^{<*>} = \mathcal{A}^{<0>} \cup \mathcal{A}^{<1>} \cup \mathcal{A}^{<2>} \cup \cdots \cup \mathcal{A}^{<k>} \cup \cdots \tag{3.1}$$

La notation Adl de la construction $\mathcal{A}^{<*>}$ est $sequence(A)$, et les séquences seront désignées à l'aide de parenthèses, par exemple () représente la séquence vide.

CONSTRUCTEUR COMPLEXE PARTITIONNEL ABÉLIEN : le constructeur *complexe partitionnel abélien* est l'équivalent du constructeur *ensemble* de l'univers non étiqueté. Les ensembles de $k$ objets sont

$$\mathcal{A}^{[k]} = \left\{ \{a_1, \ldots, a_k\} \;\middle|\; (a_1, \ldots, a_k) \in \mathcal{A}^{<k>} \right\}$$

et le complexe partitionnel abélien de $\mathcal{A}$ est

$$\mathcal{A}^{[*]} = \mathcal{A}^{[0]} \cup \mathcal{A}^{[1]} \cup \mathcal{A}^{[2]} \cup \cdots \cup \mathcal{A}^{[k]} \cup \cdots$$

En univers étiqueté, la notion de multi-ensemble n'existe pas. En effet, deux éléments d'un ensemble sont toujours différents, car même si leur structure sous-jacente est identique, au moins leurs étiquettes différent (puisque les étiquettes d'un même objet sont toutes distinctes). Nous noterons la construction de complexe partitionnel abélien $set(A)$.

CONSTRUCTEUR CYCLE ORIENTÉ : le constructeur cycle orienté se définit aussi à l'aide du complexe partitionnel :

$$\mathcal{C}_p(\mathcal{A}) = \left\{ [a_1, \ldots, a_k] \;\middle|\; k \geq 1 \text{ et } (a_1, \ldots, a_k) \in \mathcal{A}^{<*>} \right\}$$

et se note $cycle(A)$.

CONSTRUCTEUR CYCLE NON ORIENTÉ : en univers étiqueté, nous définissons des cycles *non orientés*, qui seront notés à l'aide de doubles crochets. Deux cycles non orientés $[\![a_1, \ldots, a_k]\!]$ et $[\![b_1, \ldots, b_k]\!]$ sont égaux si on peut les superposer par rotation ou retournement, c'est-à-dire que $[a_1, \ldots, a_k] = [b_1, \ldots, b_k]$, ou bien $[a_1, \ldots, a_k] = [b_k, \ldots, b_1]$. Le constructeur cycle non orienté est noté $\mathcal{UC}_p$ et *ucycle* en Adl :

$$\mathcal{UC}_p(\mathcal{A}) = \left\{ [\![a_1, \ldots, a_k]\!] \;\middle|\; k \geq 1 \text{ et } (a_1, \ldots, a_k) \in \mathcal{A}^{<*>} \right\}.$$

Lorsque le nombre $k$ d'éléments du cycle vaut 1 ou 2, un cycle non orienté génère un seul cycle orienté, car le retournement ne modifie pas le cycle : quand on retourne $[a]$, on obtient $[a]$, et quand on retourne $[a, b]$, on obtient $[b, a]$, c'est-à-dire $[a, b]$ par permutation circulaire. Par contre, pour $k \geq 3$, les cycles $[a_1, \ldots, a_k]$ et $[a_k, \ldots, a_1]$ sont toujours différents à cause des étiquettes : $a_1$ ne peut être superposé qu'avec lui-même, et alors $a_2$ et $a_k$ se superposent, mais $a_2 \neq a_k$ car $k > 2$.

**Définition 15.** *La classe $\hat{\Omega}$ est l'ensemble des spécifications en univers étiqueté dont les productions utilisent les constructeurs union, produit partitionnel, complexe partitionnel, complexe partitionnel abélien, cycle orienté et non orienté.*

Le tableau ci-dessous résume les constructions de $\hat{\Omega}$ et leur notation en Adl.

| constructeur | notation mathématique | syntaxe Adl |
|---|---|---|
| union | $\mid$ | $\mid$ |
| produit partitionnel | $\times_p$ | *product* |
| complexe partitionnel | $(\cdot)^{<*>}$ | *sequence* |
| complexe partitionnel abélien | $(\cdot)^{[*]}$ | *set* |
| cycle orienté | $\mathcal{C}_p$ | *cycle* |
| cycle non orienté | $\mathcal{UC}_p$ | *ucycle* |
| atome étiqueté de taille $k$ | $\lvert\cdot\rvert = k$ | $Latom(k)$ |

REMARQUE : La construction $Latom(k)$ définit un atome de taille $k$ portant $k$ étiquettes non ordonnées entre elles. Cette déclaration est donc équivalente à $set(Latom(1), card = k)$, qu'il est préférable d'utiliser pour une meilleure compréhension.

## 3.2 Analyse des structures de données

Dans la section précédente, nous avons défini une classe $\hat{\Omega}$ de spécifications d'objets étiquetés. Les constructeurs de $\hat{\Omega}$ étant similaires à ceux définis en univers non étiqueté, les algorithmes de décision du chapitre 1 sont valables ici également, et le caractère bien fondé est décidable dans $\hat{\Omega}$.

Ceci nous permet de passer directement à la partie plus intéressante, l'analyse algébrique des spécifications, afin de pouvoir compter les structures de données. Dans cette section, nous introduisons en premier lieu la notion de série génératrice *exponentielle*, mieux adaptée au dénombrement des objets étiquetés que la série ordinaire. Puis nous montrons que, tout comme en univers non étiqueté, les constructeurs de la classe $\hat{\Omega}$ se traduisent en opérateurs sur les séries génératrices. Ces opérateurs ont cependant une forme plus simple, notamment pour les constructeurs complexe partitionnel abélien (analogue du constructeur ensemble) et cycle orienté. A la fin de cette section sont mentionnés quelques exemples d'analyse de structures de données, avant d'analyser des programmes manipulant des objets étiquetés (section 3.3).

**Définition 16.** *(Série génératrice exponentielle) La série génératrice exponentielle associée à un ensemble $\mathcal{A}$ est*

$$\hat{A}(z) = \sum_{a \in \mathcal{A}} \frac{z^{|a|}}{|a|!} = \sum_{n=0}^{\infty} A_n \frac{z^n}{n!}$$

*où $A_n$ est le nombre d'objets de taille $n$ de $\mathcal{A}$.*

Dans ce chapitre, toutes les séries génératrices seront de type exponentiel ; nous convenons donc d'omettre l'accent circonflexe, et de noter simplement $A(z)$ au lieu de $\hat{A}(z)$. Les règles de traduction des constructeurs vers les séries génératrices exponentielles sont les suivantes.

**Règle 22.** *(Atome étiqueté)*

$$\frac{\textbf{type } A = Latom(k);}{A(z) = \frac{z^k}{k!}}$$

**Démonstration :** La déclaration $Latom(k)$ génère un seul objet, de taille $k$, donc la série exponentielle associée est $z^k/k!$. ■

Le constructeur *union* se traduit en l'opérateur somme sur les séries génératrices.

**Règle 23.** *(Union)*

$$\frac{\textbf{type } A = B \mid C;}{A(z) = B(z) + C(z)}$$

**Démonstration :** $A(z) = \sum_{a \in (\mathcal{B} \cup \mathcal{C})} \frac{z^{|a|}}{|a|!} = \sum_{a \in \mathcal{B}} \frac{z^{|a|}}{|a|!} + \sum_{a \in \mathcal{C}} \frac{z^{|a|}}{|a|!} = B(z) + C(z)$. ■

**Règle 24.** *(Produit partitionnel)*

$$\frac{\textbf{type } A = product(B, C);}{A(z) = B(z)C(z)}$$

**Démonstration :** Si $b$ est de taille $k$, et $c$ de taille $l$, le nombre de réétiquetages compatibles de la paire $(b, c)$ est $\binom{k+l}{k}$ car il suffit de choisir les $k$ étiquettes de $b$ parmi $1 \ldots k+l$ :

$$A(z) = \sum_{b \in \mathcal{B}, c \in \mathcal{C}} (\{b\} \times_p \{c\})(z) = \sum_{b \in \mathcal{B}, c \in \mathcal{C}} \binom{|b| + |c|}{|b|} \frac{z^{|b|+|c|}}{(|b| + |c|)!} = \sum_{b \in \mathcal{B}} \frac{z^{|b|}}{|b|!} \sum_{c \in \mathcal{C}} \frac{z^{|c|}}{|c|!} = B(z)C(z).$$

Ce qui fait que "ça marche" est le fait que le nombre de réétiquetages compatibles, $\binom{|b|+|c|}{|b|}$, compense le terme $(|b| + |c|)!$ du dénominateur, et alors les sommations se séparent. ■

**Règle 25.** *(Complexe partitionnel)*

$$\frac{\textbf{type } A = sequence(B);}{A(z) = \frac{1}{1 - B(z)}}$$

**Démonstration :** L'équation (3.1) définissant le complexe partitionnel se traduit grâce à la règle de l'union (23) en

$$A(z) = (\mathcal{B}^{<0>})(z) + (\mathcal{B}^{<1>})(z) + (\mathcal{B}^{<2>})(z) + \cdots + (\mathcal{B}^{<k>})(z) + \cdots$$

où $(\mathcal{B}^{<k>})(z)$ signifie "la série exponentielle associée à $\mathcal{B}^{<k>}$". Or par la règle du produit partitionnel (24), $(\mathcal{B}^{<k>})(z) = (B(z))^k$, d'où

$$A(z) = (B(z))^0 + (B(z))^1 + \cdots + (B(z))^k + \cdots = \frac{1}{1 - B(z)}.$$ ■

**Règle 26.** *(Cycle orienté)*

$$\frac{\textbf{type } A = cycle(B);}{A(z) = \log \frac{1}{1-B(z)}}$$

**Démonstration :** Chaque cycle de $k$ objets engendre $k$ séquences distinctes (du fait des étiquettes, les objets d'un même cycle sont tous différents). Par conséquent, le nombre de cycles de $k$ objets de $\mathcal{B}$ est $B(z)^k/k$, et

$$A(z) = \sum_{k\geq 1} \frac{1}{k} B(z)^k = \log \frac{1}{1 - B(z)}.$$ ■

**Règle 27.** *(Cycle non orienté)*

$$\frac{\textbf{type } A = ucycle(B);}{A(z) = \frac{1}{2}\log \frac{1}{1-B(z)} + \frac{1}{2}B(z) + \frac{1}{4}B(z)^2}$$

**Démonstration :** Les cycles non orientés de $k$ éléments sont au même nombre que les cycles orientés pour $k = 1$ et $k = 2$, et sont moitié moins pour $k \geq 3$, d'où

$$A(z) = B(z) + \frac{1}{2}B(z)^2 + \sum_{k\geq 3} \frac{1}{2k} B(z)^k.$$ ■

**Règle 28.** *(Complexe partitionnel abélien)*

$$\frac{\textbf{type } A = set(B);}{A(z) = \exp(B(z))}$$

**Démonstration :** Chaque ensemble de $k$ objets engendre $k!$ séquences distinctes, et par suite

$$A(z) = \sum_{k\geq 0} \frac{B^k(z)}{k!} = \exp(B(z)).$$ ■

Cette règle de dénombrement des complexes partitionnels abéliens est usuellement appelée "*exponential formula*" (formule exponentielle) [BG71, Foa74, Sta78].

EXEMPLE 21 : (Décomposition des permutations en cycles) La déclaration $x = Latom(1)$ définit l'atome étiqueté $x_1$ de taille 1 (en convenant de noter les étiquettes en indice). Alors $C = cycle(x)$ définit des cycles d'atomes étiquetés, comme par exemple $[x_4, x_2, x_1, x_3]$. Par conséquent, la spécification

**type** P = *set*(C);

C = *cycle*(x);
x = *Latom*(1);

définit l'ensemble $P$ des cycles d'atomes étiquetés. Un élément de $P$ de taille $n$ est alors en bijection avec la permutation des entiers de 1 à $n$ que l'on obtient en ne gardant que les étiquettes. Par exemple, l'élément $\{[x_1],[x_3,x_5],[x_2,x_6,x_4]\}$ correspond à la permutation $(1\to 1, 2\to 6, 3\to 5, 4\to 2, 5\to 3, 6\to 4)$ dont la décomposition en permutations circulaires est $\{[1],[3,5],[2,6,4]\}$.

L'analyse de la spécification ci-dessus s'effectue à l'aide des règles 22, 26 et 28, et conduit au système d'équations :

$$P(z)=\exp(C(z)),\quad C(z)=\log\frac{1}{1-x(z)},\quad x(z)=z.$$

Ce système est triangulaire (nous disons dans ce cas que la spécification est *explicite*), et nous trouvons immédiatement $P(z)=1/(1-z)$. Le nombre d'objets de taille $n$ dérivés de `P` est le coefficient de $z^n$ dans $P(z)$ *multiplié par* $n!$, puisque dans la série génératrice exponentielle, le nombre d'objets de taille $n$ est multiplié par $z^n/n!$ (voir la définition 16). Nous obtenons ainsi $n![z^n]1/(1-z)=n!$, et nous retrouvons (heureusement) le nombre de permutations de $n$ éléments distincts. □

EXEMPLE 22 : (Graphes 2-réguliers) Au cours de l'introduction de ce chapitre, nous avons donné une spécification des graphes 2-réguliers étiquetés :

**type** tworegg = *set*(component);
component = *ucycle*(node, *card*>=3);
node = *Latom*(1);

A l'aide des règles énoncées dans cette section, cette spécification se traduit en équations pour les séries exponentielles $T(z)$ et $C(z)$ associées respectivement aux graphes 2-réguliers (`tworegg`) et aux graphes connexes circulaires (`component`) :

$$T(z)=\exp(C(z)),\quad C(z)=\frac{1}{2}\log\frac{1}{1-z}-\frac{z}{2}-\frac{z^2}{4} \tag{3.2}$$

(pour obtenir l'équation de $C(z)$, nous avons retranché dans la règle 27 les termes correspondant aux cycles de longueur 1 et 2, c'est-à-dire $B(z)$ et $B(z)^2/2$). Nous avons à nouveau affaire à une spécification explicite, et la série génératrice exponentielle des graphes 2-réguliers est :

$$T(z)=\frac{e^{-z/2-z^2/4}}{\sqrt{1-z}}=1+\frac{z^3}{3!}+\frac{3z^4}{4!}+\frac{12z^5}{5!}+\frac{70z^6}{6!}+\frac{465z^7}{7!}+\frac{3507z^8}{8!}+\frac{30016z^9}{9!}+O(z^{10}).$$

L'analyse asymptotique de $T(z)$ est facile, une fois que l'on a remarqué que sa singularité dominante se trouve en $z=1$, et que le numérateur est une fonction analytique en ce point, d'où :

$$[z^n]T(z)\sim e^{-3/4}[z^n]\frac{1}{\sqrt{1-z}}=\frac{e^{-3/4}}{\sqrt{\pi n}}+O(\frac{1}{n}).$$

Pour obtenir le nombre $G_n^{2r}$ de graphes 2-réguliers ayant $n$ nœuds, il suffit de multiplier par $n!$ le coefficient de $z^n$ dans $T(z)$ : $G_3^{2r}=1$, $G_4^{2r}=3$, $G_5^{2r}=12$, $G_6^{2r}=70$, $G_7^{2r}=465$, $G_8^{2r}=3507$. □

EXEMPLE 23 : (Trains aléatoires) Voici un autre exemple de spécification explicite, mais plus complexe, inventée par Flajolet [Fla85].

Un train aléatoire (`train`) est formé d'une locomotive (`locomotive`) et d'une séquence de wagons (`wagon`). La locomotive est une séquence non vide de tranches (`slice`). Chaque tranche est constituée d'une partie de toit (`upper`), d'une partie de plancher (`lower`) et éventuellement d'un essieu (`wheel`), qui comprend un axe (`center`) et un cycle d'éléments de roue (`wheel_element`). Les wagons ressemblent à la locomotive, sauf qu'ils contiennent en plus un ensemble de passagers (`passenger`). Chaque passager a une tête (`head`) et un corps (`belly`) qui sont chacun des cycles d'"éléments de passager" (`passenger_element`).

Tout ceci s'exprime par la spécification suivante :

**type** train = *product*(locomotive,wagons);
wagons = *sequence*(wagon);
locomotive = *sequence*(slice, *card*>=1);
slice = *product*(upper,lower)
| *product*(upper,lower,wheel);
wheel = *product*(center,*cycle*(wheel_element));
wagon = *product*(locomotive,passengers);
passengers = *set*(passenger);
passenger = *product*(head,belly);
head, belly = *cycle*(passenger_element);
upper, lower, center, wheel_element, passenger_element = *Latom*(1);

et la série génératrice exponentielle des trains aléatoires a une forme explicite :

$$\begin{aligned}\mathrm{Train}(z) &= \frac{z^2+z^3\log\frac{1}{1-z}}{(1-z^2-z^3\log\frac{1}{1-z})\left(1-\frac{(z^2+z^3\log\frac{1}{1-z})e^{\log^2\frac{1}{1-z}}}{1-z^2-z^3\log\frac{1}{1-z}}\right)}\\ &= \frac{2z^2}{2!}+\frac{72z^4}{4!}+\frac{60z^5}{5!}+\frac{6720z^6}{6!}+\frac{16380z^7}{7!}+\frac{1247904z^8}{8!}+\frac{6531840z^9}{9!}+O(z^{10}).\end{aligned}$$

L'analyse asymptotique de cette fonction montre que le nombre de trains aléatoires de taille $n$ est équivalent à $CA^n n!$ lorsque $n$ tend vers l'infini, où $A \simeq 1.93029807$ et $C \simeq 0.100855759$ (voir le *CookBook* [FSZ89a], pages 78 à 87, pour une analyse détaillée). □

EXEMPLE 24 : (Arbres généraux non planaires) Les arbres enracinés non planaires dont les nœuds sont étiquetés sont d'une importance considérable en théorie des graphes. Ce sont par exemple les constituants de base des *graphes fonctionnels*, qui représentent les fonctions de $[1\ldots n]$ dans lui-même. Ils permettent aussi de définir les arbres non enracinés, ou graphes connexes acycliques (appelés aussi arbres de *Cayley*).

Un arbre général non planaire (ou simplement arbre) est soit réduit à un nœud, soit c'est un nœud auquel sont rattachés d'autres arbres, non ordonnés entre eux, qui constituent en fait un ensemble, ou plus exactement un *complexe partitionnel abélien*. Le cas du nœud isolé n'est en réalité que le cas particulier où l'ensemble est vide, d'où la spécification

**type** tree = *product*(node,*set*(tree));
node = *Latom*(1);

Cette spécification est *implicite*, c'est-à-dire récursive, et conduit à une équation *récursive* pour la série $T(z)$ des arbres généraux non planaires.

$$T(z) = ze^{T(z)} = \frac{z}{1!}+\frac{2z^2}{2!}+\frac{9z^3}{3!}+\frac{64z^4}{4!}+\frac{625z^5}{5!}+\frac{7776z^6}{6!}+\frac{117649z^7}{7!}+\frac{2097152z^8}{8!}+O(z^9).$$

Cayley a montré que le nombre d'arbres généraux enracinés de taille $n$ est $n^{n-1}$ ; nous pouvons le vérifier sur les premiers termes indiqués ci-dessus : $2^1 = 2$, $3^2 = 9$, $4^3 = 64$, ... □

## 3.3 Analyse des programmes

En univers étiqueté, nous pouvons distinguer deux types de programmes, suivant que l'ordre induit par les étiquettes influe ou non sur l'exécution. Par exemple, un programme de tri utilisera les étiquettes, et son exécution dépend des valeurs de celles-ci. Par contre, un programme calculant la longueur de cheminement d'un arbre étiqueté utilise uniquement la structure non étiquetée sous-jacente, et son exécution a même coût pour deux arbres différant seulement par les étiquettes. Dans cette section, nous étudions des schémas indépendants de l'ordre induit par les étiquettes ; ces schémas sont l'équivalent en univers étiqueté des schémas de sélection et d'itération de la classe II. La section suivante s'intéresse aux schémas particuliers à l'univers étiqueté.

Pour compter le coût d'un programme opérant sur des données étiquetées, nous employons un descripteur de complexité *exponentiel*.

**Définition 17.** *Le* descripteur de complexité exponentiel *associé à une procédure $P$ opérant sur un ensemble étiqueté $\mathcal{A}$ est la série génératrice exponentielle*

$$\widehat{\tau P}^{\mathcal{A}}(z) = \sum_{a \in \mathcal{A}} \widehat{\tau P}\{a\} \frac{z^{|a|}}{|a|!} = \sum_{n=0}^{\infty} \widehat{\tau P}_n^{\mathcal{A}} \frac{z^n}{n!}$$

*où $\widehat{\tau P}\{a\}$ est le coût de l'évaluation de $P$ sur $a$, et $\widehat{\tau P}_n^{\mathcal{A}}$ le coût cumulé sur tous les objets de $\mathcal{A}$ de taille $n$. Un schéma de programmation* **procedure** $P(a : A)$; **begin** ... **end** *est dit* admissible *si la suite $(\widehat{\tau P}_n^{\mathcal{A}})$ ne dépend que des suites $(\widehat{\tau Q}_n^{\mathcal{B}})$ associées aux procédures $Q$ se trouvant dans le corps de $P$, de la suite $(\widehat{A}_n)$ et des suites $(\widehat{C}_n)$ associées aux composantes des objets de $\mathcal{A}$.*

Comme en univers non étiqueté, nous écrirons simplement $\widehat{\tau P}$ au lieu de $\widehat{\tau P}^{\mathcal{A}}$ lorsqu'il n'y a pas d'ambiguïté sur l'ensemble des données de $P$. D'autre part, comme dans la suite de ce chapitre, toutes les séries (séries de dénombrement et descripteurs de complexité) seront de type exponentiel, nous convenons aussi d'omettre l'accent circonflexe et de noter $\tau P(z)$ le descripteur de complexité exponentiel d'une procédure $P$.

Les constructeurs étant les mêmes qu'en univers étiqueté, et les schémas de sélection et d'itération sur ces constructeurs ayant déjà été définis au chapitre 1 (section 1.5), nous nous contentons de donner ici les règles de traduction associées à ces schémas.

**Règle 29.** *(Instruction élémentaire)*

$$\frac{\textbf{procedure } P(a : A); \textbf{ begin } \text{count } \textbf{end;} \qquad \textbf{measure } \text{count} : \mu;}{\tau P(z) = \mu \cdot A(z)}$$

**Démonstration :** $\tau P(z) = \sum_{a \in \mathcal{A}} \mu \frac{z^{|a|}}{|a|!} = \mu\, A(z)$. ■

**Règle 30.** *(Exécution séquentielle)*

$$\frac{\textbf{procedure } P(a:A);\ \textbf{begin } Q(a); R(a)\ \textbf{end;}}{\tau P(z) = \tau Q(z) + \tau R(z)}$$

**Démonstration :** $\tau P(z) = \sum_{a\in\mathcal{A}} (\tau Q\{a\} + \tau R\{a\}) \frac{z^{|a|}}{|a|!} = \tau Q(z) + \tau R(z)$. ∎

**Règle 31.** *(Sélection dans une union)*

$$\frac{\textbf{type } A = B \mid C; \qquad \textbf{procedure } P(a:A);\ \textbf{casetype } a\ \textbf{of } B: Q(a);\ C: R(a)\ \textbf{end;}}{\tau P^{\mathcal{A}}(z) = \tau Q^{\mathcal{B}}(z) + \tau R^{\mathcal{C}}(z)}$$

**Démonstration :** $\sum_{\mathcal{B}\cup\mathcal{C}} \tau P\{a\} \frac{z^{|a|}}{|a|!} = \sum_{a\in\mathcal{B}} \tau Q\{a\} \frac{z^{|a|}}{|a|!} + \sum_{a\in\mathcal{C}} \tau R\{a\} \frac{z^{|a|}}{|a|!} = \tau Q^{\mathcal{B}}(z) + \tau R^{\mathcal{C}}(z)$. ∎

**Règle 32.** *(Sélection dans un produit partitionnel)*

$$\frac{\textbf{type } A = \mathit{product}(B,C); \qquad \textbf{procedure } P(a:A);\ \textbf{case } a\ \textbf{of } (b,c): Q(b)\ \textbf{end;}}{\tau P^{\mathcal{A}}(z) = \tau Q^{\mathcal{B}}(z) C(z)}$$

**Démonstration :** Il faut détailler cette preuve pour bien comprendre le mécanisme de réétiquetage :

$$\begin{aligned}
\tau P^{\mathcal{A}}(z) &= \sum_{(b',c')\in\mathcal{B}\times_p\mathcal{C}} \tau P\{(b',c')\} \frac{z^{|(b',c')|}}{|(b',c')|!} && (3.3)\\
&= \sum_{b\in\mathcal{B},c\in\mathcal{C}} \ \sum_{(b',c')\in\{b\}\times_p\{c\}} \tau Q\{b'\} \frac{z^{|b'|+|c'|}}{(|b'|+|c'|)!} && (3.4)\\
&= \sum_{b\in\mathcal{B},c\in\mathcal{C}} \ \sum_{(b',c')\in\{b\}\times_p\{c\}} \tau Q\{b\} \frac{z^{|b|+|c|}}{(|b|+|c|)!} && (3.5)\\
&= \sum_{b\in\mathcal{B}} \tau Q\{b\} \sum_{c\in\mathcal{C}} \binom{|b|+|c|}{|b|} \frac{z^{|b|+|c|}}{(|b|+|c|)!} = \tau Q^{\mathcal{B}}(z) C(z). && (3.6)
\end{aligned}$$

Le passage de (3.3) à (3.4) utilise trois faits : $\mathcal{B}\times_p\mathcal{C}$ est la réunion des $\{b\}\times_p\{c\}$, le coût de $P((b',c'))$ égale le coût de $Q(b')$, et la taille d'un produit partitionnel est la somme des tailles des composantes. Pour passer de (3.4) à (3.5), on a utilisé le fait que le coût de l'évaluation d'une procédure sur un objet étiqueté ne dépend pas des *valeurs* des étiquettes, mais uniquement de l'ordre total qu'elles induisent, et aussi l'égalité des tailles de $b$ et $b'$, et de $c$ et $c'$. Enfin, cela permet de sortir le facteur $\tau Q\{b\}$ de la sommation sur $b'$ et $c'$, ce qui fait apparaître dans (3.6) le nombre de réétiquetages compatibles, lequel compense le terme $(|b|+|c|)!$ du dénominateur. ∎

**Règle 33.** *(Sélection dans un complexe partitionnel)*

$$\frac{\textbf{type } A = \mathit{sequence}(B); \qquad \textbf{procedure } P(a:A);\ \textbf{forone } b\ \textbf{in } a\ \textbf{do } Q(b);}{\tau P(z) = \tau Q(z)/(1-B(z))}$$

**Démonstration :** L'idée de la preuve est de construire un schéma équivalent en coût à l'aide des schémas de sélection dans une union et dans un produit partitionnel, en utilisant le fait que le constructeur complexe partitionnel s'exprime (à isomorphisme près) à l'aide des constructeurs union et produit partitionnel. Ce mécanisme a déjà été utilisé dans la preuve de la règle de sélection dans une séquence en univers non étiqueté (règle 12 page 62). ■

**Règle 34.** *(Itération dans un complexe partitionnel)*

$$\frac{\textbf{type } A = \mathit{sequence}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \tau Q(z)/(1-B(z))^2}$$

**Démonstration :** Comme pour la règle de sélection, on établit cette règle en écrivant un programme Adl de même coût à l'aide des constructeurs *union* et *produit partitionnel* (voir par exemple la preuve de la règle 13 page 63). ■

**Règle 35.** *(Sélection dans un complexe partitionnel abélien)*

$$\frac{\textbf{type } A = \mathit{set}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forone } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \frac{\exp(B(z))-1}{B(z)}\tau Q(z)}$$

**Démonstration :** Soit $B^{[k]}$ la série associée aux ensembles de $k$ objets, et $B^{<k>}$ la série associée aux séquences de $k$ objets : nous avons déjà vu que $B^{<k>} = k!B^{[k]}$. Or la sélection dans $\mathcal{B}^{<k>}$ au lieu de $\mathcal{B}^{[k]}$ donnerait comme descripteur de complexité

$$\tau P^{\mathcal{B}^{<k>}}(z) = B^{k-1}(z)\tau Q(z)$$

d'après la règle 32 appliquée $k-1$ fois. De plus, l'évaluation de $P$ sur une séquence ou sur l'ensemble associé a même coût (en moyenne sur le choix de l'élément), d'où

$$\tau P^{\mathcal{B}^{<k>}}(z) = k!\,\tau P^{\mathcal{B}^{[k]}}(z)$$

et

$$\tau P^{\mathcal{B}^{[*]}}(z) = \sum_{k\geq 1} \frac{B^{k-1}(z)}{k!}\tau Q(z) = (\exp(B(z)) - 1)/B(z)\ \tau Q(z).$$ ■

**Règle 36.** *(Itération dans un complexe partitionnel abélien)*

$$\frac{\textbf{type } A = \mathit{set}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \exp(B(z))\ \tau Q(z)}$$

**Démonstration :** Comme ci-dessus, la sélection sur une séquence ou sur l'ensemble associé a même coût :

$$\tau P^{\mathcal{B}^{[k]}}(z) = \frac{1}{k!}\tau P^{\mathcal{B}^{<k>}}(z)$$

et $\tau P^{\mathcal{B}^{<k>}}(z) = kB^{k-1}(z)\tau Q(z)$ par les règles 30 et 32 ; par suite

$$\tau P^{\mathcal{B}^{[*]}}(z) = \sum_{k\geq 1} \frac{1}{(k-1)!}B^{k-1}(z)\tau Q(z) = \exp(B(z))\ \tau Q(z).$$ ■

**Règle 37.** *(Sélection dans un cycle orienté)*

$$\frac{\textbf{type } A = \mathit{cycle}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forone } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \frac{1}{B(z)} \log \frac{1}{1-B(z)} \tau Q(z)}$$

**Démonstration :** Soit $\mathcal{C}_k$ l'ensemble des cycles formés de $k$ objets de $\mathcal{B}$ :

$$C_k(z) = \frac{1}{k} B^{<k>}(z)$$

et le coût (moyen) de la sélection est le même sur un cycle que sur chacune des $k$ séquences associées, d'où

$$\tau P(z) = \sum_{k\geq 1} \tau P^{C_k}(z) = \sum_{k\geq 1} \frac{1}{k} \tau P^{B^{<k>}}(z) = \sum_{k\geq 1} \frac{1}{k} B^{k-1}(z) \tau Q(z) = \log \frac{1}{1-B(z)} \frac{\tau Q(z)}{B(z)}. \qquad \blacksquare$$

**Règle 38.** *(Sélection dans un cycle non orienté)*

$$\frac{\textbf{type } A = \mathit{ucycle}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forone } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \frac{\tau Q(z)}{2} [\frac{1}{B(z)} \log \frac{1}{1-B(z)} + 1 + \frac{B(z)}{2}]}$$

**Démonstration :** Pour $k \leq 2$, le coût est le même que pour les cycles orientés, et il est diminué de moitié pour $k \geq 3$ :

$$\tau P(z) = \tau Q(z) + \frac{1}{2} B(z) \tau Q(z) + \sum_{k\geq 3} \frac{1}{2k} B(z)^{k-1} \tau Q(z). \qquad \blacksquare$$

**Règle 39.** *(Itération dans un cycle orienté)*

$$\frac{\textbf{type } A = \mathit{cycle}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \frac{1}{1-B(z)} \tau Q(z)}$$

**Démonstration :**

$$\tau P(z) = \sum_{k\geq 1} \tau P^{C_k}(z) = \sum_{k\geq 1} \frac{1}{k} \tau P^{B^{<k>}}(z) = \sum_{k\geq 1} \frac{1}{k} k B^{k-1}(z) \tau Q(z) = \frac{1}{1-B(z)} \tau Q(z). \qquad \blacksquare$$

**Règle 40.** *(Itération dans un cycle non orienté)*

$$\frac{\textbf{type } A = \mathit{ucycle}(B); \qquad \textbf{procedure } P(a:A); \textbf{ forall } b \textbf{ in } a \textbf{ do } Q(b);}{\tau P(z) = \frac{\tau Q(z)}{2} [\frac{1}{1-B(z)} + 1 + B(z)]}$$

**Démonstration :**

$$\tau P(z) = \tau Q(z) + B(z) \tau Q(z) + \sum_{k\geq 3} \frac{1}{2} B(z)^{k-1} \tau Q(z). \qquad \blacksquare$$

Ces règles montrent d'une part qu'il existe une classe bien définie de programmes se prêtant à une analyse automatique, et d'autre part que la classe des équations générées par cette analyse est caractérisée très précisément :

**Théorème 7.** *Soit* $\hat{\Pi}$ *l'ensemble des programmes dont les données sont définies par une spécification de* $\hat{\Omega}$*, et qui utilisent des instructions élémentaires, l'exécution séquentielle et les schémas de sélection et d'itération dans une union, un produit ou un complexe partitionnel, un complexe partitionnel abélien, un cycle orienté ou non orienté.*

*Tout programme de* $\hat{\Pi}$ *se traduit en deux systèmes d'équations, un système* $(A)$ *pour les séries génératrices exponentielles de dénombrement des types, et un système* $(B)$ *pour les descripteurs de complexité exponentiels des procédures.*

*Les équations du système* $(A)$ *sont formées à partir de* $1$ *et* $z^k/k!$ *par application des opérateurs* $+$, $\times$, $Q(f) = \frac{1}{1-f}$, $E(f) = \exp(f)$, $L(f) = \log\frac{1}{1-f}$ *et* $L'(f) = \frac{1}{2}\log\frac{1}{1-f} + \frac{f}{2} + \frac{f^2}{4}$*. Les équations du système* $(B)$ *sont de la forme* $\tau = \mu f$, $\tau = \tau' + \tau''$, $\tau = f\tau'$, $\tau = \Theta(f)\tau'$*, où* $\tau$ *est un descripteur de complexité,* $\mu$ *une constante,* $f$ *une solution de* $(A)$*, et* $\Theta$ *l'un des opérateurs* $E^*(f) = \frac{\exp(f)-1}{f}$, $L^*(f) = \frac{1}{f}\log\frac{1}{1-f}$, $L'_s(f) = \frac{1}{2f}\log\frac{1}{1-f} + \frac{1}{2} + \frac{f}{4}$ *et* $L'_i(f) = \frac{1}{2(1-f)} + \frac{1}{2} + \frac{f}{2}$.

Nous appelons $\mathcal{LR}$ (pour *Labelled Recursive*) la classe des solutions d'un système de type $(A)$.

**Démonstration :** Pour le système $(A)$, la preuve découle des règles 22 à 28 : les déclarations atomiques conduisent à des membres droits du genre $z^k/k!$, l'union à $f + f'$, le produit partitionnel à $f \times f'$, le complexe partitionnel à $Q(f)$, le complexe partitionnel abélien à $E(f)$, les cycles orientés à $L(f)$ et les cycles non orientés à $L'(f)$.

Pour le système $(B)$, les instructions de coût constant (règle 29) donnent une équation du type $\tau = \mu f$ ; l'exécution séquentielle et la sélection dans une union (règles 30 et 31) conduisent à $\tau = \tau' + \tau''$ ; la sélection dans un produit partitionnel (règle 32) donne $\tau = f\tau'$, ainsi que la sélection et l'itération dans un complexe partitionnel (règles 33 et 34) car $Q(f)$ et $Q(f)^2$ sont dans $\mathcal{LR}$ dès lors que $f$ y est ; la sélection dans un complexe partitionnel abélien, et dans un cycle, orienté ou non (règles 35, 37 et 38) conduit respectivement à $\tau = E^*(f)\tau'$, $\tau = L^*(f)\tau'$ et $\tau = L'_s(f)\tau'$ ; enfin l'itération dans un complexe partitionnel abélien et dans les deux types de cycles (règles 36, 39 et 40) donne $\tau = E(f)\tau'$, $\tau = Q(f)\tau'$ et $\tau = L'_i(f)\tau'$. ■

EXEMPLE 25 : (Nombre de composantes des graphes 2-réguliers) Nous rappelons ici le programme donné au début de ce chapitre, qui détermine le nombre de composantes d'un graphe 2-régulier :

```
procedure visit (t : toworegg);
begin
   forall c in t do
      count
end;

measure count : 1;
```

où un graphe 2-régulier est un ensemble de composantes (toworegg = *set*(component)). En appliquant la règle d'itération sur un complexe partitionnel abélien, et la règle pour les instructions élémentaires, il vient

$$\tau V(z) = \exp(C(z))\,C(z),$$

où $C(z)$ est la série associée aux composantes connexes, calculée à la section précédente (équation (3.2)). Nous avons ainsi prouvé la première partie du théorème automatique 4 (page 4). La seconde partie (analyse asymptotique) est démontrée par la session MAPLE ci-dessous.

```
> read initequiv;
> C := 1/2 * L(z) - z/2 - z^2/4:
```

```
> T := exp(C):            # s.g.e. des graphes 2-re'guliers
> tauV := exp(C) * C:     # d.c.e. du nombre de composantes

> equivalent(tauV);
                              exp(- 3/4) ln(n)        1
                         (1/2 ----------------) + (O(----))
                                   1/2  1/2           1/2
                                  Pi    n             n


> equivalent(T);
                               exp(- 3/4)        1
                              (----------) + (O(----))
                                 1/2  1/2        3/2
                                Pi    n          n
```

En divisant le premier développement asymptotique, de $[z^n]\tau V(z)$, par le second, de $[z^n]T(z)$, on obtient bien $\frac{1}{2}\log n + O(1)$ pour le nombre moyen de composantes. □

Nous sommes maintenant en mesure de prouver le lemme automatique 3. Afin de mettre en évidence le caractère automatique de cette preuve, nous exhibons l'analyse, telle qu'elle a été effectuée par le système $\Lambda\Upsilon\Omega$.

**Démonstration :** (Lemme automatique 3 page 17) Le fichier de nom `unicyclic.adl` contenant le programme Adl calculant la longueur du cycle d'un graphe connexe monocyclique :

```
type c_u_graph = ucycle(tree);        % connected unicyclic graph %
     tree = node set(tree);
     node = Latom(1);

procedure count_trees (g : c_u_graph);
begin
   forall t in g do
      count;
end;

measure count : 1;
```

voici le résultat brut de l'analyse effectuée par $\Lambda\Upsilon\Omega$ en 41 secondes sur un Sun 3/60 :

```
bandol% date; luo V1.4 < in.ml; date
Sat Dec  8 11:51:17 MET 1990
Luo V1.4 Fri Dec 7 19:10:53 MET 1990

Please send bugs or remarks to luo@inria.inria.fr

Initializing maple ...
For help about Lambda-Upsilon-Omega, type help "";;

() : unit

/usr/local/lib/luo/V1.4/Caml/luoinit.ml loaded

#print_program:=false; do_asymp:=false;;
```

```
false : bool

#analyze "unicyclic";;
c_u_graph(z) = 1/2 L(- RootOf(- _Z exp(_Z) - z)) - 1/2 RootOf(- _Z exp(_Z) - z)

                                    2
     + 1/4 RootOf(- _Z exp(_Z) - z)

       tau_count_trees(z) = - 1/2 RootOf(- _Z exp(_Z) - z)

           /                                              1              \
           |1 - RootOf(- _Z exp(_Z) - z) + ----------------------------|
           \                                1 + RootOf(- _Z exp(_Z) - z)/


() : unit

#quit();;
A bientot ...
Sat Dec  8 11:51:58 MET 1990
```

Dans l'expression de la série `c_u_graph(z)`, le symbole `L` représente l'opérateur $L(y) = \log \frac{1}{1-y}$ associé au constructeur cycle orienté (cf théorème 7). D'autre part, l'expression `RootOf(- _Z exp(_Z) - z)` est la notation utilisée par MAPLE pour représenter la fonction $y(z)$ solution de l'équation $-y(z)\exp(y(z)) - z = 0$. Soit la fonction $f(z) = -y(z)$, qui est solution de $f(z)\exp(-f(z)) - z = 0$ ; les expressions brutes calculées par $\Lambda\Upsilon^{\Omega}$ s'expriment en fonction de $f(z)$ :

$$G(z) = \frac{L(f(z))}{2} + \frac{f(z)}{2} + \frac{f(z)^2}{4}, \qquad \tau\mathrm{CT}(z) = \frac{f(z)}{2}\left(1 + f(z) + \frac{1}{1-f(z)}\right)$$

où on a utilisé $G(z)$ pour `c_u_graph(z)` et $\tau\mathrm{CT}(z)$ pour `tau_count_trees(z)`. Le lecteur vérifiera que ces expressions sont identiques à celles indiquées dans le lemme 3. ■

En ce qui concerne l'analyse asymptotique, elle n'est pas entièrement automatique dans l'état actuel du système $\Lambda\Upsilon^{\Omega}$. La raison principale est que MAPLE ne sait pas effectuer des développements de Taylor de fonctions définies implicitement par des "*RootOf*" comme la fonction $f(z)$ ci-dessus. Nous effectuerons donc une partie de l'analyse "à la main".

**Démonstration :** (Lemme semi-automatique 4 page 17) Tout d'abord, nous recherchons la singularité dominante de $f(z)$.

```
% maple
> read initequiv;          # programme de Bruno Salvy
> f := RootOf(_Z=z*exp(_Z)):
> infsing(f,z,0,1);
                                    1
                               [[------], false]
                                 exp(1)
```

La singularité dominante est $\rho = e^{-1}$, et la valeur de $f$ en ce point est finie, en l'occurrence $f(\rho) = 1$. Le comportement des coefficients de $f(z)$ (donc aussi de $G(z)$ et $\tau\mathrm{CT}(z)$) est relié au comportement

local de $f(z)$ au voisinage de $\rho$. L'expression que $y = f(z)$ annule est $P(z, y) = ze^y - y$, dont les dérivées partielles en $(z = \rho, y = 1)$ sont :

$$\frac{\partial P}{\partial z}(\frac{1}{e}, 1) = e, \quad \frac{\partial P}{\partial y}(\frac{1}{e}, 1) = 0, \quad \frac{\partial^2 P}{\partial y^2}(\frac{1}{e}, 1) = 1.$$

Par conséquent, au voisinage de $z = e^{-1}$ :

$$e(z - \frac{1}{e}) + \frac{1}{2}(y - 1)^2 = O((z - \frac{1}{e})(y - 1)),$$

et le développement local de $y = f(z)$ est

$$f(z) = 1 - \sqrt{2(1 - ez)} + O(1 - ez).$$

Il suffit ensuite de remplacer ce développement dans les expressions de $G(z)$ et de $\tau\mathrm{CT}(z)$ (on pose ici $Z = 1 - ez$) :

```
> G := 1/2*log(1/(1-f(z))) + f(z)/2 + f(z)^2/4;
> subs(f(z)=1-sqrt(2*Z)+O(Z),G);
                    1/2  1/2                        1/2  1/2
          - 1/2 ln(2    Z    - O(Z)) + 1/2 - 1/2 2    Z    + 1/2 O(Z)

                           1/2  1/2        2
               + 1/4 (1 - 2    Z    + O(Z))

> series(",Z);
                           1/2             1/2             1/2
                 - 1/2 ln(2   ) - 1/2 ln(Z   ) + 3/4 + O(Z   )

> tauCT := f(z)/2 * (1 + f(z) + 1/(1-f(z))):
> subs(f(z)=1-sqrt(2*Z)+O(Z),tauCT);
                1/2  1/2         /     1/2  1/2                  1        \
     1/2 (1 - 2    Z    + O(Z)) |2 - 2    Z    + O(Z) + ----------------|
                                |                          1/2  1/2       |
                                \                        2    Z    - O(Z)/

> series(",Z);
                                    1
                          1/2 --------- + O(1)
                                1/2  1/2
                               2    Z
```

Les développements en $z = e^{-1}$ de $G(z)$ et $\tau\mathrm{CT}(z)$ sont par conséquent

$$G(z) = \frac{3}{4} - \frac{1}{4}\log 2 - \frac{1}{2}\log\sqrt{1 - ez} + O(\sqrt{1 - ez}), \qquad \tau\mathrm{CT}(z) = \frac{1}{2\sqrt{2}}\frac{1}{\sqrt{1 - ez}} + O(1).$$

Par transfert aux coefficients [FO90, théorème 1], ces deux développements conduisent à

$$\begin{aligned} [z^n]G(z) &= \frac{1}{2}\left([z^n]\log\frac{1}{\sqrt{1 - ez}}\right) + O(\frac{e^n}{n^{3/2}}), \\ [z^n]\tau\mathrm{CT}(z) &= \frac{1}{2\sqrt{2}}\left([z^n]\frac{1}{\sqrt{1 - ez}}\right) + O(\frac{e^n}{n}). \end{aligned}$$

Nous pouvons calculer les coefficients entre parenthèses à l'aide du programme `equivalent` de B. Salvy :

```
> equivalent(-log(sqrt(1-E*z)),z);
                                  exp(n)        exp(n)
                               (1/2 ------) + (O(------))
                                     n             3
                                                  n

> equivalent(1/sqrt(1-E*z),z);
                                  exp(n)          exp(n)
                               (----------) + (O(------))
                                  1/2  1/2          3/2
                                 Pi    n           n
```

ce qui donne

$$[z^n]G(z) = \frac{e^n}{4n} + O(\frac{e^n}{n^{3/2}}), \qquad [z^n]\tau\mathrm{CT}(z) = \frac{e^n}{2\sqrt{2\pi n}} + O(\frac{e^n}{n}).$$

La division du coefficient de $z^n$ de $\tau\mathrm{CT}(z)$ par celui de $G(z)$ donne la longueur moyenne du cycle des graphes connexes monocycliques, et achève la preuve du lemme semi-automatique 4 :

$$\frac{[z^n]\tau\mathrm{CT}(z)}{[z^n]G(z)} = \sqrt{\frac{2n}{\pi}} + O(1). \tag{3.7}$$

■

Comme le lecteur a pu le constater, la seule partie non automatique de cette preuve fut le développement de Taylor de fonctions implicites (ici $f(z)$) en un point (ici au voisinage de leur singularité dominante). Ce calcul pourrait très bien être fait par $\Lambda\Upsilon\Omega$, mais nous considérons qu'il relève plus du calcul formel que de l'analyse asymptotique, et devrait être implanté en MAPLE. Tous les autres calculs ont été réalisés automatiquement (`infsing` pour rechercher la singularité, `series` pour développer localement $G(z)$ et $\tau\mathrm{CT}(z)$, et `equivalent` pour calculer le coefficient de $z^n$ des termes singuliers).

La longueur $\gamma_n$ du cycle des graphes connexes monocycliques de $n$ points a été étudiée en 1959 par Rényi, qui montre que la distribution de $\gamma_n/n^{1/2}$ tend vers la distribution de la valeur absolue de la loi normale $N(0,1)$, ce qui implique (3.7). Citons également Bollobás qui a obtenu le nombre asymptotique de graphes $G_n \sim n!e^n/(4n)$ [Bol85, corollaire 19 page 113] par calcul de probabilités.

## 3.4 Enracinement du minimum

Cette section s'intéresse plus particulièrement aux constructions qui imposent des restrictions sur les valeurs relatives des étiquettes, et aux schémas de programmation qui dépendent de l'ordre induit ainsi sur les atomes.

Nous exposons d'abord (sous-section 3.4.1) sur un exemple un modèle simple de contrainte (constructeur d'enracinement du minimum) dû à Greene [Gre83]. Ce modèle simple est ensuite étendu aux schémas de descente sur les objets avec contrainte (sous-section 3.4.2). Dans la sous-section 3.4.4 est développée l'extension aux programmes du modèle général de Greene. Enfin, la dernière partie (sous-section 3.4.5) prolonge par une étude les développements de cette section.

### 3.4.1 Exemple d'enracinement du minimum

Une permutation d'éléments distincts $(i_1, \ldots, i_{2n+1})$ est dite alternée lorsque $i_{2k-1} > i_{2k} < i_{2k+1}$ pour $1 \leq k \leq n$. Une telle permutation (vue comme une séquence d'entiers) s'écrit de façon unique $\sigma_1 i \sigma_2$, où $i$ est l'élément minimum, et $\sigma_1$ et $\sigma_2$ sont *aussi* des permutations alternées. Par exemple, la permutation (6374512) se décompose en (63745)1(2); (63745) se décompose à son tour en (6)3(745), et (745) en (7)4(5). Ceci suggère d'introduire un nouveau constructeur, le constructeur d'*enracinement du minimum*, noté $\Box$ par Greene, afin de définir une grammaire pour les permutations :

$$P \rightarrow x \mid P \times_p x^{\Box} \times_p P.$$

Cette production dérive soit un seul élément $x$, soit le produit partitionnel d'une permutation alternée $p_1$, d'un élément $x$ et d'une autre permutation alternée $p_2$, avec la condition supplémentaire que l'étiquette portée par $x$ est inférieure à toutes celles de $p_1$ et $p_2$. En Adl, nous utilisons la notation $min(A)$ pour $A^{\Box}$, et la spécification des permutations alternées s'écrit :

**type** P = x | *product*(P, *min*(x), P);
x = *Latom*(1);

Le constructeur d'enracinement du minimum s'utilise donc en liaison avec le constructeur produit partitionnel. Greene a montré dans sa thèse que le constructeur d'enracinement du minimum se traduit en un opérateur *intégro-différentiel* sur les séries génératrices exponentielles :

**Règle 41.** *(Enracinement du minimum) [Gre83]*

$$\frac{\textbf{type } A = product(min(B), C);}{A(z) = \int_0^z \left(\frac{d}{dt}B(t)\right) C(t)dt}$$

**Démonstration :** Les objets de $\mathcal{A}$ sont formés à partir d'un objet $b \in \mathcal{B}$ de taille non nulle $i+1$ (puisqu'il contient la plus petite étiquette) et d'un objet $c \in \mathcal{C}$ de taille $j$ quelconque. Étant donnés deux tels objets $b$ et $c$, le nombre de réétiquetages $(b', c')$ compatibles respectant en plus la condition d'enracinement du minimum est $\frac{(i+j)!}{i!\,j!}$. En effet, la plus petite étiquette étant forcément dans $b'$, il en reste $i+j$ à répartir ; dès que l'on a choisi les $j$ étiquettes de $c'$, la répartition est imposée par la condition de compatibilité avec les ordres initiaux. D'où :

$$A(z) = \sum_{a \in (\mathcal{B}^{\Box} \times_p \mathcal{C})} \frac{z^{|a|}}{|a|!} = \sum_{\substack{b \in \mathcal{B} \\ |b| \geq 1}} \sum_{c \in \mathcal{C}} \frac{(|b|+|c|-1)!}{(|b|-1)!\,|c|!} \frac{z^{|b|+|c|}}{(|b|+|c|)!} = \sum_{i,j \geq 0} \frac{B_{i+1} C_j}{i!\,j!} \frac{z^{i+j+1}}{i+j+1}$$

et

$$\int_0^z \left(\frac{d}{dt}B(t)\right) C(t)dt = \int_0^z \left(\sum_{i \geq 0} B_{i+1} \frac{t^i}{i!}\right) \left(\sum_{j \geq 0} C_j \frac{t^j}{j!}\right) dt = \sum_{i,j \geq 0} \frac{B_{i+1} C_j}{i!\,j!} \frac{z^{i+j+1}}{i+j+1}. \qquad \blacksquare$$

Cette règle, associée aux règles 22 (atome) et 23 (union) conduit pour la série génératrice exponentielle $P(z)$ des permutations alternées à

$$P(z) = z + \int_0^z P^2(u)du. \tag{3.8}$$

En dérivant cette équation, on obtient l'équation différentielle de la fonction tangente :

$$P' = 1 + P^2,$$

qui se résout en $P(z) = \tan(z + C)$. Or l'équation (3.8) impose la condition initiale $P(0) = 0$ (les permutations alternées sont de longueur impaire) ; la solution est donc $P(z) = \tan(z)$. Le nombre de permutations alternées de taille 19 est ainsi $19![z^{19}]\tan(z)$, soit 29088885112832.

### 3.4.2 Extension aux programmes

Le constructeur d'enracinement du minimum intervient uniquement en liaison avec le constructeur *produit partitionnel*. En effet, pour imposer l'emplacement de la plus petite étiquette, il faut au moins deux composantes, et il est nécessaire que celles-ci soient ordonnées pour désigner l'une d'elles. Le seul schéma nouveau est par suite la sélection dans un produit partitionnel où apparaît le constructeur $\Box$. En fait, deux cas de figure peuvent advenir :

1. la sélection s'effectue sur l'objet contenant la plus petite étiquette,
2. la sélection s'effectue sur un autre objet du produit partitionnel.

A chacun de ces deux cas correspond une règle différente.

**Règle 42.**

$$\frac{\textbf{type } A = \mathit{product}(\mathit{min}(B), C); \qquad \textbf{procedure } P(a : A); \textbf{ case } a \textbf{ of } (b, c) : Q(b) \textbf{ end;}}{\tau P^A(z) = \int_0^z \left(\frac{d}{dt}\tau Q^B(t)\right) C(t)dt}$$

**Règle 43.**

$$\frac{\textbf{type } A = \mathit{product}(\mathit{min}(B), C); \qquad \textbf{procedure } P(a : A); \textbf{ case } a \textbf{ of } (b, c) : R(c) \textbf{ end;}}{\tau P^A(z) = \int_0^z \left(\frac{d}{dt}B(t)\right) \tau R^C(t)dt}$$

**Démonstration :** (Règles 42 et 43) Nous démontrons les deux règles en même temps, par l'étude du schéma d'itération **procedure** $P(a : A)$; **case** $a$ **of** $(b, c)$ : **begin** $Q(b); R(c)$ **end end** . L'équation (2.31) du chapitre 2 donnant l'opérateur correspondant à l'itération dans un multi-constructeur s'écrit ici :

$$\tau P(z) = \left.\frac{\partial}{\partial v}\Psi(B_1(z, v), C_1(z, v))\right|_{v=0}$$

où $\Psi(x_1, y_1) = \int_0^z \partial x_1/\partial t\, y_1 dt$. La dérivation par rapport à $v$ rentre sous l'intégrale et commute avec la dérivation par rapport à $t$ :

$$\tau P(z) = \left.\int_0^z \frac{\partial}{\partial t}\left(\frac{\partial B_1(t, v)}{\partial v}\right) C_1(t, v) + \frac{\partial B_1(t, v)}{\partial t}\frac{\partial C_1(t, v)}{\partial v}dt\right|_{v=0}.$$

Or ici $B_1(t, v) = \sum_{b\in\mathcal{B}}(1 + v\tau Q\{b\})t^{|b|}/|b|!$ et $C_1(t, v) = \sum_{c\in\mathcal{C}}(1 + v\tau R\{c\})t^{|c|}/|c|!$, d'où $B_1(t, 0) = B(t)$, $\partial B_1/\partial v(t, 0) = \tau Q^{\mathcal{B}}(t)$ et de même pour $C_1$, ce qui donne finalement

$$\tau P(z) = \int_0^z \left(\frac{\partial \tau Q^{\mathcal{B}}(t)}{\partial t}C(t) + \frac{\partial B(t)}{\partial t}\tau R^{\mathcal{C}}(t)\right) dt.$$

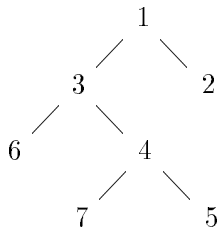



Figure 3.1 : L'arbre associé à la permutation alternée (6374512)

En remplaçant $\tau Q^{\mathcal{B}}$ ou $\tau R^{\mathcal{C}}$ par la série nulle, on obtient l'opérateur indiqué par chacune des deux règles. ■

EXEMPLE 26 : Reprenons l'exemple des permutations alternées. On peut associer à une telle permutation un arbre binaire dont la racine est l'élément minimal, et dont les sous-arbres gauche et droit sont associés aux permutations alternées constituées par les éléments respectivement à gauche et à droite de l'élément minimal (voir la figure 3.1). Les nœuds internes de cet arbre sont les *creux* de la permutation (les éléments de rang pair), et les feuilles sont les *pics* (de rang impair). Une question qui peut nous venir à l'esprit est :

*Combien de creux records a en moyenne une permutation alternée de n éléments ?*

(un creux *record* est un creux plus petit que tous ceux qui sont à sa gauche). Par exemple, la permutation (6374512) a trois creux (3, 4 et 1) dont deux sont des creux records (3 et 1). Dans l'arbre tournoi associé (figure 3.1), on voit bien que les creux records sont les nœuds internes de la branche gauche. Il est donc très simple d'écrire un programme comptant le nombre de creux records (le type `P` représente encore les permutations alternées) :

```
procedure compte_creux_records (p : P);
begin
   case p of
      x : count0;
      (p1,x,p2) : begin count1; compte_creux_records(p1) end
   end
end;

measure count0 : 0; count1 : 1;
```

Nous avons mis un coût nul (`count0`) dans le cas d'une feuille `x` car les feuilles correspondent aux pics de la permutation, qui ne peuvent pas être des creux, et à plus forte raison des creux records. Calculons l'équation du descripteur de complexité de la procédure `compte_creux_records`, que nous noterons $\tau C$. L'appel `count1` a un coût constant 1, donc par la règle 8 (page 60), sa contribution égale $1 \times Q(z)$, où $Q(z)$ est la série exponentielle des permutations de taille au moins 3 : $P(z) = z + Q(z)$, or $P(z) = \tan(z)$, d'où $Q(z) = \tan(z) - z$. Par la règle 43, la contribution de l'appel `compte_creux_records(p1)` est $\int_0^z \tau C(t) P(t) dt$. L'équation vérifiée par $\tau C$ est par conséquent

$$\tau C(z) = \tan(z) - z + \int_0^z \tau C(t) \tan(t) dt.$$

En dérivant cette équation, nous obtenons

$$\frac{d}{dz}\tau C(z) = \tan(z)^2 + \tau C(z)\tan(z)$$

avec comme condition initiale $\tau$C(0) = 0. La solution de cette équation différentielle est

$$\tau C(z) = \frac{\log \frac{1+\sin(z)}{\cos(z)} - \sin(z)}{\cos(z)}.$$

**Théorème semi-automatique 5.** *Le nombre moyen de creux records dans les permutations alternées de taille n est*

$$c_n = \frac{[z^n]\frac{\log \frac{1+\sin(z)}{\cos(z)} - \sin(z)}{\cos(z)}}{[z^n]\tan(z)}, \tag{3.9}$$

*et vaut asymptotiquement* $\log n + O(1)$.

**Démonstration :** La preuve de la première partie du théorème peut être rendue automatique, comme nous l'avons vu ci-dessus. En ce qui concerne l'analyse asymptotique, les singularités dominantes de $P(z) = \tan(z)$ et de $\tau C(z)$ sont $\pi/2$ et $-\pi/2$. Les développements locaux en $z = \pi/2$ sont

$$P(z) = \frac{1}{\frac{\pi}{2} - z} + O(\frac{\pi}{2} - z), \qquad \tau C(z) = \frac{1}{\frac{\pi}{2} - z}\log\frac{1}{\frac{\pi}{2} - z} + O(\frac{1}{\frac{\pi}{2} - z}),$$

et les développements locaux en $z = -\pi/2$ s'obtiennent en remplaçant $\frac{\pi}{2}$ par $-\frac{\pi}{2}$ ci-dessus. Il en résulte par transfert aux coefficients, que pour $n$ impair,

$$[z^n]P(z) = \pi(2/\pi)^n + O(\frac{(2/\pi)^n}{n^2}), \qquad [z^n]\tau C(z) = \pi(2/\pi)^n \log n + O((2/\pi)^n).$$

La division de ces développements donne pour le nombre moyen de creux records $\log n + O(1)$. ■

La table ci-dessous donne quelques valeurs de $c_n$ ainsi que les approximations décimales correspondantes, obtenues par développement de Taylor à partir de la formule (3.9).

| $n$ | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 |
|---|---|---|---|---|---|---|---|---|
| $c_n$ | 1 | 3/2 | 31/17 | 64/31 | 1560/691 | 52841/21844 | 2377663/929569 | 8580834/3202291 |
| | 1.00 | 1.50 | 1.82 | 2.06 | 2.26 | 2.42 | 2.56 | 2.68 |
| $\log n$ | 1.10 | 1.61 | 1.95 | 2.20 | 2.40 | 2.56 | 2.71 | 2.83 |

□

### 3.4.3 Test du minimum

Un autre schéma de programmation sur les objets étiquetées, le *test du minimum*, découle de l'équivalence des productions

$$A \to B \times_p C \qquad \text{et} \qquad A \to (B^{\square} \times_p C) \cup (B \times_p C^{\square}),$$

lorsque $B$ et $C$ ne dérivent aucun objet de taille nulle. Ainsi, le programme ci-dessous, qui appelle la procédure $Q$ lorsque le minimum est dans l'objet $b$, et la procédure $R$ lorsqu'il est dans $c$,

```
type A = product(B,C);

procedure P(a : A);
begin
   case a of
      (b,c) : if min(b)<min(c) then Q(a) else R(a);
   end;
end;
```

est équivalent au second programme que voici, qui utilise le constructeur d'enracinement du minimum et le schéma de sélection dans une union.

```
type A = A1 | A2;
     A1 = product(min(B),C);
     A2 = product(B,min(C));

procedure P(a : A);
begin
   casetype a of
      A1 : Q(a);
      A2 : R(a);
   end;
end;
```

Cette règle de transformation de programme sera utilisée dans une étude du chapitre 5 (section 5.1.4). Nous verrons aussi au chapitre 4 d'autres *règles de transformation de programme*, dont le but sera à nouveau de ramener de nouveaux schémas de programmation à ceux de la classe II.

### 3.4.4 Cas général

Le constructeur $\square$ d'enracinement du minimum est en fait seulement un cas particulier, mais néanmoins très intéressant, de la théorie de Greene. Nous décrivons ici le cas général de cette théorie, qui permet de définir un produit partitionnel en imposant un certain ordre partiel sur les étiquettes maximales et minimales.

Par exemple, la construction $A_{[a]}^{\{ef\}} \times_p B_{[bc]}^{\{g\}}$ déclare que dans l'objet dérivé par $A$, les deux étiquettes les plus grandes sont $e$ et $f$, la plus petite est $a$, et dans celui dérivé par $B$, la plus grande est $g$ et les deux plus petites sont $b$ et $c$. Ensuite, nous pouvons imposer des contraintes entre ces étiquettes extrêmes. Ainsi, la production

$$C \rightarrow \{e > f > g\}[b < c < a]\ A_{[a]}^{\{ef\}} \times_p B_{[bc]}^{\{g\}} \tag{3.10}$$

indique que l'objet dérivé par $A$ contient les deux plus grandes étiquettes ($e > f > g$), et celui dérivé par $B$ les deux plus petites ($b < c < a$).

**Définition 18.** *[Gre83, page 31] Un* ensemble partiellement ordonné *est la donnée d'un ensemble $S$ et d'une relation $\mathcal{R}$ antisymétrique et transitive sur $S$. Une* interprétation linéaire *d'un ordre partiel $\mathcal{R}$ est un ordre total $\prec$ compatible avec $\mathcal{R}$ : $a\mathcal{R}b \Rightarrow a \prec b$.*

Une production comme (3.10) définit deux ensembles partiellement ordonnés : les étiquettes les plus petites, notées en indice ($a$, $b$ et $c$), et les étiquettes les plus grandes, notées en exposant ($e$, $f$ et $g$).

Parmi les étiquettes les plus petites ($a$, $b$ et $c$ dans la production (3.10)), un élément $\beta$ est dit *frontière* s'il n'existe pas d'élément $\alpha$ tel que $\alpha > \beta$. On définit de même les éléments frontières parmi les étiquettes les plus grandes, en inversant le sens de l'inégalité. Ainsi, dans (3.10), $a$ et $g$ sont des éléments frontières.

**Définition 19.** *Parmi les étiquettes les plus petites (resp. les plus grandes), un élément est dit* actif *s'il est plus petit (resp. plus grand) que tous les éléments frontières. Un ordre partiel est* séparé *lorsque tous les éléments sont soit actifs, soit frontières.*

Dans l'exemple initial (3.10), les deux ordres partiels sur les étiquettes minimales et maximales sont séparés.

La propriété de séparation permet la traduction en séries génératrices.

**Théorème** *[Gre83, p. 34] La série génératrice exponentielle associée à la production $\{\mathcal{P}\}[\mathcal{A}]\ w$, où les ordres partiels $\mathcal{P}$ et $\mathcal{A}$ des étiquettes maximales et minimales sont séparés, est le produit*

1. *du nombre d'interprétations linéaires des éléments actifs de $\mathcal{P}$,*
2. *du nombre d'interprétations linéaires des éléments actifs de $\mathcal{A}$,*
3. *de l'intégrale $n$-ième du produit des séries génératrices exponentielles des non-terminaux de $w$, chacune dérivée autant de fois qu'il y a d'éléments actifs dans le non-terminal , et où $n$ est le nombre total d'éléments actifs.*

Dans la production (3.10), l'ordre partiel $\mathcal{P}$ sur les éléments maximaux est $e > f > g$ : $g$ est frontière tandis que $e$ et $f$ sont actifs. Sur les éléments actifs, il y a une seule interprétation linéaire possible, puisque l'ordre $e > f$ est déjà total. Pour l'ordre partiel $b < c < a$ sur les éléments minimaux, $b$ et $c$ sont actifs, et il n'y a également qu'une interprétation linéaire. Par conséquent, la production $\{e > f > g\}[b < c < a]\ A_{[a]}^{\{ef\}} \times_p B_{[bc]}^{\{g\}}$ se traduit en

$$\int_0^z \int_0^t \int_0^u \int_0^v A''(w)B''(w)\,dw\,dv\,du\,dt.$$

Un autre exemple : la production

$$T \to [a < b, a < e, c < b, c < e, d < b, d < e]\ A_{[ab]}B_{[c]}C_{[de]}$$

fait intervenir deux éléments frontières ($b$ et $e$), et trois éléments actifs ($a$, $c$ et $d$). Le graphe de gauche de la figure 3.2 représente l'ordre partiel, avec la convention que lorsque $\alpha < \beta$, l'élément $\alpha$ est dessiné sous $\beta$, et un trait les relie. Pour les trois éléments actifs, il y a six interprétations linéaires : $a < c < d$, $a < d < c$, $c < a < d$, $c < d < a$, $d < a < c$ et $d < c < a$. La traduction de cette production est par suite

$$6 \int_0^z \int_0^t \int_0^u A'(v)B'(v)C'(v)\,dv\,du\,dt.$$

En Adl, nous notons les éléments minimaux devant le non-terminal et les éléments maximaux après. Par exemple, la production (3.10) s'écrit

$$\textbf{type}\ C = \{e > f > g\}[b < c < a]\ product([a]A\{ef\}, [bc]B\{g\}),$$

et la règle correspondant au théorème de Greene s'écrit :

**Règle 44.**

$$\frac{\textbf{type } T = [\mathcal{A}]\{\mathcal{P}\}\ product([\mathcal{A}_U]U\{\mathcal{P}_U\},\ldots,[\mathcal{A}_W]W\{\mathcal{P}_W\});}{T(z) = \int^{(n)}_{t=0\ldots z} U^{(n_U)}(t)\ldots W^{(n_W)}(t)dt^n}$$

*à condition que les ordres partiels $\mathcal{A}$ et $\mathcal{P}$ soient séparés ; $n_V$ est le nombre d'éléments actifs du non-terminal $V$, $n = n_U + \cdots + n_W$ est le nombre total d'éléments actifs, $\int^{(n)}$ indique une intégrale $n$-ième et $V^{(n)}$ une dérivée $n$-ième.*

Cette règle s'étend naturellement aux programmes.

**Règle 45.**

$$\frac{\begin{array}{c}\textbf{type } T = [\mathcal{A}]\{\mathcal{P}\}\ product([\mathcal{A}_U]U\{\mathcal{P}_U\},\ldots,[\mathcal{A}_W]W\{\mathcal{P}_W\});\\ \textbf{procedure } Q(t:T);\ \textbf{case } t \textbf{ of } (u,\ldots,v,\ldots,w) : R(v) \textbf{ end;}\end{array}}{\tau Q(z) = \int^{(n)}_{t=0\ldots z} U^{(n_U)}(t)\ldots\tau R^{(n_V)}(t)\ldots W^{(n_W)}(t)dt^n}$$

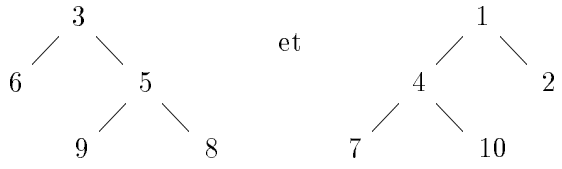
**Démonstration :** La preuve est analogue à celle des règles 42 et 43, qui constituent un cas particulier de la présente règle. Il suffit de définir $V_1(z,y) = \sum_{v\in\mathcal{V}}(1 + y\tau R\{v\})z^{|v|}/|v|!$, puis de remplacer $V(t)$ par $V(t,y)$ dans la série de dénombrement donnée par la règle précédente ; $\tau Q(z)$ est alors la valeur en $y = 0$ de la dérivée par rapport à $y$. ■

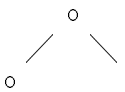
### 3.4.5 Intersection d'arbres tournois

Un exemple intéressant suggéré par R. Casas [BYCDM89] est le suivant :

*Quelle est la taille moyenne de l'intersection de deux arbres tournois de taille totale $n$ ?*

Un arbre *tournoi* est un arbre binaire dont les nœuds sont étiquetés, et tels que dans chaque sous-arbre, l'étiquette se trouvant à la racine est minimale (voir aussi la section 5.1.1). L'intersection est la partie commune (en partant de la racine) des deux arbres non étiquetés sous-jacents. D'autre part, on considère comme distribution de probabilité la distribution uniforme sur le produit partitionnel $\mathcal{T} \times_p \mathcal{T}$, où $\mathcal{T}$ est l'ensemble des arbres tournois. Par exemple, l'intersection des arbres tournois



est l'arbre de taille 3. En Adl, les paires d'arbres tournois se définissent par :

```
type TT = product(T,T);
     T = empty | product(min(x),T,T);
     x = Latom(1);
     empty = Latom(0);
```

On voudrait alors écrire la procédure calculant l'intersection de la façon suivante.

```
procedure inter (tt : TT);
begin
```

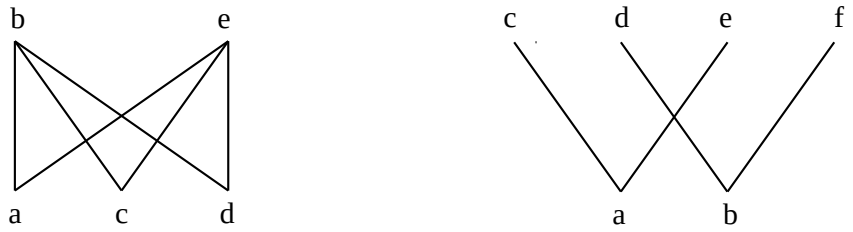



Figure 3.2 : Les ordres partiels $[a < b, a < e, c < b, c < e, d < b, d < e]$ et $[a < c, a < e, b < d, b < f]$.

```
case tt of
   (u,v) : case u of
               empty : empty;
               (x,u1,u2) : case v of
                              empty : empty;
                              (x,v1,v2) : product(o,inter(product(u1,v1)),
                                                        inter(product(u2,v2)));
                           end;
           end;
   end;
end;
```

Mais ce programme ne vérifie pas la propriété de descente : en effet, les objets $product(u1, v1)$ et $product(u2, v2)$ sur lesquels la procédure `inter` s'appelle récursivement ne sont pas de véritables composantes de l'objet `tt`, mais des produits de composantes.

On pourrait alors penser contourner la difficulté en modifiant les données par isomorphisme de façon à ce que la propriété de descente soit vérifiée. Par exemple en aplatissant les paires d'arbres tournois :

```
type TT = product(empty,T)
        | product(product(min(x),T,T),empty)
        | product(x,y,product(U1,V1),product(U2,V2));
    U1,U2,V1,V2 = T; x,y = Latom(1);
```

mais cette fois-ci, la difficulté provient de la définition des structures de données : pour que $product(x, u1, u2)$ et $product(y, v1, v2)$ soient des arbres tournois, il faut remplacer la troisième production de `TT` par

$$[a < c, a < e, b < d, b < f]product([a]x, [b]y, product([c]U1, [d]V1), product([e]U2, [f]V2))$$

mais l'ordre partiel ainsi défini n'est pas séparé : $c$, $d$, $e$ et $f$ sont des éléments frontières, $a$ et $b$ sont actifs, mais par exemple $a$ n'est pas plus petit que $d$ (graphe de droite de la figure 3.2). La solution pour traiter ce problème est d'introduire un descripteur de complexité à deux variables pour la procédure `inter` :

$$\tau I(x, y) = \sum_{(u,v) \in U \times_p V} \tau I\{(u, v)\} \frac{x^{|u|} y^{|v|}}{(|u| + |v|)!}.$$

En introduisant une seconde variable $y$, on découple les étiquettes de la première composante des paires d'arbres tournois ($a, c$ et $e$) de celles de la seconde composante ($b, d$ et $f$). Ainsi, l'ordre partiel de droite de la figure 3.2 se scinde en deux ordres partiels séparés : $[a < c, a < e]$ et $[b < d, b < f]$.

**Règle 46.**

$$\frac{\textbf{type } U = product(min(A), B);\ V = product(min(C), D);\\ \textbf{procedure } P(u:U; v:V);\ \textbf{case } u, v \textbf{ of } (a,b),(c,d) : Q(a,d) \textbf{ end;}}{\tau P(x,y) = \int_{s=0}^{x}\int_{t=0}^{y} \frac{\partial \tau Q(s,t)}{\partial s} B(s) \frac{\partial C(t)}{\partial t}\, ds\, dt}$$

*à condition que la procédure $Q$ soit indépendante des valeurs relatives des étiquettes de ses arguments.*

**Démonstration :** Décomposons le schéma en deux procédures :

```
procedure P (u : U; v : V);
case u of
   (a,b) : R(a,v)
end;
```

```
procedure R (a : A; v : V);
case v of
   (c,d) : Q(a,d)
end;
```

Le descripteur de complexité cherché est

$$\tau P(x,y) = \sum_{(u,v)\in \mathcal{U}\times_p \mathcal{V}} \tau P\{(u,v)\} \frac{x^{|u|} y^{|v|}}{(|u|+|v|)!}.$$

Si $Q$ ne dépend pas des valeurs relatives des étiquettes de ses arguments, alors il en est de même pour $P$, et le coût $\tau P\{(u,v)\}$ est le même pour chaque réétiquetage compatible de $\{u\} \times_p \{v\}$ :

$$\tau P(x,y) = \sum_{u\in\mathcal{U}, v\in\mathcal{V}} \tau P\{(u,v)\} \frac{x^{|u|}}{|u|!}\frac{y^{|v|}}{|v|!} = \sum_{u\in\mathcal{U}} \left( \sum_{v\in\mathcal{V}} \tau R\{(a,v)\} \frac{y^{|v|}}{|v|!} \right) \frac{x^{|u|}}{|u|!}.$$

Le schéma $R(a,v)$ ne dépendant que de $a$, nous pouvons appliquer la règle 42 au type $U$ et pour la variable $x$ :

$$\tau P(x,y) = \int_0^x \frac{\partial \tau R(s,t)}{\partial s} B(s) ds. \tag{3.11}$$

Le même raisonnement s'applique à la procédure $R$, pour le type $V$ et la variable $y$ :

$$\tau R(x,y) = \int_0^y \frac{\partial C(t)}{\partial t} \tau Q(s,t) dt. \tag{3.12}$$

En combinant (3.11) et (3.12), il vient la formule annoncée. ■

La règle 46 n'est qu'une partie d'un groupe de quatre règles, suivant que la procédure $Q$ prend comme argument $a$ ou $b$, $c$ ou $d$.

REMARQUE : Plus généralement, lorsqu'un ordre partiel n'est pas séparé, il est parfois possible de le scinder en plusieurs ordres partiels séparés.

**Définition 20.** *Un ordre partiel est dit* séparable *lorsque les composantes connexes du graphe associé définissent des ordres partiels séparés.*

Par exemple, l'ordre partiel de droite de la figure 3.2 est séparable. Par contre, si on enlève une contrainte dans l'ordre partiel de gauche de cette même figure, l'ordre obtenu n'est pas séparable.

**Règle 47.** *Soit un schéma*

$$\textbf{type } T = [\mathcal{A}]\{\mathcal{P}\}\ product([\mathcal{A}_U]U\{\mathcal{P}_U\}, \ldots, [\mathcal{A}_W]W\{\mathcal{P}_W\});$$
$$\textbf{procedure } P(t:T);\ \textbf{case } t\ \textbf{of } u, \ldots, w : Q(\ldots, v, \ldots)\ \textbf{end;}$$

*Supposons qu'il existe une partition* $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ *de l'ensemble* $\{U, \ldots, W\}$ *des non-terminaux du membre droit telle que pour chaque partie* $\mathcal{S}$*, les restrictions* $\mathcal{A}_\mathcal{S}$ *et* $\mathcal{P}_\mathcal{S}$ *de* $\mathcal{A}$ *et* $\mathcal{P}$ *à* $\mathcal{S}$ *sont séparées, et* $\mathcal{A}$ *et* $\mathcal{P}$ *sont la réunion des* $\mathcal{A}_\mathcal{S}$ *et des* $\mathcal{P}_\mathcal{S}$ *respectivement. Soit* $n_j$ *le nombre d'éléments actifs de* $\mathcal{A}_{\mathcal{S}_j}$ *et* $\mathcal{P}_{\mathcal{S}_j}$*. Alors le type* $T$ *admet une série génératrice à* $k$ *variables*

$$T(z_1, \ldots, z_k) = \int_{t_1=0..z_1}^{(n_1)} \cdots \int_{t_k=0..z_k}^{(n_k)} U^{(n_U)}(z_{j(U)}) \ldots W^{(n_W)}(z_{j(W)}) dz_1^{n_1} \ldots dz_k^{n_k}, \tag{3.13}$$

*et la procédure* $P$ *admet un descripteur de complexité à* $k$ *variables* $\tau P(z_1, \ldots, z_k)$ *qui s'obtient en remplaçant dans le membre droit de (3.13) le produit des séries* $V^{(n_V)}(z_{j(v)})$ *pour les types* $V$ *des arguments de* $Q$ *par la série* $\tau Q(\ldots, z_{j(V)}, \ldots)$*.*

**Démonstration :** Cette règle ne fait que généraliser la règle 46, et sa preuve est similaire. ■

Revenons au problème de l'intersection d'arbres tournois. La règle 46 nous permet d'analyser la procédure `inter` :

$$\tau I(x, y) = \int_{s=0}^{x} \int_{t=0}^{y} T^2(s)T^2(t) + 2T(s)T(t)\tau I(s, t)\, ds\, dt.$$

En remplaçant $T(z)$ par $1/(1-z)$, nous obtenons finalement :

$$\tau I(x, y) = \frac{x}{1-x}\frac{y}{1-y} + \int_{s=0}^{x} \int_{t=0}^{y} \frac{2}{(1-s)(1-t)} \tau I(s, t)\, ds\, dt. \tag{3.14}$$

**Théorème automatique 6.** *La taille de l'intersection de deux arbres tournois de taille totale* $n$ *est en moyenne (toutes les paires du produit partitionnel étant équiprobables)*

$$\overline{\tau I}_n = \frac{[z^n]\tau I(z, z)}{n+1}$$

*où* $\tau I(x, y)$ *est solution de l'équation (3.14).*

La contribution d'un monôme $s^k t^l$ de $\tau I(s, t)$ à l'intégrale donne comme plus petit terme (au sens du degré total) $2s^{k+1}/(k+1)t^{l+1}/(l+1)$. L'équation ci-dessus permet donc de déterminer les termes de degré total inférieur à $2n$ de $\tau I(x, y)$ en $n$ étapes. Il suffit ensuite de remplacer $x$ et $y$ par $z$ pour obtenir les premiers termes du descripteur de complexité à *une* variable $\tau I(z) = \tau I(z, z)$.

```
tauI := proc(n)
local S,i;
      S := 0;
      for i from 1 by 2 to n do
         subs(x=s,y=t,S);
         x/(1-x) * y/(1-y) + int(int(2*"/(1-s)/(1-t),t=0..y),s=0..x);
         convert(taylor(",x,i+1),polynom);
         convert(taylor(",y,i+1),polynom);
         S := expand(")
      od;
      series(subs(x=z,y=z,S)+z^(n+1),z,n+1)
end:
```

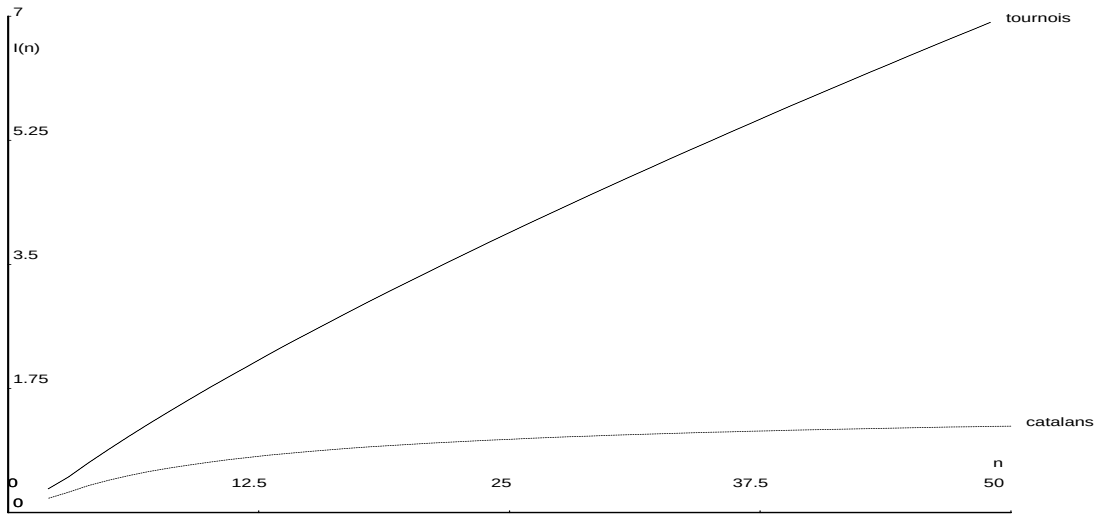



Figure 3.3 : Le graphe de $\overline{\tau I}_n$ en fonction de $n$.

Ce programme Maple détermine le développement de Taylor de $\tau I(z,z)$ jusqu'à l'ordre $n$ :

```
> tauI(10);
      2      3        4         5          6   899  7   9209  8   30043  9       10
     z  + 2 z  + 7/2 z  + 16/3 z  + 15/2 z  + --- z  + ---- z  + ----- z  + O(z  )
                                               90       720       1890
```

Un programme plus astucieux nous a permis d'aller jusqu'à $n = 50$. Le graphe 3.3 montre la variation de la taille moyenne de l'intersection de deux arbres tournois (courbe supérieure), et de deux arbres binaires non étiquetés (type "Catalan", courbe inférieure). Pour les arbres de type Catalan, la taille moyenne de l'intersection converge vers $3/2$ lorsque $n$ tend vers l'infini.

L'analyse asymptotique de $\overline{\tau I}_n$ est difficile ; elle a été réalisée par Baeza-Yates, Casas, Díaz et Martínez à l'aide de fonctions de Bessel apparaissant dans les solutions d'équations aux dérivées partielles :

**Théorème** *[BYCDM89] La taille moyenne de l'intersection de deux arbres tournois de taille totale $n$ vaut asymptotiquement*

$$\overline{\tau I}_n = c \cdot \frac{n^{2\sqrt{2}-2}}{\sqrt{\log n}} \cdot \left(1 + O(\frac{1}{\log n})\right),$$

*où* $c = (3+2\sqrt{2})/(2^{5/4}\sqrt{\pi}\Gamma(2\sqrt{2})) \simeq 0.8050738$.

Le tableau ci-dessous compare la valeur exacte de $\overline{\tau I}_n$ au premier terme $\delta_n$ du développement asymptotique donné par le théorème.

| $n$ | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\overline{\tau I}_n/\delta_n$ | 0.37 | 0.49 | 0.55 | 0.59 | 0.61 | 0.63 | 0.65 | 0.66 | 0.67 | 0.68 |

Comme nous l'avons montré dans ce chapitre, le constructeur d'enracinement du minimum conduit à des équations différentielles pour les séries génératrices associées. Réciproquement, cela permet de trouver une interprétation combinatoire à certaines équations différentielles. Considérons

par exemple l'équation

$$\frac{dy}{dt} = \alpha y + \beta y^2 + u(t)$$

d'un circuit électrique constitué d'une source $u(t)$ en parallèle avec une capacité, une résistance linéaire et une autre non linéaire [LV88a]. Par intégration de cette équation, nous obtenons

$$y = \int (\alpha y + \beta y^2 + u) dt,$$

ce qui s'interprète de façon arborescente en

$$\mathbf{type}\ Y = \alpha\ \overset{x^{\square}}{|}\ Y \quad | \quad \beta\ \overset{x^{\square}}{\diagup\ \diagdown}\ Y\ Y \quad | \quad \overset{x^{\square}}{|}\ U\ ,$$

où $x$ est un atome étiqueté de taille 1. Nous avons ainsi retrouvé aisément l'interprétation en termes d'arbres de Motzkin donnée par Leroux et Viennot dans [LV88a].

# Chapitre 4

# Fonctions à nombre fini de valeurs

*Le mieux est seulement l'ennemi du parfait. Mais comme nul n'y atteint ... Qui ne saurait faire mieux pourra toujours faire plus.*
Un professeur de rhétorique

Le langage des classes Π et $\hat{\Pi}$ est relativement restreint. Notamment, il n'autorise pas l'utilisation de fonctions. Ce chapitre considère l'extension de ce langage par des fonctions dont la valeur renvoyée varie dans un ensemble fini (par exemple des fonctions booléennes) ; nous montrons qu'il est possible d'inverser en quelque sorte une telle fonction, c'est-à-dire de générer une spécification de l'ensemble des données pour lesquelles elle prend une valeur fixée. Ce mécanisme d'inversion autorise alors l'analyse de programmes du langage étendu.

Par exemple, la fonction `contient_aaa` ci-dessous détermine si un mot du langage $L = \{a, b\}^*$ contient la séquence $aaa$, en se servant de deux fonctions auxiliaires `apres_a` (qui suppose que l'on a déjà trouvé $a$) et `apres_aa` (qui suppose que l'on a reconnu $aa$) :

```
type L = epsilon | product(lettre,L);  lettre = a | b;  a,b = atom(1);

function contient_aaa(l : L) : boolean;
begin
    case l of
        epsilon : false;
        (a,x)   : apres_a(x);
        (b,x)   : contient_aaa(x);
    end;
end;

function apres_a(l : L) : boolean;
begin
    case l of
        epsilon : false;
        (a,x)   : apres_aa(x);
        (b,x)   : contient_aaa(x);
    end;
end;

function apres_aa(l : L) : boolean;
begin
    case l of
        epsilon : false;
        (a,x)   : true;
        (b,x)   : contient_aaa(x);
    end;
end;
```

Nous verrons dans ce chapitre que cet ensemble de trois fonctions se transforme en une *spécification* du langage $L_t$ des mots contenant la séquence $aaa$ :

```
type Lt = a Mt | b Lt;          Mt = a Nt | b Lt;          Nt = a L  | b Lt;
     Lf = epsilon | a Mf | b Lf; Mf = epsilon | a Nf | b Lf; Nf = epsilon | b Lf;
```

A l'aide de cette spécification, nous pourrons déterminer la probabilité qu'un mot de taille $n$ du langage $L$ contienne le motif $aaa$ (densité du langage $L_t$ dans $L$) :

**Théorème automatique 7.** *La probabilité qu'un mot de n lettres du langage $L = \{a, b\}^*$ comporte le motif aaa est :*

$$p_n = \frac{1}{2^n}[z^n]\frac{z^3}{1 - 3z + z^2 + z^3 + 2z^4} = 1 + O(1/n).$$

Nous pourrons aussi déterminer le rang moyen d'apparition du premier motif $aaa$ :

**Théorème automatique 8.** *Le nombre moyen $l_n$ de lettres qu'il faut parcourir pour trouver le premier motif aaa dans un mot de n lettres du langage $L = \{a, b\}^*$ est asymptotiquement constant :*

$$l_n = \frac{1}{2^n}[z^n]\frac{2z(1 + z + z^2)}{1 - 3z + z^2 + z^3 + 2z^4} = 14 + O(1/n).$$

Le plan de ce chapitre est le suivant. Nous définissons en premier lieu dans la section 4.1 une extension $\Pi_{\text{bool}}$ de la classe $\Pi$ (définie au chapitre 1) par des fonctions booléennes. La section 4.2 montre alors que les programmes de $\Pi_{\text{bool}}$ se transforment en programmes de $\Pi$. L'analyse dans $\Pi$ pouvant se faire de manière automatique (cf chapitre 2), il en est donc de même dans $\Pi_{\text{bool}}$. Ensuite, quelques exemples sont développés dans la section 4.3, tandis que la section 4.4 montre que les résultats de la section 4.2 se généralisent à d'autres types de fonctions à nombre fini de valeurs (non booléennes). Nous verrons également dans cette dernière section que, lorsque les données sont des mots, le langage étendu défini dans ce chapitre correspond aux automates finis, qui caractérisent les langages réguliers.

## 4.1 La classe $\Pi_{\text{bool}}$

Nous considérons ici une classe de programmes contenant au moins les constructeurs *union* et *produit*. Pour simplifier l'exposé, nous choisissons de nous placer dans la classe $\Pi$, mais les résultats de ce chapitre sont aussi valables pour $\hat{\Pi}$. Cette section définit une nouvelle classe de programmes $\Pi_{\text{bool}}$ qui contient, outre les programmes de $\Pi$, des fonctions booléennes.

### 4.1.1 Fonctions booléennes

Nous définissons ici des fonctions booléennes sur des structures de données issues d'une spécification de $\Omega$. Décrivons d'abord leur syntaxe : nous considérons des fonctions booléennes de la forme (ci-dessous, les variables $f, g, h \ldots$ représentent des fonctions booléennes) :

```
function f(t : T) : boolean;
begin
     ⟨instr(t)⟩
end;
```

où $f$ est le nom de la fonction, $t$ le nom et $T$ le type de son argument. Le corps de la fonction $\langle instr(t)\rangle$ est soit une expression booléenne $\langle cond(t)\rangle$ dépendant de $t$, soit un schéma de sélection dans une union (**casetype**) ou dans un produit cartésien (**case**) :

$$\begin{array}{rcl} \langle instr(t)\rangle & ::= & \langle cond(t)\rangle \\ & | & \textbf{casetype } t \textbf{ of } U : g(t);\ V : h(t) \textbf{ end} \\ & | & \textbf{case } t \textbf{ of } (u,v) : \langle cond(u,v)\rangle \textbf{ end } . \end{array} \tag{4.1}$$

Le schéma $\langle cond(t)\rangle$ correspond à une expression booléenne dépendant uniquement de l'argument $t$ de la fonction, par l'intermédiaire éventuel d'autres fonctions booléennes $g(t)$ :

$$\begin{array}{rcl} \langle cond(t)\rangle & ::= & true \mid false \\ & | & g(t) \qquad \text{où } g : \mathcal{T} \to \{\text{true, false}\} \\ & | & \langle cond(t)\rangle \text{ and } \langle cond(t)\rangle \\ & | & \langle cond(t)\rangle \text{ or } \langle cond(t)\rangle \\ & | & \text{not } \langle cond(t)\rangle \end{array} \tag{4.2}$$

Ici, la sémantique des opérateurs logiques *and* et *or* est celle du langage Pascal : l'évaluation de $g(t)$ and $h(t)$ provoque l'évaluation de $g(t)$ *et* de $h(t)$, même si $g(t)$ renvoie *false*.

Le schéma **casetype** $t$ **of** $U : g(t);\ V : h(t)$ **end** s'applique à une union $T \to U \mid V$ ; il retourne la valeur de la fonction booléenne $g$ lorsque $t$ est de type $U$, et celle de $h$ lorsque $t$ est de type $V$. Le troisième schéma (**case**) s'applique à un produit $U \times V$, et permet de renvoyer une expression booléenne dépendant des deux composantes $u$ et $v$ :

$$\begin{array}{rcl} \langle cond(u,v)\rangle & ::= & true \mid false \\ & | & g(u) \mid h(v) \quad \text{où } g : \mathcal{U} \to \{\text{true, false}\} \text{ et } h : \mathcal{V} \to \{\text{true, false}\} \\ & | & \langle cond(u,v)\rangle \text{ and } \langle cond(u,v)\rangle \\ & | & \langle cond(u,v)\rangle \text{ or } \langle cond(u,v)\rangle \\ & | & \text{not } \langle cond(u,v)\rangle . \end{array} \tag{4.3}$$

Les trois fonctions `contient_aaa`, `apres_a` et `apres_aa` de l'introduction peuvent être mises sous la forme définie ci-dessus, en introduisant des types et des fonctions auxiliaires. En effet, avec la convention d'écriture introduite par la section 1.6.3 (surcharge du schéma **case** ... **end** ), la définition stricte de la fonction `contient_aaa`, par exemple, s'écrit comme suit :

**type** L = epsilon | aL | bL; aL = *product*(a,L); bL = *product*(b,L);

```
function f(l : L):boolean;
casetype l of
    epsilon : false;
    aL      : g(l);
    bL      : h(l)
end;

function g(l : aL):boolean;
case l of
    (a,x) : apres_a(x);
end;

function h(l : bL):boolean;
case l of
    (b,x) : f(x)
end;
```

La fonction `f`, équivalente à `contient_aaa`, utilise le schéma de sélection dans une union (il faudrait normalement encore définir une fonction prenant comme seul argument `epsilon` et renvoyant `false`) ; les fonctions `g` et `h` utilisent le schéma de sélection dans un produit et appellent des fonctions booléennes sur les composantes (ici `apres_a` et `f`).

Nous dirons qu'un programme (ensemble de fonctions et de procédures) est *complètement défini* lorsque toutes les fonctions ou procédures appelées sont définies dans ce programme. En d'autres termes, dans un programme complètement défini, l'ensemble des procédures (ou fonctions) est égal à sa fermeture réflexive et transitive par la relation "$P$ appelle $Q$".

### 4.1.2 Le schéma conditionnel

Maintenant que nous savons écrire des fonctions sur les structures de données, nous aimerions bien les utiliser, pour diriger l'exécution d'après la valeur de la fonction. Le schéma **if** ... **then** ... **else** nous permet cela, les autres schémas étant ceux de la classe II.

SCHÉMA CONDITIONNEL : ce schéma dirige l'exécution du programme suivant le résultat d'un appel de fonction :

$P(a : A) :=$ **if** $f(a)$ **then** $Q(a)$ **else** $R(a)$

où $f$ est une fonction booléenne dont les arguments sont de type $A$. Lorsque l'évaluation de $f(a)$ renvoie la valeur *true*, c'est $Q(a)$ qui est évalué, sinon c'est $R(a)$.

Ainsi, pour déterminer la probabilité qu'un mot du langage $L = \{a, b\}^*$ contienne le motif $aaa$, nous écrivons simplement la procédure

```
procedure proba_aaa(l : L);
begin
   if contient_aaa(l) then eureka;
end;

measure eureka : 1;
```

Le coût de l'appel `proba_aaa(l)` est le coût de l'instruction élémentaire `eureka`, soit 1, lorsque `l` contient le motif $aaa$, et 0 sinon. Par conséquent, le coût moyen sur les mots de taille $n$ est exactement la probabilité qu'un tel mot contienne le motif $aaa$.

REMARQUE : Le schéma de sélection dans une union

$P(a : A) :=$ **casetype** $a$ **of** $B : Q(a)$; $C : R(a)$ **end**

devient un cas particulier du schéma conditionnel, $f$ étant la fonction indicatrice de $\mathcal{B}$ dans $\mathcal{A}$ :

```
function f (a : A) : boolean;
begin
   casetype a of
      B : true;
      C : false
   end
end;
```

Le schéma conditionnel est plus "puissant" que le schéma de sélection dans une union, puisqu'il permet de tester des propriétés des objets à une "profondeur" quelconque. Pour être plus précis, en termes d'arbres de dérivation, le schéma de sélection dans une union consulte uniquement la racine de l'arbre, alors que dans le schéma conditionnel, la fonction $f$ peut visiter l'arbre dans sa totalité.

**Définition 21.** *La classe* $\Pi_{\text{bool}}$ *est constituée des programmes complètement définis dont les spécifications sont dans* $\Omega$*, contenant des fonctions booléennes définies suivant (4.1), (4.2) et (4.3), et des procédures utilisant les schémas de* II*, plus le schéma conditionnel.*

Il faut noter que les fonctions booléennes, telles que nous les avons définies, ne permettent pas de "descendre" dans les multi-constructions. Par conséquent, la valeur d'une fonction booléenne sur une séquence, un ensemble ou un cycle est soit toujours *true*, soit toujours *false*.

## 4.2 Analyse des programmes de $\Pi_{\mathrm{bool}}$

L'objectif de cette section est de prouver que pour tout programme de la classe $\Pi_{\mathrm{bool}}$, il existe un programme de $\Pi$ ayant le même coût (théorème 8). De plus, un tel programme équivalent est calculable, par les algorithmes **Transformation** et **Réduction** des sections 4.2.2 et 4.2.3. Ainsi, à l'aide de ces algorithmes et les règles énoncées aux chapitres précédents, tout programme de la classe $\Pi_{\mathrm{bool}}$ s'analyse automatiquement.

### 4.2.1 Sous-types

Une fonction booléenne sur un ensemble $\mathcal{A}$ d'objets peut être interprétée comme l'indicatrice d'un sous-ensemble $\mathcal{A}'$ de $\mathcal{A}$. Du fait des conditions que nous avons imposées aux fonctions booléennes (section 4.1.1), les sous-ensembles que peuvent caractériser ces fonctions sont particuliers :

**Définition 22.** *(Sous-type) Un type $A'$ est un* sous-type *d'un type $A$, ce que nous noterons par $A' \subset A$, lorsque, les spécifications de $A$ et $A'$ étant sous forme compacte (voir page 21),*

- *si $A \to a$, alors $A' \to \emptyset$ ou $A' \to a$,*
- *si $A \to B \mid C$, alors $A' \to \emptyset$ ou $A' \to B'$ ou $A' \to C'$ ou $A' \to B' \mid C'$ avec $B' \subset B$ et $C' \subset C$,*
- *si $A \to B \times C$, alors $A' \to \uplus_i B'_i \times C'_i$, avec $B'_i \subset B$ et $C'_i \subset C$,*
- *si $A \to \Phi(B)$ où $\Phi$ est un multi-constructeur , alors $A' \to \emptyset$ ou $A' \to \Phi(B)$.*

*Inversement, on dit que $A$ est un* sur-type *de $A'$.*

Notons bien, que d'après cette définition, la notion de sous-type dépend explicitement des spécifications, et non pas seulement des objets dérivés. Par exemple, observons les trois spécifications suivantes, dont la première décrit le langage $\{a,b\}^*$ déjà introduit plus haut.

$$(\Sigma_1)\begin{cases} L \to \epsilon \mid M \\ M \to xL \\ x \to a \mid b \end{cases} \qquad (\Sigma_2)\begin{cases} L' \to \epsilon \mid M' \\ M' \to x'L' \\ x' \to a \end{cases} \qquad (\Sigma_3)\begin{cases} L'' \to \epsilon \mid M'' \mid N'' \\ M'' \to aL'' \\ N'' \to bL'' \end{cases}$$

Nous avons $L' \subset L$ car $M' \subset M$ et $x' \subset x$. Par contre, $L''$ n'est pas sous-type de $L$ car la production définissant $L''$ comprend trois symboles dans son membre droit, alors qu'il n'y en a que deux pour $L$. Cependant, $L$ et $L''$ décrivent bien le même langage $\{a,b\}^*$. La notion de sous-type est donc plus forte que celle d'inclusion des langages induits : si $A' \subset A$, alors $\mathcal{L}(A') \subset \mathcal{L}(A)$ mais la réciproque est fausse comme le montre l'exemple de $L$ et $L''$.

### 4.2.2 Transformation des fonctions en types conditionnels

Nous montrons dans cette sous-section comment transformer la définition d'une fonction en une spécification des sous-types pour lesquels celle-ci retourne *true* et *false* :

**Définition 23.** *Soit une fonction booléenne $f : \mathcal{T} \to \{true, false\}$. Les* types conditionnels *de $\mathcal{T}$ pour la fonction $f$ sont les ensembles $\mathcal{T}_{f=\mathrm{true}} = \{t \in \mathcal{T} \mid f(t) = \mathrm{true}\}$ et $\mathcal{T}_{f=\mathrm{false}} = \{t \in \mathcal{T} \mid f(t) = \mathrm{false}\}$.*

**Lemme 9.** *Les types conditionnels des programmes de* $\Pi_{\mathrm{bool}}$ *sont exprimables par une spécification de la classe* $\Omega_\cap$ *(*$\Omega$ *enrichie du constructeur d'intersection).*

Avant de prouver ce lemme, il nous faut préciser ce que nous entendons par "constructeur intersection".

CONSTRUCTEUR INTERSECTION : la production $T \to U \cap V$ signifie que

1. $U$ et $V$ ont un sur-type commun,
2. les objets dérivés par $T$ sont ceux dérivés à la fois par $U$ et $V$.

**Démonstration :** Soit un programme de $\Pi_{\mathrm{bool}}$. Nous allons exhiber une spécification contenant, pour chaque fonction $f$ du programme, de type $\mathcal{T}$, deux non-terminaux $T_{f=\mathrm{true}}$ et $T_{f=\mathrm{false}}$ dérivant respectivement les objets de $\mathcal{T}_{f=\mathrm{true}}$ et de $\mathcal{T}_{f=\mathrm{false}}$. Pour cela, nous utilisons l'algorithme suivant.

**Algorithme Transformation :**
Donnée : une fonction booléenne $f$ et la spécification de son type $T$
Résultat : une spécification des types conditionnels $T_{f=\mathrm{true}}$ et $T_{f=\mathrm{false}}$
{ *le corps de la fonction est de l'une des trois sortes décrites en (4.1)* }

- si le corps de $f$ est du genre $\langle$*cond(t)*$\rangle$, d'après la définition (4.2), c'est une expression booléenne comportant des opérateurs logiques *and*, *or* ou *not*, et des valeurs booléennes *true*, *false*, $g_i(t)$, $1 \leq i \leq k$ où les $g_i$ sont des fonctions booléennes. Soit $b_i$ la valeur de $g_i(t)$ : $f(t)$ ne dépend alors que des $b_i$, et par suite il existe alors une fonction $F : \{\mathrm{true}, \mathrm{false}\}^k \to \{\mathrm{true}, \mathrm{false}\}$ telle que $f(t) = F(b_1, \ldots, b_k)$. Par conséquent,
$$T_{f=\mathrm{true}} \to \biguplus_{\substack{(b_1,\ldots,b_k)\in\{\mathrm{true},\mathrm{false}\}^k \\ F(b_1,\ldots,b_k)=\mathrm{true}}} T_{g_1=b_1} \cap \cdots \cap T_{g_k=b_k} \tag{4.4}$$
et de même pour $T_{f=\mathrm{false}}$, en remplaçant $F(b_1, \ldots, b_k) = \mathrm{true}$ par $F(b_1, \ldots, b_k) = \mathrm{false}$ (l'union est disjointe car un objet donné admet une seule image par chaque fonction $g_i$).
- si le corps de $f$ est du genre **casetype** $t$ **of** $U : g(t)$; $V : h(t)$ **end** , alors
$$T_{f=b} \to U_{g=b} \biguplus V_{h=b} \tag{4.5}$$
pour $b \in \{\mathrm{true}, \mathrm{false}\}$. Ici aussi, l'union est disjointe car $U$ et $V$ dérivent des objets distincts par hypothèse.
- si le corps de $f$ est du genre **case** $t$ **of** $(u, v)$ : $\langle$*cond(u,v)*$\rangle$ **end** , alors comme pour le premier cas, $f(t)$ s'écrit $F(b_1, \ldots, b_k, c_1, \ldots, c_l)$ où $b_i = g_i(u)$ et $c_j = h_j(v)$, d'où
$$T_{f=b} \to \biguplus_{\substack{(b_1,\ldots,b_k,c_1,\ldots,c_l)\in\{\mathrm{true},\mathrm{false}\}^{k+l} \\ F(b_1,\ldots,b_k,c_1,\ldots,c_l)=b}} (U_{g_1=b_1} \cap \cdots \cap U_{g_k=b_k}) \times (V_{h_1=c_1} \cap \cdots \cap V_{h_l=c_l}) . \tag{4.6}$$

Cet algorithme traduit toute fonction booléenne en un ensemble de productions du type (4.4), (4.5) ou bien (4.6). Ces productions forment bien une spécification de la classe $\Omega_\cap$. ∎

### 4.2.3 Réduction des intersections

Par l'algorithme **Transformation**, les fonctions de $\Pi_{\mathrm{bool}}$ se compilent en des spécifications de la classe $\Omega_\cap$. Nous montrons dans ce paragraphe qu'il est possible d'éliminer les intersections de ces spécifications.

**Définition 24.** *Soit une spécification* $\Sigma$. *Une spécification* $\Sigma'$ généralise $\Sigma$ *si chaque non-terminal de* $\Sigma$ *se trouve aussi dans* $\Sigma'$, *et y dérive le même langage.*

**Lemme 10.** *Soit* $\Sigma$ *une spécification de* $\Omega_\cap$. *Alors il existe une spécification de* $\Omega$ *qui généralise* $\Sigma$, *et qui est calculable.*

**Démonstration :** L'algorithme ci-dessous élimine les intersections d'une spécification de $\Omega_\cap$, pour en faire une spécification de $\Omega$.

**Algorithme Réduction :**
Donnée : une spécification $\Sigma$ de $\Omega_\cap$
Résultat : une spécification de $\Omega$ qui généralise $\Sigma$

$F := \{\}$ { *intersections déjà calculées :* $(\alpha, \beta, \gamma) \in F \Rightarrow \gamma \to \alpha \cap \beta$ }
$\mathcal{P} :=$ {productions de $\Sigma$}
**tant qu'**il reste une production $U_0 \to U_1 \cap U_2$ dans $\mathcal{P}$

**si** il existe un triplet $(U_1, U_2, \alpha)$ dans $F$
remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to \alpha$
**sinon**
soit $U$ un sur-type de $U_1$ et $U_2$
ajouter le triplet $(U_1, U_2, U_0)$ à $F$
**si** $U \to a$, { *nécessairement* $U_i \to \emptyset$ *ou* $U_i \to a$ }
**si** $U_1 \to \emptyset$ ou $U_2 \to \emptyset$, remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to \emptyset$
**si** $U_1 \to a$ et $U_2 \to a$, remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to a$
**si** $U \to V \mid W$, { $U_i \to V_i \mid W_i$, *avec éventuellement* $V_i = \emptyset$ *ou* $W_i = \emptyset$ }
$\beta :=$ Inter$(V_1, V_2)$ $\gamma :=$ Inter$(W_1, W_2)$
remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to \beta \mid \gamma$
**si** $U \to V \times W$, { $U_1 \to \uplus_i V_i \times W_i$ *et* $U_2 \to \uplus_j V'_j \times W'_j$ }
$\beta_{i,j} :=$ Inter$(V_i, V'_j)$ $\gamma_{i,j} :=$ Inter$(W_i, W'_j)$
remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to \uplus_{i,j} \beta_{i,j} \times \gamma_{i,j}$
**si** $U \to \Phi(V)$, { $U_i \to \emptyset$ *ou* $U_i \to \Phi(V)$ }
**si** $U_1 \to \emptyset$ ou $U_2 \to \emptyset$, remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to \emptyset$
**si** $U_1 \to \Phi(V)$ et $U_2 \to \Phi(V)$, remplacer $U_0 \to U_1 \cap U_2$ par $U_0 \to \Phi(V)$

**fin tant que**

**fonction** Inter$(U_1, U_2)$

**début**
  **si** $U_1 = U_2$, renvoyer $U_1$
  **sinon si** il existe un triplet $(U_1, U_2, \alpha)$ dans $F$, renvoyer $\alpha$
  **sinon si** il existe une production $\beta \to U_1 \cap U_2$ dans $\mathcal{P}$, renvoyer $\beta$
  **sinon**
    créer un nouveau symbole $\gamma$
    ajouter la production $\gamma \to U_1 \cap U_2$ à $\mathcal{P}$
    renvoyer $\gamma$
**fin**

Montrons que l'algorithme termine : à chaque passage dans la boucle **tant que**, une production $U_0 \to U_1 \cap U_2$ est remplacée par une production sans intersection. Le nombre d'intersections dans $\mathcal{P}$ diminue donc d'une unité, sauf éventuellement lorsque la fonction Inter est appelée. La fonction Inter introduit une nouvelle intersection $\gamma \to U_1 \cap U_2$ au plus une seule fois pour chaque paire $(U_1, U_2)$, et ce au premier appel (ensuite cette production se trouve dans $\mathcal{P}$, puis lorsqu'elle est traitée par la boucle **tant que**, le triplet $(U_1, U_2, \gamma)$ est mis dans $F$). Soit $I_{\text{initial}}$ le nombre initial d'intersections, $\mathcal{N}_0$ l'ensemble des sur-types, $n(U)$ le nombre de sous-types d'un sur-type $U \in \mathcal{N}_0$. Le nombre de passages dans la boucle **tant que** est inférieur à

$$I_{\text{initial}} + \sum_{U \in \mathcal{N}_0} \binom{n(U)}{2},$$

et par suite fini. Lorsque l'on sort de la boucle **tant que**, les productions de $\mathcal{P}$ ne contiennent plus d'intersection, donc définissent une spécification de $\Omega$. Celle-ci généralise $\Sigma$ par construction. ■

Les deux lemmes qui précèdent nous permettent d'énoncer le théorème fondamental de ce chapitre.

**Théorème 8.** *Pour tout programme $\pi$ de $\Pi_{\text{bool}}$, il existe dans $\Pi$ un programme $\pi'$ équivalent, c'est-à-dire :*

- *$\pi'$ contient les types de $\pi$,*
- *à toute procédure $P$ de $\pi$ est associée une procédure $P'$ de $\pi'$ ayant même type, et pour tout argument $a$, les évaluations de $P(a)$ et de $P'(a)$ sont identiques*[1],
- *pour toute fonction booléenne $f$ de $\pi$, de type $\mathcal{T}$, $\pi'$ contient les types conditionnels $T_{f=\text{true}}$ et $T_{f=\text{false}}$.*

**Démonstration :** (constructive) Tout d'abord, à partir des types et des fonctions de $\pi$, nous calculons une spécification pour les types conditionnels, utilisant le constructeur intersection (algorithme Transformation et lemme 9). Puis nous éliminons les intersections pour obtenir une spécification de $\Omega$ (algorithme Réduction et lemme 10). Enfin, nous remplaçons les schémas conditionnels se trouvant dans le corps des procédures de $\pi$

**if** $f(t)$ **then** $Q(t)$ **else** $R(t)$

par

[1] Ici, le terme "identique" signifie que les séquences d'instructions élémentaires provoquées par $P(a)$ et $P'(a)$ sont les mêmes ; en conséquence le coût est le même : $\tau P\{a\} = \tau P'\{a\}$.

**casetype** $t$ **of**
$T_{f=\mathrm{true}} : Q(t)$;
$T_{f=\mathrm{false}} : R(t)$;
**end** .

Cette transformation laisse inchangée l'exécution des procédures. ■

**Corollaire 4.** *Si le caractère bien fondé est décidable dans* $\Pi$, *alors il l'est aussi dans* $\Pi_{\mathrm{bool}}$.

REMARQUE : La définition des fonctions que nous avons donnée à la section 4.1 ne permet pas d'insérer des instructions élémentaires dans le corps d'une fonction. En conséquence, le coût d'un appel de fonction est toujours nul dans notre modèle, ce qui justifie le fait que le coût de **if** $f(t)$ **then** $Q(t)$ **else** $R(t)$ est le même que celui de **casetype** $t$ **of** $T_{f=\mathrm{true}} : Q(t)$; $T_{f=\mathrm{false}} : R(t)$ **end** . On pourrait aussi autoriser des instructions de coût non nul dans les fonctions. La transformation consisterait alors à traduire les fonctions en procédures, et les schémas conditionnels en séquences du genre $f(t)$; **casetype** ... **end** . Dans ce cas, il faudrait préciser la sémantique des opérateurs logiques *and* et *or* (évaluation totale ou évaluation optimisée).

Grâce au théorème 8, nous pouvons prouver les théorèmes automatiques 7 et 8, donnant respectivement la probabilité et le rang d'apparition du motif *aaa* dans le langage $L = \{a, b\}^*$. Nous allons prouver les deux théorèmes d'un seul coup. Pour cela, nous écrivons le programme Adl suivant :

```
type L = epsilon | product(lettre,L);  lettre = a | b;  a,b = atom(1);

function contient_aaa(l : L) : boolean;
begin
  case l of
    epsilon : false;
    (a,x) : begin un; apres_a(x) end;
    (b,x) : begin un; contient_aaa(x) end;
  end;
end;

function apres_aa(l : L) : boolean;
begin
  case l of
    epsilon : false;
    (a,x) : begin un; true end;
    (b,x) : begin un; contient_aaa(x) end;
  end;
end;

function apres_a(l : L) : boolean;
begin
  case l of
    epsilon : false;
    (a,x) : begin un; apres_aa(x) end;
    (b,x) : begin un; contient_aaa(x) end;
  end;
end;

procedure proba_aaa(l : L);
begin
  if contient_aaa(l) then vrai
end;

measure un : u;  vrai : v;
```

Nous avons inséré une instruction élémentaire `un` chaque fois qu'un caractère est parcouru, et l'instruction `vrai` lorsque le mot comporte le motif. De plus, nous avons affecté des coûts *symboliques* à ces instructions élémentaires, ce qui va nous permettre de distinguer leur contribution respective.

L'analyse algébrique du programme est réalisée automatiquement par $\Lambda\Upsilon\Omega$ :

```
#analyze "reg";;
```

```
                                          2      2
                                  z (2 u z  + v z  + 2 u z + 2 u)
tau_proba_aaa(z) = -------------------------------------------
                                3      4        2
                               z  + 2 z  + 1 + z  - 3 z
```

Il ne nous reste qu'à séparer les coefficients de $u$ et de $v$ : le coefficient de $v$, soit

$$p(z) = \frac{z^3}{1 - 3z + z^2 + z^3 + 2z^4},$$

est la série associée à la probabilité qu'un mot contienne le motif $aaa$, tandis que le coefficient de $u$, soit

$$l(z) = \frac{2z(1 + z + z^2)}{1 - 3z + z^2 + z^3 + 2z^4},$$

est associé au nombre moyen de lettres lues pour trouver le premier motif $aaa$. Ces séries génératrices étant rationnelles, leur analyse asymptotique est facile [Sal91].

## 4.3 Exemples

Grâce au théorème 8, une large classe de programmes comprenant des fonctions booléennes se prête à une analyse automatique. Cette section montre que par exemple, le test d'occurrence dans un arbre, le test de parité d'une expression sont dans cette classe.

### 4.3.1 Test d'occurrence de symboles dans un arbre

Soit le programme suivant, où la fonction `f` détermine si un arbre binaire (dont les feuilles sont une lettre $x$ ou $y$) contient au moins un $x$. La procédure `p` quant à elle a coût 1 pour chaque arbre contenant au moins un $x$, et 0 pour les arbres sans $x$. Son coût moyen sur les arbres de taille $n$ est la probabilité qu'un tel arbre contienne au moins un $x$.

```
type T = x | y | U;   U = product(T,T);

function f (t : T) : boolean;
begin
  casetype t of
    x : true;
    y : false;
    U : g(t)
  end
end;

procedure p (t : T);
begin
   if f(t) then occurrence
end;

measure occurrence : 1;
```

```
function g (u : U) : boolean
begin
  case u of
    (t1,t2) : f(t1) or f(t2)
  end
end;
```

Les productions obtenues par l'algorithme **Transformation** sont

$$\begin{aligned} T_{f=\text{true}} &\to x \mid U_{g=\text{true}} \\ T_{f=\text{false}} &\to y \mid U_{g=\text{false}} \\ U_{g=\text{true}} &\to T_{f=\text{true}} \times T_{f=\text{true}} \cup T_{f=\text{true}} \times T_{f=\text{false}} \cup T_{f=\text{false}} \times T_{f=\text{true}} \\ U_{g=\text{false}} &\to T_{f=\text{false}} \times T_{f=\text{false}}. \end{aligned}$$

Ici, il n'est pas nécessaire d'utiliser l'algorithme **Réduction** car la transformation n'a pas fait apparaître d'intersection. Ceci s'explique par le fait que les fonctions $f$ et $g$ ne testent qu'une propriété sur chacune des composantes $t$, $t_1$ et $t_2$. Par contre, si on avait $f(t) = g(t)$ and $h(t)$ dans le cas où $t$ est de type $U$, alors les définitions des types conditionnels de $f$ seraient $T_{f=\text{true}} \to x \mid U_{g=\text{true}} \cap U_{h=\text{true}}$ et $T_{f=\text{false}} \to y \mid U_{g=\text{true}} \cap U_{h=\text{false}} \mid U_{g=\text{false}} \cap U_{h=\text{true}} \mid U_{g=\text{false}} \cap U_{h=\text{false}}$.

Par le théorème 8, le programme de $\Pi_{\text{bool}}$ ci-dessus est équivalent au programme suivant de $\Pi$ :

```
type T = T_true | T_false;
     T_true = x | U_true;
     T_false = y | U_false;
     U_true = product(T_true,T_true) | product(T_true,T_false) | product(T_false,T_true);
     U_false = product(T_false,T_false);

procedure p (t : T);
begin
   case t of
      T_true : count;
   end;
end;

measure count : 1;
```

### Analyse de $\Lambda\Upsilon\Omega$ par lui-même

Un exemple amusant de test d'occurrence est le suivant : nous avons montré (théorème 7 page 93) que les spécifications explicites en univers étiqueté conduisent à des séries génératrices exponentielles formées à partir de 1 et $z$ par les opérateurs $+$, $\times$, $Q(f) = 1/(1-f)$, $E(f) = \exp(f)$ et $L(f) = \log(1/(1-f))$ (lorsque le constructeur cycle non orienté n'est pas utilisé). Soit $\mathcal{LI}$ (pour *Labelled Iterative*) l'ensemble de ces fonctions. Certaines des fonctions de $\mathcal{LI}$ correspondent à des spécifications qui ne sont pas bien fondées : la fonction $Q(Q(z))$ provient de la construction $\mathit{sequence}(\mathit{sequence}(\cdot))$ qui génère une infinité d'objets de taille nulle.

Lorsqu'une fonction $f \in \mathcal{LI}$ correspond à une spécification bien fondée, par exemple

$$f = E(Q(z \times E(z)) + z), \tag{4.7}$$

un certain nombre de propriétés de $f$ sont décidables de manière purement *syntaxique* sur l'expression (4.7) [FSZ91]. Par exemple, $f$ est une fonction entière (sans singularité à distance finie) si et seulement si ni $L$ ni $Q$ n'apparaissent dans $f$. Or le test d'occurrence fait partie des problèmes que l'on sait analyser ; il est donc normal que $\Lambda\Upsilon\Omega$ puisse lui-même répondre à ces questions :

*Dans la classe $\mathcal{LI}$, combien y a-t-il de fonctions formées avec $n$ symboles ? Combien de ces fonctions correspondent à des spécifications bien fondées ? Parmi ces dernières, quelle est le pourcentage de fonctions entières ?*

Le programme ADL ci-dessous répond partiellement à ces questions : la classe $\mathcal{LI}$ est décrite, puis la fonction `has_Q_or_L` renvoie *true* pour les fonctions de $\mathcal{LI}$ contenant au moins l'un des opérateurs $Q$ et $L$, et *false* sinon :

```
type LI = One | Z | Plus(LI,LI) | Times(LI,LI) | Q_op(LI) | Exp(LI) | L_op(LI);
     One,Z,Plus,Times,Q_op,Exp,L_op = atom(1);

function has_Q_or_L (f : LI) : boolean;
begin
   case f of
      One : false;
      Z : false;
      Plus(g,h) : if has_Q_or_L(g) then true else has_Q_or_L(h);
      Times(g,h) : if has_Q_or_L(g) then true else has_Q_or_L(h);
      Q_op(g) : true;
      Exp(g) : has_Q_or_L(g);
      L_op(g) : true;
   end;
end;

procedure status (f : LI);
begin
   if has_Q_or_L(f) then yes else no
end;

measure no : 1;
```

REMARQUE : Nous avons utilisé ici le schéma **if** ... **then** ... **else** *dans une fonction*, ce qui est interdit dans la classe $\Pi_{\text{bool}}$. Cependant, ce schéma s'exprime à l'aide des opérateurs logiques *and*, *or* et *not* :

$$\textbf{if } a \textbf{ then } b \textbf{ else } c \Rightarrow (f \text{ and } g) \text{ or } ((\text{not } f) \text{ and } h)$$

et réciproquement :

$$\begin{array}{rcl} a \text{ and } b & \Rightarrow & \textbf{if } a \textbf{ then } b \textbf{ else } \text{false} \\ a \text{ or } b & \Rightarrow & \textbf{if } a \textbf{ then } \text{true} \textbf{ else } b \\ \text{not } a & \Rightarrow & \textbf{if } a \textbf{ then } \text{false} \textbf{ else } \text{true}. \end{array}$$

La classe $\Pi_{\text{bool}}$ peut donc être définie indifféremment à l'aide des opérateurs logiques *and*, *or*, *not* comme nous l'avons fait, ou bien à l'aide du schéma logique **if** $a$ **then** $b$ **else** $c$.

L'analyse (faite par $\Lambda\Upsilon\Omega$) du programme ci-dessus conduit au résultat suivant :

**Théorème automatique 9.** *Le nombre de fonctions de la classe $\mathcal{LI}$ ayant $n$ symboles est*

$$LI_n = [z^n]\frac{1-3z-\sqrt{1-6z-7z^2}}{4z} = 2z + 6z^2 + 26z^3 + 126z^4 + 658z^5 + 3606z^6 + O(z^7),$$

*et vaut asymptotiquement*

$$LI_n = \sqrt{\frac{7}{8\pi}}\frac{7^n}{n^{3/2}} + O(\frac{7^n}{n^2}).$$

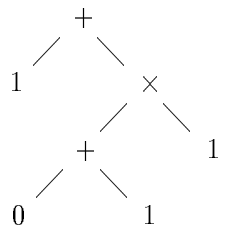
*La probabilité qu'une fonction de $\mathcal{LI}$ ayant $n$ symboles ne comporte ni $Q$ ni $L$ est*

$$p_n = \frac{[z^n]\frac{1-z-\sqrt{1-2z-15z^2}}{4z}}{LI_n} = \left(\frac{5}{7}\right)^{n+1/2} + O\left(\left(\frac{5}{7}\right)^n n^{-1/2}\right).$$

On pourrait aussi s'intéresser aux fonctions dans lesquelles la variable $z$ apparaît au moins une fois, ou à celles qui s'annulent en $z = 0$, afin que l'application les opérateurs $E$, $L$ et $Q$ soit bien fondée. En fait, il est possible d'écrire dans le langage ADL la vérification du caractère bien fondé des fonctions de $\mathcal{LI}$.

### 4.3.2 Parité d'expressions arithmétiques aléatoires

Soit l'ensemble $\mathcal{N}$ des expressions arithmétiques formées à partir des entiers 0 et 1, et des opérateurs binaires + et $\times$. Nous définissons la taille d'une telle expression comme le nombre d'opérateurs. Par exemple, l'expression $1 + ((0 + 1) \times 1)$ dont la représentation arborescente est




fait partie de $\mathcal{N}$, sa taille est 3, et sa valeur est 2. Savoir comment varie la valeur d'une expression en fonction de sa taille peut s'avérer utile. En particulier, on peut se poser la question suivante :

*Quel est le pourcentage des expressions de $\mathcal{N}$ de taille $n$ qui représentent un entier pair ?*

Le programme Adl ci-dessous formalise cette question en termes plus informatiques : la fonction `pair` détermine si une expression est paire ou non, et la procédure `proba_pair` a coût 1 lorsqu'une expression est paire, et 0 sinon. Comme dans l'exemple précédent, le coût moyen de la procédure `proba_pair` est le pourcentage d'expressions vérifiant une certaine propriété, ici avoir une valeur paire.

```
type N = zero | un | plus(N,N) | times(N,N);
     zero,un = atom(0);
     plus,times = atom(1);

function pair (n : N) : boolean;
begin
   case n of
      zero : true;
      un : false;
      plus(i,j) : if pair(i) then pair(j)
                  else if pair(j) then false else true;
      times(i,j): if pair(i) then true else pair(j)
   end
end;

procedure proba_pair (n : N);
```

```
begin
    if pair(n) then count
end;

measure count : 1;
```

Le programme ci-dessus illustre bien le fait que les fonctions sont définies indépendamment des procédures, et que ces dernières utilisent les fonctions dans des schémas conditionnels du genre **if** $f(t)$ **then** ... **else** ...

**Théorème semi-automatique 10.** *La probabilité qu'une expression de taille $n$ de $\mathcal{N}$ soit paire est*

$$p_n = \frac{[z^n]\frac{2-\sqrt{2}\ \sqrt{1+\sqrt{1-16z}}}{4z}}{[z^n]\frac{1-\sqrt{1-16z}}{4z}},$$

*et vérifie asymptotiquement*

$$p_n = \frac{1}{\sqrt{2}} + O(1/\sqrt{n}).$$

Ce théorème n'est pas entièrement automatique, car la phase de résolution des équations nécessite quelqu'aide manuelle, comme nous allons le constater.

**Démonstration :** L'analyse algébrique du programme ci-dessus conduit pour la série génératrice ordinaire $N$ et pour le descripteur de complexité $\tau P$ de la procédure `tau_pair` aux expressions suivantes (après résolution par Maple).

$$N(z) = \frac{1-\sqrt{1-16z}}{4z} \tag{4.8}$$

$$\tau P(z) = \frac{r(z) - 1 - z r(z)^2}{2 z r(z)} \text{ où} \tag{4.9}$$

$$r(z) = \mathit{RootOf}(Y^2 + Y - 3zY^3 - 8zY^2 - 2 - 6z^2Y^4 + \sqrt{1-16z}\ Y^2 - \sqrt{1-16z}\ Y - \sqrt{1-16z}\ Y^3 z).$$

Ici, la notation $r(z) = \mathit{RootOf}(F(Y,z))$ signifie que $r(z)$ est solution en $Y$ de l'équation $F(Y,z) = 0$. L'analyse asymptotique de cette forme pour $\tau P(z)$ n'est pas possible directement, dans l'état actuel du système $\Lambda\Upsilon\Omega$. Il nous faut donc mener quelques calculs "à la main", ce qui nous permettra de mieux comprendre le procédé d'analyse asymptotique.

**Recherche de singularité** : nous savons par avance que la probabilité cherchée $p_n$ est bornée par 1 lorsque $n$ tend vers l'infini. Par conséquent, la singularité dominante réelle de $\tau P(z)$ est supérieure ou égale à celle de $N(z)$. L'étude de l'expression (4.9) révèle qu'en réalité, c'est la même, soit $\rho = 1/16$.

**Développement au voisinage de la singularité** : nous commençons par développer $r(z)$ autour de $z = \rho$. Pour cela, nous faisons le changement de variable $1 - 16z = Z$, ce qui permet de factoriser l'expression "sous le *RootOf*" :

$$r(z) = \mathit{RootOf}((3Y^2Z + 16 + 16\sqrt{Z}\ Y - 3Y^2)(Y^2Z + 16 - 8Y - 8\sqrt{Z}\ Y - Y^2)).$$

La fonction $r(z)$ est racine de l'un des deux trinômes en $Y$ ; il y a en tout quatre solutions possibles :

$$Y_1 = \frac{4\sqrt{Z+3}\ -8\sqrt{Z}}{3Z-3}, \qquad Y_2 = \frac{-8\sqrt{Z}\ -4\sqrt{Z+3}}{3Z-3},$$
$$Y_3 = \frac{4+4\sqrt{Z}\ +4\sqrt{2}\ \sqrt{1+\sqrt{Z}}}{Z-1}, \quad Y_4 = \frac{4+4\sqrt{Z}\ -4\sqrt{2}\ \sqrt{1+\sqrt{Z}}}{Z-1},$$

qui correspondent aux quatre branches de l'équation du quatrième degré donnant $r(z)$.

**Détermination de la "bonne" branche** : pour déterminer laquelle des solutions $Y_i$ est la bonne, nous procédons par élimination. Nous savons en effet que la solution $\tau P(z)$ est une série entière à coefficients positifs. Nous remplaçons donc $r(z)$ par chacun des $Y_i$ dans (4.9), et nous calculons les premiers termes du développement de Taylor en $z = 0$.

$$\begin{aligned}
Y_1 &: \frac{1}{z} - 2 - 6z - 44z^2 - 430z^3 + O(z^4) \\
Y_2 &: \frac{1}{3z} - \frac{2}{3} - \frac{14z}{3} - \frac{124z^2}{3} - \frac{1270z^3}{3} - \frac{14308z^4}{3} + O(z^5) \\
Y_3 &: \frac{1}{z} - 1 - 5z - 42z^2 - 429z^3 - 4862z^4 + O(z^5) \\
Y_4 &: 1 + 5z + 42z^2 + 429z^3 + O(z^4).
\end{aligned}$$

Les trois premières solutions ne conviennent pas car elles ne correspondent pas à des séries formelles (de plus certains de leurs coefficients sont négatifs et même fractionnaires pour $Y_2$). La solution correcte est donc $Y_4$, et elle donne après simplification des radicaux

$$\tau P(z) = \frac{2 - \sqrt{2}\,\sqrt{1 + \sqrt{1-16z}}}{4z}. \tag{4.10}$$

Maintenant que nous avons une forme simple pour $\tau P(z)$, nous pouvons utiliser le programme `equivalent` de B. Salvy pour calculer les développements asymptotiques :

$$\begin{aligned}
[z^n]\tau P(z) &= \frac{\sqrt{2}\ 16^n}{\sqrt{\pi}\ n^{3/2}} + O(\frac{16^n}{n^2}) \\
[z^n]N(z) &= \frac{2\ 16^n}{\sqrt{\pi}\ n^{3/2}} + O(\frac{16^n}{n^2}).
\end{aligned} \tag{4.11}$$

La division des deux développements donne $p_n = 1/\sqrt{2} + O(1/\sqrt{n})$ comme annoncé. ■

L'étape "manuelle" de cette preuve est le passage de l'expression (4.9) à (4.10) pour le descripteur de complexité $\tau P(z)$. Si on les étudie de près, on constate que les trois phases de cette étape (recherche de la singularité, développement au voisinage de la singularité et détermination de la bonne branche) peuvent être automatisées. Cet exemple montre le besoin d'un simplificateur d'expressions de séries génératrices, sachant calculer les singularités, résoudre les équations par radicaux (éventuellement après changement de variable), et éliminer les solutions ne correspondant pas à des séries génératrices.

## 4.4 Généralisation et conclusions

Les trois premières sections de ce chapitre ont décrit l'analyse d'algorithmes dans l'extension de la classe II par des fonctions booléennes. L'exposé peut se faire en réalité avec n'importe quelle type de fonction à ensemble fini de valeurs $\mathcal{V} = \{v_1, \ldots, v_k\}$ ($\{\text{true}, \text{false}\}$ dans le cas booléen). La définition des fonctions suppose alors qu'il existe des opérateurs $\Psi : \mathcal{V} \times \cdots \times \mathcal{V} \to \mathcal{V}$ entre les valeurs (*and, or, not* dans le cas booléen). Le schéma conditionnel s'écrit

$P(a:A) :=$ **if** $f(a) = v_j$ **then** $Q(a)$ **else** $R(a)$.

L'algorithme **Transformation** se transpose facilement à un nombre fini de valeurs : il est basé sur le fait que chaque expression peut s'écrire $F(g_1(t), \ldots, g_l(t))$, et par conséquent $\mathcal{T}_{F=v_j}$ se définit à partir d'un nombre fini d'intersections de $T_{g_i=v_{\alpha_i}}$.

L'algorithme **Réduction**, quant à lui, ne dépend pas des booléens *true* et *false*, et peut être utilisé tel quel pour n'importe quelle spécification issue de l'algorithme **Transformation**.

Ainsi, l'analyse de tout programme d'une classe $\Pi$ étendue par des fonctions à nombre fini de valeurs se ramène à l'analyse d'un programme de $\Pi$. Par exemple, en prolongeant l'exemple de la section 4.3.2, on pourrait déterminer le pourcentage d'expressions arithmétiques dont la valeur est congrue à $j$ modulo $k$, pour tout couple d'entiers $(j,k)$ (la seule difficulté serait l'écriture du programme Adl).

Le résultat principal de ce chapitre, à savoir le théorème 8, a deux interprétations possibles : l'une pessimiste, l'autre optimiste. L'interprétation pessimiste consiste à dire que les programmes de $\Pi_{\text{bool}}$ ne permettent pas d'exprimer des propriétés plus complexes sur la classe $\Omega$ que ceux de $\Pi$ (néanmoins la formulation de ces propriétés est plus lisible et plus compacte dans $\Pi_{\text{bool}}$). L'interprétation optimiste, quant à elle, est que les programmes de $\Pi_{\text{bool}}$ génèrent la même classe de séries génératrices que ceux de $\Pi$, donc ils ne sont pas plus difficiles à analyser.

En fait, les fonctions de la classe $\Pi_{\text{bool}}$ forment des automates finis, sans mémoire. Dans le cas des structures de données linéaires (comme par exemple les mots du langage $\{a,b\}^*$ introduit au début de ce chapitre), de tels automates caractérisent des langages *réguliers*. Le théorème 8 ne fait alors qu'exprimer le fait que les langages réguliers sur un alphabet fini sont clos par union, intersection, et passage au complémentaire [AU72, théorème 2.8 page 129]. Les séries génératrices sont alors rationnelles, de même que les densités asymptotiques, lorsqu'elles existent.

Dans le cas général de la classe $\Omega$, nous dirons par extension que les fonctions de $\Pi_{\text{bool}}$ calculent des propriétés *régulières* sur les structures de données. On sait déjà que les langages *context-free* sont clos par intersection avec les langages réguliers [AU72, théorème 2.26 page 197], autrement dit que les propriétés régulières de structures de données *context-free* restent *context-free*. Nous obtenons alors des séries génératrices algébriques, et des densités asymptotiques (toujours lorsqu'elles existent) également algébriques (par exemple la densité d'expressions arithmétiques paires est $1/\sqrt{2}$). Le théorème 8 généralise cette propriété de clôture avec les langages réguliers à la classe $\Omega$ définie au chapitre 1.

En un sens, le théorème 8 est optimal pour la classe $\Omega$ : du fait que les langages *context-free* ne sont pas clos par intersection ou passage au complémentaire, on ne peut pas espérer pouvoir analyser des fonctions calculant des propriétés *context-free*. En effet, si $f$ est une fonction caractéristique des mots de la forme $\{a^n b^n c^i\}$, et $g$ une fonction caractéristique des mots de la forme $\{a^i b^n c^n\}$, la fonction $f$ *and* $g$ caractérise le langage $\{a^n b^n c^n\}$, qui ne peut pas être décrit dans $\Omega$.

Enfin, les résultats de ce chapitre suggèrent l'extension par des schémas booléens sur les multi-constructeurs : le schéma **forall** $b$ **in** $a$ **do** $f(b)$ teste si une propriété est vraie pour tous les éléments, tandis que le schéma **forone** $b$ **in** $a$ **do** $f(b)$ teste si l'un au moins des éléments la vérifie.

# Chapitre 5

# Études

*Une mouche ne doit pas tenir dans la tête d'un naturaliste plus de place qu'elle n'en tient dans la nature.*

Buffon

Un système d'analyse automatique d'algorithmes comme $\Lambda\Upsilon^\Omega$ peut être utilisé de plusieurs façons : pour résoudre des problèmes simples de façon entièrement automatique, pour résoudre en partie des problèmes plus complexes, ou encore comme outil de recherche. Les exemples contenus dans les chapitres précédents sont pour la plupart des problèmes simples, que $\Lambda\Upsilon^\Omega$ a pu résoudre entièrement (d'où la dénomination *théorème automatique*). Ce chapitre présentera des problèmes plus difficiles, pour lesquels néanmoins une partie de l'analyse (algébrique ou asymptotique) a pu être faite de façon automatique. En ce qui concerne l'utilisation comme outil de recherche, il s'agit principalement d'une utilisation interactive. Citons l'exemple suivant : $\Lambda\Upsilon^\Omega$ a montré que le coût en moyenne de la dérivation seconde est $\sim C_2 n^2$, et que celui de la dérivation troisième est $\sim C_3 n^{5/2}$, ce qui nous à suggéré que celui de la dérivation $k$-ième est $\sim C_k n^{1+k/2}$. Un tel système peut ainsi servir à "deviner" des lois générales, dans des situations où le caractère pénible des calculs et les possibilités d'erreur découragent à l'expérimentation.

Nous montrons dans ce chapitre des exemples typiques d'études que l'on peut effectuer en se servant de $\Lambda\Upsilon^\Omega$ comme outil d'analyse : le nombre moyen de records dans une permutation de $n$ éléments est $\log n + \gamma + O(1/n)$ (section 5.1) ; pour minimiser le nombre de comparaisons dans le tri par *QuickSort*, il faut utiliser le tri par insertion quand le nombre d'éléments devient inférieur ou égal à 8 (section 5.2) ; la valeur moyenne des expressions arithmétiques définies à la fin du chapitre précédent est $\sim \sqrt{\pi/2}/\Gamma(3/4)n^{1/4}$ (section 5.3) ; le nombre de colliers antisymétriques de $2k$ perles est $\sim 2^{n-1}/n$ (section 5.4) ; le nombre de parenthésages ternaires commutatifs de $X^{2n+1}$ est $\sim (0.52...)n^{-3/2}\alpha^n$ où $\alpha \simeq 2.82$ (section 5.5) ; enfin le nombre d'alcanes de formule brute $C_n H_{2n+2}$ est $\sim (0.66...)n^{-5/2}\alpha^n$ pour le même $\alpha$ (section 5.6).

Les différentes études sont classées par ordre de difficulté croissante : les premières sont entièrement automatiques (en univers étiqueté, les opérateurs sont plus simples), tandis que les dernières requièrent une large part d'intervention humaine pour le traitement asymptotique des opérateurs de Pólya introduits par les cycles, ensembles et multi-ensembles en univers non étiqueté. Nous avons aussi voulu mettre en évidence des extensions possibles du système $\Lambda\Upsilon^\Omega$, par rapport à son état actuel. Les quatre premières études s'intéressent plus particulièrement à l'analyse algébrique : l'étude sur les arbres tournois indique des règles de transformation de programme permettant de se

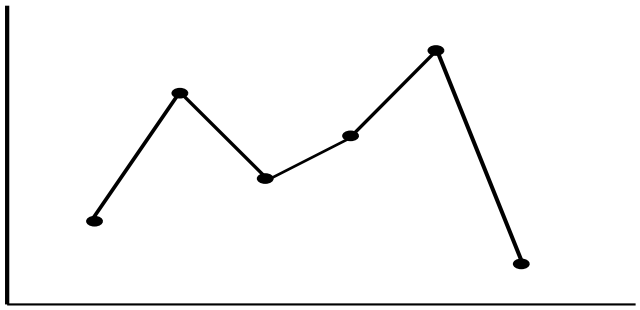


Figure 5.1 : Le graphe de la permutation [2 5 3 4 6 1].

ramener à la classe II, l'étude comparée du tri *QuickSort* et du tri par insertion dégage la nécessité d'analyses avec paramètres, l'étude de la valeur moyenne des expressions arithmétiques établit la possibilité d'analyse de coûts non additifs, l'étude des colliers bicolores montre l'intérêt des séries à plusieurs variables et du procédé de marquage. Les deux dernières études (alcanes et parenthésages ternaires commutatifs) sont de nature principalement analytique, et mettent en évidence des méthodes numériques pour obtenir un développement asymptotique à partir d'équations non algébriques.

## 5.1 Descentes, records et autres spécialités

Cette section est consacrée à des problèmes liés à l'opérateur d'enracinement du minimum introduit au chapitre 3 (section 3.4). Les quatre exemples de cette section analysent des paramètres différents d'une même de structure de données, en l'occurrence des arbres "tournois". L'arbre tournoi est une représentation interne d'une file de priorité qui conserve l'ordre d'arrivée, avec accès au minimum et suppression du minimum en temps constant, insertion d'un nouvel élément en temps moyen $O(\log n)$.

Les trois premiers exemples sont des analyses effectuées entièrement par $\Lambda\Upsilon\Omega$, à l'aide des règles du chapitre 3 pour l'analyse algébrique ; on y étudie les différents types de nœuds d'un arbre tournoi (section 5.1.1), la longueur de la branche gauche d'un arbre tournoi (c'est-à-dire le nombre de records d'une permutation, section 5.1.2), ainsi que le nombre de nœuds internes des arbres tournois généraux (section 5.1.3). Le dernier exemple est "semi-automatique", dans la mesure où l'on utilise d'abord une règle de transformation de programme (qui n'est pas implantée dans la version actuelle de $\Lambda\Upsilon\Omega$), pour se ramener à un programme dont l'analyse est automatique (section 5.1.4).

### 5.1.1 Descentes dans une permutation

Soit une permutation $\phi$ des entiers de 1 à $n$, représentée par la suite des valeurs $[\phi(1)\ldots\phi(n)]$. Par exemple, la permutation $1 \to 2, 2 \to 1, 3 \to 3$ est notée [2 1 3]. Dans une telle permutation, une *descente* est une séquence $\phi(i)\phi(i+1)$ telle que $\phi(i) > \phi(i+1)$. Par exemple, la permutation [2 5 3 4 6 1] comporte deux descentes : 5 3 et 6 1. Sur le graphe de $\phi$ (voir figure 5.1), les descentes se manifestent par une pente négative, d'où leur nom. Le problème que nous nous posons est le suivant :

*Quel est le nombre moyen de descentes dans une permutation de $n$ entiers ?*

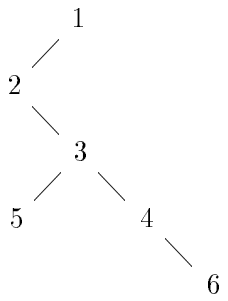
Avant de répondre à cette question, nous introduisons l'arbre tournoi associé à une permutation, sur lequel nous pourrons déterminer visuellement les descentes.

### Arbre tournoi associé à une permutation

A une séquence de nombres distincts $\alpha_1 \dots \alpha_n$ est associé un arbre binaire, dit *arbre tournoi*, de la façon suivante [Vui80] :

- à la racine est placé l'élément minimum $\alpha_j = \inf\{\alpha_i\}$,
- la branche gauche (resp. droite) est l'arbre tournoi associé à la séquence des nombres à gauche (resp. droite) de $\alpha_j$.

De façon naturelle, l'arbre tournoi associé à une permutation $\phi$ de $[1 \dots n]$ est celui qui est associé à la séquence $\phi(1) \dots \phi(n)$. Par exemple, l'arbre tournoi associé à [253461] est




Un arbre tournoi comporte quatre types de nœuds, suivant que le sous-arbre gauche et/ou droite est vide ou non :

1. les feuilles (aucun sous-arbre),
2. les nœuds *gauches* (un seul sous-arbre à gauche),
3. les nœuds *droits* (un seul sous-arbre à droite),
4. les nœuds *complets* (deux sous-arbres).

Soit un nœud de l'arbre tournoi, étiqueté par l'entier $i$ ; l'entier $j$ immédiatement à gauche de $i$ dans la permutation est soit dans le sous-arbre gauche partant de $i$, soit au-dessus vers la racine. Dans le premier cas, $i < j$ est $i$ est le second élément d'une descente. Dans le second cas, $j < i$ et $j\,i$ est une montée. Le nombre de descentes d'une permutations est donc exactement le nombre de nœuds gauches ou complets de l'arbre tournoi associé. A l'aide de $\Lambda\Upsilon^{\Omega}$, nous allons étudier les statistiques des quatre types de nœuds, et nous en déduirons le nombre moyen de descentes.

**Théorème automatique 11.** *Dans un arbre tournoi aléatoire de $n$ nœuds, la répartition asymptotique des nœuds est la suivante :*

$$\begin{array}{ll} \textit{nœuds feuilles} : n/3 + O(1) & \textit{nœuds complets} : n/3 + O(1) \\ \textit{nœuds gauches} : n/6 + O(1) & \textit{nœuds droits} : n/6 + O(1). \end{array}$$

**Démonstration :** La preuve est constituée par l'analyse du programme suivant. La spécification des arbres tournois est familière (cf section 3.4). La procédure `stat` parcourt un arbre tournoi `t`, et appelle pour chaque type de nœud une instruction élémentaire spéciale. Les coûts sont symboliques pour pouvoir distinguer les nœuds.

```
type T = fin | product(T,min(cle),T);
     fin = Latom(0);
     cle = Latom(1);

procedure stat (t : T);
begin
  case t of
    fin : rien;
    (t1,k,t2) : begin
                  stat(t1); stat(t2);
                  case t1 of
                    fin       : case t2 of
                                  fin       : noeud_feuille;
                                  otherwise : noeud_droit;
                                end;
                    otherwise : case t2 of
                                  fin       : noeud_gauche;
                                  otherwise : noeud_complet;
                                end
                  end
                end
  end
end;

measure noeud_feuille:f; noeud_droit:d; noeud_gauche:g; noeud_complet:c; rien:0;
```

Les règles principales qui ont été utilisées ici sont la règle 41 (enracinement du minimum) et la règle 43 (descente dans un produit partitionnel avec enracinement du minimum), et les équations différentielles obtenues pour $T(z)$ et $\tau\mathrm{stat}(z)$ sont

$$\begin{aligned} T(z) &= 1 + \int_0^z T(u)^2 du \\ \tau\mathrm{stat}(z) &= 2\int_0^z \tau\mathrm{stat}(u)T(u)du + fz + (d+g)\int_0^z \int_0^u T(v)^2 dvdu + c\int_0^z \left(\int_0^u T(v)^2 dv\right)^2 du \end{aligned}$$

Les intégrales doubles proviennent du fait que l'on descend de deux niveaux, dans le nœud se trouvant à la racine et dans l'un des sous-arbres `t1` ou `t2`. Ces équations se résolvent en

$$T(z) = \frac{1}{1-z}, \quad \tau\mathrm{stat}(z) = f\frac{z(3-3z+z^2)}{3(1-z)^2} + (g+d)\frac{z^2(3-2z)}{6(1-z)^2} + c\frac{z^3}{3(1-z)^2}.$$

Comme le coefficient de $z^n$ dans $T(z)$ est 1, le nombre moyen de nœud de chaque type est donné directement par le développement asymptotique de $\tau\mathrm{stat}(z)$ :

```
> equivalent(f*z*(3-3*z+z^2)/3/(1-z)^2+(g+d)*z^2*(3-2*z)/6/(1-z)^2+c*z^3/3/(1-z)^2,z);
                ((1/3 f + 1/6 g + 1/6 d + 1/3 c) n) + (O(1))
```

■

En additionnant le nombre de nœuds gauches et de nœuds complets, on trouve que le nombre moyen de descentes dans une permutation est $n/2 + O(1)$.

### 5.1.2 Records

Dans la représentation $[\phi(1)\ldots\phi(n)]$ d'une permutation, un *record* est un élément qui est plus petit que *tous* ceux qui sont à sa gauche (alors que pour les descentes, on considère seulement l'élément immédiatement à gauche). Par défaut, le premier élément est toujours un record. Par exemple, dans [253461], les records sont 2 et 1. Tous les records sauf le premier sont aussi le second élément d'une descente, et par suite : $nb(records) \leq 1 + nb(descentes)$.

Sur l'arbre tournoi associé à la permutation, les records sont les éléments de la branche gauche, et leur ordre d'apparition est de bas en haut. Le lecteur vérifiera cette affirmation sur l'arbre tournoi de la permutation [253461]. Comme pour les descentes, nous nous sommes ramenés à l'étude d'un paramètre (ici la longueur de la branche gauche) sur une structure combinatoire équivalente aux permutations (les arbres tournois).

**Lemme 11.** *Les records d'une permutation sont les éléments qui se trouvent dans la branche gauche de l'arbre tournoi associé.*

**Démonstration :** Soit $n$ le nombre d'éléments de la permutation. Pour $n = 0$, le lemme est vrai. Pour $n > 1$, les records sont le plus petit élément de la permutation (qui se trouve à la racine), et les records de la séquence des éléments à gauche de celui-ci : en effet, les éléments à droite du minimum ne peuvent pas être des records. Par induction, le lemme est vrai pour tout $n$. ■

**Théorème automatique 12.** *La longueur moyenne de la branche gauche d'un arbre tournoi de $n$ nœuds est le coefficient de $z^n$ dans*

$$\frac{1}{1-z}\log\frac{1}{1-z}.$$

*Asymptotiquement, ce coefficient vaut*

$$[z^n]\frac{1}{1-z}\log\frac{1}{1-z} = \log n + \gamma + O(1/n)$$

*où $\gamma \simeq .5772156649$ est la constante d'Euler.*

En fait, le coefficient de $z^n$ dans $\frac{1}{1-z}\log\frac{1}{1-z}$ vaut exactement $H_n = 1 + 1/2 + \cdots + 1/n$.
**Démonstration :** La preuve est constituée par l'analyse automatique du programme ci-dessous, effectuée par le système $\Lambda\Upsilon^{\Omega}$.

```
type T = fin | product(T,min(cle),T);
     fin = Latom(0);
     cle = Latom(1);

procedure longueur_branche_gauche (t : T);
begin
   case t of
      fin : zero;
      (t1,k,t2) : begin un; longueur_branche_gauche(t1) end
   end
end;

measure un:1; zero:0;
```

Ainsi définie, la procédure `longueur_branche_gauche` compte le nombre de nœuds de la branche gauche. Les principales règles utilisées sont les mêmes que pour la détermination des différents types de nœuds, à savoir la règle 41 (enracinement du minimum) et la règle 43 (descente dans un produit partitionnel avec enracinement du minimum). L'équation différentielle vérifiée par le descripteur de complexité $\tau L(z)$ de la procédure `longueur_branche_gauche` est

$$\tau L(z) = \int_0^z T(u)^2 du + \int_0^z \tau L(u) T(u) du,$$

dont la solution est

$$T(z) = \frac{1}{1-z} \quad \text{et} \quad \tau L(z) = \frac{1}{1-z} \log \frac{1}{1-z}.$$

Puis le programme de B. Salvy donne (toujours automatiquement) le coût moyen :

```
Average cost for longueur_branche_gauche on random inputs of size n is:

(ln(n)) + (gamma) + (O(1/n))
```

■

Le nombre moyen de records dans une permutation aléatoire de $n$ entiers distincts est par conséquent $\log n + O(1)$.

### 5.1.3 Nœuds internes des arbres tournois généraux

Les arbres tournois généraux sont des arbres *généraux* (l'arité de chaque nœud est variable), étiquetés, et tels que dans tout sous-arbre, la plus petite étiquette se trouve à la racine (propriété de tournoi). Ces arbres sont aussi appelés *recursive trees* (arbres récursifs) par Meir et Moon [MM78]. La question que nous nous posons est :

*Quel est le nombre moyen de nœuds internes des arbres tournois généraux ?*

**Théorème automatique 13.** *Le nombre moyen de nœuds internes des arbres tournois généraux de $n$ nœuds est égal au rapport des coefficients de $z^n$ de $\tau P(z)$ et de $T(z)$, où*

$$\tau P(z) = \frac{1-2z}{3} + \frac{3z-1}{3\sqrt{1-2z}}, \qquad T(z) = 1 - \sqrt{1-2z}\ ,$$

*et vaut asymptotiquement $n/3 + O(1)$.*

**Démonstration :** Ce théorème se prouve par l'analyse du programme Adl suivant, où la procédure `inodes` ajoute un coût unitaire pour chaque nœud interne rencontré.

```
type T = cle | min(cle) sequence(T,card>=1);
     cle = Latom(1);

procedure inodes (t : T);
begin
   casetype t of
      cle : nil;
      otherwise : begin
                     count;
```

```
                case t of
                   (c,l) : forall u in l do inodes(u)
                end
             end
    end
end;

measure count:1;
```

Les règles principales utilisées sont encore la règle 41 (enracinement du minimum) et la règle 43 (descente dans un produit partitionnel avec enracinement du minimum), plus la règle 34 (itération dans un complexe partitionnel). Les équations calculées par $\Lambda\Upsilon\Omega$ sont

$$T(z) = z + \int_0^z \frac{T(u)}{1-T(u)}du, \qquad \tau\text{inodes}(z) = \int_0^z \frac{T(u)}{1-T(u)}du + \int_0^z \frac{\tau\text{inodes}(u)}{(1-T(u))^2}du.$$

La série génératrice $T(z)$ diffère de celle des arbres tournois binaires définis auparavant :

$$T(z) = 1-\sqrt{1-2z}\ , \qquad \tau\text{inodes}(z) = \frac{1-2z}{3} + \frac{3z-1}{3\sqrt{1-2z}}.$$

L'analyse asymptotique de ces fonctions est menée à bien par le programme `equivalent` :

```
Number of arguments of inodes of size n is n! times:

                (- 1 + n)                  n              n
               2                          2              2
              (----------) + (3/16 ----------) + (O(----))
                 1/2  3/2                1/2  5/2         7/2
                Pi    n                Pi    n          n


Total cost of inodes on all arguments of size n is n! times:

                      n                       n              n
                     2           11          2              2
           (1/6 ----------) + (---- ----------) + (O(----))
                  1/2  1/2      48     1/2  3/2         5/2
                 Pi    n             Pi    n          n


Average cost for inodes on random inputs of size n is:

                          (1/3 n) + (1/3) + (O(1/n))
```

La première ligne donne le nombre d'arbres généraux de taille $n$, la seconde le coût total de la procédure `inodes` sur ces arbres, et la troisième est le coût moyen, obtenu par simple division. ■

### 5.1.4 Pagodes

L'arbre tournoi associé à une permutation peut se construire de façon incrémentale, en insérant un à un les éléments de la permutation, pris de gauche à droite. L'insertion d'un nouvel élément $n$ dans un arbre tournoi $t$ se fait de la manière suivante :

- si $t$ est l'arbre vide *nil*, le résultat de l'insertion est l'arbre dont la racine est $n$, et les deux branches sont *nil*,
- sinon, soit $m$ l'entier à la racine de $t$ ; si $n$ est plus petit que $m$, on crée un nouveau nœud dont $n$ est à la racine, $t$ est la branche gauche, et la branche droite est *nil*,
- sinon, lorsque $n$ est plus grand que $m$, on insère $n$ dans la branche droite de $t$.

Cet algorithme se décrit très facilement en Adl :

```
type T = nil | product(T,min(N),T);
     nil = Latom(0);
     N = Latom(1);

procedure insert (n : N; t : T);
begin
   case t of
      nil            : (nil,n,nil);
      (left,m,right) : if n<m then (t,n,nil) else (left,m,insert(n,right));
   end;
end;
```

Le principe de cet algorithme consiste à descendre dans la branche droite de $t$ jusqu'à rencontrer un élément plus grand que $n$ (ou la fin de la branche).

L'expérience montre que l'insertion a lieu en moyenne vers le bas de la branche droite. Plus précisément, on sait que la branche droite à pour profondeur moyenne $H_n$ (théorème automatique 12) ; Françon, Viennot et Vuillemin ont montré que le nombre de comparaisons nécessaires pour l'insertion dans un arbre tournoi de taille $n$ vaut en moyenne $H_{n+1} - 1/2$ [FVV78, proposition 8]. Ces auteurs ont par conséquent imaginé une nouvelle structure de donnée, la "pagode", où la racine pointe sur le bas des branches gauche et droite, et chaque nœud à un pointeur vers son père. Ainsi, le coût moyen d'insertion devient constant.

Notre but ici est de retrouver le nombre moyen de comparaisons $H_{n+1} - 1/2$ par analyse algébrique. Pour compter les comparaisons, nous rajoutons dans le corps de la procédure `insert` une instruction de coût 1 avant chaque comparaison :

```
(left,m,right) : begin count; if n<m then (t,n,nil) else (left,m,insert(n,right)) end;

measure count : 1;
```

La contribution de l'opération élémentaire `count` au descripteur de complexité $\tau I(z)$ de la procédure `insert` est $N(z)\int_0^z T(u)N'(u)T(u)du$ (règles 24, 29 et 41).

D'autre part, l'instruction `if n<m then (t,n,nil) else (left,m,insert(n,right))` se réécrit en

```
case (n,m) of
   (min(n),m) : (t,n,nil);
   (n,min(m)) : (left,m,insert(n,right))
end:
```

en utilisant l'égalité $\mathcal{N} \times_p (\mathcal{T} \times_p \mathcal{N}^\square \times_p \mathcal{T}) = \mathcal{N}^\square \times_p (\mathcal{T} \times_p \mathcal{N}^\square \times_p \mathcal{T}) \mid \mathcal{N} \times_p \mathcal{T} \times_p \mathcal{N}^\square \times_p \mathcal{T}$. Cette transformation est un cas particulier du schéma de *test du minimum* introduit à la section 3.4.3 page 101.

La contribution de l'appel `insert(n,right)` est par suite $\int_0^z T(u)N'(u)\tau I(u)du$. Sachant que $N(z) = z$ et $T(z) = 1/(1-z)$, nous dérivons pour la série $\tau I(z)$ l'équation

$$\tau I(z) = z\int_0^z \frac{1}{(1-u)^2}du + \int_0^z \frac{\tau I(u)}{1-u}du,$$

équation que Maple résout facilement

$$\tau I(z) = \frac{1}{1-z}\log\frac{1}{1-z} + \frac{z^2-2z}{2(1-z)}.$$

Il faut ici noter que l'argument de la procédure `insert` n'est pas uniquement un arbre tournoi $t$, mais le produit (partitionnel) d'un élément $n$ et d'un arbre tournoi $t$. La série génératrice exponentielle comptant les arguments d'`insert` est donc la série $N(z)T(z) = z/(1-z)$, dont les coefficients valent 1 pour $n > 0$.

**Théorème semi-automatique 14.** *Le nombre moyen de comparaisons pour l'insertion d'un élément dans un arbre tournoi de $n-1$ éléments est*

$$\overline{\tau I}_n = [z^n]\left(\frac{1}{1-z}\log\frac{1}{1-z} + \frac{z^2-2z}{2(1-z)}\right),$$

*et vaut asymptotiquement*

$$\overline{\tau I}_n = \log n + \gamma - \frac{1}{2} + O(\frac{1}{n}).$$

Pour que ce théorème devienne entièrement automatique, il faudra implanter en $\Lambda\Upsilon^\Omega$ la règle de transformation de programme associée au schéma de test du minimum.

**Démonstration :** Le coefficient de $z^n$ dans $\tau I(z)$ donne le coût moyen pour l'insertion d'un entier $i$ dans un arbre $t$, avec $|i| + |t| = n$, c'est-à-dire $|t| = n-1$. L'analyse asymptotique est facile : $[z^n]\tau I(z) = [z^n]f(z) + [z^n]g(z)$, avec $f(z) = \frac{1}{1-z}\log\frac{1}{1-z}$ et $g(z) = \frac{z^2-2z}{2(1-z)}$. Le coefficient de $z^n$ dans $f(z)$ est $H_n = \log n + \gamma + O(1/n)$ (cf théorème automatique 12), et $[z^n]g(z) = 1/2[z^n]\frac{z^2}{1-z} - [z^n]\frac{z}{1-z} = -1/2$ pour $n \geq 2$. ∎

Ainsi, pour l'insertion dans un arbre tournoi de taille $n$, nous retrouvons le nombre moyen $H_{n+1} - 1/2$ de comparaisons obtenu par d'autres méthodes dans [FVV78].

## 5.2 Analyse comparée de QuickSort et InsertionSort

Le but de cette section est de déterminer algébriquement le seuil optimal entre une procédure de tri utilisant l'algorithme *QuickSort* (dont le coût moyen est en $n\log n$) et une autre procédure utilisant l'algorithme de tri par insertion (coût moyen en $n^2$). En ce qui concerne l'automatisation, cet exemple met en évidence deux difficultés que l'on peut rencontrer :

- pour certains paramètres, il est difficile de trouver un modèle de structure de donnée permettant de déterminer ce paramètre par une procédure de *descente*. C'est le cas ici du nombre d'inversions d'une permutation.

- il arrive aussi que les structures de données adaptées à la modélisation d'une procédure ne conviennent pas pour une autre procédure. Ici par exemple, la modélisation de *QuickSort* se fait bien à l'aide d'arbres tournois, alors que des structures linéaires sont plus adaptées pour déterminer le coût du tri par insertion.

Ainsi, pour la première raison, l'analyse du tri par insertion sera effectué "à la main". Celle de *QuickSort* sera faite de façon semi-automatique, car la règle de test sur la taille n'est pas encore implantée en $\Lambda\Upsilon\Omega$.

Il existe de multiples façons de "coder" les algorithmes *QuickSort* et de tri par insertion. Nous avons choisi une implantation classique, que nous décrivons dans le langage PASCAL : les deux procédures prennent en argument deux indices $l$ et $r$, et trient la liste de nombres $A[l] \ldots A[r]$ dans l'ordre croissant, en supposant l'existence de deux sentinelles $A[l-1] = -\infty$ et $A[r+1] = +\infty$ :

```
procedure QuickSort (l,r : integer);
begin
    if r-l<M then InsertionSort(l,r)
    else begin
      v:=A[l]; i:=l; j:=r+1;
       repeat
          repeat i:=i+1 until A[i]>v;
          repeat j:=j-1 until A[j]<v;
          if i<j then exchange(A[i],A[j])
       until j<i;
       if l<j then exchange(A[l],A[j]);
      QuickSort(l,j-1); QuickSort(i,r)
    end
end ;

procedure InsertionSort(l,r : integer);
begin
    for i:=l+1 to r do begin
      v:=A[i]; j:=i-1;
       while v<A[j] do begin
         A[j+1]:=A[j];
         j:=j-1
       end
    end
end ;
```

La procédure *QuickSort* appelle directement *InsertionSort* lorsque le nombre d'éléments à trier ($n = r - l + 1$) est inférieur ou égal à $M$. Quelle est la valeur optimale de $M$, c'est-à-dire celle qui minimise le nombre moyen de comparaisons entre éléments lorsque $n$ tend vers l'infini ?

**Analyse du tri par insertion.** Pour insérer le $i$-ème élément $A[i]$, le nombre de comparaisons effectuées est égal au nombre d'éléments qui sont supérieurs à $A[i]$ parmi $A[1] \ldots A[i-1]$, plus un pour la dernière comparaison (qui s'effectue avec un élément plus petit, éventuellement la sentinelle).

Le nombre de comparaisons nécessaires pour trier une permutation $\sigma = \sigma_1 \ldots \sigma_n$ est donc

$$\tau I\{\sigma\} = \sum_{i=2}^{n} \left( \left( \sum_{\substack{j=1 \\ \sigma_j > \sigma_i}}^{i-1} 1 \right) + 1 \right) = \mathrm{inv}(\sigma) + (n-1) \tag{5.1}$$

où $\mathrm{inv}(\sigma)$ est le nombre d'inversions de la permutation $\sigma$. Compter le coût moyen du tri par insertion revient donc à compter le nombre moyen d'inversions d'une permutation. Or nous connaissons trois modélisations des permutations :

1. la modélisation par les arbres tournois associés (cf section 5.1.1),
2. la modélisation par décomposition en cycles (exemple 21 page 86),
3. la modélisation linéaire (une permutation est une séquence d'atomes étiquetés).

Les deux premiers modèles ne sont pas adaptés pour déterminer le nombre d'inversions par une procédure de descente. Dans le troisième modèle, on pourrait écrire le programme suivant :

```
type perm = epsilon | perm elem;
     elem = Latom(1);

procedure insertion (p : perm);
begin
   case p of
      epsilon : fini;
      (pp,e) : begin insertion(pp); insert(pp,e) end;
   end;
end;

procedure insert(p : perm; e : elem);
begin
   case p of
      epsilon : count;
      (pp,f) : begin if e<f then count; insert(pp,e) end;
   end;
end;

measure count : 1;
```

mais la règle de test sur la taille n'étant pas implantée dans la version actuelle, le test **if** $e < f$ **then** count ne peut pas encore être analysé automatiquement par $\Lambda\Upsilon^{\Omega}$.

Nous effectuons par conséquent l'analyse "à la main", en repartant de l'équation (5.1). Le descripteur de complexité exponentiel de *InsertionSort* est

$$\begin{aligned} \tau I(z) &= \sum_n \left( \sum_{\sigma \in \Sigma_n} [\mathrm{inv}(\sigma) + (n-1)] \right) \frac{z^n}{n!} = \sum_n \left( \frac{1}{2} \binom{n}{2} n! + (n-1)n! \right) \frac{z^n}{n!} \\ &= \sum_n \frac{(n-1)(n+4)}{4} z^n. \end{aligned}$$

La procédure *InsertionSort* nécessite donc en moyenne $\frac{(n-1)(n+4)}{4}$ comparaisons pour trier $n$ éléments ; $\tau I(z)$ admet une forme explicite :

$$\tau I(z) = \frac{z^2(3-2z)}{2(1-z)^3} = \frac{3z^2}{2} + \frac{7z^3}{2} + 6z^4 + 9z^5 + \frac{25z^6}{2} + \frac{33z^7}{2} + 21z^8 + 26z^9 + \frac{63z^{10}}{2} + O(z^{11}).$$

**Analyse de la procédure QuickSort.** Le programme Adl suivant effectue le tri par *QuickSort* en s'arrêtant lorsque le nombre d'éléments à trier devient inférieur ou égal à $M$ :

```
type H = epsilon | H min(key) H;
     key = Latom(1);

procedure QuickSort (h : H);
begin
  if size(h) <= M then InsertionSort(h)
  else begin
    case h of
      ()          : nil;
      (h1,k,h2) : begin part(h1); part(h2);
                        quicksort(h1); quicksort(h2);
                  end;
    end;
  end;
end;

procedure part (h : H);              % partitionnement %
begin
  case h of
    ()          : nil;
    (h1,k,h2) : begin count; part(h1); part(h2) end;
  end;
end;

measure count : 1; nil : 0;
```

L'analyse est faire de manière semi-automatique, car le test sur la taille n'est pas implanté ; soit $\sigma$ la liste des éléments à trier, et $|\sigma| = n$ sa taille :

$$QuickSort(\sigma) \Rightarrow \begin{cases} InsertionSort(\sigma) & \text{si } n \leq M \\ (n+1) \text{ comparaisons } + \ QuickSort(\sigma_<) \ + \ QuickSort(\sigma_>) & \text{si } n > M. \end{cases}$$

L'équation correspondante en termes de séries génératrices est la suivante :

$$\tau Q(z) = \tau I_{\leq M}(z) + [zP'(z) + P(z)]_{>M} + 2\left[\int_0^z P(u)\tau Q(u)du\right]_{>M} \tag{5.2}$$

où $\tau Q$ est le descripteur de complexité de la procédure *QuickSort*, $P = 1/(1-z)$ la série génératrice exponentielle des permutations, et $f_{\leq M}$ (resp. $f_{>M}$) est la série obtenue à partir de $f$ en ne gardant que les termes de degré inférieur ou égal (resp. supérieur) à $M$.

L'équation différentielle (5.2) ne se résout pas telle quelle à cause de la restriction $> M$ sur l'intégrale. Écrivons $\tau Q$ sous la forme $\tau Q_{\leq M} + \tau Q_{>M}$ ; l'équation se scinde en deux parties :

$$\tau Q_{\leq M}(z) = \tau I_{\leq M}(z),$$

| n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\overline{\tau I}$ | 0 | 1.5 | 3.5 | 6.0 | 9.0 | 12.5 | 16.5 | 21.0 | 26.0 | 31.5 | 37.5 | 44.0 | 51.0 | 58.5 | 66.5 |
| $\overline{\tau Q_{>0}}$ | 2.0 | 5.0 | 8.7 | 12.8 | 17.4 | 22.3 | 27.5 | 32.9 | 38.6 | 44.4 | 50.5 | 56.7 | 63.0 | 69.6 | 76.2 |
| $\overline{\tau Q^*}$ | 0.0 | 1.0 | 2.7 | 4.8 | 7.4 | 10.3 | 13.5 | 16.9 | 20.6 | 24.4 | 28.5 | 32.7 | 37.0 | 41.6 | 46.2 |

| $n$ | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\overline{\tau I}$ | 75.0 | 84.0 | 93.5 | 103.5 | 114.0 | 125.0 | 136.5 | 148.5 | 161.0 | 174.0 | 187.5 | 201.5 | 216.0 | 231.0 |
| $\overline{\tau Q_{>0}}$ | 82.9 | 89.8 | 96.8 | 103.9 | 111.1 | 118.4 | 125.8 | 133.3 | 140.8 | 148.4 | 156.1 | 163.9 | 171.8 | 179.7 |
| $\overline{\tau Q^*}$ | 50.9 | 55.8 | 60.8 | 65.9 | 71.1 | 76.4 | 81.8 | 87.3 | 92.8 | 98.4 | 104.1 | 109.9 | 115.8 | 121.7 |

Tableau 5.1 : Nombre moyen de comparaisons d'*InsertionSort* et *QuickSort* avec $M = 0$.

$$\tau Q_{>M}(z) = \left[zP'(z) + P(z)\right]_{>M} + 2\left[\int_0^z P(u)\tau Q_{\leq M}(u)du\right]_{>M} + 2\int_0^z P(u)\tau Q(u)_{>M}du. \quad (5.3)$$

La restriction a disparu dans le dernier terme car les coefficients sous l'intégrale sont de degré plus grand que $M$. Remplaçons maintenant $P(z)$ par $1/(1-z)$ et $\tau Q_{\leq M}$ par $\tau I_{\leq M}$ dans l'équation (5.3) :

$$\tau Q_{>M}(z) = \left[\frac{1}{(1-z)^2} + 2\int_0^z \frac{\tau I_{\leq M}(u)}{1-u}du\right]_{>M} + 2\int_0^z \frac{\tau Q(u)_{>M}}{1-u}du. \quad (5.4)$$

Cette équation différentielle nous permet de calculer $\tau Q_{>M}$ pour n'importe quelle valeur du paramètre $M$. Le cas $M = 0$ est un cas particulier important, puisqu'il correspond à l'algorithme *QuickSort* utilisé seul. La solution obtenue est alors

$$\tau Q_{>0}(z) = \frac{2}{(1-z)^2}\log\frac{1}{1-z}.$$

Comme le montre le tableau 5.1, à partir de $n = 20$, le coût moyen de la procédure *QuickSort* avec $M = 0$ est inférieur à celui de *InsertionSort*. Ceci nous permet d'affirmer que la valeur optimale du paramètre $M$ est dans l'intervalle $[0, 20]$. Pour $0 \leq M \leq 20$, nous avons calculé le descripteur de complexité $\tau Q$ à l'aide de l'équation (5.4), puis nous avons déterminé un développement asymptotique du nombre moyen de comparaisons. Nous avons obtenu pour chaque $M$ un coût moyen de la forme

$$\overline{\tau Q_{>M}}_n = 2n\log n + (2\gamma - C_M)n + O(\log n)$$

où $C_M$ est un nombre rationnel. La valeur exacte de $C_M$ pour $M \leq 14$ est indiquée dans la table ci-dessous, ainsi qu'une approximation décimale arrondie.

| $M$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $C_M$ | 2 | $\frac{8}{3}$ | $\frac{35}{12}$ | $\frac{46}{15}$ | $\frac{19}{6}$ | $\frac{97}{30}$ | $\frac{131}{40}$ | $\frac{2077}{630}$ | $\frac{1387}{420}$ | $\frac{45659}{13860}$ | $\frac{2063}{630}$ | $\frac{584663}{180180}$ | $\frac{44441}{13860}$ | $\frac{569417}{180180}$ | $\frac{2239487}{720720}$ |
| | 2.00 | 2.67 | 2.92 | 3.07 | 3.17 | 3.23 | 3.28 | 3.297 | 3.302 | 3.294 | 3.28 | 3.25 | 3.21 | 3.16 | 3.11 |

L'optimum de $M$ correspond à la valeur maximale de $C_M$, qui est obtenue pour $M = 8$.

**Théorème 9.** *La valeur de $M$ qui minimise asymptotiquement le nombre moyen de comparaisons du programme* QuickSort *défini en* PASCAL *page 136, et modélisé en* ADL *page 138, est $M = 8$, et le nombre moyen de comparaisons pour trier une liste de $n$ nombres vaut alors*

$$2\,n\log n + A\,n + O(\log n)$$

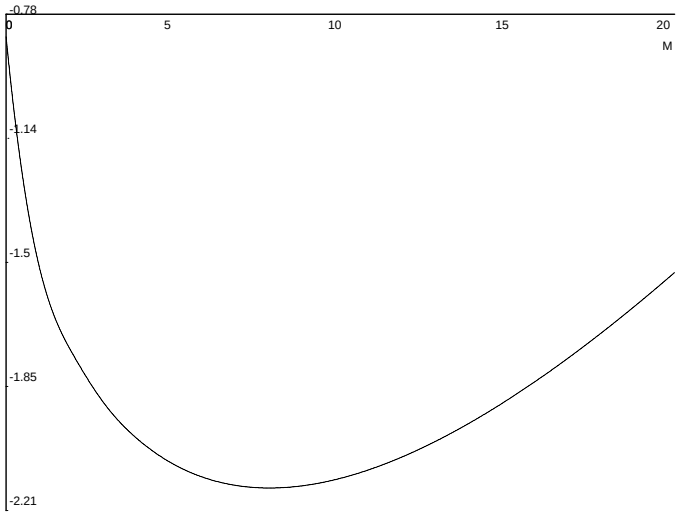



Figure 5.2 : Le graphe de $2\gamma - C_M$ en fonction de $M$

*et la constante A égale* $2\gamma - C_8 \simeq -2.147949622$.

Il faut remarquer que la valeur optimale de $M$ dépend d'une part de l'implantation précise de *QuickSort* et *InsertionSort*, et d'autre part de la définition du coût (comparaisons et/ou échanges). Par exemple, Knuth a analysé une implantation dans son langage *MIX*, et a trouvé comme valeur optimale $M = 9$ en comptant à la fois échanges et comparaisons [Knu73, pages 119–122].
REMARQUE : Prenons un exemple pour montrer que l'optimum dépend du modèle choisi. Si l'on utilise $n - 1$ comparaisons (au lieu de $n + 1$) pour le partitionnement dans *QuickSort*, la valeur optimale en ce qui concerne le nombre de comparaisons est $M = 0$ (au lieu de $M = 8$). En effet, le descripteur de complexité de la procédure *QuickSort* devient alors

$$\tau Q^*(z) = \frac{2}{(1-z)^2} \log \frac{1}{1-z} - \frac{2z}{(1-z)^2} = z^2 + \frac{8z^3}{3} + \frac{29z^4}{6} + \frac{37z^5}{5} + \frac{103z^6}{10} + \frac{472z^7}{35} + O(z^8).$$

Les coefficients sont tous plus petits que ceux du descripteur de complexité d'*InsertionSort* (figure 5.1). Cependant, nous ne prenons en compte ici que les comparaisons entre éléments à trier. Or pour réaliser l'étape de partitionnement en $n-1$ comparaisons, on augmente le nombre d'échanges d'éléments ou de tests de variables de boucle. La valeur optimale de $M$ dépend par conséquent de la valeur relative de ces opérations élémentaires.

## 5.3 Analyse de la valeur d'expressions arithmétiques

A la fin du chapitre précédent (section 4.3.2), nous avons défini un ensemble $\mathcal{N}$ d'expressions arithmétiques, et nous avons déterminé la probabilité $p_n$ qu'une expression de taille $n$ ait une valeur paire. A présent, nous allons nous intéresser à la *valeur moyenne* d'une expression de taille $n$. Berstel et Reutenauer [BR82] ont étudié ce problème, et ont montré que la fonction `eval` qui calcule la valeur d'une expression est une série formelle *reconnaissable*. Cela signifie qu'il existe un système d'équations dont le descripteur de complexité de la fonction `eval` est solution.

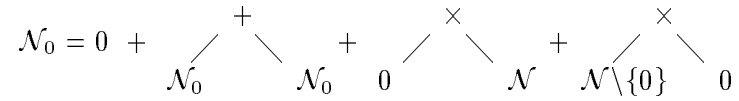
Tout d'abord, notons que la valeur d'une expression peut être exponentiellement grande par rapport à sa taille. Par exemple, l'expression $e_n$ définie par

$$\begin{aligned} e_1 &= 1+1 \\ e_k &= e_{k-1} \times e_1 \text{ pour } k > 1 \end{aligned}$$

est de taille $2n-1$, et sa valeur est $2^n$. D'autre part, nous savons que la valeur d'une expression de $\mathcal{N}$ est un entier $k \geq 0$, par croissance des opérateurs $+$ et $\times$. Le paragraphe qui suit étudie les expressions de valeur $k$ fixée ($k = 0, 1, 2, \ldots$), puis le paragraphe suivant détermine la valeur moyenne des expressions de taille $n$.

**Expressions à valeur fixée.**

Nous noterons $\mathcal{N}_k$ l'ensemble des expressions de valeur $k$. Par exemple, la spécification de $\mathcal{N}_0$ se déduit facilement de celle de $\mathcal{N}$ (page 123) :




dont il découle par les règles 2 et 3

$$N_0(z) = 1 + zN_0(z)^2 + zN(z) + z(N(z) - 1).$$

Sachant que $N(z) = \frac{1-\sqrt{1-16z}}{4z}$ (équation (4.8) page 124), nous obtenons grâce à Maple

$$N_0 = \frac{1-\sqrt{1-6z+2z\sqrt{1-16z}\ +4z^2}}{2z},$$

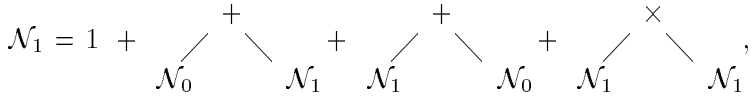
puis par le programme d'analyse asymptotique de B. Salvy

$$[z^n]N_0(z) = \frac{2}{\sqrt{41\pi}}\frac{16^n}{n^{3/2}} + O(\frac{16^n}{n^2}).$$

En divisant par le développement de $[z^n]N(z)$ qui vaut $\frac{2}{\sqrt{\pi}}\frac{16^n}{n^{3/2}} + O(\frac{16^n}{n^2})$ (équation (4.11) page 125), nous obtenons la probabilité $P_n(0)$ qu'une expression de taille $n$ ait une valeur nulle :

$$P_n(0) = \frac{1}{\sqrt{41}} + O(\frac{1}{\sqrt{n}}).$$ ■

Pour l'ensemble $\mathcal{N}_1$ des expressions qui valent 1, la spécification est




soit $N_1(z) = 1 + 2zN_0(z)N_1(z) + zN_1^2(z)$. Après calculs, il vient

$$N_1(z) = \frac{\sqrt{1-6z+2z\sqrt{1-16z}\ +4z^2}\ -\sqrt{1-10z+2z\sqrt{1-16z}\ +4z^2}}{2z},$$

d'où $$[z^n]N_1(z) = \frac{(82-10\sqrt{41}\,)16^n}{205\sqrt{\pi}\;n^{3/2}} + O(\frac{16^n}{n^2}),$$

puis $$P_n(1) = \frac{1}{5} - \frac{1}{\sqrt{41}} + O(\frac{1}{\sqrt{n}}).$$ ■

En additionnant $P_n(0)$ et $P_n(1)$, on constate qu'asymptotiquement, une expression sur cinq a une valeur inférieure ou égale à 1.

Pour $k \geq 2$, les ensembles $\mathcal{N}_k$ ont une spécification similaire :

$$\mathcal{N}_k = \biguplus_{\substack{0\leq i,j\leq k\\ i+j=k}} \mathcal{N}_i \;+\; \mathcal{N}_j \;+\; \biguplus_{\substack{1\leq i,j\leq k\\ ij=k}} \mathcal{N}_i \;\times\; \mathcal{N}_j .$$

A la différence de $\mathcal{N}_0$ et $\mathcal{N}_1$, ces spécifications conduisent à des équations *linéaires* en $N_k(z)$ :

$$N_k(z) = \sum_{i=0}^{k} zN_i(z)N_{k-i}(z) + \sum_{\substack{i=1\\ i\,|\,k}}^{k} zN_i(z)N_{\frac{k}{i}}(z).$$

Pour $k = 2$ par exemple, nous obtenons

$$P_n(2) = \frac{1}{\sqrt{41}} - \frac{17}{125} + O(\frac{1}{\sqrt{n}}).$$

**Valeur moyenne (statistique des arbres Catalans).**

Revenons au calcul de la valeur moyenne des expressions de taille $n$. Soit $V(z)$ la série comptant la valeur des expressions :

$$V(z) = \sum_{e\in\mathcal{N}} V(e)z^{|e|}.$$

D'après la spécification de $\mathcal{N}$, nous avons

$$\begin{aligned} V(z) &= 0\;z^0 + 1\;z^0 + \sum_{e=i+j} V(e)z^{|e|} + \sum_{e=i\times j} V(e)z^{|e|} \\ &= 1 + \sum_{i,j\in\mathcal{N}} (V(i)+V(j))z^{1+|i|+|j|} + \sum_{i,j\in\mathcal{N}} V(i)V(j)z^{1+|i|+|j|} \end{aligned}$$

c'est-à-dire

$$V(z) = 1 + 2zN(z)V(z) + zV^2(z). \tag{5.5}$$

Le lecteur aura remarqué que le membre droit de cette équation contient le terme non linéaire $V^2(z)$. Ce type d'équation ne peut pas être obtenu pour les descripteurs de complexité des programmes de II (cf théorème 4 page 68). Par suite il est impossible d'écrire un programme de II prenant en entrée une expression de $\mathcal{N}$, et calculant sa valeur. Néanmoins, comme nous le verrons plus loin, il est possible de simuler un tel programme par une structure de donnée, et d'obtenir $V(z)$ comme série de dénombrement.

La série $V(z)$ est *reconnaissable* au sens de Berstel-Reutenauer, puisqu'elle vérifie l'équation (5.5), qui se résout en

$$V(z) = \frac{1+\sqrt{1-16z}\;-\sqrt{2}\sqrt{1+\sqrt{1-16z}\;-16z}}{4z}.$$

Le développement asymptotique des coefficients de $V(z)$ est le suivant

$$[z^n]V(z) = \frac{\sqrt{2}\ 16^n}{\Gamma(3/4)n^{5/4}} + O(\frac{16^n}{n^{3/2}}). \tag{5.6}$$

**Théorème semi-automatique 15.** *Soit $\mathcal{N}$ l'ensemble des expressions constituées à l'aide des constantes 0 et 1, et des opérateurs binaires + et ×. La valeur moyenne d'une expression contenant $n$ opérateurs est*

$$\overline{V}_n = Cn^{1/4} + O(1),$$

*avec $C = \sqrt{2\pi}/(2\Gamma(3/4)) \simeq 1.023$.*

**Démonstration :** Il suffit de diviser la valeur totale des expressions de taille $n$ (5.6) par le nombre d'expressions de taille $n$ donné par (4.11). ■

Pour rendre ce théorème entièrement automatique, il faudrait introduire un schéma du genre

$$P(a : A; b : B) := Q(a) \times R(b) \qquad \text{où } Q : A \text{ et } R : B$$

dont le coût de $P(a, b)$ soit le *produit* du coût de $Q(a)$ par celui de $R(b)$. Ce schéma, qui est admissible ($\tau P(z) = \tau Q(z) \times \tau R(z)$) permettrait d'écrire la procédure `eval` sur les expressions de $\mathcal{N}$, et d'obtenir l'équation (5.5). Cependant, un tel schéma rompt la propriété de linéarité des descripteurs de complexité (cf théorème 4). Nous constatons au passage que les procédures de la classe II sont en quelque sorte "moins puissantes" que les structures de données, puisque ces dernières peuvent avoir des équations non linéaires. D'ailleurs, le paragraphe suivant montre qu'on peut utiliser dans certains cas une série de dénombrement pour calculer un coût non linéaire.

### Valeur moyenne (statistique des arbres tournois).

Comme nous les avons définies ci-dessus, les expressions arithmétiques de $\mathcal{N}$ sont distribuées suivant la statistique des arbres binaires non étiquetés. En particulier, ces arbres ont les propriétés statistiques suivantes

1. leur hauteur est en $\sqrt{n}$,
2. la profondeur moyenne de leur branche gauche (ou droite) est constante,
3. leur longueur de cheminement interne est en $n^{3/2}$.

Les propriétés 2 et 3 peuvent être obtenues à l'aide de $\Lambda\Upsilon\Omega$ (voir par exemple [FSZ89a, page 93] pour la longueur de cheminement). Or il existe d'autres statistiques sur les arbres binaires. Par exemple, les arbres dits *tournois*, définis à la section 5.1.1, dont les propriétés sont

1. leur hauteur est en $\log n$,
2. la profondeur moyenne de leur branche gauche (ou droite) est en $\log n$,
3. leur longueur de cheminement interne est en $n \log n$.

Ces arbres tournois ont des branches de profondeurs mieux équilibrées, et par conséquent nous pouvons nous attendre à une valeur moyenne plus élevée.

**Théorème automatique 16.** *Soit $\mathcal{N}'$ l'ensemble des expressions arithmétiques formées à partir des constantes* 0 *et* 1*, et des opérateurs binaires* + *et* $\times$*, suivant la statistique des arbres tournois. La valeur moyenne d'une expression contenant $n$ opérateurs vaut asymptotiquement*

$$\overline{V}'_n = \frac{2}{(1-e^{-4})^{n+1}} + O(\frac{(1-e^{-4})^{-n}}{n}).$$

Le facteur de croissance exponentielle est $(1-e^{-4})^{-1} \simeq 1.019$.

**Démonstration :** Le théorème est prouvé par l'analyse (automatique) du programme ADL suivant, où l'on a interprété l'équation (5.5) en termes de structure de donnée.

```
type N = zero | un | product(min(plus),N,N) | product(min(times),N,N);
     V = un | product(min(plus),N,V) | product(min(plus),V,N) | product(min(times),V,V);
     zero,un = Latom(0);
     plus,times = Latom(1);
```

Le rapport des coefficients de $z^n$ dans $V(z)$ et $N(z)$ donne la valeur moyenne d'une expression de taille $n$. La règle d'enracinement du minimum (règle 41) produit les équations différentielles

$$N(z) = 2 + 2\int_0^z N(u)^2 du, \qquad V(z) = 1 + 2\int_0^z N(u)V(u)du + \int_0^z V(u)^2 du,$$

dont les solutions sont

$$N(z) = \frac{2}{1-4z} \quad \text{et} \quad V(z) = \frac{4}{(1-4z)(4 - \log\frac{1}{1-4z})}.$$

L'analyse asymptotique conduit aux développements

$$[z^n]N(z) = 2\cdot 4^n \quad \text{et} \quad [z^n]V(z) = \left(\frac{4}{1-e^{-4}}\right)^{n+1} + O(\frac{1}{n}(\frac{4}{1-e^{-4}})^n).$$

dont la division donne le résultat annoncé. ∎

## 5.4 Un problème sur les colliers bicolores

Considérons des colliers constitués de perles blanches et noires, tels qu'en face de chaque perle blanche se trouve une perle noire (le nombre total de perles est donc pair). Nous appelons les colliers vérifiant cette propriété des colliers *antisymétriques*. Le problème posé par Jérôme Rey de Toulouse est le suivant :

> *Combien de colliers antisymétriques différents peut-on former avec $k$ perles blanches et $k$ noires ?*

Deux colliers sont dits identiques lorsque l'un s'obtient à partir de l'autre par rotation, mais sans le retourner (un collier possède un dessus et un dessous). Pour $k = 3$, il existe deux colliers antisymétriques (figure 5.3). Nous allons ramener ce problème à un problème similaire que nous saurons résoudre algébriquement.

Étant donné un collier antisymétrique de $2k$ perles, nous lui associons un cycle orienté de $k$ lettres prises parmi $D$ et $S$, en deux étapes :

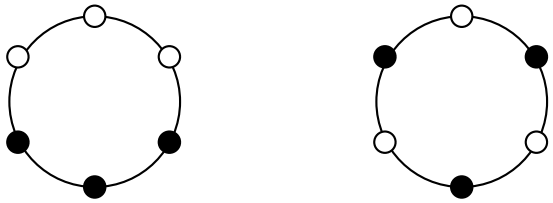


Figure 5.3 : Les deux colliers antisymétriques avec 6 perles.

1. marquer entre deux perles la lettre $S$ (pour *Semblable*) si elles sont de même couleur, et la lettre $D$ (pour *Différent*) si elles sont de couleurs différentes. On obtient ainsi un cycle de $2k$ lettres.
2. couper ce cycle en deux séquences de $k$ lettres et refermer l'une des séquences pour former un cycle de $k$ lettres (ce cycle est le même quel que soit l'endroit où l'on coupe grâce à la propriété de symétrie).

Le cycle obtenu à l'issue de la première étape est tel que les lettres diamétralement opposées sont identiques, car les perles opposées étaient de couleurs différentes. De plus, le nombre de changements de couleur (c'est-à-dire le nombre de lettres $D$) dans les séquences de la seconde étape est nécessairement impair.

**Lemme 12.** *Le nombre de colliers antisymétriques de $2k$ perles est égal au nombre de cycles orientés de $k$ lettres prises parmi $S$ et $D$ contenant un nombre impair de lettres $D$.*

**Démonstration :** La transformation décrite ci-dessus associe à chaque collier antisymétrique un cycle comportant un nombre impair de lettres $D$. Cette transformation est bijective : la première étape l'est car pour retrouver le collier antisymétrique, il suffit de partir d'une couleur au hasard, de placer une perle de même couleur que la précédente pour chaque lettre $S$, de couleur différente pour chaque lettre $D$. La seconde étape est aussi bijective : il suffit de dupliquer le cycle de $k$ lettres, et de mettre bout à bout les deux séquences obtenues. Enfin, un cycle de lettres $S$ et $D$ engendre un collier bicolore par la transformation réciproque, et ce collier est antisymétrique lorsque les lettres $D$ sont en nombre impair. ∎

Nous sommes à présent en mesure de donner la solution du problème posé par J. Rey.

**Théorème 10.** *Le nombre de colliers antisymétriques de $2n$ perles est le coefficient de $z^n$ dans*

$$h(z) = \frac{1}{2} \sum_{k \text{ impair}} \frac{\phi(k)}{k} \log \frac{1}{1-2z^k}. \tag{5.7}$$

La série génératrice ordinaire des colliers antisymétriques est donc la moitié de la partie impaire de l'opérateur de Pólya $\Phi_C$, appliqué à la fonction $f(z) = 2z$.

**Démonstration :** La série ordinaire à deux variables de l'ensemble $\{S, D\}$ est $f(s,d) = s + d$, où $s$ et $d$ marquent respectivement les lettres $S$ et $D$. Par la règle 7 du chapitre 2 (page 52), la série associée aux cycles orientés sur $\{S, D\}$ est

$$g(s,d) = \Phi_C(f)(s,d) = \sum_{k \geq 1} \frac{\phi(k)}{k} \log \frac{1}{1-(s^k+d^k)}.$$

La partie impaire par rapport à $d$ est $(g(s,d)-g(s,-d))/2$. Enfin, pour compter ensemble les cycles ayant même nombre total de lettres, nous remplaçons $s$ et $d$ par $z$, d'où

$$h(z) = \frac{g(z,z) - g(z,-z)}{2},$$

ce qui donne bien le résultat annoncé par le théorème. ■

Calculons les premiers termes de la série $h(z)$ à l'aide de Maple :

```
> h:= 1/2*sum(phi(2*k-1)/(2*k-1)*log(1/(1-2*z^(2*k-1))),k=1..6):
> series(h,z,12);
                2      3      4      5      6       7       8       9       10       11
        z + z  + 2 z  + 2 z  + 4 z  + 6 z  + 10 z  + 16 z  + 30 z  + 52 z   + O(z  )
```

Le comportement asymptotique des coefficients de $h(z)$ obtenu par analyse de singularités (cf [Sal91]) est :

$$h_n = \frac{2^{n-1}}{n} + \frac{\left[1+\cos\frac{2n\pi}{3}\right]2^{n/3}}{3n} + \frac{2\left[1+2\cos\frac{2n\pi}{5}+2\cos\frac{4n\pi}{5}\right]2^{n/5}}{5n} + O\left(\frac{2^{n/7}}{n}\right).$$

On obtient en fait un développement exact en termes de la forme $C_k(n \bmod k)\frac{2^{n/k}}{n}$ où $k$ est impair.

## 5.5 Parenthésages ternaires commutatifs

L'objectif principal de cette section est de mettre en évidence sur un exemple des méthodes de calcul asymptotique qui devraient permettre de traiter systématiquement une large classe de fonctions données par une équation non algébrique (faisant intervenir par exemple des opérateurs de Pólya).

Un parenthésage $k$-aire d'un produit commutatif de facteurs est une façon de regrouper les facteurs par groupes de $k$, sans notion d'ordre entre les $k$ éléments d'un même groupe. Pour $k = 3$, nous obtenons des parenthésages ternaires ; par exemple, il y a deux parenthésages ternaires de $X^7$ :

$$(XX(XX(XXX))) \quad \text{et} \quad (X(XXX)(XXX)).$$

Un tel parenthésage est soit une variable $X$, soit une parenthèse ouvrante suivie d'un produit non ordonné de trois parenthésages (éventuellement égaux) et d'une parenthèse fermante :

**type** A = X | par_ouvr *multiset*(A, card=3) par_ferm;
X,par_ferm = *atom*(0);
par_ouvr = *atom*(1);

**Théorème automatique 17.** *Le nombre de parenthésages ternaires commutatifs de $X^{2n+1}$ est égal au coefficient de $z^n$ dans la série solution de l'équation fonctionnelle*

$$A(z) = 1 + z\left(\frac{A(z^3)}{3} + \frac{A(z)A(z^2)}{2} + \frac{A(z)^3}{6}\right) \tag{5.8}$$

*dont les premiers termes sont*

$$A(z) = 1 + z + z^2 + 2z^3 + 4z^4 + 8z^5 + 17z^6 + 39z^7 + 89z^8 + 211z^9 + 507z^{10} + 1238z^{11} + O(z^{12}).$$

**Démonstration :** La spécification ci-dessus définit la taille d'un parenthésage comme le nombre $n$ de parenthèses ouvrantes ; le nombre de variables est alors $2n+1$. L'analyse de cette spécification par $\Lambda\Upsilon^{\Omega}$ donne le résultat suivant :

```
A(z) is the solution of the following functional equation:
                                      3             2          3
            A(z) = 1 + z (1/3 A(z ) + 1/2 A(z) A(z ) + 1/6 A(z) )
```

La production `multiset(A, card=3)` à été traduite à l'aide de l'équation (2.12) page 58 :

$$\Psi_{\mathcal{M},3}(A,z) = [u^3]\exp(u\frac{A(z)}{1} + u^2\frac{A(z^2)}{2} + +u^3\frac{A(z^3)}{3}) = \frac{A(z^3)}{3} + \frac{A(z)A(z^2)}{2} + \frac{A(z)^3}{6}. \quad \blacksquare$$

Pour déterminer les premiers termes de $A(z)$, nous avons écrit une procédure MAPLE qui calcule le développement de Taylor en $z=0$ d'une série donnée par une équation fonctionnelle :

```
Series := proc(expr,z)
local ord;
   if nargs=3 then ord:=args[3] else ord:=Order fi;
   if type(expr,equation) then
      `series/equation`(expr,z,ord)
   else
      series(expr,z,ord)
   fi
end:

`series/equation` := proc(eq,z,n)
local t,ord,B;
   t := op(1,op(2,eq));
   ord := degree(t,z)+1;
   t := t + O(z^ord);
   B := op(0,op(1,eq));
   while ord<n do
      subs(u=t,B=proc(z) u end);
      eval(subs('',op(2,eq)));
      t := series('',z,n);
      ord := op(nops(t),t);
   od;
   t
end:

> Series(A(z) = 1+z*(1/3*A(z**3)+1/2*A(z)*A(z**2)+1/6*A(z)**3),z,12);
         2      3      4      5       6       7       8        9        10       11
1 + z + z  + 2 z  + 4 z  + 8 z  + 17 z  + 39 z  + 89 z  + 211 z  + 507 z   + O(z  )
```

L'analyse algébrique des parenthésages ternaires commutatifs ne présente donc aucune difficulté, comme nous avons pu le constater. Il n'en est pas de même de l'analyse asymptotique : la difficulté provient des termes $A(z^2)$ et $A(z^3)$ qui confèrent à l'équation (5.8) un caractère non algébrique.

### 5.5.1 Évaluation asymptotique de $A_n$.

Le calcul asymptotique de $A_n$ a déjà été fait par Pólya [Pól37, PR87] à l'aide d'analyse réelle ; ce qui nous intéresse particulièrement ici n'est pas le résultat en soi, mais de montrer sur cet exemple

une méthode générale pouvant s'appliquer à d'autres équations fonctionnelles du genre de (5.8). Cette méthode basée sur l'analyse de singularités est décrite dans [HRS75] ou dans [FSZ91].

**Recherche de la singularité dominante.** Étant donnée une équation

$$A(z) = \Phi(z, A(z), A(z^2), \ldots),$$

la première chose à faire est de déterminer le rayon de convergence $\rho$ de la fonction complexe $A(z)$. Comme ici les coefficients $A_n$ sont des entiers positifs (parce que l'on a affaire à une série *ordinaire*), et qu'une infinité sont non nuls, $\rho$ est compris entre 0 et 1.

L'équation $A(z) = \Phi(z, A(z), A(z^2), \ldots)$ s'écrit aussi $P(z, A(z)) = 0$, où la fonction $P$ est définie par $P(z, y) = \Phi(z, y, A(z^2), \ldots) - y$. Ceci revient à considérer les termes $A(z^2)$, $A(z^3)$, ... comme des paramètres, ce qui est motivé par le fait que $z^2$, $z^3$, ... croissent moins vite que $z$, et par suite $A(z^2)$, $A(z^3)$, ... sont analytiques dans un disque plus grand que $A(z)$. Ici nous avons

$$P(z, y) = \frac{z}{6}y^3 + (\frac{zA(z^2)}{2} - 1)y + \frac{zA(z^3)}{3} + 1.$$

Au voisinage d'un point $M_0 = (z_0, y_0)$ tel que $P(M_0) = 0$, l'équation de la courbe $P(z, y) = 0$ s'écrit

$$\frac{dP}{dM} = \frac{\partial P}{\partial z}(M_0)(z - z_0) + \frac{\partial P}{\partial y}(M_0)(y - y_0) \simeq 0.$$

Lorsque la dérivée partielle de $P$ par rapport à $y$ est non nulle, $y$ dépend de façon linéaire de $z$ au voisinage de $M_0$ :

$$y \simeq y_0 + \frac{\frac{\partial P}{\partial z}(M_0)}{\frac{\partial P}{\partial y}(M_0)}(z_0 - z).$$

En revanche, lorsque la dérivée partielle par rapport à $y$ s'annule, $y$ s'obtient par l'extraction d'une racine carrée

$$y \simeq y_0 - \lambda\sqrt{1 - z/z_0} \qquad \text{où } \lambda = \sqrt{2z_0 \frac{\frac{\partial P}{\partial z}(M_0)}{\frac{\partial^2 P}{\partial y^2}(M_0)}}, \tag{5.9}$$

voire par une racine d'ordre supérieur si la dérivée seconde par rapport à $y$ s'annule aussi ; le point $M_0$ est alors un point singulier. Les points singuliers de la courbe $y = A(z)$ sont définis par le système

$$\begin{cases} P(z, y, A(z^2), \ldots) = 0 \\ \frac{\partial P}{\partial y}(z, y, A(z^2), \ldots) = 0. \end{cases} \tag{5.10}$$

Tel quel, ce système n'est pas très intéressant, car en un point singulier $z = \rho$, la valeur de la série $y = A(z)$ est parfois infinie. Par contre, lorsque $0 < \rho < 1$, $A(z^2)$, $A(z^3)$, ... sont définis en $z = \rho$, car les points $z = \rho^2$, $z = \rho^3$, ... sont *à l'intérieur* du disque de convergence de $A(z)$. Ainsi, l'élimination de $y$ entre les deux équations du système (5.10) fournit une expression en $z$, $A(z^2)$, ... qui s'annule aux points singuliers.

Dans notre exemple, nous obtenons le système suivant

$$\begin{cases} \frac{z}{6}y^3 + (\frac{zA(z^2)}{2} - 1)y + \frac{zA(z^3)}{3} + 1 = 0 \\ \frac{z}{2}y^2 + \frac{zA(z^2)}{2} - 1 = 0 \end{cases}. \tag{5.11}$$

L'élimination de $y$ donne

$$f(z) \stackrel{\text{def}}{=} 9z + 6z^2A(z^3) + z^3A(z^3)^2 + z^3A(z^2)^3 - 6z^2A(z^2)^2 + 12zA(z^2) - 8 = 0. \tag{5.12}$$

Cette dernière équation nous permet de déterminer numériquement $\rho$, à condition de savoir calculer $A(z^2)$, $A(z^3)$, ...(cf paragraphe suivant). La figure 5.4 montre le graphe de $f(z)$ ; on constate au vu de ce graphe que la singularité $\rho$ se trouve dans l'intervalle $[\frac{1}{4}, \frac{3}{8}]$.

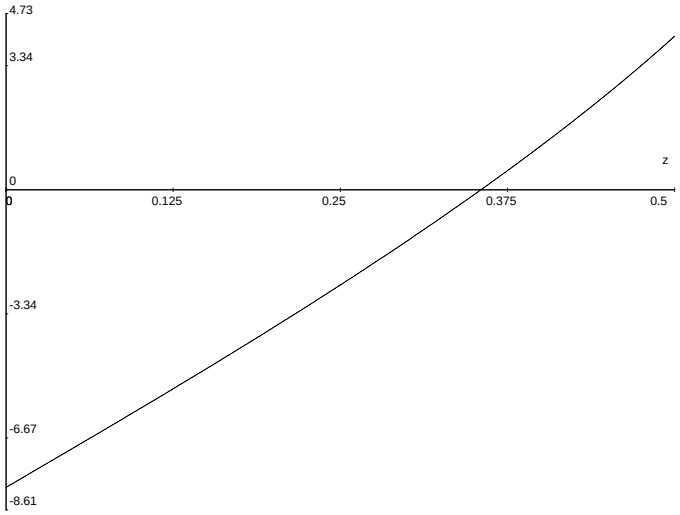



Figure 5.4 : Le graphe de la fonction $f(z)$.

**Calcul de $A(z)$ pour $z < \rho$.** Lorsque $z < \rho$, la valeur $y$ de $A(z)$ est la solution de l'équation $y = \Phi(z, y, A(z^2), \ldots)$, qui dépend des valeurs de $A$ aux puissances entières $z^k$, $k \geq 2$. Ces puissances tendent vers 0 lorsque $k$ tend vers l'infini car $z < \rho \leq 1$. Or la fonction $A(z)$ est continue en 0 (elle est en fait $C^\infty$ puisqu'elle admet un développement de Taylor à tout ordre), donc $A(z^k)$ tend vers $A(0)$ lorsque $k$ tend vers l'infini ; il tend même par valeur supérieure car les coefficients du développement de Taylor de $A(z)$ en $z = 0$ sont positifs. Nous obtenons ainsi une minoration $A^-$ de $A$ :

$$A^-(z) = \begin{cases} A(0) \text{ si } 0 \leq z \leq \epsilon, \\ \text{la plus petite racine positive de } X = \Phi(z, X, A^-(z^2), \ldots) \text{ sinon.} \end{cases} \tag{5.13}$$

Une majoration de $A(z)$ est obtenue de la façon suivante. Du fait de la croissance de $A(z)$ (qui provient de la croissance de l'opérateur $\Psi$ défini par (5.8) tel que $A(z) = 1 + z\Psi(A, z)$), nous avons les inégalités $A(z^2) \leq A(z)$ et $A(z^3) \leq A(z)$. Ceci entraîne $A(z) \leq B(z)$, où $B(z)$ est définie par l'équation

$$B(z) = 1 + z(\frac{B(z)}{3} + \frac{B(z)^2}{2} + \frac{B(z)^3}{6})$$

qui s'obtient en remplaçant dans l'équation (5.8) chaque terme $A(z^k)$ par $B(z)$. Par conséquent le rayon de convergence $\rho$ de $A(z)$ est plus grand que $\rho_B$, celui de $B(z)$. La fonction $A(z)$ étant

convexe sur $[0, \rho_B]$ (cela découle de la convexité de $\Psi$), elle vérifie sur cet intervalle l'encadrement

$$A(0) \leq A(z) \leq A(0) + \frac{B(\rho_b) - A(0)}{\rho_B} z.$$

L'équation dont $\rho_B$ est racine ne dépend quant à elle pas de $B$, contrairement à (5.12) qui dépendait de $A(z^2)$ et $A(z^3)$ :

$$z^3 + 18z^2 - 864z + 216 = 0.$$

La fonction $z^3 + 18z^2 - 864z + 216$ admet une seule racine entre 0 et 1, et cette racine vaut $\rho_B \simeq 0.2513343960$, et $B(\rho_B) \simeq 1.879385241$, ce qui donne comme majoration $A(z) \leq 1 + 7z/2$. Le programme Maple ci-dessous calcule un encadrement $A^-(z) \leq A(z) \leq A^+(z)$, en fonction du paramètre $\epsilon$ de fin de récursion utilisé dans (5.13) :

```
Agen := proc(z,epsilon)
option remember;
   if z<epsilon then RETURN(EXPR)
   else
      X = 1 + z * (procname(z^3,epsilon)/3 + X*procname(z^2,epsilon)/2 + X^3/6);
      [fsolve(",X,1..infinity)];
      if nops(")=0 then RETURN(infinity) else RETURN(min(op("))) fi
   fi
end:

Amin := subs(EXPR=1,op(Agen)):      Amax := subs(EXPR=1+7*z/2,op(Agen)):

A := proc(z,epsilon)
   Amin(z,epsilon),Amax(z,epsilon)
end:

> A(1/4,0.1);
                              1.357119307, 1.425924477

> A(0.352,0.1);
                              1.912089895, infinity
```

Pour $\epsilon = 0.01$, nous obtenons les deux courbes indiquées sur la figure 5.5, qui constituent déjà un encadrement assez fin.

La fonction $A(z)$ étant croissante, la formule (5.8) diverge pour $z > \rho$. Ceci suggère une autre méthode pour déterminer numériquement $\rho$ : si $\texttt{Amax}(z_1, \epsilon)$ converge, et $\texttt{Amin}(z_2, \epsilon)$ diverge, alors $z_1 \leq \rho \leq z_2$. Le programme Maple suivant détermine par dichotomie un intervalle contenant $\rho$, de largeur au plus $2.10^{-\text{Digits}}$, en partant d'un intervalle $[a, b]$ fourni (on pourra prendre par exemple $a = 0$ et $b = 1$).

```
int_rho := proc(a,b)
local mi,lo,hi,eps,oldDigits;
   lo:=a: hi:=b: eps:=0.01; oldDigits:=Digits; Digits:=Digits+11;
   while hi-lo>10^(-oldDigits) do
      mi := evalf((lo+hi)/2);
      Amin(mi,eps);
      if "=infinity then hi:=mi
      else
```

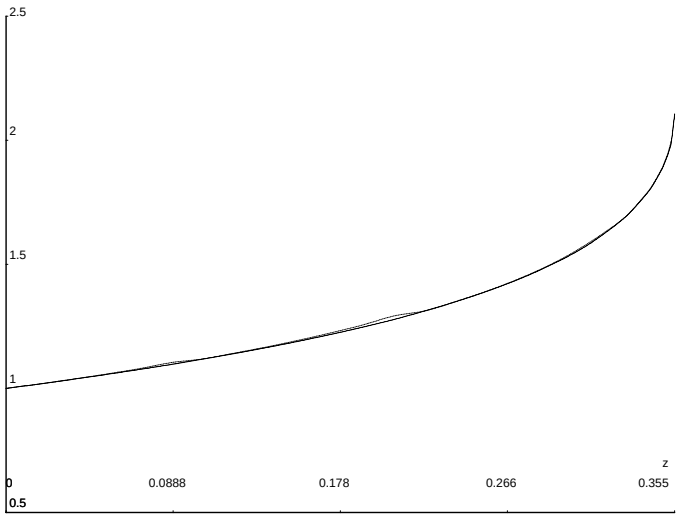



Figure 5.5 : Majoration et minoration de $A(z)$ pour $\epsilon = 0.01$.

```
        Amax(mi,eps);
        if "<>infinity then lo:=mi else eps:=eps^2 fi
      fi
   od;
   Float(5,-oldDigits-1);
   evalf(lo-",oldDigits),evalf(hi+",oldDigits)
end:

> Digits:=2: int_rho(0,1);
                                    .35, .36

> Digits:=3: int_rho(0,1);
                                   .354, .356

> Digits:=4: int_rho(0,1);
                                  .3551, .3553
```

Avec `Digits:=10`, nous obtenons $0.3551817422 \leq \rho \leq 0.3551817424$, ce que nous notons $\rho \simeq 0.3551817423$ en convenant que $x \simeq y$ signifie "$x$ égale $y$, à une unité près sur la dernière décimale de $y$". Avec `Digits:=40`, la procédure `int_rho` donne[1]

$$\rho \simeq 0.3551817423143773928822444736476326367087. \tag{5.14}$$

**Développement local au voisinage de la singularité.** L'équation (5.9) s'écrit au voisinage de $\rho$ :

$$A(z) = A(\rho) - \lambda\sqrt{1 - z/\rho} + \mu(1 - z/\rho) + O((1 - z/\rho)^{3/2}).$$

Le passage aux coefficients donne [FO90] :

$$[z^n]A(z) = -\lambda[z^n]\sqrt{1 - z/\rho} + O([z^n](1 - z/\rho)^{3/2})$$

[1] En environ 10000 secondes sur une station de travail DS 3100.

Or le coefficient de $z^n$ dans $(1-z/\rho)^\alpha$ vaut $\rho^{-n}n^{-\alpha-1}/\Gamma(-\alpha)+O(\rho^{-n}n^{-\alpha-2})$, et $\Gamma(-1/2) = -2\sqrt{\pi}$, d'où

$$A_n = [z^n]A(z) = \frac{\lambda}{2\sqrt{\pi}}n^{-3/2}\rho^{-n} + O(n^{-5/2}\rho^{-n}).$$

Il ne nous reste plus qu'à déterminer la constante $\lambda$ pour connaître complètement le premier terme du développement asymptotique de $A_n$.

**Calcul de $\lambda$.** Cette constante est définie par l'équation (5.9) :

$$\lambda = \sqrt{2\rho\frac{\frac{\partial P}{\partial z}(M_0)}{\frac{\partial^2 P}{\partial y^2}(M_0)}}.$$

Ici, elle vaut

$$\lambda = \sqrt{\frac{\frac{2A(\rho^3)}{3} + A(\rho)A(\rho^2) + \frac{A(\rho)^3}{3} + 2A'(\rho^3)\rho^3 + 2A(\rho)A'(\rho^2)\rho^2}{A(\rho)}},$$

qui se simplifie en remarquant que $\frac{2A(\rho^3)}{3} + A(\rho)A(\rho^2) + \frac{A(\rho)^3}{3} = 2(A(\rho)-1)/\rho$ (équation (5.8), valide aussi en $z=\rho$) :

$$\lambda = \sqrt{2\frac{A(\rho) - 1 + A'(\rho^3)\rho^4 + A(\rho)A'(\rho^2)\rho^3}{\rho A(\rho)}},$$

et nous retrouvons la forme donnée par Read dans [PR87, page 82]. Pour calculer numériquement $\lambda$, il nous faut déterminer $A(\rho)$, $A'(\rho^2)$ et $A'(\rho^3)$. La valeur de $A(\rho)$ s'obtient à l'aide de la seconde équation du système (5.11), qui s'écrit

$$A(\rho) = \sqrt{\frac{2}{\rho} - A(\rho^2)}.$$

A l'aide de l'approximation (5.14) de $\rho$, nous trouvons $A(\rho) \simeq 2.11742070095363102251625859046 4$. Quant à $A'(z)$ pour $z < \rho$, il s'obtient en dérivant par rapport à $z$ l'équation $P(z, A(z)) = 0$, ce qui donne après simplifications

$$A'(z) = \frac{1}{z}\frac{A(z) - 1 + z^4A'(z^3) + z^3A(z)A'(z^2)}{1 - zA(z^2)/2 - zA(z)^2/2}.$$

Cette équation permet de calculer une minoration et une majoration de $A'(z)$, comme pour $A(z)$. Nous utilisons comme minoration pour $z$ petit $A'(z) \geq 0$. Quant à la majoration, un calcul simple montre que $A'(1/4) < 3$, donc $A'(z) < 3$ pour $z \leq 1/4$ ($A$ étant convexe, $A'$ est croissante).

Après quelques calculs, nous déterminons $A'(\rho^2) \simeq 1.39495772471934167427258842514213 8$ et $A'(\rho^3) \simeq 1.10328312738141366840906415181059 9$, puis (enfin!)

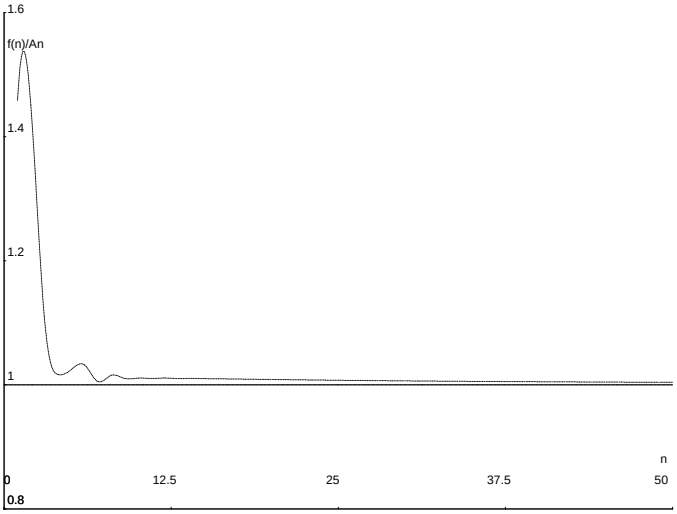
$$\lambda \simeq 1.83582228938850129966748542694589.$$

**Théorème 11.** *Le nombre $A_n$ de parenthésages ternaires commutatifs de $X^{2n+1}$ vérifie*

$$A_n = Cn^{-3/2}\alpha^n + O(n^{-5/2}\alpha^n)$$

*où $C = \frac{\lambda}{2\sqrt{\pi}} \simeq .51787590645889353699316235699285 4$ et $\alpha = \frac{1}{\rho} \simeq 2.81546003317615074652661677824270$.*

La fonction $f(n) = Cn^{-3/2}\alpha^n$ est une bonne approximation de $A_n$, comme on peut le constater sur le graphe ci-dessous, qui donne le rapport $f(n)/A_n$ en fonction de $n$.




## 5.6 Où est expliqué pourquoi le nombre d'alcanes de formule brute $C_nH_{2n+2}$ vaut asymptotiquement $C\, n^{-5/2}\, \alpha^n$

Dans cette dernière section, nous expliquons le lien existant entre les parenthésages ternaires étudiés à la section précédente, et les composés chimiques dénommés usuellement "alcanes". L'intérêt de cette étude est multiple. D'abord, on montre que le dénombrement des arbres non enracinés découle de celui des arbres enracinés (lemme 13). Ceci peut suggérer l'introduction d'un nouveau constructeur pour créer des arbres non enracinés. Nous établissons ensuite que l'analyse asymptotique des arbres non enracinés se ramène à l'étude d'une certaine puissance d'une série tronquée (lemme 15). Nous présentons alors un calcul original basé sur le développement local au voisinage de la singularité, qui redonne le résultat de Pólya et Read (théorème 12).

Les hydrocarbures saturés (appelés aussi alcanes ou paraffines) sont les composés chimiques de formule brute $C_nH_{2n+2}$, où $C$ représente un atome de carbone et $H$ un atome d'hydrogène. Ce sont les constituants de base des dérivés organiques qui nous entourent (plastiques, essences, gaz). Les atomes de carbone d'un alcane forment un graphe acyclique connexe, c'est-à-dire un arbre non enraciné. Or Jordan a montré en 1869 le résultat suivant sur les arbres non enracinés.

**Lemme 13.** *[PR87] Un arbre non enraciné de $n$ nœuds a un ou deux nœuds* exceptionnels *(tel que toutes les branches qui en partent ont au plus $n/2$ nœuds). De plus, lorsqu'il y a un seul nœud exceptionnel, les branches qui en partent ont strictement moins de $n/2$ nœuds.*

Quand il y a un seul point exceptionnel, il est appelé le *centre* de l'arbre ; lorsqu'il y en a deux, ce sont les *bicentres*. Par conséquent, nous pouvons décomposer les alcanes en deux types.

**type $\alpha$ :** ceux qui ont deux atomes de carbone centraux,

**type $\beta$ :** ceux qui ont un seul atome de carbone central.

Les alcanes du type $\alpha$ sont de la forme $C_kH_{2k+1}-C_kH_{2k+1}$. Pour ceux du type $\beta$, l'atome central est relié à quatre radicaux $C_iH_{2i+1}$ contenant chacun un nombre d'atomes de carbone dans l'intervalle $[0, n/2[$, où $n$ est le nombre total d'atomes de carbone.

On peut associer à chaque parenthésage ternaire commutatif sur $2n+1$ variables un radical alkyle de formule brute $-C_nH_{2n+1}$ de la manière suivante : chaque couple de parenthèses correspond à un atome de carbone $C$, chaque variable à un atome d'hydrogène $H$, les trois facteurs d'un parenthésage sont associés à trois radicaux liés à l'atome de carbone. Cette bijection donne sur les deux parenthésages de $X^7$ :

$$(XX(XX(XXX))) \Leftrightarrow -CH_2-CH_2-CH3 \quad \text{et} \quad (X(XXX)(XXX)) \Leftrightarrow -CH(CH_3)_2.$$

Le nombre d'alkyles en $C_nH_{2n+1}$ est donc exactement le nombre de parenthésages ternaires commutatifs $A_n$ étudié dans la section précédente. Les alkyles étant alors définis par

**type** Alkyl = H | *product*(C, *multiset*(Alkyl, *card*=3);
C = *atom*(1); H = *atom*(0);

le lemme 13 conduit à la pseudo-spécification suivante pour les alcanes :

Alcane = *multiset*(Alkyl, *card*=2, size=n/2) % *type alpha* %
| *product*(C, *multiset*(Alkyl, *card*=4, size<n/2)); % *type beta* %

Ce résultat permet de déterminer le nombre $B_n$ d'alcanes de taille $n$ si l'on connaît $A_k$ pour $0 \le k \le n/2$ :

$$B(z) = 1 + z + z^2 + z^3 + 2\,z^4 + 3\,z^5 + 5\,z^6 + 9\,z^7 + 18\,z^8 + 35\,z^9 + 75\,z^{10} + 159\,z^{11} + 355\,z^{12} + O(z^{13}).$$

**Lemme 14.** *Il existe deux suites $(Y_n)_{n\ge 0}$ et $(Z_n)_{n\ge 0}$ telles que*

1. $\forall n, Y_n \le A_n \le Z_n$,
2. *il existe deux constantes $J, K > 0$ telles que pour $n \ge 1$, $Y_n = Jn^{-3/2}\alpha^n$ et $Z_n = Kn^{-3/2}\alpha^n$,*
3. $1 \le i \le j \Rightarrow Y_iY_j \le Y_{i-1}Y_{j+1}$ *et* $Z_iZ_j \le Z_{i-1}Z_{j+1}$.

**Démonstration :** Comme $A_n = O(n^{-3/2}\alpha^n)$, il existe un entier $N$ et une constante $K_1$ tels que pour $n \ge N$, $A_n \le K_1 n^{-3/2}\alpha^n$. Soit $K_2 = \max\{A_n/(n^{-3/2}\alpha^n), 1 \le n < N\}$, et $K = \max(1/3, K_1, K_2)$. Nous définissons alors $Z_n = Kn^{-3/2}\alpha^n$ pour $n \ge 1$, et $Z_0 = 3K$. Cette suite vérifie les deux premières conditions. La troisième condition équivaut à la croissance de $Z_{n+1}/Z_n$ pour $n \ge 0$. Or pour $n \ge 1$, $Z_{n+1}/Z_n = \alpha(n/(n+1))^{3/2}$, fonction qui est évidemment croissante ; et $Z_1/Z_0 = \alpha/3 < (1/2)^{3/2}\alpha = Z_2/Z_1$. Pour la suite $(Y_n)$, le principe est identique. ■

Ce lemme nous servira à majorer les sommes de produits $A_iA_j$, où $i+j$ est constant.

**Lemme 15.** *Le nombre $B_n$ d'alcanes ayant $n$ atomes de carbone vérifie asymptotiquement*

$$B_n \sim \frac{1}{24}[z^{n-1}]\,(f_n(z))^4\,,$$

*où $f_n(z) = \sum_{0\le k<n/2} A_k z^k$, et $A_k$ est le nombre de radicaux alkyles de formule brute $C_kH_{2k+1}$.*

**Démonstration :** Lorsque $n = 2k$, le nombre $B_n^\alpha$ d'alcanes de type $\alpha$ est $\binom{A_k+1}{2}$, qui est équivalent à $A_k^2/2$. Or $A_k \sim Ck^{-3/2}\alpha^k$, d'où $B_n^\alpha = O(n^{-3}\alpha^n)$. Lorsque $n$ est impair, il n'y a pas d'alcanes de type $\alpha$.

D'après le lemme 13, les alcanes de type $\beta$ sont le produit d'un atome de carbone central par un multi-ensemble de quatre radicaux alkyles, tous de taille inférieure à $n/2$. Nous utilisons alors l'équation (2.12) donnant la série de dénombrement des multi-ensembles de longueur $k = 4$ :

$$B_n^\beta(z) = \frac{f_n(z)f_n(z^3)}{3} + \frac{f_n(z^4)}{4} + \frac{f_n(z^2)^2}{8} + \frac{f_n(z^2)f_n(z)^2}{4} + \frac{f_n(z)^4}{24}.$$

L'étude des trois premiers termes est facile : $f_n(z)f_n(z^3) = O(n^{-2}\alpha^{3n/4})$, $f_n(z^4) = O(n^{-3/2}\alpha^{n/4})$ et $f_n(z^2)^2 = O(n^{-3/2}\alpha^{n/2})$. Le quatrième terme $f_n(z^2)f_n(z)^2 = O(n^{-3}\alpha^n)$, mais ce résultat n'est pas aussi évident : $[z^{n-1}]f_n(z^2)f_n(z)^2 = S_1 + S_2$ où

$$S_1 = \sum_{0\le i<n/4} \sum_{2i+j+k=n-1} F_iF_jF_k \text{ et } S_2 = \sum_{n/4\le i<n/2} \sum_{2i+j+k=n-1} F_iF_jF_k,$$

avec $F_l = A_l$ si $0 \le l < n/2$, et $F_l = 0$ sinon. Soit $\phi$ la fonction qui à un terme $F_iF_jF_k$ de $S_2$ associe le terme $F_jF_iF_k$ si $j < n/4$, et $F_kF_jF_i$ sinon. Comme $i \ge n/4$, nécessairement $j < n/4$ ou $k < n/4$, donc le terme associé est dans $S_1$. Chaque terme de $S_1$ ayant au plus deux antécédents par $\phi$, $S_2 \le 2S_1$. D'autre part, dans $S_1$, il y a $i+1$ termes contenant $F_i$, et dans chacun de ces termes, $F_jF_k$ est majoré par $Z_{n/2}Z_{n/2-2i}$ (lemme 14). Une nouvelle utilisation du lemme 14 donne $A_iZ_{n/2}Z_{n/2-2i} \le Z_0Z_{n/2}Z_{n/2-i}$, d'où $S_1 \le Z_0Z_{n/2}\sum_{i<n/4}(i+1)Z_{n/2-i} = KZ_0Z_{n/2}(n/2)^{-3/2}\alpha^{n/2}\sum_i(i+1)(1-2i/n)^{-3/2}\alpha^{-i}$. Puisque $i < n/4$, le facteur $(1-2i/n)^{-3/2}$ est borné par $(1/2)^{-3/2}$, puis $\sum_i(i+1)\alpha^{-i}$ est aussi borné. Par suite, $S_1 \le K'Z_{n/2}n^{-3/2}\alpha^{n/2} = O(n^{-3}\alpha^n)$. Le dernier terme est prépondérant : $[z^{n-1}]f_n^4(z) \ge A_0[z^{n-1}]f_n^3(z)$. Or $[z^{n-1}]f_n^3(z)$ est une somme de $O(n^2)$ termes, chacun plus grand que $JY_{n/3}^3$ (lemme 14), donc $[z^{n-1}]f_n^4(z) = \Omega(n^{-5/2}\alpha^n)$. ∎

L'étude asymptotique des entiers $B_n$ a été faite par Pólya et Read, en utilisant des techniques d'analyse réelle et une formule d'intégration due à Abel.

**Théorème 12.** *[PR87, page 94] Le nombre d'alcanes ayant $n$ atomes de carbone vérifie*

$$B_n = \frac{\rho A(\rho)\lambda^3}{4\sqrt{\pi}} n^{-5/2}\rho^{-n}(1 + o(1)). \tag{5.15}$$

La constante $\frac{\rho A(\rho)\lambda^3}{4\sqrt{\pi}}$ vaut 0.65631869584183475026821711762108.

Nous indiquons ici une suite de calculs (dont la validité n'est pas entièrement justifiée) aboutissant au résultat du théorème ci-dessus. Ces développements pourraient ouvrir une nouvelle voie pour aboutir aux coefficients de puissances de séries tronquées.

Écrivons $f_n(z) = A(z) - \hat{f}_n(z)$, où $\hat{f}_n(z) = \sum_{k\ge n/2} A_kz^k$ est la différence entre la série $A(z)$ et le polynôme $f_n(z)$. Alors

$$f_n(z)^4 = A(z)^4 - 4A(z)^3\hat{f}_n(z) + 6A(z)^2\hat{f}_n(z)^2 - 4A(z)\hat{f}_n(z)^3 + \hat{f}_n(z)^4.$$

Les trois derniers termes ayant une valuation supérieure ou égale à $n$, le coefficient de $z^{n-1}$ dans $f_n(z)^4$ vaut :

$$[z^{n-1}]f_n(z)^4 = [z^{n-1}]\left(A(z)^4 - 4A(z)^3\hat{f}_n(z)\right).$$

La fonction $\hat{f}_n(z)$ ne différant de $A(z)$ que d'un polynôme, elle admet la même singularité $\rho$, et de plus son développement singulier en $z = \rho$ ne diffère de celui de $A(z)$ que par les puissances *entières* de $1 - z/\rho$ (puisque tout polynôme se décompose dans la base des $(1 - z/\rho)^k$) :

$$\begin{aligned} A(z) &= \kappa - \lambda(1 - \frac{z}{\rho})^{1/2} + \mu(1 - \frac{z}{\rho}) + \nu(1 - \frac{z}{\rho})^{3/2} + O((1 - \frac{z}{\rho})^2) \\ \hat{f}_n(z) &= \kappa_n - \lambda(1 - \frac{z}{\rho})^{1/2} + \mu_n(1 - \frac{z}{\rho}) + \nu(1 - \frac{z}{\rho})^{3/2} + O(\xi_n(1 - \frac{z}{\rho})^2) \end{aligned}$$

Le développement singulier de $h_n(z) = A(z)^4 - 4A(z)^3\hat{f}_n(z)$ au voisinage de $z = \rho$ est par suite

$$\begin{aligned} h_n(z) = \kappa^4 - 4\kappa^3\kappa_n + 12\kappa^2\lambda\kappa_n(1-\frac{z}{\rho})^{1/2} + (-4\kappa^3\mu_n - 6\kappa^2\lambda^2 - 12\kappa_n\mu\kappa^2 - 12\kappa_n\lambda^2\kappa + 4\kappa^3\mu)(1-\frac{z}{\rho}) \\ + (12\kappa^2\lambda\mu_n - 12\kappa_n\nu\kappa^2 + 24\kappa_n\mu\kappa\lambda + 4\kappa_n\lambda^3 + 8\lambda^3\kappa)(1-\frac{z}{\rho})^{3/2} + O(\xi_n(1-\frac{z}{\rho})^2). \end{aligned}$$

Si nous négligeons les termes dépendant de $n$, nous obtenons comme développement singulier

$$h(z) = \kappa^4 + (4\kappa^3\mu - 6\kappa^2\lambda^2)(1 - \frac{z}{\rho}) + 8\lambda^3\kappa(1 - \frac{z}{\rho})^{3/2} + O((1 - \frac{z}{\rho})^2).$$

Le premier terme singulier du développement de $h(z)$ est le terme en $(1 - z/\rho)^{3/2}$, et par transfert aux coefficients [FO90] :

$$[z^{n-1}]h(z) \sim 8\lambda^3\kappa[z^{n-1}](1 - \frac{z}{\rho})^{3/2}.$$

Le coefficient de $z^n$ dans $(1-z/\rho)^{3/2}$ vaut $\rho^{-n}n^{-5/2}/\Gamma(-3/2) + O(\rho^{-n}n^{-7/2})$, et $\Gamma(-3/2) = 4\sqrt{\pi}/3$, d'où

$$[z^{n-1}]h(z) \sim \frac{6\lambda^3\kappa}{\sqrt{\pi}} n^{-5/2}\rho^{-n+1}.$$

En divisant par 24 conformément au lemme 15, et comme $\kappa = A(\rho)$, nous retrouvons le résultat de Pólya et Read (5.15).

Le fait que l'on obtienne le même résultat n'est sans doute pas une coïncidence ; nous pensons au contraire qu'une explication profonde existe. Ainsi, la suite de calculs effectuée ici pourrait préfigurer une méthode générale d'extraction de coefficients dans les puissances de séries tronquées.

# Chapitre 6

# Conclusion

*Acta est fabula*

*Quels sont les livres qui valent l'arbre de leur papier ?*
Hubert Haddad

Nous avons étudié dans ce travail une des composantes (analyse algébrique) d'un procédé général d'analyse en moyenne d'algorithmes (la méthode des séries génératrices). Nous avons montré l'existence de classes entières de programmes ($\Pi$, $\hat{\Pi}$ et $\Pi_{\text{bool}}$) pour lesquelles l'analyse algébrique peut être effectuée de façon automatique, par application d'un certain nombre de règles bien précises. Nous avons aussi constaté que ces trois classes permettent de modéliser des problèmes non triviaux (chapitre 5). Enfin, et surtout, nous avons prouvé par de nombreux exemples que ce procédé d'analyse en moyenne peut être rendu *effectivement* automatique.

Les contributions originales de ce travail par rapport aux résultats déjà connus en analyse automatique d'algorithmes sont les suivantes : la caractérisation des algorithmes en *classes*, les algorithmes de décision du caractère bien fondé d'un programme, l'introduction d'un véritable langage pour décrire les algorithmes opérant sur des structures combinatoires, le calcul des opérateurs pour la sélection et l'itération dans les ensembles, multi-ensembles et cycles, les résultats de complexité pour le calcul des coefficients, l'introduction de schémas de programmation sur les objets étiquetés avec contrainte, l'analyse de fonctions à nombre fini de valeurs.

De plus, beaucoup de choses n'ont pas été racontées ici. Par exemple, la connaissance des séries de dénombrement permet de réaliser un générateur aléatoire de structures de données, avec probabilité uniforme parmi les objets de même taille. Un tel générateur a été implanté dans le système $\Lambda\Upsilon^{\Omega}$. Pour générer une expression aléatoire du programme de dérivation formelle, on écrit :

```
draw "expression" 10;;
plus zero times times times x one exp one x
```

et pour obtenir toutes les expressions de taille donnée :

```
#draw_all "expression" 2;;
exp zero
exp one
exp x
```

Couplé avec un interpréteur du langage ADL, ce générateur permettra d'effectuer des jeux de tests pour corroborer l'analyse automatique.

D'autre part, l'introduction de variables auxiliaires (appelées *marques*) en plus de la variable principale $z$ autorise le calcul des moments de la distribution de certains paramètres des structures de données. Cette fonctionnalité a également été implantée dans $\Lambda\Upsilon^{\Omega}$. Pour déterminer le nombre moyen de cycles d'une permutation aléatoire (premier moment), on marque les cycles par une variable auxiliaire :

```
type perm = set(mark[u] cycl);
     cycl = cycle(elem);
     elem = Latom(1);

to_analyze : moment(u,perm,1);
```

et l'analyse est effectuée automatiquement :

```
   Marked generating function for perm is:
     exp(u L(z))

   After differentiation    1    times, we obtain:
     - ln(1 - z) exp(- ln(1 - z))

Moment of order 1 associated to the mark u in perm is:
                                (ln(n)) + (O(1))
```

Cette discussion sur les moments nous amène naturellement à parler de distributions. Michèle Soria a montré dans sa thèse d'État [Sor89] qu'un grand nombre de constructions combinatoires et d'algorithmes admettent des lois limites simples et connues. De plus, ces lois limites sont aisément calculables d'après la forme des séries génératrices associées aux constructions. Ceci suggère l'extension du système $\Lambda\Upsilon^{\Omega}$, qui ne calcule que la moyenne, à un système $\Lambda\Upsilon^{\Omega}$++ visant à déterminer les distributions de coût d'algorithmes.

# Annexe A

# Règles sur les cycles orientés

*Il y a autant de mystère dans la science que dans la foi.*
Lamennais

Cette annexe rappelle d'abord les calculs élémentaires sur les séries de Dirichlet, notamment la formule d'inversion de Möbius (section A.1), puis la section A.2 décrit une preuve combinatoire de la règle de dénombrement des cycles orientés en univers non étiqueté (règle 7 page 52). Cette preuve a largement inspiré celles des règles de sélection et d'itération dans les cycles orientés (règles 18 et 19).

## A.1 Inversion de Möbius

**Lemme 16.** *(Inversion de Möbius) Soit* $A(z) = \sum_{n\geq 1} a_n z^n$ *et* $B(z) = \sum_{n\geq 1} b_n z^n$. *Alors*

$$\forall n, a_n = \sum_{d|n} b_d \Longleftrightarrow \forall n, b_n = \sum_{d|n} \mu(\frac{n}{d}) a_d \tag{A.1}$$

*et*

$$A(z) = \sum_{k\geq 1} B(z^k) \Longleftrightarrow B(z) = \sum_{k\geq 1} \mu(k) A(z^k)$$

*où* $\mu$ *est la fonction de Möbius définie par* $\mu(1) = 1$, $\mu(n) = 0$ *si* $n$ *est divisible par un carré,* $\mu(n) = (-1)^k$ *si* $n$ *est le produit de* $k$ *nombres premiers distincts.*

**Démonstration :** Il est facile de voir que les deux affirmations sont équivalentes ; l'une est formulée en termes de coefficients, et l'autre en termes de séries génératrices. Introduisons les séries de Dirichlet $\tilde{A}(s) = \sum_{n\geq 1} a_n/n^s$, $\tilde{B}(s) = \sum_{n\geq 1} b_n/n^s$ et $\zeta(s) = \sum_{n\geq 1} 1/n^s$ ($\zeta$ est la série de Dirichlet associée à la suite dont tous les termes valent 1). L'équation (A.1) n'est autre que la relation d'inversion :

$$\tilde{A}(s) = \tilde{B}(s)\zeta(s) \Longleftrightarrow \tilde{B}(s) = \frac{\tilde{A}(s)}{\zeta(s)}. \tag{A.2}$$

En effet, d'une part

$$\tilde{B}(s)\zeta(s) = \sum_{i,j\geq 1} \frac{b_i}{i^s j^s} = \sum_{n\geq 1} \left( \sum_{i|n} b_i \right) \frac{1}{n^s}$$

donc les parties gauches de (A.1) et (A.2) sont équivalentes. D'autre part, la fonction $\zeta(s)$ s'écrit aussi

$$\zeta(s) = \prod_{p \text{ premier}} \frac{1}{1 - \frac{1}{p^s}}$$

(décomposition d'Euler de la fonction $\zeta$), et

$$\frac{1}{\zeta(s)} = \prod_{p \text{ premier}} (1 - \frac{1}{p^s}) = (1 - \frac{1}{2^s})(1 - \frac{1}{3^s})(1 - \frac{1}{5^s}) \cdots = \sum_{n \geq 1} \frac{\mu(n)}{n^s}.$$

Il résulte que

$$\frac{\tilde{A}(s)}{\zeta(s)} = \sum_{i \geq 1} \frac{a_i}{i^s} \sum_{j \geq 1} \frac{\mu(j)}{j^s} = \sum_{n \geq 1} \left( \sum_{i|n} \mu(\frac{n}{i}) a_i \right) \frac{1}{n^s}.$$

donc les parties droites de (A.1) et (A.2) sont aussi équivalentes ; or (A.2) est vraie, donc (A.1) l'est également. ■

**Corollaire 5.**

$$A(z) = \sum_{k \geq 1} kB(z^k) \Longleftrightarrow B(z) = \sum_{k \geq 1} k\mu(k) A(z^k).$$

**Démonstration :** Soit $a'_n = a_n/n$ et $b'_n = b_n/n$ :

$$\begin{aligned}
A(z) = \sum_{k \geq 1} kB(z^k) &\iff \forall n, a_n = \sum_{k|n} k b_{\frac{n}{k}} \left( = \sum_{d|n} \frac{n}{d} b_d \right) \\
&\iff \forall n, a'_n = \sum_{d|n} b'_d \\
&\iff b'_n = \sum_{d|n} \mu(\frac{n}{d}) a'_d \qquad \text{(inversion de Möbius sur } A' \text{ et } B') \\
&\iff b_n = \sum_{d|n} \frac{n}{d} \mu(\frac{n}{d}) a_d \left( = \sum_{k|n} k\mu(k) a_{\frac{n}{k}} \right) \\
&\iff B(z) = \sum_{k \geq 1} k\mu(k) A(z^k).
\end{aligned}$$

■

La formule d'inversion de Möbius permet aussi de montrer des formules telles que

$$\sum_{p|k} \frac{\mu(p)}{p} = \frac{\phi(k)}{k}. \tag{A.3}$$

où $\phi$ est la fonction indicatrice d'Euler. En effet, soit $\tilde{B}$ la série de Dirichlet dont le terme général est $\mu(n)/n$ : $\tilde{B}(s) = \sum \mu(n)/n^{s+1} = \frac{1}{\zeta(s+1)}$. Alors la série $\tilde{B}(s)\zeta(s)$ vaut

$$\frac{\zeta(s)}{\zeta(s+1)} = \prod_{p \text{ premier}} \frac{1 - p^{-s-1}}{1 - p^{-s}} = \prod_{p \text{ premier}} (1 + (1 - \frac{1}{p})(\frac{1}{p^s} + \frac{1}{p^{2s}} + \frac{1}{p^{3s}} + \cdots)).$$

En développant mentalement le produit, on constate que le coefficient de $n^{-s}$, où $n$ s'écrit $p_1^{\alpha_1} \dots p_k^{\alpha_k}$ est $(1-1/p_1)\dots(1-1/p_k)$, c'est-à-dire $\phi(n)/n$. L'équation (A.3) n'est donc que la formulation en termes de coefficients de l'égalité

$$\frac{\zeta(s)}{\zeta(s+1)} = \sum_{n\geq 1} \frac{\phi(n)}{n^{s+1}}.$$

## A.2 Dénombrement des cycles orientés

Nous indiquons ici une preuve combinatoire de la règle 7, qui donne l'opérateur associé au constructeur cycle orienté en univers non étiqueté, pour les séries génératrices ordinaires. Cette preuve est due à Ph. Flajolet et M. Soria [FS91].

**Définition 25.** *Un mot $a_1 \dots a_k$ est dit* primitif *s'il n'est pas de la forme $\overbrace{\alpha \dots \alpha}^{k}$ où $\alpha$ est un mot et $k \geq 2$ (nous dirons alors que $\alpha$ est une racine de $a_1 \dots a_k$). Un cycle orienté $[a_1 \dots a_k]$ est dit* primitif *lorsque le mot $a_1 \dots a_k$ est primitif.*

La seconde partie de la définition a bien un sens car si l'un des mots représentant le cycle est non primitif, alors tous le sont. Désignons par $\mathcal{S}$ l'ensemble des mots non vides formés sur un alphabet $\mathcal{A}$, $\mathcal{PS}$ l'ensemble des mots non vides primitifs, $\mathcal{C}$ l'ensemble des cycles orientés et $\mathcal{PC}$ l'ensemble des cycles orientés primitifs.

**Lemme 17.** *[FS91] Tout mot a une unique racine primitive. Tout cycle a un unique cycle-racine primitif.*

Par exemple, le mot *aabaabaabaab* a deux racines, *aab* et *aabaab*, dont seule la première est primitive. Le cycle-racine est le cycle dont les mots qui le représentent sont les racines primitives des mots représentant le cycle initial. Par exemple, la racine primitive du cycle $[aabaabaabaab]$ est $[aab]$, qui peut se noter aussi $[aba]$ ou $[baa]$. De ces deux principes, il vient[1]

$$S(z,u) = \sum_{k=1}^{\infty} PS(z^k, u^k) \tag{A.4}$$

$$C(z,u) = \sum_{k=1}^{\infty} PC(z^k, u^k). \tag{A.5}$$

L'ensemble $\mathcal{S}$ s'écrit $\mathcal{S} = \mathcal{A}^+$, d'où l'on tire

$$S(z,u) = \frac{uA(z)}{1-uA(z)}.$$

En combinant avec l'égalité (A.4), et en utilisant la formule d'inversion de Möbius, il vient

$$PS(z,u) = \sum_{k=1}^{\infty} \mu(k) \frac{u^k A(z^k)}{1-u^k A(z^k)} \tag{A.6}$$

[1] Nous définissons la série génératrice ordinaire à deux variables $S(z,u) = \sum_{n,k} S_{n,k} z^n u^k$, où $S_{n,k}$ est le nombre de mots de $k$ lettres et de taille $n$.

A chaque cycle primitif de $k$ lettres sont associés $k$ séquences primitives distinctes, donc si $\mathcal{PC}$ désigne l'ensembles des cycles primitifs, $[z^n u^k]PC(z,u) = \frac{1}{k}[z^n u^k]PS(z,u)$, c'est-à-dire

$$PC(z,u) = \int_0^u PS(z,t)\frac{dt}{t},$$

ce qui donne en remplaçant $PS(z,t)$ par l'expression obtenue en (A.6)

$$PC(z,u) = \sum_{k=1}^{\infty} \frac{\mu(k)}{k} \log \frac{1}{1-u^k A(z^k)}.$$

Il suffit de remplacer cette expression de $PC(z,u)$ dans l'équation (A.5), et d'utiliser la formule (A.3) pour obtenir la règle 7.

NOTE : La formule d'énumération des cycles était sans doute connue de Pólya, mais la première trace écrite semble être due à Read [Rea61]. Par la suite, Robinson (1970) et Cadogan (1971) ont aussi utilisé la fonction $\mu(\cdot)$ de Möbius pour inverser des équations de séries génératrices (voir l'article [Lab86] de Gilbert Labelle pour une formulation en termes de séries indicatrices de cycle).

# Annexe B

# Différentes méthodes pour les calculs de coefficients

*C'est déjà trop que de se hausser sur la pointe des pieds.*
Proverbe persan

Le seul objectif de cette section est de montrer qu'il existe d'autres méthodes que celle utilisée dans la section 2.3, pour le calcul des coefficients des séries génératrices de dénombrement et des descripteurs de complexité. Ces algorithmes sont bien connus : en effet, lorsque les séries ont une forme explicite, il s'agit des algorithmes donnant leur développement de Taylor (en $z = 0$ ici). Ces algorithmes sont utilisés par tous les systèmes de calcul formel. Lorsque la série est donnée par une équation, il s'agit de trouver le développement de Taylor solution de cette équation (de par l'origine combinatoire de l'équation, l'existence de la solution est assurée). Nous présentons ces diverses méthodes sur un exemple ; le lecteur pourra se référer à [Knu81] pour une description générale des méthodes classiques, et à [BK75, BK78] pour les méthodes rapides et les questions de complexité. En ce qui concerne le domaine plus précis des équations dérivées d'objets combinatoires, en l'occurrence les espèces de structures, on pourra aussi consulter [BLL88].

Considérons par exemple l'équation fonctionnelle suivante, qui compte des arbres généraux non planaires étiquetés dont les branches sont soit des arbres, soit des paires d'arbres :

$$y = z \exp(y + y^2). \tag{B.1}$$

Comment calculer rapidement les $n$ premiers termes du développement de Taylor de $y$ en $z = 0$ ?

## B.1 Méthode naïve

La première méthode qui vient à l'esprit est d'itérer l'équation (B.1). Si l'on connaît les $k$ premiers termes du développement de Taylor de $y$, alors une étape supplémentaire donne un terme de plus. En partant de $y = 0$ ($k = 0$), il faut donc $n$ étapes pour obtenir les $n$ premiers termes. Une étape partant de $k$ termes nécessite le développement de Taylor de l'exponentielle d'une série de $k$ termes, ce qui peut se faire par la formule

$$f = O(z) \Longrightarrow \exp(f) = 1 + f + \frac{f^2}{2!} + \cdots + \frac{f^k}{k!} + O(z^{k+1}).$$

En utilisant le schéma de Hörner pour calculer $1+f+\cdots+f^k/k!$, le nombre d'opérations est $O(k^3)$, et le coût total jusqu'à l'ordre $n$ est donc $O(1^3+2^3+\cdots+n^3)=O(n^4)$. Le programme MAPLE correspondant à cette méthode est le suivant :

```
naif := proc(n)
local y,k;
   y := 0;
   for k to n do
      y := series(z*exp(y+y^2),z,k+1);
   od;
end:
```

En une minute, sur un Sun 3/60, ce programme nous a permis de calculer les coefficients de la série génératrice solution de (B.1) jusqu'à l'ordre 33.

$$y = z + z^2 + \frac{5\,z^3}{2} + \frac{20\,z^4}{3} + \cdots + \frac{7546382341257820875757287344283620208643353\,z^{33}}{1818648639354290831360000} + O(z^{34}).$$

## B.2 Méthode de la dérivée

C'est la méthode utilisée par l'algorithme **Calcul des coefficients** (page 74). Le principe consiste à dériver l'équation fonctionnelle vérifiée par $y$. Comme les dérivées de $E(f)=\exp(f)$, $Q(f)=1/(1-f)$ et $L(f)=\log(1/(1-f))$ s'expriment simplement en fonction de ces mêmes opérateurs, on obtient ainsi une relation de récurrence entre les coefficients de $y$ et de fonctions auxiliaires. Dans le cas de l'équation (B.1), soit $f=y+y^2$ :

$$\text{pour } k\ge 2,\quad [z^k]z\exp(y+y^2)=[z^{k-1}]\exp(f)=\frac{1}{k-1}[z^{k-2}](\exp(f))'=\frac{1}{k-1}[z^{k-2}]\exp(f)f'.$$

Or $\exp(f)$ s'écrit aussi $y/z$, ce qui conduit à la récurrence

$$y_0=f_0=0,\quad y_1=f_1=1,\quad y_k=\frac{1}{k-1}\sum_{j=1}^{k-1} j f_j y_{k-j},\quad f_k=y_k+\sum_{j=1}^{k-1} y_j y_{k-j}.$$

Cette méthode nécessite deux convolutions de $k$ termes à chaque étape. Le coût total pour calculer tous les coefficients jusqu'à l'ordre $n$ est par suite $O(1+2+\cdots+n)=O(n^2)$. Voici un programme MAPLE basé sur cette méthode

```
deriv := proc(n)
local y,f,k;
   y[1]:=1; f[1]:=1;
   for k from 2 to n do
      y[k] := 1/(k-1) * sum(j*f[j]*y[k-j],j=1..k-1);
      f[k] := y[k] + sum(y[j]*y[k-j],j=1..k-1);
   od;
   series(sum(y[j]*z^j,j=1..n)+O(z^(n+1)),z,n+1)
end:
```

qui nous a permis d'aller jusqu'à $n=58$ en une minute (toujours sur un Sun 3/60).

## B.3 Méthode de Newton

Le méthode de Newton est un procédé général pour obtenir des approximations successives de solutions d'équations, avec convergence quadratique. Kung et Brent ont montré que la méthode de Newton s'applique aussi au problème de la composition et de l'inversion (pour la loi de composition) de séries formelles [BK75]. Or trouver la série formelle $y(z)$ solution d'une équation $\Psi(y) = 0$ consiste justement à inverser la fonction $y \to z + \Psi(y)$. Soit $f(y) = 0$ l'équation dont $y$ est solution : ici, $f(y) = y - z\exp(y + y^2)$. En partant de la série $y_0 = 0$, nous définissons la suite $y_n$ par la récurrence

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}. \tag{B.2}$$

La différence entre $y_{n+1}$ et $y$ s'écrit alors

$$y_{n+1} - y = \frac{y_n f'(y_n) - f(y_n) - y f'(y_n)}{f'(y_n)} \sim \frac{(y_n - y)^2}{2}\frac{f''(y_n)}{f'(y_n)} \tag{B.3}$$

en faisant un développement de Taylor de $f$ en $y_n$. Ce calcul montre que l'écart entre $y_n$ et $y$ évolue de façon quadratique avec $n$. Plus précisément, lorsque l'équation définissant $y$ est du type $y = z\phi(y)$, où $\phi$ admet un développement de Taylor à tout ordre en $z = 0$, alors

$$y_{n+1} - y \sim -z\frac{(y_n - y)^2}{2}\frac{\phi''(y_n)}{1 - z\phi'(y_n)}.$$

Par conséquent, si $y_n - y = O(z^k)$, alors $y_{n+1} - y = O(z^{2k+1})$. Le nombre d'étapes nécessaires pour calculer les $n$ premiers termes est donc $\lceil \log_2(n+1) \rceil = O(\log n)$.

Le passage de $y_n$ à $y_{n+1}$ par la formule (B.2) demande le calcul de $f(y_n)$, celui de $f'(y_n)$, puis la division des deux développements, et enfin la soustraction à $y_n$. Tous ces calculs doivent être faits avec le nombre exact de termes attendus pour $y_{n+1}$.

Si l'on utilise des algorithmes classiques sur les séries [Knu81, page 506] en $O(n^2)$ pour les opérations nécessitées par le calcul de $f(y_n)$ et $f'(y_n)$, le calcul des $n$ premiers coefficients coûte

$$O(1^2 + 2^2 + 4^2 + \cdots + (\frac{n}{2})^2 + n^2) = O(\frac{4}{3}n^2) = O(n^2) \tag{B.4}$$

et l'on n'a rien gagné par rapport à la méthode de la dérivée. Le programme MAPLE ci-dessous utilise la méthode de Newton avec des algorithmes classiques (la fonction auxiliaire `s_newton` détermine la suite optimale des valeurs de $k$, afin que le dernier calcul donne juste la précision voulue).

```
newton := proc(n)
local y,k,l;
   l := s_newton(n);
   y := 0; k := 1; # erreur = O(z^k)
   while k<=n do
      k := op(1,l); l := subsop(1=NULL,l);
      y-(z*exp(y+y**2)-y)/(z*(1+2*y)*exp(y+y**2)-1);
      y := convert(series(",z,k),polynom);
   od;
   time()-st,series(y+z^k,z,k);
end:
```

```
s_newton := proc(n)
local l,k;
   k := n+1;  l := [];
   while k>1 do l := [k,op(l)]; k := iquo(k,2) od;
   l
end:
```

En une minute de temps machine, ce programme nous a permis d'aller jusqu'à $n = 49$.

Dans le coût (B.4), nous constatons que tout le coût est concentré dans le dernière étape, les autres étapes ne contribuant qu'à un quart du coût total. Plus généralement, si les opérations effectuées pour calculer $y_n - f(y_n)/f'(y_n)$ à l'ordre $n$ coûtent $O(M(n))$ où $M$ est une fonction sur-linéaire ($M(2n) \geq 2M(n)$), alors le calcul des $n$ premiers coefficients par la méthode de Newton coûte

$$O(M(1) + M(2) + M(4) + \cdots + M(\frac{n}{2}) + M(n)) = O(\ldots + \frac{1}{4}M(n) + \frac{1}{2}M(n) + M(n)) = O(M(n)).$$

Par exemple, $M(n)$ vaut $n \log n$ quand on utilise la transformée de Fourier rapide [BK78]. Le coût du calcul des coefficients est alors réduit à $O(n \log n)$, et la méthode devient intéressante par rapport à la méthode de la dérivée.

G. Labelle, Décoste et Leroux ont mis au point une variante de la méthode de Newton qui donne $2n + 2$ (au lieu de $2n + 1$ ici) termes exacts en partant de $n$, et surtout qui présente l'avantage de se prêter à une interprétation combinatoire [DLL82].

L'interprétation combinatoire, déjà rencontrée à la fin du chapitre 3, est utile pour mieux comprendre les équations que l'on manipule, et peut éventuellement inspirer des isomorphismes entre structures de données. Ainsi, une certaine dualité existe entre objets combinatoires et séries génératrices ; par exemple, Leroux et Viennot ont montré que la propriété de séparation des variables d'une équation différentielle s'interprète au niveau des structures de données [LV88b].

## B.4 Cas des familles simples d'arbres

Nous présentons ici une méthode due à Comtet, qui permet de calculer les coefficients jusqu'à l'ordre $n$ en $O(n)$ opérations, dans le cas où l'équation (B.1) est de la forme $y = Q(z, y)$, $Q$ étant un polynôme. Nous partons de la forme plus générale

$$P(z, y) = 0,$$

où $P(z, y)$ est un polynôme de degré $d$ en $y$. Par dérivation, cette équation donne une relation entre $y$ et $y'$ qui permet d'exprimer $y'$ comme une fraction rationnelle en $y$ et $z$ :

$$y' = -\frac{\frac{\partial P(z,y)}{\partial z}}{\frac{\partial P(z,y)}{\partial y}}. \tag{B.5}$$

Appliquons alors l'algorithme d'Euclide à $\frac{\partial P(z,y)}{\partial y}$ et $P(z, y)$, par rapport à la variable $y$. Lorsque le polynôme de départ n'a que des racines simples, le plus grand commun diviseur est une fraction rationnelle en $z$ :

$$u(z, y)\frac{\partial P(z, y)}{\partial y} + v(z, y)P(z, y) = w(z).$$

Comme $P(z,y)=0$, il s'ensuit par combinaison avec (B.5) que

$$y' = -\frac{1}{w(z)}\frac{\partial P(z,y)}{\partial z}u(z,y).$$

Par division euclidienne de $\frac{\partial P(z,y)}{\partial z}u(z,y)$ par $P(z,y)$, la fonction $y'$ s'exprime comme $y'=F_1(z,y)$, où $F_1(z,y)$ est un polynôme en $y$, de degré inférieur à $d$, dont les coefficients sont des fractions rationnelles en $z$.

Le même procédé s'applique aux dérivées d'ordre supérieur de $y$, et on obtient finalement un système d'équations

$$\begin{cases} y' &= F_1(z,y) \\ y'' &= F_2(z,y) \\ &\vdots \\ y^{(d-1)} &= F_{d-1}(z,y) \end{cases}$$

où $F_j(z,y)$ est un polynôme en $y$, de degré inférieur à $d$, dont les coefficients sont des fractions rationnelles en $z$. Ce système est un système linéaire de $d-1$ équations à $d-1$ inconnues $y$, $y^2$, ..., $y^{d-1}$. Il permet donc d'exprimer $y$ de façon linéaire en fonction des $y^{(j)}$ :

$$f_0(z)+f_1(z)y+f_2(z)y'+f_3(z)y''+\cdots+f_d(z)y^{(d-1)}=0. \tag{B.6}$$

Cette forme conduit à une récurrence linéaire pour les coefficients $y_n$. Cette récurrence autorise le calcul de $y_0, \ldots, y_n$ en $O(n)$ opérations.

Le programme MAPLE ci-dessous réalise cette méthode dans le cas où $d=2$. La procédure `comtet` prend une équation de la forme $y=Q(z,y)$ et retourne la récurrence linéaire vérifiée par $y$, en affichant au passage la combinaison linéaire (B.6) qui s'annule. La procédure auxiliaire `reduit` se charge de réduire la fraction rationnelle en une combinaison linéaire, et `sg2coeff` transforme en coefficients les termes $Cz^k y^{(j)}$ de cette combinaison linéaire.

```
comtet := proc(eq)
local y,z,P,d,PP,comb;
   y := op(1,eq);  P := op(2,eq)-op(1,eq);
   z := op(indets(P) minus{y});  d := degree(P,y);
   if d=2 then
      PP := diff(subs(y=y(z),P),z);
      solve(",diff(y(z),z));
      subs(y(z)=y,");
      comb := reduit(diff(y(z),z)=",P,y,z);
      print(comb);
      map(sg2coeff,comb);
      y(n)=collect(solve(",y(n)),[y(n-1),y(n-2)]);
   fi;
end:

reduit := proc(eq,P,y,z)
local id,F,g,u,v;
      id := op(1,eq);  F := op(2,eq);
      g := gcdex(P,denom(F),y,'u','v');
      normal(numer(F) * v/g);
```

```
      rem(numer("),P,y)/denom(");
      subs(y=y(z),normal("));
      numer(")-id*denom(");
      expand(");
end:

sg2coeff := proc(expr)
local z,y,C,k,j,e;
   z := op(indets(expr,name));
   indets(expr,function); if "={} then RETURN(0) fi;
   subs(diff=<x>,");  y := op(0,op("));
   subs(diff=proc(e) if type(e,`^`) then e*op(1,e) else op(0,e)^2 fi end,expr);
   e := subs(y(z)=y,");
   k := degree(e,z);  j := degree(e,y)-1;  C := e/z^k/y^(j+1);
   C * product(n-k+'l','l'=1..j) * y(n+j-k);
end:
```

Étudions cette méthode sur le type `expression` du programme de dérivation formelle (page 14). Les règles du chapitre 2 nous ont conduit à l'équation suivante pour la série génératrice ordinaire $E(z)$ associée au type `expression` (voir page 53) :

$$E(z) = 3z + zE(z) + 2zE^2(z).$$

Cette équation fonctionnelle est bien du genre $P(z, E) = 0$, donc la méthode de Comtet s'applique :

```
> comtet( E = 3*z + z*E + 2*z*E^2 );
                                3 /  d      \      2 /  d      \     /  d      \
  - z E(z) + E(z) - 6 z - 23 z  |---- E(z)| - 2 z  |---- E(z)| + z |---- E(z)|
                                \ dz      /        \ dz      /     \ dz      /

                        (1 - 2 n) E(n - 1)   (- 23 n + 46) E(n - 2)
              E(n) = - ------------------ - ----------------------
                              1 + n                  1 + n
```

Cette méthode s'applique aussi au calcul des coefficients des descripteurs de complexité, ce qui permet de calculer le coût moyen exact sur les données de taille $n$ en $O(n)$ opérations sur les coefficients. Plus précisément, nous nous rappelons que les équations des descripteurs de complexité $\tau D$ et $\tau C$ des procédures `diff` et `copy` étaient linéaires en $\tau D$ et $\tau C$. L'élimination de $\tau C$ conduit pour $\tau D(z)$ à une expression rationnelle en fonction de $E(z)$ :

$$\tau D(z) = \frac{6z + 12z^2E(z) + 2zE(z) + 3z^2 - E(z)}{(1 - z - 4zE(z))^2}.$$

Nous pouvons utiliser la procédure `reduit` pour transformer cette équation en une dépendance linéaire entre $\tau D(z)$ et $E(z)$ :

```
> reduit(tauD(z)=(6*z+12*z^2*E+2*z*E+3*z^2-E)/(-1+z+4*z*E)^2,3*z+z*E+2*z*E^2-E,E,z);
             2                          2                        2
 - 6 z - 12 z  E(z) - 2 z E(z) - 3 z  + E(z) - 23 tauD(z) z  - 2 tauD(z) z + tauD(z)
```

puis `sg2coeff` pour passer aux coefficients :

```
> map(sg2coeff,");
    - 12 E(n - 2) - 2 E(n - 1) + E(n) - 23 tauD(n - 2) - 2 tauD(n - 1) + tauD(n)
```

Il ne nous reste plus qu'à écrire un programme MAPLE utilisant les deux récurrences linéaires obtenues :

```
cout_moyen := proc(N)
local n,E,tauD;
   E[1]:=3; E[2]:=3;
   tauD[1]:=3; tauD[2]:=12;
   for n from 3 to N do
      E[n] := ((2*n-1)*E[n-1]+(23*n-46)*E[n-2])/(n+1);
      tauD[n] := -E[n] + 2*E[n-1] + 12*E[n-2] + 2*tauD[n-1] + 23*tauD[n-2];
   od;
   tauD[N]/E[N]
end:
```

C'est grâce à ce programme (donc à la méthode de Comtet) que nous avons pu déterminer le coût moyen exact du programme de dérivation formelle sur les expressions de taille 1000 en moins d'une minute (page 78).

## B.5 Schémas de sélection en univers non étiqueté

### B.5.1 Sélection dans un ensemble

L'une des étapes de la preuve de la règle correspondante (règle 15) donne

$$\tau P(z) = \sum_{n=1}^{\infty} \tau Q_n z^n \sum_{k=1}^{\infty} \frac{1}{k}[u^k] \frac{uB^{\mathcal{S}}(z,u)}{1+uz^n}$$

d'où

$$\tau P_i = \sum_{n=1}^{\infty} \tau Q_n \sum_{k=1}^{\infty} \frac{1}{k}[z^{i-n}u^{k-1}] \frac{B^{\mathcal{S}}(z,u)}{1+uz^n}.$$

Soit $B_{n,k}$ le coefficient de $z^n u^k$ dans $B^{\mathcal{S}}(z,u)$. Comme $B_{n,k}$ est nul lorsque $k > n$ (le nombre de composantes d'un ensemble est toujours inférieur à sa taille), l'intervalle de sommation de $k$ peut être limité à $[1, i+1-n]$ :

$$\tau P_i = \sum_{n=1}^{i} \tau Q_n \sum_{k=1}^{i+1-n} \frac{1}{k}(B_{i-n,k-1} - B_{i-2n,k-2} + B_{i-3n,k-3} - \cdots + (-1)^{j+1} B_{i-jn,k-j} + \cdots) \quad \text{(B.7)}$$

où la somme entre parenthèses s'arrête lorsque $k-j$ devient négatif où lorsque $i-jn$ devient plus petit que $k-j$. Par exemple pour $i \leq 4$ :

$$\begin{aligned}
\tau P_1 &= \tau Q_1 \\
\tau P_2 &= \tau Q_1 \frac{B_{1,1} - B_{0,0}}{2} + \tau Q_2 \\
\tau P_3 &= \tau Q_1 \left[\frac{B_{2,1}}{2} + \frac{B_{2,2} - B_{1,1} + B_{0,0}}{3}\right] + \tau Q_2 \frac{B_{1,1}}{2} + \tau Q_3 \\
\tau P_4 &= \tau Q_1 \left[\frac{B_{3,1}}{2} + \frac{B_{3,2} - B_{2,1}}{3} + \frac{B_{3,3} - B_{2,2} + B_{1,1} - B_{0,0}}{4}\right] + \tau Q_2 \left[\frac{B_{2,1} - B_{0,0}}{2} + \frac{B_{2,2}}{3}\right] + \tau Q_3 \frac{B_{1,1}}{2} + \tau Q_4.
\end{aligned}$$

Les $B_{n,k}$ sont des polynômes de degré $k$ en les $B_j$ : $B_{0,0} = 1$ et

$$\begin{array}{llll} B_{1,1} = B_1 & & & \\ B_{2,1} = B_2 & B_{2,2} = \frac{B_1(B_1-1)}{2} & & \\ B_{3,1} = B_3 & B_{3,2} = B_1 B_2 & B_{3,3} = \frac{B_1(B_1-1)(B_1-2)}{6} & \\ B_{4,1} = B_4 & B_{4,2} = B_1 B_3 + \frac{B_2(B_2-1)}{2} & B_{4,3} = \frac{B_1(B_1-1)}{2} B_2 & B_{4,4} = \frac{B_1(B_1-1)(B_1-2)(B_1-3)}{24} \end{array}$$

### B.5.2 Calcul des $B_{n,k}$

Le calcul de $\tau P_i$ par la formule (B.7) nécessite d'abord le calcul des $B_{n,k}$ pour $1 \le k \le n \le i-1$, qui sont définis par

$$B^{\mathcal{S}}(z,u) = \prod_{j=1}^{\infty} (1+uz^j)^{B_j} = \sum_{n,k} B_{n,k} z^n u^k. \tag{B.8}$$

Pour calculer ces coefficients, nous utilisons la méthode de la dérivée, déjà utilisée pour le dénombrement des ensembles (page 74).

$$\frac{\partial B^{\mathcal{S}}(z,u)}{\partial z} = B^{\mathcal{S}}(z,u) \sum_{j=1}^{\infty} \frac{j B_j u z^{j-1}}{1+uz^j}$$

et par conséquent

$$\sum_{k=1}^{n} B_{n,k} u^k = [z^n] B^{\mathcal{S}}(z,u) = \frac{1}{n} [z^{n-1}] \frac{\partial B^{\mathcal{S}}(z,u)}{\partial z} = \frac{1}{n} [z^{n-1}] \left( B^{\mathcal{S}}(z,u) \sum_{j=1}^{n} \frac{j B_j u z^{j-1}}{1+uz^j} \right). \tag{B.9}$$

Nous avons limité l'intervalle de sommation de $j$ à $[1 \ldots n]$ car les autres facteurs donnent des termes de degré plus grand que $n-1$ en $z$. A $j$ fixé, le nombre de termes de $B^{\mathcal{S}}(z,u)$ qui, multipliés par $z^{j-1}/(1+uz^j)$, donnent un terme en $z^{n-1}$ est $(n-j)+(n-2j)+(n-3j)+\cdots+(n-\lfloor \frac{n}{j} \rfloor j) = \sum_{1 \le ij \le n} (n-ij)$. Le nombre total de produits à effectuer dans (B.9) est par suite

$$\sum_{j=1}^{n} \sum_{1 \le ij \le n} (n-ij) = \sum_{k=1}^{n} d(k)(n-k)$$

où $d(k)$ est le nombre de diviseurs de l'entier $k = ij$. Soit $D(n) = d(1) + d(2) + \cdots + d(n)$ ; la précédente somme s'écrit aussi $\sum_{k=1}^{n-1} D(k)$. Or

$$D(n) = n \log n + (2\gamma - 1)n + O(\sqrt{n})$$

[HW79, théorème 320 page 264], d'où l'on déduit

$$D(1) + D(2) + \cdots + D(n) = \frac{1}{2} n^2 \log n + (\gamma - \frac{3}{4}) n^2 + O(n^{3/2}).$$

Par conséquent, le calcul des $n$ entiers $B_{n,k}$ par (B.9) coûte $O(n^2 \log n)$ en temps, et utilise un espace de $O(n^2)$ (l'unité étant la place nécessaire à un coefficient).

### B.5.3 Calcul des $\tau P_i$

Une fois que les $B_{n,k}$ sont connus, il nous reste à appliquer la formule (B.7) pour calculer les $\tau P_i$. Le nombre de termes de

$$S_{i,n,k} = B_{i-n,k-1} - B_{i-2n,k-2} + \cdots + (-1)^{j+1} B_{i-jn,k-j} + \cdots$$

étant inférieur à $i/n$, le nombre de termes dont $\tau P_i$ est la somme est inférieur à

$$\sum_{n=1}^{i} \sum_{k=1}^{i+1-n} \frac{i}{n} \leq i^2 \sum_{n=1}^{i} \frac{1}{n} = O(i^2 \log i).$$

Le calcul de $\tau P_i$ par (B.9) et (B.7) coûte donc $O(i^2 \log i)$, et le calcul de tous les $\tau P_i$ jusqu'à $i = n$ coûte

$$O(\sum_{i=1}^{n} i^2 \log i) = O(n^3 \log n).$$

REMARQUE : Les sommes $S_{i,n,k}$ peuvent se calculer au fur et à mesure. En effet, nous avons la relation $S_{i,n,k} = B_{i-n,k-1} - S_{i-n,n,k-1}$. Par cette méthode, le calcul de chaque $S_{i,n,k}$ coûte une opération et déterminer $\tau P_i$ coûte

$$\sum_{n=1}^{i} \sum_{k=1}^{i+1-n} 1 = O(i^2).$$

Cette méthode permet donc de gagner un facteur $\log N$ dans le calcul des $\tau P_i$, qui devient en $O(N^3)$, mais le coût du calcul des $B_{n,k}$ restant en $O(N^3 \log N)$, le coût total est inchangé. De plus, le stockage des $S_{i,n,k}$ nécessite un espace mémoire en $O(n^3)$, au lieu de $O(n^2)$ pour la méthode décrite ci-dessus.

### B.5.4 Sélection dans un multi-ensemble

Comme pour le constructeur ensemble, l'une des étapes de la preuve de la règle 17 donne $\tau P(z)$ en fonction des $\tau Q_n$ et de la série à deux variables $B^{\mathcal{M}}(z,u)$ :

$$\tau P(z) = \sum_{n=1}^{\infty} \tau Q_n z^n \sum_{k=1}^{\infty} \frac{1}{k} [u^k] \frac{u B^{\mathcal{M}}(z,u)}{1 - uz^n}.$$

Il en résulte

$$\tau P_i = \sum_{n=1}^{\infty} \tau Q_n \sum_{k=1}^{\infty} \frac{1}{k} [z^{i-n} u^{k-1}] \frac{B^{\mathcal{M}}(z,u)}{1 - uz^n}$$

et

$$\tau P_i = \sum_{n=1}^{i} \tau Q_n \sum_{k=1}^{i+1-n} \frac{1}{k} (B_{i-n,k-1} + B_{i-2n,k-2} + B_{i-3n,k-3} + \cdots + B_{i-jn,k-j} + \cdots)$$

où ici $B_{n,k}$ est le nombre de *multi-ensembles* de taille $n$ et de $k$ composantes, qui s'obtient par l'égalité

$$\sum_{k=1}^{n} B_{n,k} u^k = \frac{1}{n} [z^{n-1}] \left( B^{\mathcal{M}}(z,u) \sum_{j=1}^{n} \frac{j B_j u z^{j-1}}{1 - uz^j} \right).$$

Toutes ces équations sont similaires à celles obtenues pour le constructeur ensemble, et le coût des coefficients (aussi bien des $B_{n,k}$ que des $\tau P_i$) a même complexité. Cette similitude s'explique par l'égalité :

$$B^{\mathcal{M}}(z,u) = \frac{1}{B^{\mathcal{S}}(z,-u)} = (-B)^{\mathcal{S}}(z,-u).$$

# Annexe C

# Au cœur du système $\Lambda\Upsilon\Omega$

*J'ai bien des choses à vous écrire, pourtant je n'ai pas voulu le faire avec du papier et de l'encre.*

Deuxième épître de Jean, 12

Nous détaillons ici l'implantation de l'analyseur algébrique "ALAS" du système $\Lambda\Upsilon\Omega$, et du module de résolution "SOLVER" (voir schéma page 11). L'analyseur algébrique est écrit dans le langage CAML [WAL+87] ; il prend en entrée un programme Adl et produit en sortie les équations vérifiées par les séries génératrices et les descripteurs de complexité du programme. Le module de résolution est écrit dans le langage MAPLE [CGG+88] ; il prend en entrée un système d'équations (produit par l'analyseur algébrique) et essaie de résoudre ce système, en éliminant les solutions ne correspondant pas à de véritables séries génératrices.

## C.1 L'analyseur algébrique (ALAS)

Lorsqu'on lance le système $\Lambda\Upsilon\Omega$, on entre en fait sous l'interprète du langage CAML, avec une session MAPLE en arrière-plan :

```
% luo
Luo V1.3 Mon Sep 3 15:09:50 MET DST 1990

Please send bugs or remarks to luo@inria.inria.fr

Initializing maple ...
For help about Lambda-Upsilon-Omega, type help "";;

() : unit

/usr/local/lib/luo/V1.3/Caml/luoinit.ml loaded

#
```

(le caractère `#` est le signe d'invite de l'interprète CAML). Pour analyser un fichier Adl de nom `diff.adl`, on écrit alors `analyze "diff";;` (il faut deux points-virgules pour faire évaluer une suite de commandes à CAML). Voilà ce qui se passe dès lors : le fichier `diff.adl` est analysé

syntaxiquement (section C.1.1), puis l'analyseur algébrique transforme l'arbre de syntaxe obtenu en équations de séries génératrices (section C.1.3). Ces équations sont ensuite écrites dans la syntaxe MAPLE pour être passées au module de résolution (section C.2).

### C.1.1 Analyse syntaxique

L'analyse syntaxique est grandement facilitée par l'interface entre CAML et le méta-analyseur syntaxique YACC. L'on définit tout d'abord des types CAML pour représenter les différentes parties du programme (syntaxe abstraite) :

```
type instruction = Cases of string list & (pattern list & instruction) list
                 | Casetype of string & (pattern list & instruction) list
                 | Instructions of instruction list
                 | IfThenElse of condition & instruction & instruction
                 | Forall of string & string & instruction
                 | Forone of string & string & instruction
                 | Expression of expression
                 | Repeat of instruction & repeat
                 | Empty_instr ;;
```

On définit la syntaxe concrète du programme, et l'on crée directement les objets CAML par des "actions" :

```
grammar for values adl =
        ...
and instruction = parse
  Literal "case"; ident_list d2; Literal "of"; cases d4; Literal "end" -> Cases (d2,d4)
| Literal "casetype"; IDENT d2; Literal "of"; cases d4; Literal "end" -> Casetype (d2,d4)
| Literal "begin"; instructions d2; Literal "end" -> Instructions d2
| Literal "if"; condition d2; Literal "then"; instruction d4; Literal "else";
    instruction d6 -> IfThenElse (d2,d4,d6)
| Literal "forall"; IDENT d2; Literal "in"; IDENT d4; Literal "do"; instruction d6
    -> Forall (d2,d4,d6)
| Literal "forone"; IDENT d2; Literal "in"; IDENT d4; Literal "do"; instruction d6
    -> Forone (d2,d4,d6)
| expression d1 -> Expression d1
| Literal "to"; repeat d2; Literal "do"; instruction d4 -> Repeat (d4,d2)
| -> Empty_instr
```

On obtient ainsi facilement l'arbre de syntaxe d'un programme Adl (les lignes commençant par le caractère **#** sont entrées à la main, les suivantes sont produites par CAML) :

```
#<< type a = atom(1); L = sequence(a);
#
#   procedure len (l : L);
#   begin
#      forall x in l do
#         count;
#   end;
#
#   measure count : 2; >>;;
(File
```

```
([("count",(NumCost 2))],
 [("len",[("l",(Ident ("L",No)))],"",
   (Instructions
    [(Forall ("x","l",(Expression (Exp_id "count")))); Empty_instr]))],
 [])) :
adl_file
```

### C.1.2 Vérification du caractère bien fondé

A partir de l'arbre de syntaxe, on commence par vérifier le caractère bien fondé du programme. Dans l'actuelle version de développement (version V1.4), le calcul des valuations (algorithme A page 28) est effectué, et l'on vérifie que les multi-constructeurs (*sequence*, *set*, *multiset*, *cycle*) ne sont pas appliqués à des ensembles de valuations nulle. On obtient par exemple pour le programme de dérivation formelle (page 14) :

```
% luo V1.4
Luo V1.4 Fri Dec 7 19:10:53 MET 1990
                                    ...
#printlevel:=3; analyze "diff";;
                                    ...
Checking type declarations ...
The valuation of expression is 1
The valuation of zero is 1
The valuation of one is 1
The valuation of x is 1
The valuation of plus is 1
The valuation of times is 1
The valuation of exp is 1
```

La détection des cycles dans les définitions de structures de données et de procédures (algorithmes B page 33 et C page 41) n'est par contre pas encore implantée.

### C.1.3 Application des règles

Les règles d'analyse des structures de données et des procédures sont regroupées dans un même fichier de moins de 200 lignes. A chaque constructeur est associée une fonction CAML :

```
let gf_sequence gftype A c = match c with        (* doesn't depend on gftype *)

      All     -> Op("Q",A)
    | Eq(k)     -> Power(A,k)
    | Le(k)     -> Ratio(OneMinus(Power(A,k+1)),OneMinus(A))
    | Odd     -> Times(A,Op("Q",Power(A,2)))
    | Even     -> Op("Q",Power(A,2))
    | Ge(all,k)     -> Times(Power(A,k),Op("Q",A))
    | None     -> Gf_num(0)
    | _             -> pm "gf_sequence" "other type of card" ;;
```

et il en est de même pour chaque schéma d'itération ou de sélection sur un constructeur précis :

```
let compl_forall_sequence gftype tq A c =
if gftype = "ordinary" or gftype = "exponential" then (match c with
```

```
        All       -> Times(tq,Power(Op("Q",A),2))
      | Eq k -> Times(Gf_num(k),Times(tq,Power(A,k-1)))
      | Ge(All,1) -> Times(tq,Power(Op("Q",A),2))
      | Ge(All,2) -> Times(tq,Minus(Power(Op("Q",A),2),One))
      | Le k -> Times(tq,Tder(A,k))
      | _         -> nyi "compl_forall_sequence" "card other than All")
  else err "compl_forall_sequence"
           "type is neither ordinary nor exponential" gftype;;
```

Nous constatons au passage qu'une grande partie du code est consacrée à différentes restrictions possibles sur la longueur des multi-constructeurs (longueur paire ou impaire, égale, supérieure ou inférieure à $k$). La règle générale (sans restriction) est celle correspondant au type `All`.

Dans l'état actuel, sont implantées en univers non étiqueté toutes les règles de dénombrement sauf celle sur les cycles (règles 1 à 6), ainsi que les règles de sélection et d'itération dans un produit cartésien ou une séquence (règles 8 à 13). En univers étiqueté, sont implantées toutes les les règles de dénombrement (règles 22 à 28), et toutes les règles de sélection et d'itération sauf celles sur les cycles non orientés (règles 29 à 37 et 39). Pour ce qui est des contraintes sur les étiquettes en univers étiqueté, la règle 41 (enracinement du minimum) et les règles 42 et 43 (descente dans un produit partitionnel avec enracinement du minimum) sont implantées. Enfin, pour l'analyse des programmes contenant des fonctions booléennes (chapitre 4), un sous-ensemble simplifié des algorithmes **Transformation** et **Réduction** est implanté.

A l'aide de ces règles, le programme est traduit en équations de séries de dénombrement et de descripteurs de complexité, équations qui sont simplifiées par MAPLE (seules des simplifications élémentaires du genre $x + 0 \to x$, $x + x \to 2x$, $xx \to x^2$ sont appliquées) :

```
#printlevel:=2; analyze "diff";;
                                    ...
Introducing the new complexity descriptor tau_copy1 over expo expression
                                    ...
Counting generating functions:
  expression(z)=zero(z)+one(z)+x(z)+plus(z)*expression(z)**2+times(z)*expression
(z)**2+expo(z)*expression(z)
  plus(z)=z
  times(z)=z
  expo(z)=z
  zero(z)=z
  one(z)=z
  x(z)=z

Complexity descriptors:
  tau_diff(z)=plus(z)*expression(z)**2+2*plus(z)*tau_diff(z)*expression(z)+3*tim
es(z)*expression(z)**2+2*times(z)*tau_diff(z)*expression(z)+2*times(z)*expressio
n(z)*tau_copy(z)+2*expo(z)*expression(z)+expo(z)*tau_diff(z)+expo(z)*tau_copy(z)
+zero(z)+one(z)+x(z)
  tau_diffscp(z)=plus(z)*expression(z)**2+2*plus(z)*tau_diffscp(z)*expression(z)
+3*times(z)*expression(z)**2+2*times(z)*tau_diffscp(z)*expression(z)+expo(z)*exp
ression(z)+expo(z)*tau_diffscp(z)+zero(z)+one(z)+x(z)
  tau_copy(z)=plus(z)*expression(z)**2+2*plus(z)*tau_copy(z)*expression(z)+times
(z)*expression(z)**2+2*times(z)*expression(z)*tau_copy(z)+expo(z)*expression(z)+
```

```
expo(z)*tau_copy(z)+zero(z)+one(z)+x(z)
  tau_copy1(z)=expo(z)*expression(z)+expo(z)*tau_copy(z)
```

Il apparaît dans cette analyse que le système a introduit une procédure auxiliaire `copy1`, pour la copie des expressions de type exponentiel. Le descripteur de complexité associé, `tau_copy1`, vient naturellement s'ajouter aux séries génératrices à déterminer.

## C.2 Le module de résolution (SOLVER)

La résolution des systèmes non récursifs se réduit à une suite de substitutions, ce que sait faire tout système de calcul formel digne de ce nom. Dans le cas des systèmes récursifs, il est parfois possible de résoudre, c'est-à-dire de trouver des formes explicites pour les fonctions inconnues, à l'aide de fonctions usuelles comme $+$, $-$, $\times$, $/$, $\sqrt{\ }$. Cette résolution combine des techniques de résolution de systèmes linéaires, d'équations algébriques et différentielles. Ces techniques sont implantées dans le système de calcul formel MAPLE. Cependant, il nous a fallu écrire un module de résolution propre à $\Lambda\Upsilon\Omega$ pour les raisons suivantes :

- dans l'état actuel de MAPLE, il n'existe pas de fonction générale de résolution. Pour résoudre des équations algébriques, il faut utiliser `solve`, et pour des équations différentielles, `dsolve` :

```
> solve(diff(P(z),z) = 1+P(z)^2, P(z));
Error, (in solve) To solve differential equations, please use dsolve
> dsolve(P(z)=1+z*P(z),P(z));
Error, (in dsolve/diffeq) not a differential equation in specified variables
```

- certaines solutions données par la fonction `solve` ne correspondent pas à des séries génératrices (termes en $1/z$ ou coefficients négatifs dans le développement de Taylor à l'origine). Par exemple, l'équation (2.6) page 53 de la série associée au type `expression` admet en théorie deux solutions, dont une seule est valide :

```
> solve(E(z) = 3*z + z*E(z) + 2*z*E(z)^2, E(z));
                                  2 1/2                                 2 1/2
        - 1 + z + (1 - 2 z - 23 z )           - 1 + z - (1 - 2 z - 23 z )
   - 1/4 ----------------------------, - 1/4 ----------------------------
                      z                                     z

> map(series,["],z,4);
              2       3      4        -1                   2       3      4
    [3 z + 3 z  + 21 z  + O(z ), 1/2 z   - 1/2 - 3 z - 3 z  - 21 z  + O(z )]
```

  Il faut détecter et éliminer le plus tôt possible les solutions non valides, afin d'empêcher une explosion du nombre de solutions théoriques, dont on sait de part l'origine combinatoire du problème qu'une seule d'entre elles est valide.

- les systèmes produits par l'analyse algébrique contiennent des opérateurs inconnus de MAPLE ($Q$, $L$, ...). La résolution n'est parfois pas possible sans substituer ces opérateurs :

```
> solve(f(z) = z*Q(f(z)), f(z));
> solve(f(z) = z/(1-f(z)), f(z));
                                  1/2                       1/2
                1/2 + 1/2 (1 - 4 z)   , 1/2 - 1/2 (1 - 4 z)
```

mais en même temps on désire conserver le plus possible ces opérateurs en vue de l'analyse asymptotique, car ils décrivent la structure des séries génératrices. Il est donc nécessaire de disposer d'une fonction de résolution qui ne substitue ces opérateurs qu'en cas de besoin.

- quelquefois, la forme des solutions que donne MAPLE n'est valable que dans un certain domaine, qui est rarement celui qui nous intéresse. Par exemple, lors de la résolution de l'équation différentielle de *QuickSort* (cf page 139),

```
> dsolve( diff(tauQ(z),z) = 2/(1-z)^3 + 2*tauQ(z)/(1-z), tauQ(z) );
                                  ln(- 1 + z)        _C1
                     tauQ(z) = - 2 ------------ + ------------
                                                2              2
                                   1 - 2 z + z    1 - 2 z + z
```

MAPLE suppose implicitement que $z-1$ est positif, ce qui n'est pas vrai au voisinage de $z=0$.

Ainsi, le module que nous avons écrit combine les fonctionnalités de `solve` et de `dsolve` :

```
>    luo({P(z)=1+z*P(z)});    {P   =     proc(z)     1/(1-z)   end}  >
luo({P(z)=z+int(P(u)^2,u=0..z)}); {P = proc(z) sin(z)/cos(z) end}
```

élimine les fonctions non valides combinatoirement :

```
> luo({E(z) = 3*z + z*E(z) + 2*z*E(z)^2});
             {E = proc(z) -1/4*1/z*(-1+z+(1-2*z-23*z^2)^(1/2)) end}
```

connaît les opérateurs $Q$, $L$, ..., et ne les substitue qu'en cas de besoin :

```
> luo({f(z)=z*Q(f(z)),g(z)=Q(f(z))});
{g = proc(z) Q(1/2-1/2*(1-4*z)^(1/2)) end, f = proc(z) 1/2-1/2*(1-4*z)^(1/2) end}
```

enfin choisit la forme des solutions valable dans le domaine contenant l'origine :

```
> luo({tauQ(z)=1/(1-z)^2+2*int(tauQ(u)/(1-u),u=0..z)});
          {tauQ = proc(z) -2*1/(1-2*z+z^2)*ln(1-z)+1/(1-2*z+z^2) end}
```

# Références bibliographiques

*Rappelle-toi les jours d'autrefois,*
*remonte le cours des années, de génération en génération,*
*demande à ton père, et il te l'apprendra,*
*à tes anciens, et ils te le diront.*
Livre du Deutéronome, 32.7


[AU72] A. V. Aho et J. D. Ullman. *The Theory of Parsing, Translation, and Compiling; Volume 1: Parsing.* Prentice-Hall, 1972.

[AU77] A. V. Aho et J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[BC88] F. Bergeron et G. Cartier. Darwin: Computer Algebra and Enumerative Combinatorics. Dans R. Cori et M. Wirsing, éditeurs, *Proceedings of STACS-88*, numéro 294 dans Lecture Notes in Computer Science, pages 393–394, 1988. Existe aussi en rapport technique numéro 54 du Département de Mathématiques et d'Informatique de l'Université du Québec à Montréal.

[BG71] E. A. Bender et J. R. Goldman. Enumerative Uses of Generating Functions. *Indiana University Mathematical Journal*, pages 753–765, 1971.

[Bir89] S. Bird. Transaction ⟨1107@epistemi.ed.ac.uk⟩ de usenet.sci.math, 1989.

[BK75] R. P. Brent et H. T. Kung. $O((n \log n)^{3/2})$ algorithms for composition and reversion of power series. Dans J. F. Traub, éditeur, *Analytic Computational Complexity*, pages 217–225, 1975. Proceedings of the Symposium on Analytic Computational Complexity, Carnegie-Mellon University.

[BK78] R. P. Brent et H. T. Kung. Fast Algorithms for Manipulating Formal Power Series. *Journal of the ACM*, 25(4):581–595, 1978.

[BLL88] F. Bergeron, G. Labelle et P. Leroux. Functional Equations for Data Structures. Dans *Proceedings STACS-88*, numéro 294 dans Lecture Notes in Computer Science, pages 73–80, 1988.

[BLL90] F. Bergeron, G. Labelle et P. Leroux. Combinatoire et structures arborescentes, 1990. Version préliminaire.

[Bol85] B. Bollobás. *Random Graphs.* Academic Press, 1985.

[BR82] J. Berstel et C. Reutenauer. Recognizable formal power series on trees. *Theoretical Computer Science*, 18:115–148, 1982.

[BYCDM89] R. Baeza-Yates, R. Casas, J. Díaz et C. Martínez. On the average size of the intersection of binary trees. Rapport technique RR LSI-89-23, Universitat Politècnica de Catalunya, Barcelona, 1989. A paraître dans SIAM Journal of Computing.

[CGG+88] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan et S.M. Watt. *MAPLE: Reference Manual.* University of Waterloo, 1988. 5th edition.

[DLL82] H. Décoste, G. Labelle et P. Leroux. Une approche combinatoire pour l'itération de Newton-Raphson. *Advances in Applied Mathematics*, 3:407–416, 1982.

[Fla85] P. Flajolet. Elements of a general theory of combinatorial structures. Dans Lothar Budach, éditeur, *Fundamentals of Computation Theory*, volume 199 de *Lecture Notes in Computer Science*, pages 112–127. Springer Verlag, 1985. Comptes-rendus de FCT'85, Cottbus, GDR, septembre 1985.

[FO90] P. Flajolet et A. M. Odlyzko. Singularity analysis of generating functions. *SIAM Journal on Discrete Mathematics*, 3(2):216–240, 1990.

[Foa74] D. Foata. *La série génératrice exponentielle dans les problèmes d'énumération.* S.M.S. Montreal University Press, 1974.

[FS81] P. Flajolet et J-M. Steyaert. A complexity calculus for classes of recursive search programs over tree structures. Dans *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 386–393. IEEE Computer Society Press, 1981.

[FS91] P. Flajolet et M. Soria. The cycle construction. *SIAM Journal on Discrete Mathematics*, 4(1), février 1991. 2 pages. A paraître.

[FSZ89a] P. Flajolet, B. Salvy et P. Zimmermann. Lambda–Upsilon–Omega: The 1989 Cookbook. Rapport de recherche 1073, Institut National de Recherche en Informatique et en Automatique, août 1989. 116 pages.

[FSZ89b] P. Flajolet, B. Salvy et P. Zimmermann. Lambda-Upsilon-Omega : An Assistant Algorithms Analyzer. Dans T. Mora, éditeur, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 de *Lecture Notes in Computer Science*, pages 201–212, 1989. Comptes-rendus de AAECC-6, Rome, juillet 1988.

[FSZ91] P. Flajolet, B. Salvy et P. Zimmermann. Automatic Average–case Analysis of Algorithms. *Theoretical Computer Science*, 79(1):37–109, février 1991.

[FVV78] J. Françon, G. Viennot et J. Vuillemin. Description et analyse d'une représentation performante des files de priorité. Dans *Proceedings of the 19th IEEE Annual Symposium on Foundations of Computer Science*, pages 1–7, 1978.

[Gre83] D. H. Greene. *Labelled formal languages and their uses.* Thèse de PhD, Stanford University, 1983.

[HC88] T. Hickey et J. Cohen. Automating program analysis. *Journal of the ACM*, 35:185–220, 1988.

[HRS75] F. Harary, R. W. Robinson et A. J. Schwenk. Twenty-step algorithm for determining the asymptotic number of trees of various species. *J. Austral. Math. Soc.* (Series A), 20:483–503, 1975.

[HW79] G. H. Hardy et E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, cinquième édition, 1979.

[JL86] J-P. Jouannaud et P. Lescanne. La Réécriture. *Technique et Science Informatiques*, 5(6):433–452, 1986.

[Joy81] A. Joyal. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3 : Sorting and Searching. Addison-Wesley, 1973.

[Knu81] D. E. Knuth. *The Art of Computer Programming*, volume 2 : Seminumerical Algorithms. Addison-Wesley, deuxième édition, 1981.

[Lab86] G. Labelle. Some New Computational Methods in the Theory of Species. Numéro 1234 dans Lecture Notes in Mathematics, pages 192–209, Montréal Québec, 1986. Springer-Verlag.

[LV88a] P. Leroux et G. X. Viennot. A Combinatorial Approach to Non Linear Functional Expansions: An Introduction With an Example. Dans *Proceedings of the 27th IEEE Conference on Decision and Control*, pages 1314–1319, Austin, Texas, décembre 1988. Existe également en rapport technique numéro 79 de l'Université du Québec à Montréal, département de Mathématiques et d'Informatique.

[LV88b] P. Leroux et G. X. Viennot. Combinatorial Resolution of Systems of Differential Equations IV: Separation of Variables. *Discrete Mathematics*, 72:237–250, 1988.

[Met88] D. Le Metayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[MM78] A. Meir et J. W. Moon. On the altitude of nodes in random trees. *Canadian Journal of Mathematics*, 30:997–1015, 1978.

[MW54] J. Macintyre et R. Wilson. Operational methods and the coefficients of certain power series. *Mathematische Annalen*, 127:243–250, 1954.

[Nor90] G. H. Norton. On the Asymptotic Analysis of the Euclidean Algorithm. *Journal of Symbolic Computation*, 10:53–58, 1990.

[Pól37] G. Pólya. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, 68:145–254, 1937.

[PR87] G. Pólya et R. C. Read. *Combinatorial Enumeration of Groups, Graphs and Chemical Componds.* Springer Verlag, New York, 1987.

[Rea61] R. C. Read. A note on the number of functional digraphs. *Mathematische Annalen*, 143:109–110, 1961.

[Sal91] B. Salvy. *Asymptotique automatique et fonctions génératrices.* Thèse de troisième cycle, École Polytechnique, 1991. En préparation.

[Sor89] M. Soria. *Méthodes d'analyse pour les constructions combinatoires et les algorithmes.* Thèse d'état, Université de Paris-Sud, 1989.

[Sta78] R. P. Stanley. Generating Functions. Dans G-C. Rota, éditeur, *Studies in Combinatorics,* M.A.A. Studies in Mathematics, volume 17, pages 100–141. The Mathematical Association of America, 1978.

[Ste84] J-M. Steyaert. *Structure et complexité des algorithmes.* Thèse d'état, Université de Paris VII, 1984.

[VF90] J. Vitter et P. Flajolet. Analysis of algorithms and data structures. Dans J. van Leeuwen, éditeur, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapitre 9, pages 431–524. North Holland, 1990.

[Vui80] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, avril 1980.

[WAL+87] P. Weis, M.V. Aponte, A. Laville, M. Mauny et A. Suárez. *The CAML Reference Manual.* INRIA-ENS, 1987.

[Weg75] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[Wil90] H. S. Wilf. *Generatingfunctionology.* Academic Press, 1990.

[Zim88] P. Zimmermann. Alas : un système d'analyse algébrique. Rapport de DEA, Université de Paris VII, 1988. 120 pages. Disponible aussi en rapport de recherche INRIA (numéro 968).

[Zim90] W. Zimmermann. *Automatische Komplexitätsanalyse von funktionalen Programmen.* Thèse de PhD, Fakultät für Informatik der Universität Karlsruhe, juin 1990. Publié aussi dans la collection *Informatik Fachberichte*, numéro 261, Springer Verlag.

# Index

[**définitions en gras**, *règles en italique*]

Séries génératrices et analyse automatique d'algorithmes

**Résumé** : l'objet de cette thèse est la mise en évidence de procédés *systématiques* pour déterminer *automatiquement* le coût moyen d'un algorithme. Ces procédés s'appliquent en général aux schémas de *descente* dans des structures de données *décomposables*, ce qui permet de modéliser une vaste classe de problèmes.

Cette thèse s'intéresse plus précisément à la première phase de l'analyse d'un algorithme, l'*analyse algébrique*, qui traduit le programme en objets mathématiques, tandis que la seconde phase extrait de ces objets les informations désirées sur le coût moyen. Nous définissons un langage de spécification pour définir des structures de données décomposables et des procédures de descente sur celles-ci. Lorsque l'on utilise comme objets mathématiques des séries génératrices (de dénombrement pour les données, de coût pour les procédures), nous montrons que les algorithmes décrits dans ce langage se traduisent *directement* en systèmes d'équations pour les séries génératrices associées, et de surcroît par des règles *simples*. A partir de ces équations, nous pouvons ensuite déterminer en temps polynomial le coût moyen exact pour une valeur fixée de la taille des données. On pourra aussi utiliser ces équations pour calculer par *analyse asymptotique* le coût moyen lorsque la taille des données tend vers l'infini, puisque l'on sait par ailleurs que le coût moyen asymptotique est directement lié au comportement des séries génératrices au voisinage de leurs singularités. Ainsi nous montrons qu'à une classe donnée d'algorithmes correspond une classe bien définie de séries génératrices, et par conséquent une certaine classe de formules pour le coût moyen asymptotique.

Ces règles d'analyse algébrique ont été incorporées dans un système d'analyse automatique d'algorithmes, Lambda-Upsilon-Omega ($\Lambda\Upsilon\Omega$), qui s'est avéré être un outil très utile pour l'expérimentation et la recherche.

Mots-clés : analyse automatique d'algorithmes, coût moyen, analyse algébrique, série génératrice.