Ordered Generation of Classes of Combinatorial Structures

Tesi doctoral presentada al
Programa de Doctorat de Matemàtica Aplicada
de la Universitat Politècnica de Catalunya

per optar al grau de
Doctor en Matemàtiques

per
Xavier Molinero Albareda

sota la direcció del Doctor
Conrado Martínez Parra

i la tutoria del Doctor
Oriol Serra Albó

Barcelona, 25 de juliol de 2005

This thesis was presented at the Departament de Llenguatges i Sistemes Informàtics of the Universitat Politècnica de Catalunya on October 10th, 2005.

The disseration committee consisted of:

- Marc Noy (Chair) - Departament de Matemàtica Aplicada II, Universitat Politècnica de Catalunya.
- Salvador Roura (Secretary) - Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.
- Philippe Flajolet - INRIA Rocquencourt.
- Simone Rinaldi - Dipartimento di Scienze Matematiche e Informatique, Università degli Studi di Siena.
- Nicolas M. Thiéry - Laboratoire de Mathématiques, Institut Universitaire de Technologie d'Orsay.

This work is dedicated to
Trini, Jesús, Maria, Ester, David and, my love, Lourdes.

Thanks for being here.

Resum

Existeixen quatre problemes molt relacionats sobre la generació d'objectes combinatoris: *drawing* (donada una classe combinatòria i una talla, generar un objecte a l'atzar), *ranking* (calcular el *rang* d'un objecte donat a partir d'una classe combinatòria, segons un ordre prèviament establert), *unranking* (donada una classe combinatòria, generar un objecte de rang i talla donats, segons un ordre prèviament establert) i *iteratió* o *generació exhaustiva* (generar tots els objectes d'una classe combinatòria i talla donades, segons un ordre prèviament establert). En aquesta tesi doctoral es combinen alguns principis coneguts amb noves idees en un àmbit genèric per solucionar eficientment els tres darrers problemes. Els algorismes són genèrics en el sentit que les entrades consideren una especificació finita de la classe combinatòria a la qual pertanyen els objectes combinatoris.

Es donen algorismes d'*unranking* i *ranking* d'objectes de talla n amb cost mitjà $\mathrm{O}(n\sqrt{n})$ per l'ordre lexicogràfic i $\mathrm{O}(n \log n)$ per l'ordre bustrofèdic. En el cas de las classes iteratives (classes on no hi ha recursió), el cost és $\Theta(n)$. També s'estudien diferents heurístiques per millorar l'execució dels algorismes d'*unranking*, i la distribució de probabilitat del cost de l'*unranking* de les classes etiquetades amb l'ordre lexicogràfic.

Per al problema de la iteració, s'han dissenyat algorismes amb cost amortitzat constant per una àmplia família de classes admissibles, incluent les més rellevants (classes admissibles etiquetades superexponencials, i classes admissibles no etiquetades superpolinòmiques). També es presenten diferents tècniques (com per exemple, la utilització dels *punters de cerca*) per millorar l'eficiència dels algorismes.

La flexibilitat dels algorismes presentats en aquesta tesi permeten adaptar-los fàcilment a les biblioteques de software combinatori, com per exemple el paquet combstruct per MAPLE o el paquet MuPAD-combinat per MuPAD.

Abstract

There are four distinct but closely related problems about the generation of combinatorial objects: *drawing* (generate a random combinatorial object of a given combinatorial class and size), *ranking* (given a combinatorial class and an object from that class, compute the *rank* of the given object, that is, the number of objects smaller than the given object, according to some previously fixed order), *unranking* (generate a combinatorial object whose rank and size are given, according to a fixed specific order) and *iteration* or *exhaustive generation* (generate all objects of a given combinatorial class and size, according to some fixed order). In this thesis we combine some well-known principles and novel ideas in a generic framework to design efficient solutions to the last three problems. Our algorithms are generic in the sense that their input includes a finite specification of the combinatorial class to which the objects belong.

We provide algorithms for unranking and ranking objects of size n with average cost $\mathrm{O}(n\sqrt{n})$ for the lexicographic order and $\mathrm{O}(n\log n)$ for the boustrophedonic order. In the case of iterative classes (classes where recursion is not used to specify them), the cost is $\Theta(n)$. Furthermore, we have also studied several heuristics to improve the performance of the unranking algorithms, and the probability distribution of the cost of unranking labelled classes with lexicographic ordering.

For the iteration problem, we have designed algorithms with constant amortized cost for a large family of admissible classes, including the most interesting classes (what we call *superexponential* labelled admissible classes and *superpolynomial* unlabelled admissible classes). We have also introduced several techniques (e.g., the use of *finger pointers*) to improve the performance of these algorithms.

The flexibility of our generic algorithms makes them attractive for rapid prototyping and for their inclusion in general combinatorial libraries, like the combstruct package for MAPLE or the MuPAD-combinat package for MuPAD.

Acknowledgements

Many people have helped me while conducting this research. In first place, I want to mention Conrado Martínez whose advice, unlimited patience, guide and incessant encouragement made this thesis possible.

I would like to thank, particularly, Josep Díaz, Rafel Cases, Philippe Flajolet and Salvador Roura, with whom I started this project.

During the development of this thesis, I had the privilege to have fruitful discussions with Cyril Banderier, Alberto Del Lungo, Simone Rinaldi, Bruno Salvy, Joe Sawada and Nicolas Thiery, among others. I would like to thank them for being so patient and kind, they have drawn my attention to articles, have sent me their programs and have helped me testing them.

I am thankful for the support that I have received from the Doctoral Program in Applied Math, specially to Professor Oriol Serra, who has been my tutor, and to Carme Capdevila, for her invaluable help to deal with all the bureaucratic details. I am also deeply grateful for the support provided by the Departament de Llenguatges i Sistemes Informàtics of the Universitat Politècnica de Catalunya. I owe particular thanks to Maria J. Blesa, Sandra Moral and Jordi Petit for their companionship and moral support. Sharing the office with them benefited me with an excellent human and scientific environment. Facing the risk of forgetting some people, I also want to express special thanks to Albert Abelló, José Luis Balcázar, Christian Blum, Amalia Duch, Xavier Franch, Jordi Marco, Xavier Pérez, Montse Rubia, Maria José Serna, Pere Pau Vázquez, Sebastià Vila and Fatos Xhafa.

I would also like to thank all my friends and professors of my undergraduate years, Tevi and Maria, Paquita and Miguel, and Mireia and Francesc; Josep Grané, Joan de Solà-Morales, Marc Noy and Pere Pasqual.

I also thank all my friends out of the academic world, but they are too numerous to list all of them here. However, Josep Camprubí, Manel Olmo, Albert Parera, Miquel Parera and my friends of *frontó*, Jordi Morera, Jordi Munyarch, Lluis Pol and Xavier Seuba, deserve a particular mention.

Finally, I dedicate this work to Trini, Jesús, Maria, Ester, David and, my love, Lourdes. I will always be infinitely grateful for their love, support, patience and for being so nice.

Support

This research was partially supported by the ALCOM-IT Project (Algorithms and Complexity in Information Technology, European Union ESPRIT LTR Project 20244), the ALCOM-FT Project (Algorithms and Complexity - Future Technologies, European Union FET IST-1999-14186), the AEDRI Project (Algorithms and Data Structures for Information Retrieval, the Spanish Ministerio de Educación y Ciencia, DGES PB98-0926) and the AEDRI-II Project (Algorithms and Data Structures for Information Retrieval II, the Spanish Ministerio de Ciencia y Tecnología, TIC2002-00190).

Contents

Introduction **1**

I Preliminaries **5**

1 Mathematical Preliminaries **7**
1.1 Admissible Classes and Generating Functions 7
1.2 Bivariate Generating Functions 13
1.3 Analysis Techniques 15

2 Isomorphisms and Orders **21**
2.1 Basic Operators 22
2.2 Labelled Containers 27
2.3 Unlabelled Containers 30
2.4 Other Containers 33

II Unranking and Ranking **37**

3 Labelled Unranking **41**
3.1 The Algorithms 42
3.2 Analysis of the Cost 45
3.3 Examples 57
3.4 Variants and Heuristics 62

4 Unlabelled Unranking **65**
4.1 The Algorithms 66
4.2 Analysis of the Cost 72
4.3 Examples 78

5 Experiments on Unranking **81**
5.1 Experiments on the Performance 83
5.2 Unranking v.s. Random Generation 90
5.3 Conclusions and Final Remarks 91

6 Ranking **93**
6.1 Algorithms for Labelled Classes . . . 94
6.2 Algorithms for Unlabelled Classes . . . 96
6.3 Analysis of the Cost . . . 99
6.4 Membership . . . 99
6.5 Conclusions and Final Remarks . . . 101

III Exhaustive Generation **103**

7 Labelled Generation **107**
7.1 Iterators . . . 108
7.2 Implementation and Algorithms . . . 111
7.3 Variants and Heuristics . . . 120
7.4 Analysis of the Cost . . . 122

8 Unlabelled Generation **131**
8.1 Implementation and Algorithms . . . 131
8.2 Analysis of the Cost . . . 137
8.3 Variants and Heuristics . . . 142
8.4 Generating Cycles . . . 143

IV Conversions of Representations **149**

9 Conversions of Representations **151**
9.1 Representations . . . 152
9.2 Conversions LNF $\leftrightarrow$ TNF . . . 154
9.3 Conversions LNF $\leftrightarrow$ INF . . . 158

V Conclusions and Future Work **163**

A Unranking Experimental Data **167**
A.1 Tables . . . 167
A.2 Plots . . . 174

List of Figures

1.1 Substitution. 11

2.1 Lexicographic order induced by $\mathsf{Seq}(Z)$. 21
2.2 Lexicographic order induced by $\mathsf{Seq}(\mathsf{Cycle}(Z))$. 21
2.3 Unlabelled binary trees of size 4. 23
2.4 Labelled binary trees of size 3 in PF order. 24
2.5 Labelled binary trees of size 3 in SF order. 25
2.6 $\vartheta((Z,Z),Z)$. 26
2.7 Bijection between $\vartheta\mathsf{Set}(\mathcal{A})$ and $\Delta\vartheta\mathcal{A} \times \mathsf{Set}(\mathcal{A})$. 26
2.8 Lexicographic-PF trees of size 4. 28
2.9 Lexicographic-SF trees of size 4. 29

5.1 CPU time $\overline{\tau}_{n,\mathcal{B}}$ for binary trees (seconds). 87

7.1 Cayley trees of size 3 in PF order. 108
7.2 Cayley trees of size 3 in SF order. 108
7.3 Iterator for the first Cayley tree of size 3. 112
7.4 Iterator for $(Z_4 \star Z_1) \star (Z_5 \star (Z_3 \star Z_2))$. 118

8.1 Internal representation of a PSet-object. 134
8.2 Internal representation of a Set-object. 134
8.3 Internal representation of a Cycle-object. 144

9.1 Common framework for the ordered generation. 151

A.1 Experimental average cost $\overline{\mu}_{n,\mathcal{B}}$ for binary trees. 174
A.2 CPU time $\overline{\tau}_{n,\mathcal{B}}$ for binary trees (secs.). 174
A.3 Experimental average cost $\overline{\mu}_{n,\mathcal{B}}$ for binary trees. 175
A.4 Experimental average cost $\overline{\mu}_{n,\mathcal{M}}$ for Motzkin trees. 175
A.5 CPU time $\overline{\tau}_{n,\mathcal{M}}$ for Motzkin trees (secs.). 175
A.6 CPU time $\overline{\tau}_{n,\mathcal{F}}$ for functional graphs (secs.). 175
A.7 CPU time $\overline{\tau}_{n,\mathcal{S}}$ for surjections and integer compositions (secs.). 176

List of Tables

1.1 Combinatorial operators and generating functions. 9
1.2 Examples of labelled classes and their specifications. 11
1.3 Examples of unlabelled classes and their specifications. 12

2.1 Isomorphisms for $\mathsf{PSet}(\mathcal{A})$. 31
2.2 Order criteria on the leading block. 31
2.3 Different isomorphisms give different orders. 32
2.4 Isomorphisms for labelled restricted containers. 34
2.5 Isomorphisms for unlabelled restricted containers. 34

5.1 Labelled and unlabelled classes used in the unranking experiments. 84
5.2 Notation for experimental results. 85
5.3 CPU time to compute the counting of some classes. 86
5.4 Lexicographic labelled binary trees. 87
5.5 Unlabelled powersets and sets of binary trees. 90
5.6 Ratios of CPU times for unranking v.s. random generation. 91

7.1 Some attributes generated for $(Z_2, Z_1) \star (Z_3, (Z_2, Z_1))$. 118

A.1 Lexicographic labelled binary trees. 167
A.2 Boxed heuristic of lexicographic binary trees. 168
A.3 Lexicographic unlabelled binary trees. 168
A.4 Boustrophedonic labelled binary trees. 168
A.5 Boxed heuristic of boustrophedonic binary trees. 169
A.6 Boustrophedonic unlabelled binary trees. 169
A.7 Lexicographic labelled Motzkin trees. 169
A.8 Lexicographic unlabelled Motzkin trees. 170
A.9 Boustrophedonic labelled Motzkin trees. 170
A.10 Boustrophedonic unlabelled Motzkin trees. 170
A.11 Lexicographic (labelled) functional graphs. 171
A.12 Boustrophedonic (labelled) functional graphs. 171
A.13 Lexicographic labelled surjections. 171
A.14 Lexicographic unlabelled integer compositions. 172
A.15 Boustrophedonic labelled surjections. 172
A.16 Boustrophedonic unlabelled integer compositions. 172
A.17 Lexicographic (unlabelled) integer partitions. 173

A.18 Lexicographic (unlabelled) integer partitions. 173
A.19 Boustrophedonic (unlabelled) integer partitions. 173
A.20 Boustrophedonic (unlabelled) integer partitions. 174

Introduction

In this thesis we design and analyze efficient algorithms for the ordered generation of combinatorial structures. The generation of combinatorial structures is an important problem with numerous applications that has attracted the interest of combinatorialists and computer scientists alike for many years. There is a vast literature on the topic and many ingenious techniques and efficient algorithms have been devised for the generation of objects of relevant combinatorial classes. For instance, binary trees [51, 69, 71, 72, 73, 74], Cayley trees [49], Dyck paths [50], permutations [77, 84], strings [12, 37] and many other combinatorial structures [13, 29, 39, 46, 47, 67, 70, 85, 86]. Furthermore, several distinct natural and useful orderings have been considered for the generation of combinatorial classes, for example, lexicographic, boustrophedonic and Gray ordering. Many state-of-the-art algorithms can be found and executed in the Combinatorial Object Server [10], where the interested reader can also find further references.

There are four distinct but closely related problems about the generation of combinatorial objects:

1. *Drawing*: Given a combinatorial class and a size, generate a random object of the given class and size.

2. *Ranking*: Given a combinatorial class and an object from that class, compute the *rank* of the given object, that is, the number of objects of the class of that size smaller than the given object, according to some previously fixed order.

3. *Unranking*: Given a combinatorial class and a size, generate the object whose rank is given, according to a fixed specific order.

4. *Iteration* or *Exhaustive Generation*: Given a combinatorial class and a size, construct an *iterator*, that is, a 3-tuple of functions $\langle \mathsf{first}, \mathsf{is\_last}, \mathsf{next} \rangle$ that allows for the sequential traversal of all the objects of the given class with the given size, according to some fixed order.

In this thesis we combine some well-known principles and novel ideas in a generic framework to design efficient solutions to the last three problems. Our algorithms are generic in the sense that their input includes a finite specification of the combinatorial class to which the objects belong. The approach that we follow in this thesis was pioneered by Flajolet *et al.* [26] for the random generation of combinatorial objects (the *drawing* problem). We provide algorithms for unranking and ranking objects of size n with average cost $\mathrm{O}(n\sqrt{n})$ for the lexicographic order and $\mathrm{O}(n \log n)$ for the boustrophedonic order. In the case of iterative classes (classes where recursion is not used to specify them), the cost is $\Theta(n)$. The same complexities appear for the random generation of combinatorial objects in [26]. Furthermore, we have also studied several heuristics to improve the performance of the unranking algorithms, and the probability distribution of the cost of unranking labelled classes with lexicographic ordering.

For the iteration problem, we have designed algorithms with constant amortized cost (CAT algorithms) for a large family of admissible classes, including the most interesting classes (we call them *superexponential* labelled admissible classes, and *superpolynomial* unlabelled admissible classes). We have also introduced several techniques (e.g., the use of *finger pointers*) to improve the performance.

Since the algorithms described here are generic they do not take advantage of particular features of the combinatorial class whose objects we want to generate. However, the algorithms are competitive when compared with algorithms specifically designed for the corresponding class. Most likely, the price of being general is that the involved constant factors are larger than in the *ad-hoc* counterparts, but the flexibility of our generic algorithms makes them attractive for rapid prototyping and for their inclusion in general combinatorial libraries, like the combstruct package for MAPLE [7, 19, 20, 87, 89] or the MuPAD-combinat package for MuPAD [8, 27, 28, 38, 53, 65].

This thesis is organized in five parts. Part I contains basic definitions and a brief description of the analysis techniques that will be used in the rest of the thesis. In Chapter 1 we introduce labelled and unlabelled admissible classes, counting and bivariate generating functions, and we summarize some analysis techniques to find asymptotic estimates of the coefficients of generating functions. In Chapter 2 we present several fundamental isomorphisms and orders (lexicographic, boustrophedonic, etc.) for the admissible classes of combinatorial structures that will be considered in this thesis.

Our main contributions are presented in Parts II and III. In Part II we describe the unranking and ranking algorithms and their analysis. We also present a throughly experimental study on the practical performance of these algorithms. In this part we prove that the unranking and ranking of iterative classes can be done in linear time. For non-iterative classes, the average cost of unranking and ranking is $O(n\sqrt{n})$ or $O(n \log n)$ depending whether the lexicographic or the boustrophedonic order is used, respectively. Moreover, we describe the *big-endian heuristic* and the *boxed heuristic* to improve the performance of unranking, and we also give the probability distribution of the cost of unranking labelled classes with lexicographic ordering.

Chapter 3 is dedicated to the unranking of labelled classes. This chapter is based upon *Unranking of Labelled Combinatorial Structures* [54] and *A Generic Approach for the Unranking of Labeled Combinatorial Classes* [56].

Chapter 4 covers the unranking of unlabelled classes. Some of the results presented in this chapter have also appeared in *A Generic Approach for the Unranking of Labeled Combinatorial Classes* [56].

In Chapter 5 we describe in more detail the experimental study of the performance of the unranking algorithms that we have conducted. The experimental data strongly agree with the theoretical results presented in the previous chapters. Moreover, we also compare these algorithms with the random generation implemented in the combstruct package for MAPLE. The extended abstract *An Experimental Study of Unranking Algorithms* [59] is the basis for this chapter.

In the last chapter of this part we study the ranking of labelled and unlabelled admissible classes: We present algorithms and analyze their performance.

In Part III we consider the iteration problem. In this part we design algorithms which work in constant amortized time for most of labelled and unlabelled admissible classes, that is, the cost of generating all N objects of size n in the given class is $\mathrm{O}(N)$. This holds whenever the combinatorial class is superexponential for the labelled case or superpolynomial for the unlabelled case. The most common combinatorial classes fall inside this family. We have also considered several techniques to improve the performance for the iteration problem.

Chapters 7 and 8 are devoted to the exhaustive generation of labelled and unlabelled classes, respectively. An important contribution in this part is the algorithm for the exhaustive generation of unlabelled cycles in constant amortized time. The main results of Chapter 7 have appeared in *Efficient Iteration in Admissible Combinatorial Classes* [60] and *Generic Algorithms for the Exhaustive Generation of Labelled Objects* [55]. Additional contributions have also been published in *Iterating Through Labelled Combinatorial Structures* [64] and *Generic Algorithms for the Generation of Combinatorial Objects* [57]. On the other hand, some of the results presented in Chapter 8 have been published in *Efficient Iteration in Admissible Combinatorial Classes* [60] and *An Efficient Generic Algorithm for the Generation of Unlabelled Cycles* [58].

In Part IV we discuss representational issues and how they play an important rôle not only from the point of view of the efficiency, but also to make it possible a seamless integration of the various tools (counting, drawing, unranking, ...) in a combinatorial software package.

Finally, Part V contains our conclusions, a discussion of open problems and our plans for future research.

Part I

Preliminaries

Chapter 1

Mathematical Preliminaries

In this chapter we recall the formal definition of admissible classes of combinatorial structures and their corresponding counting generating functions. We also review some techniques to analyze the cost of the algorithms presented in this thesis.

1.1 Admissible Classes and Generating Functions

Most of the material in this section is standard and can be found elsewhere, see for instance [22, 25, 79]. However, to make this thesis more self-contained and to fix notation, we will briefly introduce some basic definitions and concepts. We begin with the formal definition of a combinatorial class.

Definition 1. *A* combinatorial class *is a pair* $(\mathcal{A}, |\cdot|_{\mathcal{A}})$ *such that* $\mathcal{A}$ *is a finite or infinite denumerable set and* $|\cdot|_{\mathcal{A}} : \mathcal{A} \to \mathbb{N}$ *is a* size *function such that, for all* $n \geq 0$*,* $\mathcal{A}_n = \{\alpha \in \mathcal{A} \mid |\alpha|_{\mathcal{A}} = n\}$ *is finite.*

We use upper case script letters ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ...) to denote combinatorial classes. Also, we use subscripts under a class' name to denote the subset of objects of that class $\mathcal{A}$ with a given size n, for example, $\mathcal{A}_n$. In a similar way, $\mathcal{A}_{>n}$, $\mathcal{A}_{<n}$, $\mathcal{A}_{\geq n}$ and $\mathcal{A}_{\leq n}$ denote the subset of objects of that class $\mathcal{A}$ with size *larger*, *smaller*, *larger or equal* and *smaller or equal* than n. The convention of using the corresponding Roman upper case letters for counting generating functions (see below) will also be used throughout this thesis. Moreover, if the class is implied, we will drop the subscript in $|\cdot|_{\mathcal{A}}$. Shall no confusion arise, we will use the same name for the class and for the set of objects belonging to that class.

Typically, complex objects in a given class are composed of smaller units, called *atoms* and generically denoted by Z. Atoms are objects of size 1 and the size of an object is the number of atoms it contains. For instance, a string is composed by the concatenation of symbols, where each of these is an atom, and the size of the string is its length or the number of symbols it is composed of. Similarly, a tree is built out of nodes —its atoms— and the size of the tree is its number of nodes. Objects of size 0 are normally denoted by ϵ[1].

[1]Some authors use λ to denote an object of size 0, called the empty object. The same symbol (ϵ,λ) is often used to denote the class which does only contain the empty object. Likewise, Z often denotes an atomic class, shall no confusion arise.

Two main types of combinatorial classes can be defined depending on whether the atoms that compose a given object can be distinguished or not. In the former case, we say that the class is *labelled* whereas in the latter we say the class is *unlabelled*. Examples of labelled classes include permutations, Cayley trees, functional graphs and a host of other important combinatorial classes. A valid standard labelling of an object of size n is a bijection from the object's atoms to $[1..n]$, or equivalently, a permutation of size n.

As it will become apparent, an efficient solution to the problem of counting, namely, given a specification of a class and a size, compute the number of objects with the given size, is fundamental for our approach to the iteration, unranking and ranking problems. Hence, we turn our attention to so-called *admissible combinatorial classes*. Those are constructed from *admissible operators* over classes that yield new classes, and such that the number of objects of a given size in the new class can be computed from the number of objects of that size or smaller sizes in the constituent classes.

We can formalize the notion of admissibility through the notion of *counting generating functions*.

Definition 2. *The (counting) generating function of an unlabelled combinatorial class $\mathcal{A}$ is the ordinary generating function for the sequence $\{a_n\}_{n\geq 0}$, where $a_n = \#\mathcal{A}_n$ is the number of objects in $\mathcal{A}$ of size n. The n-th coefficient of $A(z)$ is $a_n = [z^n]A(z)$. That is,*

$$A(z) = \sum_{n\geq 0} a_n\, z^n = \sum_{\alpha\in\mathcal{A}} z^{|\alpha|} \ .$$

Definition 3. *The (counting) generating function of a labelled combinatorial class $\mathcal{A}$ is the exponential generating function for the sequence $\{a_n\}_{n\geq 0}$, where $a_n = \#\mathcal{A}_n$ is the number of objects in $\mathcal{A}$ of size n. The n-th coefficient of $A(z)$ is denoted $[z^n]A(z)$; hence, $a_n = n!\cdot[z^n]A(z)$. That is,*

$$A(z) = \sum_{n\geq 0} a_n \frac{z^n}{n!} = \sum_{\alpha\in\mathcal{A}} \frac{z^{|\alpha|}}{|\alpha|!} \ .$$

We now define *admissible operators*.

Definition 4. *An* operator *(also called* constructor*) Ψ over combinatorial classes $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ is* admissible *if and only if there exists some operator Φ over the corresponding counting generating functions $A_1(z), \ldots, A_k(z)$ such that*

$$\mathcal{C} = \Psi(\mathcal{A}_1, \ldots, \mathcal{A}_k) \implies C(z) = \Phi(A_1(z), \ldots, A_k(z)) \ ,$$

where $C(z)$ is the counting generating function of $\mathcal{C}$.

Examples of admissible labelled operators include *disjoint unions* (denoted by '+' or Union), *partitional* (or labelled) *products* ('$\star$' or Prod), *sequences* (Seq), *sets* (Set), *cycles* (Cycle), *substitutions* ('$\circ$' or Subst) and *sequences*, *sets* and *cycles with restricted cardinality*. Analogous operators are admissible in the unlabelled case: *disjoint unions* ('+' or Union), *Cartesian* (or unlabelled) *products* ('$\times$' or Prod), *sequences* (Seq), *powersets* (PSet), *sets* (Set)[2], *cycles* (Cycle), *substitutions* ('$\circ$' or Subst) and *sequences*, *powersets*, *sets* and

Class	EGF	OGF
$\mathsf{Union}(\mathcal{A},\mathcal{B}) = \mathcal{A}+\mathcal{B}$	$A(z)+B(z)$	$A(z)+B(z)$
$\mathsf{Prod}(\mathcal{A},\mathcal{B}) = \mathcal{A}\star\mathcal{B}$	$A(z)\ B(z)$	—
$\mathsf{Prod}(\mathcal{A},\mathcal{B}) = \mathcal{A}\times\mathcal{B}$	—	$A(z)\ B(z)$
$\mathsf{Seq}(\mathcal{A})$	$\frac{1}{1-A(z)}$	$\frac{1}{1-A(z)}$
$\mathsf{PSet}(\mathcal{A})$	—	$\exp\left(\sum_{n>0}(-1)^{n-1}\frac{A(z^n)}{n}\right)$
$\mathsf{Set}(\mathcal{A})$	$\exp(A(z))$	$\exp\left(\sum_{n>0}\frac{A(z^n)}{n}\right)$
$\mathsf{Cycle}(\mathcal{A})$	$\log\left(\frac{1}{1-A(z)}\right)$	$\sum_{n>0}\frac{\phi(n)}{n}\log\left(\frac{1}{1-A(z^n)}\right)$
$\mathsf{Subst}(\mathcal{A},\mathcal{B})^3 = \mathcal{A}\circ\mathcal{B}$	$(A\circ B)(z)$	$(A\circ B)(z)$

Table 1.1: Combinatorial operators and generating functions ('EGF' and 'OGF' denote exponential and ordinary generating functions, respectively).

cycles with restricted cardinality. In Table 1.1 we summarize the relations between these constructions and the corresponding generating functions (see also [26]).

We briefly describe now the combinatorial operators mentioned above.

- We take the *disjoint union* (also called *sum*) of two classes $\mathcal{A}$ and $\mathcal{B}$ to represent the union of two disjoint copies, $\mathcal{A}^o$ and $\mathcal{B}^o$, of $\mathcal{A}$ and $\mathcal{B}$. One way to formalize this notion is to introduce two distinct "markers" $\epsilon_{\mathcal{A}}$ and $\epsilon_{\mathcal{B}}$, each of size zero, and define the (disjoint) union $\mathcal{A}+\mathcal{B}$ by

$$\mathcal{A}+\mathcal{B} = (\mathcal{A}\times\epsilon_{\mathcal{A}})\cup(\mathcal{B}\times\epsilon_{\mathcal{B}})\ .$$

  Disjoint union is thus equivalent to a standard union whenever it is applied to disjoint sets. The size of $\gamma\in\mathcal{A}+\mathcal{B}$, i.e., $|\gamma|_{\mathcal{A}+\mathcal{B}}$, is $|\gamma|_{\mathcal{A}}$ if $\gamma\in\mathcal{A}$ or $|\gamma|_{\mathcal{B}}$ if $\gamma\in\mathcal{B}$.

- The *Cartesian product* of $\mathcal{A}$ and $\mathcal{B}$ is composed by the pairs (α,β) such that $\alpha\in\mathcal{A}$ and $\beta\in\mathcal{B}$,

$$\mathcal{A}\times\mathcal{B} = \bigcup_{\alpha\in\mathcal{A}}\bigcup_{\beta\in\mathcal{B}}(\alpha,\beta)\ ,$$

  with $|(\alpha,\beta)|_{\mathcal{A}\times\mathcal{B}} = |\alpha|_{\mathcal{A}}+|\beta|_{\mathcal{B}}$.

- If $\mathcal{A}$ is a class then the *sequence*[4] class is defined as the infinite sum

$$\mathsf{Seq}(\mathcal{A}) = \epsilon+\mathcal{A}+(\mathcal{A}\times\mathcal{A})+(\mathcal{A}\times\mathcal{A}\times\mathcal{A})+\cdots$$

  To guarantee the number of sequences of each size will be finite it is necessary to impose that $\mathcal{A}$ contains no object of size 0 (i.e., $a_0=0$). From the definition of size for sums and products, the size of a sequence is the sum of the sizes of its components: if $\gamma=(\alpha_1,\ldots,\alpha_k)\in\mathsf{Seq}(\mathcal{A})$ then $|\gamma| = |\alpha_1|+\cdots+|\alpha_k|$.

[2] Also called multisets or bags, for they allow repetitions.

[3] The substitution $\mathcal{A}\circ\mathcal{B}$ and the composition $(A\circ B)(z)$ are also denoted by $\mathcal{A}[\mathcal{B}]$ and $A(B(z))$, respectively [15, 30].

[4] It is analogous to the *Kleene star operator* $\mathcal{A}^*$.

- The *powerset* class of $\mathcal{A}$ is defined as the class consisting of all finite subsets of $\mathcal{A}$-objects without repeated components. The following isomorphism is often used:

$$\mathsf{PSet}(\mathcal{A}) = \prod_{\alpha \in \mathcal{A}} (\epsilon + \alpha) \ .$$

Powersets do not require that $a_0 = 0$, but we assume it to be the use.

- *Sets* are like powersets except that the repetitions of components are allowed. Hence

$$\mathsf{Set}(\mathcal{A}) = \prod_{\alpha \in \mathcal{A}} \mathsf{Seq}(\alpha) \ .$$

As in the case of sequences, $\mathcal{A}$ must not contain objects of size 0.

- *Cycles* are just sequences defined up to cyclic permutations: $\mathsf{Cycle}(\mathcal{A}) = \mathsf{Seq}(\mathcal{A})/\!\sim$, with $\sim$ the equivalence relation between sequences defined by $(\alpha_1, \ldots, \alpha_r) \sim (\beta_1, \ldots, \beta_r)$ if and only if there exists some cyclic permutation σ of $[1..n]$ such that for all j, $\beta_{\sigma(j)} = \alpha_j$; in other words, for some d, $\beta_j = \alpha_{1+(j+d) \bmod n}$.

- The *substitution* of $\mathcal{B}$ into $\mathcal{A}$ (or *composition* of $\mathcal{A}$ and $\mathcal{B}$) is defined by

$$\mathcal{A} \circ \mathcal{B} = \bigcup_{k \geq 0} \mathcal{A}_k \times \mathsf{Seq}(\mathcal{B}, \mathrm{card} = k) \ , \tag{1.1}$$

where $\mathsf{Seq}(\mathcal{B}, \mathrm{card} = k)$ is the class of k-tuples of $\mathcal{B}$ objects. Objects of $\mathcal{A} \circ \mathcal{B}$ may thus be viewed as obtained by selecting in all possible ways an α object of $\mathcal{A}$ and substituting for each of its atoms by an arbitrary object β of $\mathcal{B}$. Formally,

$$\mathcal{C} = \{(\alpha, \beta_1, \ldots, \beta_{|\alpha|}) \mid \alpha \in \mathcal{A} \text{ and } \beta_1, \ldots, \beta_{|\alpha|} \in \mathcal{B}\} \ ,$$

and $|(\alpha, \beta_1, \ldots, \beta_{|\alpha|})| = |\beta_1| + \cdots + |\beta_{|\alpha|}|$.

$\mathcal{A} \circ \mathcal{B}$ is well defined if and only if either $b_0 = 0$ or $|\mathcal{A}| < +\infty$; however, in general, we assume that $a_0 = b_0 = 0$.

Labelled operators like unions, sequences, sets and cycles are defined as their unlabelled counterparts. But products and substitutions have some differences:

- Given two labelled objects α and β of sizes j and $n-j$, respectively, their partitional product is a set of $\binom{n}{j}$ labelled objects of size n which result from the $\binom{n}{j}$ consistent relabellings of the pair (α, β) so that each atom of the pair has a distinct label in the range $[1..n]$ while respecting the order of the original labels of α and β. For instance, if $\alpha = 132$ and $\beta = 21$ (these two objects belong to the labelled class $\mathsf{Seq}(Z)$, i.e., permutations) then[5] $\alpha \star \beta = \{13254, 14253, 14352, \ldots, 35421\}$. The labelled product of the labelled classes $\mathcal{A}$ and $\mathcal{B}$ is defined then

$$\mathcal{A} \star \mathcal{B} = \bigcup_{\alpha \in \mathcal{A}} \bigcup_{\beta \in \mathcal{B}} \alpha \star \beta \ ,$$

with $|\alpha \star \beta|_{\mathcal{A} \star \mathcal{B}} = |\alpha|_{\mathcal{A}} + |\beta|_{\mathcal{B}}$.

[5] We are making a slight abuse of notation here: we have refrained from writing $\alpha = (Z_1, (Z_3, (Z_2, \epsilon)))$, $\beta = (Z_2, (Z_1, \epsilon))$, etc, in favor of the usual and more readable form $\alpha = 132$ and $\beta = 21$.

- Labelled substitution is defined by

$$\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathsf{Set}(\mathcal{B}, \mathrm{card} = k) \ . \tag{1.2}$$

The component with the smallest label in the object of $\mathsf{Set}(\mathcal{B}, \mathrm{card} = k)$ (the *leader*) substitutes the atom with the smallest label of the $\mathcal{A}$ object, and the same procedure is repeated with the remaining components of the object of $\mathsf{Set}(\mathcal{B}, \mathrm{card} = k)$ and the remaining atoms of the $\mathcal{A}$ object until all atoms have been replaced by components of the object in $\mathsf{Set}(\mathcal{B}, \mathrm{card} = k)$. Figure 1.1 shows an example.

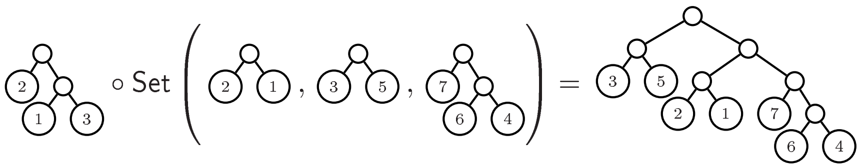


Figure 1.1: Substitution.

Besides these operators, other operators will also be considered: *boxed products* ($^{\square}\star$), *pointings* (ϑ), *diagonals* (Δ), etc.

Though the collection of operators given above is small, it can be used to describe many important and useful combinatorial classes (see Tables 1.2 and 1.3). We provide sample

Labelled class	**Specification**
non-plane (Cayley) trees	$\mathcal{T} = Z \star \mathsf{Set}(\mathcal{T})$
binary plane trees	$\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$
general plane trees	$\mathcal{G} = Z \star \mathsf{Seq}(\mathcal{G})$
set partitions	$\mathcal{P} = \mathsf{Set}(\mathsf{Set}(Z, \mathrm{card} \geq 1))$
non-plane binary trees	$\mathcal{D} = Z + \mathsf{Set}(\mathcal{D}, \mathrm{card} = 2)$
non-plane ternary trees	$\mathcal{E} = Z + \mathsf{Set}(\mathcal{E}, \mathrm{card} = 3))$
hierarchies	$\mathcal{H} = Z + \mathsf{Set}(\mathcal{H}, \mathrm{card} \geq 2))$
3-balanced hierarchies	$\mathcal{I} = \mathsf{Set}(\mathsf{Set}(\mathsf{Set}(Z, \mathrm{card} \geq 1), \mathrm{card} \geq 1))$
surjections	$\mathcal{S} = \mathsf{Seq}(\mathsf{Set}(Z, \mathrm{card} \geq 1)))$
functional graphs	$\mathcal{F} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{T}))$

Table 1.2: Examples of labelled classes and their specifications.

specifications for a few interesting classes, but it is clear that there exist many alternative ways to specify a given class. For instance, the class of permutations can be described by either $\mathsf{Seq}(Z)$ or $\mathsf{Set}(\mathsf{Cycle}(Z))$, where Z denotes an atomic class, i.e., a class that contains only one object of size 1.

Unfortunately, not all combinatorial operators are admissible, most notably the *intersection*, the *difference*, or the *union*.

In order to formalize the definition of *admissible class* we define first *admissible specifications*.

Unlabelled class	Specification
integer partition	$\mathcal{P} = \mathsf{Set}(\mathsf{Seq}(Z, \text{card} \geq 1))$
binary plane trees	$\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}$
binary sequences	$\mathcal{S} = \mathsf{Seq}(Z + Z)$
necklaces	$\mathcal{N} = \mathsf{Cycle}(\mathsf{Set}(Z, \text{card} \geq 1))$
rooted unlabelled trees	$\mathcal{T} = Z \times \mathsf{Set}(\mathcal{T})$
non-plane binary trees	$\mathcal{D} = Z + \mathsf{Set}(\mathcal{D}, \text{card} = 2)$
non-plane ternary trees	$\mathcal{E} = Z + \mathsf{Set}(\mathcal{E}, \text{card} = 3)$
unlabelled hierarchies	$\mathcal{H} = Z + \mathsf{Set}(\mathcal{H}, \text{card} \geq 2))$
random mapping patterns	$\mathcal{F} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{T}))$
2-3 trees	$\mathcal{K} = Z + \mathcal{K} \circ ((Z \times Z) + (Z \times Z \times Z))$
partitions without repetition	$\mathcal{W} = \mathsf{PSet}(\mathsf{Seq}(Z, \text{card} \geq 1))$

Table 1.3: Examples of unlabelled classes and their specifications.

Definition 5. *An* admissible specification S *is a collection of equations of the form*

$$\mathcal{A}_i = \Psi_i(\mathcal{A}^{(i)}_{j_0}, \ldots, \mathcal{A}^{(i)}_{j_i}) \ ,$$

where no two equations have the same left-hand side, each Ψ_i is an admissible operator, and each $\mathcal{A}^{(i)}_{j_r}$ is either an ϵ class, an atomic class, or there is an equation in the collection with that class as its left-hand side. Each of the classes that appear in the left-hand sides of the equations in S is said to be specified by S.

Definition 6. *If a class $\mathcal{A}$ is specified by an admissible specification, then the class itself is called admissible. Admissible classes are also called either* decidable *or* well-founded *classes [87].*

In this thesis, by an *admissible class* we mean the class that can be finitely specified using an ϵ class, atomic classes (those which contain a single object of size 1), disjoint unions (Union, $+$), Cartesian products (Prod, $\times$), partitional products (Prod, $\star$), sequences (Seq), powersets (PSet), sets (Set), cycles (Cycle), substitutions (Subst, $\circ$), sequences with restricted cardinality ($\mathsf{Seq}(\ldots, \text{card} \lesseqqgtr k)$), powersets with restricted cardinality ($\mathsf{PSet}(\ldots, \text{card} \lesseqqgtr k)$), sets with restricted cardinality ($\mathsf{Set}(\ldots, \text{card} \lesseqqgtr k)$) and cycles with restricted cardinality ($\mathsf{Cycle}\{\ldots, \text{card} \lesseqqgtr k)$) of admissible classes. In other words, we do not consider arbitrary admissible operators; only those given in the list above.

For convenience, we sometimes abbreviate specifications by writing for instance $\mathcal{P} = \mathsf{Set}(\mathsf{Cycle}(Z))$ instead of $\{\mathcal{P} = \mathsf{Set}(\mathcal{C}), \mathcal{C} = \mathsf{Cycle}(Z)\}$ or $\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$ instead of $\{\mathcal{B} = Z + \mathcal{D}, \mathcal{D} = \mathcal{B} \star \mathcal{B}\}$.

Last but not least, we will frequently use the following notation in later chapters:

$$\begin{aligned} A(z) \ll B(z) &\iff (a_n = \mathrm{o}(b_n)) \ ; \\ A(z) \asymp B(z) &\iff (a_n = \Theta(b_n)) \ ; \\ A(z) \gg B(z) &\iff (b_n = \mathrm{o}(a_n)) \ . \end{aligned}$$

1.2 Bivariate Generating Functions

Combinatorics and the analysis of algorithms often demand informations on probabilistic properties of parameters of combinatorial objects. For instance, it may be of interest to know how many inversions a permutation is likely to have, on average, or —if feasible— in distribution, either exactly or asymptotically.

It turns out that the symbolic methods initially developed for counting both unlabelled and labelled structures adapts nicely to the analysis of various sorts of parameters of such structures.

Bivariate generating functions —ordinary or exponential— can keep track, for instance, of the number of components in a composite construction like a sequence, a (power)set, or a cycle. From such generating functions, there results either explicit probability distributions or, at least, mean and variance expressions. Typical applications include the number of summands in a composition, the number of blocks in a set partition, the number of cycles in a permutation, or the root degree of a tree. Also we can use bivariate generating functions to keep track of the number of inputs of size n which incur a cost k for a certain algorithm.

Consider a class $\mathcal{A}$ of combinatorial structures and a *parameter* ξ on the class being a function from $\mathcal{A}$ to the natural numbers $\mathbb{N}$. Let $a_{k,n}$ denote the number of objects in $\mathcal{A}_n$ whose value of the ξ-parameter equals k: $a_{k,n} = \#\{\alpha \in \mathcal{A} \mid |\alpha| = n \text{ and } \xi(\alpha) = k\}$. The quantities ξ over $\mathcal{A}_n$ are proportional to the probabilities of $a_{k,n}$ since

$$\Pr\{\xi = k \mid \mathcal{A}_n\} = \frac{a_{k,n}}{a_n} = \frac{a_{k,n}}{\sum_k a_{n,k}} \ .$$

An analysis giving the values of the $a_{k,n}$ is, for that reason, often called an analysis in *distribution*.

For unlabelled classes, the *ordinary bivariate generating function* of $\mathcal{A}$ with respect to ξ is defined as[6]

$$A(u,z) = \sum_{n,k} a_{k,n}\, u^k\, z^n$$

and, for labelled classes, the *exponential bivariate generating function* is

$$A(u,z) = \sum_{n,k} a_{k,n}\, u^k \frac{z^n}{n!} \ .$$

The counting of $\mathcal{A}$-structures of size n and ξ-parameter equal to k is recovered by applying the coefficient operator twice,

$$A_{k,n} = \omega_n \cdot [u^k z^n] A(u,z) \ ,$$

where either $\omega_n = 1$ (ordinary bivariate generating function, unlabelled case) or $\omega_n = n!$ (exponential bivariate generating function, labelled case). From the definition, we also see

[6]One often says that $A(u,z)$ is the (bivariate) generating function of $\mathcal{A}$ with the variable u "*marking*" the parameter ξ.

that $A(1,z)$ coincides with the plain (ordinary and exponential) generating function of $\mathcal{A}$. We observe that bivariate generating functions admit the "combinatorial" forms

$$A(u,z) = \sum_{\alpha\in\mathcal{A}} u^{\xi(\alpha)} z^{|\alpha|} \quad \text{and} \quad A(u,z) = \sum_{\alpha\in\mathcal{A}} u^{\xi(\alpha)} \frac{z^{|\alpha|}}{|\alpha|!} \ .$$

The *probability generating function* of ξ over $\mathcal{A}_n$ is

$$\mathbb{E}\Big[u^{\xi} \mid \mathcal{A}_n\Big] = \frac{[z^n]A(u,z)}{[z^n]A(1,z)} \ ,$$

and with the notation $A_n(u) = \omega_n\ [z^n]A(u,z)$ the probability generating function is expressed as $A_n(u)/A_n(1)$. The *mean* or *expectation* of ξ over $\mathcal{A}_n$ is obtained by a differentiation,

$$\mu_{n,\xi} = \mathbb{E}[\xi \mid A_n] = \frac{A_n'(1)}{A_n(1)} \quad \text{where} \quad A_n'(1) = \omega_n\ [z^n]\ \frac{\partial}{\partial u} A(u,z)\bigg|_{u=1} \ ,$$

as results from standard computations with probability generating functions.

In combinatorial enumerations, it is usually the unnormalized quantity appearing in the numerator that plays an important rôle:

$$A_n'(1) \equiv \sum_k k\ a_{k,n} \equiv \sum_{\alpha\in\mathcal{A}_n} \xi(\alpha) \ .$$

This quantity represents the "total" value or "cumulated" value of ξ over $\mathcal{A}_n$. It determines the expectation after normalization, so that we call it the *unnormalized mean*. From what we have seen, its generating function is

$$\frac{\partial}{\partial u} A(u,z)\bigg|_{u=1} = \sum_{\alpha\in\mathcal{A}} \xi(\alpha)\, z^{|\alpha|} \quad \text{or} \quad \frac{\partial}{\partial u} A(u,z)\bigg|_{u=1} = \sum_{\alpha\in\mathcal{A}} \xi(\alpha)\, \frac{z^{|\alpha|}}{|\alpha|!} \ ,$$

which we call the *generating function of (unnormalized) means.*

Estimates of *variance* are useful in order to get an idea of the dispersion of a parameter. For computations, one starts with the second order moment which involves a double differentiation

$$\mathbb{E}\big[\xi^2 \mid \mathcal{A}_n\big] = \frac{A_n''(1)}{A_n(1)} + \frac{A_n'(1)}{A_n(1)} \ .$$

From there, the standard deviation $\sigma_{n,\xi}$ and the variance $\sigma_{n,\xi}^2$ are recovered,

$$\sigma_{n,\xi}^2 \equiv \mathbb{E}\big[(\xi-\mu_n)^2 \mid \mathcal{A}_n\big] = \frac{A_n''(1)}{A_n(1)} + \frac{A_n'(1)}{A_n(1)} - \left(\frac{A_n'(1)}{A_n(1)}\right)^2 \ .$$

Alternatively, we might consider the generating functions for the factorial moments of ξ

$$A^{(r)}(z) = \sum_{n\geq 0} \mathbb{E}[\xi(\xi-1)(\xi-2)\cdots(\xi-r+1) \mid A_n]\, z^n = \sum_{n\geq 0} \mathbb{E}[\xi^{\underline{r}} \mid A_n]\, z^n$$

which can be obtained from successive differentiation of $A(u,z)$ w.r.t. u and setting $u=1$:

$$A^{(r)}(z) = \frac{\partial^r A(u,z)}{\partial u^r}\bigg|_{u=1} \ .$$

By convention, $A^{(0)}(z) = A(z)$. Then

$$\mu_{n,\xi} = \mathbb{E}[\xi \mid \mathcal{A}_n] = \frac{[z^n]A^{(1)}(z)}{[z^n]A(z)} \quad \text{and} \quad \sigma^2_{n,\xi} = \frac{[z^n]A^{(2)}(z)}{[z^n]A(z)} + \mu_{n,\xi} - \mu^2_{n,\xi} \ .$$

1.3 Analysis Techniques

Elementary techniques permit to estimate asymptotically counting results when they are already in the form of combinatorial sums. We examine here the asymptotic determination of coefficients of generating functions directly from an expression for the function itself, and without a need for explicit expansions.

There are many references of analysis techniques, for instance, the surveys by Flajolet [15, 16, 17], Flajolet and Sedgewick [21, 22, 24], Flajolet and Vitter [25], Sedgewick [78], the classical books of Knuth [42, 43, 44], Greene and Knuth [33], Graham, Knuth and Patashnik [31], Bender and Orszag [5], etc.

The crucial observation is that most of the generating functions that occur in combinatorial enumerations are also *analytic functions*: their expansions converge in a neighborhood of the origin and Cauchy's integral formula expresses coefficients of such analytic functions as contour integrals.

The theorems which we are going to see apply to almost any conceivable combinatorial generating function that has a closed form expression, even though the coefficients themselves need not have a tractable or explicit form. In a variety cases, they can furthermore be applied to generating functions *implicitly* defined by *functional equations* provided that local expansions can be obtained and even though the functional equations themselves may admit no closed form solution.

We now introduce some definitions on complex functions and some basic theorems of complex analysis.

Definition 7. *A function $f(z)$ of the complex variable z is* analytic *at point $z = a$ if it is defined in a neighborhood of $z = a$ and is representable there by a convergent power series expansion $f(z) = \sum_{n\geq 0} f_n(z-a)^n$.*

Definition 8. *A function $h(z)$ of the complex variable z is* meromorphic *at point $z = a$ if in a neighborhood of $z = a$ with $z \neq a$ it is representable by an expansion of the form $h(z) = \sum_{n\geq -M} h_n(z-a)^n$, for some $M > 0$. If $h_{-M} \neq 0$, then $h(z)$ is said to have a pole of order M at $z = a$. Equivalently, $h(z)$ is meromorphic at $z = a$ if and only if it can be represented as $f(z)/g(z)$, with $f(z)$ and $g(z)$ being analytic at $z = a$.*

Definition 9. *When a meromorphic function $h(z)$ has a pole of order $M \geq 1$ at $z = a$, then the coefficient h_{-1} is called the* residue *of $h(z)$ at $z = a$ and it is designated by $Res[h(z); z = a]$ or occasionally, using an extension of previous notations, $[1/(z-a)]h(z)$.*

Definition 10. *A function is* analytic *(respectively* meromorphic*) in a domain if and only if it is analytic (respectively meromorphic) at any point of the domain.*

We have the important *residue theorem* whose merit is to relate global properties of a meromorphic function (its integral along curves) to its local properties at designated points, the poles.

Theorem 1 (Cauchy's Residue Theorem). *Let Γ be a simple closed curve oriented positively and situated inside a simply connected region Ω (like a disk), and assume $h(z)$ to be meromorphic in Ω and analytic on Γ. Then*

$$\frac{1}{2\pi i}\int_{\Gamma} h(z)\,dz = \sum_{s} Res[h(z); z = s] \ ,$$

where the sum is extended to all poles s of $h(z)$ enclosed in Γ.

Theorem 2 (Cauchy's Integral Formula). *Let $F(z)$ be an analytic function in an open domain D enclosing the origin, and let Γ be a simple closed curve entirely lying on D that also encloses the origin. Then the n-th coefficient of the Taylor series development of $F(z)$ around $z = 0$ is given by*

$$[z^n]F(z) = \frac{1}{2\pi i}\int_{\Gamma} \frac{F(z)}{z^{n+1}}\,dz \ .$$

We now present some general theorems useful for the singularity analysis of generating functions that are no longer restricted to polar singularities and extends to any function of moderate growth or decay at its dominant singularities.

Definition 11. *A point such that the function ceases to be* analytic *is called* singularity *of a function. Moreover, the* dominant *singularity is one with the smallest modulus.*

The basic principle of the analysis of singularities is the existence of a *correspondence* between *the asymptotic expansion of a function near its dominant singularities* and *the asymptotic expansion of the function's coefficients.* This mapping preserves orders of growth in the sense that larger functions have larger coefficients.

Theorem 3 (Darboux). *Let $F(z)$ be an analytic function in an open disk $|z| < \rho$ and assume that there is a unique singularity in the convergence circle at $z = \rho$. Furthermore, assume that in a neighborhood of $z = \rho$, $F(z)$ satisfies*

$$F(z) = \left(1 - \frac{z}{\rho}\right)^{-\alpha} G(z) + H(z) \ ,$$

for some analytic functions $G(z)$ and $H(z)$ at $z = \rho$, with $G(\rho) \neq 0$, and for some real $\alpha \notin \{0, -1, -2, -3, \ldots\}$. Then,

$$[z^n]F(z) = f_n \sim \rho^{-n}\, n^{\alpha-1}\,\frac{G(\rho)}{\Gamma(\alpha)}\left(1 + \mathrm{O}\left(\frac{1}{n}\right)\right) \ ,$$

where $\Gamma(z)$ denotes Euler's gamma function and $\sim$ denotes asymptotic equivalence.

The standard function scale's theorem is quite similar to Darboux's [18].

Theorem 4 (Standard Function Scale). *Let α be a number not belonging to the set $\{0, -1, -2, \ldots\}$. The coefficient of z^n in*

$$f(z) = (1 - z)^{-\alpha}$$

admits for large n a full asymptotic expansion in descending powers of n,

$$[z^n]f(z) \sim \frac{n^{\alpha-1}}{\Gamma(\alpha)} \left(1 + \sum_{k=1}^{\infty} \frac{e_k(\alpha)}{n^k}\right) ,$$

where $e_k(\alpha)$ is a polynomial in α of degree $2k$.

The polynomial $e_k(\alpha)$ turns out to be divisible by $\alpha(\alpha-1)\ldots(\alpha-k)$. In particular, we have

$$f_n = [z^n]f(z) \sim \frac{n^{\alpha-1}}{\Gamma(\alpha)} \left(1 + \frac{\alpha(\alpha-1)}{2n} + \frac{\alpha(\alpha-1)(\alpha-2)(3\alpha-1)}{24n^2} + \frac{\alpha^2(\alpha-1)^2(\alpha-2)(\alpha-3)}{48n^3} + \cdots\right) .$$

Other general theorems are the transfer lemma and the standard theorem of the scaled coefficients of logarithms [18].

Theorem 5 (Transfer Lemma). *Assume that $F(z)$ is an analytic function in the domain $\Delta = \{z \mid |z| \leq 1+\nu,\ |arg(z-1)| \geq \phi\}$, with $\nu > 0$ and $0 < \phi < \pi/2$. Moreover, as $z \to 1$ in Δ,*

$$F(z) = (1-z)^{-\alpha} \log^{\beta}\left(\frac{1}{1-z}\right) \left[\sum_{j=0}^{m-1} c_j \log^{-j}\left(\frac{1}{1-z}\right) + \mathrm{O}\left(\log^{-m}\left(\frac{1}{1-z}\right)\right)\right]$$

for some $\alpha, -\beta \notin \{0, -1, -2, \ldots\}$. Then as $n \to \infty$

$$f_n = \frac{n^{\alpha-1}}{\Gamma(\alpha)} \log^{\beta} n \left(\sum_{j=0}^{m-1} c'_j \log^{-j} n + \mathrm{O}(\log^{-m} n)\right) .$$

In particular, $c_0 = c'_0$ and the other c'_j can also be explicitly computed from the c_j.

Corollary 1 (Transfer, First Kind). *Assume that $f(z)$ is Δ-analytic and that it satisfies in the intersection of a neighborhood of 1 and of its Δ-domain the condition*

$$f(z) = \mathrm{O}\left((1-z)^{-\alpha} \log^{\beta}\left(\frac{1}{1-z}\right)\right) .$$

Then

$$[z^n]f(z) = \mathrm{O}(n^{\alpha-1} \log^{\beta} n) .$$

Corollary 2 (Transfer, Second Kind). *Assume that $f(z)$ is Δ-analytic and that it satisfies in the intersection of a neighborhood of 1 and of its Δ-domain the condition*

$$f(z) = \mathrm{o}\left((1-z)^{-\alpha} \log^{\beta}\left(\frac{1}{1-z}\right)\right) .$$

Then

$$[z^n]f(z) = \mathrm{o}(n^{\alpha-1} \log^{\beta} n) .$$

Theorem 6 (Standard Coefficient Scale, Logarithms). *Let α be a number not in $\{0, -1, -2, \ldots\}$. The coefficient of z^n in*

$$f(z) = (1-z)^{-\alpha} \left(\frac{1}{z} \log \frac{1}{1-z} \right)^{\beta}$$

admits for large n a full asymptotic expansion in descending powers of $\log n$,

$$[z^n] f(z) \sim \frac{n^{\alpha-1}}{\Gamma(\alpha)} \log^{\beta} n \left(1 + \frac{C_1(\alpha)}{1!} \frac{\beta}{\log n} + \frac{C_2(\alpha)}{2!} \frac{\beta(\beta-1)}{\log^2 n} + \cdots \right) ,$$

where

$$C_k(\alpha) = \Gamma(\alpha) \frac{d^k}{ds^k} \frac{1}{\Gamma(s)} \bigg|_{s=\alpha} .$$

We observe that if $f(z)$ has its dominant singularity at $z = \rho$, then $g(z) = f(z/\rho)$ has its dominant singularity at $z = 1$, and it verifies the "scaling rule":

$$[z^n] f(z) = \rho^n \, [z^n] \, f\left(\frac{z}{\rho}\right) = \rho^n \, [z^n] \, g(z) .$$

On the other hand, other theorems and propositions that we will use apply to implicitly defined functions.

Theorem 7 (Meir and Moon [61, 62, 63]). *Let $\phi(u)$ be an analytic function in the disk $|u| < \rho \leq \infty$ and let $f(z)$ denote the solution of $f(z) = z\,\phi(f(z))$ in a vicinity of $z = 0$. For simplicity, assume that $\gcd\{n \mid \phi_h \neq 0\} = 1$. Then, $[z^n]f(z)$ is asymptotically equivalent to*

$$f_n = [z^n] f(z) \sim \alpha \rho^{-n} n^{-3/2} \left(1 + \mathrm{O}\left(\frac{1}{n}\right) \right) ,$$

where

$$\alpha = \sqrt{\frac{\phi(\tau)}{2\pi \phi''(\tau)}} , \quad \rho = \frac{\tau}{\phi(\tau)}$$

and τ is the root of smallest modulus of the equation $\phi(u) - u\phi'(u) = 0$.

Moreover, the singularity expansion of F at ρ is

$$f(z) - \tau = \sum_{j=1}^{\infty} d_j^* \left(1 - \frac{z}{\rho} \right)^{j/2} ,$$

for some constants d_j^*. In particular, we have

$$d_1^* = \sqrt{\frac{2\phi(\tau)}{\phi''(\tau)}} .$$

Theorem 8 (Lagrange-Börhman's Inversion Formula [22]). *Let $\phi(u) = \sum_{j=0}^{\infty} \phi_j u^j$ be a formal power series with $\phi_0 \neq 0$, and let $f(z)$ be the unique formal power series solution of the equation $f(z) = z\,\phi(f(z))$. The coefficients of $f(z)$, $f^k(z)$, and $\phi(f(z))$ (for an arbitrary series ϕ) are given by*

$$[z^n]f(z) = \frac{1}{n}\,[u^{n-1}](\phi(u))^n \ , \quad [z^n]f^k(z) = \frac{k}{n}\,[u^{n-k}](\phi(u))^n$$

and

$$[z^n]\phi(f(z)) = \frac{1}{n}\,[u^{n-1}](\phi(u))^n\,\phi'(u) \ .$$

Proposition 1 ([21]). *Let $y(z)$ be the solution analytic at the origin of $y(z) = z\,\phi(y(z))$, where ϕ is an entire function with nonnegative Taylor coefficients. Then $y(z)$ has a dominant singularity at $z = \rho$, where $\rho = \tau/\phi(\tau)$, and τ is the unique positive real root of the equation $\phi(\tau) - \tau\phi'(\tau) = 0$. The singular expansion of y at ρ is*

$$y(z) - \tau = \sum_{j=1}^{\infty} d_j^* \left(1 - \frac{z}{\rho}\right)^{j/2}$$

for some constants d_j^*. In particular we have

$$d_1^* = \sqrt{\frac{2\phi(\tau)}{\phi''(\tau)}} \ .$$

Proposition 2 ([21]). *Let $y(z)$ be the solution analytic at the origin of $y(z) = z\,\phi(y(z))$. Then, provided $\phi(u)$ is aperiodic, and with the notation of Proposition 1, the coefficients satisfy*

$$[z^n]y(z) \sim \frac{d_1^*\,\rho^{-n}}{2\sqrt{\pi\,n^3}} \left(1 + \sum_{k=1}^{\infty} \frac{e_k}{n^k}\right) \ ,$$

for some effectively computable coefficient sequence e_k.

Other analytical techniques that are usually applied in combinatorics and analysis of algorithms include Tauberian theorems [6, 14, 18, 33, 34, 83] and saddle point methods [23, 35, 36, 68]. However, we are not using them in this thesis and therefore we will not explain them here.

Chapter 2

Isomorphisms and Orders

Crucial to the definition of unranking, ranking and exhaustive generation is the existence of a total order among the objects of the same size in a given class. In this thesis, the ordering is not a "parameter", but fixed *a priori*. We will choose whatever ordering is more suitable for the design of the unranking, ranking and exhaustive generation algorithms, to make them as efficient as possible. It is a desirable feature that the ordering is "natural", but this will be not a requirement.

Orderings are not really defined w.r.t. combinatorial classes, but w.r.t. specifications. In general, the order in a class $\mathcal{C}$ is not dictated by the combinatorial structure of $\mathcal{C}$ itself, but by the specification that we have used—and there will be many different but equivalent ways to specify a given class. For instance, if permutations are specified by $\mathsf{Seq}(Z)$ then the order of the objects will be different from the order that we would have if we had specified permutations using $\mathsf{Set}(\mathsf{Cycle}(Z))$. Figures 2.1 and 2.2 show the permutations of size 3 in the orders induced by the specifications $\mathsf{Seq}(Z)$ and $\mathsf{Set}(\mathsf{Cycle}(Z))$, respectively.

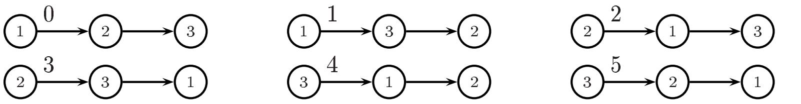


Figure 2.1: Lexicographic order induced by $\mathsf{Seq}(Z)$.

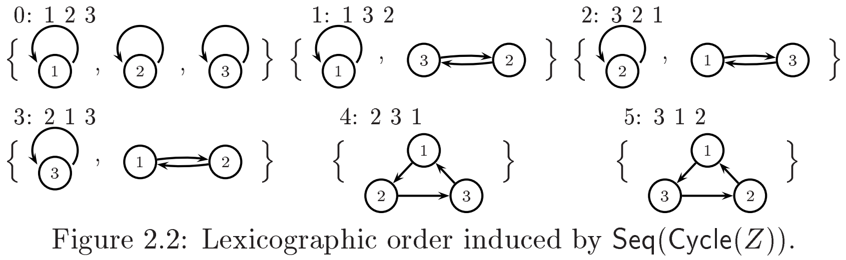


Figure 2.2: Lexicographic order induced by $\mathsf{Seq}(\mathsf{Cycle}(Z))$.

We want our algorithms to be deterministic so that they always produce the same list of objects given an admissible specification for $\mathcal{A}$ and a size n. Since our algorithms should work for any admissible class $\mathcal{A}$, what we need now is to provide an inductive definition of the order $\prec_{\mathcal{A}_n}$ in terms of the orders of the classes and operators used to build $\mathcal{A}$. That is, if a class $\mathcal{C}$ depends on the classes $\mathcal{A}_1, \ldots, \mathcal{A}_k$, $\mathcal{C}=\Psi(\mathcal{A}_1, \ldots, \mathcal{A}_k)$, then the order of $\mathcal{C}$ is defined recursively from the orders of $\mathcal{A}_1, \ldots, \mathcal{A}_k$, $\prec_{\mathcal{C}}=\Psi'(\prec_{\mathcal{A}_1}, \ldots, \prec_{\mathcal{A}_k})$.

2.1 Basic Operators

The order for the ϵ and atomic classes is trivial as they only contain one object.

For labelled and unlabelled unions[1], if $\mathcal{C} = \mathcal{A} + \mathcal{B}$ then we say that the objects in $\mathcal{A}$ come first, then the objects in $\mathcal{B}$. Such "natural" order is called *lexicographic order*. More formally, if γ and γ' are two objects of $\mathcal{C}_n$ then

$$\begin{aligned}\gamma \prec_{\mathcal{C}_n} \gamma' \iff & (\gamma \prec_{\mathcal{A}_n} \gamma' \text{ and } \gamma, \gamma' \in \mathcal{A}) \text{ or } (\gamma \prec_{\mathcal{B}_n} \gamma' \text{ and } \gamma, \gamma' \in \mathcal{B}) \text{ or} \\ & (\gamma \in \mathcal{A} \text{ and } \gamma' \in \mathcal{B}) \ .\end{aligned}$$

It is important to point out here that even though $\mathcal{A} + \mathcal{B} \simeq \mathcal{B} + \mathcal{A}$ the orders induced by these two isomorphic specifications are not the same, i.e., $\prec_{\mathcal{A}+\mathcal{B}} \neq \prec_{\mathcal{B}+\mathcal{A}}$.

The union $\mathcal{A} + \mathcal{B}$ could also consider first the $\mathcal{B}$-objects and, afterwards, the $\mathcal{A}$-objects. Such order is the so-called *anti-lexicographic*. However this order will not be considered here because it makes no difference as far as efficiency is concerned.

Things get more interesting when we consider products. In the case of (unlabelled) Cartesian products $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, if we have two objects $\gamma = (\alpha, \beta)$ and $\gamma' = (\alpha', \beta')$ of $\mathcal{C}$ such that $|\alpha| = |\alpha'| = j$ and $|\beta| = |\beta'| = n - j$ then it is natural to use the *lexicographic* criterion to order them: $\gamma \prec_{\mathcal{C}_n} \gamma'$ if $\alpha \prec_{\mathcal{A}_j} \alpha'$, or $\alpha = \alpha'$ and $\beta \prec_{\mathcal{B}_{n-j}} \beta'$. But we need also to define the order when $|\alpha| \neq |\alpha'|$. The most obvious solution is again the so-called lexicographic order, which is induced by the specification

$$\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \mathcal{A}_2 \times \mathcal{B}_{n-2} + \cdots + \mathcal{A}_n \times \mathcal{B}_0 \ . \tag{2.1}$$

Hence, if $\gamma = (\alpha, \beta)$ and $\gamma' = (\alpha', \beta')$ are two objects of $\mathcal{C}_n$, then

$$\begin{aligned}\gamma \prec_{\mathcal{C}_n} \gamma' \iff & (|\alpha| < |\alpha'|) \text{ or } (j = |\alpha| = |\alpha'| \text{ and } \alpha \prec_{\mathcal{A}_j} \alpha') \text{ or} \\ & (\alpha = \alpha' \text{ and } j = |\alpha| \text{ and } \beta \prec_{\mathcal{B}_{n-j}} \beta') \ .\end{aligned}$$

Other choices are possible, like the *boustrophedonic order* [26, 56], induced by the specification

$$\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_n \times \mathcal{B}_0 + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \mathcal{A}_{n-1} \times \mathcal{B}_1 + \cdots \ . \tag{2.2}$$

Formally,

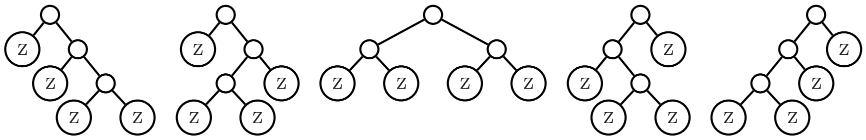
$$\begin{aligned}\gamma \prec_{\mathcal{C}_n} \gamma' \iff & (\min\{|\alpha|, |\beta|\} < \min\{|\alpha'|, |\beta'|\}) \text{ or} \\ & (\min\{|\alpha|, |\beta|\} = \min\{|\alpha'|, |\beta'|\} \text{ and } |\alpha| < |\alpha'|) \text{ or} \\ & (j = |\alpha| = |\alpha'| \text{ and } \alpha \prec_{\mathcal{A}_j} \alpha') \text{ or } (\alpha = \alpha' \text{ and } j = |\alpha| \text{ and } \beta \prec_{\mathcal{B}_{n-j}} \beta') \ .\end{aligned}$$

The boustrophedonic order is relevant because it helps to reduce the average cost of unranking of products as we will see in Part II. However, for the generation problem (Part III) we will only consider the lexicographic order because the boustrophedonic order does not lead to better performance.

[1] For brevity, we shall use the term 'union' when we actually refer to 'disjoint union', unless explicitly started otherwise.

Figure 2.3 depicts the lexicographic and boustrophedonic order for unlabelled binary trees ($\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}$) of size 4.

(a) Lexicographic order



(b) Boustrophedonic order

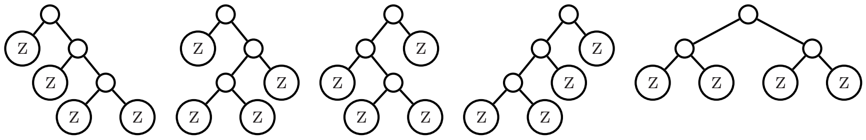


Figure 2.3: Unlabelled binary trees of size 4.

Moreover, it is also possible to define the *anti-lexicographic* order

$$\mathcal{C}_n = \mathcal{A}_n \times \mathcal{B}_0 + \mathcal{A}_{n-1} \times \mathcal{B}_1 + \mathcal{A}_{n-2} \times \mathcal{B}_2 + \cdots + \mathcal{A}_0 \times \mathcal{B}_n \ ,$$

and the *anti-boustrophedonic* order

$$\mathcal{C}_n = \mathcal{A}_n \times \mathcal{B}_0 + \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_{n-1} \times \mathcal{B}_1 + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \cdots \ ,$$

where the Cartesian products '$\times$' must be replaced by partitional products '$\star$' in the case of labelled structures. They will not be considered here any further because they do not provide any noticeable advantage (simplicity or better performance) over the other orders already defined.

When labelled products are considered we need to take also into account the labels of the atoms. Given a partition ρ of $[1..n]$ into a j-subset $\{\ell_1, \ldots, \ell_j\}$ and a $(n-j)$-subset $\{\ell'_1, \ldots, \ell'_{n-j}\}$, we denote (α, β, ρ) the labelled object of size n which results when we relabel the atoms of the pair (α, β) according to ρ. For example, if $\alpha = 21$, $\beta = 213$ and $\rho = \langle\{1,3\},\{2,4,5\}\rangle$ then $(\alpha, \beta, \rho) = 31425$. We say that ρ is a (n,j)-partition. Let $\mathcal{S}_{n,j}$ denote the set of all possible (n,j)-partitions and assume that it is equipped with some well-defined order $\prec_{\mathcal{S}_{n,j}}$. For instance, if $n = 7$ and $j = 4$ then $\mathcal{S}_{n,j}$ contains $\binom{7}{4} = 35$ partitions:

$$\{\langle\{1,2,3,4\},\{5,6,7\}\rangle, \langle\{1,2,3,5\},\{4,6,7\}\rangle, \ldots,$$
$$\langle\{3,5,6,7\},\{1,2,4\}\rangle, \langle\{4,5,6,7\},\{1,2,3\}\rangle\} \ .$$

The set $\alpha \star \beta$ is the set of $\binom{n}{j}$ labelled objects that we obtain by relabelling the atoms of the pair (α, β) according to each of the (n,j)-partitions:

$$\alpha \star \beta = \{(\alpha, \beta, \rho) \,|\, \rho \in \mathcal{S}_{n,j}\} \ .$$

Two different orders for the objects in $\mathcal{A}_j \star \mathcal{B}_{n-j}$ arise then in a natural way stemming from the following specifications:

$$\mathcal{A}_j \star \mathcal{B}_{n-j} = \bigcup_{\alpha \in \mathcal{A}_j} \bigcup_{\beta \in \mathcal{B}_{n-j}} \bigcup_{\rho \in \mathcal{S}_{n,j}} (\alpha, \beta, \rho) \ , \tag{2.3}$$

and

$$\mathcal{A}_j \star \mathcal{B}_{n-j} = \bigcup_{\rho \in \mathcal{S}_{n,j}} \bigcup_{\alpha \in \mathcal{A}_j} \bigcup_{\beta \in \mathcal{B}_{n-j}} (\alpha, \beta, \rho) \ . \tag{2.4}$$

In other words, if we use the specification (2.3), we have that for two objects $\gamma = (\alpha, \beta, \rho)$ and $\gamma' = (\alpha', \beta', \rho')$ of $\mathcal{A}_j \star \mathcal{B}_{n-j}$ then γ is smaller than γ' if and only if α is smaller than α' according to the order in $\mathcal{A}_j$, or $\alpha = \alpha'$ and β is smaller than β' according to the order in $\mathcal{B}_{n-j}$, or $\langle \alpha, \beta \rangle = \langle \alpha', \beta' \rangle$ and ρ is smaller than ρ' in the order induced by $\prec_{\mathcal{S}_{n,j}}$. That is,

$$\begin{aligned} \gamma \prec_{\mathcal{A}_j \star \mathcal{B}_{n-j}} \gamma' \iff & (\alpha \prec_{\mathcal{A}_j} \alpha') \text{ or } (\alpha = \alpha' \text{ and } \beta \prec_{\mathcal{B}_{n-j}} \beta') \text{ or} \\ & (\langle \alpha, \beta \rangle = \langle \alpha', \beta' \rangle \text{ and } \rho \prec_{\mathcal{S}_{n,j}} \rho') \ . \end{aligned}$$

On the other hand, the specification (2.4) first compare partitions (ρ and ρ') and, afterwards, the "internal" labelled pairs ($\langle \alpha, \beta \rangle$ and $\langle \alpha', \beta' \rangle$):

$$\begin{aligned} \gamma \prec_{\mathcal{A}_j \star \mathcal{B}_{n-j}} \gamma' \iff & (\rho \prec_{\mathcal{S}_{n,j}} \rho') \text{ or } (\rho = \rho' \text{ and } \alpha \prec_{\mathcal{A}_j} \alpha') \text{ or} \\ & (\rho = \rho' \text{ and } \alpha = \alpha' \text{ and } \beta \prec_{\mathcal{B}_{n-j}} \beta') \ . \end{aligned}$$

We will use the name *Partition-First* (or PF for short) or *standard* for the first order, whereas the second one will be called *Structure-First* (or SF for short). Figures 2.4 and 2.5 depict labelled binary trees of size 3 in PF and SF order. We write the rank and a conventional linear representation under the pictorial representation of each tree.

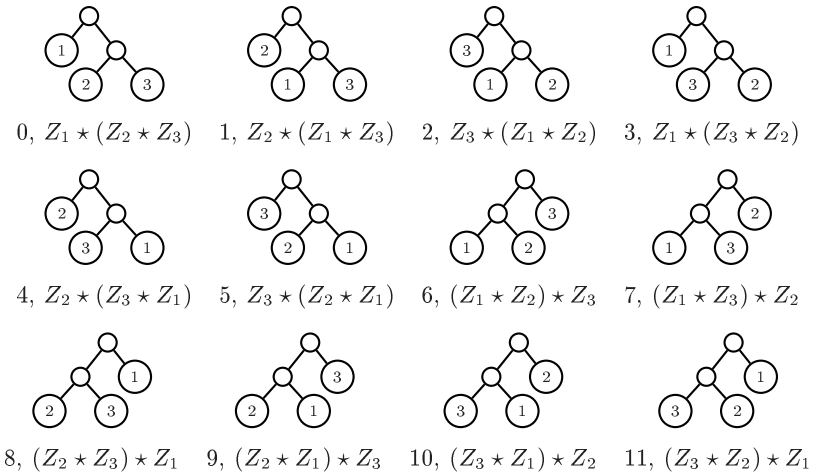


Figure 2.4: Labelled binary trees of size 3 in PF order.

Notice that the choice of the order according to the size of the first component (lexicographic/boustrophedonic) and the "priority" of the label (PF/SF) is orthogonal so that four different orderings can be obtained.

For sequences, we induce the corresponding order using the isomorphism

$$\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \times \mathsf{Seq}(\mathcal{A}) \ . \tag{2.5}$$

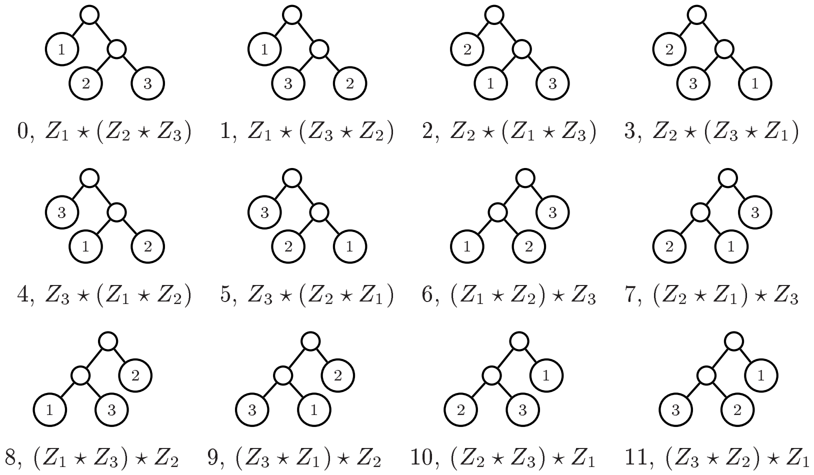


Figure 2.5: Labelled binary trees of size 3 in SF order.

The same isomorphism is used to induce the order of labelled sequences, replacing the unlabelled operators ($+$, $\times$) by their labelled counterparts ($+$, $\star$):

$$\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \star \mathsf{Seq}(\mathcal{A}) \ . \tag{2.6}$$

For unlabelled sets (also called *multisets* or *bags*, see [88] and [90]) — a collection of objects where the repetitions are allowed —, we can obtain a similar isomorphism using the *diagonal operator* (also called *stack operator*) and the (unlabelled) *pointing* [26].

The *diagonal operator* is defined by

$$\begin{aligned} \Delta\mathcal{A} &= \{(\alpha) \mid \alpha \in \mathcal{A}\} + \{(\alpha,\alpha) \mid \alpha \in \mathcal{A}\} + \{(\alpha,\alpha,\alpha) \mid \alpha \in \mathcal{A}\} + \cdots \\ &= \bigcup_{k>0} \{\underbrace{(\alpha,\ldots,\alpha)}_{k \text{ times}} \mid \alpha \in \mathcal{A}\} \ , \end{aligned}$$

and the corresponding diagonal operator for ordinary generating functions is

$$\Delta A(z) = A(z) + A(z^2) + A(z^3) + \cdots = \sum_{k>0} A(z^k) \ .$$

The *pointing* or *marking* operator [88], denoted by ϑ, is a construction $\vartheta\mathcal{A}$ where each atom of each object α of $\mathcal{A}$ is marked giving $|\alpha|$ different objects. Hence, if a_n is the number of objects of size n in $\mathcal{A}$, $n \cdot a_n$ is the number of objects of size n in $\vartheta\mathcal{A}$. The generating functions are related by the derivative

$$\mathcal{B} = \vartheta\mathcal{A} \ \Rightarrow \ B(z) = z\,\frac{d}{dz}A(z) \ .$$

Marking an atom of an object does not increase the size of the object. For instance (see Figure 2.6), when the operation ϑ is applied to $((Z,Z),Z)$, we get a set of 3 marked unlabelled objects (one for each atom in the original object): $(((Z,Z),Z),1) \equiv ((\mathbf{Z},Z),Z)$, $(((Z,Z),Z),2) \equiv ((Z,\mathbf{Z}),Z)$ and $(((Z,Z),Z),3) \equiv ((Z,Z),\mathbf{Z})$.

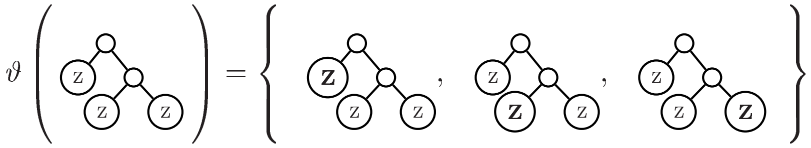


Figure 2.6: $\vartheta((Z,Z),Z)$.

Now we can introduce the following known relation for sets [88],

$$\vartheta \mathsf{Set}(\mathcal{A}) = \Delta\vartheta\mathcal{A} \times \mathsf{Set}(\mathcal{A}) \ . \tag{2.7}$$

The basis of this isomorphism is that the marked sets can be obtained by making some copies of some objects of $\mathcal{A}$, but marking a different atom in each copy. On the other hand, if we mark an atom in a set we might think that we have marked the object that contains the atom, say the m-th copy of the object; on the right hand side we produce the marked object, make m copies of the marked object and attach a set. In Figure 2.7 we sketch the bijection between $\vartheta\mathsf{Set}(\mathcal{A})$ and $\Delta\vartheta\mathcal{A} \times \mathsf{Set}(\mathcal{A})$.

$$\begin{array}{ccc} \vartheta\mathsf{Set}(\mathcal{A}) & = & \Delta\vartheta\mathcal{A} \times \mathsf{Set}(\mathcal{A}) \\ \{\underbrace{\alpha,\ldots,\alpha}_{k-1 \text{ times}}, \alpha \times i, \underbrace{\alpha,\ldots,\alpha}_{p-k \text{ times}}, \beta, \gamma\} & \leftrightarrow & (\alpha \times i)^k \times \{\underbrace{\alpha,\ldots,\alpha}_{p-k \text{ times}}, \beta, \gamma\} \end{array}$$

Figure 2.7: Bijection between $\vartheta\mathsf{Set}(\mathcal{A})$ and $\Delta\vartheta\mathcal{A} \times \mathsf{Set}(\mathcal{A})$.

Notice that Δ and ϑ are not commutative: $\vartheta\Delta \neq \Delta\vartheta$.

We also can define the *odd diagonal* (or *stack*) *powerset operator* over combinatorial classes, denoted by $\Delta^{[\text{odd}]}$. For each object α of $\mathcal{A}$, the odd diagonal includes tuples with an odd number of copies of each object

$$\Delta^{[\text{odd}]}\mathcal{A} = \bigcup_{k>0} \{\alpha^{2k-1} \mid \alpha \in \mathcal{A}\} \ .$$

In the same way, we can define the *even diagonal* (or *stack*) *powerset operator* over combinatorial classes, denoted by $\Delta^{[\text{even}]}$:

$$\Delta^{[\text{even}]}\mathcal{A} = \bigcup_{k>0} \{\alpha^{2k} \mid \alpha \in \mathcal{A}\} \ .$$

The corresponding $\Delta^{[\text{odd}]}$ and $\Delta^{[\text{even}]}$ operators over generating functions are equal to

$$\Delta^{[\text{odd}]}A(z) = \sum_{k>0} A(z^{2k-1}) \quad \text{and} \quad \Delta^{[\text{even}]}A(z) = \sum_{k>0} A(z^{2k}) \ ,$$

respectively.

Let $\hat{\Delta}$ be an operator over generating functions formally defined by the difference between the operators $\Delta^{[\text{even}]}$ and $\Delta^{[\text{odd}]}$:

$$\hat{\Delta}A(z) = \Delta^{[\text{even}]}A(z) - \Delta^{[\text{odd}]}A(z) = \sum_{k>0} \Big(A(z^{2k-1}) - A(z^{2k})\Big) \ .$$

We can write

$$\vartheta \mathsf{PSet}(\mathcal{A}) = \hat{\Delta}\vartheta\mathcal{A} \times \mathsf{PSet}(\mathcal{A}) \ , \tag{2.8}$$

and if $\mathcal{B} = \mathsf{PSet}(\mathcal{A})$ then $\vartheta B(z) = \hat{\Delta}\vartheta A(z) \cdot B(z)$. Unfortunately, neither (2.7) nor (2.8) can be exploited for unranking or exhaustive generation; the reasons for such failure are later explained in Section 2.2.

Like before, operators $\hat{\Delta}$ and ϑ do not commute.

More generally, given a numeric sequence $\{u(k)\}_{k=1}^{\infty}$, the *generalized diagonal* is defined by

$$\Delta_{\{u(k)\}} A(z) = \sum_{k=1}^{\infty} u(k)\ A(z^k) \ .$$

Combinatorially, this means taking a weighted combinatorial of diagonals. Analytically, this also defines a linear operator over generating functions involving a weighted combination of $A(z)$, $A(z^2)$, $A(z^3)$, ...

In particular, if $\mathcal{B} = \mathsf{Cycle}(\mathcal{A})$, we have the following important theorem due to Pólya [88]:

$$\vartheta B(z) = \Delta_{\{\phi(k)\}} \frac{\vartheta A(z)}{1 - A(z)} \ , \tag{2.9}$$

where $\phi(k)$ denotes the Euler totient function of k (that is, number of positive integers not exceeding k and relatively prime to k). Unfortunately, such relation at the level of generating functions has no a direct counterpart at a combinatorial level which can be exploited for the unranking, the ranking or the exhaustive generation of unlabelled cycles.

The unranking and ranking of unlabelled cycles remains as an open problem, whereas we use techniques rather different to those used for the exhaustive generation of unlabelled cycles (see Chapter 8).

For labelled sets and cycles, there are well known isomorphisms that use the pointing operator. The (labelled) pointing has the same definition as in the unlabelled case (each object α generates $|\alpha|$ different objects where each one has a different marked atom).

Now, we can give the known isomorphisms for labelled sets and cycles:

$$\vartheta \mathsf{Set}(\mathcal{A}) = \vartheta\mathcal{A} \star \mathsf{Set}(\mathcal{A}) \quad \text{and} \quad \vartheta \mathsf{Cycle}(\mathcal{A}) = \vartheta\mathcal{A} \star \mathsf{Seq}(\mathcal{A}) \ , \tag{2.10}$$

respectively. That is, since the components of a labelled set and cycle are distinguishable (because their atoms bear distinct labels), the pointing of a set consists of a pointed object plus a set of objects without mark, and the pointing of a cycle is the marked object followed by a sequence. But these isomorphisms cannot be used to induce an order on labelled sets and labelled cycles as we will see soon, in next section.

2.2 Labelled Containers: Sequences, Sets and Cycles

The generation and unranking of sequences will be based upon the order induced by isomorphism (2.6) using unions and labelled products, $\mathsf{Seq}(\mathcal{A}) = \epsilon + Z \star \mathsf{Seq}(\mathcal{A})$.

Unfortunately, the isomorphisms based on the ϑ operator are unusable. Suppose we wanted to induce the order on sets or cycles using (2.10), we would have to define some order on $\vartheta\mathcal{A}_n$ first. Given some order $\prec_{\mathcal{A}_n}$ there are two reasonable ways to order the objects in $\vartheta\mathcal{A}_n$. The first choice is that we list first all n objects that we obtain by marking of the n atoms of the first object α_0 in $\mathcal{A}_n$, then the n objects that we obtain by marking of the n atoms of the second object α_1, etc. Alternatively, we can list first all objects in $\mathcal{A}_n$ with their first atom marked, then all objects in $\mathcal{A}_n$ again but with their second atom marked, etc. Now, if we consider the class $\mathcal{C} = \vartheta\mathcal{A} \star \mathcal{B}$ no matter which order is induced by the product (lexicographic/boustrophedonic; PF/SF) the objects in $\mathcal{C}_n$ will not come in any of the two orders we might expect for a marked class, namely, something like 1) $(\alpha_0, 1), (\alpha_0, 2), \ldots, (\alpha_0, n), (\alpha_1, 1), (\alpha_1, 2), \ldots, (\alpha_1, n), (\alpha_2, 1), \ldots$ or 2) $(\alpha_0, 1), (\alpha_1, 1), \ldots, (\alpha_{N-1}, 1), (\alpha_0, 2), (\alpha_1, 2), \ldots, (\alpha_{N-1}, 2), (\alpha_0, 3), \ldots$ So the order induced in $\mathcal{C}_n$ by the definition $\mathcal{C} = \vartheta\mathcal{A} \star \mathcal{B}$ cannot coincide with the order induced by a definition $\mathcal{C} = \vartheta\mathcal{D}$ for some class $\mathcal{D}$ and this renders the isomorphisms (2.10) useless for their application to the unranking/ranking problem. The isomorphisms (2.7) and (2.8) for unlabelled sets and powersets are equally unusable. We have thus to consider alternative combinatorial constructions which help us overcome this problem. We examine these alternatives in this and the next section.

The elegant solution for the case of labelled *containers*, i.e., labelled sets and labelled cycles, is based upon the so-called *boxed product*. Boxed products, denoted ${}^{\Box}\!\star$, are very similar to labelled products, but with the condition that each pair in $\alpha\,{}^{\Box}\!\star\beta$ must be relabelled in such a way that the smallest label is attached to some atom in the first component. Since the smallest label must be used to label some α-atom there are only $\binom{|\alpha|+|\beta|-1}{|\alpha|-1}$ objects in $\alpha\ {}^{\Box}\!\star\ \beta$. For instance, $(Z_1, Z_2)\ {}^{\Box}\!\star\ (Z_2, Z_1)$ just generates $\binom{4-1}{2-1} = 3$ different new labelled objects: $((Z_1, Z_2), (Z_4, Z_3))$, $((Z_1, Z_3), (Z_4, Z_2))$ and $((Z_1, Z_4), (Z_3, Z_2))$.

The definition of lexicographic and boustrophedonic order, and the SF and PF order for boxed products are immediate. For instance, Figure 2.8 shows the objects of $\mathcal{B} = \epsilon + Z\ {}^{\Box}\!\star\ (\mathcal{B} \star \mathcal{B})$ of size 4 in lexicographic-PF order[2]; Figure 2.9 shows the same objects in lexicographic-SF order.

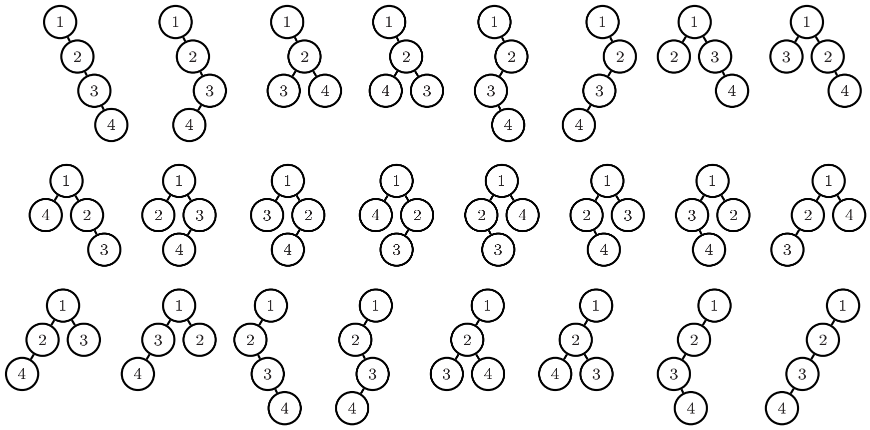


Figure 2.8: All trees of size 4 in $\mathcal{B} = \epsilon + Z\ {}^{\Box}\!\star\ (\mathcal{B} \star \mathcal{B})$ in lexicographic-PF order.

[2]We have omitted the ϵ-objects (leaves) for clarity

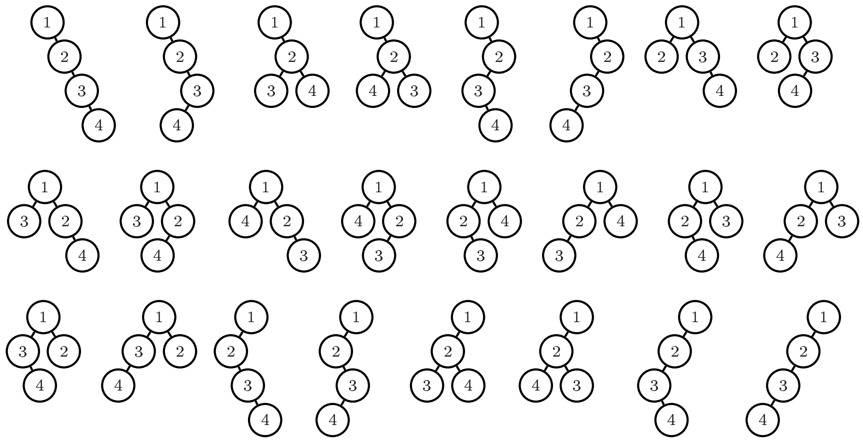


Figure 2.9: All trees of size 4 in $\mathcal{B} = \epsilon + Z\,{}^{\square}\!\star (\mathcal{B} \star \mathcal{B})$ in lexicographic-SF order.

The exponential generating functions verify the following equation:

$$\mathcal{C} = \mathcal{A}\,{}^{\square}\!\star \mathcal{B} \;\Rightarrow\; C(z) = \int_0^z \left(\frac{d}{dt}A(t)\right) B(t)\,dt\ .$$

Once boxed product and the order(s) it induces have been introduced, we will use the following isomorphisms for labelled sets and labelled cycles:

$$\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}\,{}^{\square}\!\star \mathsf{Set}(\mathcal{A}) \tag{2.11}$$

and

$$\mathsf{Cycle}(\mathcal{A}) = \mathcal{A}\,{}^{\square}\!\star \mathsf{Seq}(\mathcal{A})\ , \tag{2.12}$$

respectively.

There also is an alternative way to define boxed products, namely, that the smallest label is attached to an atom of the second member of the pair. It is easy to see that this alternative definition, which we call *right boxed product* and denote $\star^{\square}$, satisfies $\mathcal{A}\,{}^{\square}\!\star\mathcal{B} \simeq \mathcal{B}\star^{\square}\mathcal{A}$. Later on, we will consider the alternative isomorphisms for sets and cycles using $\star^{\square}$ and the effects of their use on the performance of the algorithms for unranking and ranking.

There exists a very close relation between the boxed product and the pointing:

$$\vartheta\mathcal{C} = \vartheta\mathcal{A} \star \mathcal{B} \iff \mathcal{C} = \mathcal{A}\,{}^{\square}\!\star \mathcal{B}\ .$$

As sequences, sets and cycles can be expressed using unions, partitional products and boxed products, we say that a specification is in *standard form* if it is expressed with these three operators and the substitution. It is rather easy to transform a given specification into its equivalent standard form; for instace, functional graphs $\mathcal{F} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{T}))$, where $\mathcal{T} = Z\star\mathsf{Set}(\mathcal{T})$, can be specified in the standard form by $\{\mathcal{F} = \mathsf{Union}(\epsilon, \mathcal{S})$, $\mathcal{S} = \mathsf{BProd}(C, \mathcal{F})$, $\mathcal{C} = \mathsf{BProd}(\mathcal{T}, \mathcal{Q})$, $\mathcal{Q} = \mathsf{Union}(\epsilon, \mathcal{R})$, $\mathcal{R} = \mathsf{Prod}(\mathcal{T}, \mathcal{Q})$, $\mathcal{T} = \mathsf{Prod}(Z, \mathcal{U})$, $\mathcal{U} = \mathsf{Union}(\epsilon, \mathcal{W})$, $\mathcal{W} = \mathsf{BProd}(\mathcal{T}, \mathcal{U})\}$.

2.3 Unlabelled Containers: Sequences, Powersets and Sets

Unlabelled sequences pose no special problem and we handle them using isomorphism (2.5): $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \times \mathsf{Seq}(\mathcal{A})$.

To define the order among unlabelled powersets we decompose each powerset in two parts: a *leading block* containing all components of smallest size, and a *tail* with the remaining components. We will define the order using a lexicographic criterion to compare leading blocks and the following isomorphism

$$\mathcal{B} = \mathsf{PSet}(\mathcal{A}) = \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \mathsf{PSet}(\mathcal{A}_j, \mathrm{card} = k) \times \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \tag{2.13}$$

where $\mathcal{A}_{>j}$ is the set of objects in $\mathcal{A}$ whose size is strictly larger than j.

Let β and β' be two objects of $\mathcal{B}_n$, $\beta = \{\alpha_1, \alpha_2, \ldots, \alpha_k, \gamma\}$ and $\beta' = \{\alpha'_1, \alpha'_2, \ldots, \alpha'_{k'}, \gamma'\}$, where $\{\alpha_1, \ldots, \alpha_k\} \in \mathsf{PSet}(\mathcal{A}_j, \mathrm{card} = k)$ forms the leading block of β, $\{\alpha'_1, \ldots, \alpha'_{k'}\} \in \mathsf{PSet}(\mathcal{A}_{j'}, \mathrm{card} = k')$ is the leading block of β', $\gamma \in \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})$ is the tail of β and $\gamma' \in \mathsf{PSet}_{n-j'k'}(\mathcal{A}_{>j'})$ is the tail of β'. We assume furthermore that $\alpha_1 \prec \ldots \prec \alpha_k$ and $\alpha'_1 \prec \ldots \prec \alpha'_{k'}$. Then

$$\begin{aligned}\beta \prec_{\mathcal{B}_n} \beta' \iff & (j < j') \text{ or } (j = j' \text{ and } k > k') \text{ or} \\ & (j = j' \text{ and } k = k' \text{ and } \exists i : (\alpha_i \prec_{\mathcal{A}_j} \alpha'_i \text{ and } \forall l : 1 \le l < i : \alpha_l = \alpha'_l)) \text{ or} \\ & (\{\alpha_1, \ldots, \alpha_k\} = \{\alpha'_1, \ldots, \alpha'_{k'}\} \text{ and } \gamma \prec_{\mathcal{B}_{n-kj}} \gamma')\ .\end{aligned} \tag{2.14}$$

This order is very natural and can be equivalently defined as the lexicographic ordering of powersets given their "canonical" representations, i.e. $\gamma = \{\gamma_1, \ldots, \gamma_m\}$ with $\gamma_i \prec_{\mathcal{A}} \gamma_{i+1}$ for all $i : 1 \le i < m$, being $\gamma_i \prec_{\mathcal{A}} \gamma_{i+1}$ if and only if $|\gamma_i| < |\gamma_{i+1}|$, or $j = |\gamma_i| = |\gamma_{i+1}|$ and $\gamma_i \prec_{\mathcal{A}_j} \gamma_{i+1}$.

We can also consider a few alternatives of the basic isomorphism (2.13). For instance, the order of powersets depends on the size and the cardinality of their leading blocks. Therefore we might:

- Order the powersets in ascending (S) or descending (s) order of their leading block components' size; i.e. we can use lexicographic (S) or antilexicographic (s) order for the size of the components of the leading blocks.
- Order the powersets in ascending (C) or descending (c) order of their number of objects in their leading blocks; again we can use a lexicographic (C) or antilexicographic (c) criterion on this.
- Order first on leading block components' size (S/s), then on the leading block cardinalities (C/c), or the other way around. This is similar with the choice that we had when defining SF and PF orders for labelled products.

In Table 2.1 we show different isomorphisms for unlabelled powersets which induce different orders depending on the size and cardinality of leading blocks. On the other hand, we can define the leading block as the subset containing the components of smallest (L) or largest (l) size (Table 2.2).

Isomorphism	Notation
$\epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=1}^{\lfloor n/j \rfloor} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{SC}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{Sc}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{j=n}^{1} \bigcup_{k=1}^{\lfloor n/j \rfloor} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{sC}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{j=n}^{1} \bigcup_{k=\lfloor n/j \rfloor}^{1} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{sc}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{k=1}^{\lfloor n/j \rfloor} \bigcup_{j=1}^{n} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{CS}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{j=1}^{n} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{cS}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{k=1}^{\lfloor n/j \rfloor} \bigcup_{j=n}^{1} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{Cs}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{j=n}^{1} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \cdots$	$\ldots\mathsf{cs}\ldots$

Table 2.1: Isomorphisms for $\mathsf{PSet}(\mathcal{A})$.

Isomorphism	Notation
$\epsilon + \bigcup_{n>0} \bigcup_{\cdots}^{\cdots} \bigcup_{\cdots}^{\cdots} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \mathsf{PSet}_{n-kj}(A_{>j})$	$\mathsf{L}\ldots$
$\epsilon + \bigcup_{n>0} \bigcup_{\cdots}^{\cdots} \bigcup_{\cdots}^{\cdots} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \mathsf{PSet}_{n-kj}(A_{<j})$	$\mathsf{l}\ldots$

Table 2.2: Leading block of components of minimum or maximum size induce different orders.

The order in the tail is defined recursively. Thus, the only thing that we needed to complete the definition of order in powersets is the order within leading blocks when they have the same cardinality and components' size. To compare the leading blocks of two powersets where both have k components of size j, the obvious choice is to arrange their components in ascending order of ranks and compare lexicographically (this order is denoted by R).

Altogether we have $2^4 = 16$ combinations of these criteria giving rise to corresponding different orders. For instance, the order induced by (2.13) is LScR order, and the order induced by the following specification[3]

$$\mathsf{PSet}(\mathcal{A}) = \epsilon + \bigcup_{n>0} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{j=n}^{1} \mathsf{PSet}(\mathcal{A}_j, \text{card} = k) \times \mathsf{PSet}_{n-kj}(\mathcal{A}_{<j})$$

[3] We will systematically use the notation $\Psi_n(\mathcal{A})$, where Ψ is an operator instead of the more cumbersome $(\Psi(\mathcal{A}))_n$ to indicate the subset of objects of size n in $\Psi(\mathcal{A})$.

is denoted IcSR.

For sets we have the same criteria as for powersets but now the components can be repeated. Then we also can take into account the *frequencies* of the components lexicographically (the so-called F order). For instance, if $|a| = |b| = |c| = 1$, then $\{a, b, b, b, c, c\} \equiv abbbcc \succ abbbbc \equiv \{a, b, b, b, b, c\}$ in ascending rank (R) because the 5-th component of $abbbbc$ is smaller than the 5-th component of $abbbcc$, but $\{a, b, b, b, c, c\} \equiv ab^3c^2 \prec ab^4c \equiv \{a, b, b, b, b, c\}$ in ascending frequency (F) because the sequence $(1, 3, 2)$ is smaller than $(1, 4, 1)$ in lexicographic order. We note that the R order is equivalent to comparing frequencies in descending order, while the F order is equivalent to comparing ranks in descending order.

Thus, for sets there exist $2^5 = 32$ different possible orders (s/S, c/C, l/L, R/F, and order with first on leading block components' size then the cardinality or vice versa). For instance, in Table 2.3 we show an example involving how these different isomorphisms induce different orders for $\mathsf{Set}(\mathcal{A})$, where $\mathcal{A} = \{a, b, A, B\}$, with $|a| = |b| = 1$ and $|A| = |B| = 2$.

Rank	**0**	**1**	**2**	**3**	**4**	**5**	**6**
lSCR	aaA	aaB	abA	abB	bbA	bbB	aaaa
lScR	aaaa	aaab	aabb	abbb	bbbb	aaA	aaB
lsCR	AA	AB	BB	aaA	aaB	abA	abB
lscR	AA	AB	BB	aaaa	aaab	aabb	abbb

7	**8**	**9**	**10**	**11**	**12**	**13**
aaab	aabb	abbb	bbbb	AA	AB	BB
abA	abB	bbA	bbB	AA	AB	BB
bbA	bbB	aaaa	aaab	aabb	abbb	bbbb
bbbb	aaA	aaB	abA	abB	bbA	bbB

Table 2.3: Different isomorphisms give different orders.

It is also possible to consider boustrophedonic variants for the order of components' size and number of components of the leading blocks, i.e., we can replace the lexicographic criterion s/S or c/C by a boustrophedonic criterion $\mathsf{s}_b/\mathsf{S}_b$ or $\mathsf{c}_b/\mathsf{C}_b$. Of course, other orders also can be defined, but, in principle, they will not give a better performance and there is no compelling reason to adopt them.

We have been not able to find similar recursive decompositions for unlabelled cycles. Hence, we cannot provide efficient solutions for the unranking and ranking of classes involving unlabelled cycles. But the exhaustive generation of unlabelled cycles is efficiently solved using an specific order that will be introduced in Chapter 8.

The isomorphisms for unlabelled sets and powersets could also be used for labelled sets, replacing Cartesian products by partitional products. However, the isomorphisms presented in Section 2.2 using the boxed product are simpler, clearer, more elegant and more efficient.

In short, we will use the LScR order for the unranking and iteration problems because it is a very natural choice. We note that this LScR order is the lexicographic order of sets and powersets assuming the components are ordered in ascending order of ranks and the natural extension of $\prec_{\mathcal{A}_n}$ to $\prec_{\mathcal{A}}$. We will also use the boustrophedonic order $\mathsf{LS}_b\mathsf{c}_b\mathsf{R}$ for unranking because it yields better performance than the lexicographic variant.

Since we cannot use the isomorphisms of Section 2.1 to cope with unlabelled powersets, sets or cycles, it seems that we cannot reduce the set of combinatorial unlabelled operators to some simpler set. Hence, we say that an unlabelled specification as defined by Definition 5 is already in standard form. For instance, rooted unlabelled trees $\mathcal{T} = Z \times \mathsf{Set}(\mathcal{T})$ can be specified in standard form by $\{\mathcal{T} = \mathsf{Prod}(Z, \mathcal{U}), \mathcal{U} = \mathsf{Set}(\mathcal{T})\}$.

2.4 Other Containers

In this section, we consider the extension of the "container" operators Seq, Set, ... so that restrictions on the number of components (cardinality) can be also specified. We also study here the order induced by substitutions.

2.4.1 Restricted Cardinalities

The ability to specify restrictions of cardinality in various combinatorial containers has useful applications. The explicit cardinality restrictions ease the specification of labelled classes like non-plane ternary trees, hierarchies, 3-balanced hierarchies, surjections, etc. (Table 1.2); and unlabelled classes as well: integer partitions, non-plane binary trees, non-plane ternary trees, unlabelled hierarchies, partitions without repetition, etc. (Table 1.3). Although many restrictions can be coped with the collection of already considered operators, it is convenient to give more expressiveness to our specification language by including cardinality restrictions as part of that language. For instance, it is clearer and more easy to understand $\mathsf{Set}(\mathcal{B}, \text{card} \geq 2)$ than $\mathcal{B} \,{}^{\square}\!\star (\mathcal{B} \,{}^{\square}\!\star \mathsf{Set}(\mathcal{B}))$.

In general we shall use the notation $\phi(\mathcal{B}, \text{card } \mathsf{R}\ k)$ where ϕ is Seq, Set, PSet or Cycle, R is a relation operator ($=, <, \geq, \ldots$) and k is a positive integer. Moreover, for unlabelled classes, we will also use the notation $\mathcal{A}^{\text{condition}}$ for sequences with restricted cardinality, $\mathcal{A}^{\{\text{condition}\}}$ for powersets and $\mathcal{A}^{\{\!\{\text{condition}\}\!\}}$ for sets. For the labelled case, we will use a similar notation, $\mathcal{A}^{\text{condition}}$ for sequences with restricted cardinality, $\mathcal{A}^{\{\text{condition}\}}$ for sets and $\mathcal{A}^{\langle\text{condition}\rangle}$ for cycles.

For labelled classes we will consider isomorphisms that just imply unions and partitional and boxed products [22]. Such isomorphisms are a generalization of the isomorphisms presented in equations (2.11) and (2.12). In Table 2.4 we present the isomorphisms for labelled sequences, sets and cycles with restricted cardinality, where $\mathcal{A}^0 = \mathcal{A}^{\leq 0} = \mathcal{A}^{\{0\}} = \mathcal{A}^{\{\leq 0\}} = \epsilon$, $\mathcal{A}^{\langle 0\rangle} = \mathcal{A}^{\langle \leq 0\rangle} = \emptyset$, $\mathcal{A}^{\geq 0} = \mathsf{Seq}(\mathcal{A})$, $\mathcal{A}^{\{\geq 0\}} = \mathsf{Set}(\mathcal{A})$ and $\mathcal{A}^{\langle \geq 0\rangle} = \mathsf{Cycle}(\mathcal{A})$. It is also possible to consider other isomorphisms like $\mathcal{A}^{\geq k} = \mathcal{A} \star \mathcal{A}^{\geq k-1}$, $\mathcal{A}^{\{\geq k\}} = \mathcal{A} \,{}^{\square}\!\star \mathcal{A}^{\{\geq k-1\}}$ and $\mathcal{A}^{\langle \geq k\rangle} = \mathcal{A} \,{}^{\square}\!\star \mathcal{A}^{\geq k-1}$.

For unlabelled classes, the isomorphisms are based in the ideas developed in Section 2.3. In Table 2.5 we show a summary of these isomorphisms, where $\mathcal{A}^0 = \mathcal{A}^{\leq 0} = \mathcal{A}^{\{0\}} = \mathcal{A}^{\{\leq 0\}} = \mathcal{A}^{\{\!\{0\}\!\}} = \mathcal{A}^{\{\!\{\leq 0\}\!\}} = \epsilon$, $\mathcal{A}^{\geq 0} = \mathsf{Seq}(\mathcal{A})$, $\mathcal{A}^{\{\geq 0\}} = \mathsf{PSet}(\mathcal{A})$ and $\mathcal{A}^{\{\!\{\geq 0\}\!\}} = \mathsf{Set}(\mathcal{A})$. In the same way, the alternative isomorphisms that we have discussed in Section 2.3 (and the orders they induce like, for instance, $\mathcal{A}^{\geq k} = \mathcal{A}^k \times \mathsf{Seq}(\mathcal{A})$) can also be generalized to the case where we have cardinality restrictions.

Lab. Operator	**Isomorphism** $(k > 0)$
Seq	$\mathcal{A}^{k} = \mathcal{A} \star \mathcal{A}^{k-1}$
	$\mathcal{A}^{\leq k} = \epsilon + \mathcal{A} \star \mathcal{A}^{\leq k-1}$; $\mathcal{A}^{\geq k} = \mathcal{A}^{k} \star \mathsf{Seq}(\mathcal{A})$
Set	$\mathcal{A}^{\{k\}} = \mathcal{A}\ {}^{\square}\!\!\star \mathcal{A}^{\{k-1\}}$
	$\mathcal{A}^{\{\leq k\}} = \epsilon + \mathcal{A}\ {}^{\square}\!\!\star \mathcal{A}^{\{\leq k-1\}}$; $\mathcal{A}^{\{\geq k\}} = \mathcal{A}^{\{k\}}\ {}^{\square}\!\!\star \mathsf{Set}(\mathcal{A})$
Cycle	$\mathcal{A}^{\langle k\rangle} = \mathcal{A}\ {}^{\square}\!\!\star \mathcal{A}^{k-1}$
	$\mathcal{A}^{\langle\leq k\rangle} = \mathcal{A}\ {}^{\square}\!\!\star \mathcal{A}^{\leq k-1}$; $\mathcal{A}^{\langle\geq k\rangle} = \mathcal{A}^{\langle k\rangle}\ {}^{\square}\!\!\star \mathsf{Seq}(\mathcal{A})$

Table 2.4: Isomorphisms for labelled restricted containers.

Unl. Operator	**Isomorphism** $(k > 0)$
Seq	$\mathcal{A}^{k} = \mathcal{A} \times \mathcal{A}^{k-1}$
	$\mathcal{A}^{\leq k} = \epsilon + \mathcal{A} \times \mathcal{A}^{\leq k-1}$; $\mathcal{A}^{\geq k} = \mathcal{A} \times \mathcal{A}^{\geq k-1}$
PSet	$\mathcal{A}^{\{k\}} = \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{q=\min\{\lfloor n/j\rfloor,k\}}^{1} \mathcal{A}_j^{\{q\}} \times \mathcal{A}_{>j}^{\{k-q\}}$
	$\mathcal{A}^{\{\leq k\}} = \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{q=\min\{\lfloor n/j\rfloor,k\}}^{1} \mathcal{A}_j^{\{q\}} \times \mathcal{A}_{>j}^{\{\leq k-q\}}$
	$\mathcal{A}^{\{\geq k\}} = \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{q=\lfloor n/j\rfloor}^{1} \mathcal{A}_j^{\{q\}} \times \mathcal{A}_{>j}^{\{\geq k-q\}}$
Set	$\mathcal{A}^{\{\!\{k\}\!\}} = \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{q=\min\{\lfloor n/j\rfloor,k\}}^{1} \mathcal{A}_j^{\{\!\{q\}\!\}} \times \mathcal{A}_{>j}^{\{\!\{k-q\}\!\}}$
	$\mathcal{A}^{\{\!\{\leq k\}\!\}} = \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{q=\min\{\lfloor n/j\rfloor,k\}}^{1} \mathcal{A}_j^{\{\!\{q\}\!\}} \times \mathcal{A}_{>j}^{\{\!\{\leq k-q\}\!\}}$
	$\mathcal{A}^{\{\!\{\geq k\}\!\}} = \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{q=\lfloor n/j\rfloor}^{1} \mathcal{A}_j^{\{\!\{q\}\!\}} \times \mathcal{A}_{>j}^{\{\!\{\geq k-q\}\!\}}$

Table 2.5: Isomorphisms for unlabelled restricted containers.

2.4.2 Card-Lexicographic Order

So far we have considered isomorphisms and orders which "see" a container as a product of some component or block followed by some container (the tail). Another possible way of ordering objects in container classes is what we call *card-lexicographic order* which first compares container objects according to their number of components: $\mathsf{Seq}(\mathcal{A}) = \epsilon + \bigcup_{k>0} \mathcal{A}^{k}$, $\mathsf{PSet}(\mathcal{A}) = \epsilon + \bigcup_{k>0} \mathcal{A}^{\{k\}}$, $\mathsf{Set}(\mathcal{A}) = \epsilon + \bigcup_{k>0} \mathcal{A}^{\{\!\{k\}\!\}}$ and $\mathsf{Cycle}(\mathcal{A}) = \bigcup_{k>0} \mathcal{A}^{\langle k\rangle}$. Then to define the order within a class $\phi(\mathcal{A}, \mathrm{card} = k)$ we can use similar ideas to those on previous sections.

Similar isomorphisms give the *anti-card-lexicographic* order where the objects with larger cardinality come first.

2.4.3 Substitution

Another admissible operator is the substitution $\mathcal{B}$ into $\mathcal{A}$ (or the composition of $\mathcal{A}$ and $\mathcal{B}$), $\mathcal{A} \circ \mathcal{B}$ (also noted by $\mathsf{Subst}(\mathcal{A}, \mathcal{B})$ or $\mathcal{A}[\mathcal{B}]$).

For labelled classes, the substitution consists by replacing each atom of each object of the class $\mathcal{A}$ by every possible object in $\mathcal{B}$. That is, $\mathcal{A}$-objects whose atoms are replaced by $\mathcal{B}$-objects. The corresponding isomorphism can be defined using labelled unions, labelled sets of fixed cardinality and Cartesian products (see isomorphism (1.2)):

$$\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^{\{k\}} \ .$$

In the unlabelled case, the substitution is defined using unlabelled unions, unlabelled sequences of fixed cardinality and Cartesian products (see isomorphism (1.1)):

$$\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^{k} \ .$$

Although the substitution is expressed in terms of an infinitive number of unions, when generating, unranking, ranking, etc. objects of size n, we need only consider a finite number of terms and everything is then well defined.

Part II

Unranking and Ranking

In this part we study the *unranking* and *ranking* problems. It is divided in four chapters covering the unranking of labelled classes (Chapter 3), the unranking of unlabelled classes (Chapter 4), an experimental study of the performance of our unranking algorithms (Chapter 5), and the ranking of labelled and unlabelled classes (Chapter 6).

Here we draw upon the seminal work of Flajolet *et al.* [26] on the random generation of labelled combinatorial structures. By *unranking*, we mean the generation of a combinatorial object given its *rank*, its size and a finite description (specification) of the combinatorial class to which the object belongs. The *rank* of an object is the number of objects of the same size in the class that are smaller than the object, according to some fixed ordering. While there exist ad-hoc efficient algorithms to generate permutations, strings, binary trees and many other combinatorial structures (see for instance [10, 51]), there have been few attempts to devise more general algorithms, where a specification of the combinatorial class is a parameter itself. This is the approach successfully applied to counting and random generation by Flajolet *et al.*. One of our main contributions is to show the feasibility of this approach to cope with the unranking and ranking problems.

In Chapter 3 we study the unranking of labelled combinatorial structures. Its first section describes the algorithms which we propose to unrank labelled classes. The following section is devoted to the analysis of the performance of the unranking algorithms. We give results on the worst-case complexity of unranking, and provide the so-called Υ- and $\hat{\Upsilon}$-algebras for the average and probability distribution of the cost of unranking labelled classes with lexicographic ordering. As an example of application of the Υ- and $\hat{\Upsilon}$-algebras, we consider the average cost and variance of unranking simply generated families of (non)-ordered trees. The last section of this chapter explores the use of alternative isomorphisms and its effects on the performance of unranking.

Chapter 4 is about the unranking of unlabelled combinatorial structures and we address similar questions as in the previous chapter.

In the following chapter we present an experimental study of our unranking algorithms. We perform an exhaustive and systematic comparison of the predictions of our theoretical analysis with the experimental data, and we also compare the performance of unranking algorithms with that of random generation algorithms.

In Chapter 6 we consider the ranking of labelled and unlabelled combinatorial structures. We first introduce the important issue of the representation of combinatorial objects (but we defer a full and deeper treatment to Part IV) and then the algorithms to rank them. We also present the analysis of the performance of these algorithms.

Some of the results of this part have been presented in the papers *Unranking of Labelled Combinatorial Structures* [54], *A Generic Approach for the Unranking of Labeled Combinatorial Classes* [56], and *An Experimental Study of Unranking Algorithms* [59]. Other results are new and have not been published yet.

Chapter 3

Unranking of Labelled Classes

In this chapter, we formalize the unranking problem, we propose efficient general algorithms to solve it for labelled classes, and we analyze the performance of these algorithms.

Given a specification of a combinatorial class ($\mathcal{A}$), a size (n) and a rank from that class (r) —the rank goes from 0 to $a_n - 1$—, the unranking computes the object of the class with the given rank and size.

There are many algorithms that solve the unranking problem for particular and specific classes like the examples presented in preliminaries (binary trees [69], Cayley trees [49], Dyck paths [50], strings [37] and many other combinatorial structures [13, 29, 39, 46, 47, 67, 70, 85, 86]; also access The Combinatorial Object Server [10]). Here, we consider generic solutions where the algorithm receives an specification of the class as a parameter.

From now on, for a labelled object γ belonging to some partitional or boxed product we will make the labelling explicit by giving its rank. That is, if ρ is the ℓ-th (n, j)-partition in $\mathcal{S}_{n,j}$ then we use $\langle\alpha, \beta, \ell\rangle$ to represent[1] the object (α, β, ρ). Also we use ${}^{\square}\langle\alpha, \beta, \ell\rangle$ to represent the object in $\alpha \,{}^{\square}\!\star\, \beta$ relabelled by the ℓ-th (n, j)-partition in $\mathcal{S}_{n,j}$. The order in $\prec_{\mathcal{S}_{n,j}}$ and the reconstruction of the label given its rank is described in full detail in Chapter 9 (Section 9.2). For instance, $\langle\langle Z, Z, 0\rangle, \langle Z, Z, 1\rangle, 2\rangle$ denotes the binary tree $((Z_1, Z_4), (Z_3, Z_2))$, and ${}^{\square}\langle{}^{\square}\langle Z, {}^{\square}\langle Z, Z, 0\rangle, 0\rangle, {}^{\square}\langle{}^{\square}\langle Z, Z, 0\rangle, {}^{\square}\langle Z, \epsilon, 0\rangle, 0\rangle, 5\rangle$ denotes the permutation $\mathsf{Set}(\mathsf{Cycle}(Z_1, Z_3, Z_5), \mathsf{Cycle}(Z_2, Z_4), \mathsf{Cycle}(Z_6))$ because

$$\begin{aligned}
&{}^{\square}\langle{}^{\square}\langle Z, \mathsf{Cycle}(Z_1, Z_2), 0\rangle, {}^{\square}\langle\mathsf{Cycle}(Z_1, Z_2), \mathsf{Cycle}(Z_1), 0\rangle, 5\rangle \\
&\qquad \equiv {}^{\square}\langle\mathsf{Cycle}(Z_1, \mathsf{Cycle}(Z_2, Z_3)), \mathsf{Set}(\mathsf{Cycle}(Z_1, Z_2), \mathsf{Cycle}(Z_3)), 5\rangle \\
&\qquad \equiv \mathsf{Set}(\mathsf{Cycle}(Z_1, \mathsf{Cycle}(Z_3, Z_5)), \mathsf{Set}(\mathsf{Cycle}(Z_2, Z_4), \mathsf{Cycle}(Z_6))) \ .
\end{aligned}$$

We recall now a few of the orders discussed in Chapter 2 and how are they used to solve the unranking problem. For unions, $\mathcal{A} + \mathcal{B}$, the $\mathcal{A}$-objects come first and afterwards the $\mathcal{B}$-objects. For partitional and boxed products, we will consider the lexicographic and boustrophedonic orders to choose the size of the first component and, afterwards, we consider the PF and SF orders.

Sequences are based on the isomorphism (2.6), $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \star \mathsf{Seq}(\mathcal{A})$. Sets and cycles follow the isomorphisms (2.11) and (2.12), $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}\,{}^{\square}\!\star\, \mathsf{Set}(\mathcal{A})$ and $\mathsf{Cycle}(\mathcal{A}) =$

[1] Actually, α and β in $\langle\alpha, \beta, \ell\rangle$ are representations of objects, but we use the same letters to denote the objects themselves.

$\mathcal{A}\,^{\square}\!\star\, \mathsf{Seq}(\mathcal{A})$, respectively. Similar ideas follow for sequences, sets and cycles with restricted number of components (see Table 2.4). Finally, substitutions are based on the labelled unions, the Cartesian products and the labelled sets of fixed cardinality as defined by (1.2): $\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^{\{k\}}$. We note the objects that come from the substitution are based in an unlabelled product, so it does not generate any new relabelling and, in general, such objects will be noted by $\circ\langle\alpha, \beta\rangle$, where α and β belong to $\mathcal{A}_k$ and $\mathcal{B}^{\{k\}}$, respectively. Also, we recall that $|\circ\langle\alpha, \beta\rangle| = |\beta|$.

The rest of this chapter is organized as follows. In Section 3.1 we describe the algorithms which we propose for the unranking problem. The following section is devoted to the analysis of the performance of the unranking algorithms. After a general treatment of the subject and a few basic results on the worst-case complexity of unranking, we provide the so-called Υ- and $\hat{\Upsilon}$-algebras for the average and probability distribution of the cost of unranking labelled classes with lexicographic ordering. Examples of its use are given in Section 3.3. In particular, we consider the average cost and variance of unranking simply generated families of (non-)ordered trees. Finally, section 3.4 explores the use of alternative isomorphisms and the effects on the performance of unranking; for instance, we consider the *big-endian heuristic* and we introduce the *boxed heuristic*.

The main results of this chapter appear in references [54, 56].

3.1 The Algorithms

3.1.1 Basic Algorithms: $+$, $\star$ and $^{\square}\!\star$

Given the trivial algorithms that solve the unranking problem for the ϵ classes and for atomic classes, and after a preprocessing step that converts the initially given specification to standard form, it is clear that, for labelled classes, we only need to solve the unranking problem for unions and partitional and boxed products. Moreover, we assume that there exists a function $Count(\mathcal{A}, n)$ which returns the number of objects of size n in the combinatorial class $\mathcal{A}$ [26].

For unions, the object of rank i in $(\mathcal{A} + \mathcal{B})_n$ is the object of rank i in $\mathcal{A}_n$ if $0 \le i < a_n$; otherwise, it is the object of rank $i - a_n$ in $\mathcal{B}_n$ (Algorithm 1).

Algorithm 1 Unranking of disjoint unions

Unrank($\mathcal{A} + \mathcal{B}, n, i$)
  $i' := i -$ Count($\mathcal{A}, n$)
  if $i' < 0$ **then** Unrank($\mathcal{A}, n, i$) **else** Unrank($\mathcal{B}, n, i'$) **fi**

To solve the unranking for partitional products $\mathcal{A} \star \mathcal{B}$, assuming the lexicographic-PF order, the first step is to determine the size, say j, of the first component of the object; afterwards, we unrank the $\mathcal{A}$- and $\mathcal{B}$-objects and, finally, the rank of the corresponding partition (Algorithm 2). That is, if $\gamma = \langle\alpha, \beta, \ell_\gamma\rangle$ is the object of rank i in $(\mathcal{A} \star \mathcal{B})_n$ then $j = |\alpha|$ satisfies

$$\sum_{0 \le k < j} \binom{n}{k} \cdot a_k \cdot b_{n-k} \le i < \sum_{0 \le k \le j} \binom{n}{k} \cdot a_k \cdot b_{n-k}$$

Algorithm 2 Unranking of products (lexicographic-PF order)

1. Unrank$(\mathcal{A} \star \mathcal{B}, n, i)$
2. $\quad c := 0; j := 0; d := \text{Count}(\mathcal{A}, j) \cdot \text{Count}(\mathcal{B}, n-j)$
3. $\quad$ **while** $c + d \leq i$ **do**
4. $\qquad c := c + d; j := j + 1; d := \binom{n}{j} \cdot \text{Count}(\mathcal{A}, j) \cdot \text{Count}(\mathcal{B}, n-j)$
5. $\quad$ **end**
6. $\quad i' := i - c; b := \binom{n}{j} \cdot \text{Count}(\mathcal{B}, n-j)$
7. $\quad \alpha := \text{Unrank}(\mathcal{A}, j, i' \mathbin{\mathbf{div}} b); \beta := \text{Unrank}(\mathcal{B}, n-j, (i' \mathbin{\mathbf{mod}} b) \mathbin{\mathbf{div}} \binom{n}{j})$
8. $\quad \ell := (i' \mathbin{\mathbf{mod}} b) \mathbin{\mathbf{mod}} \binom{n}{j}$
9. $\quad$ **return** $\langle \alpha, \beta, \ell \rangle$

and γ is the object of rank $i' = i - \sum_{0 \leq k < j} \binom{n}{k} \cdot a_k \cdot b_{n-k}$ in $\mathcal{A}_j \star \mathcal{B}_{n-j}$. Each object in $\mathcal{A}_j$ is used in the construction of $b = \binom{n}{j} \cdot b_{n-j}$ different objects of $\mathcal{A}_j \star \mathcal{B}_{n-j}$. Hence, if $\gamma = \langle \alpha, \beta, \ell \rangle$ is the object of rank i' in $\mathcal{A}_j \star \mathcal{B}_{n-j}$ then α is the object of rank $i' \mathbin{\mathbf{div}} b$ in $\mathcal{A}_j$. Similarly, β is the object of rank $(i' \mathbin{\mathbf{mod}} b) \mathbin{\mathbf{div}} \binom{n}{j}$ in $\mathcal{B}_{n-j}$, and the label ℓ is the $\left((i' \mathbin{\mathbf{mod}} b) \mathbin{\mathbf{mod}} \binom{n}{j}\right)$-th partition among the $\binom{n}{j}$ different labellings corresponding to $\alpha \star \beta$.

If the boustrophedonic-PF order is used instead, the unranking of products is similar, the most significant difference being the way in which the size of the first component is determined (Algorithm 3). The size of the first component goes from the *smallest not-used-size* to the *largest not-used-size*, and so on.

Algorithm 3 Unranking of products (boustrophedonic-PF order)

1. Unrank$(\mathcal{A} \star \mathcal{B}, n, i)$
2. $\quad c := 0; k := 0; j := 0; d := \text{Count}(\mathcal{A}, j) \cdot \text{Count}(\mathcal{B}, n-j)$
3. $\quad$ **while** $c + d \leq i$ **do**
4. $\qquad c := c + d; k := k + 1; j := k \mathbin{\mathbf{div}} 2$
5. $\qquad$ **if** even(k) **then** $d := \binom{n}{j} \cdot \text{Count}(\mathcal{A}, j) \cdot \text{Count}(\mathcal{B}, n-j)$
6. $\qquad$ **else** $d := \binom{n}{j} \cdot \text{Count}(\mathcal{A}, n-j) \cdot \text{Count}(\mathcal{B}, j)$
7. $\qquad$ **fi**
8. $\quad$ **end**
9. $\quad$ **if** odd(k) **then** $j := n - j$ **fi**
10. $\quad \triangleright$ Lines (6) to (9) of Algorithm 2 go here

In a similar way, it is also possible to consider the lexicographic and boustrophedonic-SF orders, where we change first the "internal" structures and, afterwards, the labelling. Once the size j of the first component and the rank i' of the object within $\mathcal{A}_j \star \mathcal{B}_{n-j}$ have been computed, let $c = a_j \cdot b_{n-j}$, then ℓ is the $(i' \mathbin{\mathbf{div}} c)$-th (n, j)-partition, α is the object of rank $(i' \mathbin{\mathbf{mod}} c) \mathbin{\mathbf{div}} b_{n-j}$ in $\mathcal{A}_j$, and β is the object of rank $(i' \mathbin{\mathbf{mod}} c) \mathbin{\mathbf{mod}} b_{n-j}$ in $\mathcal{B}_{n-j}$. That is, the first loop of Algorithms 2 and 3 are equal in PF and SF order, but not the last part as we reflect in Algorithm 4.

Last but not least, the unranking of boxed products is analogous to that products, with

Algorithm 4 Unranking of products (SF order)

Unrank($\mathcal{A} \star \mathcal{B}, n, i$)
 ▷ Lines (2) to (5) of Algorithm 2 for lexicographic order or
 ▷ lines (2) to (9) of Algorithm 3 for boustrophedonic order
 $i' := i - c; b := \mathrm{Count}(\mathcal{B}, n-j);\ c := \mathrm{Count}(\mathcal{A}, j) \cdot b$
 $\ell := i'\ \mathbf{div}\ c$
 $\alpha := \mathrm{Unrank}(\mathcal{A}, j, (i'\ \mathbf{mod}\ c)\ \mathbf{div}\ b); \beta := \mathrm{Unrank}(\mathcal{B}, n-j, (i'\ \mathbf{mod}\ c)\ \mathbf{mod}\ b)$
 return $\langle \alpha, \beta, \ell \rangle$

the provision that the boxed product $\alpha\ {}^{\square}\!\star\ \beta$ of every pair of objects contains only $\binom{n-1}{j-1}$ labelled objects, being $n = |\alpha| + |\beta|$ and $j = |\alpha|$. That means that it is enough to substitute the binomial coefficients $\binom{n}{j}$ by $\binom{n-1}{j-1}$ and to start with $j := 1$ in the algorithm for the unranking of products. Also, to unrank $\mathcal{A} \star^{\square} \mathcal{B}$ (Section 2.2) it suffices to use $\binom{n-1}{j}$ instead of $\binom{n}{j}$.

3.1.2 Algorithms for Sequences, Sets and Cycles with Restricted Cardinality, and Substitutions

The variants of sequences, sets and cycles with restricted number of components are based in unions and partitional and boxed products (Table 2.4) and they are quite obvious. For instance, Algorithm 5 shows the unranking of sets with restricted number of components which is an immediate consequence of the isomorphisms in Table 2.4.

A different order results if we consider the card-lexicographic order (Subsection 2.4.2). The unranking of sets in such order is presented in Algorithm 6, while the unranking of sequences and cycles is similar.

Algorithm 7 for substitution is based upon analogous ideas; basically, once the size k of the $\mathcal{A}$-object has been found, it suffices to unrank the appropriate $\mathcal{A}$-object and the set of k $\mathcal{B}$-objects of total size n.

Algorithm 5 Unranking of sets with restricted cardinalities

Unrank($\mathcal{A}^{\{k\}}, n, i$)
 if $n = 0$ **then return** ϵ **else return** Unrank($\mathcal{A}\ {}^{\square}\!\star\ \mathcal{A}^{\{k-1\}}, n, i$) **fi**

Unrank($\mathcal{A}^{\{\leq k\}}, n, i$)
 if $n = 0$ **then return** ϵ **else return** Unrank($\mathcal{A}\ {}^{\square}\!\star\ \mathcal{A}^{\{\leq k-1\}}, n, i$) **fi**

Unrank($\mathcal{A}^{\{\geq k\}}, n, i$)
 if $n = 0$ **then return** ϵ **else return** Unrank($\mathcal{A}\ {}^{\square}\!\star\ \mathcal{A}^{\{\geq k-1\}}, n, i$) **fi**

Algorithm 6 Unranking of sets (card-lexicographic order)

Unrank$(\mathsf{Set}(\mathcal{A}), n, i)$
 $c := 0; k := 0; d := \mathrm{Count}(\mathcal{A}^{\{k\}}, n)$
 while $c + d \leq i$ **do** $c := c + d; k := k + 1; d := \mathrm{Count}(\mathcal{A}^{\{k\}}, n)$ **end**
 return Unrank$(\mathcal{A}^{\{k\}}, n, i - c)$

Algorithm 7 Unranking of substitutions

Unrank$(\mathcal{A} \circ \mathcal{B}, n, i)$
 $c := 0; k := 0; d := \mathrm{Count}(\mathcal{A}, k) \cdot \mathrm{Count}(\mathcal{B}^{\{k\}}, n)$
 while $c + d \leq i$ **do** $c := c + d; k := k + 1; d := \mathrm{Count}(\mathcal{A}, k) \cdot \mathrm{Count}(\mathcal{B}^{\{k\}}, n)$ **end**
 $i' := i - c; b := \mathrm{Count}(\mathcal{B}^{\{k\}}, n)$
 $\alpha := \mathrm{Unrank}(\mathcal{A}, k, i'$ **div** $b); \beta := \mathrm{Unrank}(\mathcal{B}^{\{k\}}, n, i'$ **mod** $b)$
 return $\circ\langle \alpha, \beta \rangle$

Similar techniques can also be used to solve the unrank of other operators. In particular, the unranking of *pointing* is presented in Section 3.2.2.

3.2 Analysis of the Cost

This section is divided in five subsections. In the first one we establish the main results of the average cost of the labelled unranking algorithms. In the second one we study the cumulated cost (the Υ-rules). In the next two subsections we extend the Υ-rules to higher-order moments ($\hat{\Upsilon}$-rules). Finally, in the last subsection, we use the Υ-rules to analyze the average cost for simply generated families of trees and other particular examples.

3.2.1 Performance of the Unranking Algorithms

In order to investigate the complexity of the unranking algorithms, the measure of cost that we will consider is the number of arithmetic operations. We assume that the cardinalities of the classes, as given by the function *Count*, are stored in precomputed tables of size $\mathrm{O}(n)$ integers[2] and we will not take into account in the analysis the time needed by this preprocessing, which is $\mathrm{O}(n^2)$.

The conversion of a *non-standard specification* into the corresponding *standard specification* is independent of n and effected only once, so this contribution will be also neglected. For instance, non-standard specifications for Cayley trees and Motzkin trees are $\{\mathcal{T} = \mathsf{Prod}(Z, \mathsf{Set}(\mathcal{T}))\}$ and $\{\mathcal{M} = \mathsf{Union}(Z, \mathsf{Union}(\mathsf{Prod}(\mathcal{M}, \mathcal{M}), \mathsf{Prod}(\mathcal{M}, \mathsf{Prod}(\mathcal{M}, \mathcal{M}))))\}$, respectively; while the corresponding standard specifications are $\{\mathcal{T} = \mathsf{Prod}(Z, \mathcal{S}),\ \mathcal{S} = \mathsf{Set}(\mathcal{T})\}$ and $\{\mathcal{M} = \mathsf{Union}(Z, \mathcal{M}23),\ \mathcal{M}23 = \mathsf{Union}(\mathcal{M}2, \mathcal{M}3),\ \mathcal{M}2 = \mathsf{Prod}(\mathcal{M}, \mathcal{M}),\ \mathcal{M}3 = \mathsf{Prod}(\mathcal{M}, \mathcal{M}2)\}$, respectively.

Furthermore, the unrank returns the objects as 3-tuples and we do not consider here the corresponding reconstruction of the labels.

[2] Since the stored integers are usually huge, the space requirement in bytes would typically be $\mathrm{O}(n^2 \log n)$.

Thus the main contribution to the cost of the unranking algorithms comes from the loops for the determination of the size of the first component in the case of products and boxed products, and their recursive nature. The next two theorems state the worst-case complexity of unranking objects of size n, for lexicographic and boustrophedonic orders, respectively. They show that the unranking algorithms have the same performance as the algorithms for random generation, and not surprisingly, the proof of the theorems given here and their counterparts for random generation is identical [26].

Theorem 9. *The worst-case time complexity of unranking for objects of size n in any admissible labelled class $\mathcal{A}$ using lexicographic ordering is $\mathrm{O}(n^2)$ arithmetic operations.*

Proof. Consider the parse tree associated to the unranking of an object of size n. Such a tree has size proportional to n and its arity is ≤ 2, with each of its nodes associated to a recursive call to the unranking procedure. The non-recursive cost associated to a node is at most linear in the size of the object produced as the result of that call, i.e., linear in the size of the subtree rooted at that node. Hence, the worst-case cost of unranking an object of size n is proportional to the worst-case path length of a tree of size $\mathrm{O}(n)$, which is $\mathrm{O}(n^2)$. □

Theorem 10. *The worst-case time complexity of unranking for objects of size n in any admissible labelled class $\mathcal{A}$ using boustrophedonic ordering is $\mathrm{O}(n \log n)$ arithmetic operations.*

Proof. The number of iterations to determine the size of α in a boustrophedonic product (α, β) is $2 \cdot (\min(|\alpha|, |\beta|) + 1)$. The same holds true for boxed products. The arithmetic operations outside the loop that determines the size j of the first component in products and boxed products will contribute a linear term in the total cost by an argument largely equivalent to that given in the proof of the previous theorem. Thus the main contribution in the worst-case cost of unranking satisfies a recurrence of the form

$$U(n) = \max_k \left(U(k) + U(n-k) + c \cdot \min(k, n-k)\right) ,$$

for some constant c, whose solution is $\mathrm{O}(n \log n)$ [33, 48]. □

The last result concerns *iterative classes.* A class is called iterative if the dependency graph of its specification is acyclic (not in standard form, thus allowing the use of sequences, sets, cycles, their restrictions and substitutions). In contrast, classes which are not iterative are called *recursive.* Typical examples of iterative classes include surjections $(\mathsf{Seq}(\mathsf{Set}(Z, \mathrm{card} \geq 1)))$ and permutations.

Theorem 11. *The cost of unranking an object of size n for any iterative labelled class $\mathcal{A}$ is $\Theta(n)$.*

Proof. It is obvious that the cost of unranking objects of size n in any class is $\Omega(n)$. So it is only necessary to establish that the worst-case complexity of unranking for iterative classes is $\mathrm{O}(n)$; this is easily accomplished by structural induction. For example, for products there is a linear cost to find the size of the first component plus the linear costs (by the inductive hypothesis) to unrank both components of the pair; altogether, the cost is bounded by a linear function. For $\mathcal{C} = \mathsf{Set}(\mathcal{A})$, it is necessary to compute the cost of unranking the boxed product $\mathcal{A}^{\square}\star\mathcal{C}$; by induction the cost of unranking in $\mathcal{A}$ is linear. Furthermore, the cost of unranking the smaller set in $\mathcal{C}$ is also linear because the sum of the costs to unrank in turn each component of the set (belonging to $\mathcal{A}$) is bounded, by induction, by a constant times the sum of the sizes, that is, the size of the set, and the result follows. □

Notice, however, that if the alternative isomorphism given by $\mathcal{C} = \epsilon + \mathcal{C}^{\square} \star \mathcal{A}$ is used then the inductive hypothesis would no longer hold since the depth of recursion could be linear and then the sum of the costs to determine the size of the first component in a recursive fashion yields a worst-case cost in $\Theta(n^2)$. Similarly, the inductive hypothesis also fails when the specification is recursive, like in $\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$.

The proof for the other constructions (sequences, cycles) works in a similar way because the isomorphisms use just unions, products and boxed products (see again Table 2.4).

3.2.2 A Calculus for the Average Cost of Unranking

The simplicity of the lexicographic unranking procedures allows for a detailed analysis of the average cost and higher order moments. Following the spirit of the cost algebra for random generation introduced by Flajolet *et al.* [26], the cumulated costs are defined by

$$\Upsilon\mathcal{A}_n = \sum_{\alpha \in \mathcal{A}_n} \mathrm{cu}(\alpha, \mathcal{A}) \ ,$$

where $\mathrm{cu}(\alpha, \mathcal{A})$ denotes the cost of $\mathit{Unrank}(\mathcal{A}, n, i)$ with α being the object of rank i in $\mathcal{A}_n$. The exponential generating function of the cumulated costs $\Upsilon\mathcal{A}_n$ is then

$$\Upsilon\mathcal{A}(z) = \sum_{n \geq 0} \Upsilon\mathcal{A}_n \frac{z^n}{n!} = \sum_{\alpha \in \mathcal{A}} \mathrm{cu}(\alpha, \mathcal{A}) \frac{z^{|\alpha|}}{|\alpha|!} \ , \tag{3.1}$$

and the average cost $\mu_{n,\mathcal{A}}$ of unranking objects in $\mathcal{A}_n$ is given by

$$\mu_{n,\mathcal{A}} = \frac{[z^n]\Upsilon\mathcal{A}(z)}{[z^n]A(z)} = \frac{\Upsilon\mathcal{A}_n}{a_n} \ .$$

These average costs will not include the preprocessing time needed to compute the tables of counts, nor the time to reconstruct the labels. The number of arithmetic operations done in the non-recursive part of the unranking algorithm for products or boxed products with lexicographic ordering is $c_1 \cdot j + c_2$, where j is the size of the first component of the unranked pair and c_1, c_2 are suitable constants. Hence, if the cumulated costs $\Upsilon\mathcal{A}_n$ only count the number of iterations made to determine the sizes of components, then the actual average complexity of unranking will be $\Theta(\mu_{n,\mathcal{A}})$.

Under the hypothesis above, the operator Υ behaves exactly like the operator Γ introduced in [26] to analyze the average complexity of random generation and obeys the rules given below here (called Υ-rules).

Theorem 12. *Let $\mathcal{A}$ and $\mathcal{B}$ be two labelled combinatorial classes, then*[3]

1. $\Upsilon\epsilon = \Upsilon Z = 0$.

[3] If there is no confusion, generating functions will omit its variable; for instance, F will be used instead of $F(z)$.

2. $\Upsilon(\mathcal{A}+\mathcal{B}) = \Upsilon\mathcal{A} + \Upsilon\mathcal{B}$.

3. $\Upsilon(\mathcal{A}\star\mathcal{B}) = \vartheta A\cdot B + \Upsilon\mathcal{A}\cdot B + A\cdot\Upsilon\mathcal{B}$.

4. $\vartheta\Upsilon(\mathcal{A}\,{}^{\square}\!\star\mathcal{B}) = \vartheta^2 A\cdot B + \vartheta\Upsilon\mathcal{A}\cdot B + \vartheta A\cdot\Upsilon\mathcal{B}$.

Proof. The first two statements are immediate from the definition of Υ and the corresponding (trivial) unranking algorithms.

For products, the cost of unranking the object $\gamma = (\alpha,\beta)$ in $\mathcal{A}\star\mathcal{B}$ is the cost of determining the size of α plus the cost of unranking α plus the cost of unranking β; hence,

$$\begin{aligned}\Upsilon(\mathcal{A}\star\mathcal{B}) &= \sum_{\gamma\in\mathcal{A}\star\mathcal{B}} \frac{z^{|\gamma|}}{|\gamma|!}\,\mathrm{cu}(\gamma,\mathcal{A}\star\mathcal{B})\\ &= \sum_{\alpha\in\mathcal{A}}\sum_{\beta\in\mathcal{B}} \binom{|\alpha|+|\beta|}{|\alpha|}\frac{z^{|\alpha|+|\beta|}}{(|\alpha|+|\beta|)!}\,(|\alpha| + \mathrm{cu}(\alpha,\mathcal{A}) + \mathrm{cu}(\beta,\mathcal{B}))\ ,\end{aligned}$$

and the statement given in the theorem follows after a few easy manipulations.

In case of boxed products we have

$$\Upsilon(\mathcal{A}\,{}^{\square}\!\star\mathcal{B}) = \sum_{\alpha\in\mathcal{A}}\sum_{\beta\in\mathcal{B}} \binom{|\alpha|+|\beta|-1}{|\alpha|-1}\frac{z^{|\alpha|+|\beta|}}{(|\alpha|+|\beta|)!}\,(|\alpha| + \mathrm{cu}(\alpha,\mathcal{A}) + \mathrm{cu}(\beta,\mathcal{B}))\ .$$

Applying ϑ to both sides and since $n\binom{n-1}{j-1} = j\binom{n}{j}$, the given rule follows

$$\begin{aligned}\vartheta\Upsilon(\mathcal{A}\,{}^{\square}\!\star\mathcal{B}) &= \sum_{\alpha\in\mathcal{A}}\sum_{\beta\in\mathcal{B}} |\alpha|\binom{|\alpha|+|\beta|}{|\alpha|}\frac{z^{|\alpha|+|\beta|}}{(|\alpha|+|\beta|)!}\,(|\alpha| + \mathrm{cu}(\alpha,\mathcal{A}) + \mathrm{cu}(\beta,\mathcal{B}))\\ &= \vartheta^2 A\cdot B + \vartheta\Upsilon\mathcal{A}\cdot B + \vartheta A\cdot\Upsilon\mathcal{B}\ .\end{aligned}$$

The last rule above proves the identity $\Upsilon\vartheta \equiv \vartheta\Upsilon$ (the operator ϑ in the left hand side is the marking operator for combinatorial classes; the operator ϑ in the right hand side acts on generating functions, $\vartheta \equiv z\frac{d}{dz}$), a result which can be obtained by reasoning by structural induction. Indeed, if $\mathcal{C} = \mathcal{A}\,{}^{\square}\!\star\mathcal{B}$ then $\vartheta\mathcal{C} = \vartheta\mathcal{A}\star\mathcal{B}$; hence,

$$\begin{aligned}\Upsilon\vartheta\mathcal{C} &= \Upsilon(\vartheta\mathcal{A}\star\mathcal{B}) = \vartheta^2 A\cdot B + \Upsilon\vartheta\mathcal{A}\cdot B + \vartheta A\cdot\Upsilon\mathcal{B}\\ &= \vartheta^2 A\cdot B + \vartheta\Upsilon\mathcal{A}\cdot B + \vartheta A\cdot\Upsilon\mathcal{B} = \vartheta\Upsilon(\mathcal{A}\,{}^{\square}\!\star\mathcal{B}) = \vartheta\Upsilon\mathcal{C}\ .\end{aligned}$$

A direct argument for the "commutativity" rule follows also from the straightforward complementary algorithms to unrank $\vartheta\mathcal{A}$ given an unranking algorithm for $\mathcal{A}$ and conversely, to unrank $\mathcal{A}$ given an unranking algorithm for $\vartheta\mathcal{A}$ (Algorithm 8). We use the notation (α,β) for a marked object where α is the considered $\mathcal{A}$-object and m means that the m-th atom is marked. The unranking algorithm for the class $\vartheta\mathcal{A}$ is straightforward using the unranking algorithm for the class $\mathcal{A}$: compute $i' = i\ \mathbf{div}\ n$, generate the i'-th object in $\mathcal{A}$, and mark the $((i\ \mathbf{mod}\ n)+1)$-th atom of the object. Therefore, $\Upsilon\vartheta\mathcal{A}_n = n\cdot\Upsilon\mathcal{A}_n$ and $\Upsilon\vartheta\mathcal{A} = \vartheta\Upsilon\mathcal{A}$.

Algorithm 8 Unranking $\vartheta\mathcal{A} \leftrightarrow \mathcal{A}$

Unrank$(\vartheta\mathcal{A}, n, i)$ $\quad \alpha :=$ Unrank$(\mathcal{A}, n, i$ **div** $n)$ $\quad$ **return** $(\alpha, (i$ **mod** $n) + 1)$	Unrank$(\mathcal{A}, n, i)$ $\quad (\alpha, m) :=$ Unrank$(\vartheta\mathcal{A}, n, i \cdot n)$ $\quad$ **return** α

But, as we have seen in Section 2.2, they cannot be used as a replacement of the algorithm for boxed products. □

From Theorem 12 and the isomorphisms of Table 2.5, it is possible to obtain rules for sequences, sets and cycles, their restricted-cardinality variants (this subsection shows here just the rules for sequences, sets and cycles of exactly k components, the other are presented in Subsection 3.2.4 for completeness), and for substitutions.

Corollary 3. *Let $\mathcal{A}$ be a combinatorial class such that $\epsilon \notin \mathcal{A}$ (equivalently, $A(0) = 0$). Then*

1. $\Upsilon \mathsf{Seq}(\mathcal{A}) = \frac{\vartheta A + \Upsilon\mathcal{A}}{(1-A)^2}$.
2. $\Upsilon \mathsf{Set}(\mathcal{A}) = \exp(A)\,(\vartheta A + \Upsilon\mathcal{A})$.
3. $\Upsilon \mathsf{Cycle}(\mathcal{A}) = \frac{\vartheta A + \Upsilon\mathcal{A}}{1-A}$.
4. $\Upsilon \mathcal{A}^k = k\,A^{k-1}\,(\vartheta A + \Upsilon\mathcal{A})$, *for* $k > 0$.
5. $\Upsilon \mathcal{A}^{\{k\}} = \frac{A^{k-1}}{(k-1)!}\,(\vartheta A + \Upsilon\mathcal{A})$, *for* $k > 0$.
6. $\Upsilon \mathcal{A}^{\langle k\rangle} = A^{k-1}\,(\vartheta A + \Upsilon\mathcal{A})$, *for* $k > 1$.
7. $\Upsilon(\mathcal{A} \circ \mathcal{B}) = (\vartheta A + \Upsilon\mathcal{A})(B) + (\vartheta B + \Upsilon\mathcal{B}) \cdot A'(B)$.

Proof. The rule for sequences follows by easy algebraic manipulation and the well-known fact that the generating function of $\mathsf{Seq}(\mathcal{A})$ is $1/(1 - A(z))$:

$$\Upsilon\mathsf{Seq}(\mathcal{A}) = \Upsilon(\epsilon + \mathcal{A} \star \mathsf{Seq}(\mathcal{A})) = \Upsilon\epsilon + \vartheta A\,\frac{1}{1-A} + \Upsilon\mathcal{A}\,\frac{1}{1-A} + A \cdot \Upsilon\mathsf{Seq}(\mathcal{A})$$
$$\Rightarrow \Upsilon\mathsf{Seq}(\mathcal{A}) = \frac{\vartheta A + \Upsilon\mathcal{A}}{(1-A)^2}\ .$$

The rule for sets is the solution of a linear first-order differential equation that we obtain applying the rule for boxed products and since the generating function of sets is $\exp(A)$:

$$\begin{aligned}\vartheta\Upsilon\mathsf{Set}(\mathcal{A}) &= \Upsilon\vartheta\mathsf{Set}(\mathcal{A}) = \Upsilon(\vartheta\mathcal{A} \star \mathsf{Set}(\mathcal{A}))\\ &= \vartheta^2 A \cdot \exp(A) + \Upsilon\vartheta\mathcal{A} \cdot \exp(A) + \vartheta A \cdot \Upsilon\mathsf{Set}(\mathcal{A})\\ &= \vartheta^2 A \cdot \exp(A) + \vartheta\Upsilon\mathcal{A} \cdot \exp(A) + \vartheta A \cdot \Upsilon\mathsf{Set}(\mathcal{A})\\ &\Rightarrow \Upsilon\mathsf{Set}(\mathcal{A}) = \exp(A)\,(\vartheta A + \Upsilon\mathcal{A})\ .\end{aligned}$$

Notice that this rule differs from that given in [26] as they use the isomorphism $\vartheta\mathsf{Set}(\mathcal{A}) = \mathsf{Set}(\mathcal{A}) \star \vartheta\mathcal{A}$.

The rule for cycles is derived by immediate integration of the equation $\mathsf{Cycle}(\mathcal{A}) = \mathcal{A}\ {}^{\square}\!\star \mathsf{Seq}(\mathcal{A})$.

The rule for sequences of k components is also straightforward from its definition:

$$\begin{aligned}\Upsilon\mathcal{A}^k &= \Upsilon(\mathcal{A}\star\mathcal{A}^{k-1}) = \vartheta A\cdot A^{k-1} + \Upsilon\mathcal{A}\cdot A^{k-1} + A\cdot\Upsilon\mathcal{A}^{k-1} = \ldots\\ &= (k-1)\,(\vartheta A + \Upsilon\mathcal{A})\cdot A^{k-1} + A^{k-1}\cdot\Upsilon\mathcal{A}^1\\ &= (k-1)\,(\vartheta A + \Upsilon\mathcal{A})\cdot A^{k-1} + A^{k-1}\cdot\Upsilon(\mathcal{A}\star\epsilon) = k\,A^{k-1}\,(\vartheta A + \Upsilon\mathcal{A})\ ,\end{aligned}$$

where $\mathcal{A}^1 = \mathcal{A}\star\mathcal{A}^0$ and $\mathcal{A}^0 = \epsilon$.

Sets of k components can be dealt with introducing the generating function

$$G(z,t) = \sum_{k>0}\Upsilon\mathcal{A}^{\{k\}}\cdot t^k\ .$$

Then, from the Υ-rules,

$$\vartheta\Upsilon\mathcal{A}^{\{k\}} = \Upsilon\vartheta\mathcal{A}^{\{k\}} = \Upsilon(\vartheta\mathcal{A}\star\mathcal{A}^{\{k-1\}}) = \vartheta^2 A\cdot A^{\{k-1\}} + \Upsilon\vartheta\mathcal{A}\cdot A^{\{k-1\}} + \vartheta A\cdot\Upsilon\mathcal{A}^{\{k-1\}}\ ,$$

so we have

$$\vartheta G(z,t) = \vartheta^2 A\cdot t\cdot\sum_{k>0}\frac{(At)^{k-1}}{(k-1)!} + \Upsilon\vartheta\mathcal{A}\cdot t\cdot\sum_{k>0}\frac{(At)^{k-1}}{(k-1)!} + \vartheta A\cdot t\cdot G(z,t)\ .$$

That is,

$$\vartheta G(z,t) - \vartheta A\cdot t\cdot G(z,t) = (\vartheta^2 A + \Upsilon\vartheta\mathcal{A})\cdot t\cdot\exp(At)\ .$$

Now, the variation-of-constant method yields

$$G(z,t)\ =\ \exp(At)\cdot t\cdot(\vartheta A + \Upsilon\mathcal{A})\ .$$

The result, $\Upsilon\mathcal{A}^{\{k\}} = \frac{A^{k-1}}{(k-1)!}\,(\vartheta A + \Upsilon\mathcal{A})$, follows by extracting the corresponding coefficients from the previous equation,

$$\sum_{k>0}\ \Upsilon\mathcal{A}^{\{k\}}\cdot t^k = \sum_{k>0}\ \frac{(At)^k}{k!}\cdot t\cdot(\vartheta A + \Upsilon\mathcal{A})\ .$$

Another way to get the Υ-rule for sets of k components consists in solving the recursion. Assuming that $\mathcal{A}^{\{1\}} = \mathcal{A}\ {}^{\square}\!\star\mathcal{A}^{\{0\}}$ and $\mathcal{A}^{\{0\}} = \epsilon$, we have

$$\begin{aligned}\vartheta\Upsilon\mathcal{A}^{\{k\}} &= \Upsilon\vartheta\mathcal{A}^{\{k\}} = \Upsilon(\vartheta\mathcal{A}\star\mathcal{A}^{\{k-1\}})\\ &= \vartheta^2 A\,\frac{A^{k-1}}{(k-1)!} + \Upsilon\vartheta\mathcal{A}\,\frac{A^{k-1}}{(k-1)!} + \vartheta A\,\Upsilon\mathcal{A}^{\{k-1\}}\\ &= \vartheta(\vartheta A + \Upsilon\mathcal{A})\,\frac{A^{k-1}}{(k-1)!} + \vartheta A\cdot\Upsilon\mathcal{A}^{\{k-1\}}\\ &\Rightarrow \Upsilon\mathcal{A}^{\{k\}} = \frac{A^{k-1}}{(k-1)!}\,(\vartheta A + \Upsilon\mathcal{A})\ .\end{aligned}$$

The rule for cycles of k components is derived by immediate integration of the equation used in the definition and the previous results:

$$\begin{aligned}
\vartheta\Upsilon\mathcal{A}^{\langle k\rangle} &= \Upsilon\vartheta\mathcal{A}^{\langle k\rangle} = \Upsilon(\vartheta\mathcal{A}\star\mathcal{A}^{k-1}) = \vartheta^2 A\cdot A^{k-1} + \Upsilon\vartheta\mathcal{A}\cdot A^{k-1} + \vartheta A\cdot\Upsilon\mathcal{A}^{k-1}\\
&= \vartheta^2 A\cdot A^{k-1} + \vartheta\Upsilon\mathcal{A}\cdot A^{k-1} + \vartheta A\cdot((k-1)\,A^{k-2}\,(\vartheta A + \Upsilon\mathcal{A}))\\
&\Rightarrow \Upsilon\mathcal{A}^{\langle k\rangle} = A^{k-1}\,(\vartheta A + \Upsilon\mathcal{A})\ .
\end{aligned}$$

The rule for substitution easily follows from the isomorphism (1.2):

$$\begin{aligned}
\Upsilon(\mathcal{A}\circ\mathcal{B}) &= \Upsilon\left(\bigcup_{k>0}\mathcal{A}_k\star\mathcal{B}^{\{k\}}\right) = \sum_{k>0}\vartheta A_k\,\frac{B^k}{k!} + \sum_{k>0}\Upsilon\mathcal{A}_k\,\frac{B^k}{k!} + \sum_{k>0} A_k\cdot\Upsilon\mathcal{B}^{\{k\}}\\
&= (\vartheta A + \Upsilon\mathcal{A})(B) + \sum_{k>0} A_k\,\frac{B^{k-1}}{(k-1)!}\,(\vartheta B + \Upsilon\mathcal{B})\\
&= (\vartheta A + \Upsilon\mathcal{A})(B) + (\vartheta B + \Upsilon\mathcal{B})\cdot A'(B)\ . \qquad\square
\end{aligned}$$

3.2.3 Probability Distribution and Higher-Order Moments

In this section we extend the Υ-calculus of the previous section to an operator $\hat{\Upsilon}$ for distributions. Formally, given a class $\mathcal{A}$, the $\hat{\Upsilon}$ operator applied to $\mathcal{A}$ yields the following bivariate generating function:

$$\hat{\Upsilon}\mathcal{A}(z,u) = \sum_{\alpha\in\mathcal{A}}\frac{z^{|\alpha|}}{|\alpha|!}\,u^{\mathrm{cu}(\alpha,\mathcal{A})}\ . \tag{3.2}$$

By definition, the coefficient of $z^n\,u^k$ is the number of objects in $\mathcal{A}_n$ such that the cost of unranking them is k, divided by $n!$. Furthermore, $\hat{\Upsilon}\mathcal{A}(z,1) = A(z)$. Hence (see Section 1.2)

$$\Pr\{\mathrm{cu}(\alpha,\mathcal{A}) = k \mid \alpha\in\mathcal{A}_n\} = \frac{[z^n u^k]\hat{\Upsilon}\mathcal{A}(z,u)}{[z^n]A(z)} = \frac{[z^n u^k]\hat{\Upsilon}\mathcal{A}(z,u)}{[z^n]\hat{\Upsilon}\mathcal{A}(z,1)}\ ,$$

and

$$p_n(u) = \frac{[z^n]\hat{\Upsilon}\mathcal{A}(z,u)}{[z^n]\hat{\Upsilon}\mathcal{A}(z,1)}$$

is the probability generating function of the random variable $Y_{n,\mathcal{A}} =$ "cost of unranking an object of size n in $\mathcal{A}$". Moreover, since the cost $\mathrm{cu}(\alpha,\mathcal{A})$ to unrank an object α is basically the same as the cost of randomly generating that object, it follows that the results below apply to the complexity of random generation of labelled classes as well.

Successive differentiation of $\hat{\Upsilon}\mathcal{A}$ with respect to u yields the factorial moments of the random variable $Y_{n,\mathcal{A}}$. Indeed, if the following definition is considered,

$$\Upsilon^{(r)}\mathcal{A}(z) = \left.\frac{\partial^r\,\hat{\Upsilon}\mathcal{A}(z,u)}{\partial u^r}\right|_{u=1}\ , \tag{3.3}$$

then

$$\mathbb{E}\left[Y_{n,\mathcal{A}}^{\underline{r}}\right] = \mathbb{E}[Y_{n,\mathcal{A}}(Y_{n,\mathcal{A}}-1)\cdots(Y_{n,\mathcal{A}}-r+1)] = \frac{[z^n]\Upsilon^{(r)}\mathcal{A}(z)}{[z^n]A(z)} = \frac{[z^n]\Upsilon^{(r)}\mathcal{A}(z)}{[z^n]\Upsilon^{(0)}\mathcal{A}(z)} \ .$$

Obviously, $\Upsilon^{(1)} \equiv \Upsilon$, where Υ is the operator introduced in Section 3.2.2. Ordinary and central moments can be easily recovered. For instance, if $\mu_{n,\mathcal{A}} = \mathbb{E}[Y_{n,\mathcal{A}}]$ then the variance is given by

$$\mathbb{V}[Y_{n,\mathcal{A}}] = \sigma_{n,\mathcal{A}}^2 = \frac{[z^n]\Upsilon^{(2)}\mathcal{A}(z)}{[z^n]\Upsilon^{(0)}\mathcal{A}(z)} + \mu_{n,\mathcal{A}} - \mu_{n,\mathcal{A}}^2 \ .$$

From now to the end, if there is no confusion and the opposite is not indicated, $\hat{\Upsilon}\mathcal{A}(z,u)$ will be noted by $\hat{\Upsilon}\mathcal{A}$.

The basic rules for computing $\hat{\Upsilon}\mathcal{A}$ are given in the next theorem.

Theorem 13. *Let $\mathcal{A}$ and $\mathcal{B}$ be two labelled combinatorial classes.*

1. $\hat{\Upsilon}\epsilon = 1$.
2. $\hat{\Upsilon}Z = z$.
3. $\hat{\Upsilon}(\mathcal{A}+\mathcal{B}) = \hat{\Upsilon}\mathcal{A} + \hat{\Upsilon}\mathcal{B}$.
4. $\hat{\Upsilon}(\mathcal{A}\star\mathcal{B}) = \hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{B}(z,u)$.
5. $\vartheta\hat{\Upsilon}(\mathcal{A}\,{}^{\square}{\star}\,\mathcal{B}) = \vartheta\hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{B}(z,u)$.

Here the operators ϑ and $\hat{\Upsilon}$ also have the "commutativity" rule, $\vartheta\hat{\Upsilon} \equiv \hat{\Upsilon}\vartheta$, by the same arguments discussed at the previous section for ϑ and Υ.

Proof. Statements (1) and (2) trivially follow from the definition of $\hat{\Upsilon}$. If $\gamma \in \mathcal{A}+\mathcal{B}$, $\mathrm{cu}(\gamma,\mathcal{A}+\mathcal{B}) = \mathrm{cu}(\gamma,\mathcal{A})$ if $\gamma \in \mathcal{A}$, and $\mathrm{cu}(\gamma,\mathcal{A}+\mathcal{B}) = \mathrm{cu}(\gamma,\mathcal{B})$ if $\gamma \in \mathcal{B}$. Rule (3) then easily follows.

For products we have

$$\begin{aligned}\hat{\Upsilon}(\mathcal{A}\star\mathcal{B}) &= \sum_{\gamma\in\mathcal{A}\star\mathcal{B}} \frac{z^{|\gamma|}}{|\gamma|!}\,u^{\mathrm{cu}(\gamma,\mathcal{A}\star\mathcal{B})} = \sum_{\alpha\in\mathcal{A}}\sum_{\beta\in\mathcal{B}} \binom{|\alpha|+|\beta|}{|\alpha|} \frac{z^{|\alpha|+|\beta|}}{(|\alpha|+|\beta|)!}\,u^{|\alpha|+\mathrm{cu}(\alpha,\mathcal{A})+\mathrm{cu}(\beta,\mathcal{B})} \\ &= \left(\sum_{\alpha\in\mathcal{A}} \frac{z^{|\alpha|}}{|\alpha|!}\,u^{|\alpha|+\mathrm{cu}(\alpha,\mathcal{A})}\right)\left(\sum_{\beta\in\mathcal{B}} \frac{z^{|\beta|}}{|\beta|!}\,u^{\mathrm{cu}(\beta,\mathcal{B})}\right) = \hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{B}(z,u) \ .\end{aligned}$$

Similarly, for boxed products,

$$\begin{aligned}\vartheta\hat{\Upsilon}(\mathcal{A}\,{}^{\square}{\star}\,\mathcal{B}) &= \sum_{\gamma\in\mathcal{A}^{\square}\star\mathcal{B}} \frac{z^{|\gamma|}}{(|\gamma|-1)!}\,u^{\mathrm{cu}(\gamma,\mathcal{A}^{\square}\star\mathcal{B})} \\ &= \sum_{\alpha\in\mathcal{A}}\sum_{\beta\in\mathcal{B}} \binom{|\alpha|+|\beta|-1}{|\alpha|-1} \frac{z^{|\alpha|+|\beta|}}{(|\alpha|+|\beta|-1)!}\,u^{|\alpha|+\mathrm{cu}(\alpha,\mathcal{A})+\mathrm{cu}(\beta,\mathcal{B})} \\ &= \left(\sum_{\alpha\in\mathcal{A}} \frac{z^{|\alpha|}}{(|\alpha|-1)!}\,u^{|\alpha|+\mathrm{cu}(\alpha,\mathcal{A})}\right)\left(\sum_{\beta\in\mathcal{B}} \frac{z^{|\beta|}}{|\beta|!}\,u^{\mathrm{cu}(\beta,\mathcal{B})}\right) = \vartheta\hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{B}(z,u) \ .\end{aligned}$$

□

Corollary 4. *Let $\mathcal{A}$ be a combinatorial class such that $\epsilon \notin \mathcal{A}$ (equivalently, $A(0) = 0$). Then*

1. $\hat{\Upsilon}\mathsf{Seq}(\mathcal{A}) = \frac{1}{1-\hat{\Upsilon}\mathcal{A}(zu,u)}$.
2. $\hat{\Upsilon}\mathsf{Set}(\mathcal{A}) = \exp\left(\hat{\Upsilon}\mathcal{A}(zu,u)\right)$.
3. $\hat{\Upsilon}\mathsf{Cycle}(\mathcal{A}) = \log\left(\frac{1}{1-\hat{\Upsilon}\mathcal{A}(zu,u)}\right)$.
4. $\hat{\Upsilon}\mathcal{A}^k = \left(\hat{\Upsilon}\mathcal{A}(zu,u)\right)^k$, *for* $k \geq 0$.
5. $\hat{\Upsilon}\mathcal{A}^{\{k\}} = \frac{\left(\hat{\Upsilon}\mathcal{A}(zu,u)\right)^k}{k!}$, *for* $k \geq 0$.
6. $\hat{\Upsilon}\mathcal{A}^{\langle k\rangle} = \frac{\left(\hat{\Upsilon}\mathcal{A}(zu,u)\right)^k}{k}$, *for* $k > 0$.
7. $\hat{\Upsilon}(\mathcal{A}\circ\mathcal{B}) = \hat{\Upsilon}\mathcal{A}(\hat{\Upsilon}\mathcal{B}(zu,u)\cdot u, u)$.

Proof. Since $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \star \mathsf{Seq}(\mathcal{A})$, it follows that

$$\hat{\Upsilon}\mathsf{Seq}(\mathcal{A}) = 1 + \hat{\Upsilon}(\mathcal{A}\star\mathsf{Seq}(\mathcal{A})) = 1 + \hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathsf{Seq}(\mathcal{A})$$
$$\Rightarrow \hat{\Upsilon}\mathsf{Seq}(\mathcal{A}) = \frac{1}{1-\hat{\Upsilon}\mathcal{A}(zu,u)} .$$

From the isomorphism $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}\,{}^{\square}\!\star\,\mathsf{Set}(\mathcal{A})$, the following equality results

$$\vartheta\hat{\Upsilon}\mathsf{Set}(\mathcal{A}) = \vartheta\hat{\Upsilon}(\mathcal{A}\,{}^{\square}\!\star\,\mathsf{Set}(\mathcal{A})) = \vartheta\hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathsf{Set}(\mathcal{A}) ,$$

and the rule follows by integration.

The rule for cycles follows from the observation that

$$\vartheta\hat{\Upsilon}\mathsf{Cycle}(\mathcal{A}) = \vartheta\hat{\Upsilon}\mathcal{A}(zu,u)\,\frac{1}{1-\hat{\Upsilon}\mathcal{A}(zu,u)} = \vartheta\left(\log\frac{1}{1-\hat{\Upsilon}\mathcal{A}(zu,u)}\right) .$$

The rule for sequences of fixed cardinality is easily established iterating the rule for products on the isomorphism $\mathcal{A}^k = \mathcal{A}\star\mathcal{A}^{k-1}$:

$$\hat{\Upsilon}\mathcal{A}^k(z,u) = \hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{A}^{k-1}(z,u) = \ldots = (\hat{\Upsilon}\mathcal{A}(zu,u))^k .$$

For sets of k components, let $\mathcal{A}^{\{k\}} = \mathsf{Set}(\mathcal{A}, \mathrm{card} = k)$ and introduce

$$S(z,u,t) = \sum_{k\geq 0}\hat{\Upsilon}\mathcal{A}^{\{k\}}(z,u)\cdot t^k .$$

From the rule for boxed products and the inductive definition

$$\mathcal{A}^{\{k\}} = \begin{cases}\mathcal{A}\,{}^{\square}\!\star\,\mathcal{A}^{\{k-1\}} & \text{if } k\geq 1,\\ \epsilon & \text{if } k = 0.\end{cases}$$

we obtain the differential equation

$$\vartheta S(z,u,t) = \vartheta \hat{\Upsilon}\mathcal{A}(zu,u) \cdot t \cdot S(z,u,t)$$

whose solution is $S(z,u,t) = \exp(t \cdot \hat{\Upsilon}\mathcal{A}(zu,u))$, and hence

$$\hat{\Upsilon}\mathcal{A}^{\{k\}} = [t^k]S(z,u,t) = \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^k}{k!} \ .$$

Since $\mathcal{A}^{\langle k\rangle} = \mathcal{A}\ {}^{\square}\!\star \mathcal{A}^{k-1}$, we have

$$\vartheta\hat{\Upsilon}\mathcal{A}^{\langle k\rangle} = \vartheta\hat{\Upsilon}\mathcal{A}(zu,u) \cdot \hat{\Upsilon}\mathcal{A}^{k-1}(z,u) = \vartheta\hat{\Upsilon}\mathcal{A}(zu,u) \cdot (\hat{\Upsilon}\mathcal{A}(z,u))^{k-1} \ ,$$

and the rule for cycles of k components follows by integration.

Last but not least, the substitution rule can be proved as follows:

$$\begin{aligned}
\hat{\Upsilon}(\mathcal{A}\circ\mathcal{B}) &= \sum_{\circ\langle\alpha,\beta\rangle\in\mathcal{A}\circ\mathcal{B}} \frac{z^{|\beta|}}{|\beta|!} u^{|\alpha|+\mathrm{cu}(\alpha,\mathcal{A})+\mathrm{cu}(\beta,\mathcal{B})} \\
&= \sum_{\alpha\in\mathcal{A}} u^{|\alpha|+\mathrm{cu}(\alpha,\mathcal{A})} \sum_{\beta\in\mathcal{B}^{\{|\alpha|\}}} \frac{z^{|\beta|}}{|\beta|!} u^{\mathrm{cu}(\beta,\mathcal{B})} = \sum_{\alpha\in\mathcal{A}} u^{|\alpha|}\, u^{\mathrm{cu}(\alpha,\mathcal{A})}\, \hat{\Upsilon}(\mathcal{B}^{\{|\alpha|\}}) \\
&= \sum_{\alpha\in\mathcal{A}} \frac{(\hat{\Upsilon}\mathcal{B}(zu,u)\cdot u)^{|\alpha|}}{|\alpha|!} u^{\mathrm{cu}(\alpha,\mathcal{A})} = \hat{\Upsilon}\mathcal{A}(\hat{\Upsilon}\mathcal{B}(zu,u)\cdot u, u) \ .
\end{aligned}$$

□

Using Leibniz differential formula [1] for products, the $\Upsilon^{(r)}$-rules given in the following theorem easily follow, including the rules given in Theorem 12. We will not give $\Upsilon^{(r)}$-rules for sequences, sets, etc. though.

Theorem 14. *Let $\mathcal{A}$ and $\mathcal{B}$ be two labelled combinatorial classes, then*

1. $\Upsilon^{(r)}\epsilon = \Upsilon^{(r)}Z = 0.$
2. $\Upsilon^{(r)}(\mathcal{A}+\mathcal{B}) = \Upsilon^{(r)}\mathcal{A} + \Upsilon^{(r)}\mathcal{B}.$
3. $\Upsilon^{(r)}(\mathcal{A}\star\mathcal{B}) = \sum_{j=0}^{r} \binom{r}{j}\ \Upsilon^{(j)}\mathcal{B} \sum_{k=0}^{r-j} \binom{r-j}{k}\ z^k\ \frac{\partial^k}{\partial z^k}\Upsilon^{(r-j-k)}\mathcal{A}.$
4. $\vartheta\Upsilon^{(r)}(\mathcal{A}\ {}^{\square}\!\star \mathcal{B}) = \sum_{j=0}^{r} \binom{r}{j}\ \Upsilon^{(j)}\mathcal{B} \sum_{k=0}^{r-j} \binom{r-j}{k}\ z^{k+1}\ \frac{\partial^{k+1}}{\partial z^{k+1}}\Upsilon^{(r-j-k)}\mathcal{A}.$

Proof. The first two items of ϵ, Z and $\mathcal{A}+\mathcal{B}$ stem from the $\Upsilon^{(r)}$-definition.

The identity of $\mathcal{A}\star\mathcal{B}$ results from Theorem 13,

$$\hat{\Upsilon}(\mathcal{A}\star\mathcal{B}) = \hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{B}(z,u) \ , \tag{3.4}$$

the Leibniz differential formula [1] for products,

$$\frac{\partial^n}{\partial x^n}(F(z,u)\cdot G(z,u)) \;=\; \sum_{j=0}^{n}\binom{n}{j}\frac{\partial^j}{\partial x^j}F(z,u)\,\frac{\partial^{n-j}}{\partial x^{n-j}}G(z,u)\ , \tag{3.5}$$

and the chain rule.

In order to complete the proof we prove by induction that

$$\frac{\partial^s}{\partial u^s}\hat{\Upsilon}\mathcal{A}(zu,u) = \sum_{k=0}^{s}\binom{s}{k}\ \frac{\partial^s}{\partial x^k\ \partial y^{s-k}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ z^k\ . \tag{3.6}$$

For $s=1$ the statement is obviously true. Assuming that (3.6) is true for s, then, for $s+1$, we have

$$\begin{aligned}
\frac{\partial^{s+1}}{\partial u^{s+1}}\hat{\Upsilon}\mathcal{A}(zu,u) &= \frac{\partial}{\partial u}\frac{\partial^s}{\partial u^s}\hat{\Upsilon}\mathcal{A}(zu,u) = \frac{\partial}{\partial u}\left(\sum_{k=0}^{s}\binom{s}{k}\ \frac{\partial^s}{\partial x^k\ \partial y^{s-k}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ z^k\right)\\
&= \sum_{k=0}^{s}\binom{s}{k}\ \frac{\partial^{s+1}}{\partial x^{k+1}\ \partial y^{s-k}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ z^{k+1}\ +\\
&\quad \sum_{k=0}^{s}\binom{s}{k}\ \frac{\partial^{s+1}}{\partial x^{k}\ \partial y^{s-k+1}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ z^{k}\\
&= \binom{s}{0}\ \frac{\partial^{s+1}}{\partial y^{s+1}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ +\\
&\quad \sum_{k=1}^{s}\left(\binom{s}{k}+\binom{s}{k-1}\right)\ \frac{\partial^{s+1}}{\partial x^{k}\ \partial y^{s-k+1}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}} z^{k}\ +\\
&\quad \binom{s}{s}\ \frac{\partial^{s+1}}{\partial x^{s+1}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ z^{s+1}\\
&= \sum_{k=0}^{s+1}\binom{s+1}{k}\ \frac{\partial^{s+1}}{\partial x^{k}\ \partial y^{s+1-k}}\hat{\Upsilon}\mathcal{A}(x,y)\Big|_{\substack{x=zu\\ y=u}}\ z^{k}\ .
\end{aligned}$$

Finally, from (3.4), (3.5) and (3.6) setting $u=1$ the formula results:

$$\Upsilon^{(r)}(\mathcal{A}\star\mathcal{B}) = \sum_{j=0}^{r}\ \binom{r}{j}\ \Upsilon^{(j)}\mathcal{B}\ \sum_{k=0}^{r-j}\ \binom{r-j}{k}\ z^k\ \frac{\partial^k}{\partial z^k}\Upsilon^{(r-j-k)}\mathcal{A}\ .$$

The rule for boxed products is similarly proven. □

3.2.4 Containers with Bounded Cardinality

This subsection collects the basic $\hat{\Upsilon}$-rules for sequences, sets and cycles with lower and upper bounded cardinality, that is, with restrictions of the form $< k$, $\leq k$, etc. The Υ- and $\Upsilon^{(r)}$-rules can be easily derived from these by differentiation.

Theorem 15. *Let $\mathcal{A}$ be a labelled combinatorial class such that $\epsilon \notin \mathcal{A}$. The corresponding $\hat{\Upsilon}$-rules for sequences, sets and cycles with restricted but not-fixed cardinality are:*

1. $\hat{\Upsilon}\mathcal{A}^{\leq k} = \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{k+1}-1}{\hat{\Upsilon}\mathcal{A}(zu,u)-1}$.

2. $\hat{\Upsilon}\mathcal{A}^{\geq k} = \frac{(\hat{\Upsilon}\mathcal{A}(zu,u)^{k}}{1-\hat{\Upsilon}\mathcal{A}(zu,u)}$.

3. $\hat{\Upsilon}\mathcal{A}^{\{\leq k\}} = \sum_{0\leq i\leq k} \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{i}}{i!}$.

4. $\hat{\Upsilon}\mathcal{A}^{\{\geq k\}} = \exp(\hat{\Upsilon}\mathcal{A}(zu,u)) - \sum_{0\leq i< k} \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{i}}{i!}$.

5. $\hat{\Upsilon}\mathcal{A}^{\langle\leq k\rangle} = \sum_{1\leq i\leq k} \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{i}}{i}$.

6. $\hat{\Upsilon}\mathcal{A}^{\langle\geq k\rangle} = \log\left(\frac{1}{1-\hat{\Upsilon}\mathcal{A}(zu,u)}\right) - \sum_{1\leq i< k} \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{i}}{i}$.

Proof. These results are straightforward from the isomorphisms given in Table 2.4 and the $\hat{\Upsilon}$-rules in Theorem 13. For instance,

$$\begin{aligned}\hat{\Upsilon}\mathcal{A}^{\leq k} &= 1 + \hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{A}^{\leq k-1} = 1 + \hat{\Upsilon}\mathcal{A}(zu,u)\cdot(1+\hat{\Upsilon}\mathcal{A}(zu,u)\cdot\hat{\Upsilon}\mathcal{A}^{\leq k-2}) = \dots\\ &= \sum_{i=0}^{k}(\hat{\Upsilon}\mathcal{A}(zu,u))^{i} = \frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{k}-1}{\hat{\Upsilon}\mathcal{A}(zu,u)-1}\ .\end{aligned}$$ □

Also, it is easy to prove the more general statements:

$$\begin{aligned}\hat{\Upsilon}\mathsf{Seq}(\mathcal{A},\text{card}\in\Omega) &= \sum_{k\in\Omega}(\hat{\Upsilon}\mathcal{A}(zu,u))^{k}\ ,\\ \hat{\Upsilon}\mathsf{Set}(\mathcal{A},\text{card}\in\Omega) &= \sum_{k\in\Omega}\frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{k}}{k!}\ ,\\ \hat{\Upsilon}\mathsf{Cycle}(\mathcal{A},\text{card}\in\Omega) &= \sum_{k\in\Omega}\frac{(\hat{\Upsilon}\mathcal{A}(zu,u))^{k}}{k}\ ,\end{aligned}$$

and the rules given in Theorem 15 are just particular cases of those above.

We note all the $\hat{\Upsilon}$-rules given in Corollary 4 and Theorem 15 have the form

$$\hat{\Upsilon}(\Psi(\mathcal{A})) = \Phi(\hat{\Upsilon}\mathcal{A}(z\cdot u,u))\ ,$$

where Ψ is an operator on combinatorial classes and Φ the corresponding operator for (counting) generating functions.

3.3 Examples

3.3.1 Simply Generated Families of Trees

Consider any simply generated family of ordered trees $\mathcal{F}$, freely generated by a set of symbols $\mathcal{S}$ equipped with an arity $\nu : \mathcal{S} \to \mathbb{N}$:

$$\mathcal{F} = \mathcal{S}_0 \star \mathcal{F}^0 + \mathcal{S}_1 \star \mathcal{F}^1 + \mathcal{S}_2 \star \mathcal{F}^2 + \mathcal{S}_3 \star \mathcal{F}^3 + \cdots = \bigcup_{k \geq 0} \mathcal{S}_k \star \mathcal{F}^k \ , \tag{3.7}$$

where $\mathcal{S}_k$ is the subset of symbols of arity k ($\mathcal{S}_k = \{s \in \mathcal{S} \mid \nu(s) = k\}$), the size of all subsets $\mathcal{S}_k$ is bounded by some constant and $\mathcal{S}_0$ is not empty. Here, $\mathcal{S}_k \star \mathcal{F}^k$ is short hand for $\bigcup_{s \in \mathcal{S}_k} \{s\} \star \mathcal{F}^k$.

It is well known (see for instance [21, 61, 79]) that the counting generating function of $\mathcal{F}$ satisfies

$$F(z) = z \cdot \phi(F(z)) \ ,$$

where

$$\phi(u) = \sum_{k \geq 0} \phi_k \, u^k$$

and ϕ_k is the number of symbols of arity k. Under some mild assumptions, it can easily be proved that

$$F(z) \sim \tau - c\,(1 - z/\rho)^{1/2} \ ,$$

as $z \to \rho$, where $\rho = \psi(\tau)$, $\psi(u) = u/\phi(u)$, τ is the smallest positive root of $\psi'(x) = 0$ and

$$c = \sqrt{\frac{-2\rho}{\psi''(\tau)}} = \sqrt{\frac{2\phi(\tau)}{\phi''(\tau)}} \ .$$

Then, using standard singularity analysis, it is easy to show that the number of trees of size n in $\mathcal{F}$ is asymptotically given

$$[z^n]F(z) \sim n!\,\rho^{-n}\,n^{-3/2}\,\frac{c}{2\sqrt{\pi}} \ .$$

Let us consider now the average cost of unranking an object from a simply generated family $\mathcal{F}$. From the rule for sequences of k components (see Corollary 3), it follows that $\Upsilon\mathcal{F}^k = k\,F^{k-1}\,(\vartheta F + \Upsilon\mathcal{F})$, for $k \geq 0$, where $\mathcal{F}^k$ is used as a shorthand for $\mathsf{Seq}(\mathcal{F}, \mathrm{card} = k)$. We assume thus that the specification is given as

$$\mathcal{F} = Z \star (\underbrace{\mathcal{F}^0 + \cdots + \mathcal{F}^0}_{\phi_0 \text{ times}} + \underbrace{\mathcal{F}^1 + \cdots + \mathcal{F}^1}_{\phi_1 \text{ times}} + \underbrace{\mathcal{F}^2 + \cdots + \mathcal{F}^2}_{\phi_2 \text{ times}} + \cdots) \ .$$

On the other hand, $\Upsilon(Z \star \mathcal{C}) = z\,(C + \Upsilon\mathcal{C})$, and thus

$$\begin{aligned} \Upsilon\mathcal{F} = \Upsilon(Z \star \phi(\mathcal{F})) = z \cdot \phi(F) + z \cdot \Upsilon(\phi(\mathcal{F})) &= F + z \sum_{k \geq 0} \phi_k \, \Upsilon\mathcal{F}^k \\ &= F + (\vartheta F + \Upsilon\mathcal{F}) \cdot (\vartheta\phi)(F) = \frac{(\vartheta F)^2}{F} \ , \end{aligned}$$

where $\phi(\mathcal{F})$ stands for the right hand side of Equation (3.7). The last equality involves a somewhat lengthy computation along the lines suggested by the second-to-last equality, and the use of some identities, in particular, $\vartheta F = F + \vartheta F \cdot (\vartheta\phi)(F)$.

Using standard analytic techniques it is easy to establish the following expansion of $\Upsilon\mathcal{F}$ around the dominant singularity $z = \rho$:

$$\Upsilon\mathcal{F}(z) \sim \frac{c^2}{4\tau}\,(1 - z/\rho)^{-1} ,$$

where ρ, τ and c are defined as above; the rest is routine application of singularity analysis, yielding the following lemma.

Lemma 1. *Let $\mathcal{F}$ be a simply generated family of trees whose counting generating function $F(z)$ satisfies $F(z) = z \cdot \phi(F(z))$, with $\phi(u) = \sum_{k\geq 0} \phi_k\, u^k$ and $\phi_k =$ number of symbols of arity k. Furthermore, assume that $\gcd\{k \mid \phi_k \neq 0\} = 1$. Then the expected cost $\mu_{n,\mathcal{F}}$ of unranking an object of size n in $\mathcal{F}$ is*

$$\mu_{n,\mathcal{F}} = K_{\mathcal{F}}\,\frac{1}{2}\,n\,\sqrt{\pi n} + \mathrm{O}(n) ,$$

where

$$K_{\mathcal{F}} = \frac{c}{\tau} = \frac{1}{\tau}\sqrt{\frac{2\phi(\tau)}{\phi''(\tau)}} , \qquad \rho = \psi(\tau) , \qquad \psi(u) = \frac{u}{\phi(u)} ,$$

and τ is the smallest positive root of $\psi'(x) = 0$.

If we had used the specification $\mathcal{F} = \mathcal{S}_0 + \mathcal{S}_1 \star \mathcal{F} + \mathcal{S}_2 \star \mathcal{F} \star \mathcal{F} + \mathcal{S}_3 \star \mathcal{F} \star \mathcal{F} \star \mathcal{F} + \cdots$ for simply generated families of trees, then

$$\Upsilon\mathcal{F} = \frac{z\,\phi_0\,\vartheta F}{F}\left(\frac{\vartheta F}{F} - 1\right)$$

because $\Upsilon\mathcal{F}^k \neq \Upsilon(\overbrace{\mathcal{F} \star \ldots \star \mathcal{F}}^{k \text{ times}})$ for any $k > 0$. In that case the expected cost will be

$$\mu_{n,\mathcal{F}} = K^*_{\mathcal{F}}\,\frac{1}{2}\,n\,\sqrt{\pi n} + \mathrm{O}(n) ,$$

where

$$K^*_{\mathcal{F}} = \frac{\rho c \phi_0}{\tau^2} .$$

Lemma 1 also applies to unlabelled families of trees, as the definition of simply generated families only involves unions and products, which obey the same rules for both labelled and unlabelled unranking (see Chapter 4).

For instance, for general trees

$$\mathcal{G} = \{\circ\} \star \epsilon + \{\circ\} \star \mathcal{G}^1 + \{\circ\} \star \mathcal{G}^2 + \cdots = \{\circ\} \star \mathsf{Seq}(\mathcal{G})$$

we have $\phi(u) = 1/(1-u)$. Then $\tau = 1/2$, $\rho = 1/4$ and $\phi''(u) = 16$. Hence, $K_{\mathcal{G}} = 1$ and $\mu_{n,\mathcal{G}} = n\sqrt{\pi n}/2 + \mathrm{O}(n)$.

Another example are *Motzkin trees* or *unary-binary trees* $\mathcal{M} = Z \star (\epsilon + \mathcal{M}^1 + \mathcal{M}^2)$, for which $\phi(u) = 1 + u + u^2$. The dominant singularity of $\mathcal{M}$ (and hence of $\Upsilon\mathcal{M}$) is located at $z = 1/3$ and applying the lemma above one easily obtains $\mu_{n,\mathcal{M}} = \frac{1}{2} n\sqrt{3\pi n} + \mathrm{O}(n)$, with $K_{\mathcal{M}} = \sqrt{3}$.

The same result can be arrived at by using the specification $\mathcal{M} = \mathcal{B}[\mathcal{L}]$, where $\mathcal{B} = Z + Z \star \mathcal{B} \star \mathcal{B}$ and $\mathcal{L} = Z \star \mathsf{Seq}(Z)$, i.e., a unary-binary tree is the result of substituting each node of a binary tree ($\mathcal{B}$) by a sequence of 1 or more atoms ($\mathcal{L}$), and applying rule 7 of Corollary 3 for substitutions.

The conditions of the lemma can be somewhat relaxed, in particular, the condition $\gcd\{k \mid \phi_k \neq 0\} = 1$, and still get a similar result. However, one gets multiple dominant singularities in that case and has to sum up their contributions. As a result, the asymptotic estimate for the average cost of unranking is $\mathrm{O}(n\sqrt{n})$, but the computation of the constant requires some additional effort. Moreover, if $d \geq 2$ is the gcd of the ϕ_k's then the value of $\mu_{n,\mathcal{F}} = 0$ unless $n \ \mathbf{mod}\ d = 1$. For instance, for binary trees defined by $\mathcal{B} = Z \star (\epsilon + \mathcal{B}^2)$ ($\phi(u) = 1 + u^2$, $d = 2$), the average cost of unranking is 0 if n is even—which is not a surprise, as there are no binary trees of even size.

Lemma 1 does apply to simply generated families of non-ordered trees as well, as shown in [26]. In particular,

$$\Upsilon\mathcal{F} = F + (\vartheta F + \Upsilon\mathcal{F}) \cdot (\vartheta\phi)(F) = \frac{(\vartheta F)^2}{F} \ ,$$

where we have now

$$\phi(F) = \sum_{k\geq 0} \phi_k \frac{F^k}{k!} \ .$$

For instance, for Cayley trees we have $T(z) = z \cdot \exp(T(z))$; hence $\rho = e^{-1}$, $\tau = 1$ and $K_{\mathcal{T}} = \sqrt{2}$. Then

$$\mu_{n,\mathcal{T}} = n\sqrt{\frac{n\pi}{2}} + \mathrm{O}(n) \ .$$

Back again to the cost of unranking simply generated families of ordered trees, the evaluation of $\Upsilon^{(2)}\mathcal{F}$ from the rules given in Theorem 14 is a lengthy computation, although feasible with the help of a symbolic computation system. The derived rule for $\Upsilon^{(2)}\mathcal{F}^k$ is by no means as simple and short as that for $\Upsilon\mathcal{F}^k$ and it will not be given here.

Another way to get the variance of the cost of unranking a simple family $\mathcal{F}$ is to obtain a recurrence for $\hat{\Upsilon}\mathcal{F}$, differentiate it twice w.r.t. u and set $u = 1$. Given the simplicity of the rule for $\hat{\Upsilon}\mathcal{F}$ and the independent interest of that formula, we will follow and briefly describe this second approach. On passing, the functional equation satisfied by $\hat{\Upsilon}\mathcal{F}(z, u)$ strongly suggests an asymptotic Airy distribution for the cost of unranking simple families of trees [80, 81, 82].

Lemma 2. *The bivariate generating function $\hat{\Upsilon}\mathcal{F}$ satisfies*

$$\hat{\Upsilon}\mathcal{F}(z, u) = z\, u \cdot \phi(\hat{\Upsilon}\mathcal{F}(zu, u)) \ .$$

Several identities, like $\hat{\Upsilon}\mathcal{F}(z, 1) = F(z)$, $F(z) = z \cdot \phi(F(z))$, etc. have to be applied in order to reduce the expression for $\Upsilon^{(2)}\mathcal{F}$ to a manageable form

$$\Upsilon^{(2)}\mathcal{F}(z) = 5\,\frac{(\vartheta F)^2 \cdot \vartheta^2 F}{F^2} + \frac{(\vartheta F)^3}{F^2} - \frac{\vartheta F \cdot \vartheta^2 F}{F} - \frac{(\vartheta F)^2}{F} - 4\,\frac{(\vartheta F)^4}{F^3} \ .$$

Once $\Upsilon^{(2)}\mathcal{F}$ has been expressed in terms of $F(z)$ and its first and second derivatives, the corresponding asymptotic estimate of $\Upsilon^{(2)}(\mathcal{F})$ around the singularity $z = \rho$ is

$$\Upsilon^{(2)}\mathcal{F}(z) \sim \frac{5c^3}{16\tau^2}\,(1 - z/\rho)^{-5/2} ,$$

which can be readily used to get the asymptotic form of its coefficients and finally, the variance.

Lemma 3. *Let $\mathcal{F}$ be a simply generated family of (non)ordered trees whose counting generating function $F(z)$ satisfies $F(z) = z \cdot \phi(F(z))$, with $\phi(u) = \sum_{k\geq 0} \phi_k\, u^k$ and $\phi_k =$ number of symbols of arity k. Furthermore, assume that $\gcd\{k \mid \phi_k \neq 0\} = 1$. Then the variance of the cost $\sigma^2_{n,\mathcal{F}}$ of unranking an object of size n in $\mathcal{F}$ is*

$$\sigma^2_{n,\mathcal{F}} = \frac{[z^n]\Upsilon^{(2)}\mathcal{F}(z)}{[z^n]F(z)} + \mu_{n,\mathcal{F}} - \mu^2_{n,\mathcal{F}} = K^2_{\mathcal{F}}\left(\frac{5}{6} - \frac{\pi}{4}\right) n^3 + \mathrm{O}\left(n^{5/2}\right) ,$$

where $K_{\mathcal{F}}$ is as in Lemma 1.

For instance, the variance of unranking general trees is $0.0479351698\ldots\cdot n^3 + \mathrm{o}(n^3)$, while that of unary-binary trees is $0.1438055094\ldots\cdot n^3 + \mathrm{o}(n^3)$. For Cayley trees, the variance is $0.0958703\ldots\cdot n^3 + \mathrm{o}(n^3)$.

3.3.2 Binary Trees

Let $\mathcal{B}$ be the labelled class of binary trees $\mathcal{B} = Z + \mathcal{B}\star\mathcal{B}$, then (Theorem 12)

$$B(z) = \frac{1-\sqrt{1-4z}}{2} \quad \text{and} \quad \Upsilon\mathcal{B} = \frac{z}{2\,(1-4z)} - \frac{z}{2\sqrt{1-4z}} .$$

Thus, the n-th coefficient of $B(z)$ is given by (Theorem 3)

$$\left(\frac{1}{n!}\right)[z^n]B(z) = \frac{1}{4\sqrt{\pi}}\,4^n\,n^{-3/2}\left(1 + \frac{3}{8\,n} + \frac{25}{128\,n^2} + \mathrm{O}(n^{-3})\right)$$

and the n-th coefficient of the asymptotic expansion of $\Upsilon\mathcal{B}$ verifies (Theorem 4)

$$\left(\frac{1}{n!}\right)[z^n]\Upsilon\mathcal{B} = \frac{4^n}{8} - \frac{1}{8\sqrt{\pi}}\,4^n\,n^{-1/2}\left(1 - \frac{1}{8\,n} + \frac{1}{128\,n^2} + \mathrm{O}(n^{-3})\right) .$$

From these calculations, the average cost is

$$\mu_{n,\mathcal{B}} = \frac{\sqrt{\pi}}{2}\,n^{3/2} - \frac{1}{2}\,n - \frac{3\sqrt{\pi}}{16}\,n^{1/2} + \frac{1}{4} + \mathrm{O}(n^{-1/2}) . \tag{3.8}$$

3.3.3 Motzkin Trees

For labelled Motzkin trees, $\mathcal{M} = Z + (Z \star \mathcal{M} + Z \star (\mathcal{M} \star \mathcal{M}))$, the corresponding generating function and the cumulating cost (Theorem 12) are

$$M(z) = \frac{1 - z - \sqrt{1 - 2z - 3z^2}}{2z} \quad \text{and} \quad \Upsilon\mathcal{M} = \frac{z}{1 - 2z - 3z^2} - \frac{z}{\sqrt{1 - 2z - 3z^2}} \ ,$$

respectively. Thus,

$$\left(\frac{1}{n!}\right) [z^n] M(z) = \frac{\sqrt{3}}{2\sqrt{\pi}} 3^n \, n^{-3/2} \left(1 + \frac{3}{8n} + \frac{25}{128\, n^2} + \mathrm{O}(n^{-3})\right)$$

and

$$\left(\frac{1}{n!}\right) [z^n] \Upsilon\mathcal{M} = \frac{3^n}{4} - \frac{\sqrt{3}}{6\sqrt{\pi}} 3^n \, n^{-1/2} \left(1 - \frac{1}{8n} + \frac{1}{128\, n^2} + \mathrm{O}(n^{-3})\right) \ ,$$

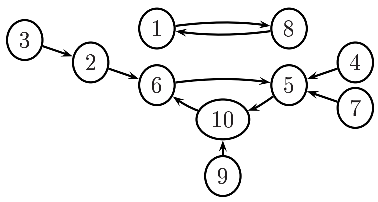
and the average cost of unranking Motzkin trees of size n is

$$\mu_{n,\mathcal{M}} = \frac{\sqrt{3\,\pi}}{6} n^{3/2} - \frac{1}{3} n - \frac{\sqrt{3\,\pi}}{16} n^{1/2} + \frac{1}{6} + \mathrm{O}(n^{-1/2}) \ , \tag{3.9}$$

We remark here that the coefficient of the main term of the average cost is $\sqrt{3\pi}/6$ since $K^*_{\mathcal{M}} = \sqrt{3}/3$. If we had used the specification $\mathcal{M} = Z \star \mathcal{M}^0 + Z \star \mathcal{M}^1 + Z \star \mathcal{M}^2$ then we would have that this coefficient is $\sqrt{3\pi}/2$ since $K_{\mathcal{M}} = \sqrt{3}$.

3.3.4 Functional Graphs

Functional graphs $\mathcal{F}$ are sets of cycles of Cayley trees $\mathcal{T}$, $\mathcal{F} = \mathsf{Set}(\mathcal{C})$ where $\mathcal{C} = \mathsf{Cycle}(\mathcal{T})$ and $\mathcal{T} = Z \star \mathsf{Set}(\mathcal{T})$. For instance, the graph for the function f, with $f(1) = 8, f(2) = 6, f(3) = 2, f(4) = 5, f(5) = 10, f(6) = 5, f(7) = 5, f(8) = 1, f(9) = 10, f(10) = 6$ is



which is the 5000000-th object of size 10 in $\mathcal{F}$ in lexicographic-PF order.

Routine application of the rules given in Section 3.2.2 yields

$$\Upsilon\mathcal{F} = \frac{T(z)\ (3 - 2\,T(z))}{(1 - T(z))^4} \ ,$$

where

$$T(z) = \sum_{n \ge 0} n^{n-1} \frac{z^n}{n!} = 1 - \sqrt{2}(1 - ez)^{1/2} + \frac{2}{3}(1 - ez) + \mathrm{O}((1 - ez)^{3/2})$$

is the counting generating function of Cayley trees. Then

$$F(z) = \frac{1}{1 - T(z)} = \frac{1}{3} + \frac{\sqrt{2}}{2} (1 - ez)^{-1/2} + \mathrm{O}((1 - ez)^{1/2})$$

and

$$\Upsilon\mathcal{F} = c_f + \frac{1}{4}(1-ez)^{-2} + \frac{7\sqrt{2}}{12}(1-ez)^{-3/2} + \mathrm{O}((1-ez)^{-1}) \ ,$$

for some constant c_f. Thus,

$$\left(\frac{1}{n!}\right)[z^n]F(z) = \frac{\sqrt{2}}{2\sqrt{\pi}}\, e^n\, n^{-1/2} + \mathrm{O}(e^n\, n^{-3/2})$$

and

$$\left(\frac{1}{n!}\right)[z^n]\Upsilon\mathcal{F} = \frac{e^n\, n}{4} + \frac{7\sqrt{2}}{6\sqrt{\pi}}\, e^n\, n^{1/2} + \frac{e^n}{4} + \frac{7\sqrt{2}}{16\sqrt{\pi}}\, e^n\, n^{-1/2} + \mathrm{O}(e^n\, n^{-3/2}) \ ,$$

and the average cost of unranking functional graphs is

$$\mu_{n,\mathcal{F}} = \frac{\sqrt{2\pi}}{4}\, n^{3/2} + \frac{7}{3}\, n + \frac{\sqrt{2\,\pi}}{4}\, n^{1/2} + \frac{7}{24} + \mathrm{O}(n^{-1/2}) \ .$$

3.3.5 Surjections

According to the rules given in Theorem 12 and Corollary 3, for the class of surjections $\mathcal{C} = \mathsf{Seq}(\mathsf{Set}(Z, \mathrm{card} \geq 1))$ we have

$$C(z) = \frac{1}{1 - z\,\exp(z)} \qquad \text{and} \qquad \Upsilon\mathcal{C} = \frac{2\,z\,\exp(z)\,(1+z)}{(1-z\,\exp(z))^2} \ .$$

Thus, the unranking average cost is linear: $\mu_{n,\mathcal{C}} = 2\,n$.

3.4 Variants and Heuristics

It is important to mention that there can be slight differences of the cost of unranking a class depending on the given specification. For instance, permutations can be defined either sequences of atoms or sets of cycles of atoms. Let permutations be defined as sequences of atoms, i.e. $\mathcal{P} = \mathsf{Seq}(Z)$. By Corollary 3, the equation for $\Upsilon\mathcal{P}$ is $\Upsilon\mathcal{P} = z/(1-z)^2$. It is obvious that the generating function of permutations is $P(z) = 1/(1-z)$. Thus, extracting coefficients, the average cost is $\mu\mathcal{P}_n = n$. But, for permutations defined as sets of cycles of atoms, $\mathcal{P} = \mathsf{Set}(\mathcal{C})$ with $\mathcal{C} = \mathsf{Cycle}(Z)$, then $\Upsilon\mathcal{P} = \exp(C)\cdot(\vartheta C + \Upsilon\mathcal{C})$ where $\Upsilon\mathcal{C} = z/1-z$, and again $P(z) = 1/(1-z)$. Hence, $\Upsilon\mathcal{P} = 2z/(1-z)^2$ and $\mu\mathcal{P}_n = 2n$. Even though in both cases the cost of unranking is linear, the second specification leads to an unranking procedure whose cost is twice the other.

The "commutative" isomorphisms $\mathcal{A}\star\mathcal{B} \simeq \mathcal{B}\star\mathcal{A}$ and $\mathcal{A}^{\,\Box}\!\star\mathcal{B} \simeq \mathcal{B}\star^{\Box}\mathcal{A}$ can be systematically exploited to improve the performance of unranking. They constitute the basis for the *big-endian heuristic* introduced by Flajolet *et al.* [26] in the context of random generation.

Using the specification $\mathcal{A}\star\mathcal{B}$ or $\mathcal{B}\star\mathcal{A}$ makes no difference from the point of view of the algorithm—it is even possible that the unranking algorithm transforms an original specification $\mathcal{A}\star\mathcal{B}$ to the isomorphic specification $\mathcal{B}\star\mathcal{A}$, performs the actual unranking w.r.t. $\mathcal{B}\star\mathcal{A}$ and performs the inverse transformation on the returned object. But the performance can radically change from one specification to the other. In particular, if $\Upsilon(\mathcal{A}\star\mathcal{B})$ and

$\Upsilon(\mathcal{B} \star \mathcal{A})$ are compared, then the former specification would be the best choice if the n-th coefficient of $\vartheta A \cdot B$ is asymptotically smaller than the n-th coefficient of $\vartheta B \cdot A$, and the other way around. For instance, consider the products $\mathcal{A} \star \mathsf{Seq}(\mathcal{A})$ and $\mathsf{Seq}(\mathcal{A}) \star \mathcal{A}$. Typically one has

$$[z^n]\vartheta\left(\frac{1}{1-A}\right) \cdot A \gg [z^n]\vartheta A \cdot \frac{1}{1-A} \ ,$$

and hence it is advisable to define sequences by the specification $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \star \mathsf{Seq}(\mathcal{A})$ rather than the equivalent (isomorphic) specification $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathsf{Seq}(\mathcal{A}) \star \mathcal{A}$. In general, the specification $\mathcal{A} \star \mathcal{B}$ is to be preferred over $\mathcal{B} \star \mathcal{A}$ if the n-th coefficient of A is of a lower order of magnitude than the n-th coefficient of B.

On the other hand, the equivalences $\mathcal{C} = \mathcal{A}^{\square}\!\star \mathcal{B} \iff \vartheta\mathcal{C} = \vartheta\mathcal{A} \star \mathcal{B}$ and $\mathcal{C} = \mathcal{A} \star^{\square} \mathcal{B} \iff \vartheta\mathcal{C} = \mathcal{A} \star \vartheta\mathcal{B}$ yield the following rules for the Υ-operator:

$$\begin{aligned} \vartheta\Upsilon(\mathcal{A}^{\square}\!\star \mathcal{B}) &= \vartheta^2 A \cdot B + \vartheta A \cdot \Upsilon\mathcal{B} + B \cdot \vartheta\Upsilon\mathcal{A} \quad \text{and} \\ \vartheta\Upsilon(\mathcal{B} \star^{\square} \mathcal{A}) &= \vartheta B \cdot \vartheta A + \vartheta A \cdot \Upsilon\mathcal{B} + B \cdot \vartheta\Upsilon\mathcal{A} \ , \end{aligned}$$

so the comparison should be now between the n-th coefficients of $\vartheta^2 A \cdot B$ and $\vartheta A \cdot \vartheta B$. This comparison is significant when choosing an appropriate specification for sets and cycles in terms of boxed products. Recall that in the previous sections we have used $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}^{\square}\!\star \mathsf{Set}(\mathcal{A})$ and $\mathsf{Cycle}(\mathcal{A}) = \mathcal{A}^{\square}\!\star \mathsf{Seq}(\mathcal{A})$, but, in general, using the alternative specifications $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathsf{Set}(\mathcal{A}) \star^{\square} \mathcal{A}$ and $\mathsf{Cycle}(\mathcal{A}) = \mathsf{Seq}(\mathcal{A}) \star^{\square} \mathcal{A}$ yields better performance. For example, while the average cost of unranking Cayley trees is $\Theta(n^{3/2})$ (see Section 3.3), if the alternative isomorphism for sets is used then the average cost is $\Theta(n \log n)$ [26]. Indeed, for the specification $\{\mathcal{T} = Z \star \mathsf{Set}(\mathcal{T}),\ \mathsf{Set}(\mathcal{T}) = \epsilon + \mathsf{Set}(\mathcal{T}) \star^{\square} \mathcal{T}\}$, we have

$$\vartheta\Upsilon\mathcal{T} = \Upsilon\mathcal{T} + T \cdot \vartheta\Upsilon\mathcal{T} + \vartheta T \cdot \Upsilon\mathcal{T} + T \cdot (\vartheta T)^2 \ ,$$

so that

$$\Upsilon\mathcal{T} = \vartheta T \cdot \log\left(\frac{1}{1-T}\right) - \vartheta T \cdot T$$

whose n-th coefficient is $\Theta(e^n\, n^{-1/2} \log n)$. Since $[z^n]T(z)$ is $\Theta(e^n\, n^{-3/2})$ the result follows. The result is easily generalized to simply generated families of non-ordered trees; all these can be unranked with average cost $\Theta(n \log n)$, using the right boxed products ($\star^{\square}$) to specify sets and sets of restricted cardinality.

However, the $\hat{\Upsilon}$- and Υ-rules for sets and cycles, their variants with restricted cardinality and for substitutions, as given in Section 3.2, are much simpler than those for the specifications based upon the big-endian heuristic. For instance, for $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathsf{Set}(\mathcal{A}) \star^{\square} \mathcal{A}$ and $\mathsf{Cycle}(\mathcal{A}) = \mathsf{Seq}(\mathcal{A}) \star^{\square} \mathcal{A}$, the rules are

$$\Upsilon\mathsf{Set}(\mathcal{A}) = \exp(A)\left(\Upsilon\mathcal{A} + \int \frac{(\vartheta A)^2}{z}\, dz\right) \quad \text{and} \quad \Upsilon\mathsf{Cycle}(\mathcal{A}) = \frac{\Upsilon\mathcal{A}}{(1-A)^2} + \int \frac{2}{z}\frac{(\vartheta A)^2}{(1-A)^2}\, dz \ ,$$

respectively. These rules are obtained from the following equations

$$\begin{aligned} \vartheta\Upsilon\mathsf{Set}(\mathcal{A}) &= \Upsilon\vartheta\mathsf{Set}(\mathcal{A}) = \Upsilon(\mathsf{Set}(\mathcal{A}) \star \vartheta\mathcal{A}) \\ &= \vartheta(\exp(A)) \cdot \vartheta A + \Upsilon\mathsf{Set}(\mathcal{A}) \cdot \vartheta A + \exp(A) \cdot \Upsilon\vartheta\mathcal{A} \end{aligned}$$

$$= \exp(A) \cdot ((\vartheta A)^2 + \vartheta \Upsilon \mathcal{A}) + \Upsilon \mathsf{Set}(\mathcal{A}) \cdot \vartheta A$$

and

$$\begin{aligned}
\vartheta \Upsilon \mathsf{Cycle}(\mathcal{A}) &= \Upsilon \vartheta \mathsf{Cycle}(\mathcal{A}) = \Upsilon(\mathsf{Seq}(\mathcal{A}) \star \vartheta \mathcal{A}) \\
&= \vartheta\left(\frac{1}{1-A}\right) \cdot \vartheta A + \Upsilon \mathsf{Seq}(\mathcal{A}) \cdot \vartheta A + \frac{1}{1-A} \cdot \Upsilon \vartheta \mathcal{A} \\
&= \frac{(\vartheta A)^2}{(1-A)^2} + \frac{\vartheta A + \Upsilon \mathcal{A}}{(1-A)^2}\, \vartheta A + \frac{\vartheta \Upsilon \mathcal{A}}{1-A} = \frac{2\,(\vartheta A)^2}{(1-A)^2} + \vartheta\left(\frac{\Upsilon \mathcal{A}}{(1-A)^2}\right) .
\end{aligned}$$

Furthermore, for substitutions $\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^{\{k\}}$, we also can apply the big-endian heuristic for sets with fixed cardinality, $\mathcal{A}^{\{k\}} = \mathcal{A}^{\{k-1\}} \star^{\Box} \mathcal{B}$, and then we obtain the rule

$$\Upsilon(\mathcal{A} \circ \mathcal{B}) = (\vartheta A + \Upsilon \mathcal{A}) \circ \mathcal{B} + \Upsilon \mathcal{B} \cdot (A' \circ B) + \int \frac{\vartheta^2 B}{z}\, dz \cdot (A'' \circ B) .$$

In any case, the big-endian heuristic is useful both for unranking and for random generation, and yields the same type of improvements.

We also can consider the following heuristic based on the isomorphism

$$\mathcal{A} \star \mathcal{B} \simeq \mathcal{A} {}^{\Box}\!\star \mathcal{B} + \mathcal{A} \star^{\Box} \mathcal{B}$$

together with big-endian heuristic. If we take

$$\mathcal{A} \star \mathcal{B} \simeq \mathcal{A} {}^{\Box}\!\star \mathcal{B} + \mathcal{B} {}^{\Box}\!\star \mathcal{A} \tag{3.10}$$

then $\vartheta \Upsilon(\mathcal{A} \star \mathcal{B}) = \vartheta^2 A \cdot B + A \cdot \vartheta^2 B + \vartheta(\Upsilon \mathcal{A} \cdot B + A \cdot \Upsilon \mathcal{B})$, whereas if we take

$$\mathcal{A} \star \mathcal{B} \simeq \mathcal{B} \star^{\Box} \mathcal{A} + \mathcal{A} \star^{\Box} \mathcal{B} \tag{3.11}$$

then $\vartheta \Upsilon(\mathcal{A} \star \mathcal{B}) = 2 \cdot \vartheta A \cdot \vartheta B + \vartheta(\Upsilon \mathcal{A} \cdot B + A \cdot \Upsilon \mathcal{B})$. Thus, we should take the isomorphism based on right boxed products (3.11) because $2 \cdot \vartheta A \cdot \vartheta B \ll \vartheta^2 A \cdot B + A \cdot \vartheta^2 B$. We call the heuristic resulting from (3.11) the *boxed heuristic*.

For instance, rather than the usual specification $\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$ for binary trees, it is more convenient to use

$$\mathcal{B} \simeq Z + \mathcal{B} \star^{\Box} \mathcal{B} + \mathcal{B} \star^{\Box} \mathcal{B} \simeq Z + (\mathcal{B} + \mathcal{B}) \star^{\Box} \mathcal{B} .$$

Then we have $\vartheta \Upsilon \mathcal{B} \cdot (1 - 2\,B) = 2\,(\vartheta B)^2 + 2\,\vartheta B \cdot \Upsilon \mathcal{B}$, and solving this first-order linear differential equation, we get

$$\Upsilon \mathcal{B} = \frac{1}{8} \log\left(\frac{1}{1-4z}\right) \frac{1}{\sqrt{1-4z}} - \frac{z}{2} \frac{1}{\sqrt{1-4z}} .$$

Hence, we can do the unranking of binary trees in time $\mathrm{O}(n \log n)$, in particular,

$$\mu_{n,\mathcal{B}} = \frac{1}{2} n \log n + \frac{2 \log 2 + \gamma - 1}{2} n - \frac{1}{4} \log n - \frac{2 \log 2 + \gamma - 1}{8} + \mathrm{O}\left(\frac{\log n}{\sqrt{n}}\right) ,$$

where $\gamma = 0.5772156649\ldots$ is Euler's constant.

The boxed heuristic for unranking is equivalent to *differential heuristic* for random generation introduced in [26]; however, we cannot directly use the differential heuristic because of the same reasons that we discussed in Section 2.2.

Chapter 4

Unranking of Unlabelled Classes

In this chapter we study the unranking of unlabelled classes. As in the previous chapter, we solve the unranking problem for any class given by an admissible specification and we analyze the performance of these algorithms. However, no efficient solution has been found for those involving (unlabelled) cycles and we will not cope with them. Hence, we assume that the admissible specifications for unlabelled classes are built from unions, Cartesian products, sequences, powersets, sets[1], substitutions, and sequences, powersets and sets with restricted cardinality.

For $\mathcal{A} + \mathcal{B}$ we assume that the $\mathcal{A}$-objects come first and afterwards the $\mathcal{B}$-objects. For Cartesian products we will consider the lexicographic and the boustrophedonic orders. Sequences follow the isomorphism (2.6). For powersets and sets we will consider their unranking according to LScR order (Section 2.3). Sequences, powersets and sets with restricted cardinalities are unranked using similar isomorphisms (Table 2.5). Finally, the unranking of substitutions is based in isomorphism (1.1).

Moreover, we use the same notation for unlabelled objects as for labelled objects, but we will use 2-tuples instead of 3-tuples to represent them. For instance, $\gamma = \langle Z, \langle\langle Z, \epsilon\rangle, \langle Z, \epsilon\rangle\rangle\rangle$ of rooted unlabelled trees $\mathcal{D} = Z \times \mathsf{Set}(\mathcal{D})$ represents the object $\gamma' = Z \times \mathsf{Set}(Z \times \epsilon, Z \times \epsilon)$ because

$$\gamma \equiv \langle Z, \langle Z \times \epsilon, Z \times \epsilon\rangle\rangle \equiv \langle Z, \mathsf{Set}(Z \times \epsilon, Z \times \epsilon)\rangle \equiv \gamma' \ .$$

For substitutions, we have the special notation $\circ\langle \alpha, \beta\rangle$, where α and β are objects expressed in the same format.

This chapter is organized in a similar way as the previous chapter. In Section 4.1 we describe the unranking algorithms, and in Section 4.2 we analyze the performance of the unranking algorithms (we give a few basic results on the worst-case complexity of unranking and we provide a Υ-algebra). We also give some examples and remarks about the effects of alternative isomorphisms (e.g., the *big-endian heuristic*).

Some of the results here described are included in the article [56].

[1] Also called multisets or bags.

4.1 The Algorithms

4.1.1 Basic Algorithms: + and ×

The algorithms for the unranking of ϵ and atomic classes are extremely simple and we shall no extend further with their description. The algorithm for unlabelled unions is as Algorithm 1 of Section 3.1 for labelled unions.

Cartesian products $\mathcal{A} \times \mathcal{B}$ in lexicographic or boustrophedonic order are based in the same ideas as the unranking of partitional products; but now we do not have to care of the atoms's labels. For instance, Algorithm 9 describes the lexicographic unranking of products (compare to Algorithm 2).

Algorithm 9 Unranking of products (lexicographic order)

Unrank($\mathcal{A} \times \mathcal{B}, n, i$)
  $c := 0; j := 0; d := \text{Count}(\mathcal{A}, j) \cdot \text{Count}(\mathcal{B}, n-j)$
  while $c + d \leq i$ **do** $c := c + d; j := j + 1; d := \text{Count}(\mathcal{A}, j) \cdot \text{Count}(\mathcal{B}, n-j)$ **end**
  $i' := i - c;\ b := \text{Count}(\mathcal{B}, n-j)$
  $\alpha := \text{Unrank}(\mathcal{A}, j, i' \textbf{ div } b);\ \beta := \text{Unrank}(\mathcal{B}, n-j, i' \textbf{ mod } b)$
  return $\langle \alpha, \beta \rangle$

4.1.2 Algorithms for Sequences, Powersets and Sets

The unranking of sequences in lexicographic order is the same as in the labelled case because it is based in the same isomorphism (2.6). The same holds for the unranking of sequences in card-lexicographic order and sequences with restricted cardinality.

For powersets we consider the LScR order (2.13),

$$\mathsf{PSet}(\mathcal{A}) = \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \mathcal{A}_j^{\{k\}} \times \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \ . \tag{4.1}$$

Let $\hat{\delta}_{n,j}$ be the cardinality of the class of powersets of $\mathcal{A}$ such that their smallest component is of size at least j. Obviously, $\hat{\delta}_{n,1} = \#\mathsf{PSet}_n(\mathcal{A}_{\geq 1}) = \#\mathsf{PSet}_n(\mathcal{A})$. Fixing a size n, this class can be recursively defined by

$$\mathsf{PSet}_n(\mathcal{A}_{\geq j}) = \mathsf{PSet}_n(\mathcal{A}_{\geq j+1}) + \bigcup_{1 \leq k \leq \lfloor n/j \rfloor} \mathcal{A}_j^{\{k\}} \times \mathsf{PSet}_{n-kj}(\mathcal{A}_{\geq j+1}) \ ,$$

$\mathsf{PSet}_n(\mathcal{A}_{\geq j}) = \emptyset$ if $j > n > 0$ and $\mathsf{PSet}_0(\mathcal{A}_{\geq j}) = \epsilon$ if $j \geq 0$. Therefore we have $\hat{\delta}_{0,j} = 1$ for all $j \geq 0$ and for $1 \leq n \leq N$ the recurrence satisfied by the $\hat{\delta}_{n,j}$'s is

$$\hat{\delta}_{n,j} = \begin{cases} 0 & \text{if } j > n, \\ \hat{\delta}_{n,j+1} + \sum_{1 \leq k \leq \lfloor n/j \rfloor} \binom{a_j}{k} \hat{\delta}_{n-kj,j+1} & \text{if } j \leq n. \end{cases}$$

Thus, $\hat{\delta}_{n,j}$ can be computed for $1 \le j \le n \le N$ in $\mathrm{O}(N^2 \log N)$ arithmetic operations:

$$\sum_{1\le n\le N} \sum_{1\le j\le n} \sum_{1\le k\le \lfloor n/j \rfloor} \mathrm{O}(1) = \sum_{1\le n\le N} \mathrm{O}(n \log n) = \mathrm{O}(N^2 \log N) \ .$$

It is obvious that the number of powersets of size n with at least one component of size smaller or equal than j, i.e. the cardinality of

$$\bigcup_{l=1}^{j-1} \bigcup_{k=\lfloor n/l \rfloor}^{1} \mathcal{A}_l^{\{k\}} \times \mathsf{PSet}_{n-kl}(\mathcal{A}_{\ge l+1}) \ ,$$

is $\hat{\delta}_{n,1} - \hat{\delta}_{n,j} = \#\mathsf{PSet}_n(\mathcal{A}_{\ge 1}) - \#\mathsf{PSet}_n(\mathcal{A}_{\ge j})$. Assuming the values $\hat{\delta}_{n,j}$ have been precomputed and stored in tables, we can unrank powersets using Algorithm 10. The first step is

Algorithm 10 Unranking of powersets

Unrank$(\mathsf{PSet}(\mathcal{A}), n, i)$
  if $n = 0$ **then return** ϵ **fi**
  $j := 1;$ **while** $\hat{\delta}_{n,1} - \hat{\delta}_{n,j+1} \le i$ **do** $j := j + 1$ **end**
  ▷ The smallest component has size j
  $i' := i - (\hat{\delta}_{n,1} - \hat{\delta}_{n,j})$
  if $j = n$ **then** $k := 1; i'' := i'$
  else $k := 1; c := 0; d := \mathrm{Count}(\mathcal{A}_j^{\{k\}}, kj) \cdot \hat{\delta}_{n-kj,j+1}$
    while $(k+1) \cdot j \le n$ **and** $c + d \le i'$ **do**
      $c := c + d; k := k + 1; d := \mathrm{Count}(\mathcal{A}_j^{\{k\}}, kj) \cdot \hat{\delta}_{n-kj,j+1}$
    end
    $i'' := i' - c$
  fi ▷ There are k components of size j
  $i_\alpha := i''$ **div** $\hat{\delta}_{n-kj,j+1}; i_\beta := (i''$ **mod** $\hat{\delta}_{n-kj,j+1}) + (\hat{\delta}_{n-kj,1} - \hat{\delta}_{n-kj,j+1})$
  ▷ The tail β is recovered as an object in $\mathsf{PSet}(\mathcal{A})$, not in $\mathsf{PSet}(\mathcal{A}_{>j})$;
  ▷ that's why we need to add $(\hat{\delta}_{n-kj,1} - \hat{\delta}_{n-kj,j+1})$
  $\alpha := \mathrm{Unrank}(\mathcal{A}_j^{\{k\}}, kj, i_\alpha);\ \beta := \mathrm{Unrank}(\mathsf{PSet}(\mathcal{A}), n - kj, i_\beta)$
  return $\langle \alpha, \beta \rangle$

to determine the smallest size j among the components of the powerset, then the number k of components of size j (the cardinality of the leading block). It is possible to improve the algorithm using a dichotomic search to find the size j for each leading block. In fact, this improvement will be used in the analysis of the cost (Section 4.2). Moreover, as each leading block of $\alpha \in \mathsf{PSet}(\mathcal{A})$ has components of size larger than the previous one,

$$\alpha = \boxed{\lambda_1 \in \mathcal{A}_{j_1}^{\{k_1\}}}\boxed{\lambda_2 \in \mathcal{A}_{j_2}^{\{k_2\}}}\boxed{\cdots}\boxed{\lambda_l \in \mathcal{A}_{j_l}^{\{k_l\}}}$$

with $j_1 < j_2 < \ldots < j_l$, we could add an additional parameter l to do the linear or dichotomic search from l to n (not 1 to n). So, Algorithm 10 can be replaced by Algorithm 11.

Algorithm 11 Unranking of powersets (version 2)

Unrank($\mathsf{PSet}(\mathcal{A}), n, i, l$)
  if $n = 0$ **then return** ϵ **fi**
  $j := l$; **while** $\hat{\delta}_{n,1} - \hat{\delta}_{n,j+1} \leq i$ **do** $j := j + 1$ **end**
  $\triangleright$ Lines (5) to (13) of Algorithm 10 go here
  $\alpha :=$ Unrank($\mathcal{A}_j^{\{k\}}, kj, i_\alpha$); $\beta :=$ Unrank($\mathsf{PSet}(\mathcal{A}), n - kj, i_\beta, j + 1$)
  return $\langle \alpha, \beta \rangle$

Next, it is necessary to unrank leading blocks $\mathcal{A}_j^{\{k\}}$ in R-order

$$\mathcal{A}_j^{\{k\}} = \{\alpha \in \mathcal{A}_j\} \times (\mathcal{A}_j^{(\succ\alpha)})^{\{k-1\}} \ , \tag{4.2}$$

where $\mathcal{A}^{(\succ\alpha)} = \{\alpha' \in \mathcal{A} \,|\, \alpha' \succ \alpha\}$, that is, the subset of $\mathcal{A}$-objects whose size is strictly larger than $|\alpha|$ or their rank is larger than the rank of α[2]; hence, $\mathcal{A}_j^{(\succ\alpha)} = \{\alpha' \in \mathcal{A}_j \,|\, \alpha' \succ_{\mathcal{A}_j} \alpha\}$. The components in $\mathcal{A}_j^{\{k\}}$ are computed in ascending order by rank, so the algorithm finds the rank of the smallest component and the remaining part is recursively done. The algorithm is based in the fact that there are $\binom{a_j-1}{k-1}$ powersets with the smallest component of rank 0, there are $\binom{a_j-2}{k-1}$ powersets with the smallest component of rank 1, there are $\binom{a_j-3}{k-1}$ powersets with the smallest component of rank 2, and so on. Once the rank m of the smallest component has been computed, the other components $L[l]$ are found in the same way (see Algorithm 12). We note that after each loop of lines (6) to (10) the current rank $L[l]$ is the computed rank m plus the previously computed rank $L[l-1]$ (see line (13)) because components are in ascending order. The linear search to find m has cost $\mathrm{O}(a_j)$, but since $\sum_{m=1}^{l} \binom{a_j-m}{k-1} = \binom{a_j}{k} - \binom{a_j-l}{k}$ we can determine m using dichotomic search, with average cost $\mathrm{O}(\log a_j)$.

We consider now the unranking of unlabelled (multi)sets, according to LScR order,

$$\mathsf{Set}(\mathcal{A}) = \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \mathcal{A}_j^{\{\!\{k\}\!\}} \times \mathsf{Set}_{n-kj}(\mathcal{A}_{>j}) \ . \tag{4.3}$$

Let $\delta_{n,j}$ be the cardinality of sets of size n with the smallest component of size at least j; that is, the cardinality of

$$\mathsf{Set}_n(\mathcal{A}_{\geq j}) = \mathsf{Set}_n(\mathcal{A}_{\geq j+1}) + \bigcup_{1 \leq k \leq \lfloor n/j \rfloor} \mathcal{A}_j^{\{\!\{k\}\!\}} \times \mathsf{Set}_{n-kj}(\mathcal{A}_{\geq j+1}) \ ,$$

where $\mathsf{Set}_n(\mathcal{A}_{\geq j}) = \emptyset$ if $j > n > 0$ and $\mathsf{Set}_0(\mathcal{A}_{\geq j}) = \epsilon$ if $j \geq 0$. Then, we have $\delta_{0,j} = 1$ for all $j \geq 0$ and for $1 \leq n \leq N$ the recurrence

$$\delta_{n,j} \ = \ \begin{cases} 0 & \text{if } j > n, \\ \delta_{n,j+1} + \sum_{1 \leq k \leq \lfloor n/j \rfloor} \binom{a_j+k-1}{k} \delta_{n-kj,j+1} & \text{if } j \leq n. \end{cases}$$

[2] We are making an abuse of notation here: we write "$\alpha' \succ \alpha$" instead of "($|\alpha'| > |\alpha|$) or ($|\alpha'| = |\alpha|$ and $\alpha' \succ \alpha$)".

Algorithm 12 Unranking of powersets of k components of size j

1. $\mathrm{Unrank}(\mathcal{A}_j^{\{k\}}, n, i)$
2. **if** $n = 0$ **then return** ϵ **fi**
3. $L[0] := 0; i' := i$
4. **for** $l := 1$ **to** k **do**
5. $m := 1; r := 0; d := \binom{\mathrm{Count}(\mathcal{A},j) - L[l-1] - m}{k-1}$
6. **while** $r + d \leq i'$ **do**
7. $r := r + d; m := m + 1; d := \binom{\mathrm{Count}(\mathcal{A},j) - L[l-1] - m}{k-1}$
8. $\triangleright$ $L[l-1]$ is subtracted because the first $L[l-1]$ ranks
9. $\triangleright$ of $\mathcal{A}_j$ cannot be repeated in the new search
10. **end**
11. $i' := i' - r$
12. $\triangleright$ m is the rank of the current component l
13. $L[l] := m + L[l-1]$
14. $\triangleright$ Components are in ascending order, thus we have to add
15. $\triangleright$ the rank of the previous component $L[l-1]$
16. **end**
17. **for** $l := 1$ **to** k **do** $\alpha_l := \mathrm{Unrank}(\mathcal{A}, j, L[l])$ **end**
18. **return** $\langle \alpha_1, \langle \alpha_2, \langle \ldots, \langle \alpha_{k-1}, \alpha_k \rangle\rangle\rangle\rangle$

Analogously to $\hat{\delta}_{n,j}$, these values can be computed in $\mathrm{O}(N^2 \log N)$ arithmetic operations for all $1 \leq j \leq n \leq N$. Now, the algorithm for sets is almost the same of Algorithm 10 for powersets but using $\delta_{n,j}$ instead of $\hat{\delta}_{n,j}$: Once the smallest size j of the components of sets has been determined, we compute the cardinality k of the leading block. Again, it is also possible to improve such algorithm using a dichotomic search to get the size j.

Finally, we need to unrank sets of k components of size j (leading blocks):

$$\mathcal{A}_j^{\{\!\{k\}\!\}} = \bigcup_{l=k}^{1} \{\alpha^l \in \mathcal{A}_j\} \times (\mathcal{A}_j^{(\succ\alpha)})^{\{\!\{k-l\}\!\}} \ , \tag{4.4}$$

where again $A_j^{(\succ\alpha)} = \{\alpha' \in \mathcal{A}_j \mid \alpha' \succ_{\mathcal{A}_j} \alpha\}$. Comparing with (4.2), it is easy to see that the unranking of sets is similar to Algorithm 12: It is just necessary to use $\binom{x+y-1}{y}$ in lines (5) and (7) instead of $\binom{x}{y}$. In Algorithm 13 we show the final version to unrank unlabelled sets with k components of size j using a dichotomic search to compute the ranks of the components. Moreover, once we have computed the first rank m, it is also possible to check if the next one has the same rank or not with just one comparison (line (6)). For instance, to compute the ranks of the components of the set $\{a, a, a, a, b, b, b\} \equiv \{a^4 b^3\}$ is equivalent to find the rank of a and b and to do $4 + 3 - 1 = 6$ checkings (comparisons).

We can also use the card-lexicographic order to unrank powersets and sets (Subsection 2.4.2). The corresponding algorithms follow the same steps as Algorithm 6 for the labelled case.

Algorithm 13 Unranking of sets of k components of size j

Unrank($\mathcal{A}_j^{\{\!\{k\}\!\}}, n, i$)
    if $n = 0$ **then return** ϵ **fi**
    $L[0] := 0; i' := i$
    for $l := 1$ **to** k **do**
        $r := \binom{\text{Count}(\mathcal{A},j)-L[l-1]-1}{k}$
        if $r \leq i$ **then** $i' := i - r;\ L[l] := L[l-1]$
        else $(m, r) :=$ Dichotomic_Search($\mathcal{A}, j, L, i'$)
            ▷ m is the rank of the current component l
            $i' := i - r;\ L[l] := m + L[l-1]$
            ▷ Components are in ascending order, thus we need
            ▷ to add the rank of the previous component $L[l-1]$
        fi
    end
    $\alpha_1 :=$ Unrank($\mathcal{A}, j, L[1]$)
    for $l := 2$ **to** k **do**
        if $L[l]{=}L[l-1]$ **then** $\alpha_l{:=}\alpha_{l-1}$ **else** $\alpha_l{:=}$Unrank($\mathcal{A}, j, L[l]$) **fi**
    end
    return $\langle \alpha_1, \langle \alpha_2, \langle \ldots, \langle \alpha_{k-1}, \alpha_k \rangle\rangle\rangle\rangle$

4.1.3 Algorithms for Sequences, Powersets and Sets with Restricted Cardinality, and Substitutions

The unranking of sequences with restricted cardinality is based in the isomorphisms of Table 2.5 and it follows the same ideas as in the case of labelled containers with restricted cardinality. For instance, compare Algorithm 14 for unlabelled sequences with Algorithm 5 for labelled sets.

Algorithm 14 Unranking of sequences with restricted cardinalities

Unrank($\mathcal{A}^k, n, i$)
  if $n = 0$ **then return** ϵ **else return** Unrank($\mathcal{A} \star \mathcal{A}^{k-1}, n, i$) **fi**

Unrank($\mathcal{A}^{\leq k}, n, i$)
  if $n = 0$ **then return** ϵ **else return** Unrank($\mathcal{A} \star \mathcal{A}^{\leq k-1}, n, i$) **fi**

Unrank($\mathcal{A}^{\geq k}, n, i$)
  if $n = 0$ **then return** ϵ **else return** Unrank($\mathcal{A} \star \mathcal{A}^{\geq k-1}, n, i$) **fi**

To unrank powersets and sets with fixed cardinality we consider the LScR order of Table 2.5. For powersets we need to define $\hat{\delta}_{n,j}^{\{k\}}$, the number of powersets of size n with k components of size at least j; that is,

$$\hat{\delta}_{n,j}^{\{k\}} = \#\,\mathsf{PSet}_n(\mathcal{A}_{\geq j}, \text{card} = k) = \#(\mathcal{A}_{\geq j}^{\{k\}})_n \ .$$

In particular, $\hat{\delta}^{\{k\}}_{n,1} = \#(\mathcal{A}^{\{k\}}_j)_n$. So, we have $\hat{\delta}^{\{0\}}_{0,j} = 1$ for all $j \geq 0$ and for $n > 0$ the recurrence is

$$\hat{\delta}^{\{h\}}_{n,j} = \begin{cases} 0 & \text{if } j > n, \\ \hat{\delta}^{\{h\}}_{n,j+1} + \sum_{1 \leq l \leq \min\{\lfloor n/j \rfloor, h\}} \binom{a_j}{l} \cdot \hat{\delta}^{\{h-l\}}_{n-lj,j+1} & \text{if } j \leq n, \end{cases}$$

where $1 \leq h \leq \min\{\lfloor n/j \rfloor, k\}$. Thus, it is quite easy to compute these values for $1 \leq j \leq n \leq N$ and $1 \leq h \leq \min\{\lfloor n/j \rfloor, k\}$ in $\mathrm{O}\left(N^2 \min\{\log N, k\}^2\right)$ arithmetic operations:

$$\sum_{1 \leq n \leq N} \sum_{1 \leq j \leq n} \sum_{1 \leq h \leq \min\{\lfloor n/j \rfloor, k\}} \sum_{1 \leq l \leq \min\{\lfloor n/j \rfloor, h\}} \mathrm{O}(1) = \mathrm{O}\left(N^2 \min\{\log N, k\}^2\right) .$$

Assuming that these values are precomputed and stored for later use, the number of powersets with some component of size strictly smaller than j and cardinality k is $\hat{\delta}^{\{k\}}_{n,1} - \hat{\delta}^{\{k\}}_{n,j} = \#(\mathcal{A}^{\{k\}}_{\geq 1})_n - \#(\mathcal{A}^{\{k\}}_{\geq j})_n$. Then, the unranking of $\mathcal{A}^{\{k\}}$ is described in Algorithm 15 which is very similar to Algorithms 10 and 12, and has the same main steps. Moreover, again it is possible the use a dichotomic search to find the size j.

Algorithm 15 Unranking of powersets with fixed cardinality

unrank$(\mathcal{A}^{\{k\}}, n, i)$
  if $n = 0$ **then return** ϵ **fi**
  $j := 1$; **while** $\hat{\delta}^{\{k\}}_{n,1} - \hat{\delta}^{\{k\}}_{n,j+1}$ **do** $j := j + 1$ **end**
  ▷ The smallest component has size j
  $i' := i - (\hat{\delta}^{\{k\}}_{n,1} - \hat{\delta}^{\{k\}}_{n,j})$;
  $p := 1; c := 0; d := \binom{a_j}{p}\ \hat{\delta}^{\{k-1\}}_{n,j+1}$
  while $c + d \leq i$ **and** $p \leq k$ **do**
    $c := c + d; p := p + 1; d := \binom{a_j}{p}\ \hat{\delta}^{\{k-p\}}_{n-pj,j+1}$
  end ▷ There are exactly p components with size j
  $i'' := i' - c$
  $i_\alpha := i''$ **div** $\hat{\delta}^{\{k-p\}}_{n-pj,j+1}$; $i_\beta := (i''$ **mod** $\hat{\delta}^{\{k-p\}}_{n-pj,j+1}) + (\hat{\delta}^{\{k-p\}}_{n-pj,0} - \hat{\delta}^{\{k-p\}}_{n-pj,j+1})$
  ▷ The tail β is unranked in $\mathcal{A}^{\{k-p\}}$, not in $\mathcal{A}^{\{k-p\}}_{>j}$;
  ▷ that's why we add $(\hat{\delta}^{\{k-p\}}_{n-pj,0} - \hat{\delta}^{\{k-p\}}_{n-pj,j+1}) = \#((\mathcal{A}^{\{k-p\}}_{\leq j})_{n-pj})$
  $\alpha :=$ Unrank$(\mathcal{A}^{\{p\}}_j, pj, i_\alpha)$; $\beta :=$ Unrank$(\mathcal{A}^{\{k-p\}}, n - pj, i_\beta)$
  return $\langle \alpha, \beta \rangle$

Similar ideas are used to solve the unranking of $\mathcal{A}^{\{\leq k\}}$ and $\mathcal{A}^{\{\geq k\}}$. Finally, the unranking of sets with fixed cardinality is the same as powersets (Algorithm 15) but using $\binom{x+y-1}{y}$ instead of $\binom{x}{y}$ because the repetitions are allowed. The unranking of $\mathcal{A}^{\{\!\{\leq k\}\!\}}$ and $\mathcal{A}^{\{\!\{\geq k\}\!\}}$ follows analogous arguments.

Substitution of $\mathcal{B}$ into $\mathcal{A}$ is performed on the basis of unions and Cartesian products: $\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^k$. First we determine the size k of the $\mathcal{A}$-object (which coincides with the cardinality of the set of components from $\mathcal{B}$), and then we determine the ranks of the $\mathcal{A}$-object and of the set of k $\mathcal{B}$-objects, like for ordinary products (Algorithm 16).

Algorithm 16 Unranking of substitutions

unrank($\mathcal{A} \circ \mathcal{B}, n, i$)
  $c := 0; k := 0; d :=$ Count($\mathcal{A}, k$) $\cdot$ Count($\mathcal{B}^k, n$)
  while $c + d \leq i$ **do** $c := c + d; k := k + 1; d :=$ Count($\mathcal{A}, k$) $\cdot$ Count($\mathcal{B}^k, n$) **end**
  $i' := i - c;\ b :=$ Count($\mathcal{B}^k, n$)
  $\alpha :=$ Unrank($\mathcal{A}, k, i'$ **div** b); $\beta :=$ Unrank($\mathcal{B}^k, n, i'$ **mod** b)
  return $\circ\langle \alpha, \beta \rangle$

4.2 Analysis of the Cost

This section is divided in three subsections. In the first one we establish some general results on the average cost of the unlabelled unranking algorithms. In the second one we study the cumulated cost and we give a set of Υ-rules in the spirit of Subsection 3.2.2. Finally, in the last subsection, we use the Υ-rules (Theorem 19 and Corollary 5) to compute the average cost of unranking for some particular examples (binary trees, sequences, k-shuffles, etc.).

4.2.1 Performance of the Unranking Algorithms

The same arguments given for the labelled case (Theorems 9, 10 and 11) prove the following theorems.

Theorem 16. *The worst-case time complexity of unranking for objects of size n in any admissible unlabelled class $\mathcal{A}$ (classes that only involve disjoint union, standard product, sequence, powerset, set and substitution operators) using lexicographic ordering is* $\mathrm{O}(n^2)$ *arithmetic operations.*

Theorem 17. *The worst-case time complexity of unranking for objects of size n in any admissible unlabelled class $\mathcal{A}$ using boustrophedonic ordering is* $\mathrm{O}(n \log n)$ *arithmetic operations.*

Theorem 18. *The cost of unranking an object of size n for any iterative unlabelled class $\mathcal{A}$ is $\Theta(n)$.*

4.2.2 A Calculus for the Average Cost of Unranking

As we defined for labelled classes (Section 3.2), the ordinary generating function of the cumulated costs of unranking all objects of $\mathcal{A}$ is

$$\Upsilon\mathcal{A}(z) = \sum_{\alpha \in \mathcal{A}} \mathrm{cu}(\alpha, \mathcal{A})\, z^{|\alpha|} = \sum_{n \geq 0} \Upsilon\mathcal{A}_n\, z^n \ ,$$

where $\mathrm{cu}(\alpha, \mathcal{A})$ is the cost of *unrank*$(\mathcal{A}, n, i)$ with α being the object of rank i in $\mathcal{A}_n$. Then, the average cost is $\mu_{n,\mathcal{A}} = [z^n]\Upsilon\mathcal{A}(z)/[z^n]A(z) = \Upsilon\mathcal{A}_n/a_n$.

The Υ-rules for elementary classes, unions and products follow from the same arguments as in the labelled case. First of all, we have that $\Upsilon\epsilon = \Upsilon Z = 0$. For unions we have that

$$\Upsilon(\mathcal{A} + \mathcal{B}) = \sum_{\gamma \in \mathcal{A}+\mathcal{B}} \mathrm{cu}(\gamma, \mathcal{A} + \mathcal{B})\, z^{|\gamma|} = \sum_{\alpha \in \mathcal{A}} \mathrm{cu}(\alpha, \mathcal{A})\, z^{|\alpha|} + \sum_{\beta \in \mathcal{B}} \mathrm{cu}(\beta, \mathcal{B})\, z^{|\beta|} = \Upsilon\mathcal{A} + \Upsilon\mathcal{B}\ .$$

For products $\mathcal{A} \times \mathcal{B}$, we compute the size of the first component with cost proportional to that size and, afterwards, we unrank the $\mathcal{A}$- and $\mathcal{B}$-components, recursively. Thus

$$\begin{aligned}\Upsilon(\mathcal{A} \times \mathcal{B}) &= \sum_{\gamma \in \mathcal{A}\times\mathcal{B}} \mathrm{cu}(\gamma, \mathcal{A} \times \mathcal{B})\, z^{|\gamma|} = \sum_{\langle\alpha,\beta\rangle \in \mathcal{A}\times\mathcal{B}} (|\alpha| + \mathrm{cu}(\alpha, \mathcal{A}) + \mathrm{cu}(\beta, \mathcal{B}))\ z^{|\alpha|+|\beta|} \\ &= \vartheta A \cdot B + \Upsilon\mathcal{A} \cdot B + A \cdot \Upsilon\mathcal{B}\ .\end{aligned}$$

As it was the case for labelled sequences (Corollary 3), the rule for sequences comes from the isomorphism (2.6) and the previous Υ-rules:

$$\begin{aligned}\Upsilon\mathsf{Seq}(\mathcal{A}) = \Upsilon(\epsilon + \mathcal{A} \times \mathsf{Seq}(\mathcal{A})) &= \frac{\vartheta A}{1-A} + \frac{\Upsilon\mathcal{A}}{1-A} + A \cdot \Upsilon\mathsf{Seq}(\mathcal{A}) \\ \Rightarrow \Upsilon\mathsf{Seq}(\mathcal{A}) &= \frac{\vartheta A + \Upsilon\mathcal{A}}{(1-A)^2}\ .\end{aligned}$$

For powersets we consider the isomorphisms (4.1) and (4.2):

$$\begin{aligned}\mathsf{PSet}(\mathcal{A}) &= \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{\lambda \in \mathcal{A}_j^{\{k\}}} \bigcup_{\tau \in \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})} \lambda \times \tau \\ &= \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{\alpha \in \mathcal{A}_j} \bigcup_{\beta \in (\mathcal{A}_j^{(\succ\alpha)})^{\{k-1\}}} \bigcup_{\tau \in \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})} \alpha \times \beta \times \tau\ .\end{aligned}$$

with $\mathcal{A}_j^{(\succ\alpha)} = \{\alpha' \in \mathcal{A}_j \mid \alpha' \succ_{\mathcal{A}_j} \alpha\}$. We can write

$$\Upsilon\mathsf{PSet}(\mathcal{A}) = \Upsilon_1(z) + \Upsilon_2(z) + \Upsilon_3(z)$$

where Υ_1 collects the contribution to the cost of computing the sizes of leading blocks, Υ_2 accumulates the cost of computing the cardinality of leading blocks, and Υ_3 the contribution to find the ranks of each component and making the corresponding unranks.

To study the contribution of Υ_1, we consider a powerset $\gamma \in \mathsf{PSet}(\mathcal{A})$ as a set of blocks where each block λ_i is a maximal subset with components of identical size j_i. That is, each object $\gamma \in \mathsf{PSet}(\mathcal{A})$ is decomposed into a block of components with minimum size (its leading block), and the remaining part is recursively decomposed in the same way.

$$\gamma = \begin{array}{|c|c|c|c|}\hline \lambda_1 \in \mathsf{PSet}(\mathcal{A}_{j_1}) & \lambda_2 \in \mathsf{PSet}(\mathcal{A}_{j_2}) & \cdots & \lambda_l \in \mathsf{PSet}(\mathcal{A}_{j_l}) \\ \hline\end{array}$$

with $j_1 < j_2 < \ldots < j_l$. So, the cost of computing the sizes of blocks (each block will be the leading block in some recursive call to unrank) is

$$\begin{aligned}\Upsilon_1(z) &= \sum_{l>0} \sum_{j_1<j_2<\ldots<j_l} \sum_{\lambda_1 \in \mathsf{PSet}(\mathcal{A}_{j_1})} j_1\, z^{|\lambda_1|} \sum_{\lambda_2 \in \mathsf{PSet}(\mathcal{A}_{j_2})} j_2\, z^{|\lambda_2|} \cdots \sum_{\lambda_l \in \mathsf{PSet}(\mathcal{A}_{j_l})} j_l\, z^{|\lambda_l|} \\ &= \sum_{j>0} \sum_{\tilde\lambda \in \mathsf{PSet}(\mathcal{A}_j)} j\, z^{|\tilde\lambda|} \sum_{\tilde\tau \in \mathsf{PSet}(\mathcal{A} \setminus \mathcal{A}_j)} z^{|\tilde\tau|}\ ,\end{aligned}$$

which is clearly upper bounded by the cost of computing the sizes of all components:

$$\Upsilon_1(z) \ll \sum_{\alpha\in\mathcal{A}} |\alpha|\, z^{|\alpha|} \sum_{\gamma\in\mathsf{PSet}(\mathcal{A}\setminus\{\alpha\})} z^{|\gamma|} \ . \tag{4.5}$$

To analyze the contribution of Υ_2, we consider again the decomposition of γ into leading blocks, but now we care about their cardinalities. So, if k_i is the cardinality of λ_i we have

$$\gamma = \underbrace{\begin{array}{|c|c|c|c|}\hline \lambda_1 \in \mathcal{A}_{j_1}^{\{k_1\}} & \lambda_2 \in \mathcal{A}_{j_2}^{\{k_2\}} & \cdots & \lambda_l \in \mathcal{A}_{j_l}^{\{k_l\}} \\ \hline \end{array}}_{\mathrm{card}=k_1+k_2+\cdots+k_3=k}$$

Now, we can write

$$\Upsilon_2(z) = \sum_{k>0} \sum_{\gamma\in\mathcal{A}^{\{k\}}} k\, z^{|\gamma|} = \sum_{\alpha\in\mathcal{A}} z^{|\alpha|} \sum_{\gamma\in\mathsf{PSet}(\mathcal{A}\setminus\{\alpha\})} z^{|\gamma|} \tag{4.6}$$

because powersets do not allow repeated components. Finally, assuming that we use dichotomic search to locate ranks, we have that Υ_3 is

$$\Upsilon_3(z) = \sum_{\alpha\in\mathcal{A}} (\log_2 a_{|\alpha|} + \mathrm{cu}(\alpha,\mathcal{A}))\, z^{|\alpha|} \sum_{\gamma\in\mathsf{PSet}(\mathcal{A}\setminus\{\alpha\})} z^{|\gamma|} \tag{4.7}$$

where we use now the decomposition

$$\gamma= \begin{array}{|c|c|c|}\hline \mathsf{PSet}(\mathcal{A}\setminus\mathcal{A}^{(\succ\alpha)}) & \alpha\in\mathcal{A} & \mathsf{PSet}(\mathcal{A}^{(\succ\alpha)}) \\ \hline \end{array}$$

Thus, adding (4.5), (4.6) and (4.7) we get the following upper bound for $\Upsilon\mathsf{PSet}(\mathcal{A})$

$$\begin{aligned}\Upsilon\mathsf{PSet}(\mathcal{A}) &\ll \sum_{\alpha\in\mathcal{A}} (|\alpha| + 1 + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha,\mathcal{A}))\, z^{|\alpha|} \sum_{\gamma\in\mathsf{PSet}(\mathcal{A}\setminus\{\alpha\})} z^{|\gamma|} \\ &= \sum_{\alpha\in\mathcal{A}} (|\alpha| + 1 + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha,\mathcal{A}))\, z^{|\alpha|} \frac{\mathsf{PSet}(A)}{1+z^{|\alpha|}}\end{aligned}$$

where $\mathsf{PSet}(A) = \exp\left(\sum_{k>0}(-1)^{k-1}\frac{A(z^k)}{k}\right)$. Now, thanks to the equality

$$\sum_{\alpha\in\mathcal{A}} \frac{z^{|\alpha|}}{1+z^{|\alpha|}} = \sum_{\alpha\in\mathcal{A}} z^{|\alpha|} \sum_{k\geq 0} (-1)^k z^{k\,|\alpha|} = \sum_{\alpha\in\mathcal{A}} \sum_{k>0} (-1)^{k+1} z^{k\,|\alpha|} = \hat{\Delta}\, A \ ,$$

we have that

$$\Upsilon\mathsf{PSet}(\mathcal{A}) \ll \hat{\Delta}(\vartheta A + A + \mathrm{L}A + \Upsilon\mathcal{A}) \cdot \mathsf{PSet}(A) = \vartheta\mathsf{PSet}(A) + \hat{\Delta}(A + \mathrm{L}A + \Upsilon\mathcal{A}) \cdot \mathsf{PSet}(A)$$

where $\mathrm{L}A = \sum_{\alpha\in\mathcal{A}} \log_2 a_{|\alpha|}\, z^{|\alpha|} = \sum_{n\geq 0} a_n \log_2 a_n\, z^n$. We assume that $a_n \log_2 a_n = 0$ if $a_n = 0$.

For sets, the unranking is based in the isomorphisms (4.3) and (4.4):

$$\begin{aligned}
\mathsf{Set}(\mathcal{A}) &= \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{\lambda \in \mathcal{A}_j^{\{\!\{k\}\!\}}} \bigcup_{\tau \in \mathsf{Set}_{n-kj}(\mathcal{A}_{>j})} \lambda \times \tau \\
&= \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \bigcup_{l=k}^{1} \bigcup_{\alpha \in \mathcal{A}_j} \bigcup_{\beta \in (\mathcal{A}_j^{(\succ\alpha)})^{\{\!\{k-l\}\!\}}} \bigcup_{\tau \in \mathsf{Set}_{n-kj}(\mathcal{A}_{>j})} \alpha^l \times \beta \times \tau \,,
\end{aligned}$$

where again $\mathcal{A}_j^{(\succ\alpha)} = \{\alpha' \in \mathcal{A}_j \mid \alpha' \succ_{\mathcal{A}_j} \alpha\}$. We can decompose $\Upsilon\mathsf{Set}(\mathcal{A})$ in three parts, as we have done for powersets. Υ_1 collects the contribution to the cost of computing the size of leading blocks, Υ_2 accumulates the cost of computing the cardinality of leading blocks, and Υ_3 is the cost of computing how many times each component appears (that is, the cardinality l), the rank of each component (assuming the dichotomic search) and making the corresponding unrank:

$$\begin{aligned}
\Upsilon_1(z) &= \sum_{j>0} \sum_{\tilde\lambda \in \mathsf{Set}(\mathcal{A}_j)} j\, z^{|\tilde\lambda|} \sum_{\tilde\tau \in \mathsf{Set}(\mathcal{A} \setminus \mathcal{A}_j)} z^{|\tilde\tau|} \ll \sum_{k>0} \sum_{\alpha \in \mathcal{A}} |\alpha|\, z^{k|\alpha|} \sum_{\gamma \in \mathsf{Set}(\mathcal{A} \setminus \{\alpha\})} z^{|\gamma|} \,, && (4.8) \\
\Upsilon_2(z) &= \sum_{k>0} \sum_{\gamma \in \mathcal{A}^{\{\!\{k\}\!\}}} k\, z^{|\gamma|} = \sum_{k>0} \sum_{\alpha \in \mathcal{A}} k\, z^{k|\alpha|} \sum_{\gamma \in \mathsf{Set}(\mathcal{A} \setminus \{\alpha\})} z^{|\gamma|}
\end{aligned}$$

and

$$\begin{aligned}
\Upsilon_3(z) &= \sum_{l>0} \sum_{\alpha \in \mathcal{A}} (l + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{l|\alpha|} \sum_{\gamma \in \mathsf{Set}(\mathcal{A} \setminus \{\alpha\})} z^{|\gamma|} \\
&= \sum_{k>0} \sum_{\gamma \in \mathcal{A}^{\{\!\{k\}\!\}}} k\, z^{|\gamma|} + \left(\sum_{l>0} \sum_{\alpha \in \mathcal{A}} (\log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{l|\alpha|} \right) \sum_{\gamma \in \mathsf{Set}(\mathcal{A} \setminus \{\alpha\})} z^{|\gamma|} \\
&= \Upsilon_2(z) + \left(\sum_{l>0} \sum_{\alpha \in \mathcal{A}} (\log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{l|\alpha|} \right) \sum_{\gamma \in \mathsf{Set}(\mathcal{A} \setminus \{\alpha\})} z^{|\gamma|} \,.
\end{aligned}$$

Therefore, an upper bound for $\Upsilon\mathsf{Set}(\mathcal{A})$ is:

$$\begin{aligned}
\Upsilon\mathsf{Set}(\mathcal{A}) &\ll \sum_{k>0} \sum_{\alpha \in \mathcal{A}} (|\alpha| + 2k + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{k|\alpha|} \sum_{\gamma \in \mathsf{Set}(\mathcal{A} \setminus \{\alpha\})} z^{|\gamma|} \\
&= \sum_{k>0} \sum_{\alpha \in \mathcal{A}} (|\alpha| + 2k + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{k|\alpha|} \cdot \mathsf{Set}(A) \cdot (1 - z^{|\alpha|}) \\
&= \Bigg(\sum_{k>0} \sum_{\alpha \in \mathcal{A}} (|\alpha| + 2k + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{k|\alpha|} - \\
&\qquad \sum_{k>0} \sum_{\alpha \in \mathcal{A}} (|\alpha| + 2k + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A}))\, z^{(k+1)|\alpha|} \Bigg) \mathsf{Set}(A) \,,
\end{aligned}$$

where $\mathsf{Set}(A) = \exp\left(\sum_{k>0} \frac{A(z^k)}{k}\right)$. Now we note that

$$\sum_{k>0} \sum_{\alpha \in \mathcal{A}} k\, z^{k|\alpha|} - \sum_{k>0} \sum_{\alpha \in \mathcal{A}} k\, z^{(k+1)|\alpha|} = \sum_{k>0} \sum_{\alpha \in \mathcal{A}} z^{k|\alpha|} \,,$$

thus

$$\begin{aligned}\Upsilon\mathsf{Set}(\mathcal{A}) &\ll \sum_{\alpha\in\mathcal{A}} (|\alpha| + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha,\mathcal{A}))\, z^{|\alpha|}\, \mathsf{Set}(A) + 2\sum_{k>0}\sum_{\alpha\in\mathcal{A}} z^{k|\alpha|}\, \mathsf{Set}(A)\\ &= (\vartheta A + \mathrm{L}A + \Upsilon\mathcal{A} + 2\Delta A)\, \mathsf{Set}(A)\ .\end{aligned}$$

We cast the results obtained so far in the following theorem.

Theorem 19. *Let $\mathcal{A}$ and $\mathcal{B}$ be two admissible unlabelled classes such that $\epsilon \notin \mathcal{A}$, with the previous considerations, then*

1. $\Upsilon\epsilon = \Upsilon Z = 0$,
2. $\Upsilon(\mathcal{A}+\mathcal{B}) = \Upsilon\mathcal{A} + \Upsilon\mathcal{B}$,
3. $\Upsilon(\mathcal{A}\times\mathcal{B}) = \vartheta A\cdot B + \Upsilon\mathcal{A}\cdot B + A\cdot\Upsilon\mathcal{B}$,
4. $\Upsilon\mathit{Seq}(\mathcal{A}) = \frac{\vartheta A(z)+\Upsilon\mathcal{A}}{(1-A(z))^2}$,
5. $\Upsilon\mathit{PSet}(\mathcal{A}) \ll \vartheta\mathit{PSet}(A)\ +\ \hat{\Delta}(A + \mathrm{L}A + \Upsilon\mathcal{A})\cdot\mathit{PSet}(A)$,
6. $\Upsilon\mathit{Set}(\mathcal{A}) \ll (\vartheta A + 2\Delta A + \mathrm{L}A + \Upsilon\mathcal{A})\cdot\mathit{Set}(A)$,

where $\mathit{PSet}(A) = \exp\left(\sum_{n>0}(-1)^{n-1}\frac{A(z^n)}{n}\right)$, $\mathit{Set}(A) = \exp\left(\sum_{n>0}\frac{A(z^n)}{n}\right)$, and the operators ϑ, Δ, $\hat{\Delta}$ and L applied to a generating function $C(z) = \sum_{n\geq 0} c_n\, z^n$ are $\vartheta C(z) = z\frac{d}{dz}C(z)$, $\Delta C(z) = \sum_{n>0} C(z^n)$, $\hat{\Delta}C(z) = \sum_{n>0}(-1)^{n-1}\ C(z^n)$ and $\mathrm{L}C = \sum_{n\geq 0} c_n\, \log_2 c_n\, z^n$, respectively.

Now, we consider restricted cardinalities and substitutions.

Corollary 5. *Let $\mathcal{A}$ be a combinatorial class such that $\epsilon\notin\mathcal{A}$. Assuming the previous notation and $k>0$, then*

1. $\Upsilon\mathcal{A}^k = k\ A^{k-1}\ (\vartheta A + \Upsilon\mathcal{A})$,
   $\Upsilon\mathcal{A}^{\leq k} = kA^{\leq k-1}\ (\vartheta A + \Upsilon\mathcal{A})$,
   $\Upsilon\mathcal{A}^{\geq k} = kA^{\geq k-1}\ (\vartheta A + \Upsilon\mathcal{A}) + A^k\cdot\Upsilon\mathit{Seq}(\mathcal{A})$,
2. $\Upsilon\mathcal{A}^{\{P(k)\}} \ll \vartheta A^{\{P(k-1)\}} + \hat{\Delta}\,(A + \mathrm{L}A + \Upsilon\mathcal{A})\cdot A^{\{P(k-1)\}}$,
3. $\Upsilon\mathcal{A}^{\{\!\{P(k)\}\!\}} \ll (\vartheta A + 2\Delta A + \mathrm{L}A + \Upsilon\mathcal{A})\cdot A^{\{\!\{P(k-1)\}\!\}}$,
4. $\Upsilon(\mathcal{A}\circ\mathcal{B}) = (\vartheta A + \Upsilon\mathcal{A})(B) + (\vartheta B + \Upsilon\mathcal{B})\cdot A'(B)$,

where $P(k)$ stands for any of $\leq k$, k or $\geq k$.

Proof. Sequences with fixed cardinality can be unranked as in the labelled case: Assuming that $\mathcal{A}^1 = \mathcal{A} \times \mathcal{A}^0$, where $\mathcal{A}^0 = \epsilon$, we have

$$\begin{aligned}\Upsilon\mathcal{A}^k &= \Upsilon(\mathcal{A} \times \mathcal{A}^{k-1}) = \vartheta A \cdot A^{k-1} + \Upsilon\mathcal{A} \cdot A^{k-1} + A \cdot \Upsilon\mathcal{A}^{k-1} = \dots \\ &= (k-1)(\vartheta A + \Upsilon\mathcal{A}) \cdot A^{k-1} + A^{k-1} \cdot \Upsilon\mathcal{A}^1 \\ &= (k-1)(\vartheta A + \Upsilon\mathcal{A}) \cdot A^{k-1} + A^{k-1} \cdot \Upsilon(\mathcal{A} \times \mathcal{A}^0) = kA^{k-1}(\vartheta A + \Upsilon\mathcal{A}) \ .\end{aligned}$$

For powersets with fixed cardinality we have

$$\begin{aligned}\Upsilon\mathcal{A}^{\{k\}} &\ll \sum_{\alpha\in\mathcal{A}} \big(|\alpha| + 1 + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A})\big)\, z^{|\alpha|} \sum_{\substack{\gamma\in\mathcal{A}^{\{k-1\}}\\ \alpha\notin\gamma}} z^{|\gamma|} \\ &= \sum_{\alpha\in\mathcal{A}} \big(|\alpha| + 1 + \log_2 a_{|\alpha|} + \mathrm{cu}(\alpha, \mathcal{A})\big)\, z^{|\alpha|} \frac{A^{\{k-1\}}}{1 + z^{|\alpha|}} \\ &= \hat{\Delta}\,(\vartheta A + A + \mathrm{L}A + \Upsilon\mathcal{A}) \cdot A^{\{k-1\}} = \vartheta A^{\{k-1\}} + \hat{\Delta}\,(A + \mathrm{L}A + \Upsilon\mathcal{A}) \cdot A^{\{k-1\}} \ .\end{aligned}$$

The other equalities for powersets and sets with restricted cardinality follow similar ideas.

Finally, the rule for substitutions is easily obtained:

$$\begin{aligned}\Upsilon(\mathcal{A} \circ \mathcal{B}) &= \Upsilon\left(\bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^k\right) = \sum_{k>0} \vartheta A_k \cdot B^k + \sum_{k>0} \Upsilon\mathcal{A}_k \cdot B^k + \sum_{k>0} A_k \cdot \Upsilon\mathcal{B}^k \\ &= (\vartheta\mathcal{A} + \Upsilon\mathcal{A})(B) + \sum_{k>0} A_k\ k\ B^{k-1}\ (\vartheta B + \Upsilon\mathcal{B}) \\ &= (\vartheta\mathcal{A} + \Upsilon\mathcal{A})(B) + (\vartheta B + \Upsilon\mathcal{B}) \cdot A'(B) \ .\end{aligned}$$

□

If c_n is at most $\rho^n \cdot n^\alpha$, for some ρ and α then

$$\log_2 c_n \le \log_2(\rho^n \cdot n^\alpha) = n \log_2 \rho + \alpha \log_2 n \sim n \log_2 \rho \ .$$

Thus, the contribution of LC has asymptotically the same order of magnitude as that of ϑC. Moreover, similar arguments for the operators Δ and $\hat{\Delta}$ allow us to prove the following lemma.

Lemma 4. *If $a_n = \mathrm{O}(\rho^n\, n^\alpha)$ for some ρ and α*

1. $\Upsilon \mathsf{PSet}(\mathcal{A}) \ll \vartheta \mathsf{PSet}(A) + \Upsilon\mathcal{A} \cdot \mathsf{PSet}(A)$.
2. $\Upsilon \mathsf{Set}(\mathcal{A}) \ll (\vartheta A + \Upsilon\mathcal{A}) \cdot \mathsf{Set}(A)$.
3. $\Upsilon\mathcal{A}^{\{P(k)\}} \ll \vartheta A^{\{P(k-1)\}} + \Upsilon\mathcal{A} \cdot A^{\{P(k-1)\}}$.
4. $\Upsilon\mathcal{A}^{\{\!\{P(k)\}\!\}} \ll (\vartheta A + \Upsilon\mathcal{A}) \cdot A^{\{\!\{P(k-1)\}\!\}}$.

Finally, if we use the dichomotic search to find j for each leading block λ_i, the corresponding contribution of (4.5) is bounded by

$$\Upsilon_1(z) = \sum_{l>0} \sum_{j_1<\ldots<j_l} \sum_{\lambda_1\in\mathsf{PSet}(\mathcal{A}_{j_1})} \cdots$$

$$\sum_{\lambda_l \in \mathsf{PSet}(\mathcal{A}_{j_l})} \Big[\log_2(|\lambda_1| + \cdots + |\lambda_l|) + \cdots + \log_2|\lambda_l|\Big]\, z^{|\lambda_1|+\cdots+|\lambda_k|}$$
$$\ll \sum_{\gamma \in \mathsf{PSet}(\mathcal{A})} \mathrm{B}(\gamma)\, \log_2|\gamma|\, z^{|\gamma|}\,,$$

where $\mathrm{B}(\gamma)$ is the number of (leading) blocks of γ. The same argument can also be applied for sets in (4.8).

4.3 Examples

4.3.1 Sets with Different Atoms

Let $\mathcal{B}$ be a class of sets built from I different atoms, that is, $\mathcal{B} = \mathsf{Set}(\mathcal{A})$ where $\mathcal{A} = \cup_{i=1}^{I} Z_i$. Then $A(z) = I \cdot z$, $\Upsilon\mathcal{A} = 0$ and $B(z) = 1/(1-z)^I$. The cumulated cost of unranking all $\mathcal{A}$-sets is

$$\Upsilon\mathcal{B} = \Upsilon_1\mathcal{B} + \Upsilon_2\mathcal{B} + \Upsilon_3\mathcal{B} \ll B + \vartheta B + (\vartheta B + \mathrm{L}A \cdot B) = \frac{2\,I\,z}{(1-z)^{I+1}} + \frac{1 + I\,\log I}{(1-z)^I}\,.$$

That means the average cost is linear, $\mu_{n,\mathcal{B}} \le 2\,n + I\,(2 + \log I) + 1$.

4.3.2 Binary and Motzkin Trees

For unlabelled binary and Motzkin trees we use the same rules as for their labelled counterparts. So, the average cost is given by equations (3.8) and (3.9), respectively. If we consider sequences of binary trees, we get $\Upsilon\mathsf{Seq}(\mathcal{B}) = 2z/(1-4z) + \mathrm{O}((1-4z)^{-1/2})$. So, the average cost of unranking sequences of binary trees has the same behaviour as the average cost of unranking binary trees.

4.3.3 k-Shuffles

Let a and b be two atoms, we say that their *k-shuffle* is the set of sequences of a's and b's that contains exactly k b's. We denote this class by $\mathcal{L}_k$ and it can be defined by the following specification

$$\mathcal{L}_k = \begin{cases} \mathsf{Seq}(a) & \text{if } k = 0, \\ \mathcal{L}_0 \times (b \times \mathcal{L}_{k-1}) & \text{if } k > 0. \end{cases}$$

In general, it is clear that $[z^n]L_k(z) = [z^n]z^k/(1-z)^{k+1} = \binom{n}{k} \sim n^k/k!$ Moreover, $\Upsilon\mathcal{L}_0 = z/(1-z)$ and, by induction, it is easy to prove that

$$\Upsilon\mathcal{L}_k = \vartheta L_0\, z\, L_{k-1} + \Upsilon\mathcal{L}_0\, z\, L_{k-1} + L_0\, \Upsilon(b \times \mathcal{L}_{k-1}) = \frac{(2k+1)z^{k+1}}{(1-z)^{k+2}} + \frac{k\, z^k}{(1-z)^{k+1}}\,.$$

Thus, the average cost is linear on n and k: $\mu_{n,\mathcal{L}_k} = (2+k)\,n + \mathrm{o}(n)$.

On the other hand, it is also possible to consider k-shuffles defined by the specification

$$\mathcal{L}_k = \begin{cases} \mathsf{Seq}(a) & \text{if k=0}, \\ (\mathcal{L}_0 \times b) \times \mathcal{L}_{k-1} & \text{if } k > 0. \end{cases}$$

Again $\Upsilon\mathcal{L}_0 = z/(1-z)$, but now by induction we have that

$$\Upsilon\mathcal{L}_k = \vartheta(L_0 \cdot z)\ L_{k-1} + \Upsilon(\mathcal{L}_0 \times b)\ L_{k-1} + L_0 \cdot z\ \Upsilon\mathcal{L}_{k-1} = \frac{k\ z^k}{(1-z)^{k+1}} + \frac{(3k+1)z^{k+1}}{(1-z)^{k+2}}\ .$$

Thus, now the average cost is linear on n and k, $\mu_{n,\mathcal{L}_k} = (3k+1)\,n + \mathrm{o}(n)$, but the multiplying factor is larger.

4.3.4 Sequences of Z

The performance of the unranking can be largely affected by the specification which we use for the class to be unranked. This is clearly shown by the following simple example.

Let $\mathcal{S}_1$ be the sequences of Z defined by $\mathcal{S}_1 = \epsilon + Z \times \mathcal{S}_1$. Then $\Upsilon\mathcal{S}_1 = \Upsilon\epsilon + \vartheta z\, S_1(z) + \Upsilon Z\, S_1(z) + z\,\Upsilon\mathcal{S}_1$, i.e., $\Upsilon\mathcal{S}_1 = z/(1-z)^2$, and $\mu_{\mathcal{S}_1,n} = n + \mathrm{o}(n)$.

However, if the specification $\mathcal{S}_2 = \epsilon + \mathcal{S}_2 \times Z$ is considered, then the average cost is quadratic, since $\Upsilon\mathcal{S}_2 = \Upsilon\epsilon + \vartheta S_2(z)\, z + \Upsilon\mathcal{S}_2\, z + S_2(z)\,\Upsilon Z$, i.e., $\Upsilon\mathcal{S}_2 = z^2/(1-z)^3$, and $\mu_{\mathcal{S}_2,n} = n^2 + \mathrm{o}(n^2)$.

So, for the unlabelled case, it makes sense to look for alternative specifications of a given class that improve the performance of unranking just as it did for the labelled case. For instance, the big-endian heuristic, i.e., using $\mathcal{A} \times \mathcal{B}$ if $\mathcal{A} \lll \mathcal{B}$ and $\mathcal{B} \times \mathcal{A}$ if $\mathcal{B} \lll \mathcal{A}$ (see Section 3.4), may yield significant improvements in the performance. See, for instance, the last two examples of this subsection, where the use of this heuristic provides important savings.

Chapter 5

An Experimental Study of Unranking Algorithms

We describe in this chapter the experiments that we have conducted in order to fullfil the following goals:

a) to obtain a working implementation of the unranking algorithms that we have presented in Chapters 3 and 4;

b) to assess the validity and range of application of our theoretical analysis of the performance of these algorithms;

c) to provide preliminary figures on the practical performance of these algorithms under a reasonable environment; and finally,

d) to compare these algorithms with the algorithms for random generation [9, 26, 89].

Our analysis of the performance of the unranking algorithms in Sections 3.2 and 4.2 introduced several simplifications, so two important questions were left unanswered:

1. Which is the constant factor in the average-case complexity of the unranking algorithm for a given class $\mathcal{A}$? In particular, for objects which are unranked using the boustrophedonic order the analysis didn't provide even a rough estimation of such constant.

2. How does unranking compare to random generation? Both (lexicographic and boustrophedonic) unranking and random generation have complexities with identical order of magnitude, but the analysis did not settled how the respective constant factors compare to each other.

In order to provide an answer (even if partial) to these questions we have developed working implementations of our unranking algorithms in Maple and conducted a series experiments to measure their practical performance.

We have instrumented our programs to collect data about the number of arithmetical operations that they perform. In this context this is the usual choice to measure implementation-independent performance, much in the same manner that key comparisons are used to measure the performance of sorting and searching algorithms. On the other hand, we have also collected data on their actual execution times. While these last measurements should have been done in several platforms and for several different implementations in order to extract well-grounded conclusions, they confirm the main trends that the implementation-independent measurements showed up when we compared unranking and random generation, and address goal c) in our initial wish list.

In spite of the theoretical equivalence of the complexity of unranking and random generation, say $g(n)$, the comparison on practical terms is interesting, for instance, if we want to produce samples without repetition. We can use the *rejection method* (Algorithm 17), generating objects of size n at random until we have collected m distinct objects so that the expected total cost is $\mathrm{O}(m \log m \cdot g(n))$, or we can generate m distinct ranks at random in $\Theta(m)$ time using Floyd's algorithm (Algorithm 18) and then make the corresponding m calls to the function *Unrank* hence, with expected total cost $\Theta(m \cdot g(n))$. While the second approach is theoretically better, the answer could be different on practical grounds (especially if m is not very big and random generation were more efficient than unranking by a large factor).

Algorithm 17 *Rejection method* to generate m different objects of $\mathcal{A}_n$

Rejection($\mathcal{A}, m, n$)
    $S := \emptyset$
    while $|S| < m$ **do** $x :=$ Draw($\mathcal{A}, n$); **if** $x \notin S$ **then** $S := S \cup \{x\}$ **fi end**

Algorithm 18 *Floyd's algorithm* to generate m different objects of $\mathcal{A}_n$

Floyd_Algorithm($\mathcal{A}, m, n$)
    $S := \emptyset$; $C := \emptyset$ ▷ C contains the ranks of the objects in S ◁
    for $j :=$ Count($\mathcal{A}, n$) $- m$ **to** Count($\mathcal{A}, n$) **do**
        $u :=$ Random_Integer($[1..j]$)
        if $u \notin C$ **then** $C := C \cup \{u\}$; $S := S \cup \{$Unrank($\mathcal{A}, n, u$)$\}$
        else $u \in C$ **then** $C := C \cup \{j\}$; $S := S \cup \{$Unrank($\mathcal{A}, n, j$)$\}$
        fi
    end

The Υ-rules of Chapters 3 and 4, allow us to compute very precise estimates of the average cost of unranking $\mathcal{A}$-objects of size n ($\mu_{n,\mathcal{A}}$) in most cases; however, recall that to obtain these rules we made several simplifications. Also, for unlabelled powersets and sets we only have upper bounds (Theorem 19). Therefore another goal is to check whether these bounds are far away from the real values or, on the contrary, they are tight.

First of all, we do not count the preprocessing cost of parsing and converting specifications to standard form. We also neglect the cost of converting the object generated by the unranking algorithm back to the *non-standard form* in which the original specification was given (see the introduction of Part II). We disregard the cost of calling the function

Count, too. Last but not least, we assumed that the cost of the non-recursive part of the unranking algorithms is exactly the number of iterations made to determine the size of the first component in a product or sequence, the leading component in a cycle or set, etc. For instance, if j is the size of α in the object $\langle \alpha, \beta \rangle \in \mathcal{A} \times \mathcal{B}$ of size n which we want to produce, then we assume that the cost of unranking is $j + \mathrm{cu}(\alpha, \mathcal{A}) + \mathrm{cu}(\beta, \mathcal{B})$, but in fact we should take $c_1 \cdot j + c_2 + \mathrm{cu}(\alpha, \mathcal{A}) + \mathrm{cu}(\beta, \mathcal{B})$. While the simplifications do not invalidate the computation from the point of view of the order of growth of the average-complexity, a more accurate computation is necessary to determine precise estimations of the actual constant factors and lower order terms in the average-case complexity of unranking.

This chapter has three sections. In Section 5.1 we present the experimental data of our unranking algorithms for some labelled and unlabelled classes. In Section 5.2 we compare our unranking algorithms with the algorithms for random generation. Finally, we expound our conclusions.

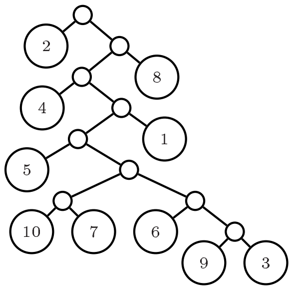
The results presented in this chapter have been published in [59].

5.1 Experiments on the Performance

We have implemented the unranking algorithms in MAPLE (Maple V Release 8) and we have run the experiments under Windows 98 in a Pentium 4 at 1.7 GHz with 512 Mb of RAM. The unranking algorithms use the basic facilities for counting and parsing of specifications already provided by the combstruct package [89]. The interface of the function *Unrank* is similar to that of the function *Draw* for random generation also provided by the combstruct package; for instance, we might write

$$\mathit{Unrank}([\mathcal{B}, \{\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}\}, \mathsf{labelled}], \mathsf{size} = 10,\ \mathsf{rank} = 3982254681)$$

to obtain the following labelled binary tree of size 10:



The *standard notation* for such object is

$$\mathsf{Prod}(Z_2, \mathsf{Prod}(\mathsf{Prod}(Z_4, \mathsf{Prod}(\mathsf{Prod}(Z_5, \mathsf{Prod}(\mathsf{Prod}(Z_{10}, Z_7), \mathsf{Prod}(Z_6, \mathsf{Prod}(Z_9, Z_3)))), Z_1)), Z_8))$$

although the unranking proper returns

$$< Z, << Z, << Z, << Z, Z, 1 >, < Z, < Z, Z, 1 >, 1 >, 8 >, 1 >, Z, 6 >, 2 >, Z, 2 >, 1 >$$

Specification	Lab.	Unl.	Name
$\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$	$\surd$	$\surd$	Binary trees
$\mathcal{M} = Z + Z \star \mathcal{M} + Z \star \mathcal{M} \star \mathcal{M}$	$\surd$	$\surd$	Motzkin Trees
$\mathcal{F} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{T})), \mathcal{T} = Z \star \mathsf{Set}(\mathcal{T})$	$\surd$		Functional graphs
$\mathcal{P}_1 = \mathsf{Set}(Z \times \mathcal{S}_1), \mathcal{S}_1 = \mathsf{Seq}(Z)$		$\surd$	Integer Partitions
$\mathcal{P}_2 = \mathsf{Set}(Z \times \mathcal{S}_2), \mathcal{S}_2 = \mathsf{Set}(Z)$		$\surd$	Integer Partitions
$\mathcal{S} = \mathsf{Seq}(\mathsf{Set}(Z, \text{card} \geq 1))$	$\surd$		Surjections
$\mathcal{S} = \mathsf{Seq}(\mathsf{Set}(Z, \text{card} \geq 1))$		$\surd$	Integer Compositions

Table 5.1: Labelled and unlabelled classes used in the unranking experiments.

and a further labelling step is necessary.

The function *Unrank* also accepts several optional parameters, in particular we can specify which order we want to use: lexicographic (default) or boustrophedonic, LeftBoxedProduct (default) or RightBoxedProduct, SF (default) or PF, etc.

The first piece of our experimental setup was the choice of the combinatorial classes to be used. Our aim was to find a representative collection. It was specially interesting to find cases where both the unlabelled and labelled versions made sense. As a counterexample, the labelled class $\mathsf{Seq}(Z)$ is interesting (permutations), but the unlabelled class $\mathsf{Seq}(Z)$ is not, since there is only one element of each size. Table 5.1 presents our selection[1].

We use $\mu_{n,\mathcal{A}}$ and $\mu'_{n,\mathcal{A}}$ to denote the average cost and the average number of arithmetic operations to unrank an object of size n in the class $\mathcal{A}$, respectively. By definition, it is clear that $\mu_{n,\mathcal{A}} \leq \mu'_{n,\mathcal{A}}$. We recall that the theoretical analysis in the previous chapters was based upon several simplifying assumptions which disregarded some of the necessary arithmetical operations. Nevertheless $\mu_{n,\mathcal{A}}$, as $\mu'_{n,\mathcal{A}}$, does not take into account the arithmetic operations needed to parse specifications and to fill the counting tables ($\hat{\delta}_{n,j}$, $\delta_{n,j}$, etc.), since once this has been done as a preprocessing phase, all subsequent *Unrank* (or *Draw*) calls on the corresponding class do not need to recompute the tables or to parse the specification. One goal of the first set of experiments that we have conducted was to empirically measure the cost of unranking in a sample of $N = 1000$ random ranks of $\mathcal{A}$-objects of size n (denoted by $\overline{\mu}_{n,\mathcal{A}}$) when using lexicographic ordering and to compare it with the theoretical asymptotic estimate that we can get for $\mu_{n,\mathcal{A}}$ using the tools of Chapters 3 and 4, so that we could determine the range of practical validity of that asymptotic estimate. A second goal was to measure $\overline{\mu}'_{n,\mathcal{A}}$, which is the empirical estimate of $\mu'_{n,\mathcal{A}}$. Finally, the corresponding best fit curves (BF) have also been computed. The inspection of the code of the unranking algorithms suggests that $\mu'_{n,\mathcal{A}} = 3\,\mu_{n,\mathcal{A}} + o(\mu_{n,\mathcal{A}})$ for labelled classes, and $\mu'_{n,\mathcal{A}} = 2\,\mu_{n,\mathcal{A}} + o(\mu_{n,\mathcal{A}})$ for unlabelled classes, whenever $\mathcal{A}$ is not an iterative class, an hypothesis which is consistent with the empirical data that we have collected.

We have also measured the *relative error*

$$\epsilon_{n,\mathcal{A}} = \frac{|\mu_{n,\mathcal{A}} - \overline{\mu}_{n,\mathcal{A}}|}{\mu_{n,\mathcal{A}}}$$

[1] There '$\star$' stands for both labelled and unlabelled products.

Symbol	Definition
$\mu_{n,\mathcal{A}}$	Average cost to unrank an object of size n in the class $\mathcal{A}$
$\mu'_{n,\mathcal{A}}$	Average number of arithmetic operations to unrank an object of size n in the class $\mathcal{A}$
$\overline{\mu}_{n,\mathcal{A}}$	Empirical estimate of $\mu_{n,\mathcal{A}}$
$\overline{\mu}'_{n,\mathcal{A}}$	Empirical estimate of $\mu'_{n,\mathcal{A}}$
$\epsilon_{n,\mathcal{A}}$	Relative error of experimental data
$\mathrm{AQE}_{\mathcal{A}}$	Average quadratic error of experimental data
$\overline{\tau}_{n,\mathcal{A}}$	Empirical estimate of CPU time (in seconds) to unrank an object of size n in the class $\mathcal{A}$
$\hat{\tau}_{n,\mathcal{A}}$	Empirical estimate of CPU time (in seconds) to generate a random object of size n in the class $\mathcal{A}$

Table 5.2: Notation for experimental results.

and the *Average Quadratic Error* (also called the *sum-of-squares error function*, AQE for short):

$$\mathrm{AQE}_{\mathcal{A}} = \frac{1}{|S|} \sum_{n \in S} \left(\mu_{n,\mathcal{A}} - \overline{\mu}_{n,\mathcal{A}}\right)^2 ,$$

where S is the set of sizes used in our measurements.

We have measured the empirical estimate of CPU time to unrank objects of size n in the class $\mathcal{A}$ (denoted by $\overline{\tau}_{n,\mathcal{A}}$), and the best fit curves under the hypothesis that $\overline{\tau}_{n,\mathcal{A}} = \Theta(\mu_{n,\mathcal{A}})$. Finally, we have computed the CPU time to generate a random object of size n in the class $\mathcal{A}$ ($\hat{\tau}_{n,\mathcal{A}}$).

We have performed the same experiments using the boustrophedonic unranking algorithms. The theoretical analysis establishes that the worst-case for boustrophedonic unranking of any class is $\mathrm{O}(n \log n)$; however, there are no specific results for its average-case complexity. The experiments support our conjecture that the average complexity is $\Theta(n \log n)$ whenever the class is not iterative. The experiments show also that it performs slightly worse than lexicographic unranking for iterative classes.

We summarize the relevant quantities and notations defined so far in Table 5.2. Moreover, given a class $\mathcal{A}$, we use subscripts 'L' or 'U', like $\mathcal{A}_{\mathsf{L}}$ or $\mathcal{A}_{\mathsf{U}}$, to indicate on if the class is labelled or not, whenever necessary. Finally, we use the superscript '*' when we only have a theoretical upperbound as those given by Theorem 19, and not a precise estimate.

The number of combinatorial structures in a given class typically grows exponentially and superexponentially in n. Due to the huge size of the numbers involved in the arithmetical computations performed by unranking we have only considered in our experiments objects of size up to 900. In Table 5.3 we show a small list of CPU times (in seconds) to compute the counting of some classes of Table 5.1 and (unlabelled) random mapping patterns (denoted by $\mathcal{F}_{\mathsf{U}}$). Moreover, for other combinatorial classes, initializing the tables of counts to unrank objects of size 300 can take up to 30 seconds of CPU! For larger n, we face the problem of storing and manipulating huge counters. However, Maple V Release 9.5 has improved computation with these counters because it uses the *GNU Multiprecision*

Maple V Release 8

n	$\mathcal{B}_{\mathsf{L}}$	$\mathcal{B}_{\mathsf{U}}$	$\mathcal{F}_{\mathsf{L}}$	$\mathcal{F}_{\mathsf{U}}$	$\mathcal{S}_{\mathsf{L}}$	$\mathcal{S}_{\mathsf{U}}$
300	5.56	1.28	29.87	4.45	4.07	1.08
600	42.93	13.78	173.07	93.06	21.66	4.62
900	180.15	110.44	848.93	1176.57	73.23	11.65

Maple V Release 9.5

n	$\mathcal{B}_{\mathsf{L}}$	$\mathcal{B}_{\mathsf{U}}$	$\mathcal{F}_{\mathsf{L}}$	$\mathcal{F}_{\mathsf{U}}$	$\mathcal{S}_{\mathsf{L}}$	$\mathcal{S}_{\mathsf{U}}$
300	3.29	1.34	9.45	4.97	2.87	1.42
600	18.05	5.35	56.06	20.75	12.37	5.63
900	58.68	13.01	193.21	50.29	32.35	13.17

Table 5.3: CPU time to compute the counting of some classes.

(GMP) Library[2]. In Table 5.3 we also present the CPU times to compute the counters in Maple V Release 9.5. In general, Maple V Release 9.5 is much faster than Maple Release 8 in this task.

Our experiments show that the unranking algorithms have acceptable running times — there is some penalty that we have to pay for the generality and flexibility— in a relatively common and modest platform when $n \leq 800$.

Last but not least, the experiments have been useful to show that our asymptotic analysis provides good estimations for already small values of n. All the tables and figures with the experimental results are summarized in Appendix A. The following subsections highlight the most relevant aspects of the collected data.

5.1.1 A Representative Example: Binary Trees

We know that the average cost of lexicographic unranking for labelled binary trees is (3.8)

$$\mu_{n,\mathcal{B}} = 0.8862\ldots\, n\sqrt{n} + 0.5\, n - 0.3323\ldots\, \sqrt{n} + 0.25 + \mathrm{o}(1)\ .$$

From the measured data $\overline{\mu}_{n,\mathcal{B}}$ of Table 5.4 (Table A.1 in Appendix A), the best fit curve with a function of the form $a\, n\sqrt{n} + b\, n$ is

$$\mathrm{BF}_{\mathcal{B}}(n) = 0.8738\ldots\, n\sqrt{n} - 0.3793\ldots\, n\ . \tag{5.1}$$

The collected data shows that the theoretical asymptotic estimate is very accurate even for $n = 50$: Look at the row $\epsilon_{n,\mathcal{B}}$ of Table 5.4 (also Table A.1), where $\epsilon_{n,\mathcal{A}} = |\mu_{n,\mathcal{A}} - \overline{\mu}_{n,\mathcal{A}}|/\mu_{n,\mathcal{A}}$ is the relative error, and note that the difference is almost unnoticeable. It is also possible to compute the AQE:

$$\mathrm{AQE}_{\mathcal{B}} = \frac{1}{|S|} \sum_{n\in S} \left(\mu_{n,\mathcal{B}} - \overline{\mu}_{n,\mathcal{B}}\right)^2 = 81.8534\ldots\ ,$$

[2] http://www.maplesoft.com/applications/index.aspx

where S is the set of sizes $\{50, 100, 150, 200, 250\}$ (thus, $|S| = 5$). Finally, the experimental data for $\overline{\mu}'_{n,\mathcal{B}}$ and $\overline{\mu}_{n,\mathcal{B}}$ supports the hypothesis $\mu'_{n,\mathcal{B}} = 3\,\mu_{n,\mathcal{B}} + \mathrm{o}(\mu_{n,\mathcal{B}})$.

The average cost of lexicographic unranking for unlabelled binary trees (Table A.3) is the same as for labelled trees, namely, it is given by Equation (3.8). Again, the match between the theoretical results and the experimental data is very accurate, with the relative error being less than 1.33%. The best fit curve with a function of the form $a\,n\sqrt{n} + b\,n$ is

$$0.9123\ldots\ n\sqrt{n} - 0.9007\ldots\ n\ .$$

and the $\mathrm{AQE}_{\mathcal{B}} = 130.2508\ldots$. Finally, the experimental data for $\overline{\mu}'_{n,\mathcal{B}}$ and $\overline{\mu}_{n,\mathcal{B}}$ supports the hypothesis $\mu'_{n,\mathcal{B}} = 2\,\mu_{n,\mathcal{B}} + \mathrm{o}(\mu_{n,\mathcal{B}})$.

In the labelled case, the empirical estimate of the CPU time grows as $n\sqrt{n}$ in the considered range of sizes (see Figure 5.1 or A.2 solid lines). The experimental data in the case of unlabelled binary trees does not substantially differ and the conclusions are basically identical, except that the empirical estimate of the CPU times are noticeably smaller, as the magnitude of the involved counts is very small compared with that of the labelled case.

n	50	100	150	200	250
$\mu_{n,\mathcal{B}}$	285.972	832.899	1549.029	2401.925	3372.862
$\overline{\mu}_{n,\mathcal{B}}$	285.215	833.860	1548.209	2406.245	3353.153
$\overline{\mu}'_{n,\mathcal{B}}$	1149.645	3095.582	5538.629	8412.736	11553.461
$\epsilon_{n,\mathcal{B}}$	0.26 %	0.11 %	0.53 %	0.18 %	0.58 %
$\overline{\tau}_{n,\mathcal{B}}$	0.044886	0.131197	0.252442	0.421947	0.637975
$\mu_{n,\mathcal{B}}$	$0.8862\ldots\ n\sqrt{n} + 0.5\,n - 0.3323\ldots\ \sqrt{n} + 0.25 + \mathrm{o}(1)$				
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.8738\ldots\ n\sqrt{n} - 0.3793\ldots\ n$				
$\mathrm{AQE}_{\mathcal{B}}$	$81.8534\ldots$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$2.1518\ldots \cdot 10^{-4}\,n\sqrt{n} - 8.8639\ldots \cdot 10^{-4}\,n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} \approx 3\,\overline{\mu}_{n,\mathcal{B}} + 6\,n - 6$				

Table 5.4: Lexicographic labelled binary trees.

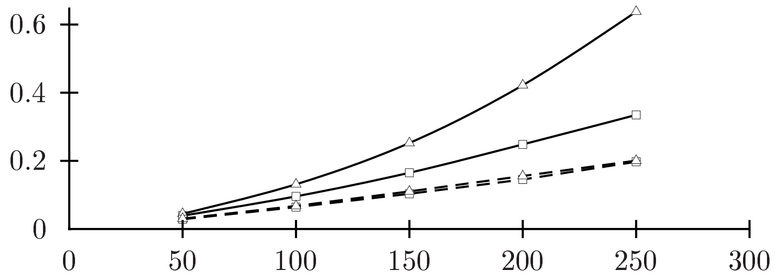


Figure 5.1: CPU time $\overline{\tau}_{n,\mathcal{B}}$ for binary trees (seconds).

Tables A.4 and A.6 give the experimental data for the boustrophedonic unranking of binary trees. This data is consistent with the hypothesis that this cost is $\Theta(n \log n)$ on average. For instance, the best fit curves for labelled and unlabelled cases are $0.6506\ldots\ n \log n + 1.2111\ldots\ n$ and $0.6610\ldots\ n \log n + 1.1584\ldots\ n$, respectively. This suggests that the performance of the labelled and unlabelled versions of the boustrophedonic unranking for products is essentially the same. We will see later that this is no longer true when we compare the performance of the unranking (either lexicographic or boustrophedonic) of labelled and unlabelled sets. Moreover, we have for boustrophedonic unranking the relation $\mu'_{n,\mathcal{B}} = 3\,\mu_{n,\mathcal{B}} + 8\,n - 8$ for the labelled case, and $\mu'_{n,\mathcal{B}} \approx 2\,\mu_{n,\mathcal{B}} + 4.5\,n - 5$ for the unlabelled case, which more or less coincide with those we found for lexicographic unranking. Side by side, Figure A.1 plots the corresponding values $\overline{\mu}_{n,\mathcal{B}}$ for boustrophedonic and lexicographic order to ease the comparison. The $n \log n$ behavior of the former versus the $n\sqrt{n}$ behavior of the latter is quite evident from the plot.

Finally, labelled and unlabelled best fit curves of the CPU time are also $\Theta(n \log n)$. See Figure 5.1 (dashed lines) and Tables A.4 and A.6, and Figure A.2.

On the other hand, if we use the boxed heuristic then the average cost to unrank binary trees in lexicographic order decreases to $\Theta(n \log n)$. The experimental data fits well with the theoretical analysis (see Section 3.4). For boustrophedonic order, using the boxed heuristic (Table A.5) it is not clear whether the average cost is $\mathrm{O}(n \log n)$ with a very small leading factor, if it is linear or it is something in between. Tables A.2 and A.5 collect the experimental data related to unranking with the boxed heuristic.

5.1.2 Other Classes

Now we discuss some of the results obtained for the other classes that we have considered in our study.

The average cost of unranking Motzkin trees in lexicographic order is (Equation (3.9)):

$$\mu_{n,\mathcal{M}} = 0.5116\ldots\ n\sqrt{n} - 0.3333\ldots\ n - 0.1918\ldots\ n + 0.1666\ldots + \mathrm{o}(1)\ .$$

In Tables A.7 and A.8 and Figure A.5 we show the experimental data. The best fit curves are $0.4969\ldots\ n\sqrt{n} - 0.1621\ldots\ n$ for the labelled case, and $0.5356\ldots\ n\sqrt{n} - 0.5946\ldots\ n$ for the unlabelled one. The small values of the relative error $\epsilon_{n,\mathcal{M}}$ assess the good fit of experimental data to the asymptotic estimates. However the first curve (for labelled case) provides a poorer fit than the second. In particular, the AQE for the labelled case is $149.4744\ldots$ whereas for the unlabelled case is $202.0279\ldots$. Moreover, the values $\overline{\mu}_{n,\mathcal{M}}$ and $\overline{\mu}'_{n,\mathcal{M}}$ suggest $\mu'_{n,\mathcal{M}} = 3\,\mu_{n,\mathcal{M}} + 7\,n - 7$ for the labelled case, and $\mu'_{n,\mathcal{M}} = 2\,\mu_{n,\mathcal{M}} + 5\,n - 5$ for the unlabelled case. On the other hand, in Tables A.9 and A.10 we show the boustrophedonic best fit curves under the assumption of $\Theta(n \log n)$ behavior. Again, the data support that the performance of unranking is (almost) identical for the unranking of labelled and unlabelled versions of an specification if they only involve unions, products and sequences; this holds for either lexicographic or boustrophedonic unranking. The values of $\overline{\mu}_{n,\mathcal{M}}$ and $\overline{\mu}'_{n,\mathcal{M}}$ suggest the relation $\mu'_{n,\mathcal{M}} = 3\,\mu_{n,\mathcal{M}} + 8\,n - 8$ for the labelled case and $\mu'_{n,\mathcal{M}} \approx 2\,\mu_{n,\mathcal{M}} + 4.2\,n$ for the unlabelled case.

The lexicographic average CPU time follows the same patterns as binary trees; that is, even though both are $\Theta(n\sqrt{n})$, the average CPU time for the unlabelled case is clearly smaller because of the involved counts. Moreover, as usual, the average CPU time to unrank these non-iterative classes in boustrophedonic order grows as $n \log n$ (see the corresponding plot in Figure A.5).

For functional graphs in lexicographic order (Tables A.11), the data ($\overline{\mu}_{n,\mathcal{F}}$) for small size agrees with the theoretical results computed in Section 3.3 ($\mu_{n,\mathcal{F}}$). In particular, the relative error in the lexicographic order $\epsilon_{n,\mathcal{F}}$ is less than 1.86% in the range of sizes we have considered. Moreover, when comparing $\overline{\mu}_{n,\mathcal{F}}$ with $\mu'_{n,\mathcal{F}}$ we obtain the relation $\mu'_{n,\mathcal{F}} \approx 3\,\mu_{n,\mathcal{F}} + 11.2\,n$. On the other hand, the experimental data for the boustrophedonic order (Table A.12) support the hypothesis that this cost is $\Theta(n \log n)$ on average. For instance, the best fit curve for the experimental data is $1.7981\ldots\; n \log n - 2.4923\ldots\; n$. We conjecture also that $\mu'_{n,\mathcal{F}} \approx 3\,\mu_{n,\mathcal{F}} + 14.5\,n$ for boustrophedonic unranking.

The lexicographic CPU time follows a best fit curve of the form $a\,n\sqrt{n} + b\,n$ (Table A.11) which is $1.7405\ldots \cdot 10^{-4}\, n\sqrt{n} - 5.5017\ldots \cdot 10^{-4}\, n$, and for the boustrophedonic case we have a best fit curve of the form $a\,n \log n + b\,n$ (Table A.12), which is $3.9558\ldots \cdot 10^{-4}\, n \log n - 8.1434\ldots \cdot 10^{-4}\, n$. Figure A.6 depicts these best fit curves.

In Tables A.13 and A.14 we show the experimental data for labelled surjections and unlabelled integer compositions in lexicographic order. For surjections we have $\mu_{n,\mathcal{S}_\mathsf{L}} = 2\,n$, and for integer compositions we have $S_\mathsf{U}(z) = (1-z)/(1-2z)$ and $\Upsilon\mathcal{S}_\mathsf{U} \ll z(2+2z-z^2)/(1-2z)^2$. The relative error (computed w.r.t. the upper bound $\mu^*_{n,\mathcal{S}_\mathsf{U}}$) for the integer compositions is about 9.13%. On the other hand, as we will see for integer partitions, the lexicographic order gives a better performance than the boustrophedonic order (Tables A.15 and A.16).

Here we also see that when the specification involves sets, the performance of unranking depends on whether the class is labelled or not, contrary to what happens with specifications that only involve unions, products and sequences.

5.1.3 Unlabelled Powersets and Sets

For integer partitions we consider two specifications: $\mathcal{P}_1 = \mathsf{Set}(\mathcal{T}_1)$ where $\mathcal{T}_1 = Z \times \mathcal{S}_1$ and $\mathcal{S}_1 = \mathsf{Seq}(Z)$, and $\mathcal{P}_2 = \mathsf{Set}(\mathcal{T}_2)$ where $\mathcal{T}_2 = Z \times \mathcal{S}_2$ and $\mathcal{S}_2 = \mathsf{Set}(Z)$. For the lexicographic order, their corresponding experimental data is summarized in Tables A.17 and A.18. In both cases we only have upper bounds on their average complexity (see Theorem 19). For P_1 we have $\Upsilon\mathcal{P}_1 \ll (\vartheta T_1 + 2\,\Delta T_1 + \mathrm{L}T_1 + \Upsilon\mathcal{T}_1)\cdot P_1$, where $T_1 = z/(1-z)$, $\Upsilon\mathcal{T}_1 = z/(1-z)^2$ and $P_1 = \exp\left(\sum_{n>0} T_1(z^n)/n\right)$; and for P_2 we have $\Upsilon\mathcal{P}_2 \ll (\vartheta T_2 + 2\,\Delta T_2 + \mathrm{L}T_2 + \Upsilon\mathcal{T}_2)\cdot P_2$, where $T_2 = z/(1-z)$, $\Upsilon\mathcal{T}_2 = (1+2z-z^2)\,z/(1-z)^2$ and $P_2 = \exp\left(\sum_{n>0} T_2(z^n)/n\right)$. We note that, even though $\overline{\mu}_{n,\mathcal{P}_1} \ll \overline{\mu}_{n,\mathcal{P}_2}$, we have that $\overline{\mu}'_{n,\mathcal{P}_1} \gg \overline{\mu}'_{n,\mathcal{P}_2}$ and $\tau_{n,\mathcal{P}_1} \gg \tau_{n,\mathcal{P}_2}$. Here we have an example where $\mu_{n,\mathcal{P}_1} < \mu_{n,\mathcal{P}_2}$ but $\mu'_{n,\mathcal{P}_1} > \mu'_{n,\mathcal{P}_2}$. The experimental data suggest that $\mu'_{n,\mathcal{P}_1} \approx 4 \cdot \mu_{n,\mathcal{P}_1} + 17$, $\mu'_{n,\mathcal{P}_2} \approx 2.65 \cdot \mu_{n,\mathcal{P}_2} + 64$ but $\mu_{n,\mathcal{P}_2} \approx 1.4 \cdot \mu_{n,\mathcal{P}_1}$. Since we only have upper bound to $\mu^*_{n,\mathcal{P}_i}$ for the average cost of unranking in $\mathcal{P}_1$ and $\mathcal{P}_2$, we just give relative errors $\epsilon^*_{n,\mathcal{P}_i}$ w.r.t. the corresponding upper bounds. For the boustrophedonic order, we have already $\overline{\mu}_{n,\mathcal{P}_1} > \overline{\mu}_{n,\mathcal{P}_2}$, $\overline{\mu}'_{n,\mathcal{P}_1} > \overline{\mu}'_{n,\mathcal{P}_2}$ and $\tau_{n,\mathcal{P}_1} > \tau_{n,\mathcal{P}_2}$ (see Tables A.19 and A.20).

n	50	100	150	200	250
$\mu^*_{n,\mathsf{PSet}\mathcal{B}}$	412.786	1095.937	1948.916	2938.899	4047.046
$\overline{\mu}_{n,\mathsf{PSet}\mathcal{B}}$	407.907	1094.567	1942.740	2939.697	4024.715
$\epsilon^*_{n,\mathsf{PSet}\mathcal{B}}$	1.18%	0.12%	0.32%	0.27%	0.55%
$\mu^*_{n,\mathsf{Set}\mathcal{B}}$	412.713	1095.298	1947.832	2937.437	4045.250
$\overline{\mu}_{n,\mathsf{Set}\mathcal{B}}$	408.567	1095.620	1981.904	2905.537	4053.802
$\epsilon^*_{n,\mathsf{Set}\mathcal{B}}$	1.00%	0.03%	1.75%	1.08%	0.21%

Table 5.5: Unlabelled powersets and sets of binary trees.

Even though both orders are linear, like in the cost of surjections and integer compositions, the lexicographic order performs slightly better than the boustrophedonic order. Moreover, we conjecture that the same phenomenon occurs for any iterative class.

Finally, we have also checked the upper bounds of unlabelled powersets and sets of binary trees. Both theoretical upper bounds of the Υ-rules given by Theorem 19 are very accurate with the experimental data. For instance, the relative error w.r.t. the upper bounds is between 0.03% to 1.75%: See Table 5.5.

In general, these experiments indicate that boustrophedonic unranking could be slightly more inefficient than lexicographic unranking for iterative classes (surjections, integer compositions and integer partitions). On the contrary, for non-iterative classes (binary trees, Motzkin trees and functional graphs), boustrophedonic unranking clearly outperforms lexicographic unranking, even for small n.

5.2 Unranking v.s. Random Generation

From a theoretical point of view this question has a clear cut answer: they have identical complexity. However, the hidden constant factor and lower order terms may markedly differ; the goal of the experiments described in this section is to provide evidence about this particular aspect. We have chosen the implementation of random generation already present in the combstruct package for a fair comparison, since the platform, programming language, etc. are identical then. From the point of view of users, the underlying ordering of the class is irrelevant when generating objects at random, hence the default order used by the function *Draw* is the boustrophedonic order, which is never significantly worse than the lexicographic order and it is frequently much faster (Section 5.1). Hence, to get a meaningful comparison we will also use boustrophedonic unranking in all the experiments of this section.

As in the previous section we will not take into account all the preprocessing needed by both unranking and random generation, therefore we will "start the chrono" once the initialization of counting tables and the parsing of specifications have already been finished. In the previous section we have given the data concerning the empirical estimate of the CPU time to unrank an object of specific sizes and classes (Tables A.4, A.6, A.9, A.10, A.12, A.19, A.20, A.15 and A.16). For moderately sized objects ($n = 250$) the empirical

n	$\rho_{n,\mathcal{B}_{\mathsf{L}}}$	$\rho_{n,\mathcal{B}_{\mathsf{U}}}$	$\rho_{n,\mathcal{M}_{\mathsf{L}}}$	$\rho_{n,\mathcal{M}_{\mathsf{U}}}$	$\rho_{n,\mathcal{F}}$	$\rho_{n,\mathcal{P}_1}$	$\rho_{n,\mathcal{P}_2}$	$\rho_{n,\mathcal{S}_{\mathsf{L}}}$	$\rho_{n,\mathcal{S}_{\mathsf{U}}}$
50	1.12	2.14	1.12	2.20	2.70	2.45	2.41	2.55	2.61
100	1.13	2.23	1.11	2.23	2.70	2.33	2.54	2.67	2.61
150	1.12	2.25	0.97	2.25	2.67	2.13	2.73	2.68	2.63
200	1.10	2.22	0.99	2.20	2.64	2.34	2.95	2.65	2.54
250	1.01	2.18	0.89	2.17	2.61	2.37	3.40	2.69	2.75

Table 5.6: Ratios of empirical estimate of CPU times for (boustrophedonic) unranking v.s. random generation.

estimate of CPU time is typically around 0.2 seconds; while for the largest objects (n=800) this time is around 1.2 seconds.

Since we want here to compare unranking and random generation we will systematically work with the ratio $\rho_{n,\mathcal{A}}$ between the empirical estimate of the CPU time to unrank an object of size n in the class $\mathcal{A}$ and the empirical estimate of the CPU time to generate a random object of size n in the class $\mathcal{A}$; i.e., $\rho_{n,\mathcal{A}} = \tau_{n,\mathcal{A}}/\overline{\tau}_{n,\mathcal{A}}$. Table 5.6 summarizes this comparative analysis, where $\mathcal{B}_{\mathsf{L}}$ and $\mathcal{B}_{\mathsf{U}}$ denote labelled and unlabelled binary trees, $\mathcal{M}_{\mathsf{L}}$ and $\mathcal{M}_{\mathsf{U}}$ denote labelled and unlabelled Motzkin trees, $\mathcal{F}$ are the (labelled) functional graphs, $\mathcal{P}_1$ and $\mathcal{P}_2$ are the (unlabelled) integer partitions, $\mathcal{S}_{\mathsf{L}}$ are the (labelled) surjections, and $\mathcal{S}_{\mathsf{U}}$ the (unlabelled) integer compositions. The data varies widely from one case to other, so that no firm conclusions can be drawn. Some preliminary results (with small samples for large sizes) confirm the theoretical prediction that $\rho_{n,\mathcal{A}}$ must tend to a constant $\rho_{\mathcal{A}}$ as $n \to \infty$, but there seems that there is no easy rule to describe/compute these constants. For instance, the first five classes are non-iterative and the other ones are, but there is no "commonality" in the behavior of the five first classes. Neither there is such commonality between the arborescent classes $\mathcal{B}$ and $\mathcal{M}$, or the unlabelled classes. From the small collection we have worked with, however, it seems that $\rho_{\mathcal{A}}$ will be typically between 1.0 and 3.0.

If we do further tests and analytical work, then sampling based on unranking plus Floyd's algorithm should be the alternative of choice as long as the number of elements to be sampled were ≥ 1000.

5.3 Conclusions and Final Remarks

We have investigated the use of alternative isomorphisms to specify a given class and which improvements in performance can be thus achieved. Our results show that the complexity of unranking is comparable to that of random generation [26] and that the same approach can be successfully applied to the unranking problem.

One of the results of our experiments is that the asymptotic estimates provided by our theoretical analysis yield excellent approximations even for small values of n. Moreover, the experiments suggest $\mu'_{n,\mathcal{A}} = 3\,\mu_{n,\mathcal{A}} + \mathrm{o}(\mu_{n,\mathcal{A}})$ for labelled classes, and $\mu'_{n,\mathcal{A}} = 2\,\mu_{n,\mathcal{A}} + \mathrm{o}(\mu_{n,\mathcal{A}})$ for unlabelled classes, with $\mu'_{n,\mathcal{A}}$ counting the actual # of arithmetical operations performed.

The empirical estimate of the cost to unrank iterative classes is $\Theta(n)$, where n is the size of the computed object. The experiments indicate that the boustrophedonic order is slightly worse than the lexicographic order for iterative classes. For non-iterative classes, the empirical estimate of the cost is $\Theta(n\sqrt{n})$ for the lexicographic order when no heuristic is used, and $\Theta(n \log n)$ for the boustrophedonic order. These results agree with the theoretical analysis.

Another outcome of this study is the observation that labelled and unlabelled classes have similar constant factor in the average-case complexity when they only use unions, products and sequences; but, if the specifications involve sets then the constant factor is different (compare surjections with integer compositions), as we could anticipate by examination of the unranking algorithms: while unions and products are basically identical for labelled and unlabelled classes, it is obviously not the case for sets. Also, the complexity of the arithmetical operations and size of the counters has a significant impact on actual CPU performance, with labelled unranking needing much more time than unlabelled unranking because of the larger differences that we typically have on the cardinalities of the classes.

Chapter 6

Ranking of Labelled and Unlabelled Classes

The two previous chapters have been devoted to the unranking of labelled and unlabelled classes. Now, we are going to consider the inverse problem: *ranking*. Given a specification of a combinatorial class $\mathcal{A}$, and an object α of size $n = |\alpha|$ belonging to this class, the ranking problem is to compute the object's rank, that is, the number of objects of size n in $\mathcal{A}$ which are smaller than α, according to some previously fixed order. In particular, if that order coincides with the one we use to unrank then $\mathit{Unrank}(\mathcal{A}, n, \mathit{Rank}(\mathcal{A}, \alpha)) = \alpha$ and $\mathit{Rank}(\mathcal{A}, \mathit{Unrank}(\mathcal{A}, n, i)) = i$ for any $\mathcal{A}$, n, $\alpha \in \mathcal{A}_n$ and $0 \leq i < a_n$.

This chapter is organized in the following sections. In Sections 6.1 and 6.2 we describe the ranking algorithms for labelled and unlabelled classes, respectively. In Section 6.3 we briefly analyze the performance of the ranking algorithms. In Section 6.4 we study the problem of membership, in particular, how to efficiently implement the function is_a. Finally, we introduce the problem of the representation of combinatorial objects, which we study later in depth in Chapter 9.

We will concentrate in the basic issues of determining the rank by assuming that the objects are represented as tuples. The problem of converting objects expressed in a more human-readable way into the more convenient internal format of tuples and vice versa is studied in Chapter 9. For instance, the objects

$$\mathsf{PSet}(\mathsf{Prod}(Z, \mathsf{Prod}(Z, Z)), \mathsf{Prod}(Z, Z)) \quad \text{and} \quad \mathsf{Seq}(\mathsf{Prod}(\mathsf{Set}(Z_3, Z_2), Z_4), Z_1)$$

are represented by the tuples

$$\langle\langle Z, Z\rangle, \langle Z, \langle Z, Z\rangle\rangle\rangle \quad \text{and} \quad \langle\langle{}^{\square}\langle Z, Z, 0\rangle, Z, 0\rangle, Z, 3\rangle \ ,$$

respectively.

When the given object α belongs to $\mathcal{A} + \mathcal{B}$, it is necessary to know whether it belongs to $\mathcal{A}$ or to $\mathcal{B}$. This can be done in two different ways. The first one is to have a boolean function is_a for membership testing:

$$\text{is\_a}(\mathcal{A}, \alpha) \ = \begin{cases} \mathsf{true}, & \text{if } \alpha \in \mathcal{A}, \\ \mathsf{false}, & \text{otherwise.} \end{cases}$$

Section 6.4 considers how to implement such function.

The other possibility is to attach a tag (of size 0) to every object indicating the class it belongs to. In particular, this means that we use the formal definition

$$\mathcal{A}+\mathcal{B}=\mathcal{A}\times\{\epsilon_{\mathcal{A}}\}+\mathcal{B}\times\{\epsilon_{\mathcal{B}}\}\ ,$$

where $\epsilon_{\mathcal{A}}$ and $\epsilon_{\mathcal{B}}$ are distinct objects of size 0 (*tags*). Hence, the objects are represented by pairs like $\alpha\times\epsilon_{\mathcal{X}}$ if α belongs to $\mathcal{X}$. The drawback of this solution is that the representation of objects is not suitable from the user's standardpoint. For instance, we should write $\mathsf{Prod}(Z\times\epsilon_Z,\mathsf{Prod}(Z\times\epsilon_Z,Z\times\epsilon_Z)\times\epsilon_{\mathcal{B}\times\mathcal{B}})\times\epsilon_{\mathcal{B}\times\mathcal{B}}$ for the binary tree $\mathsf{Prod}(Z,\mathsf{Prod}(Z,Z))$ assuming that the binary trees are specified by $\mathcal{B}=Z+\mathcal{B}\times\mathcal{B}$. Clearly, if the representation of objects carries tags to know whether an object in $\mathcal{A}+\mathcal{B}$ belongs to $\mathcal{A}$ or to $\mathcal{B}$ then the function is_a can be trivially implemented.

For the following two sections, we assume that the function is_a is available, and that ranking's input is given as either 2- or 3-tuples.

6.1 Algorithms for Labelled Classes

6.1.1 Basic Algorithms: $+$, $\star$ and $^{\square}\star$

As we have seen in Chapter 3, using a few basic isomorphisms (see Equations (2.6), (2.11), (2.12), and Table 2.4), we need only consider objects built from unions, and partitional, boxed products and substitutions.

The algorithm for $\mathcal{A}+\mathcal{B}$ just needs to check whether the considered object belongs to $\mathcal{A}$ or to $\mathcal{B}$ and then perform the ranking in the appropriate class (see Algorithm 19).

Algorithm 19 Ranking of disjoint unions

Rank$(\mathcal{A}+\mathcal{B},\alpha)$
  if is_a$(\mathcal{A},\alpha)$ **then** Rank$(\mathcal{A},\alpha)$ **else** Count$(\mathcal{A},|\alpha|)$ + Rank$(\mathcal{B},\alpha)$ **fi**

Now, we consider the ranking of partitional products in lexicographic-PF order. If the size of α in the object $\langle\alpha,\beta,\ell\rangle$ is j then this means that at least

$$\binom{n}{0}\cdot a_0\cdot b_n+\binom{n}{1}\cdot a_1\cdot b_{n-1}+\cdots+\binom{n}{j-1}\cdot a_{j-1}\cdot b_{n-j+1}$$

objects are smaller than the given one according to the lexicographic order. Then we need to determine how many objects of size n whose first component has size j are also smaller than the given one. These are all the elements in

$$\alpha_j^{(0)}\star\mathcal{B}_{n-j}+\alpha_j^{(1)}\star\mathcal{B}_{n-j}+\cdots+\alpha_j^{(r_\alpha-1)}\star\mathcal{B}_{n-j}$$

with r_α the rank of α in $\mathcal{A}_j=\{\alpha_j^{(0)},\alpha_j^{(1)},\ldots,\alpha_j^{(r_\alpha-1)},\alpha_j^{(r_\alpha)},\ldots\}$ and $\alpha_j^{(i)}\prec\alpha_j^{(r_\alpha)}=\alpha$ for all $0\le i<r_\alpha$. There are at least $\binom{n}{j}\cdot r_\alpha\cdot b_{n-j}$ such objects. In a similar way, we also need to

determine how many objects of $\alpha_j^{(r_\alpha)} \star \mathcal{B}_{n-j}$ are smaller than the given one. These are all the elements in

$$\alpha_j^{(r_\alpha)} \star \beta_{n-j}^{(0)} + \alpha_j^{(r_\alpha)} \star \beta_{n-j}^{(1)} + \cdots + \alpha_j^{(r_\alpha)} \star \beta_{n-j}^{(r_\beta-1)}$$

with r_β the rank of β in $\mathcal{B}_{n-j} = \{\beta_{n-j}^{(0)}, \beta_{n-j}^{(1)}, \ldots, \beta_{n-j}^{(r_\beta-1)}, \beta_{n-j}^{(r_\beta)}, \ldots\}$ and $\beta_{n-j}^{(i)} \prec \beta_{n-j}^{(r_\beta)} = \beta$ for all $0 \le i < r_\beta$. There are at least $\binom{n}{j} \cdot r_\beta$ such objects. Finally, we have to take into account the rank r_ℓ of the (n,j)-partition. Thus, the sought rank of $\langle \alpha, \beta, \ell \rangle$ is

$$\sum_{l=0}^{j-1} \binom{n}{l} \cdot a_l \cdot b_{n-l} + \binom{n}{j} \cdot r_\alpha \cdot b_{n-j} + \binom{n}{j} \cdot r_\beta + r_\ell \ ,$$

where j is the size of α, r_α is the rank of α in $\mathcal{A}_j$, r_β is the rank of β in $\mathcal{B}_{n-j}$ and r_ℓ is the rank of the labelling (which we assume coincides with ℓ).

If we had considered the lexicographic-SF order, then we would have obtained a similar expression,

$$\sum_{l=0}^{j-1} \binom{n}{l} \cdot a_l \cdot b_{n-l} + r_\ell \cdot a_j \cdot b_{n-j} + r_\alpha \cdot b_{n-j} + r_\beta \ .$$

If, on the other hand, we were asking for the rank of $\langle \alpha, \beta, \ell \rangle$ according to the boustrophedonic-PF order (see Equation (2.2) in Chapter 2) the algorithm is slightly more complicated, see Algorithm 20.

Algorithm 20 Ranking of products (boustrophedonic-PF order)

Rank($\mathcal{A} \star \mathcal{B}, \langle \alpha, \beta, \ell \rangle$)
  $j := |\alpha|; n := |\langle \alpha, \beta, \ell \rangle|$
  if $j \le n - j$ **then**
    $r' := \Sigma_{l=0}^{j-1} \left(\binom{n}{l} \cdot \text{Count}(\mathcal{A}, l) \cdot \text{Count}(\mathcal{B}, n-l) + \binom{n}{n-l} \cdot \text{Count}(\mathcal{A}, n-l) \cdot \text{Count}(\mathcal{B}, l)\right)$
  else
    $r' := \Sigma_{l=0}^{n-j-1} \left(\binom{n}{l} \cdot \text{Count}(\mathcal{A}, l) \cdot \text{Count}(\mathcal{B}, n-l) + \right.$
      $\left.\binom{n}{n-l} \cdot \text{Count}(\mathcal{A}, n-l) \cdot \text{Count}(\mathcal{B}, l)\right)$
    $r' := r' + \binom{n}{n-j} \cdot \text{Count}(\mathcal{A}, n-j) \cdot \text{Count}(\mathcal{B}, j)$
  fi
  $r_\alpha := \text{Rank}(\mathcal{A}, \alpha); r_\beta := \text{Rank}(\mathcal{B}, \beta); r_\ell := \ell$
  return $r' + \binom{n}{j} \cdot r_\alpha \cdot \text{Count}(\mathcal{B}, n-j) + \binom{n}{j} \cdot r_\beta + r_\ell$

And finally, the ranking of products in the boustrophedonic-SF order follows the same steps as Algorithm 20, except for the last line which should be

$$\textbf{return } r' + r_\ell \cdot \mathit{Count}(\mathcal{A}, j) \cdot \mathit{Count}(\mathcal{B}, n-j) + r_\alpha \cdot \mathit{Count}(\mathcal{B}, n-j) + r_\beta \ .$$

The ranking of boxed products is equal to the ranking of partitional products, but using the binomial coefficients $\binom{n-1}{j-1}$ instead of $\binom{n}{j}$.

6.1.2 Algorithms for Other Combinatorial Operators

The algorithms for sequences, sets and cycles of restricted cardinality are based upon the isomorphisms in Table 2.4, and hence the problem is reduced to the ranking of unions, partitional and boxed products.

On the other hand, the algorithms for sequences, sets and cycles in card-lexicographic order rely on those for their fixed cardinalities. We just have to find the cardinality and to apply the corresponding ranking of sequences, sets and cycles with fixed cardinality (e.g., see Algorithm 21 for sets).

Algorithm 21 Ranking of sets (card-lexicographic order)

Rank($\mathsf{Set}(\mathcal{A}), \alpha$)
  $n := |\alpha|$; **if** $n = 0$ **then return** 0 **fi**
  $k := \#$ of components in α
  $r' := \Sigma_{l=0}^{k-1}$ Count($\mathcal{A}^{\{l\}}, n$); $r_\alpha :=$ Rank($\mathcal{A}^{\{k\}}, \alpha$)
  return $r' + r_\alpha$

Finally, substitution is based in the isomorphism $\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^{\{k\}}$. So, the ranking is like a product but without labels; if the size of the first component of the given object $\circ\langle\alpha, \beta\rangle$ is j, then at least $\sum_{k=1}^{j-1} a_k\, (b^{\{k\}})_n$ objects are smaller than the given one, and then we also need to determine how many objects of size n whose first component has size j are smaller than the given one, which is readily given by $Rank(\mathcal{A}, \alpha) \cdot (b^{\{j\}})_n$. Last but not least, we need to determine the rank of β in $\mathcal{B}^{\{j\}}$ (see Algorithm 22).

Algorithm 22 Ranking of substitutions (lexicographic order)

Rank($\mathcal{A} \circ \mathcal{B}, \circ\langle\alpha, \beta\rangle$)
  $j := |\alpha|; n := |\beta|$
  $r' := \Sigma_{k=1}^{j-1}$ Count($\mathcal{A}, k$) $\cdot$ Count($\mathcal{B}^{\{k\}}, n$)
  $r_\alpha :=$ Rank($\mathcal{A}, \alpha$); $r_\beta :=$ Rank($\mathcal{B}^{\{j\}}, \beta$)
  return $r' + r_\alpha \cdot$ Count($\mathcal{B}^{\{j\}}, n$) $+ r_\beta$

6.2 Algorithms for Unlabelled Classes

6.2.1 Basic Algorithms: + and ×

Ranking of unlabelled unions is solved as we did for labelled unions (Algorithm 19). In the same way to the ranking of partitional products, the ranking of Cartesian products computes the size of the first component, and the ranks of its components. For instance, the ranking of lexicographic Cartesian product is given in Algorithm 23. The algorithm for ranking in boustrophedonic order is similar to the labelled case (Algorithm 20) but without the binomial factors $\binom{n}{l}$; and we do not add r_ℓ at the end since we do not have labels now.

Algorithm 23 Ranking of products (lexicographic order)

Rank($\mathcal{A} \times \mathcal{B}, \langle \alpha, \beta \rangle$)
  $j := |\alpha|; n := |\langle \alpha, \beta \rangle|$
  $r' := \Sigma_{l=0}^{j-1}$ Count$(\mathcal{A}, l) \cdot$ Count$(\mathcal{B}, n-l)$
  $r_\alpha :=$ Rank$(\mathcal{A}, \alpha); r_\beta :=$ Rank$(\mathcal{B}, \beta)$
  return $r' + r_\alpha \cdot$ Count$(\mathcal{B}, n-j) + r_\beta$

6.2.2 Algorithms for Other Combinatorial Operators

Sequences are defined from unions and products, $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \times \mathsf{Seq}(\mathcal{A})$, hence, the ranking is recursively solved from the solutions we have for unions and products.

Powersets pose a more difficult task. The isomorphism used to unrank powersets follows the LScR order (other orders can be solved analogously) is:

$$\mathsf{PSet}(\mathcal{A}) \;=\; \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=\lfloor n/j \rfloor}^{1} \mathcal{A}_j^{\{k\}} \times \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \;.$$

We assume that the objects are given in the same format as the output of our unranking algorithms. In general, we should not assume that the user's inputs satisfy this type of assumption (e.g. the components of a set are given in canonic order); but in Chapter 9 we will make sure that the algorithms which transform the user's inputs given in linear form to the internal representation arrange the components if necessary to meet this requirement. That is, given $\langle \alpha, \beta \rangle \in \mathsf{PSet}_n(\mathcal{A})$, we assume that $\alpha = \langle \alpha_1, \langle \alpha_2, \langle \ldots, \langle \alpha_{k-1}, \alpha_k \rangle \ldots \rangle \rangle \rangle \in \mathcal{A}_j^{\{k\}}$ and $\beta \in \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})$, where the components of the leading block α are in increasing order by rank ($\alpha_1 \prec \alpha_2 \prec \ldots \prec \alpha_k$), and β follows the same format to the given object $\langle \alpha, \beta \rangle$. With these assumptions, we need to obtain the common size j of the components of α and the number k of components in α. Now (a) the powersets with at least one component of size strictly smaller than j are smaller than the given one, and (b) the powersets with more than k components of size j are also smaller than the given one. Recalling that $\hat{\delta}_{n,j}$ denotes the number of powersets with no components of size $< j$ (see Subsection 4.1.2), we have that the first quantity is $\hat{\delta}_{n,1} - \hat{\delta}_{n,j+1}$, and the second one is given by $\sum_{l=\lfloor n/j \rfloor}^{k+1} \binom{a_j}{l} \hat{\delta}_{n-lj,j+1}$. Once we have computed the number of powersets in (a) and (b), call it r' (see Line (6) in Algorithm 24), we have to compute the ranking of the given object $\langle \alpha, \beta \rangle$ within $\mathcal{A}_j^{\{k\}} \times \mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})$, which is solved as any other Cartesian product. In Algorithm 24 we show the complete algorithm for ranking powersets.

First we need the rank of α among the $\mathsf{PSetBlock}$'s of k components of size j whose ranks are at least i within $\mathcal{A}_j$; this is computed by the call to *RankPSetBlock* in line (7) (Algorithm 24). Afterwards, we need the rank of β within $\mathsf{PSet}(\mathcal{A}_{>j})$ (see line (8)); this is the same to the rank within $\mathsf{PSet}(\mathcal{A})$ minus the number of powersets of $\mathcal{A}$ whose components of the leading block have size smaller than j, i.e. $\mathit{RankPSet}(\mathcal{A}, j, \beta) = \mathit{Rank}(\mathsf{PSet}(\mathcal{A}), \beta) - (\hat{\delta}_{m,1} - \hat{\delta}_{m,j+1})$, where $m = |\beta|$.

$\mathit{RankPSetBlock}(\mathcal{A}, j, k, i, \alpha)$ has the additional parameter i to specify that the rank of α is to be computed for $\mathsf{PSetBlock}$'s of k components of size j whose ranks are at least i within

Algorithm 24 Ranking of powersets

1. $\text{Rank}(\mathsf{PSet}(\mathcal{A}), \langle \alpha, \beta \rangle)$
2. $\quad n := |\langle \alpha, \beta \rangle|$; **if** $n = 0$ **then return** 0 **fi**
3. $\quad j :=$ size of the components α_i of $\alpha = \langle \alpha_1, \langle \ldots, \langle \alpha_{k-1}, \alpha_k \rangle \ldots \rangle \rangle$
4. $\quad k := \#$ of components in α
5. $\quad \triangleright$ See items (a) + (b) in page 97 $\triangleleft$
6. $\quad r' := (\hat{\delta}_{n,1} - \hat{\delta}_{n,j+1}) + \Sigma_{k+1 \le l \le \lfloor n/j \rfloor} \binom{\text{Count}(\mathcal{A},j)}{l} \cdot \hat{\delta}_{n-lj,j+1}$
7. $\quad r_\alpha := \text{RankPSetBlock}(\mathcal{A}, j, k, 0, \alpha)$
8. $\quad r_\beta := \text{RankPSet}(\mathcal{A}, j, \beta)$
9. $\quad$ **return** $r' + r_\alpha \cdot \hat{\delta}_{n-kj,j+1} + r_\beta$

$\mathcal{A}_j$. This additional parameter allows us to give a recursive solution. See Algorithm 25, where we use an isomorphism similar to isomorphism (4.2):

$$\{\alpha' \in \mathcal{A}_j \mid \alpha' \succ \alpha\} \times \mathsf{PSet}(\{\alpha'' \in \mathcal{A}_j \mid \alpha'' \succ_{\mathcal{A}_j} \alpha'\}, \text{card} = k - 1) \ .$$

The algorithm almost writes itself. The only "tricky" part is to notice that the number of powersets that can be constructed out of k elements of size j with ranks not smaller than that of α' is $\binom{r_{\alpha'}}{k}$, where $r_{\alpha'}$ denotes the rank of α' within $\mathcal{A}_j$. Notice that the call in line (7) (Algorithm 24) puts no restriction on the rank of the components by setting $i = 0$.

Algorithm 25 Ranking of powersets of $\mathcal{A}$ with k components of size j with components of rank larger than i

$\text{RankPSetBlock}(\mathcal{A}, j, k, i, \langle \alpha', \alpha'' \rangle)$
$\quad i' := \text{Rank}(\mathcal{A}, \alpha'); r_{\alpha'} := i' - i$
$\quad$ **if** $k = 1$ **then return** $\binom{r_{\alpha'}}{k}$
$\quad$ **else return** $\binom{r_{\alpha'}}{k} + \text{RankPSetBlock}(\mathcal{A}, j, k, i', \alpha'')$
$\quad$ **fi**

Similar algorithms solve the ranking of sets, but now allowing repetitions among the components; for instance, we should use $\delta_{n,j}$ instead of $\hat{\delta}_{n,j}$ in Algorithm 24, and we should use $\binom{x+y-1}{y}$ instead of $\binom{x}{y}$ in Algorithms 24 and 25.

Sequences, powersets and sets with restricted cardinality follow the isomorphisms of Table 2.5. Hence, the ranking of sequences with restricted cardinalities are based upon the ranking of products; while the ranking of powersets and sets with restricted cardinalities follow similar steps to the ranking of unrestricted powersets and sets (e.g., in Algorithm 24 we just have to consider that β has also restricted cardinality).

The variant with card-lexicographic order follows the same ideas as in the labelled case (see Subsection 6.1.2).

For unlabelled substitutions $\mathcal{A} \circ \mathcal{B} = \bigcup_{k>0} \mathcal{A}_k \times \mathcal{B}^k$, the ranking is quite similar to the labelled case because of the isomorphisms are basically the same; it is just necessary to use sequences with fixed cardinality instead of sets with fixed cardinality in Algorithm 22.

6.3 Analysis of the Cost

Given the objects either in the 2- or 3-tuple representation, if we assume that the cost of the function is_a is constant then it is quite clear than the cost of ranking has the same behavior as the cost of unranking because the ranking algorithms just work as their unranking counterparts. This is summarized in the following theorem.

Theorem 20. *If the cost of the function is_a$(\mathcal{A}, \alpha)$ is constant, the complexity (either in the worst or in the average case) of ranking objects in $\mathcal{A}$ is of the same order of magnitude as the complexity of unranking for that class.*

This means that Theorems 9, 10 and 11 for the labelled case, their counterparts (Theorems 16, 17 and 18) for the unlabelled case, and the Υ-rules in Chapters 3 and 4 apply.

6.4 Membership

Given an object α and a class $\mathcal{A}$, the function is_a checks whether the given object belongs to the given class or not[1]. Function is_a typically performs recursive calls for the subobjects a given object is made of. For instance, given a labelled object $\langle \alpha, \beta, \ell \rangle$ and a class $\mathcal{A} \star \mathcal{B}$ it is necessary to check that α belongs to $\mathcal{A}$, β belongs to $\mathcal{B}$ and the ℓ-th labelling has a correct rank. Recall that we assume that in the case of powersets, sets, etc. we are assuming that the 2-tuples are in canonic order.

Algorithm 26 Function is_a for labelled classes

is_a$(\mathcal{C}, \gamma)$
  if $(\gamma = \epsilon$ **and** $\mathcal{C} = \{\epsilon\})$ **or** $(\gamma = Z$ **and** $\mathcal{C} = \{Z\})$ **then return** true
  elif $\gamma = \epsilon$ **and** $(\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{Set}(\mathcal{A}))$ **then return** true
  elif $\mathcal{C} = \mathcal{A} + \mathcal{B}$ **then return** (is_a$(\mathcal{A}, \gamma)$ **or** is_a$(\mathcal{B}, \gamma)$)
  elif $\mathcal{C} = \mathcal{A} \star \mathcal{B}$ **and** $\gamma = \langle \alpha, \beta, \ell \rangle$ **then**
      return $\left(\text{is\_a}(\mathcal{A}, \alpha) \textbf{ and } \text{is\_a}(\mathcal{B}, \beta) \textbf{ and } 0 \le \ell < \binom{|\gamma|}{|\alpha|}\right)$
  elif $\mathcal{C} = \mathcal{A}\, {}^{\square}\!\!\star \mathcal{B}$ **and** $\alpha = {}^{\square}\langle \alpha, \beta, \ell \rangle$ **then**
      return $\left(\text{is\_a}(\mathcal{A}, \alpha) \textbf{ and } \text{is\_a}(\mathcal{B}, \beta) \textbf{ and } 0 \le \ell < \binom{|\gamma|-1}{|\alpha|-1}\right)$
  elif $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **then return** is_a$(\mathcal{A} \star \mathcal{C}, \gamma)$
  elif $\mathcal{C} = \mathsf{Set}(\mathcal{A})$ **then return** is_a$(\mathcal{A}\, {}^{\square}\!\!\star \mathcal{C}, \gamma)$
  elif $\mathcal{C} = \mathsf{Cycle}(\mathcal{A})$ **then return** is_a$(\mathcal{A}\, {}^{\square}\!\!\star \mathsf{Seq}(\mathcal{A}), \gamma)$
  elif $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$ **and** $\gamma = {}^{\circ}\langle \alpha, \beta \rangle$ **then**
      return $(|\alpha| = (\#\text{ of components in } \beta) \textbf{ and } \text{is\_a}(\mathcal{A} \times \mathsf{Set}(\mathcal{A}), \langle \alpha, \beta \rangle))$
  else return false
  fi

---

[1] To be more precise, we check if the representation can be derived from the given "grammar" (specification) of the class. Thus, $\langle Z, \langle Z, Z, 1\rangle, 1\rangle$ is recognized as an object in $\mathsf{Seq}(Z)$ but not as an object in $\mathsf{Set}(\mathsf{Cycle}(Z))$; even though $\mathsf{Seq}(Z) \simeq \mathsf{Set}(\mathsf{Cycle}(Z))$.

For the labelled case, there are different cases depending on whether $\mathcal{A}$ is the ϵ-class, an atomic class, a union, a partitional or boxed product, a sequence, a set or a cycle. From the isomorphisms considered in Chapter 2 we can establish Algorithm 26. Sequences, sets and cycles with restricted cardinalities can be checked using the function is_a for the unrestricted case and then we have also to check if the object satisfies the restriction on the cardinality.

For the unlabelled case, we have a similar algorithm but without labels, and taking care that the decomposition of powersets and sets is as given in Subsection 6.2.2. See Algorithm 27, where we have omitted unlabelled cycles because we have not provide solutions neither for their unranking nor for their ranking. Finally, sequences, powersets and sets with restricted cardinalities can also be considered in a similar way, checking that the cardinality restriction is satisfied.

Algorithm 27 Function is_a of unlabelled classes

is_a($\mathcal{C}, \gamma$)
  if $(\gamma = \epsilon$ **and** $\mathcal{C} = \{\epsilon\})$ **or** $(\gamma = Z$ **and** $\mathcal{C} = \{Z\})$ **then return** true
  elif $\gamma = \epsilon$ **and** $(\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{PSet}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{Set}(\mathcal{A}))$ **then return** true
  elif $\mathcal{C} = \mathcal{A} + \mathcal{B}$ **then return** (is_a($\mathcal{A}, \gamma$) **or** is_a($\mathcal{B}, \gamma$))
  elif $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ **and** $\gamma = \langle \alpha, \beta \rangle$ **then return** (is_a($\mathcal{A}, \alpha$) **and** is_a($\mathcal{B}, \beta$))
  elif $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **then return** is_a($\mathcal{A} \times \mathsf{Seq}(\mathcal{A}), \gamma$)
  elif $(\mathcal{C} = \mathsf{PSet}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{Set}(\mathcal{A}))$ **and** $\gamma = \langle \alpha, \beta \rangle$ **then**
    $k :=$ # of components in α
    $j :=$ size of the components α_i of $\alpha = \langle \alpha_1, \langle \ldots, \langle \alpha_{k-1}, \alpha_k \rangle \ldots \rangle \rangle$
    ▷ Notice that $\alpha \in \mathcal{A}_j^{\{k\}}$ or $\alpha \in \mathcal{A}_j^{\{\{k\}\}}$ ◁
    for i **from** 1 **to** k **do if not** is_a($\mathcal{A}, \alpha_i$) **then return** false **fi end**
    return is_a($\mathcal{C}, \beta$)
  elif $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$ **and** $\gamma = {}^\circ\langle \alpha, \beta \rangle$ **then**
    return $(|\alpha| =$ (# of components in β) **and** is_a($\mathcal{A} \times \mathsf{Seq}(\mathcal{A}), \langle \alpha, \beta \rangle$))
  else return false
  fi

Given an object α and a class $\mathcal{A}$, function is_a usually needs to check whether the subobjects of α belong to the corresponding subclass or not. That is, the function is_a finishes when either one subobject does not belong to a specific subclass or all subobjects have correctly been checked. For instance, if we want to check if the labelled object $\langle Z, \langle Z, Z, 1 \rangle, 0 \rangle$ belongs to $\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$, we recursively have to check if Z belongs to $\mathcal{B}$, $\langle Z, Z, 1 \rangle$ belongs to $\mathcal{B} \star \mathcal{B}$ and $0 \le 0 < \binom{3}{2}$, and so on. In the same vein, if we want to check if $\langle Z, \langle Z, Z \rangle \rangle$ belongs to $\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}$, we recursively need to check if Z belongs to $\mathcal{B}$ and $\langle Z, Z \rangle$ belongs to $\mathcal{B} \times \mathcal{B}$, and so on.

It is by now clear that the cost is at most linear in the size of the given object. On the other hand, if $\alpha \in \mathcal{A}$, we will need to examine all atoms and thus the cost will be actually linear.

Theorem 21. *Given an α object and a class $\mathcal{A}$, it can be checked whether α belongs to $\mathcal{A}$ or not in $\mathrm{O}(|\alpha|)$ steps. In the worst-case, the cost is $\Theta(|\alpha|)$.*

This can be seen as bad news, since that would increase substantially the complexity of ranking. However this is only because we are solving the ranking problem using top-down recursion and thus asking repeatedly about the membership of subobjects. The inefficiency can be hence easily overcome by adapting the algorithms presented in this section so that given an object in the conventional linear representation or in tuple form they "tag" all the subobjects. Therefore, once applied this preprocessing step with cost $\Theta(n)$, the calls to is_a in the ranking algorithms can be replaced by efficient ($\Theta(1)$) checks to the appropriate tags and then the complexity of ranking satisfies Theorem 20.

6.5 Conclusions and Final Remarks

In this chapter we have shown that the problem of ranking can be efficiently solved drawing upon the same ideas that we have used to solve the problem of unranking. The number of steps to know the rank of a given object is proportional to the number of steps to unrank the corresponding rank; hence, the cost is asymptotically equivalent (Theorem 20). However, given an object α from a union $\mathcal{A} + \mathcal{B}$ the ranking needs to know if α belongs to $\mathcal{A}$ or to $\mathcal{B}$. This implies the need of a function is_a (Section 6.4) or of some specific representation with 0-sized tags.

One important issue here concerns the representation of combinatorial objects: on the one hand, the human-readable form (called the *Linear Normal Form*, LNF), which is very convenient for both input and output, is not the most convenient to handle; on the other hand, the 2- and 3-tuples that we have assumed here, make the algorithms easier but are not "user-friendly". In the chapters of Part III and in Chapter 9 we will present an even more rich and ellaborate representation for combinatorial objects called *Internal Normal Form* (INF). In Chapter 9, we will specifically address the problems of converting one form of representation into others in an efficient way. When computing the INF-form of an object we compute the ranks along the way (using a slight variation of the techniques presented in this chapter), so once the INF-form has been computed, the rank of the object has already been computed. That is, the ranking becomes a by-product of the transformation of LNF to INF of a given object.

Part III

Exhaustive Generation

The exhaustive generation of all objects of a given size in a combinatorial class has been the subject of intense research for already four decades (see for instance [70, 45, 67, 86]). Many efforts are still devoted in the quest for efficient algorithms for combinatorial classes for which no such algorithms are known, as well as to the improvement of existing algorithms.

Efficient algorithms exist for the exhaustive generation of permutations, trees, strings, cycles of k beads, partitions, subsets of k elements out of n, etc. (see, for instance, [10, 45, 67]). Although it is not always possible to attain it, the ultimate goal in efficiency is to generate all N objects of size n of a given class in time proportional to N; in other words, to achieve constant amortized time (CAT) per generated object. Sometimes it is desired to generate the objects in a particular order, for instance, permutations in lexicographic order or bitstrings in Gray order, but in many other situations we can choose whatever generation order yields the best performance. But in any case the vast majority of results in this area concern particular combinatorial classes or more or less restricted families of them.

In this part we follow a rather different approach, since our goal is to develop a generic algorithms to solve the exhaustive generation problem when the class of the objects to be generated is a parameter itself. Other generic approaches, such as the ECO method (see for instance [2, 3, 4, 52]) have been proposed, but they differ substantially with the one presented here, each approach imposing different requirements on the classes that can be generated, the way to express them, etc. In particular, as it happened with the unranking and ranking algorithms of the previous part, our generating algorithms need to be able to count how many objects there are in a given combinatorial class, hence the combinatorial classes we can deal with are those built using the so-called admissible operators.

One of the main contributions of this part is to show the viability of the generic approach of Flajolet *et al.* [26] and of Chapters 3 and 4 to cope also with the exhaustive generation problem, providing an elegant, effective and efficient solution. Examples of combinatorial classes which can be exhaustively generated using our approach include permutations, functional graphs, integer partitions, integer compositions, surjections, regular languages, simply generated families of trees (including binary trees, Cayley trees and plane general trees, among others), Dyck paths, etc. For instance, to list all partitions of the number n we could supply our algorithm with the specification $\mathcal{P} = \mathsf{Set}(Z \times \mathsf{Seq}(Z))$, and ask it to exhaustively generate all objects of size n in the class $\mathcal{P}$.

Since the algorithms described in this part are generic they do not take advantage of particular features of the combinatorial class whose objects we want to generate. However, the algorithms achieve constant amortized time per generated object for a large family of (labelled and unlabelled) admissible classes, including the most interesting ones, achieve constant amortized time per generated object, and thus are competitive when compared with algorithms specifically designed for the corresponding class. The fact that our algorithms exhibit CAT performance for a large family of admissible classes is one of the main results.

The interest that such generic algorithms can have is well exemplified by the following situation: the `combstruct` package provides efficient routines for counting and random generation given a finite specification of a combinatorial class; it also provides a routine `allstructs` which produces a list of all combinatorial objects of a given size. But, except for a few particular cases where `allstructs` calls a specialized efficient algorithm, the task is performed by repeatedly drawing objects at random until all the objects of size n have been generated. Of course, this is useless if the objects have to be processed in some specific order as they are generated (we cannot wait to have all them and sort them at the end) or different executions have to generate the objects in the same order. Furthermore, it is usually not as efficient as using an algorithm specifically designed for exhaustive generation, like the ones here.

This part is organized in two chapters, where we study the exhaustive generation of labelled and unlabelled admissible classes, respectively. These two chapers are organized in a similar way as Chapters 3 and 4 on unranking. In Chapter 7 we present the concept of *iterator*, the generation algorithms of labelled admissible classes, variants and heuristics that use *finger pointers* to improve the performance and, finally, we prove these algorithms work in constant amortized time (CAT) per generated object for a large family of admissible classes. Chapter 8 has analogous sections but they dealt with unlabelled admissible classes.

The results of this part have been presented in the extended abstracts *Generic Algorithms for the Exhaustive Generation of Labelled Objects* [55], *Iterating Through Labelled Combinatorial Structures* [64], *Generic Algorithms for the Generation of Combinatorial Objects* [57] and *An Efficient Generic Algorithm for the Generation of Unlabelled Cycles* [58]. The paper *Efficient Iteration in Admissible Combinatorial Classes* [60] collects the results in [55, 57, 64].

Chapter 7

Generation of Labelled Classes

In this chapter we will study the exhaustive generation of labelled objects. We design efficient general algorithms and analyze their performance for labelled admissible classes. Given a specification of a combinatorial class ($\mathcal{A}$) and a size (n), the exhaustive generation produces all objects of size n in the given class $\mathcal{A}$, without repetition. There are many specific algorithms for the exhaustive generation of combinatorial objects like binary trees [51, 71, 72, 73, 74], permutations [77, 84], strings [12] and many other (see [13, 29, 39, 46, 47, 67, 70, 85, 86] or *the Combinatorial Object Server* [10]). But the goal of our algorithms is that they can be used for any admissible labelled or unlabelled class (see next chapter for the generation of unlabelled objects).

Before we proceed, let us briefly summarize the classes and the orderings that we will consider here. For unions $\mathcal{A}+\mathcal{B}$, the $\mathcal{A}$-objects come first, then the $\mathcal{B}$-objects. For partitional products $\mathcal{A}\star\mathcal{B}$, we have the lexicographic order or the boustrophedonic order depending on whether we order objects in increasing order of the size of their first component, i.e., when we assume the order induced by the specification $(\mathcal{A}\star\mathcal{B})_n = \mathcal{A}_0\star\mathcal{B}_n+\mathcal{A}_1\star\mathcal{B}_{n-1}+\mathcal{A}_2\star\mathcal{B}_{n-2}+\ldots$, or we order objects in alternating order of the size of their first component, i.e., when we assume the order induced by the specification $(\mathcal{A}\star\mathcal{B})_n = \mathcal{A}_0\star\mathcal{B}_n+\mathcal{A}_n\star\mathcal{B}_0+\mathcal{A}_1\star\mathcal{B}_{n-1}+\ldots$. However, we just use the lexicographic order (see [55]) since the boustrophedonic order is less intuitive and it does not yield better performance for exhaustive generation. Moreover, since we are considering labelled objects, we have two orders depending on whether the criterion of higher priority is the labelling (PF order, e.g. see Figure 7.1) or the structure of the object (SF order, see Figure 7.2). For boxed products we use the same ideas as for partitional products. On the other hand, the generation of sequences, sets and cycles follows from the isomorphisms $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A}\star\mathsf{Seq}(\mathcal{A})$, $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}\ {}^{\square}\!\!\star\ \mathsf{Set}(\mathcal{A})$ and $\mathsf{Cycle}(\mathcal{A}) = \mathcal{A}\ {}^{\square}\!\!\star\ \mathsf{Set}(\mathcal{A})$, respectively. Finally, the exhaustive generation for substitutions are based of upon the isomorphism $\mathcal{A}\circ\mathcal{B} = \bigcup_{k>0}\mathcal{A}_k\times\mathcal{B}^{\{k\}}$.

This chapter is organized as follows. In Section 7.1 we define the concept of *iterator* to represent the objects to be manipulated by the algorithms of exhaustive generation. In the following section we describe the generic algorithms that we propose for the generation of labelled classes. In Section 7.3, we present variants and heuristics, and we introduce the use of *fingers* to improve the performance of the generation algorithms. The last section is devoted to the analysis of the performance of these algorithms, and some examples are also presented.

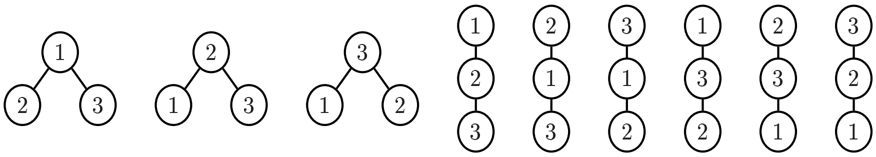


Figure 7.1: Cayley trees of size 3 in PF order.

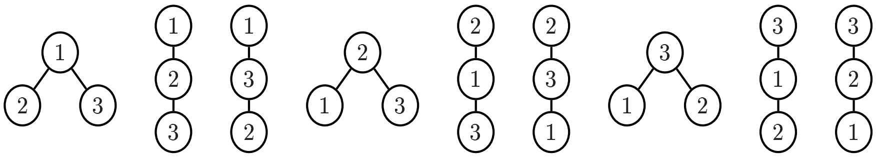


Figure 7.2: Cayley trees of size 3 in SF order.

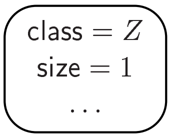
The results of this chapter have been mainly presented in the papers [55, 60]. Additional results appear in [57, 64]. The paper [60] also presents most results on generation of unlabelled classes (see Chapter 8).

7.1 Iterators

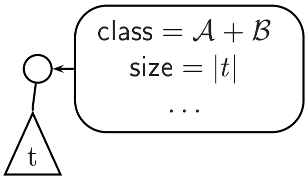
An *iterator* is a pair $\langle \alpha, p \rangle$ where α is the representation of a combinatorial object (the *current* object) and p records additional information to help the generation of $next(\alpha)$, that is, the object following α. Alternatively, we might think of an iterator as a "pointer" to the ordered list of objects of size n in some class $\mathcal{A}$ (of course, this is just an alternative abstract view of iterators).

An iterator contains a representation of a combinatorial object as a *tree structure*, but also additional information which helps and speeds up the generation process. Most of this additional information is also organized in the tree structure and reflects that of the corresponding combinatorial object. Hence each node contains information about the class, the size, the rank, the labelling of the "subobject" rooted at the node, etc. The information that is needed to initialize an iterator is a class ($\mathcal{A}$), an specification (set of equations $\mathcal{A}_j = \Psi(\mathcal{A}_{j1}, \mathcal{A}_{j2}, \ldots, \mathcal{A}_{kj})$ which defines the class $\mathcal{A}$) and a size (n). The iterator contains the tree structure where each node has attributes which describe the class, size, rank, etc. of the subobject represented by the subtree rooted at the node. If we denote $ts(\mathcal{A}, \gamma)$ the tree structure representing the object γ in the class $\mathcal{A}$ then

1. $ts(\{\epsilon\}, \epsilon) = \boxed{\begin{array}{c} \mathsf{class} = \epsilon \\ \mathsf{size} = 0 \\ \ldots \end{array}}.$

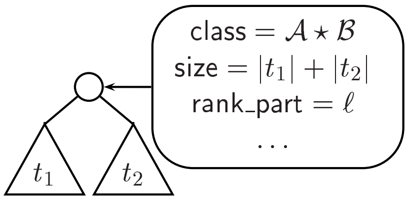
2. For any atom Z, $ts(\{Z\}, Z) =$ 

.

3. If $\gamma \in \mathcal{A} + \mathcal{B}$ then $ts(\mathcal{A} + \mathcal{B}, \gamma) =$ 

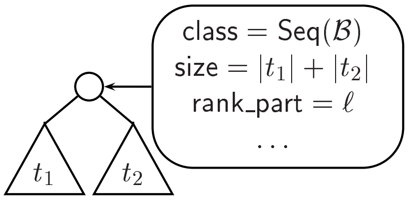
,

   where $t = ts(\mathcal{A}, \gamma)$ or $t = ts(\mathcal{B}, \gamma)$ depending on whether $\gamma \in \mathcal{A}$ or $\gamma \in \mathcal{B}$, respectively.

4. If $\gamma \in \mathcal{A} \star \mathcal{B}$ then $ts(\mathcal{A} \star \mathcal{B}, \gamma) =$ 

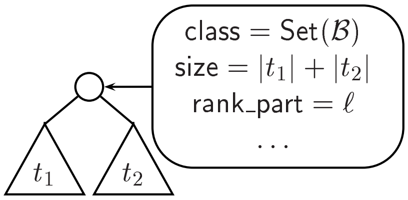
,

   where $\gamma = \langle \alpha, \beta, \ell \rangle \in \mathcal{A} \star \mathcal{B}$, $\alpha \in \mathcal{A}$, $\beta \in \mathcal{B}$, $t_1 = ts(\mathcal{A}, \alpha)$ and $t_2 = ts(\mathcal{B}, \beta)$, for $n = |\gamma|$ and $j = |\alpha|$.

5. $ts(\mathsf{Seq}(\mathcal{A}), \gamma) =$ 

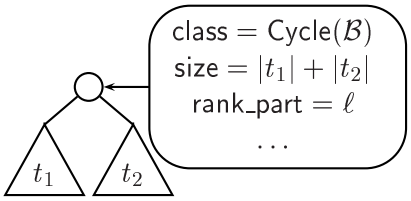
,

   where $\gamma = \langle \alpha, \beta, \ell \rangle \in \mathsf{Seq}(\mathcal{A})$, $\alpha \in \mathcal{A}$, $\beta \in \mathsf{Seq}(\mathcal{A})$, $t_1 = ts(\mathcal{A}, \alpha)$ and $t_2 = ts(\mathsf{Seq}(\mathcal{A}), \beta)$, for $n = |\gamma|$ and $j = |\alpha|$.

6. $ts(\mathsf{Set}(\mathcal{A}), \gamma) =$ 

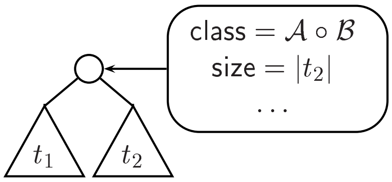
,

   where $\gamma = {}^{\square}\langle \alpha, \beta, \ell \rangle \in \mathcal{A} \, {}^{\square}\!\star \, \mathsf{Set}(\mathcal{A})$, $\alpha \in \mathcal{A}$, $\beta \in \mathsf{Set}(\mathcal{A})$, $t_1 = ts(\mathcal{A}, \alpha)$ and $t_2 = ts(\mathsf{Set}(\mathcal{A}), \beta)$, for $n = |\gamma|$ and $j = |\alpha|$.

7. $ts(\mathsf{Cycle}(\mathcal{A}), \gamma) =$ 

,

   where $\gamma = {}^{\square}\langle \alpha, \beta, \ell \rangle \in \mathcal{A} \, {}^{\square}\!\star \, \mathsf{Seq}(\mathcal{A})$, $\alpha \in \mathcal{A}$, $\beta \in \mathsf{Seq}(\mathcal{A})$, $t_1 = ts(\mathcal{A}, \alpha)$ and $t_2 = ts(\mathsf{Seq}(\mathcal{A}), \beta)$, for $n = |\gamma|$ and $j = |\alpha|$.

8. $ts(\mathcal{A} \circ \mathcal{B}, \gamma) =$ 

,

   where $\gamma = {}^{\circ}\langle \alpha, \beta \rangle \in \mathcal{A}_k \times \mathcal{B}^{\{k\}}$, for some integer k, $\alpha \in \mathcal{A}_k$, $\beta \in \mathcal{B}^{\{k\}}$, $t_1 = ts(\mathcal{A}, \alpha)$ and $t_2 = ts(\mathcal{B}^{\{k\}}, \beta)$, for $n = |\gamma|$ and $j = |\alpha|$.

To initialize an iterator for labelled generation, we use the function *Init_Iter* which returns the iterator *itr* that contains the tree structure corresponding to the first object of size n in the given class:

$$itr := Init\_Iter([\mathsf{class}, \mathsf{spec\_equations}, \mathsf{labelled}], n) \ .$$

For the time being we consider that *itr* is a tuple with four fields called perm, inv_perm, root and spec_equations, which are the permutation that labels the atoms of the object and its inverse, a pointer to the root of the current tree structure and the specification that defines the given class[1]. Later, in Section 7.3, we will add several fields, like a *finger pointer*, for better performance.

There are also other three functions used for the exhaustive generation: *Is_Last*, *Get_Obj* and *Next*.

1. *Is_Last*(*itr*) returns true if the iterator corresponds to the next of the last object α of size n and returns false otherwise; i.e., the iterator represents a fictitious sentinel which "marks" the end of the sequence of objects of size n (this is usually known as *past-the-end* convention).

2. *Get_Obj*(*itr*) retrieves the combinatorial object from the given iterator, in order to print, manage or process it, as needed. This function traverses the corresponding tree structure that the iterator *itr* stores, to produce a conventional linear representation of the current combinatorial object. If *Is_Last*(*itr*) is true, then *Get_Obj*(*itr*) signals an **error**.

3. *Next*(*itr*) computes the next object by updating the information of the nodes in the tree structure that represents the current combinatorial object in the iterator *itr*. Obviously, next depends on the order in which we want to generate the objects (PF, SF, lexicographic, etc). If *Is_Last*(*itr*) $=$ true, then *Next*(*itr*) does nothing. This, for obvious reason, the most relevant function and the main subject of the two chapters of this Part.

These functions allow a complete and exhaustive traversal of all objects using a simple loop like the one given in Algorithm 28. For instance, if we call *print*(*Get_Obj*(*it*)) applied to the first binary tree, $\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$, of size 4 then it returns $\mathsf{Prod}(Z_1, \mathsf{Prod}(Z_2, \mathsf{Prod}(Z_3, Z_4)))$, and if applied to the first labelled Cayley tree, $\mathcal{T} = Z \star \mathsf{Set}(\mathcal{T})$ of size 4 then it returns $\mathsf{Prod}(Z_1, \mathsf{Set}(\mathsf{Prod}(Z_2, \mathsf{Set}(\mathsf{Prod}(Z_3, \mathsf{Set}(\mathsf{Prod}(Z_4, \epsilon)))))))$.

[1]Fields class and size in *itr* to store the class and size, respectively, are not necessary because they are already accessible in *itr*.root$\to$class and *itr*.root$\to$size, respectively.

Algorithm 28 Loop for the exhaustive generation

itr := Init_Iter([class, spec_equations, labelled], n)
while not Is_Last(itr) **do**
    print(Get_Obj(itr)); Next(itr)
end

7.2 Implementation and Algorithms

This section details how functions *Init_Iter*, *Is_Last* and *Next* work. We have already discussed which information has to be supplied to create an iterator *itr*, namely the specification, the class and the size. Moreover, we also need additional information (attributes) to generate successive objects. We use p to denote a pointer to a node and $p \rightarrow$ Field to refer to the attribute Field in the node pointed to by p. The full list of attributes of each node is:

- class: The class where the object represented by the subtree belongs to, say the class $\mathcal{A}$.
- size: The size n of the object represented by the subtree.
- count: Number of labelled objects in $\mathcal{A}_n$ where $\mathcal{A}$ is the class of the node and n is the size of the subobject represented by the node.
- rank: Rank of the subobject rooted at the current node (it goes from 0 to count-1).
- ind_perm: *itr*.perm and *itr*.inv_perm store the labelling of the full object and we will need to store a new field ind_perm pointing to the appropriate place of the *itr*.perm and *itr*.inv_perm arrays at each node of tree.
- part: Current labelling partition; that is, two subsets P_1 and P_2, where P_1 and P_2 have the labels of the atoms of the left and right subobjects, respectively. For convenience, the labels of these two subsets are in increasing order and they are stored in a single array; i.e., part $= [p_1, \ldots, p_n]$, where $n =$ size.
- inv_part: Inverse of part, i.e., inv_part[part[k]] $= k$ for $1 \le k \le$ size.
- count_part: Number of consistent relabellings (labelling partitions) that can be applied to the object rooted at the node; in particular, for a node representing a partitional product the value of this field is $\binom{n}{j}$, where n is the size of the object rooted at the node and j is the size of the left subobject; for a node representing a boxed product the value is $\binom{n-1}{j-1}$.
- rank_part: Rank of the current labelling partition part (it goes from 0 to count_part-1).
- left/right: Each node has a pointer to the roots of its respective left and right subtrees. For instance, if the class of a specific node is $\mathcal{A}\star\mathcal{B}$, the left child (left) represents the $\mathcal{A}$-object (first component), while the right child (right) represents the $\mathcal{B}$-object (second component). For unions $\mathcal{A}+\mathcal{B}$, we just need one child; but, we will use the left child if the object actually belongs to $\mathcal{A}$, and the right child if the object belongs to $\mathcal{B}$.

Actually, functions *Init_Iter*, *Is_Last* and *Next* do most of their job through corresponding private functions called *Init_Obj*, *Is_Last_Obj* and *Next_Obj*, respectively. For instance, to compute the first Cayley tree of size 3 we execute

$$itr := Init\_Iter([\mathcal{T}, \{\mathcal{T} = Z \star \mathsf{Set}(\mathcal{T})\}, \mathsf{labelled}], 3)\ ,$$

which calls *Init_Obj*($\mathcal{T}, 3, 1, itr$) and returns the corresponding tree structure. Figure 7.3 depicts the contents of the iterator *itr* so created, and Algorithm 29 shows how *Init_Iter* works.

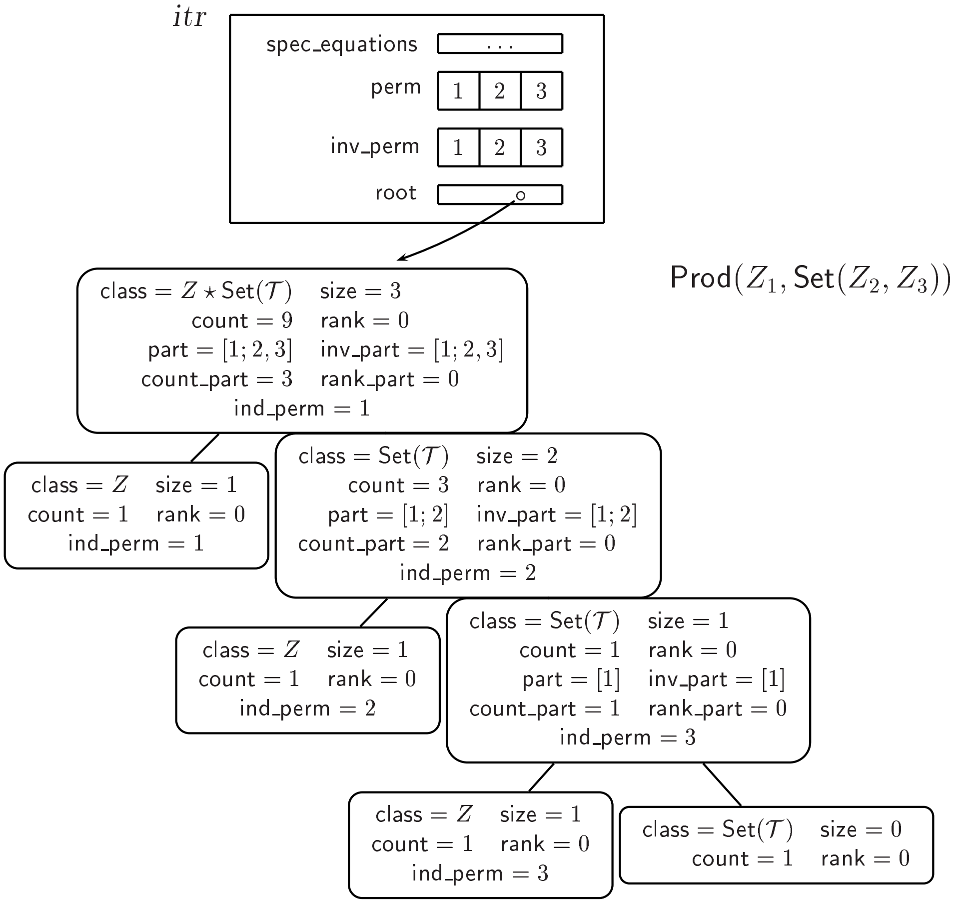


Figure 7.3: Iterator for the first Cayley tree of size 3.

Algorithm 29 Function *Init_Iter*

Init_Iter([class, spec_equations, labelled], n)
  Iterator itr ▷ Return a fresh, uninitialized iterator ◁
  itr.class := class; itr.spec_equations := spec_equations
  itr.root := Init_Obj(class, n, 1, itr)
  return itr

On the other hand, *Is_Last*(itr) returns true if and only if itr represents a sentinel, a fictitious object right after the last real object. This is easily checked: if $itr.\mathsf{root} \to \mathsf{rank} \geq itr.\mathsf{root} \to \mathsf{count}$ then the object is such a sentinel. The input of *Is_Last_Obj* is a pointer p and it returns true if and only if $p \to \mathsf{rank} \geq p \to \mathsf{count} - 1$ and returns false otherwise[2].

Basically, *Init_Iter*$([\mathsf{class}, \mathsf{spec\_equations}, \mathsf{labelled}], n)$ calls *Init_Obj*$(\mathsf{class}, n, 1, itr)$ in order to fill all attributes with initial values, where the third parameter '1' indicates the place of the first label of $itr.\mathsf{perm}$ and $itr.\mathsf{inv\_perm}$: In Algorithm 30 we show how these attributes are initialized. We note here that if the given size is 0, or the given class is $\mathcal{A} + \mathcal{B}$, or the given size is 1 and the given class is Z, then there are no labelling partitions to take care of (fields part, inv_part, count_part and rank_part have not to be initialized).

7.2.1 The Function *Next*

The function *Next* calls *Next_Obj*$(itr.\mathsf{Root}, itr)$ to do the real job. In turn, *Next_Obj*(p, itr) checks the class of the object pointed to by p and calls the appropriate routine to process that type of objects (see Algorithm 31). *Next_Obj* is trivial when applied to the ϵ and atomic objects, since there does not exist a next object. So, here we just study *Next_Union* and *Next_Prod* in the (lexicographic-) SF and PF orders.

For unions, we generate the next $\mathcal{A}$-object while it is possible, and we compute the first or the next $\mathcal{B}$-object otherwise. See Algorithm 32 for details, where *Init_Obj*$(\mathcal{A}, n, ip, itr)$ returns a pointer to the root of $ts(\mathcal{A}, \alpha_0)$, with α_0 being the first object of the class $\mathcal{A}$ of size n (see Algorith 30). Recall that if the current object $\gamma \in \mathcal{A} + \mathcal{B}$ actually belongs to $\mathcal{A}$ then it is stored as the left child of the root's node of γ; if it belongs to $\mathcal{B}$ then we store its representation as right child.

In some implementations, the replacement of subtree representing $\mathcal{A}_n$'s last object by $\mathcal{B}_n$'s first object will require that the nodes of the subtree $\mathcal{A}_n$ must be explicitly released back to the free storage. In other implementations we may have developed an special purpose memory allocator that recycles nodes. This is particularly useful for leaf nodes, since an object of size n will exactly contain n atom nodes and at most $n + 1$ ϵ-nodes. Hence, each time that we need a "new" leaf node we could actually reuse one that has just been released. However, in Algorithm 32 and for the rest of this thesis, we assume for simplicity that our "programming language" has some garbage collection system that will automatically take care of the unused nodes.

For partitional products, the main difference between the PF and SF orders is that in PF we iterate through all possible labelling partitions of an underlying pair of labelled objects before changing to the next pair, whereas SF fixes a labelling partition and iterates through all possible partitional pairs before changing to the next labelling partition. In Algorithms 33 and 34 we detail how the iteration through the partitional product works for the PF and SF orders, respectively.

For the PF order, let $j = p \to \mathsf{left} \to \mathsf{size}$, $n = p \to \mathsf{size}$ and $ip = p \to \mathsf{ind\_perm}$. If $p \to \mathsf{rank\_part} = p \to \mathsf{count\_part} - 1$ (line (6) in Algorithm 33) we have to reset the current partition to the initial partition $[1..j, j + 1..n]$ (line (7) in Algorithm 33). Reseting the partition means that we have to update the global permutation accordingly. In particular,

[2] Notice that *Is_Last_Obj*(p) returns true if the object pointed to by p is the last object in the class; so we do not follow the same convention for this function as for *Is_Last*.

Algorithm 30 Function *Init_Obj* (first version)

```
Init_Obj(C, n, ip, itr)
    p := New_Node ▷ Return a fresh, uninitialized node ◁
    p→class := C; p→size := n; p→count := Count(C, n); p→rank := 0
    if n > 0 then p→ind_perm := ip fi
    if C = Z then itr.perm[ip] := ip; itr.inv_perm[ip] := ip fi
    ▷ If (n = 0) or (C = A + B) or (n = 1 and C = Z), then fields ◁
    ▷     part, inv_part, count_part and rank_part are meaningless ◁
    if (n ≠ 0) and (C ≠ A + B) and (n ≠ 1 or C ≠ Z) then
       p→part := [1..n]; p→inv_part := [1..n]; p→rank_part := 0
     fi
     if (n = 0 and (C = ϵ or C = Seq(A) or C = Set(A))) or (n = 1 and C = Z) then
        p→left := NULL; p→right := NULL
     elif C = A + B then
          if Count(A, n) ≠ 0 then
             p→left := Init_Obj(A, n, ip, itr); p→right := NULL
          else p→left := NULL; p→right := Init_Obj(B, n, ip, itr)
          fi
     elif n = 0 and C = A ⋆ B then
          p→left := Init_Obj(A, n, ip, itr); p→right := Init_Obj(B, n, ip, itr)
     else if C = A ⋆ B then lC := A; rC := B; j := 0
          elif C = Seq(A) or C = Set(A) then lC := A; rC := C; j := 1
          elif C = Cycle(A) then lC := A; rC := Seq(A); j := 1
          fi
          while Count(lC, j) · Count(rC, n − j) = 0 do j := j + 1 end
          p→left := Init_Obj(lC, j, ip, itr)
          p→right := Init_Obj(rC, n − j, ip + j, itr)
          if C = A ⋆ B or C = Seq(A) then p→count_part := binom(n, j)
          elif C = Set(A) or C = Cycle(A) then p→count_part := binom(n−1, j−1)
          fi
    fi
    return p
```

if $j \le n-j$ then the j smallest labels in $itr.\mathsf{perm}[ip+j..ip+n-1]$ must be moved to $itr.\mathsf{perm}[ip..ip+j-1]$, respecting the order of the j labels that occupied this left subarray; similarly, the j labels in $itr.\mathsf{perm}[ip..ip+j-1]$ must be moved to $itr.\mathsf{perm}[ip+j..ip+n-1]$, also respecting the order of the $n-j$ labels that occupied this right subarray. This can be accomplished by simply decreasing the labels in the left by $n-j$ and increasing the labels in the right by j. We proceed in a similar vein if $j \ge n-j$. See the procedure *Update_Perm* in Algorithm 33.

Algorithm 34 for SF order works in a similar way as Algorithm 33. We have "highlighted" the common points of both Algorithms. Roughly speaking, Algorithm 34 is a simple rearrangement of the code of Algorithm 33.

Algorithm 31 Next labelled object of the node p (first version)

Next_Obj(p, itr)
    if Is_Last_Obj(p) **then** $p \to \mathsf{rank} := p \to \mathsf{rank} + 1$; **return fi**
    $p \to \mathsf{rank} := p \to \mathsf{rank} + 1$
    if $p \to \mathsf{class} = \mathcal{A} + \mathcal{B}$ **then** Next_Union(p, itr)
    elif $p \to \mathsf{class} = \mathcal{A} \star \mathcal{B}$ **then** Next_Prod(p, itr)
    ▷ Similar code to iterate through $\mathsf{Seq}(\mathcal{A})$, $\mathsf{Set}(\mathcal{A})$, etc. (see Algorithm 38) ◁
    fi

Algorithm 32 Next for disjoint unions

Next_Union(p, itr)
    if $p \to \mathsf{left} = \mathsf{NULL}$ **then** Next_Obj($p \to \mathsf{right}$, itr)
    elif not Is_Last_Obj($p \to \mathsf{left}$) **then** Next_Obj($p \to \mathsf{left}$, itr)
    else $p \to \mathsf{left} := \mathsf{NULL}$; $p \to \mathsf{right} :=$ Init_Obj($\mathcal{B}, p \to \mathsf{size}, p \to \mathsf{ind\_perm}$, itr)
    fi

Algorithm 34 also uses the function *Next_Lab* to generate successive labelling partitions; that is, *Next_Lab*(p, *itr*) computes the following labelling partition ($p \to \mathsf{part}$ and $p \to \mathsf{inv\_part}$), and updates the global permutation and the global inverse permutation (*itr*.perm and *itr*.$\mathsf{inv\_perm}$). This function uses an adaptation of the Nijenhuis & Wilf's [67] or Kemp's [41] algorithm to compute the next labelling partitions in constant amortized time (CAT). In the following subsection we give the details.

7.2.2 Updating the labelling

We complete now our description of the generation of partitional products giving the missing details of *Next_Lab*. The node p of the current object belonging to a product $\mathcal{A}_j \star \mathcal{B}_{n-j}$ stores the current labelling partition as an array of size n called Part. For instance, given $\alpha = Z_2 \star Z_1$ and $\beta = Z_3 \star (Z_2 \star Z_1)$, then their product admits $\binom{5}{2} = 10$ relabellings. To generate these 10 relabellings in $\alpha \star \beta$, we iterate through all the labelling partitions (field part), updating the corresponding attributes (field $\mathsf{inv\_part}$, and *itr*.perm and *itr*.$\mathsf{inv\_perm}$) on each iteration: See Table 7.1.

On the other hand, in Figure 7.4 we show how the tree structure of $(Z_4 \star Z_1) \star (Z_5 \star (Z_3 \star Z_2))$ is.

Our routine *Next_Lab* is based upon Kemp's algorithm [41] for the next j-subset (alternatively, Nijenhuis & Wilf's algorithm [67] could has also been similarly adapted). One of the modifications that we need is to update the arrays $p \to \mathsf{part}$, $p \to \mathsf{inv\_part}$, *itr*.perm and *itr*.$\mathsf{inv\_perm}$ for a specific node p that belongs to the iterator *itr*, and to reflect the swaps made in the new labelling. Basically, each time the algorithm swaps a label ℓ with a label

Algorithm 33 Next for products (lexicographic-PF order)

```
Next_Prod(p, itr)
    A := p→left→class; B := p→right→class
    if p→rank_part < p→count_part − 1 then
        Next_Lab(p, itr)  ▷ Next partition and permutation ◁
        p→rank_part := p→rank_part + 1; return
    else n := p→size; j := p→left→size; ip := p→ind_perm
          p→rank_part := 0; p→part := [1..n]; p→inv_part := [1..n]
    fi
    if not Is_Last_Obj(p→right) then Next_Obj(p→right, itr)
          Update_Perm(itr.perm, itr.inv_perm, ip, n, j)
     elif not Is_Last_Obj(p→left) then
          Next_Obj(p→left, itr); p→right := Init_Obj(A, n − j, ip + j, itr)
          Update_Perm(itr.perm, itr.inv_perm, ip, n, j)
     else j := j + 1
          while Count(A, j) · Count(B, n − j) = 0 do  j := j + 1 end
          p→left := Init_Obj(A, j, ip, itr); p→right := Init_Obj(B, n − j, ip + j, itr)
          p→count_part := (n choose j)
     fi

Update_Perm(perm, inv_perm, i, n, j)
      for k := i to i + j − 1 do
          perm[k] := perm[k] − (n − j); inv_perm[perm[k]] := k
      end
      for k := i + j to i + n − 1 do
          perm[k] := perm[k] + j; inv_perm[perm[k]] := k
      end
```

ℓ' in $p \to \mathsf{part}$ then we have to perform the following swaps:

$$\begin{array}{c|c}
il := p \to \mathsf{inv_part}[\ell] & il := itr.\mathsf{inv_perm}[\ell + ip - 1] \\
il' := p \to \mathsf{inv_part}[\ell'] & il' := itr.\mathsf{inv_perm}[\ell' + ip - 1] \\
p \to \mathsf{part}[il] \leftrightarrow p \to \mathsf{part}[il'] & itr.\mathsf{perm}[il] \leftrightarrow itr.\mathsf{perm}[il'] \\
p \to \mathsf{inv_part}[\ell] \leftrightarrow p \to \mathsf{inv_part}[\ell'] & itr.\mathsf{inv_perm}[\ell] \leftrightarrow itr.\mathsf{inv_perm}[\ell']
\end{array}$$

where $ip = p \to \mathsf{ind\_perm}$. But a more important modification is necessary to be able to iterate through labelling partitions; that is, through all the possible values for the attribute part. Both Kemp's and Nijenhuis & Wilf's algorithms maintain only the j-subset of the current labelling partition (the $(n-j)$-subset in the labelling partition is implicit), and they do it only for $j \le \lfloor n/2 \rfloor$. Thus we modify Kemp's algorithm in order to maintain the labelling partition for $0 \le j \le n$, while still guaranteeing that it works in constant amortized time. The situation where $j > \lfloor n/2 \rfloor$ is easily dealt with, we only need to reverse the rôles played by the first half and the second half of the array part. But maintaining the full labelling partition and being able to compute the next one in constant amortized time

Algorithm 34 Next for products (lexicographic-SF order)

```
Next_Prod(p, itr)
    A := p→left→class; B := p→right→class
    if not Is_Last_Obj(p→right) then ▷ See line (10) in Algorithm 33 ◁
        Next_Obj(p→right, itr)
    elif not Is_Last_Obj(p→left) then ▷ See line (13) in Algorithm 33 ◁
        Next_Obj(p→left, itr)
        p→right := Init_Obj(B, p→right→size, p→ind_perm, itr)
    elif p→rank_part < p→count_part − 1 then ▷ See lines (4)-(5) in Algorithm 33 ◁
        Next_Lab(p, itr) ▷ Next partition and permutation ◁
            p→rank_part := p→rank_part + 1; ip := p→ind_perm
            p→left := Init_Obj(A, p→left→size, ip, itr)
            p→right := Init_Obj(B, p→right→size, ip + j, itr)
            return
    else ▷ See lines (16)-(18) in Algorithm 33 ◁
            n := p→size; j := p→left→size + 1; ip := p→ind_perm
            while Count(A, j) · Count(B, n − j) = 0 do j := j + 1 end
            p→left := Init_Obj(A, j, ip, itr); p→right := Init_Obj(B, n − j, ip + j, itr)
            p→part := [1..n]; p→inv_part := [1..n]
            p→rank_part := 0; p→count_part := binom(n, j)
    fi
```

is more complicated; the array inv_part is a must here, because it allows us to know where and how do we have to update a label which "moves" from the j-subset to the $(n-j)$-subset or vice versa (in the attribute part).

The bulk of *Next_Lab* code is given in Algorithm 35. For simplicity, we have not included the code necessary to update arrays perm and inv_perm (we just consider the attributes part and inv_part); we have also omitted references to the node that stores the corresponding arrays and the updating of the attribute rank_part. The cost to generate the $\binom{n}{j}$ labelling partitions is proportional to $\binom{n}{j}$; hence, generating each labelling partition takes constant amortized time. The only difference in performance between Kemp's algorithm and Algorithm 35 comes from lines (14)-(16). A simple counting argument reveals that there are exactly $\binom{n-r-2}{n-r-j}$ labelling partitions for which the loop (14)-(16) (or its analogue for $j > \lfloor n/2 \rfloor$) makes exactly r iterations. Hence, the total cost for the $\binom{n}{j}$ labelling partitions contributed by these lines is proportional to

$$\sum_{r=2}^{n-j-1} r \binom{n-r-2}{n-r-j} = \frac{n+j-2}{n-j-2} \binom{n-3}{j} - (n-j) \ ,$$

which is $\mathrm{O}(\binom{n}{j})$. Thus, this algorithm works in CAT.

itr.root$\to$part	itr.root$\to$inv_part	itr.perm	itr.inv_perm	rank_part
$[1,2;3,4,5]$	$[1,2;3,4,5]$	$[2,1;5,4,3]$	$[2,1;5,4,3]$	0
$[1,3;2,4,5]$	$[1,3;2,4,5]$	$[3,1;5,4,2]$	$[2,5;1,4,3]$	1
$[1,4;2,3,5]$	$[1,3;4,2,5]$	$[4,1;5,3,2]$	$[2,5;4,1,3]$	2
$[1,5;2,3,4]$	$[1,3;4,5,2]$	$[5,1;4,3,2]$	$[2,5;4,3,1]$	3
$[2,3;1,4,5]$	$[3,1;2,4,5]$	$[3,2;5,4,1]$	$[5,2;1,4,3]$	4
$[2,4;1,3,5]$	$[3,1;4,2,5]$	$[4,2;5,3,1]$	$[5,2;4,1,3]$	5
$[2,5;1,3,4]$	$[3,1;4,5,2]$	$[5,2;4,3,1]$	$[5,2;4,3,1]$	6
$[3,4;1,2,5]$	$[3,4;1,2,5]$	$[4,3;5,2,1]$	$[5,4;2,1,3]$	7
$[3,5;1,2,4]$	$[3,4;1,5,2]$	$[5,3;4,2,1]$	$[5,4;2,3,1]$	8
$[4,5;1,2,3]$	$[3,4;5,1,2]$	$[5,4;3,2,1]$	$[5,4;3,2,1]$	9

Table 7.1: Some attributes generated for $\alpha \star \beta$, where $\alpha = Z_2 \star Z_1$ and $\beta = Z_3 \star (Z_2 \star Z_1)$.

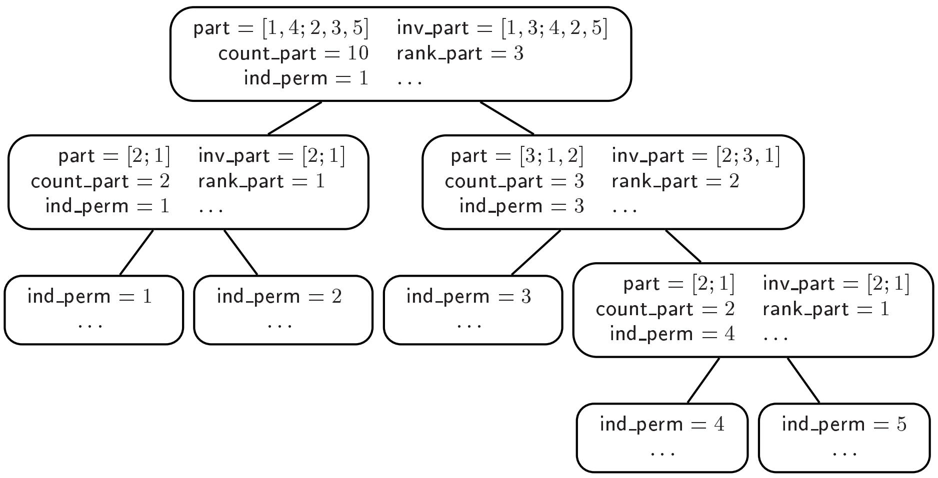


Figure 7.4: Iterator for $(Z_4 \star Z_1) \star (Z_5 \star (Z_3 \star Z_2))$.

7.2.3 Algorithms for Containers

We start this subsection with the observation that the (n, j)-partitions whose left subset contains the smallest label come first in the order that we have imposed on $\mathcal{S}_{n,j}$. Therefore, Algorithms 33 and 34 are also valid to generate the objects in a boxed product $\alpha \,{}^{\Box}\!\star \beta$, provided that the corresponding iterator is initialized so that the attribute count_part is $\binom{n-1}{j-1}$ instead of $\binom{n}{j}$, where j is the size of the left subobject and n is the total size of the object. For the generation in PF order we also have to change *Update_Perm* in Algorithm 33 to *Update_BPerm* (see Algorithm 36 bellow because now the smallest label (denoted by i) always belongs to the left subset of the boxed product.

Algorithm 35 Generation the next (n, j)-partition in *Next_Lab*

```
1.  if j > ⌊n/2⌋ then ...
2.  else i := j
3.      while i ≥ 1 and part[i] = n − j + i do i := i − 1 end
4.      i′ := part[i] + 1; i″ := part[i]
5.      part[i] ↔ part[inv_part[i′]]; inv_part[i″] ↔ inv_part[i′]
6.      ▷ Update also perm and inv_perm ◁
7.      if part[i] + 1 ≠ part[i + 1] − 1 then
8.          for k := i + 1 to j do
9.              k′ := part[k − 1] + 1; k″ := part[k]
10.             part[k] ↔ part[inv_part[k′]]; inv_part[k″] ↔ inv_part[k′]
11.             if (inv_part[k″] < n) and
12.                (part[inv_part[k″]] > part[inv_part[k″] + 1]) then
13.                z := inv_part[k″]
14.                for t := k + 1 to j do
15.                    part[t] := part[t − 1] + 1; inv_part[part[t]] := t
16.                end
17.                for t := z to n do part[t] := t; inv_part[t] := t end
18.                break ▷ Finishes loop on k (lines (8)-(20)) ◁
19.             fi
20.         end
21.     fi
22. fi
```

Algorithm 36 *Update_BPerm* for sets and cycles in PF order

```
Update_BPerm(perm, inv_perm, i, n, j)
    for k := i to i + j − 1 do
        if perm[k] ≠ i then
            perm[k] := perm[k] − (n − j); inv_perm[perm[k]] := k
        fi
    end
    for k := i + j to i + n − 1 do
        perm[k] := perm[k] + (j − 1); inv_perm[perm[k]] := k
    end
```

This is very convenient to exhaustively generate sets and cycles in an easy way, thanks to the isomorphisms of (2.11) and (2.12). For these classes (sets and cycles), we will have to use, in the case of PF order, *Next_BProd* instead of *Next_Prod*, where *Next_BProd* is like *Next_Prod* but using *Update_BPerm* instead of *Update_Perm* (lines (11) and (14) in Algorithm 33). But that's all.

Assuming the usual isomorphisms and the previous remark if the boxed product is considered, we can extend Algorithm 31 to compute the next object in $\mathsf{Seq}(\mathcal{A})$, $\mathcal{A}^k$, $\mathsf{Set}(\mathcal{A})$,

$\mathcal{A}^{\{k\}}$, etc., and obtain Algorithm 37.

Algorithm 37 Next labelled object of the node p (final version)

Next_Obj(p, itr)
    if Is_Last_Obj(p) **then** $p \to \mathsf{rank} := p \to \mathsf{rank} + 1$ **return fi**
    $p \to \mathsf{rank} := p \to \mathsf{rank} + 1$
    if $p \to \mathsf{class} = \mathcal{A} + \mathcal{B}$ **then** Next_Union(p)
    elif $p \to \mathsf{class} = \mathcal{A} \star \mathcal{B}$ **or** $p \to \mathsf{class} = \mathsf{Seq}(\mathcal{A})$ **or**
        $p \to \mathsf{class} = \mathcal{A}^k$ **then** Next_Prod(p, itr)
    elif $p \to \mathsf{class} = \mathsf{Set}(\mathcal{A})$ **or** $p \to \mathsf{class} = \mathsf{Cycle}(\mathcal{A})$ **or**
        $p \to \mathsf{class} = \mathcal{A}^{\{k\}}$ **or** $p \to \mathsf{class} = \mathcal{A}^{\langle k \rangle}$ **then** Next_BProd(p, itr)
    elif $p \to \mathsf{class} = \mathcal{A} \circ \mathcal{B}$ **then** Next_Subst(p, itr)
    fi

We also give in Algorithm 38 the final version of *Init_Obj*, which takes account the observations here, like the initialization of $\mathsf{count\_part}$ according to whether the object is a product or a boxed product.

Finally, we have to describe the code for the function *Next_Subst*. We note that substitutions do not involve relabellings, so this function is similar to the function *Next_Prod* (Algorithms 33 and 34) but without taking care of the labelling partitions (see Algorithm 39).

It is also possible to do the exhaustive generation for sequences, sets and cycles in card-lexicographic order. It is just necessary to use the specifications $\mathsf{Seq}(\mathcal{A}) = \bigcup_{k \geq 0} \mathcal{A}^k$, $\mathsf{Set}(\mathcal{A}) = \bigcup_{k \geq 0} \mathcal{A}^{\{k\}}$ and $\mathsf{Cycle}(\mathcal{A}) = \bigcup_{k \geq 1} \mathcal{A}^{\langle k \rangle}$ instead of $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \star \mathsf{Seq}(\mathcal{A})$, $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A}^{\square}\star \mathsf{Set}(\mathcal{A})$ and $\mathsf{Cycle}(\mathcal{A}) = \mathcal{A}^{\square}\star \mathsf{Seq}(\mathcal{A})$, respectively.

7.3 Variants and Heuristics

It is quite easy to see that using the big-endian heuristic, the boxed heuristic, the right boxed product or the boustrophedonic order does not improve the performance for the exhaustive generation. However, we will show that there is room for improvements on the performance of exhaustive generation.

The cost of *Next* can be reduced if we use *finger pointers*. Each node contains on not only the attributes (class, size, count, rank, etc.) that we have already discussed, but a pointer (called finger) to the last right node that was modified by the application of the function *Next_Obj*[3]; it is likely that when moving from the current object to the *Next_Obj* we will only need to update that node or some node in its vicinity (its parent, one of its children). In the case of PF order the use of the fingers makes little sense since we spend most of the time updating the root's partition.

For the SF order, we have incorporated to all our algorithms the possibility of using fingers to improve the performance. When computing the next object we access in one

[3] For unions, if the right node is NULL, then we will apply the same criterion from the left node.

Algorithm 38 *Init_Obj* (final version)

Init_Obj($\mathcal{C}, n, ip, \mathrm{itr}$)
  $\triangleright$ See lines (2)-(10) in Algorithm 30 $\triangleleft$
  if ... **then** $\triangleright$ See lines (11)-(19) in Algorithm 30 $\triangleleft$
  elif $\mathcal{C} = \mathcal{A}^0$ **or** $\mathcal{C} = \mathcal{A}^{\{0\}}$ **then** $\triangleright$ Fields $\mathsf{count\_part}$ and $\mathsf{rank\_part}$ are meaningless $\triangleleft$
    $p \to \mathsf{left} := \mathsf{NULL}; p \to \mathsf{right} := \mathsf{NULL}$
  elif $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$ **then** $\triangleright$ Fields $\mathsf{count\_part}$ and $\mathsf{rank\_part}$ are meaningless $\triangleleft$
    $k := 1$
    while $\mathrm{Count}(\mathcal{A}, k) \cdot \mathrm{Count}(\mathcal{B}^{\{k\}}, n) = 0$ **do** $k := k + 1$ **end**
    $p \to \mathsf{left} := \mathrm{Init\_Obj}(\mathcal{A}, k, ip, \mathrm{itr}); p \to \mathsf{right} := \mathrm{Init\_Obj}(\mathcal{B}^{\{k\}}, n, ip, \mathrm{itr})$
  else if $\mathcal{C} = \mathcal{A} \star \mathcal{B}$ **then** $lC := \mathcal{A}; rC := \mathcal{B}; j := 0$
    elif $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{Set}(\mathcal{A})$ **then** $lC := \mathcal{A}; rC := \mathcal{C}; j := 1$
    elif $\mathcal{C} = \mathsf{Cycle}(\mathcal{A})$ **then** $lC := \mathcal{A}; rC := \mathsf{Seq}(\mathcal{A}); j := 1$
    elif $\mathcal{C} = \mathcal{A}^k$ **then** $lC := \mathcal{A}; rC := \mathcal{A}^{k-1}; j := 1$
    elif $\mathcal{C} = \mathcal{A}^{\{k\}}$ **then** $lC := \mathcal{A}; rC := \mathcal{A}^{\{k-1\}}; j := 1$
    elif $\mathcal{C} = \mathcal{A}^{\langle k \rangle}$ **then** $lC := \mathcal{A}; rC := \mathcal{A}^{k-1}; j := 1$
    fi
    while $\mathrm{Count}(lC, j) \cdot \mathrm{Count}(rC, n - j) = 0$ **do** $j := j + 1$ **end**
    $p \to \mathsf{left} := \mathrm{Init\_Obj}(lC, j, ip, \mathrm{itr}); p \to \mathsf{right} := \mathrm{Init\_Obj}(rC, n - j, ip + j, \mathrm{itr})$
    if $\mathcal{C} = \mathcal{A} \star \mathcal{B}$ **or** $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **or** $\mathcal{C} = \mathcal{A}^k$ **then** $p \to \mathsf{count\_part} := \binom{n}{j}$
    elif $\mathcal{C} = \mathsf{Set}(\mathcal{A})$ **or** $\mathcal{C} = \mathcal{A}^{\{k\}}$ **or** $\mathcal{C} = \mathsf{Cycle}(\mathcal{A})$ **or** $\mathcal{C} = \mathcal{A}^{\langle k \rangle}$ **then**
      $p \to \mathsf{count\_part} := \binom{n-1}{j-1}$
    fi
  fi
  return p

Algorithm 39 Next for substitutitons

Next_Subst(p, itr)
  if not Is_Last_Obj($p \to \mathsf{right}$) **then** Next_Obj($p \to \mathsf{right}$)
  elif not Is_Last_Obj($p \to \mathsf{left}$) **then**
    Next_Obj($p \to \mathsf{left}, \mathrm{itr}$)
    $p \to \mathsf{right} := \mathrm{Init\_Obj}(p \to \mathsf{right} \to \mathsf{class}, p \to \mathsf{size}, p \to \mathsf{ind\_perm}, \mathrm{itr})$
  else $n := p \to \mathsf{size}; k := p \to \mathsf{left} \to \mathsf{size} + 1$
    while $\mathrm{Count}(\mathcal{A}, k) \cdot \mathrm{Count}(\mathcal{B}^{\{k\}}, n) = 0$ **do** $k := k + 1$ **end**
    $p \to \mathsf{left} := \mathrm{Init\_Obj}(\mathcal{A}, k, p \to \mathsf{ind\_perm}, \mathrm{itr})$
    $p \to \mathsf{right} := \mathrm{Init\_Obj}(\mathcal{B}^{\{k\}}, n, p \to \mathsf{ind\_perm}, \mathrm{itr})$
  fi

single step the right node; most of the times, computing the next object will only need change the right subobject (right subtree structure) rooted at the node pointed to by the finger, so we can avoid branching to the right from the root node and go directly to the right node pointed to by the finger. If the subobject that we access through the finger is the last in its class then we have to backtrack; hence, the need for an attribute parent in each node (that is, each node requires a new pointer to its parent). Also, while performing a backtrack we have to "correct" the ascendants of the finger up the way for several of their attributes, since some of them (rank, ...) are outdated.

In Algorithms 40 and 41 we show how to initialize and to update the finger of a node p in SF order. To initialize the finger we start from p and we go down to the right node of $p\to\mathsf{finger}$ while it is not the last object of its class and size. On the other hand, to update the finger of a node p, we have to backtrack through the parent node of $p\to\mathsf{finger}$ while the current subobject is the last object of its class and size.

Algorithm 40 Initialize the finger of a node p in SF order and *Init_Obj* using this finger

```
Init_Finger(p)
    p.finger := p  ▷ Node pointed to itself ◁
    while not Is_Last_Obj(p→finger→right) do
        p→finger := p→finger→right
    end

Init_Obj(C, n, ip, itr)
    ▷ See lines (2)-(23) in Algorithm 38 ◁
    Init_Finger(p); return p
```

Using a finger, the initialization of nodes is given by Algorithm 40, while the generation of successive objects is given by Algorithm 41.

Algorithm 41 Update the finger of a node p in SF order and *Next_Obj* using this finger

```
Update_Finger(p)
        if Is_Last_Obj(p→finger) then
            while p→finger ≠ p and Is_Last_Obj(p→finger→parent) do
                rnk := p→finger→rank; p→finger := p→finger→parent
                p→finger→rank := p→finger→rank + rnk
            end
        fi

Next_Obj(p, itr)
    ▷ See lines (2)-(10) in Algorithm 37 ◁
    Update_Finger(p)
```

7.4 Analysis of the Cost

In this section we analyze the cost of the algorithms for exhaustive generation. We establish some rules for the computation of the cumulated cost of generating all objects in $\mathcal{A}_n$, the so-called Λ-rules, and we prove that the algorithms have constant amortized time (CAT) when they are applied to the so-called superexponentially growing labelled admissible classes.

Our cost model essentially accounts for the arithmetic operations made to compute the next object and disregards other costs involved in the generation. For instance, we will not include the preprocessing time needed to compute the tables of counts. That is, we do not take into account the cost of calling *Count* at several places of the algorithms. All the necessary *Count*'s can be computed once and conveniently stored into tables for later use. The cost contributed by the computation of counts to generate the objects in $\mathcal{A}_n$ is $\mathrm{O}(\min\{n^2, \sum_{0\leq j\leq n} a_j\})$ and can be safely disregarded.

We will charge an unitary cost to all non-recursive steps made within each of the algorithms of the previous section. This is clearly an oversimplification; for instance, generating the next object in $\mathcal{A}+\mathcal{B}$ requires less non-recursive overhead than generating the next object in $\mathcal{A}\star\mathcal{B}$. In a more realistic setting we should charge each operator the cost of the non-recursive steps to generate an object: to generate one object in $\mathcal{A}+\mathcal{B}$ this would be c_+, for products it would be $c_\star$, and so on, for suitable constants $c_+, c_\star, \ldots$. But the fact that the algorithms' performance is CAT is not affected by this simplification, hence we should proceed with this simple accounting scheme. We will assume that we use the finger pointers and that we generate (partitional and boxed) products in lexicographic SF order. The results for SF order provide an upper bound for the cost of the generation in PF order, because it is more inefficient. On the other hand, it also turns out that it is easier to analyze the cost of generation in SF order.

Finally, we will not include the time to parse specifications, as this cost does not depend on n.

Let $\Lambda\mathcal{A}_n$ denote the cost to generate all objects in $\mathcal{A}_n$. Then

$$\Lambda\mathcal{A}_n = f(\mathcal{A}, n) + \sum_{\alpha\in\mathcal{A}_n} \mathrm{cn}(\alpha, \mathcal{A}) \ ,$$

where $f(\mathcal{A}, n)$ denotes the cost to compute the first object in $\mathcal{A}_n$, and $\mathrm{cn}(\alpha, \mathcal{A})$ denotes the cost to apply *Next* to object α. We assume that $\mathrm{cn}(\alpha^\perp, \mathcal{A}) = 0$ if $\alpha^\perp$ is the last object in $\mathcal{A}_n$[4].

We denote $\Lambda\mathcal{A}(z)$ the exponential generating function of the cumulated costs $\Lambda\mathcal{A}_n$:

$$\Lambda\mathcal{A}(z) = \sum_{n\geq 0} \Lambda\mathcal{A}_n \frac{z^n}{n!} \ .$$

Hence

$$\mu_{\mathcal{A},n} = \frac{[z^n]\,\Lambda\mathcal{A}(z)}{[z^n]\,A(z)} = \frac{\Lambda\mathcal{A}_n}{a_n}$$

is the amortized cost per generated object.

In general, from the generation algorithms, it is easy to infer that the cost to generate one object from the previous one is at most proportional to the size of the object. For instance, $\mathrm{cn}(\alpha, \mathcal{A}+\mathcal{B})$ requires at most to generate the first object of $\mathcal{B}$ with cost proportional to $|\alpha|$. For products, $\mathrm{cn}(\langle\alpha, \beta, \ell\rangle, \mathcal{A}\star\mathcal{B})$ requieres at most to change the sizes of the components α and β, say j and $n-j$, respectively, and to generate the first objects of $\mathcal{A}_j$ and $\mathcal{B}_{n-j}$. Similar

[4]From now on, we will denote the last object of a given class and size using the superscript '$\perp$'.

arguments apply for the other operators. Hence, we have that $\Lambda\mathcal{A}_n = \mathrm{O}(n \cdot a_n)$. This would give $\mu_{\mathcal{A},n} = \Theta(n)$; we will show that for most classes $\mathcal{A}$ we actually have $\mu_{\mathcal{A},n} = \Theta(1)$.

We start establishing a set of rules for the Λ operator, in the spirit of the Υ-rules for unranking.

Theorem 22. *Given two admissible labelled classes $\mathcal{A}$ and $\mathcal{B}$, then*

1. $\Lambda\epsilon = 1$,
2. $\Lambda Z = z$,
3. $\Lambda(\mathcal{A}+\mathcal{B}) = \Lambda\mathcal{A} + \Lambda\mathcal{B} + [\![\mathcal{A}]\!] + [\![\mathcal{B}]\!] - [\![\mathcal{A}+\mathcal{B}]\!]$,
4. $\Lambda(\mathcal{A}\star\mathcal{B}) = \Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + A\cdot\Lambda\mathcal{B} + A\cdot[\![\mathcal{B}]\!] - [\![\mathcal{A}\star\mathcal{B}]\!]$,
5. $\vartheta\Lambda(\mathcal{A}\,{}^{\Box}\!\!\star\mathcal{B}) = \vartheta\Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + \vartheta A\cdot\Lambda\mathcal{B} + \vartheta A\cdot[\![\mathcal{B}]\!] - \vartheta[\![\mathcal{A}\,{}^{\Box}\!\!\star\mathcal{B}]\!]$,

where $[\![\mathcal{C}]\!] = \sum_{n\geq 0}[\![c_n \neq 0]\!]\frac{z^n}{n!}$.

Proof. The first two statments are immediate from the definition of Λ and the corresponding (trivial) iteration algorithms.

The rule for unions is straightforward, but we must take care to charge the cost corresponding to compute the next of the last element in $\mathcal{A}_n$ if it exists. This cost is given by $[\![\mathcal{A}]\!]_n + [\![\mathcal{B}]\!]_n - [\![\mathcal{A}+\mathcal{B}]\!]_n$, where n is the size. In general,

$$\mathrm{cn}(\alpha,\mathcal{A}+\mathcal{B}) = \begin{cases} \mathrm{cn}(\alpha,\mathcal{A}) & \text{if } \alpha\in\mathcal{A} \text{ and } \alpha\neq\alpha^{\perp},\\ [\![a_n\neq 0 \wedge b_n \neq 0]\!] + f(\mathcal{B},n) & \text{if } \alpha=\alpha^{\perp}\in\mathcal{A},\\ \mathrm{cn}(\alpha,\mathcal{B}) & \text{if } \alpha\in\mathcal{B} \text{ and } \alpha\neq\alpha^{\perp},\\ 0 & \text{if } \alpha=\alpha^{\perp}\in\mathcal{A}+\mathcal{B}, \end{cases}$$

where $n = |\alpha|$ and $[\![a_n \neq 0 \wedge b_n \neq 0]\!]$ counts the passing from $\mathcal{A}$ to $\mathcal{B}$.

For products, if we fix the size n then

$$\Lambda(\mathcal{A}\star\mathcal{B})_n = \sum_{j=0}^{n}\Lambda(\mathcal{A}_j\star\mathcal{B}_{n-j}) + \sum_{j=0}^{n}[\![a_j\neq 0]\!]\cdot[\![b_{n-j}\neq 0]\!] - [\![\sum_{j=0}^{n}\mathcal{A}_j\star\mathcal{B}_{n-j}\neq\emptyset]\!]\ , \tag{7.1}$$

where $\Lambda(\mathcal{A}_j\star\mathcal{B}_{n-j})$ is the cost to generate all objects in $\mathcal{A}_j\star\mathcal{B}_{n-j}$, and the remaining cost is the contribution to change the sizes of the components ($\sum_{j=0}^{n}[\![a_j\neq 0]\!]\cdot[\![b_{n-j}\neq 0]\!]$) minus the first one ($[\![\sum_{j=0}^{n}\mathcal{A}_j\star\mathcal{B}_{n-j}\neq\emptyset]\!]$).

For $\mathcal{A}_j\star\mathcal{B}_{n-j}$, we generate the first object with cost $f(\mathcal{A},j)+f(\mathcal{B},n-j)$ if it exists, and for succesive objects we get

$$\mathrm{cn}(\langle\alpha,\beta,\ell\rangle,\mathcal{A}_j\star\mathcal{B}_{n-j}) =$$

$$\begin{cases} \mathrm{cn}(\beta,\mathcal{B}_{n-j}) & \text{if } \beta\neq\beta^{\perp},\\ 1+\mathrm{cn}(\alpha,\mathcal{A}_j)+f(\mathcal{B},n-j) & \text{if } \alpha\neq\alpha^{\perp} \text{ and } \beta=\beta^{\perp},\\ 1+f(\mathcal{A},j)+f(\mathcal{B},n-j) & \text{if } \alpha=\alpha^{\perp},\ \beta=\beta^{\perp} \text{ and } 0\leq\ell<\binom{n}{j}-1,\\ 0 & \text{if } \alpha=\alpha^{\perp},\ \beta=\beta^{\perp} \text{ and } \ell=\binom{n}{j}-1. \end{cases}$$

For each partition and $\mathcal{A}_j$-object, we generate all $\mathcal{B}_{n-j}$-objects (line 1); if $\beta = \beta^\perp$ then we update the finger and generate the successor of α and the first object in $\mathcal{B}_{n-j}$ (line 2); if $\alpha = \alpha^\perp$ and $\beta = \beta^\perp$ then we generate the following partition and the first objects in $\mathcal{A}_j$ and $\mathcal{B}_{n-j}$ (line 3), finally the last object has no cost (line 4). Plugging the cost of $\Lambda(\mathcal{A}_j \star \mathcal{B}_{n-j})$ into (7.1) we get that

$$\begin{aligned}\Lambda(\mathcal{A}\star\mathcal{B})_n = &\sum_{j=0}^{n}\binom{n}{j}\cdot\Lambda\mathcal{A}_j\cdot[\![b_{n-j}\neq 0]\!] + \sum_{j=0}^{n}\binom{n}{j}\cdot a_j\cdot\Lambda\mathcal{B}_{n-j} + \\ &\sum_{j=0}^{n}\binom{n}{j}\cdot a_j\cdot[\![b_{n-j}\neq 0]\!] \;-[\![\sum_{j=0}^{n}\mathcal{A}_j\star\mathcal{B}_{n-j}\neq\emptyset]\!]\ .\end{aligned}$$

Hence,

$$\Lambda(\mathcal{A}\star\mathcal{B}) = \Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + A\cdot\Lambda\mathcal{B} + A\cdot[\![B]\!] - [\![\mathcal{A}\star\mathcal{B}]\!]\ . \tag{7.2}$$

For boxed products, we follow the same criteria as for (partitional) products (rule 4) and take into account that $n\cdot\binom{n-1}{j-1} = j\cdot\binom{n}{j}$ to obtain rule (5). □

From the Λ-rules of Theorem 22 and the standard isomorphisms for sequences, sets, cycles and substitutions ((2.6), (2.11), (2.12) and (1.2)), we get the following corollary.

Corollary 6. *Let $\mathcal{A}$ be an admissible labelled class, then*

1. $\Lambda\mathit{Seq}(\mathcal{A}) = (\Lambda\mathcal{A} + A)\cdot\frac{[\![\mathit{Seq}(\mathcal{A})]\!]}{1-A} + \frac{\Lambda\epsilon+[\![\epsilon]\!]-[\![\mathit{Seq}(\mathcal{A})]\!]}{1-A}$.
2. $\Lambda\mathit{Set}(\mathcal{A}) = \exp(A)\cdot\int_0^z \frac{dt}{t\exp(A(t))}\Big(\vartheta(\Lambda\mathcal{A}(t) + [\![\mathcal{A}]\!](t))\cdot[\![\mathit{Set}(\mathcal{A})]\!](t) + \vartheta\Lambda\epsilon + [\![\epsilon]\!] - \vartheta[\![\mathit{Set}(\mathcal{A})]\!](t)\Big)$.
3. $\Lambda\mathit{Cycle}(\mathcal{A}) = \int_0^z \frac{dt}{t}\Big(\vartheta(\Lambda\mathcal{A}(t) + A(t))\cdot([\![\mathit{Cycle}(\mathcal{A})]\!](t) + [\![\epsilon]\!]) + \vartheta A(t)\cdot\Lambda\mathit{Seq}(\mathcal{A})(t)\Big) - [\![\mathit{Cycle}(\mathcal{A})]\!]$.
4. $\Lambda(\mathcal{A}\circ\mathcal{B}) = \Lambda\mathcal{A}([\![\mathcal{B}]\!]) + A(\Lambda\mathcal{B}) + A([\![\mathcal{B}]\!]) - [\![\mathcal{A}\circ\mathcal{B}]\!]$.

Notice that $\Lambda\epsilon = 1$, $[\![\epsilon]\!] = 1$ and $\vartheta\Lambda\epsilon = 0$.

For sequences, sets and cycles with restricted cardinality, similar Λ-rules can be obtained considering the specifications of Table 2.4.

Before proving that the exhaustive generation is CAT for a large family of labelled admissible classes, including the most interesting ones, we introduce some definitions and observations.

Definition 12. *An admissible class $\mathcal{A}$ is $c_{\mathcal{A}}$-*generable *if and only if $\Lambda\mathcal{A}_n \leq c_{\mathcal{A}}\cdot a_n$, for any $n \geq 0$ and some constant $c_{\mathcal{A}} \geq 1$. In particular, if $\mathcal{A}$ is $c_{\mathcal{A}}$-generable it means that the exhaustive generation of $\mathcal{A}$ is CAT.*

Definition 13. *A labelled admissible class $\mathcal{A}$ is* superexponentially growing *(superexponential, for short) if and only if for any constant c, $c^n = \mathrm{O}(a_n)$.*

Let $\mathcal{A}$ and $\mathcal{B}$ be $c_{\mathcal{A}}$- and $c_{\mathcal{B}}$-generable, respectively. Then from Definition 12 of c-generability and Theorem 22, it follows that $\mathcal{A}+\mathcal{B}$ is a $(\max\{c_{\mathcal{A}}, c_{\mathcal{B}}\}+1)$-generable, and $\mathcal{A}\star\mathcal{B}$ and $\mathcal{A}\,{}^{\square}\!\!\star\,\mathcal{B}$ are $(c_{\mathcal{A}}+c_{\mathcal{B}}+1)$-generable.

For products $\mathcal{A}\star\mathcal{B}$, we have the Λ-rule

$$\Lambda(\mathcal{A}\star\mathcal{B}) = \Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + A\cdot\Lambda\mathcal{B} + A\cdot[\![\mathcal{B}]\!] - [\![\mathcal{A}\star\mathcal{B}]\!]\ .$$

If $\mathcal{B}$ is superexponential, and taking into account that $\Lambda\mathcal{A}_n = \mathrm{O}(n\cdot a_n)$, for any $n>0$, then $\Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] \ll A\cdot B$ and $A\cdot[\![\mathcal{B}]\!] \ll A\cdot B$. Hence,

$$\Lambda(\mathcal{A}\star\mathcal{B}) \asymp A\cdot\Lambda\mathcal{B}\ .$$

If we assume now that $\mathcal{B}$ is $c_{\mathcal{B}}$-generable then $\mathcal{A}\star\mathcal{B}$ is also $c_{\mathcal{B}}$-generable.

Similar arguments prove that $\mathcal{A}\,{}^{\square}\!\!\star\mathcal{B}$ and $\mathcal{A}\circ\mathcal{B}$ are $c_{\mathcal{B}}$-generable, if $\mathcal{B}$ is superexponential and $c_{\mathcal{B}}$-generable. For instance,

$$\vartheta\Lambda(\mathcal{A}\,{}^{\square}\!\!\star\,\mathcal{B}) = \vartheta\Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] + \vartheta A\cdot\Lambda\mathcal{B} + \vartheta A\cdot[\![\mathcal{B}]\!] - \vartheta[\![\mathcal{A}\,{}^{\square}\!\!\star\,\mathcal{B}]\!]$$

implies that

$$\vartheta\Lambda(\mathcal{A}\,{}^{\square}\!\!\star\,\mathcal{B}) \asymp \vartheta A\cdot\Lambda\mathcal{B}\ ,$$

since $\vartheta\Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!] \ll \vartheta A\cdot B$ and $\vartheta A\cdot[\![\mathcal{B}]\!] \ll \vartheta A\cdot B$.

Notice that if $\mathcal{A}$ is superexponential and $c_{\mathcal{A}}$-generable but $\mathcal{B}$ were not, then it does not follow that $\mathcal{A}\star\mathcal{B}$ is $c_{\mathcal{A}}$-generable even if $\mathcal{B}$ were $c_{\mathcal{B}}$-generable and $c_{\mathcal{B}} \le c_{\mathcal{A}}$. That is so because $(\Lambda\mathcal{A}\cdot[\![\mathcal{B}]\!])_n$, $(\mathcal{A}\cdot\Lambda\mathcal{B})_n$ and $(\mathcal{A}\cdot[\![\mathcal{B}]\!])_n$ might be of a similar order of magnitude. But if we were generating products iterating first over first components and then on second components, then $\Lambda(\mathcal{A}\star\mathcal{B}) \asymp B\cdot\Lambda\mathcal{A}$ and we would have that $\mathcal{A}\star\mathcal{B}$ is $c_{\mathcal{A}}$-generable because $\mathcal{A}$ does.

This suggests the introduction of the *little-endian heuristic* (the opposite of the big-endian heuristic, see Section 3.4 and [26]), namely, reorder the specification of products to generate them so that the second components belong to the larger class, i.e., if $\mathcal{A}\gg\mathcal{B}$ then generate $\mathcal{B}\star\mathcal{A}$, otherwise generate $\mathcal{A}\star\mathcal{B}$. The application of this heuristic might guarantee generation in CAT (if any of $\mathcal{A}$ and $\mathcal{B}$ is superexponential) and a smaller generation constant. The price to pay is that the order of generation is no longer lexicographic. In fact it might turn out to be very difficult to express and describe this order if the heuristic were systematically applied, e.g., if we had $\mathcal{C} = \mathcal{A}_1\star\mathcal{A}_2\star\cdots\star\mathcal{A}_m$.

Now, we can establish the following theorem.

Theorem 23. *Let $\mathcal{A}$ and $\mathcal{B}$ be labelled admissible classes, then*

1. *If $\mathcal{A}$ and $\mathcal{B}$ are $c_{\mathcal{A}}$- and $c_{\mathcal{B}}$-generable, respectively, then $\mathcal{A}+\mathcal{B}$ is $(\max\{c_{\mathcal{A}}, c_{\mathcal{B}}\}+1)$-generable.*

2. *If $\mathcal{A}$ and $\mathcal{B}$ are $c_{\mathcal{A}}$- and $c_{\mathcal{B}}$-generable, respectively, then $\mathcal{A}\star\mathcal{B}$ and $\mathcal{A}\,{}^{\square}\!\!\star\,\mathcal{B}$ are $(c_{\mathcal{A}}+c_{\mathcal{B}}+1)$-generable.*

3. *If $\mathcal{B}$ is c-generable and superexponential, then $\mathcal{A} \circ \mathcal{B}$ is c-generable.*

4. *If $\mathcal{A}$ is c-generable and superexponential, and $\mathcal{B} \ll \mathcal{A}$, then $\mathcal{A} + \mathcal{B}$ and $\mathcal{B} + \mathcal{A}$ are c-generable.*

   In particular, if $\mathcal{A}$ and $\mathcal{B}$ are $c_{\mathcal{A}}$- and $c_{\mathcal{B}}$-generable, respectively, and both classes are superexponential, then $\mathcal{A} + \mathcal{B}$ is $\max\{c_{\mathcal{A}}, c_{\mathcal{B}}\}$-generable.

5. *If either $\mathcal{A}$ or $\mathcal{B}$ is c-generable and superexponential, then $\mathcal{A} \star \mathcal{B}$ and $\mathcal{A} \,{}^{\square}\!\!\star\, \mathcal{B}$ are c-generable (eventually using the little-endian heuristic).*

   Let $\mathcal{A}$ and $\mathcal{B}$ be superexponential. In particular, if $\mathcal{A}$ is $c_{\mathcal{A}}$-generable and $\mathcal{B}$ is not $c_{\mathcal{B}}$-generable for any constant $c_{\mathcal{B}}$, then $\mathcal{A} \star \mathcal{B}$ and $\mathcal{A} \,{}^{\square}\!\!\star\, \mathcal{B}$ are $c_{\mathcal{A}}$-generable; if $\mathcal{B}$ is $c_{\mathcal{B}}$-generable and $\mathcal{A}$ is not $c_{\mathcal{A}}$-generable for any constant $c_{\mathcal{A}}$, then $\mathcal{A} \star \mathcal{B}$ and $\mathcal{A} \,{}^{\square}\!\!\star\, \mathcal{B}$ are $c_{\mathcal{B}}$-generable; and if $\mathcal{A}$ and $\mathcal{B}$ are $c_{\mathcal{A}}$- and $c_{\mathcal{B}}$-generable, respectively, then $\mathcal{A} \star \mathcal{B}$ and $\mathcal{A} \,{}^{\square}\!\!\star\, \mathcal{B}$ are $\min\{c_{\mathcal{A}}, c_{\mathcal{B}}\}$-generable.

Theorem 24. *Let $\mathcal{C}$ be a superexponential labelled admissible class, then $\mathcal{C}$ is c-generable, for some constant $c \geq 1$.*

Proof. We prove the statement of the theorem by structural induction on the specification of the class.

If $\mathcal{C} = \mathcal{A} + \mathcal{B}$, then either $\mathcal{A}$ or $\mathcal{B}$ is also superexponential. If both $\mathcal{A}$ and $\mathcal{B}$ are superexponential, then $\mathcal{C}$ is c-generable by induction and the first statement of Theorem 23, for some constant $c \geq 1$. If only one of them, either $\mathcal{A}$ or $\mathcal{B}$, is superexponential, then $\mathcal{C}$ is c-generable by induction and the fourth statement of Theorem 23, for some constant $c \geq 1$.

If $\mathcal{C} = \mathcal{A} \star \mathcal{B}$ or $\mathcal{C} = \mathcal{A} \,{}^{\square}\!\!\star\, \mathcal{B}$, then either $\mathcal{A}$ or $\mathcal{B}$ is superexponential. Hence, $\mathcal{C}$ is c-generable by induction and the fifth statement of Theorem 23, for some constant $c \geq 1$.

For sequences $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$, we consider the isomorphism $\mathcal{C} = \epsilon + \mathcal{A} \star \mathcal{C}$. Then, applying the Λ-rules of Theorem 22, we get the following equality:

$$\Lambda\mathcal{C} = \Lambda\epsilon + \Lambda\mathcal{A} \cdot [\![\mathcal{C}]\!] + A \cdot \Lambda\mathcal{C} + A \cdot [\![\mathcal{C}]\!] - [\![\mathcal{A} \star \mathcal{C}]\!] \ .$$

If $\mathcal{C}$ is superexponential then $A \cdot [\![\mathcal{C}]\!] \ll A \cdot C$. Moreover, $\Lambda\mathcal{A} \cdot [\![\mathcal{C}]\!] \ll A \cdot C$ because $\Lambda\mathcal{A}_n = \mathrm{O}(n \cdot a_n)$ and $\mathcal{C}$ is superexponential. Hence,

$$\Lambda\mathcal{C} \asymp A \cdot \Lambda\mathcal{C} \asymp \frac{1}{1 - A} \ .$$

That is, $\mathcal{C}$ is $c_{\mathsf{Seq}(\mathcal{A})}$-generable, for some constant $c_{\mathsf{Seq}(\mathcal{A})} \geq 1$ that is computable given A, $\Lambda\mathcal{A}$ and the Λ-rules of Theorem 22. Notice that the isomorphism $\mathcal{C} = \epsilon + \mathcal{A} \star \mathcal{C}$ is the one that we should use by the little-endian heuristic (instead of $\mathcal{C} = \epsilon + \mathcal{C} \star \mathcal{A}$).

Similar arguments prove that $\mathsf{Set}(\mathcal{A}) = \epsilon + \mathcal{A} \,{}^{\square}\!\!\star\, \mathsf{Set}(\mathcal{A})$ and $\mathsf{Cycle}(\mathcal{A}) = \mathcal{A} \,{}^{\square}\!\!\star\, \mathsf{Seq}(\mathcal{A})$ are $c_{\mathsf{Set}(\mathcal{A})}$- and $c_{\mathsf{Seq}(\mathcal{A})}$-generable, respectively, for some constants $c_{\mathsf{Set}(\mathcal{A})}, c_{\mathsf{Seq}(\mathcal{A})} \geq 1$. For sets $\mathcal{C} = \mathsf{Set}(\mathcal{A})$, we have that

$$\vartheta\Lambda\mathcal{C} = \vartheta\Lambda\epsilon + \vartheta\Lambda\mathcal{A} \cdot [\![\mathcal{C}]\!] + \vartheta A \cdot \Lambda\mathcal{C} + \vartheta A \cdot [\![\mathcal{C}]\!] - \vartheta[\![\mathcal{A} \star \mathcal{C}]\!] \asymp \vartheta A \cdot \Lambda\mathcal{C} \ .$$

For cycles $\mathcal{C} = \mathsf{Cycle}(\mathcal{A})$, if $\mathcal{S} = \mathsf{Seq}(\mathcal{A})$ then

$$\vartheta\Lambda\mathcal{C} = \vartheta\Lambda\mathcal{A} \cdot [\![\mathcal{S}]\!] + \vartheta A \cdot \Lambda\mathcal{S} + \vartheta A \cdot [\![\mathcal{S}]\!] - \vartheta[\![\mathcal{A} \star \mathcal{S}]\!] \asymp \vartheta A \cdot \Lambda\mathcal{S} \ . \qquad \square$$

Since sequences, sets and cycles with restricted cardinalities are build upon unions and (partitional and boxed) products (see Table 2.4), the corresponding Λ-rules of Corollary 6 can be obtained directly, and Theorem 24 also holds.

7.4.1 Examples

Non-Superexponential Classes

We have seen that any superexponential labelled admissible class is c-generable. However, there exist non-superexponential classes (e.g., classes with a constant or exponential number of objects for each size) that are not c-generable.

1. For instance, $\mathcal{A}^{(0)} = \mathsf{Set}(Z)$ is not superexponential (there is only one set of each size) and
$$\mu_{\mathcal{A}^{(0)},n} = \frac{\Lambda\mathcal{A}_n^{(0)}}{a_n^{(0)}} = \frac{[z^n]\,(z+1)\,e^z}{[z^n]\,e^z} = n+1 \ .$$
Let $\mathcal{A}^{(k)} = Z \star \mathcal{A}^{(k-1)}$, for any $k>0$. Then $A^{(k)} = z^k\,e^z$ and $\Lambda\mathcal{A}^{(k)} \asymp z^{k+1}\,e^z + 3\,z^k\,e^z$. Hence we have that $\mu_{\mathcal{A}^{(k)},n} = (n-k)+3+\mathrm{o}(1)$.

   In a similar way, let $\mathcal{B}^{(k)} = \mathcal{B}^{(k-1)} \star Z$, for any $k>0$, where $\mathcal{B}^{(0)} = \mathsf{Set}(Z)$. Then $B^{(k)} = z^k\,e^z$ and $\Lambda\mathcal{B}^{(k)} \asymp z^{k+1}\,e^z + (2\,k+1)\,z^k\,e^z$. We have now that $\mu_{\mathcal{B}^{(k)},n} = (n-k)+(2\,k+1)+\mathrm{o}(1) = n+k+1+\mathrm{o}(1)$.

2. Consider now $\mathcal{C}^{(1)} = \mathsf{Set}(Z)$ and $\mathcal{C}^{(k)} = \mathcal{C}^{(1)} \star \mathcal{C}^{(k-1)}$ for any $k>1$; that is, $C^{(k)}$ is the class of k-tuples of sets; this class is not superexponential for it only contains $\Theta(n^k)$ objects of size n. Then $C^{(k)} = e^{k\,z}$ and $\Lambda\mathcal{C}^{(k)} \asymp 2\,z\,e^{k\,z} + 3\,e^{k\,z}$. Thus, we have that $\mu_{\mathcal{C}^{(k)},n} = 2\,n/k+3+\mathrm{o}(1)$.

   For the specification $\mathcal{D}^{(k)} = \mathcal{D}^{(k-1)} \star \mathcal{D}^{(1)}$, for any $k>1$, where $\mathcal{D}^{(1)} = \mathsf{Set}(Z)$, we have that $D^{(k)} = e^{k\,z}$ and $\Lambda\mathcal{D}^{(k)} \asymp k\,z\,e^{k\,z} + (2\,k-1)\,e^{k\,z}$. Hence $\mu_{\mathcal{C}^{(k)},n} = n+2\,k-1+\mathrm{o}(1)$.

However, if we consider that the cost to generate the first object of each class and size is neglected (i.e. $\Lambda\epsilon = \Lambda Z = 0$), then $\Lambda\mathcal{A}_n = \Theta(a_n)$ for any of the examples above. In particular, $\mu_{\mathcal{A}^{(k)},n} = 2+\mathrm{o}(1)$, $\mu_{\mathcal{B}^{(k)},n} = k+1+\mathrm{o}(1)$, $\mu_{\mathcal{C}^{(k)},n} = 3+\mathrm{o}(1)$ and $\mu_{\mathcal{D}^{(k)},n} = 2\,k-1+\mathrm{o}(1)$, for any $k>0$. The cost of generating the classes thus comes by frequent "initialization" of first objects.

Superexponential Classes

Let us now consider some examples where we have generation in CAT.

1. For binary trees $\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$, the cumulating generating function is
$$\Lambda\mathcal{B} = \frac{\Lambda Z - B \cdot [\![\mathcal{B}]\!] - [\![\mathcal{B} \star \mathcal{B}]\!]}{1 - [\![\mathcal{B}]\!] - B} \ .$$
Thus,
$$\mu_{\mathcal{B},n} = \frac{2\,(-2\,e^{1/2}+4\,e^{1/4}-1)}{(2\,e^{1/4}-3)^2} + \mathrm{o}(1) = 8.989803382\ldots + \mathrm{o}(1) \ .$$

Moreover, we also have that $\mu_{\mathcal{B}\star\mathcal{B},n} = \mu_{\mathcal{B},n}$ (as we expected).

For sequences of binary trees $\mathcal{S} = \mathsf{Seq}(\mathcal{B})$, we have that

$$\mu_{\mathcal{S},n} = \frac{4\,(5\,e^{1/2} - 19\,e^{1/4} + 18)}{(2\,e^{1/4} - 3)^2} = 39.59958470\ldots + \mathrm{o}(1)\ .$$

2. Following the Λ-rules for Motzkin trees, we get different constants depending on the specification:
   - If $\mathcal{M} = Z + Z \star \mathcal{M} + Z \star (\mathcal{M} \star \mathcal{M})$, then
   $$\mu_{\mathcal{M},n} = \frac{12\,e^{1/3} - 3\,e^{2/3} - 4}{3\,(e^{1/3} - 2)^2} + \mathrm{o}(1) = 6.300249066\ldots + \mathrm{o}(1)\ .$$
   - If $\mathcal{M} = Z + Z \star \mathcal{M} + (Z \star \mathcal{M}) \star \mathcal{M}$, then
   $$\mu_{\mathcal{M},n} = \frac{2\,(3\,e^{1/3} - e^{2/3} - 1)}{(e^{1/3} - 2)^2} + \mathrm{o}(1) = 6.784321668\ldots + \mathrm{o}(1)\ .$$
   - If $\mathcal{M} = Z \star (\epsilon + \mathcal{M} + \mathcal{M} \star \mathcal{M})$, then
   $$\mu_{\mathcal{M},n} = \frac{3\,e^{1/3} - e^{2/3} + 1}{(e^{1/3} - 2)^2} + \mathrm{o}(1) = 8.867347631\ldots + \mathrm{o}(1)\ .$$
3. For permutations, we consider either $\mathcal{P}_1 = \mathsf{Seq}(Z)$ or $\mathcal{P}_2 = \mathsf{Set}(\mathsf{Cycle}(Z))$. Our algorithms have CAT performance for both specifications:
   - For $\mathcal{P}_1 = \mathsf{Seq}(Z)$, we have that
   $$\mu_{\mathcal{P}_1,n} = e + 2 = 4.718281828\ldots$$
   - For $\mathcal{P}_2 = \mathsf{Set}(\mathsf{Cycle}(Z))$, we have that
   $$\mu_{\mathcal{P}_2,n} = \frac{12\,e - e^2 - 9}{4} = 4.057581460\ldots$$

Chapter 8

Generation of Unlabelled Classes

This chapter covers the generation algorithms for unlabelled admissible classes and their performance. In a similar way as for the labelled case, we give efficient general algorithms and analyze their performance for the so-called *unlabelled superpolynomial growing admissible classes* (classes $\mathcal{A}$ such that for any constant c, $n^c = \mathrm{o}(a_n)$).

Before going into the subject matter of this chapter, a few remarks are in order. As in the case of labelled exhaustive generation, for products we will only consider the lexicographic order because the boustrophedonic order is less intuitive and it does not yield a better performance as far as exhaustive generation is concerned. For powersets and sets we proposed in Chapter 2 four ordering criteria whose combination yields up to 16 and 32 variants, respectively, times the additional options to use lexicographic, anti-lexicographic, boustrophedonic or anti-boustrophedonic order (see Section 2.3). However, we will only consider the generation of objects in lexicographic LScR order (see Equations (4.1) and (4.3)). The other orders can also be dealt with similarly. Finally, cycles have a special consideration and we will study them in Section 8.4.

This chapter is organized as Chapter 7. In Section 8.1 we describe the algorithms for the generation of unlabelled classes. Afterwards, we analyze their performance. In Section 8.3 we discuss variants and heuristics. And finally, we study the generation of unlabelled cycles in a separate section because we use there rather different techniques and tools in their design of the generation algorithm and its analysis.

The main contributions of this chapter have been presented in [58, 60]. Some additional results appear also in [57, 64].

8.1 Implementation and Algorithms

Our algorithms act upon iterators, those structures that we already presented and described in Chapter 7. Here we just present a brief reminder.

An iterator contains a combinatorial object represented in a tree structure. The leaves of such structure correspond to the ϵ objects and atoms. Besides the name of the operator, each internal node contains a wealth of information concerning the (sub)object represented by the subtree rooted at that node: the size n (size), the rank of the object within its class (rank), the class $\mathcal{A}$ the object belongs to (class), the number of objects a_n (count), etc. They might also contain information giving direct access to the linear or conventional representation of the object so that the updates in the tree representation can be readily and efficiently be reflected in the linear representation of the object. Since we are now dealing with unlabelled objects we do not need some of the fields that we used before: part, $\mathsf{ind\_perm}$, etc.

To initialize the iterator and to compute the successive objects until to get the last one, we shall offer the same functions that we introduced in Section 7.1:

1. *Init_Iter*([class, spec_equations, unlabelled], n),
2. *Is_Last*(*itr*),
3. *Get_Obj*(*itr*),
4. *Next*(*itr*).

These functions are typically used as in Algorithm 28 in Chapter 7.

We consider here internal "private" functions that receive a pointer to the root node of the tree structure as we did for the generation of labelled classes in Chapter 7: *Init_Obj*($\mathcal{A}, n$) computes the first tree structure which represents the first object of size n in the class $\mathcal{A}$, *Next_Obj*(p) performs the actual work of *Next*(*itr*), and *Is_Last_Obj*(p) returns true if and only if $p \rightarrow \mathsf{rank} \geq p \rightarrow \mathsf{count} - 1$. The function *Init_Obj* for unlabelled unions, products and sequences is similar to the function *Init_Obj* for labelled unions, products and sequences, respectively, but without the labelling attributes. The function *Init_Obj* for the other combinatorial operators will be considered later, in Subsection 8.1.2. On the other hand, the function *Next_Obj* is quite similar to the function *Next_Obj* described in the previous chapter: See Algorithm 42 and compare it to Algorithm 31.

Algorithm 42 Next unlabelled object of the node p

Next_Obj(p)
  if Is_Last_Obj(p) **then** $p \rightarrow \mathsf{rank} := p \rightarrow \mathsf{rank} + 1$; **return fi**
  $p \rightarrow \mathsf{rank} := p \rightarrow \mathsf{rank} + 1$
  if $p \rightarrow \mathsf{class} = \mathcal{A} + \mathcal{B}$ **then** Next_Union(p)
  elif $p \rightarrow \mathsf{class} = \mathcal{A} \times \mathcal{B}$ **then** Next_Prod(p)
  elif $p \rightarrow \mathsf{class} = \mathsf{PSet}(\mathcal{A})$ **then** Next_PSet(p)
  elif $p \rightarrow \mathsf{class} = \mathsf{Set}(\mathcal{A})$ **then** Next_Set(p)
  elif $p \rightarrow \mathsf{class} = \mathsf{Cycle}(\mathcal{A})$ **then** Next_Cycle(p)
  fi

Moreover, recall that we assume that our "programing language" has some garbage collection system that will automatically take care of the unused nodes.

8.1.1 Basic Algorithms: $+$ and $\times$

The ϵ class and atomic classes have no next object. So, the implementation is trivial. For unions we have the same algorithm as for the labelled case (Algorithm 32). For products we follow Algorithm 34, but we have not to bother about partitions and labels: See Algorithm 43. Notice that lines (2)-(5) of Algorithm 43 and lines (3)-(7) of Algorithm 34 are identical, and lines (8)-(9) of Algorithm 43 are almost identical to lines (16)-(18) of Algorithm 34.

Algorithm 43 Next object for $\mathcal{A} \times \mathcal{B}$

1. Next_Prod(p)
2. **if not** Is_Last_Obj($p\to\mathsf{right}$) **then** Next_Obj($p\to\mathsf{right}$)
3. **elif not** Is_Last_Obj($p\to\mathsf{left}$) **then**
4. Next_Obj($p\to\mathsf{left}$)
5. $p\to\mathsf{right} :=$ Init_Obj($p\to\mathsf{right}\to\mathsf{class}, p\to\mathsf{right}\to\mathsf{size}$)
6. **else** $lC := p\to\mathsf{left}\to\mathsf{class}; rC := p\to\mathsf{right}\to\mathsf{class}$
7. $n := p\to\mathsf{size}; j := p\to\mathsf{left}\to\mathsf{size} + 1$
8. **while** $\mathrm{Count}(lC, j) \cdot \mathrm{Count}(rC, n-j) = 0$ **do** $j := j + 1$ **end**
9. $p\to\mathsf{left} :=$ Init_Obj(lC, j); $p\to\mathsf{right} :=$ Init_Obj($rC, n-j$)
10. **fi**

8.1.2 Algorithms for Sequences, PowerSets and Sets

Sequences are readily dealt with using the isomorphism $\mathsf{Seq}(\mathcal{A}) = \epsilon + \mathcal{A} \times \mathsf{Seq}(\mathcal{A})$. In particular, it suffices to apply *Next_Prod* to Seq-nodes as if they were Prod-nodes ($\times$-nodes).

For powersets and sets we need an specific representation. Following the isomorphisms (4.1) and (4.3) for LScR order, a non-empty object γ is seen as a pair (B, γ') with B a block of k components of size j in ascending order and γ' a powerset or set with components of size $> j$. In the representation of an object γ we have thus a root node labelled PSet or Set which contains the usual fields of information; but it also contains a field min_size that stores the minimal possible size among its components. We can therefore use (recursively) the same representation for γ'. The representation of blocks B is different since it must represent a collection of k objects of size j. The root node of such a block B is labelled PSetBlock (when no repetitions are allowed) or SetBlock (when repetitions are allowed), and it contains the number of objects or cardinality of the block. The left subtree of a PSetBlock of k objects represents the smallest object in the PSetBlock according to $\prec_{\mathcal{A}_j}$, and the right subtree represents a PSetBlock of $k-1$ objects. On the other hand, the left subtree of a SetBlock of k objects contains a Δ object, that is, the representation of some object α in $\mathcal{A}_j$ (in particular, the smallest object in the SetBlock according to $\prec_{\mathcal{A}_j}$) together with the number of times, say d, that it appears on the SetBlock; and the right subtree represents a SetBlock of $k-d$ objects. Both PSetBlock and SetBlock nodes contain an attribute min_rank which corresponds to the smallest possible rank for an object within the block. This attribute is 0 for the first PSetBlock or SetBlock node in the left subtree of a PSet or Set node, otherwise is one more than the rank of the (possibly repeated) object represented in a left subtree which is a brother of the PSetBlock or SetBlock node.

Figure 8.1 schematically depicts the internal representation of the powerset

$$\{aa, ba, aba, abb, bbb, abaa, bbba, bbbaab\} \in \mathsf{PSet}(\mathsf{Seq}(a+b)) .$$

PSet-nodes are represented by oval nodes, together with the values of the size and min_size attributes. PSetBlock nodes are represented by circular nodes; within each PSetBlock node we write the values of the size and min_rank attributes. The components of the powerset belong to $\mathsf{Seq}(a+b)$ but we have shown them "collapsed" into single rectangular nodes.

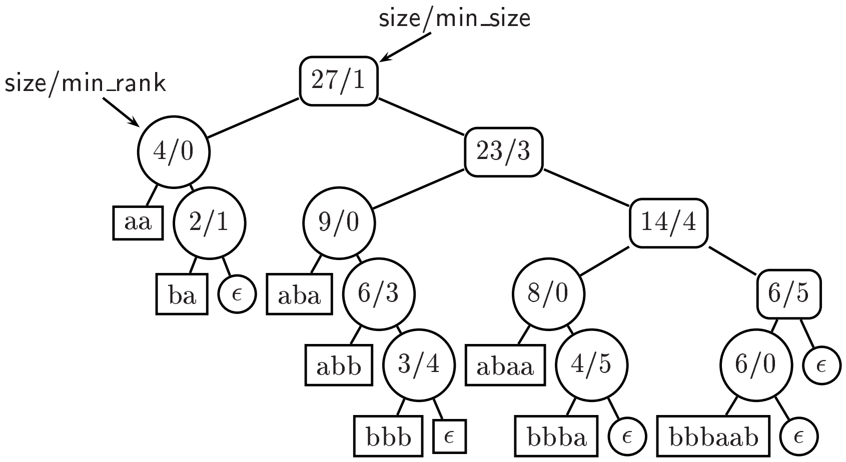


Figure 8.1: Internal representation of a PSet-object.

Figure 8.2 corresponds to the internal representation of the set

$$\{aa, aa, ba, aba, aba, abb, bbb, abaa, bbba, bbbaab\} \in \mathsf{Set}(\mathsf{Seq}(a+b)) \ .$$

We use the same conventions as in Figure 8.2; additionally, the diamond nodes represent Δ-nodes, and store the size and the number of times that a particular component occurs in the set (they have a field called occurrences).

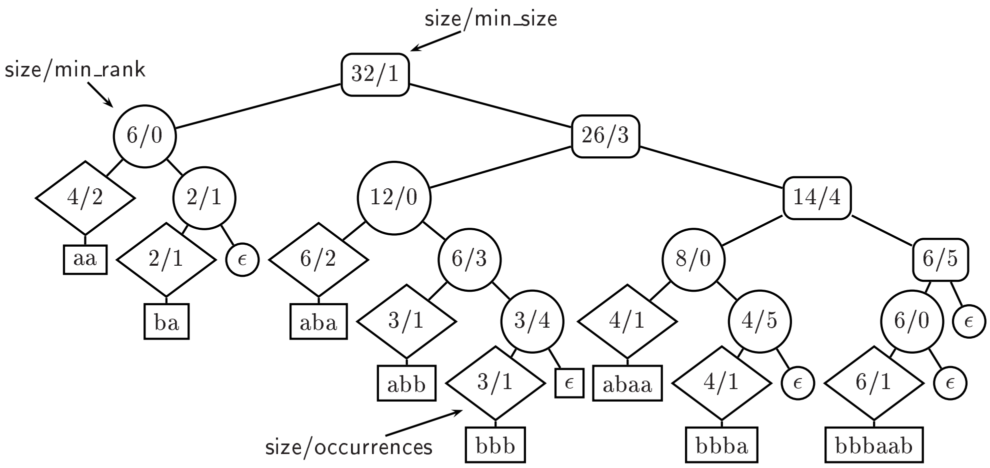


Figure 8.2: Internal representation of a Set-object.

The algorithms to iterate through unlabelled powersets and sets are very similar and they rely on the algorithms *Next_PSetBlock* and *Next_SetBlock*, respectively. We give in Algorithm 44 the pseudocode to generate powersets; except for minor details, the code for sets is identical.

Algorithm 44 Next object for $\mathsf{PSet}(\mathcal{A})$

Next_PSet(p)
    $\triangleright\, p$ points to a node representing an object in $\mathsf{PSet}(\mathcal{A})$ $\triangleleft$
    if not Is_Last_Obj($p\rightarrow\mathsf{right}$) **then** Next_PSet($p\rightarrow\mathsf{right}$); **return**
    else $n := p\rightarrow\mathsf{size}; j := p\rightarrow\mathsf{left}\rightarrow\mathsf{left}\rightarrow\mathsf{size}; l := p\rightarrow\mathsf{left}\rightarrow\mathsf{size}$
        $\triangleright\, l = \#$ of components $\times$ size of components $\triangleleft$
        $\mathcal{A} := p\rightarrow\mathsf{left}\rightarrow\mathsf{left}\rightarrow\mathsf{class}$
    fi
    if not Is_Last_Obj($p\rightarrow\mathsf{left}$) **then**
      Next_PSetBlock($p\rightarrow\mathsf{left}$); $p\rightarrow\mathsf{right}$:= Init_PSet($\mathcal{A}, n-l, j+1$)
      return
    fi
    $k := l/j - 1$
    while $k > 0$ **and** CountPSet($\mathcal{A}, n-k\cdot j, j$) $= 0$ **do** $k := k-1$ **end**
    if $k > 0$ **then**
      $p\rightarrow\mathsf{left}$:= Init_PSetBlock($\mathcal{A}, k, j, 0$); $p\rightarrow\mathsf{right}$:= Init_PSet($\mathcal{A}, n-k\cdot j, j+1$)
      return
    fi
    $j := j+1; k := \lfloor n/j \rfloor$
    while $j \le n$ **and** (Count($\mathcal{A}, j$)$=0$ **or** CountPSet($\mathcal{A}, n-k\cdot j, j$)$=0$) **do**
        if Count($\mathcal{A}, j$) $\neq 0$ **then**
          while $k>0$ **and** CountPSet($\mathcal{A}, n-k\cdot j, j$)$=0$ **do** $k:=k-1$ **end**
        fi
        if Count($\mathcal{A}, j$) $= 0$ **or** $k = 0$ **then** $j := j+1; k := \lfloor n/j \rfloor$ **fi**
    end
    $p\rightarrow\mathsf{left}$:= Init_PSetBlock($\mathcal{A}, k, j, 0$); $p\rightarrow\mathsf{right}$:= Init_PSet($\mathcal{A}, n-k\cdot j, j+1$)
    $\triangleright$ If $n-k\cdot j = 0$ then Init_PSet will return a representation for ϵ $\triangleleft$

Notice that *Next_PSet* uses the procedure *Init_PSet*, which receives an specification of the class of the components of the powerset[1], the total size of the powerset, and the minimum size of the components of the powerset. See Algorithm 45 for details. There is also an analogous procedure *Init_Set* to initialize sets, with the same three parameters. They also use the procedures *Init_PSetBlock* and *Init_SetBlock*, respectively, each receiving four parameters: an specification of the class of the components of the block, the cardinality of the block (k), the size of the components of the block (j), and the smallest allowed rank of its components (r). The implementation of *Init_PSetBlock* and *Init_SetBlock* relies upon efficient algorithms for unranking in $\mathcal{A}_j$; we use algorithms such as those developed in Chapter 4. In Algorithm 46 we just show the powerset case. For sets, it is necessary to do the initialization of the Δ-nodes (that is, to care about the attribute $p\rightarrow\mathsf{left}\rightarrow\mathsf{occurrences}$), to use that $p\rightarrow\mathsf{count} := \binom{a_j - r + k - 1}{k}$ and to modify the input of *Init_SetBlock* depending on the number of occurrences.

[1] We cannot specify the subset of objects of size j (of size $> j, \ldots$) of a class $\mathcal{A}$ in our specification language; hence, we supply the restriction on the size of the components of the powerset as a separate parameter.

Algorithm 45 Function *Init_PSet*

Init_PSet$(\mathcal{A}, n, j)$
  $p :=$ New_Node( ) ▷ New node to initialize its attributes ◁
  $p \rightarrow$class $:=$ PSet$(\mathcal{A})$; $p \rightarrow$size $:= n$
  $p \rightarrow$count $:= \text{CountPSet}(\mathcal{A}, n, j-1)$; $p \rightarrow$rank $:= 0$
  $k := \lfloor n/j \rfloor$; **while** $\text{CountPSet}(\mathcal{A}, n - k \cdot j, j) = 0$ **do** $k := k - 1$ **end**
  $p \rightarrow$left $:= \text{Init\_PSetBlock}(\mathcal{A}, k, j, 0)$; $p \rightarrow$right $:= \text{Init\_PSet}(\mathsf{PSet}(\mathcal{A}), n - k \cdot j, j+1)$
  return p

Algorithm 46 Function *Init_PSetBlock*

Init_PSetBlock$(\mathcal{A}, k, j, r)$
  $p :=$ New_Node( ) ▷ New node to initialize its attributes ◁
  ▷ $p \rightarrow$class is not initialized since none of the algorithms will use it ◁
  $p \rightarrow$size $:= k \cdot j$; $p \rightarrow$count $:= \binom{\text{Count}(\mathcal{A}, j) - r}{k}$; $p \rightarrow$rank $:= 0$
  $p \rightarrow$left $:= \text{Unrank}(\mathcal{A}, j, r)$; $p \rightarrow$right $:= \text{Init\_PSetBlock}(\mathcal{A}, k-1, j, r+1)$
  return p

Finally, *Next_PSet* and *Init_PSet* need the specialized counting routine *CountPSet*$(\mathcal{A}, n, j)$. This routine uses the generic counting function *Count* to compute the number of powersets of $\mathcal{A}$'s of size $> j$ with total size n. For the generation of sets, there exists a similar specialized procedure *CountSet*$(\mathcal{A}, n, j)$.

Algorithm 47 gives the pseudocode for the generation of SetBlocks, the pseudocode for PSetBlocks being very similar and in fact simpler. As usual, when computing the next SetBlock we try to change the right subtree first. If it is not possible, we decrease by one the number of occurrences of the smallest component of the SetBlock (the object in $\mathcal{A}_j$ of smallest rank) and start with the smallest possible SetBlock in the right. If the number of occurrences of the smallest component were already one, we have to move to the next object in $\mathcal{A}_j$, with maximum number occurrences and initialize the right subtree to an empty SetBlock.

The implementation of any of the numerous orderings discussed in Chapter 4 for unlabelled powersets and sets is also simple; it is just a matter of reordering a few lines of code in Algorithm 47, how to traverse values of j, k, ...

Unlabelled cycles do not admit a nice recursive decomposition so we need a rather different approach to iterate through them. It combines Sawada's algorithm for k-ary cycles of fixed content [76] and the algorithm for sets. Despite unlabelled cycles are dealt with in a different way, the representation of cycles and the algorithm have been designed so that they can be integrated and seamlessly work together with the other algorithms. Also, the algorithm works in constant amortized time per generated cycle (provided that the components could be generated in constant amortized time, which we assume indeed to be the case). We give details in Section 8.4 [58].

Algorithm 47 Next object for $\mathcal{A}_j^{\{\!\{k\}\!\}}$

```
Next_SetBlock(p)
    ▷ p points to a node representing an object in A_j^{k} ◁
    A := p→left→left→class
    if not Is_Last_Obj(p→right) then Next_SetBlock(p→right)
    elif p→left→occurrences > 1 then
        p→left→occurrences := p→left→occurrences − 1
        r := p→left→left→rank + 1; j := p→left→left→size
        p→left→size:=p→left→size−j; n:=p→size−p→left→size
        p→right := Init_SetBlock(A, n/j, j, r)
    else ▷ Is_Last_Obj(p→left→left) is false ◁
        r := p→left→left→rank; j := p→left→left→size
        p→left→occurrences := p→size/j; p→left→size := p→size
        Next_Obj(p→left→left); p→right := Init_SetBlock(A, 0, j, r + 1)
    fi
```

8.1.3 Algorithms for Sequences, Sets and PowerSets with Restricted Cardinalities, and Substitutions

As we have shown in Chapter 2 (Table 2.5), the isomorphisms for sequences, sets and powersets with restricted cardinality are similar to the sequences, sets and powersets without restricted cardinality. Thus, its generation algorithms are trivially constructed from the generation algorithms for sequences, sets and powersets studied in the previous subsection.

Substitutions follow the isomorphism (1.1) and they are also generated using the previous algorithms.

Finally, the card-lexicographic order just needs to pre-compute the corresponding cardinality, then we apply the given above algorithms with restricted cardinalities.

8.2 Analysis of the Cost

We have anticipated already that all algorithms are CAT for a large set of unlabelled admissible classes, including the most interesting ones (in particular, those that we call superpolynomial). Furthermore, we give also here Λ-rules to compute the cost of iteration through unlabelled classes.

As we have done for the labelled case, we consider some simplifications in our analysis (see Section 7.4). For instance, we do not take into account the cost of calling *Count* at several places of the algorithms. All the necessary *Count*'s can be computed only once and conveniently stored into tables for later use, either on-demand or in a preprocessing phase. We will charge an unitary cost to all non-recursive steps made in each of the algorithms. Finally, we will also disregard the cost to generate the leaf nodes, and the cost to parse the specification and transform it to standard form.

In a similar vein as we did in Section 7.4 for labelled classes, let $\Lambda\mathcal{A}_n$ be the cost to generate all objects in $\mathcal{A}_n$. Then

$$\Lambda\mathcal{A}_n = f(\mathcal{A}, n) + \sum_{\alpha \in \mathcal{A}_n} \mathrm{cn}(\alpha, \mathcal{A}) \ ,$$

where $f(\mathcal{A}, n)$ denotes the cost to compute the first object in $\mathcal{A}_n$, and $\mathrm{cn}(\alpha, \mathcal{A})$ denotes the cost to apply *Next* to object α. We also assume that $\mathrm{cn}(\alpha^{\perp}, \mathcal{A}) = 0$ if $\alpha^{\perp}$ is the last object in $\mathcal{A}_n$. We denote $\Lambda\mathcal{A}(z)$ the ordinary generating function of the cumulated costs $\Lambda\mathcal{A}_n$:

$$\Lambda\mathcal{A}(z) = \sum_{n \geq 0} \Lambda\mathcal{A}_n \ z^n \ .$$

Hence, $\mu_{\mathcal{A},n} = \Lambda\mathcal{A}_n / a_n$ is the amortized cost per generated object.

Following similar arguments as those given in Theorem 22 and Corollary 6 for labelled classes, we can establish the corresponding Λ-rules for unlabelled admissible classes.

Theorem 25. *Given two admissible unlabelled classes $\mathcal{A}$ and $\mathcal{B}$, then*

1. $\Lambda\epsilon = 1$.
2. $\Lambda Z = z$.
3. $\Lambda(\mathcal{A} + \mathcal{B}) = \Lambda\mathcal{A} + \Lambda\mathcal{B} + [\![\mathcal{A}]\!] + [\![\mathcal{B}]\!] - [\![\mathcal{A} + \mathcal{B}]\!]$.
4. $\Lambda(\mathcal{A} \times \mathcal{B}) = \Lambda\mathcal{A} \cdot [\![\mathcal{B}]\!] + A \cdot \Lambda\mathcal{B} + A \cdot [\![\mathcal{B}]\!] - [\![\mathcal{A} \times \mathcal{B}]\!]$.
5. $\Lambda \mathsf{Seq}(\mathcal{A}) = (\Lambda\mathcal{A} + A) \cdot \frac{[\![\mathsf{Seq}(\mathcal{A})]\!]}{1-A} + \frac{\Lambda\epsilon + [\![\epsilon]\!] - [\![\mathsf{Seq}(\mathcal{A})]\!]}{1-A}$.
6. $\Lambda(\mathcal{A} \circ \mathcal{B}) = \Lambda\mathcal{A}([\![\mathcal{B}]\!]) + A(\Lambda\mathcal{B}) + A([\![\mathcal{B}]\!]) - [\![\mathcal{A} \circ \mathcal{B}]\!]$.

For powersets and sets, the corresponding Λ-rules can be derived from the Λ-rules for unions and products, but they do not have a nice form

$$\Lambda\mathsf{PSet}_n(\mathcal{A}) = \sum_{j=1}^{n} \sum_{k=1}^{\lfloor n/j \rfloor} \Bigg(\#\mathcal{A}_j^{\{k\}} \cdot \Lambda\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) + \tag{8.1}$$

$$\Lambda\mathcal{A}_j^{\{k\}} \cdot [\![\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \neq \emptyset]\!] + \tag{8.2}$$

$$\#\mathcal{A}_j^{\{k\}} \cdot [\![\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \neq \emptyset]\!] \Bigg) - \tag{8.3}$$

$$[\![\mathsf{PSet}_n(\mathcal{A}) \neq \emptyset]\!]$$

and

$$\Lambda\mathsf{Set}_n(\mathcal{A}) = \sum_{j=1}^{n} \sum_{k=1}^{\lfloor n/j \rfloor} \Bigg(\#\mathcal{A}_j^{\{\!\{k\}\!\}} \cdot \Lambda\mathsf{Set}_{n-kj}(\mathcal{A}_{>j}) + \Lambda\mathcal{A}_j^{\{\!\{k\}\!\}} \cdot [\![\mathsf{Set}_{n-kj}(\mathcal{A}_{>j}) \neq \emptyset]\!] +$$

$$\#\mathcal{A}_j^{\{\!\{k\}\!\}} \cdot [\![\mathsf{Set}_{n-kj}(\mathcal{A}_{>j}) \neq \emptyset]\!] \Bigg) - [\![\mathsf{Set}_n(\mathcal{A}) \neq \emptyset]\!] \ ,$$

where $\#\mathcal{A}_j^{\{k\}} = \binom{a_j}{k}$ and $\#\mathcal{A}_j^{\{\!\{k\}\!\}} = \binom{a_j+k-1}{k}$.

We also need to compute the cost to generate PSetBlocks and SetBlocks, that is, $\Lambda\mathcal{A}_j^{\{k\}}$ and $\Lambda\mathcal{A}_j^{\{\!\{k\}\!\}}$. Notice that we have to iterate through PSetBlocks and SetBlocks of size $k \cdot j$, since they have a fixed number k of components which are objects of size j in some class $\mathcal{A}$; therefore, the cumulating generating functions have only one term corresponding to z^{kj}.

In particular, each object in $\mathcal{A}_j$ is generated exactly once to act as the leader of the PSetBlock, and the rest accounts for the cost of generating the PSetBlock with $k-1$ components:

$$\Lambda\mathcal{A}_j^{\{k\}} = \Lambda\mathcal{A}_j + \sum_{\alpha\in\mathcal{A}_j} \Lambda\mathsf{PSet}(\mathcal{A}_j^{(\succ\alpha)}, \mathrm{card} = k-1) \ , \tag{8.4}$$

where, as usual, $\mathcal{A}_j^{(\succ\alpha)} = \{\alpha' \in \mathcal{A}_j \mid \alpha' \succ_{\mathcal{A}_j} \alpha\}$.

The Λ-rule for SetBlocks is similar since each object in $\mathcal{A}_j$ is generated exactly once to act as the leader of the SetBlock, and we have the terms for the costs of generating SetBlocks with components of larger rank when the leader occurs once, twice, three times, etc.:

$$\begin{aligned}\Lambda\mathcal{A}_j^{\{\!\{k\}\!\}} &= \Lambda\mathcal{A}_j + \sum_{\alpha\in\mathcal{A}_j}\left(1 + \sum_{1\le l<k} \Lambda\mathsf{Set}(\mathcal{A}_j^{(\succ\alpha)}, \mathrm{card} = l)\right) - [\![\mathcal{A}_j \neq \emptyset]\!] \\ &= \Lambda\mathcal{A}_j + a_j - [\![a_j \neq 0]\!] + \sum_{\alpha\in\mathcal{A}_j}\sum_{1\le l<k} \Lambda\mathsf{Set}(\mathcal{A}_j^{(\succ\alpha)}, \mathrm{card} = l) \ .\end{aligned}$$

To establish analogous results of Theorem 23, we have to adapt Definition 13 for unlabelled admissible classes.

Definition 14. *An unlabelled admissible class $\mathcal{A}$ is* superpolynomial growing *(superpolynomial, for short) if and only if for any constant c, $n^c = \mathrm{O}(a_n)$.*

The little-endian heuristic described in Section 7.4 also applies for unlabelled classes. We have analogous theorems for unlabelled generation.

Theorem 26. *Let $\mathcal{A}$ and $\mathcal{B}$ be unlabelled admissible classes, then*

1. *If $\mathcal{A}$ and $\mathcal{B}$ are $c_\mathcal{A}$- and $c_\mathcal{B}$-generable, respectively, then $\mathcal{A}+\mathcal{B}$ is $(\max\{c_\mathcal{A}, c_\mathcal{B}\}+1)$-generable.*

2. *If $\mathcal{A}$ and $\mathcal{B}$ are $c_\mathcal{A}$- and $c_\mathcal{B}$-generable, respectively, then $\mathcal{A}\times\mathcal{B}$ is $(c_\mathcal{A}+c_\mathcal{B}+1)$-generable.*

3. *If $\mathcal{B}$ is c-generable and superpolynomial, then $\mathcal{A}\circ\mathcal{B}$ is c-generable.*

4. *If $\mathcal{A}$ is c-generable and superpolynomial, and $\mathcal{B} \ll \mathcal{A}$, then $\mathcal{A}+\mathcal{B}$ and $\mathcal{B}+\mathcal{A}$ are c-generable.*

   In particular, if $\mathcal{A}$ and $\mathcal{B}$ are $c_\mathcal{A}$- and $c_\mathcal{B}$-generable, respectively, and both classes are superpolynomial, then $\mathcal{A}+\mathcal{B}$ is $\max\{c_\mathcal{A}, c_\mathcal{B}\}$-generable.

5. *If either $\mathcal{A}$ or $\mathcal{B}$ is c-generable and superpolynomial, then $\mathcal{A}\times\mathcal{B}$ is c-generable (eventually using the little-endian heuristic).*

   Let $\mathcal{A}$ and $\mathcal{B}$ be superexponential. In particular, if $\mathcal{A}$ is $c_{\mathcal{A}}$-generable and $\mathcal{B}$ is not $c_{\mathcal{B}}$-generable for any constant $c_{\mathcal{B}}$, then $\mathcal{A}\times\mathcal{B}$ is $c_{\mathcal{A}}$-generable; if $\mathcal{B}$ is $c_{\mathcal{B}}$-generable and $\mathcal{A}$ is not $c_{\mathcal{A}}$-generable for any constant $c_{\mathcal{A}}$, then $\mathcal{A}\times\mathcal{B}$ is $c_{\mathcal{B}}$-generable; and if $\mathcal{A}$ and $\mathcal{B}$ are $c_{\mathcal{A}}$- and $c_{\mathcal{B}}$-generable, respectively, then $\mathcal{A}\times\mathcal{B}$ is $\min\{c_{\mathcal{A}},c_{\mathcal{B}}\}$-generable.

Since all algorithms are build upon on unions and products, we have an equivalent result of Theorem 24 for unlabelled admissible classes.

Theorem 27. *Let $\mathcal{C}$ be an superpolynomial unlabelled admissible class, then $\mathcal{C}$ is c-generable, for some constant $c \geq 1$.*

Proofs of these two theorems follow similar steps and ideas as their labelled counterparts.

We are just going to sketch the proof for powersets. Let $\mathsf{PSet}(\mathcal{A})$ be a superpolynomial class. Since for any class $\mathcal{A}$ we have $\Lambda\mathcal{A}_n = \mathrm{O}(n\cdot a_n)$, the $\Lambda\mathcal{A}_j^{\{k\}} \leq c\cdot j\cdot k\cdot \#\mathcal{A}_j^{\{k\}}$, for some constant $c\geq 1$. In particular, Equation (8.4) is upper bounded by $\mathrm{O}(\#(\vartheta\mathcal{A}_j^{\{k\}}))$:

$$\begin{aligned}
\Lambda\mathcal{A}_j^{\{k\}} &= \Lambda\mathcal{A}_j + \sum_{\alpha\in\mathcal{A}_j} \Lambda\mathsf{PSet}(\mathcal{A}_j^{(\succ\alpha)}, \mathrm{card}=k-1)\\
&= \sum_{\alpha_1\in\mathcal{A}_j} \mathrm{cn}(\alpha_1,\mathcal{A}_j) \sum_{\alpha_2\in\mathcal{A}_j^{(\succ\alpha_1)}} \mathrm{cn}(\alpha_2,\mathcal{A}_j)\ \ldots \sum_{\alpha_k\in\mathcal{A}_j^{(\succ\alpha_{k-1})}} \mathrm{cn}(\alpha_k,\mathcal{A}_j)\\
&\leq \sum_{\alpha_1\in\mathcal{A}_j} c_1\cdot|\alpha_1| \sum_{\alpha_2\in\mathcal{A}_j^{(\succ\alpha_1)}} c_2\cdot|\alpha_2|\ \ldots \sum_{\alpha_k\in\mathcal{A}_j^{(\succ\alpha_{k-1})}} c_k\cdot|\alpha_k|\\
&\leq \max\{c_1,c_2,\ldots,c_k\}\cdot j\cdot k\cdot \#\mathcal{A}_j^{\{k\}} = c\cdot j\cdot k\cdot\#\mathcal{A}_j^{\{k\}}\ ,
\end{aligned}$$

with $c = \max\{c_1,c_2,\ldots,c_k\}$ for some constants $c_1,c_2,\ldots,c_k\geq 1$.

Moreover, since $\mathsf{PSet}(\mathcal{A})$ is superpolynomial, then $\mathsf{PSet}(\mathcal{A}_{>j})$ is too. If $\mathsf{PSet}(\mathcal{A}_{>j})$ were not superpolynomial, then $\mathsf{PSet}(\mathcal{A})$ would not be superpolynomial either, because, for any $j>0$, $\mathsf{PSet}_n(\mathcal{A})$ minus $\mathsf{PSet}_n(\mathcal{A}_{>j})$ has a finite number of objects. So, (8.2) is upper bounded by $c\cdot\#\mathsf{PSet}(\mathcal{A})$:

$$\begin{aligned}
\sum_{j=1}^{n}\sum_{k=1}^{\lfloor n/j\rfloor} \Lambda\mathcal{A}_j^{\{k\}}\cdot [\![\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})\neq\emptyset]\!] &\leq \sum_{j=1}^{n}\sum_{k=1}^{\lfloor n/j\rfloor} c\cdot j\cdot k\cdot\#\mathcal{A}_j^{\{k\}}\cdot[\![\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j})]\!]\\
&\ll \sum_{j=1}^{n}\sum_{k=1}^{\lfloor n/j\rfloor} c\cdot\#\mathcal{A}_j^{\{k\}}\cdot\#\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \qquad (8.5)\\
&= c\cdot\#\mathsf{PSet}(\mathcal{A})\ .
\end{aligned}$$

Now it is clear that (8.3) is also upper bounded by $\#\mathsf{PSet}(\mathcal{A})$ as well. Consequently, from (8.1) and (8.5), we can establish that

$$\Lambda\mathsf{PSet}_n(\mathcal{A}) \asymp \sum_{j=1}^{n} \sum_{k=1}^{\lfloor n/j \rfloor} \#\mathcal{A}_j^{\{k\}} \cdot \Lambda\mathsf{PSet}_{n-kj}(\mathcal{A}_{>j}) \ .$$

Hence, by structural induction, $\mathsf{PSet}(\mathcal{A})$ is $c_{\mathsf{PSet}(\mathcal{A})}$-generable, for some constant $c_{\mathsf{PSet}(\mathcal{A})} \geq 1$.

For sets, similar steps also prove that if $\mathsf{Set}(\mathcal{A})$ is superpolynomial, then $\mathsf{Set}(\mathcal{A})$ is $c_{\mathsf{Set}(\mathcal{A})}$-generable, for some constant $c_{\mathsf{Set}(\mathcal{A})} \geq 1$.

8.2.1 Examples

Non-Superpolynomial Classes

There exist unlabelled classes that are not generable in CAT.

1. For instance, the class $\mathcal{S} = \mathsf{Seq}(Z)$ is not generated in CAT because it has just one object of each size n:

$$\mu_{\mathcal{S},n} = \frac{\Lambda\mathcal{S}_n}{s_n} = \frac{[z^n]\frac{2-z}{(1-z)^2}}{[z^n]\frac{1}{1-z}} = n+2 \ .$$

   Let $\mathcal{A}^{(0)} = \mathsf{Seq}(Z)$ and $\mathcal{A}^{(k)} = Z \times \mathcal{A}^{(k-1)}$ for any $k > 0$. Then $A^{(k)} = z^k/(1-z)$ and $\Lambda\mathcal{A}^{(k)} = (z^k\,((k+2)-(k+1)\,z))/(1-z)$. Hence we have that $\mu_{\mathcal{A}^{(k)},n} = n+2$.

   In the same vein, let $\mathcal{B}^{(k)} = \mathcal{B}^{(k-1)} \times Z$, for any $k > 0$, where $\mathcal{B}^{(0)} = \mathsf{Seq}(Z)$. Then $\mu_{\mathcal{B}^{(k)},n} = n+2$.

2. Consider now $\mathcal{C}^{(1)} = \mathsf{Seq}(Z)$ and $\mathcal{C}^{(k)} = \mathcal{C}^{(1)} \times \mathcal{C}^{(k-1)}$, for any $k > 1$. Notice that $\mathcal{C}^{(k)}$ is the class of k-tuples of sequences. This is not superpolynomial since there are only $\Theta(n^k)$ objects of size n. We have here that $\mu_{\mathcal{C}^{(k)},n} = 2\,n + \mathrm{o}(n)$.

   For the specification $\mathcal{D}^{(k)} = \mathcal{D}^{(k-1)} \times \mathcal{D}^{(1)}$, for any $k > 1$, where $\mathcal{D}^{(1)} = \mathsf{Seq}(Z)$, we have that $\mu_{\mathcal{C}^{(k)},n} = k\,n + \mathrm{o}(n)$.

3. Another example is the so-called k-shuffles, noted by $\mathcal{L}_k$. A k-shuffle is a sequence of a's and b's that contains exactly k b's (see Subsection 4.3). Let $\mathcal{L}_0 = \mathsf{Seq}(a)$ and $\mathcal{L}_k = \mathsf{Seq}(a) \times (b \times \mathcal{L}_{k-1})$ for any $k > 0$. It is clear that $[z^n]\,L_k(z) = [z^n]\,z^k/(1-z)^{k+1} = \binom{n}{k}$. From Λ-rules it results that $\mu_{\mathcal{L}_k,n} = 2\,n + \mathrm{o}(n)$, for any $k > 0$. If we use the alternative specification $\mathcal{K}_0 = \mathsf{Seq}(a)$ and $\mathcal{K}_k = \mathcal{K}_{k-1} \times (b \times \mathsf{Seq}(a))$ for any $k > 0$, then $\mu_{\mathcal{K}_k,n} = n + \mathrm{o}(n)$.

However, as it happened with labelled classes, if we do not account for the cost to generate the first object of each class and size (i.e. $\Lambda\epsilon = \Lambda Z = 0$), then $\Lambda\mathcal{A}_n = \Theta(a_n)$ for the previous examples; in particular, $\mu_{\mathcal{A}^{(k)},n} = 0$, $\mu_{\mathcal{B}^{(k)},n} = 0$, $\mu_{\mathcal{C}^{(k)},n} = 1 + \mathrm{o}(1)$, $\mu_{\mathcal{D}^{(k)},n} = k - 1 + \mathrm{o}(1)$, $\mu_{\mathcal{L}_k,n} = 1 + \mathrm{o}(1)$, $\mu_{\mathcal{K}_k,n} = 0$, for any $k > 0$.

Superpolynomial Classes

Let us now consider examples where we have generation in CAT.

1. For binary trees $\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}$, we have that the cumulating cost is

$$\Lambda\mathcal{B} = \frac{\Lambda Z - B \cdot [\![\mathcal{B}]\!] - [\![\mathcal{B} \times \mathcal{B}]\!]}{1 - [\![\mathcal{B}]\!] - B} \; .$$

   Thus, $\Lambda\mathcal{B}_n = 14 \cdot b_n + \mathrm{o}(b_n)$. In the same vein, we have that $\mu_{\mathcal{B}\times\mathcal{B},n} = \mu_{\mathcal{B},n}$.

   For sequences of binary trees $\mathcal{S} = \mathsf{Seq}(\mathcal{B})$, we also have the same result $\Lambda\mathcal{S}_n = 14 \cdot s_n + \mathrm{o}(s_n)$.

2. For Motzkin trees, our algorithms also have CAT performance; the constants differ when we use different alternative specifications:

   - For $\mathcal{M} = Z + Z \times \mathcal{M} + Z \times (\mathcal{M} \times \mathcal{M})$, one readily gets $\mu_{\mathcal{M},n} = \frac{29}{3} + \mathrm{o}(1) = 9.666666666\ldots + \mathrm{o}(1)$.
   - For $\mathcal{M} = Z + Z \times \mathcal{M} + (Z \times \mathcal{M}) \times \mathcal{M}$, one gets $\mu_{\mathcal{M},n} = 10 + \mathrm{o}(1)$.
   - Finally, for $\mathcal{M} = Z \times (\epsilon + \mathcal{M} + \mathcal{M} \times \mathcal{M})$, we have that $\mu_{\mathcal{M},n} = 13 + \mathrm{o}(1)$.

8.3 Variants and Heuristics

It is easy to adapt the previous algorithms to the orders given in Section 2.3: LSCR order, LSCr order, LScR order, LScr order, etc. However, they do not give any improvement of the performance, so we will omit them. The same happens with the boustrophedonic product.

On the other hand, it should be possible to use the *delta decomposition* to generate powersets and sets. It consists in the following isomorphisms:

$$\mathsf{PSet}(\mathcal{A}) = \epsilon + \bigcup_{\alpha\in\mathcal{A}} \alpha \times \mathsf{PSet}(\mathcal{A}^{(\succ\alpha)})$$

and

$$\mathsf{Set}(\mathcal{A}) = \epsilon + \bigcup_{\alpha\in\mathcal{A}} \Delta(\alpha) \times \mathsf{Set}(\mathcal{A}^{(\succ\alpha)}) \; ,$$

respectively, where $\mathcal{A}^{(\succ\alpha)}$ again is the subset of $\mathcal{A}$-objects whose size is strictly larger than $|\alpha|$ or their rank is larger than the rank of α (i.e. the subset of $\alpha' \in \mathcal{A}$ such that $\alpha' \succ \alpha$). We just note that it follows the same steps as algorithms of Subsection 8.1.2, but on the left child node we just have one object (α) with the corresponding cardinality. Moreover, the analysis would follows the same proof but with the corresponding delta decomposition.

Finally, we have implemented the algorithms using the *finger pointer* with the same behaviour as in Subsection 7.3.

8.4 Generating Cycles

First of all, we describe the canonical form of a cycle. Given a sequence $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$, we say that α is a canonical cycle if and only if, for all non-trivial circular rotations $\alpha_\sigma = (\alpha_{\sigma(1)}, \alpha_{\sigma(2)}, \ldots, \alpha_{\sigma(k)})$ of α, α is smaller than or equal to α_σ in lexicographic order, that is, $\exists i : (\alpha_i \prec \alpha_{\sigma(i)}$ and $\forall l : 1 \leq l < i : \alpha_l = \alpha'_l)$, or $\forall l : 1 \leq l \leq k : \alpha_l = \alpha_{\sigma(l)}$. As usual, we make an abuse of notation because we write $\alpha_i \prec \alpha_j$ to mean that either $(|\alpha_i| < |\alpha_j|)$ or $(|\alpha_i| = |\alpha_j|$ and $\alpha_i \prec \alpha_j)$.

This definition provides a first way to generate unlabelled cycles: Generate sequences and reject those which are not in canonical form. However, this method is not CAT because there are too many unlabelled sequences to be rejected. The new algorithm presented here is CAT provided that the class $\mathcal{A}$ is CAT too. The algorithm is based in two steps. Given a class $\mathcal{A}$ and a size n, all unlabelled cycles of $\mathcal{A}$ can be generated as follows:

1. Generate a set of size n. Let γ be such set:

   $$\gamma = \{\alpha_1^{d_1}, \alpha_2^{d_2}, \ldots, \alpha_q^{d_q}\} \in \mathsf{Set}_n(\mathcal{A}) \ ,$$

   where $\alpha_1 \prec \alpha_2 \prec \cdots \prec \alpha_q$ and $n = |\gamma| = \sum_{1 \leq i \leq q} d_i \ |\alpha_i|$.

   We note that $d = [d_1, \ldots, d_q]$ is the sequence with the number of ocurrences (also called frequencies or *densities*) of the different objects in γ; that is, d_i indicates how many times the i-th different object α_i appears in γ.

2. For each set generated in the first step, generate all canonical cycles that can be built just using the components of that set.

   It is possible to generate all different canonical cycles from the densities d_i using a CAT algorithm adapting Sawada's algorithm for the *generation of unlabelled cycles with fixed density* [58, 75, 76].

   The number of canonical cycles that can be generated from γ is

   $$\frac{1}{N} \sum_{j \mid \gcd(d_1, d_2, \ldots, d_q)} \phi(j) \ \frac{(N/j)!}{(d_1/j)! \ \cdots \ (d_q/j)!} \ ,$$

   where $\phi(n)$ is Euler's totient function (see Section 2.1) and $N = d_1 + \cdots + d_q$ is the cardinality of γ.

   For instance, given the set $\{\alpha^2, \beta^2, \gamma^1\}$, the densities are $[2, 2, 1]$, and the number of cycles generated in step 2 is

   $$\frac{1}{5} \sum_{j \mid \gcd(2,2,1)} \frac{\phi(j) \ (5/j)!}{(2/j)!(2/j)!(1/j)!} = \frac{1}{5} \ \frac{\phi(1) \ (5/1)!}{(2/1)!(2/1)!(1/1)!} = 6 \ ,$$

   namely,

   $$\begin{array}{lll} (\alpha, \alpha, \beta, \beta, \gamma), & (\alpha, \alpha, \beta, \gamma, \beta), & (\alpha, \alpha, \gamma, \beta, \beta), \\ (\alpha, \beta, \alpha, \beta, \gamma), & (\alpha, \beta, \alpha, \gamma, \beta), & (\alpha, \beta, \beta, \alpha, \gamma). \end{array}$$

To represent a cycle we will use a tree structure similar to those for sequences, sets, etc. The root node will be a Cycle-node and will have two children. The left child (Set-node) is the tree representation of the underlying set γ. The right child (*FixDen*-node) contains a single node (of type *FixDen*) which stores information about the densities and about the order of the components of γ in that particular cycle. This *FixDen*-node has also other special attributes needed by Sawada's algorithm.

For instance, Figure 8.3 schematically depicts the internal representation of the cycle

$$(a, a, d, a, d) \in \mathsf{Cycle}(a + b + c + d) \ .$$

In the attribute cyc of Figure 8.3, each 0 represents the object a (the first smallest component of the current set $\{a^3, d^2\}$), and each 1 represents an occurrence of the object d (the second smallest component). The rectangular node with rounded corners and double line is a Cycle-node. The rectangular node with double line is a *FixDen*-node. Note that it just stores the information needed by Sawada's algorithm and it tells the order "to read" the components in the set object to its left.

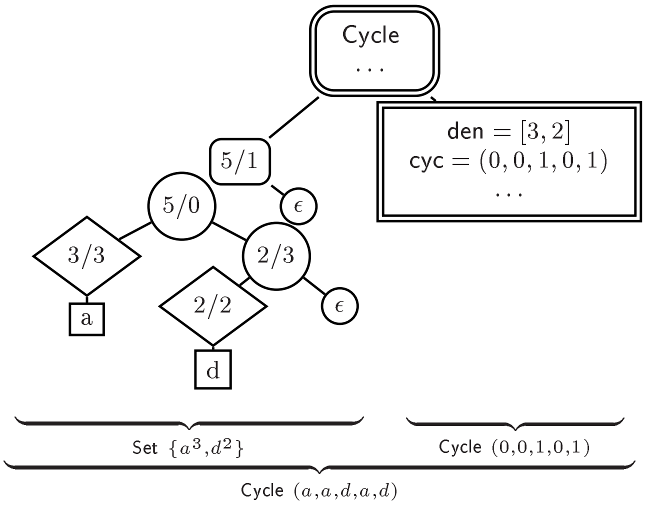


Figure 8.3: Internal representation of a Cycle-object.

8.4.1 Algorithm to Generate Unlabelled Cycles

We now describe in more detail the algorithm just sketched in the previous section.

To begin with, we need the procedure *Init_Cycle*$(\mathcal{A}, n)$ to generate the first Cycle$(\mathcal{A})$ of size n. In Algorithm 48 we show the details. We see that it is necessary to generate the first set of $\mathcal{A}$ of size n, that is, to call *Init_Set*(Set$(\mathcal{A}), n)$. Once we have computed the first set of $\mathcal{A}$ of size n, say $\gamma = \{\alpha_1^{d_1}, \alpha_2^{d_2}, \ldots, \alpha_q^{d_q}\}$, we compute the densities $[d_1, d_2, \ldots, d_q]$ of its components using the function *Densities*. This function traverses the tree representation of the set γ and retrieves the densities which appear at Δ-nodes. Finally, function *Init_FixDen* initializes the attributes of the first canonical cycle generated from the densities $[d_1, d_2, \ldots, d_q]$ and the attributes required by Sawada's algorithm.

Algorithm 48 Function *Init_Cycle*

Init_Cycle($\mathcal{A}, n$)
  $p := \mathsf{New\_Node}(\ )$ ▷ New node to initialize its attributes ◁
  $p \to \mathsf{class} := \mathsf{Cycle}(\mathcal{A}); p \to \mathsf{size} := n$
  $p \to \mathsf{count} := \mathrm{Count}(\mathsf{Cycle}(\mathcal{A}), n); p \to \mathsf{rank} := 0$
  ▷ Function Init_Set was introduced in Subsection 8.1.2 ◁
  $p \to \mathsf{left} := \mathrm{Init\_Set}(\mathcal{A}, n, 1)$
  ▷ Densities($p \to \mathsf{left}$) computes the densities $[d_1, d_2, \ldots, d_q]$ of the ◁
  ▷ components of $p \to \mathsf{left}$ ◁
  $p \to \mathsf{right} := \mathrm{Init\_FixDen}(\mathrm{Densities}(p \to \mathsf{left}))$
  return p

Algorithm 49 describes *Init_FixDen*. It is fairly straightforward. Notice that the first cycle is (see lines (7)-(10))

$$(\overbrace{0, \ldots, 0}^{d_1 \text{ times}}, \overbrace{1, \ldots, 1}^{d_2 \text{ times}}, \ldots, \overbrace{q-1, \ldots, q-1}^{d_q \text{ times}}) \ ,$$

where 0 represents the smallest component of the current set (α_1), 1 represents the second smallest component of the current set (α_2), and so on.

Algorithm 49 Function *Init_FixDen* (to call Sawada's algorithm)

1. Init_FixDen($[d_1, d_2, \ldots, d_q]$)
2. $p := \mathsf{New\_Node}(\ )$ ▷ New node to initialize its attributes ◁
3. ▷ $p \to \mathsf{class}$ is not initialized since none of algorithms will use it ◁
4. $N := d_1 + d_2 + \cdots + d_q; p \to \mathsf{size} := N; p \to \mathsf{den} := [d_1, d_2, \ldots, d_q]$
5. $p \to \mathsf{count} := \frac{1}{N} \Sigma_{j \mid \gcd(d_1, d_2, \ldots, d_q)} \phi(j) \frac{(N/j)!}{(d_1/j)! \cdots (d_q/j)!}; p \to \mathsf{rank} := 0$
6. ... ▷ Additional attributes needed by Sawada's algorithm [76] ◁
7. $k := 1$
8. **for** i **from** 0 **to** $q-1$ **do**
9. **for** j **from** 1 **to** d_i **do** $p \to \mathsf{cyc}[k] := i; k := k+1$ **end**
10. **end**
11. $p \to \mathsf{left} := \mathsf{NULL}; p \to \mathsf{right} := \mathsf{NULL}$
12. **return** p

Finally, Algorithm 50 generates successive canonical cycles. We check if it is possible to compute the next cycle with the current densities; if yes, we update $p \to \mathsf{right}$ using Sawada's algorithm in *Next_FixDen* (in particular, the attribute $p \to \mathsf{right} \to \mathsf{cyc}$); if not, we compute the next set in $p \to \mathsf{left}$ and initialize $p \to \mathsf{right}$.

The order in which cycles are generated by this algorithm is not the lexicographic order. For instance, if $\alpha, \beta, \gamma \in \mathcal{A}_n$ and $\alpha \prec_{\mathcal{A}} \beta \prec_{\mathcal{A}} \gamma$, since $\{\alpha^2, \beta^2, \gamma^2\} \prec_{\mathsf{Set}(\mathcal{A})} \{\alpha^2, \beta, \gamma^3\}$,

Algorithm 50 Next object for $\mathsf{Cycle}(\mathcal{A})$

Next_Cycle(p)
  if not Is_Last_Obj($p \to \mathsf{right}$) **then**
    $p \to \mathsf{right} \to \mathsf{rank} := p \to \mathsf{right} \to \mathsf{rank} + 1$
    $p \to \mathsf{right} \to \mathsf{cyc} :=$ Next_FixDen($p \to \mathsf{right}$) ▷ Sawada's algorithm ◁
  else Next_Obj($p \to \mathsf{left}$) ▷ To generate the following set ◁
     $p \to \mathsf{right} :=$ Init_FixDen(Densities($p \to \mathsf{left}$))
  fi
  return p

then the cycle $(\alpha, \alpha, \gamma, \beta, \beta, \gamma)$ is generated before the cycle $(\alpha, \alpha, \beta, \gamma, \gamma, \gamma)$. However, $(\alpha, \alpha, \gamma, \beta, \beta, \gamma)$ is larger than $(\alpha, \alpha, \beta, \gamma, \gamma, \gamma)$ in lexicographic order. In general, since Sawada's algorithm generates all (canonical) cycles of fixed density in lexicographic order, given two canonical cycles $\alpha = (\alpha_1, \ldots, \alpha_k), \alpha' = (\alpha'_1, \ldots, \alpha'_{k'}) \in \mathsf{Cycle}(\mathcal{A})$, α is generated by Algorithm 50 before α' if and only if $\{\alpha_1, \ldots, \alpha_k\} \prec_{\mathsf{Set}(\mathcal{A})} \{\alpha'_1, \ldots, \alpha'_{k'}\}$, or $\{\alpha_1, \ldots, \alpha_k\} = \{\alpha'_1, \ldots, \alpha'_{k'}\}$ and $\alpha \prec_{\mathsf{Seq}(\mathcal{A})} \alpha'$.

On the other hand, cycles of restricted cardinality can be easily generated if we generate the basis sets γ obeying the cardinality restriction.

8.4.2 Analysis of the Cost

As we have done with other classes, the cost to generate all cycles of $\mathcal{A}$ and size n is the cost to generate the first cycle of $\mathcal{A}$ and size n plus the cost to iterate the generation algorithm until to get the last cycle of $\mathcal{A}$ and size n:

$$\Lambda\mathsf{Cycle}_n(\mathcal{A}) = f(\mathsf{Cycle}(\mathcal{A}), n) + \sum_{\gamma \in \mathsf{Cycle}_n(\mathcal{A})} \mathrm{cn}(\gamma, \mathsf{Cycle}_n(\mathcal{A})) \ .$$

Now, if $\gamma \in \mathsf{Cycle}_n(\mathcal{A})$ and we follow the exhaustive generation described here, then we can consider $\gamma = \langle \alpha, \beta_\alpha \rangle$ for some $\alpha \in \mathsf{Set}(\mathcal{A})$ and $\beta_\alpha \in \mathit{FixDen}(\alpha)$, where $\mathit{FixDen}(\alpha)$ denotes the set of cycles whose components are those in α. So, the cost to generate the next labelled object in $\mathsf{Cycle}_n(\mathcal{A})$ is given by

$$\mathrm{cn}(\langle \alpha, \beta_\alpha \rangle, \mathsf{Cycle}_n(\mathcal{A})) =$$
$$\begin{cases} \mathrm{cn}(\beta_\alpha, \mathit{FixDen}(\alpha)) & \text{if } \beta_\alpha \neq {\beta_\alpha}^{\perp}, \\ 1 + \mathrm{cn}(\alpha, \mathsf{Set}(\mathcal{A})) + f(\mathit{FixDen}(\alpha'), n) & \text{if } \alpha \neq \alpha^{\perp} \text{ and } \beta_\alpha = {\beta_\alpha}^{\perp}, \text{ where } \alpha' \text{ is the} \\ & \quad \text{next object of } \alpha \text{ in } \mathsf{Set}_n(\mathcal{A}), \\ 0 & \text{if } \alpha = \alpha^{\perp},\ \beta_\alpha = {\beta_\alpha}^{\perp}. \end{cases}$$

Since the exhaustive generation of sets and cycles with fixed densities can be done in CAT (see Section 8.2 and [76]), we just have to study the cost

$$\sum_{\alpha \in \mathsf{Set}_n(\mathcal{A})} f(\mathit{FixDen}(\alpha), n) \ ,$$

to prove the exhaustive generation of cycles is also in CAT.

Notice that it requires to initialize the densities of $\alpha \in \mathsf{Set}(\mathcal{A})$ (line (6) in Algorithm 50) and the attributes used in Sawada's algorithm (even one array of length n [76]). Since $f(\mathit{FixDen}(\alpha), n) = \mathrm{O}(n)$, then $\sum_{\alpha \in \mathsf{Set}_n(\mathcal{A})} f(\mathit{FixDen}(\alpha), n)$ has cost $\mathrm{O}(\#(\vartheta \mathsf{Set}_n(\mathcal{A})))$. Hence, to prove that the generation of cycles is in CAT, we would need to show that $\#\mathsf{Cycle}_n(\mathcal{A}) = \mathrm{O}(\#(\vartheta \mathsf{Set}_n(\mathcal{A})))$. But, this is not always true. However, we can update the attributes of $\beta_{\alpha'}$ from those of β_α when β_α is the last element in $\mathit{FixDen}(\alpha)$, $\beta_{\alpha'}$ is the first element in $\mathit{FixDen}(\alpha')$, and α' is the next object of α in $\mathsf{Set}_n(\mathcal{A})$.

For instance, the densities of

$$\alpha = \{aa^2, ba^3, aba^3, abb, bbb, abaa^5, abab, bbba^3, bbbaa^2, bbbaab^2\}$$

where $\alpha \in \mathsf{Set}(\mathsf{Seq}(a + b))$, are $[2, 3, 3, 1, 1, 5, 1, 3, 2, \mathbf{2}]$, and the densities of

$$\alpha' = \{aa^2, ba^3, aba^3, abb, bbb, abaa^5, abab, bbba^3, bbbaa^2, bbbaab, bbbabb\}$$

are $[2, 3, 3, 1, 1, 5, 1, 3, 2, \mathbf{1}, \mathbf{1}]$, where we have just changed the last density "2" by "1, 1".

In general, to update these attributes of $\beta_{\alpha'}$ from those of β_α in an efficient manner, we will have to add some attributes (densities den, etc.) in Set-nodes, and to update them while iterating in Set-node. Then, the contribution of $\sum_{\alpha \in \mathsf{Set}_n(\mathcal{A})} f(\mathit{FixDen}(\alpha), n)$ would be $\mathrm{O}(\Lambda \mathsf{Set}_n(\mathcal{A}))$, and hence $\Lambda \mathsf{Cycle}_n(\mathcal{A}) \leq c_{\mathsf{Cycle}(\mathcal{A})} \cdot \#\mathsf{Cycle}_n(\mathcal{A})$ since $\Lambda \mathsf{Set}_n(\mathcal{A}) \leq c_{\mathsf{Set}(\mathcal{A})} \cdot \#\mathsf{Set}_n(\mathcal{A}) \leq c_{\mathsf{Set}(\mathcal{A})} \cdot \#\mathsf{Cycle}_n(\mathcal{A})$, for some constants $c_{\mathsf{Cycle}(\mathcal{A})}, c_{\mathsf{Set}(\mathcal{A})} \geq 1$.

Part IV

Conversions of Representations of Combinatorial Objects

Chapter 9

Conversions of Representations

One of the main goals of this thesis has been to furnish a general and a common framework for the generation of combinatorial objects: the drawing or the random generation [26], the unranking (Chapters 3 and 4), the ranking (Chapter 6) and the exhaustive generation (Chapters 7 and 8).

For instance, we could generate the binary trees $\mathcal{B}$ of size n from the N_0-th object to the N_f-th object (in an specific order, lexicographic, boustrophedonic, etc.) using unranking:

$$\textbf{for } i \textbf{ from } N_0 \textbf{ to } N_f \textbf{ do } \text{Unrank}(\mathcal{B}, n, i) \textbf{ end}$$

However, it would be interesting (and in fact more efficient) to generate an iterator *itr* for the N_0-th object with the function *Unrank* and, afterwards, to apply the function *Next* on the given iterator to generate the following $N_f - N_0$ objects:

$$\text{itr} := \text{Unrank}(\mathcal{B}, n, N_0); \textbf{for } i \textbf{ from } N_0 + 1 \textbf{ to } N_f \textbf{ do } \text{Next}(\text{itr}) \textbf{ end}$$

From this point of view, it would be very convenient that all these functions generated, and received as input when necessary, combinatorial objects in the internal normal form (INF, for short) presented in Part III. This would allow for a seamless interaction of these algorithms as suggested by Figure 9.1.

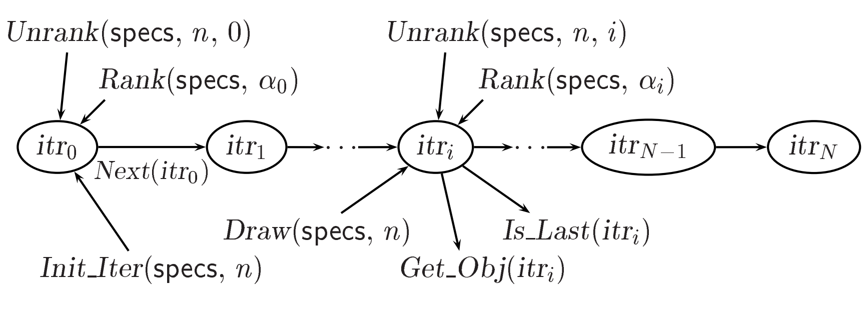


Figure 9.1: Common framework for the ordered generation.

Thus, it would be useful to provide new functions with the following *profile*:

- *Draw_itr*(specs, size=n) $\rightarrow$ *itr*

- $Unrank\_itr(\mathsf{specs},\ \mathsf{size}{=}n,\ \mathsf{rank}{=}i) \to itr$
- $Rank\_itr(\mathsf{specs},\ \alpha) \to itr$
- $Init\_Iter\_\mathrm{itr}(\mathsf{specs},\ \mathsf{size}{=}n) \to itr;\quad Next(itr) \to itr$

where specs is a tuple [class, spec_equations, options], being options some (alternative) specific words to design the *labelling type* (unlabelled or labelled), the *order* (lexicographic or boustrophedonic), the *labelling order* (SF or PF), etc. Then, for intance, we could have utility functions such as

$$Unrank(\mathcal{A}, n, i) \equiv Get\_Obj(Unrank\_itr(\mathsf{specs}, \mathsf{size} = n, \mathsf{rank} = i))$$

or

$$Rank(\alpha, \mathsf{specs}) \equiv Rank\_itr(\alpha, \mathsf{specs}).\mathsf{root} \to \mathsf{rank}\ .$$

Therefore, unranking would work in two steps: i) given the specification specs, size n and rank i generate an iterator for the object of rank i of the given size n; ii) return the object itself, by applying a transformation from iterator representation to the so-called linear normal form (denoted by LNF).

Ranking should also be implemented as a two step procedure: i) given the LNF of the object, produce the corresponding iterator using a transformation; ii) get the actual rank with a simple query to the rank attribute stored at the root of the iterator representation obtained in the first step.

Random generation should also produce a suitable iterator representation of the random object.

Alternatively, unranking and random generation could produce the so-called tuple normal form (denoted by TNF) which can be easily converted to iterator form (in a similar way to a transformation from linear form to iterator form, but we will not give this transformation from TNF to a corresponding iterator here).

Finally, the exhaustive generation works as we have described before (see Part III): i) generate or update the corresponding iterator; ii) return the object itself.

In this chapter we formalize the definitions of lineal normal form (LNF), tuple normal form (TNF) and internal or iterator normal form (INF). In Section 9.2, we present the transformation from a LNF term into its corresponding TNF, and the reverse transformation. Finally, in Section 9.3, we detail the transformation from a LNF term into its corresponding INF, and the reverse transformation.

9.1 Representations

There are three kind of representations of combinatorial objects that we have considered in this thesis: the *linear normal form* (denoted by LNF) that we use for the human-readable input and output of the generation algorithms, the *tuple normal form* (TNF) used in the unranking algorithms (Part II), and the *internal or iterator normal form* (INF) to store the objects as a tree structure to do the exhaustive generation (Part III). Notice that the first one is suitable for interaction with the users, but the other are internal formats, and are not.

The goal of this chapter is to establish a connection between all normal forms (LNF, TNF and INF) and all generation problems (drawing, unranking, ranking and iteration). Here we will assume that all representations in TNF or INF are both syntactically and semantically correct. This assumption cannot be made in the case of terms in LNF. We will need specific procedures to check the validity of a given LNF term or to convert a given term into some canonical equivalent one. For instance, we shall be able to cope with mistakenly written terms such as $\mathsf{Prod}(Z)))$ or $\mathsf{Prod}(\mathsf{Prod}(Z_1, Z_5), Z_2)$, or be able to convert the term $\mathsf{Set}(Z_2, \mathsf{Prod}(Z_1, Z_3))$ to $\mathsf{Set}(\mathsf{Prod}(Z_1, Z_3), Z_2)$, rearranging the components in increasing order of the minimum label attached to some of the component's atoms.

The rest of this chapter is organized as follows. First, we define formally LNF and TNF. Afterwards, in Sections 9.2 and 9.3 we detail the conversions between LNF and TNF, and between LNF and INF, respectively.

We describe now the grammars for terms in LNF and terms in TNF using Backus-Naur Form [40, 66]. The LNF is the usual way to describe an object in combstruct [19, 26] and it has been also regularly used in the literature (see, for instance [11, 32, 87]).

The grammar for LNF terms representing labelled objects is:

$$
\begin{aligned}
\texttt{Epsilon}{::=}\ & \epsilon \mid \epsilon' \mid \epsilon'' \mid \ldots \\
\texttt{Atom}{::=}\ & \mathbb{N} \\
\texttt{LNF}{::=}\ & \texttt{Epsilon} \mid \texttt{Atom} \mid \\
& \mathsf{Prod}(\texttt{LNF}, \texttt{LNF}) \mid \texttt{LabCont}(\texttt{LNF}[, \texttt{LNF}]^*) \mid \\
& \mathsf{Subst}(\texttt{LNF}, \mathsf{Set}(\texttt{LNF}[, \texttt{LNF}]^*)) \\
\texttt{LabCont}{::=}\ & \mathsf{Seq} \mid \mathsf{Set} \mid \mathsf{Cycle}
\end{aligned}
$$

where we use the typewriter font to denote nonterminal syntatic categories (e.g., $\texttt{LNF}$), $[\mathcal{X}]^*$ denotes zero or more repetitions of $\mathcal{X}$, and $\mathcal{X} \mid \mathcal{Y}$ means $\mathcal{X}$ or $\mathcal{Y}$, as usual. For unlabelled classes we have a similar definition:

$$
\begin{aligned}
\texttt{Epsilon}{::=}\ & \epsilon \mid \epsilon' \mid \epsilon'' \mid \ldots \\
\texttt{Atom}{::=}\ & Z \mid Z' \mid a \mid b \mid \ldots \\
\texttt{LNF}{::=}\ & \texttt{Epsilon} \mid \texttt{Atom} \mid \\
& \mathsf{Prod}(\texttt{LNF}, \texttt{LNF}) \mid \texttt{UnlCont}(\texttt{LNF}[, \texttt{LNF}]^*) \mid \\
& \mathsf{Subst}(\texttt{LNF}, \mathsf{Seq}(\texttt{LNF}[, \texttt{LNF}]^*)) \\
\texttt{UnlCont}{::=}\ & \mathsf{Seq} \mid \mathsf{PSet} \mid \mathsf{Set} \mid \mathsf{Cycle}
\end{aligned}
$$

We assume that $\texttt{Epsilon} \cap \texttt{Atom} = \emptyset$. We also assume that no component of a sequence, powerset, set or substitution represents an object of size 0, that the size of the left component of a substitution is equal to the cardinality of its right component, and that the atoms of any term in LNF representing a labelled object is a permutation of $\mathcal{S}_n$, where n is the number of atoms in the term. All these assumptions can be easily checked at the syntactical level.

Now we formally describe the tuple normal form (TNF) introduced in Part II. For labelled classes we have:

$$
\texttt{Epsilon}{::=}\ \epsilon \mid \epsilon' \mid \epsilon'' \mid \ldots
$$

$$\begin{aligned}
\texttt{Atom}{::=}\ & \mathbb{N}\\
\texttt{rank}{::=}\ & \mathbb{N}\\
\texttt{TNF}{::=}\ & \texttt{Epsilon} \mid \texttt{Atom} \mid \langle \texttt{TNF}, \texttt{TNF}, \texttt{rank}\rangle \mid \\
& {}^{\Box}\langle \texttt{TNF}, \texttt{TNF}, \texttt{rank}\rangle \mid {}^{\circ}\langle \texttt{TNF}, \texttt{TNF}\rangle
\end{aligned}$$

and for unlabelled classes we have:

$$\begin{aligned}
\texttt{Epsilon}{::=}\ & \epsilon \mid \epsilon' \mid \epsilon'' \mid \dots\\
\texttt{Atom}{::=}\ & Z \mid Z' \mid a \mid b \mid \dots\\
\texttt{TNF}{::=}\ & \texttt{Epsilon} \mid \texttt{Atom} \mid \langle \texttt{TNF}, \texttt{TNF}\rangle \mid {}^{\circ}\langle \texttt{TNF}, \texttt{TNF}\rangle
\end{aligned}$$

Here, we make similar assumptions as we did for terms in LNF. In particular, we assume that $\texttt{Epsilon} \cap \texttt{Atom} = \emptyset$, that the size of the left component of a substitution is equal to the cardinality of its right component, and that no component of a sequence, set, powerset or substitution represents an object of size 0. Moreover, for a tuple $\langle t_1, t_2, r\rangle$ we also suppose that the rank r is a natural number that satisfies $0 \le r < \binom{n}{j}$, where n is the size of the object and j is the size of the second component[1]. A similar condition holds for r in a tuple of the form ${}^{\Box}\langle t_1, t_2, r\rangle$ but now satisfying $0 \le r < \binom{n-1}{j-1}$.

Observe that it is quite easy to define the function size ($|\cdot|$, for short) on the terms generated by any of the two grammars defined above. For instance, if t is a labelled object given in TNF, then

$$|t| = \begin{cases} 0 & \text{if } t \in \texttt{Epsilon},\\ 1 & \text{if } t \in \texttt{Atom},\\ |t_1| + |t_2| & \text{if } t = \langle t_1, t_2, r\rangle \text{ or } t = {}^{\Box}\langle t_1, t_2, r\rangle,\\ |t_2| & \text{if } t = {}^{\circ}\langle t_1, t_2\rangle. \end{cases}$$

Finally, the internal normal form (INF) was described in Part III; it is a tree structure where each node has abundant additional information (fields like class, size, right, left, ...). We refer the reader to Chapters 7 and 8 for the detailed description of INF.

9.2 Conversions LNF $\leftrightarrow$ TNF

To convert a term in LNF into its corresponding TNF we use the function *L2T*. This function just takes into account the definitions given above. For labelled objects we have:

1. If $t \in \texttt{Epsilon}$ or $t \in \texttt{Atom}$, then $L2T(t) = t$.
2. If $t = \mathsf{Prod}(t_1, t_2)$, then $L2T(t) = \langle L2T(t_1), L2T(t_2), \mathit{rank\_lab}(t_1, t_2)\rangle$.

   The function $\mathit{rank\_lab}(t_1, t_2)$ gives the rank of the labelling partition of the object $\mathsf{Prod}(t_1, t_2)$.
3. If $t = \mathsf{Seq}(t_1)$, then $L2T(t) = \langle L2T(t_1), \epsilon, 0\rangle$.

[1]From now on, we will only consider the so-called *well-formed TNF* terms that result when we apply the algorithms of Part II. However, if there is no confusion we will just call them TNF terms.

4. If $t = \mathsf{X}(t_1)$ where $\mathsf{X} \in \{\mathsf{Set}, \mathsf{Cycle}\}$, then $\mathit{L2T}(t) = {}^{\Box}\langle \mathit{L2T}(t_1), \epsilon, 0\rangle$.
5. If $t = \mathsf{Seq}(t_1, t_2, \ldots, t_k)$ where $k > 1$, then
   $\mathit{L2T}(t) = \langle \mathit{L2T}(\mathit{head}(t)), \mathit{L2T}(\mathit{tail}(t)), \mathit{rank\_lab}(\mathit{head}(t), \mathit{tail}(t))\rangle$.
   The auxiliary functions *head* and *tail* are defined below. Intuitively, the *head* is the first component of a non-empty sequence or the first block of a powerset or set, and the *tail* is a container with the remaining components.
6. If $t = \mathsf{X}(t_1, t_2, \ldots, t_k)$ where $\mathsf{X} \in \{\mathsf{Set}, \mathsf{Cycle}\}$ and $k > 1$, then
   $\mathit{L2T}(t) = {}^{\Box}\langle \mathit{L2T}(\mathit{head}(t)), \mathit{L2T}(\mathit{tail}(t)), \mathit{rank\_lab}(\mathit{head}(t), \mathit{tail}(t))\rangle$.
7. If $t = \mathsf{Subst}(t_1, t_2)$, then $\mathit{L2T}(t) = {}^{\circ}\langle \mathit{L2T}(t_1), \mathit{L2T}(t_2)\rangle$.

For unlabelled objects we have:

1. If $t \in$ `Epsilon` or $t \in$ `Atom`, then $\mathit{L2T}(t) = t$.
2. If $t = \mathsf{Prod}(t_1, t_2)$, then $\mathit{L2T}(t) = \langle \mathit{L2T}(t_1), \mathit{L2T}(t_2)\rangle$.
3. If $t = \mathsf{X}(t_1)$ where $\mathsf{X} \in \{\mathsf{Seq}, \mathsf{PSet}, \mathsf{Set}\}$[2], then $\mathit{L2T}(t) = \langle \mathit{L2T}(t_1), \epsilon\rangle$.
4. If $t = \mathsf{X}(t_1, t_2, \ldots, t_k)$ where $\mathsf{X} \in \{\mathsf{Seq}, \mathsf{PSet}, \mathsf{Set}\}$ and $k > 1$, then
   $\mathit{L2T}(t) = \langle \mathit{L2T}(\mathit{head}(t)), \mathit{L2T}(\mathit{tail}(t))\rangle$.
   The functions *head* and *tail* are similar to those for labelled objects.
5. If $t = \mathsf{Subst}(t_1, t_2)$, then $\mathit{L2T}(t) = {}^{\circ}\langle \mathit{L2T}(t_1), \mathit{L2T}(t_2)\rangle$.

The rules for the *head* and *tail* for labelled objects are as follows:

1. If $t = \mathsf{X}(t_1, t_2, \ldots, t_k)$ where $\mathsf{X} \in \{\mathsf{Seq}, \mathsf{Set}\}$ and $k > 1$, then
   $\mathit{head}(t) = t_1$ and $\mathit{tail}(t) = \mathsf{X}(t_2, \ldots, t_k)$.
2. If $t = \mathsf{Cycle}(t_1, t_2, \ldots, t_k)$ where $k > 1$, then $\mathit{head}(t) = t_1$ and $\mathit{tail}(t) = \mathsf{Seq}(t_2, \ldots, t_k)$.

The rules for unlabelled objects are similar:

1. If $t = \mathsf{Seq}(t_1, t_2, \ldots, t_k)$ where $k > 1$, then $\mathit{head}(t) = t_1$ and $\mathit{tail}(t) = \mathsf{Seq}(t_2, \ldots, t_k)$.
2. If $t = \mathsf{X}(t_1, t_2, \ldots, t_k)$ where $\mathsf{X} \in \{\mathsf{PSet}, \mathsf{Set}\}$, $k > 1$, and there exists i such that $i = \min\{1 \le j < k; |t_j| < |t_{j+1}|\}$, then $\mathit{head}(t) = \mathsf{X}(t_1, \ldots, t_i)$ and $\mathit{tail}(t) = \mathsf{X}(t_{i+1}, \ldots, t_k)$.
3. If $t = \mathsf{X}(t_1, t_2, \ldots, t_k)$ where $\mathsf{X} \in \{\mathsf{PSet}, \mathsf{Set}\}$, $k > 1$, $|t_1| = |t_2| = \ldots = |t_k|$ and there exists i such that $i = \min\{1 \le j < k; t_j \prec t_{j+1}\}$, then $\mathit{head}(t) = \mathsf{X}(t_1, \ldots, t_i)$ and $\mathit{tail}(t) = \mathsf{X}(t_{i+1}, \ldots, t_k)$.
4. If $t = \mathsf{Set}(t_1, t_2, \ldots, t_k)$ where $k > 1$ and $t_1 = t_2 = \ldots = t_k$, then $\mathit{head}(t) = t_1$ and $\mathit{tail}(t) = \mathsf{Set}(t_2, \ldots, t_k)$.

[2] We will not ever use the TNF of an unlabelled cycle. Hence, we give no rule for the corresponding conversion.

For labelled objects, the algorithm to convert sets or cycles from LNF to TNF assumes the components are organized in the order given by the boxed product isomorphism of Section 2.2. If not, we have to do a preprocess to get the components in that way. For instance, $\mathsf{Set}(\mathsf{Prod}(Z_3, Z_4), \mathsf{Prod}(Z_5, Z_2), Z_1)$ should be given as $\mathsf{Set}(Z_1, \mathsf{Prod}(Z_5, Z_2), \mathsf{Prod}(Z_3, Z_4))$.

Furthermore, we use the additional function *rank_lab* to compute the rank of the labelling corresponding to a (partitional or boxed) product of two labelled objects. Function *rank_lab* returns the rank of the labelling partition corresponding to the given (labelled) components. Recall that for two labelled objects α and β of size j and $n-j$ respectively, $\alpha \star \beta$ is the set of $\binom{n}{j}$ labelled objects that we obtain by relabelling the atoms of the pair (α, β) according to each of the (n, j)-partitions (Subsection 2.1). Given two labelled objects γ_1 and γ_2 in $\alpha \star \beta$, recall also that $\gamma_1 \prec \gamma_2$ if and only if the set of labels of the first component of γ_1 in ascending order is lexicographically smaller than the set of labels of the first component of γ_2 also in ascending order. To determine the rank of an object γ within $\alpha \star \beta$ given its LNF $t_\gamma = \mathsf{Prod}(t_\alpha, t_\beta)$[3], we need thus to consider the set of labels of the first component of t_γ, i.e. the labels that appear in the term t_α, in ascending order and apply any ranking algorithm for $|t_\alpha|$-sets (e.g., see [46] or Algorithm 51). Notice that Algorithm 51 works on the basis that the first $\binom{|t_\alpha|+|t_\beta|-1}{|t_\alpha|-1}$ objects in $\alpha \star \beta$ have its smallest label in the first component. Thus, *rank_lab* can also be used to deal with boxed products, $\alpha\, {}^{\square}\!\star\, \beta$.

Algorithm 51 Rank of $\mathsf{Prod}(t_\alpha, t_\beta)$ in $\alpha \star \beta$

rank_lab(t_α, t_β)
  $S := \{$Array of labels of t_α in ascending order$\}$
  $n := |t_\alpha| + |t_\beta|; j := |t_\alpha|; r := 0; S[0] := 0$
  for i **from** 1 **to** $|S|$ **do**
    if $S[i-1]+1 \le S[i]-1$ **then**
      for k **from** $S[i-1]+1$ **to** $S[i]-1$ **do** $r := r + \binom{n-k}{j-i}$ **end**
    fi
  end
  return r

Finally, for unlabelled objects, the algorithm to convert powersets or sets from LNF to TNF assumes the components are organized in the LScR order of Section 2.3. That is, components are in increasing order and grouped by sizes and ranks; if not, we should reorganize the components in that way. For instance, given $\mathsf{Set}(\alpha, \beta, \gamma, \beta, \alpha, \alpha)$ where $\alpha \prec \beta$ and $|\alpha| = |\beta| < |\gamma|$, a preprocessing step should reorganize its components to the "canonical form" $\mathsf{Set}(\alpha, \alpha, \alpha, \beta, \beta, \gamma)$. Afterwards, we can apply the *L2T* transformation.

We describe now how to transform a term in TNF into its corresponding LNF using the function *T2L*. Here it is necessary to use the function is_a$(\mathcal{C}, t)$ (see Section 6.4) to check if a given object t belongs to a given class $\mathcal{C}$ or not, and, afterwards, to apply recursively

[3] This also applies to terms of the form $t_\gamma = \mathsf{Seq}(t_\alpha, t_\beta)$, $t_\gamma = \mathsf{Set}(t_\alpha, t_\beta)$ or $t_\gamma = \mathsf{Cycle}(t_\alpha, t_\beta)$.

the *T2L* transformation. For labelled classes, given a labelled object t and a labelled class $\mathcal{C}$, we define $T2L(\mathcal{C}, t)$ as follows:

1. If ($t \in$ `Epsilon` and $\mathcal{C} = \{\epsilon\}$) or ($t \in$ `Atom` and $\mathcal{C} = \{Z\}$), then $T2L(\mathcal{C}, t) = t$.
2. If $t = \langle t_1, t_2, r\rangle$ and $\mathcal{C} = \mathcal{A}+\mathcal{B}$, then $T2L(\mathcal{C}, t) = T2L(\mathcal{A}, t)$ if is_a$(\mathcal{A}, t)$ and $T2L(\mathcal{C}, t) = T2L(\mathcal{B}, t)$ if is_a$(\mathcal{B}, t)$; exactly one of the two cases must hold if t is a valid representation of an object in $\mathcal{C}$ (we assume this to be the case).
3. If $t = \langle t_1, t_2, r\rangle$ and $\mathcal{C} = \mathcal{A} \star \mathcal{B}$, then

   $T2L(\mathcal{C}, t) = put\_lab(\mathsf{Prod}(T2L(\mathcal{A}, t_1), T2L(\mathcal{B}, t_2), r))$.

   The function $put\_lab(\mathsf{X}(\alpha, \beta, r))$ computes the labelling of the r-th object in $\alpha \star \beta$ if $\mathsf{X} \in \{\mathsf{Prod}, \mathsf{Seq}\}$, or the labelling of the r-th object in $\alpha \,{}^{\Box}\!\!\star\, \beta$ if $\mathsf{X} \in \{\mathsf{Set}, \mathsf{Cycle}\}$ (e.g., see any of the algorithms in [46], see also in [70, 67]), and attaches the labels to the atoms. For instance, $put\_lab(\mathsf{Prod}(\mathsf{Prod}(Z_1, Z_2), \mathsf{Prod}(Z_2, Z_1)), 2)$ is transformed into $\mathsf{Prod}(\mathsf{Prod}(Z_3, Z_4), \mathsf{Prod}(Z_2, Z_1))$.
4. If $t = \langle t_1, t_2, r\rangle$ and $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$, or $t = {}^{\Box}\langle t_1, t_2, r\rangle$ and $\mathcal{C} = \mathsf{Set}(\mathcal{A})$ or $\mathcal{C} = \mathsf{Cycle}(\mathcal{A})$, then $T2L(\mathcal{C}, t) = clean(aT2L(\mathcal{C}, t))$.

   Intuitively the function *clean* removes the possible trailing `Epsilon` which "marks" the end of a container and it also transforms the "binary" form of the operators Seq, Set, ... into m-ary form. For example, $clean(\mathsf{Seq}(Z_2, \mathsf{Seq}(Z_1, \epsilon))) = \mathsf{Seq}(Z_2, Z_1)$. The detailed behavior of $clean(\cdot)$ is given later.

   Here the function *aT2L* is defined depending on its components,

$$aT2L(\mathcal{C}, t) = \begin{cases} put\_lab(\mathsf{Seq}(aT2L(\mathcal{A}, t_1), aT2L(\mathsf{Seq}(\mathcal{A}), t_2), r)) \\ \qquad\qquad \text{if } t = \langle t_1, t_2, r\rangle \text{ and } \mathcal{C} = \mathsf{Seq}(\mathcal{A}), \\ put\_lab(\mathsf{Set}(aT2L(\mathcal{A}, t_1), aT2L(\mathsf{Set}(\mathcal{A}), t_2), r)) \\ \qquad\qquad \text{if } t = {}^{\Box}\langle t_1, t_2, r\rangle \text{ and } \mathcal{C} = \mathsf{Set}(\mathcal{A}), \\ put\_lab(\mathsf{Cycle}(aT2L(\mathcal{A}, t_1), aT2L(\mathsf{Seq}(\mathcal{A}), t_2), r)) \\ \qquad\qquad \text{if } t = {}^{\Box}\langle t_1, t_2, r\rangle \text{ and } \mathcal{C} = \mathsf{Cycle}(\mathcal{A}). \end{cases}$$

5. If $t = {}^{\circ}\langle t_1, t_2\rangle$ and $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$, then $T2L(\mathcal{C}, t) = \mathsf{Subst}(T2L(\mathcal{A}, t_1), T2L(\mathsf{Set}(\mathcal{B}), t_2))$

In a similar way, for unlabelled objects we define $T2L(\mathcal{C}, t)$ as follows:

1. If ($t \in$ `Epsilon` and $\mathcal{C} = \{\epsilon\}$) or ($t \in$ `Atom` and $\mathcal{C} = \{Z\}$), then $T2L(\mathcal{C}, t) = t$.
2. If $t = \langle t_1, t_2\rangle$ and $\mathcal{C} = \mathcal{A} + \mathcal{B}$, then $T2L(\mathcal{C}, t) = T2L(\mathcal{A}, t)$ if is_a$(\mathcal{A}, t)$ and $T2L(\mathcal{C}, t) = T2L(\mathcal{B}, t)$ if is_a$(\mathcal{B}, t)$.
3. If $t = \langle t_1, t_2\rangle$ and $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, then $T2L(\mathcal{C}, t) = \mathsf{Prod}(T2L(\mathcal{A}, t_1), T2L(\mathcal{B}, t_2))$.
4. If $t = \langle t_1, t_2\rangle$ and $\mathcal{C} = \mathsf{X}(\mathcal{A})$ where $\mathsf{X} \in \{\mathsf{Seq}, \mathsf{PSet}, \mathsf{Set}\}$[4], then

   $T2L(\mathcal{C}, t) = clean(\mathsf{X}(T2L(\mathcal{A}, t_1), (T2L(\mathcal{C}, t_2))))$, where the function *clean* is the same as that for labelled objects.

[4] Again, we give no rule for unlabelled cycles.

5. If $t = \circ\langle t_1, t_2\rangle$ and $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$, then $T2L(\mathcal{C}, t) = \mathsf{Subst}(T2L(\mathcal{A}, t_1), T2L(\mathsf{Seq}(\mathcal{B}), t_2))$.

Now we define the function *clean* used above as follows:

$$clean(t) = \begin{cases} t & \text{if } t \in \texttt{Epsilon} \text{ or } t \in \texttt{Atom}, \\ \mathsf{Prod}(clean(t_1), clean(t_2)) & \text{if } t = \mathsf{Prod}(t_1, t_2), \\ \mathsf{X}(clean(t_1), aclean(t_2)) & \text{if } t = \mathsf{X}(t_1, t_2), \end{cases}$$

where $\mathsf{X} \in \{\mathsf{Seq}, \mathsf{PSet}, \mathsf{Set}, \mathsf{Cycle}\}$[5] and

$$aclean(t) = \begin{cases} t & \text{if } t \in \texttt{Epsilon} \text{ or } t \in \texttt{Atom}, \\ t_1 & \text{if } t = \mathsf{X}(t_1, \texttt{Epsilon}), \\ t_1, aclean(t_2) & \text{if } t = \mathsf{X}(t_1, t_2) \text{ and } t_2 \notin \texttt{Epsilon}. \end{cases}$$

For instance, consider the labelled object in $\mathcal{C} = \mathsf{Seq}(Z)$ represented by the TNF term $t = \langle Z, \langle Z, \langle Z, \langle Z, \langle Z, \epsilon, 0\rangle, 0\rangle, 0\rangle, 1\rangle, 3\rangle$, then t is transformed into its corresponding LNF as follows:

$$\begin{aligned} T2L(\mathcal{C}, t) &= clean(aT2L(\mathcal{C}, t)) = \dots \\ &= clean(put\_lab(\mathsf{Seq}(Z_1, \mathsf{Seq}(Z_2, \mathsf{Seq}(Z_1, \mathsf{Seq}(Z_3, \mathsf{Seq}(Z_4, \epsilon)))), 3))) \\ &= clean(\mathsf{Seq}(Z_4, \mathsf{Seq}(Z_2, \mathsf{Seq}(Z_1, \mathsf{Seq}(Z_3, \mathsf{Seq}(Z_5, \epsilon)))))) \\ &= \mathsf{Seq}(Z_4, Z_2, Z_1, Z_3, Z_5)\ . \end{aligned}$$

9.3 Conversions LNF $\leftrightarrow$ INF

Now we are going to study how to convert a LNF term into its corresponding INF. As we have described in Part III, the INF representation or iterator consists of several global attributes of the represented object, among others, spec_equations, class and root (and perm and inv_perm for labelled classes). The root attribute points to the tree representation of the object. Each node of this tree structure has the attributes class, size, count, rank, left and right (and ind_perm, part, inv_part, count_part and rank_part for labelled classes).

Given a term t in LNF and the class $\mathcal{C}$ where t belongs, we can generate its corresponding INF considering the same cases as in the function *L2T*, but INF is a richer representation.

For labelled terms, once we have created an empty iterator *itr* (see Algorithm 52), and following the same cases as in the transformation *L2T* in Section 9.2, we generate its corresponding tree structure using the auxiliary function *aL2I* (see Algorithm 53). The additional parameter of *aL2I* corresponds to the index attribute called ind_perm; i.e., the index of the first label of the generated tree in the global attribute *itr*.perm.

The function *aL2I* for labelled classes is detailed in Algorithm 53, where $\mathsf{X} \in \{\mathsf{Seq}, \mathsf{Set}\}$. We use the function *Canonical*(t) which relabels the atoms from 1 to $|t|$ and preserves the induced partition between the two components of t. To cope with terms containing sets or cycles we perform an intermediate step using an "internal" operator BProd, inspired by the isomorphisms of Chapter 2. This yields cleaner and simpler code.

For unlabelled terms, to convert an unlabelled term in LNF into its corresponding INF, we do not have to consider the partition attributes. Thus, once we have created an empty

[5] We assume that Cycle is only used for labelled classes.

Algorithm 52 *L2I* converts a labelled LNF into its corresponding INF

L2I($\mathcal{C}, t$, spec_equations)
  Iterator itr ▷ Return a fresh, uninitialized iterator ◁
  itr.class := $\mathcal{C}$; itr.spec_equations := spec_equations
  itr.root := aL2I($\mathcal{C}, t, 1$, itr)

Algorithm 53 *aL2I* converts a labelled LNF into its corresponding INF

aL2I($\mathcal{C}, t, ip$, itr)
  p := New_Node ▷ Return a fresh, uninitialized node ◁
  $p\to$class := $\mathcal{C}$; $p\to$size := $|t|$; $p\to$count := Count($\mathcal{C}, p\to$size)
  if $n > 0$ **then** $p\to$ind_perm := ip **fi**
  if $t \in$ `Atom` **and** $\mathcal{C}=\{Z\}$ **then** itr.perm[ip]:=ip; itr.inv_perm[ip]:=ip **fi**
  ▷ If $(n = 0)$ or $(\mathcal{C} = \mathcal{A} + \mathcal{B})$ or $(n = 1$ and $\mathcal{C} = Z)$, then fields ◁
  ▷   part, inv_part, count_part and rank_part are meaningless ◁
  if ($t \in$ `Epsilon` **and** ($\mathcal{C} = \{\epsilon\}$ **or** $\mathcal{C}$ = Seq($\mathcal{A}$) **or** $\mathcal{C}$ = Set($\mathcal{A}$))) **or** ($t \in$ `Atom`) **then**
    $p\to$left := NULL; $p\to$right := NULL; $p\to$rank := 0
  elif $\mathcal{C} = \mathcal{A} + \mathcal{B}$ **then**
    if is_a($\mathcal{A}$, L2T(t)) **then**
      $p\to$left := aL2I($\mathcal{A}, t, ip$, itr); $p\to$right := NULL; $p\to$rank := $p\to$left$\to$rank
    else $p\to$left := NULL; $p\to$right := aL2I($\mathcal{B}, t, ip$, itr)
      $p\to$rank := Count($\mathcal{A}, p\to$size) + $p\to$right$\to$rank
    fi
  elif (t = Prod(t_1, t_2) **and** $\mathcal{C} = \mathcal{A} \star \mathcal{B}$) **or** ($t$ = BProd(t_1, t_2) **and** $\mathcal{C} = \mathcal{A}\,^{\Box}\!\star\, \mathcal{B}$) **then**
    $p\to$part := Canonical(t)
    for i **from** 1 **to** $p\to$size **do** $p\to$inv_part[$p\to$part[i]] := i **end**
    $p\to$left := aL2I($\mathcal{A}, t_1, ip$, itr); $p\to$right := aL2I($\mathcal{B}, t_2, ip + p\to$left$\to$size, itr)
    $p\to$rank_part := rank_lab(Canonical(t_1), Canonical(t_2))
    if $\mathcal{C} = \mathcal{A} \star \mathcal{B}$ **then** $p\to$count_part := $\binom{p\to\text{size}}{p\to\text{left}\to\text{size}}$
    else $\mathcal{C} = \mathcal{A}\,^{\Box}\!\star\, \mathcal{B}$ **then** $p\to$count_part := $\binom{p\to\text{size}-1}{p\to\text{left}\to\text{size}-1}$
    fi
    $p\to$rank := ... ▷ Compute the rank given the children' ranks ◁
  elif t = Seq($t_1, t_2, \ldots, t_k$) **and** $\mathcal{C}$ = Seq($\mathcal{A}$) **then**
    p := aL2I($\mathcal{A} \star$ Seq($\mathcal{A}$), Prod(head(t), tail(t)), ip, itr)
  elif t = Set($t_1, t_2, \ldots, t_k$) **and** $\mathcal{C}$ = Set($\mathcal{A}$) **then**
    p := aL2I($\mathcal{A}\,^{\Box}\!\star$ Set($\mathcal{A}$), BProd(head(t), tail(t)), ip, itr)
  elif t = Cycle($t_1, t_2, \ldots, t_k$) **and** $\mathcal{C}$ = Cycle($\mathcal{A}$) **then**
    p := aL2I($\mathcal{A}\,^{\Box}\!\star$ Seq($\mathcal{A}$), BProd(head(t), tail(t)), ip, itr)
  elif t = Subst(t_1, t_2) **and** $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$ **then** ▷ count_part and rank_part are meaningless ◁
    $p\to$left := aL2I($\mathcal{A}, t_1, ip$, itr); $p\to$right := aL2I(Set($\mathcal{B}$), t_2, ip, itr)
  fi

iterator *itr* as in Algorithm 52, we have to change line (4) by "*itr*.root := *aL2I*($\mathcal{C}$, *itr*)" and use Algorithm 54 instead.

Algorithm 54 *aL2I* converts an unlabelled LNF into its corresponding INF

---

aL2I($\mathcal{C}, t$)
    $p := \mathsf{New\_Node}$ ▷ Return a fresh, uninitialized node ◁
    $p \to \mathsf{class} := \mathcal{C}; p \to \mathsf{size} := |t|; p \to \mathsf{count} := \mathrm{Count}(\mathcal{C}, p \to \mathsf{size})$
    $p \to \mathsf{rank} := \ldots$ ▷ Compute the rank given the children' ranks ◁
    if ($t \in$ `Epsilon` **and** ($\mathcal{C} = \{\epsilon\}$ **or** $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{PSet}(\mathcal{A})$ **or** $\mathcal{C} = \mathsf{Set}(\mathcal{A})$)) **or**
      ($t \in$ `Atom`) **then** $p \to \mathsf{left} := \mathsf{NULL}; p \to \mathsf{right} := \mathsf{NULL}$
    elif $\mathcal{C} = \mathcal{A} + \mathcal{B}$ **then**
        if is_a($\mathcal{A}$, L2T(t)) **then** $p \to \mathsf{left} := \mathrm{aL2I}(\mathcal{A}, t); p \to \mathsf{right} := \mathsf{NULL}$
        else $p \to \mathsf{left} := \mathsf{NULL}; p \to \mathsf{right} := \mathrm{aL2I}(\mathcal{B}, t)$
        fi
    elif $t = \mathsf{Prod}(t_1, t_2)$ **and** $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ **then**
        $p \to \mathsf{left} := \mathrm{aL2I}(\mathcal{A}, t_1); p \to \mathsf{right} := \mathrm{aL2I}(\mathcal{B}, t_2)$
    elif $t = \mathsf{Seq}(t_1, t_2, \ldots, t_k)$ **and** $\mathcal{C} = \mathsf{Seq}(\mathcal{A})$ **then**
        $p \to \mathsf{left} := \mathrm{aL2I}(\mathcal{A}, \mathrm{head}(t)); p \to \mathsf{right} := \mathrm{aL2I}(\mathcal{C}, \mathrm{tail}(t))$
    elif $t = \mathsf{PSet}(t_1, t_2, \ldots, t_k)$ **and** $\mathcal{C} = \mathsf{PSet}(\mathcal{A})$ **then**
        if $\exists i = \min\{1 \le j < k;\ |t_j| < |t_{j+1}|\}$ **then**
            $p \to \mathsf{left} := \mathrm{aL2IPSetBlock}(\mathcal{A}, j, |t_1|, 0, \mathsf{PSet}(t_1, \ldots, t_i))$
            $p \to \mathsf{right} := \mathrm{aL2IPSet}(\mathcal{A}, |t| - j \cdot |t_1|, |t_1| + 1, \mathsf{PSet}(t_{i+1}, \ldots, t_k))$
        else $p \to \mathsf{left} := \mathrm{aL2IPSetBlock}(\mathcal{A}, k, |t_1|, 0, t); p \to \mathsf{right} := \mathsf{NULL}$
        fi
    elif $t = \mathsf{Set}(t_1, t_2, \ldots, t_k)$ **and** $\mathcal{C} = \mathsf{Set}(\mathcal{A})$ **then**
        if $\exists i = \min\{1 \le j < k;\ |t_j| < |t_{j+1}|\}$ **then**
            $p \to \mathsf{left} := \mathrm{aL2ISetBlock}(\mathcal{A}, j, |t_1|, 0, \mathsf{Set}(t_1, \ldots, t_i))$
            $p \to \mathsf{right} := \mathrm{aL2ISet}(\mathcal{A}, |t| - j \cdot |t_1|, |t_1| + 1, \mathsf{Set}(t_{i+1}, \ldots, t_k))$
        else $p \to \mathsf{left} := \mathrm{aL2ISetBlock}(\mathcal{A}, k, |t_1|, 0, t); p \to \mathsf{right} := \mathsf{NULL}$
        fi
    elif $t = \mathsf{Subst}(t_1, t_2)$ **and** $\mathcal{C} = \mathsf{Subst}(\mathcal{A}, \mathcal{B})$ **then**
        $p \to \mathsf{left} := \mathrm{aL2I}(\mathcal{A}, t_1); p \to \mathsf{right} := \mathrm{aL2I}(\mathsf{Seq}(\mathcal{B}), t_2)$
    fi

The function *aL2IPSetBlock*($\mathcal{A}, k, j, i, t$) produces the INF of the object represented by t in $\mathsf{PSet}(\mathcal{A})$ with k components of size j and rank not smaller than i (see Algorithm 55). On the other hand, the function *aL2IPSet*($\mathcal{A}, n, j, t$) corresponds to the object t in $\mathsf{PSet}_n(\mathcal{A})$ with components of size not smaller than j (see Algorithm 56). Finally, in a similar way we define *aL2ISetBlock* and *aL2ISet*, but allowing repetitions among the components and using the attribute occurrences as we did in Subsection 8.1.2.

On the other hand, we also can consider the variant that uses a finger pointer finger (see Section 7.3). That is, each node has a pointer to the last right node that can be modified by the application of the function *Next_Obj*. For unlabelled classes and for labelled classes in SF order, we showed in Algorithm 40 how to initialize these finger pointers. The same

Algorithm 55 *aL2IPSetBlock* converts an unlabelled $\mathsf{PSet}(\mathcal{A})$ of k-components of size j of rank not smaller than i given LNF into its corresponding INF

aL2IPSetBlock$(\mathcal{A}, k, j, i, t)$
    $p := \mathsf{New\_Node}$ ▷ Return a fresh, uninitialized node ◁
    $p \rightarrow \mathsf{class} := \mathcal{C}; p \rightarrow \mathsf{size} := |t|; p \rightarrow \mathsf{count} := \binom{\mathrm{Count}(\mathcal{A},j)-(i+1)}{k}$
    $p \rightarrow \mathsf{rank} := \mathrm{RankPSetBlock}(\mathcal{A}, j, k, i, \mathrm{L2T}(t))$
    $p \rightarrow \mathsf{left} := \mathrm{aL2I}(\mathcal{A}, t_1)$
    $p \rightarrow \mathsf{right} := \mathrm{aL2IPSetBlock}(\mathcal{A}, k-1, j, \mathrm{Rank}(t_1, \mathcal{A}) + 1, \mathsf{PSet}(t_2, \ldots, t_k))$

Algorithm 56 *aL2IPSet* converts the LNF of an unlabelled $\mathsf{PSet}_n(\mathcal{A}_{>j})$ into its corresponding INF

aL2IPSet$(\mathcal{A}, n, j, t)$
    $p := \mathsf{New\_Node}$ ▷ Return a fresh, uninitialized node ◁
    $p \rightarrow \mathsf{class} := \mathcal{C}; p \rightarrow \mathsf{size} := |t|$
    $p \rightarrow \mathsf{count} := (\hat{\delta}_{n,1} - \hat{\delta}_{n,j+1}); p \rightarrow \mathsf{rank} := \mathrm{RankPSet}(\mathcal{A}, j, \mathrm{L2T}(t))$
    ▷ Lines (16)-(20) of Algorithm 54 ◁

techniques shall be applied when transforming a term in LNF into its corresponding INF if we want to equip it with finger pointers.

The reverse transformation, from INF to LNF, is trivially accomplished with a preorder traversal of the tree structure which represents the object.

Part V

Conclusions and Future Work

Throughout this thesis we have designed and analyzed algorithms for the ordered generation of combinatorial structures, in particular unranking, ranking and exhaustive generation.

We have proven that the analysis for our unranking and ranking algorithms follows the same steps as the analysis of the random generation algorithms presented by Flajolet *et al.* [26] and that it is possible to achieve equivalent complexities (both on the worst-case and on the average-case). We have also provided the analysis for the variance of unranking algorithms (which does also apply to random generation algorithms) and a calculus for the probability distribution of their costs, and for the r-th factorial moments. In particular, the worst-case complexity of unranking and ranking an object of a given class $\mathcal{A}$ and size n is $\mathrm{O}(n^2)$ for the lexicographic order and $\mathrm{O}(n \log n)$ for the boustrophedonic order. We have also studied variants and heuristics to improve the performance of unranking.

We have presented exhaustive generation algorithms with constant amortized time (CAT) per generated object under some mild assumptions on the generated class, namely when the combinatorial class is superexponential in the labelled case and superpolynomial in the unlabelled case. Moreover, we have shown that using "finger pointer" significantly improves the performance.

We have also considered how to transform back and forth from human-readable representations of combinatorial objects (LNF) to convenient internal representations (INF, TNF). These transformations allow for a seamless integration of all the generation algorithms: drawing, unranking, ranking and iteration.

We have implemented in MAPLE almost all algorithms described in this thesis, except a few cases of ranking. The algorithms work for all classes built using the admissible operators (Union, Prod, PSet, Set, etc.), except unlabelled cycles. We have provided, however, a CAT algorithm for the exhaustive generation of unlabelled cycles.

A new line of research would be to study generation algorithms for unlabelled powersets, sets and cycles based upon the isomorphisms using diagonal and generalized diagonal operators, namely "$\vartheta\mathsf{Set}(\mathcal{A}) = \Delta\vartheta\mathcal{A} \times \mathsf{Set}(\mathcal{A})$", "$\vartheta\mathsf{PSet}(\mathcal{A}) = \hat{\Delta}\vartheta\mathcal{A} \times \mathsf{PSet}(\mathcal{A})$" and "$\vartheta\mathsf{Cycle}(\mathcal{A}) = \Delta_{\{\phi(k)\}}(\vartheta\mathcal{A} \times \mathsf{Seq}(\mathcal{A}))$", where $\phi(k)$ denotes the Euler totient function of k, the number of positive integers not exceeding k and relatively prime to k. In fact, a sort of Δ-product should be introduced, which would play a rôle analogous to the boxed products that we have used for the unranking and exhaustive generation of labelled classes.

For the exhaustive generation, a few problems still remain open. For instance, it would be desirable to have a detailed analysis of the performance of the little-endian heuristic sketched in Section 7.3. It could also be interesting to conduct larger and more systematic experiments to study the practical performance of the algorithms developed there. It would be interesting to devise a suitable recursive decomposition of unlabelled cycles; lacking such recursive decomposition, we have come up with an efficient algorithm for exhaustive generation that departs largely in its design from the other algorithms, but still have no solution to the ranking/unranking problem.

Another line of research is to redesign the exhaustive generation algorithms so that there is no need to know the cardinality of the involved classes. Basically, the important issue is then to find an efficient and alternative way to compute *Is_Last* for any admissible combinatorial class.

Furthermore, it should also be interesting to provide general solutions for the generation problems (drawing, unranking, ranking and iteration) applied to classes which do not fit in the framework developed here, such as graphs.

Last but not least, we have planned to add these algorithms in the combstruct package of Maple, the MuPAD-combinat package of MuPAD and as a library for the C++ programming language.

Appendix A

Experimental Data of Unranking Algorithms

A.1 Tables

In this first section, we summarize the experimental data of unranking algorithms for the admissible classes of Table 5.1: (labelled and unlabelled) binary trees, (labelled and unlabelled) Motzkin trees, (labelled) functional graphs, (unlabelled) integer partitions, (labelled) surjections and (unlabelled) integer compositions. Here $\mu_{n,\mathcal{A}}$ and $\mu'_{n,\mathcal{A}}$ denote the average cost and the average number of arithmetic operations to unrank an object of size n in $\mathcal{A}$, respectively; $\overline{\mu}_{n,\mathcal{A}}$ and $\overline{\mu}'_{n,\mathcal{A}}$ denote the empirical estimates of $\mu_{n,\mathcal{A}}$ and $\mu'_{n,\mathcal{A}}$, respectively; whereas $\epsilon_{n,\mathcal{A}}$ and $\mathrm{AQE}_{\mathcal{A}}$ denote the relative error and the average quadratic error of the experimental data, respectively; $\overline{\tau}_{n,\mathcal{A}}$ denotes the empirical value of the CPU time (in seconds) to unrank an object of size n in $\mathcal{A}$; and finally, BF stands for *best fit curve*.

n	50	100	150	200	250
$\mu_{n,\mathcal{B}}$	285.972	832.899	1549.029	2401.925	3372.862
$\overline{\mu}_{n,\mathcal{B}}$	285.215	833.860	1548.209	2406.245	3353.153
$\overline{\mu}'_{n,\mathcal{B}}$	1149.645	3095.582	5538.629	8412.736	11553.461
$\epsilon_{n,\mathcal{B}}$	0.26 %	0.11 %	0.53 %	0.18 %	0.58 %
$\overline{\tau}_{n,\mathcal{B}}$	0.044886	0.131197	0.252442	0.421947	0.637975
$\mu_{n,\mathcal{B}}$	$0.8862\ldots\, n\sqrt{n} + 0.5\,n - 0.3323\ldots\,\sqrt{n} + 0.25 + o(1)$				
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.8738\ldots\, n\sqrt{n} - 0.3793\ldots\, n$				
$\mathrm{AQE}_{\mathcal{B}}$	$81.8534\ldots$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$2.1518\ldots\cdot 10^{-4}\, n\sqrt{n} - 8.8639\ldots\cdot 10^{-4}\, n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} \approx 3\,\overline{\mu}_{n,\mathcal{B}} + 6\,n - 6$				

Table A.1: Lexicographic labelled binary trees.

n	50	100	150	200	250
$\mu_{n,\mathcal{B}}$	120.420	276.792	446.317	624.367	808.750
$\overline{\mu}_{n,\mathcal{B}}$	121.232	279.204	443.912	629.353	804.103
$\overline{\mu}'_{n,\mathcal{B}}$	657.696	1431.612	2225.736	3082.059	3906.309
$\epsilon_{n,\mathcal{B}}$	0.67 %	0.87 %	0.54 %	0.80 %	0.57 %
$\overline{\tau}_{n,\mathcal{B}}$	0.033217	0.073667	0.116195	0.164338	0.218593
$\mu_{n,\mathcal{B}}$	$0.5\,n\log n + 0.4817\ldots\,n - 0.25\,\log n - 0.1204\ldots + \mathrm{o}(n)$				
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.4844\ldots\,n\log n + 0.5524\ldots\,n$				
$\mathrm{AQE}_{\mathcal{B}}$	$11.7433\ldots$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$0.3120\ldots\cdot 10^{-5}\,n\log n + 1.5116\ldots\cdot 10^{-4}\,n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} = 3\,\overline{\mu}_{n,\mathcal{B}} + 6\,n - 6$				

Table A.2: Lexicographic labelled binary trees using the boxed heuristic, $\mathcal{B} = Z + (\mathcal{B} + \mathcal{B}) \star^{\Box} \mathcal{B}$.

n	50	100	150	200	250
$\mu_{n,\mathcal{B}}$	285.972	832.899	1549.029	2401.925	3372.862
$\overline{\mu}_{n,\mathcal{B}}$	284.815	830.361	1528.593	2396.812	3386.994
$\overline{\mu}'_{n,\mathcal{B}}$	814.630	2155.722	3802.186	5788.624	8018.988
$\epsilon_{n,\mathcal{B}}$	0.40 %	0.30 %	1.32 %	0.21 %	0.42 %
$\overline{\tau}_{n,\mathcal{B}}$	0.039137	0.095819	0.165068	0.248248	0.334731
$\mu_{n,\mathcal{B}}$	$0.8862\ldots\,n\sqrt{n} + 0.5\,n - 0.3323\ldots\,\sqrt{n} + 0.25 + \mathrm{o}(1)$				
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.9123\ldots\,n\sqrt{n} - 0.9007\ldots\,n$				
$\mathrm{AQE}_{\mathcal{B}}$	$130.2508\ldots$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$6.5359\ldots\cdot 10^{-5}\,n\sqrt{n} + 3.0812\ldots\cdot 10^{-4}\,n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} = 2\,\overline{\mu}_{n,\mathcal{B}} + 5\,n - 5$				

Table A.3: Lexicographic unlabelled binary trees.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{B}}$	188.068	420.525	669.945	932.956	1200.360
$\overline{\mu}'_{n,\mathcal{B}}$	956.204	2053.575	3201.835	4390.868	5593.080
$\overline{\tau}_{n,\mathcal{B}}$	0.030374	0.067658	0.110360	0.155554	0.200628
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.6506\ldots\,n\log n + 1.2111\ldots\,n$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$1.3022\ldots\cdot 10^{-4}\,n\log n + 8.4503\ldots\cdot 10^{-5}\,n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} = 3\,\overline{\mu}_{n,\mathcal{B}} + 8\,n - 8$				

Table A.4: Boustrophedonic labelled binary trees.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{B}}$	72.250	147.050	222.140	297.500	371.210
$\overline{\mu}'_{n,\mathcal{B}}$	510.108	1034.901	1558.896	2084.040	2608.704
$\overline{\tau}_{n,\mathcal{B}}$	0.030654	0.065914	0.104733	0.149516	0.195042
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.0177\ldots\ n\log n + 1.389453491\,n$				
BF of $\overline{\mu}_{n,\mathcal{B}}$	$1.5016\ldots\ n - 0.6583\ldots\ \log n$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$1.2718\ldots\cdot 10^{-4}\,n\log n - 7.4267\ldots\cdot 10^{-5}\,n$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$8.5938\ldots\cdot 10^{-4}\,n - 4.0718\ldots\cdot 10^{-3}\ \log n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} \approx 3\,\overline{\mu}_{n,\mathcal{B}} + 6\,n - 6$				

Table A.5: Boustrophedonic labelled binary trees using the boxed heuristic, $\mathcal{B} = Z + (\mathcal{B} + \mathcal{B}) \star^{\square} \mathcal{B}$.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{B}}$	187.089	421.678	668.795	932.662	1202.282
$\overline{\mu}'_{n,\mathcal{B}}$	594.205	1288.281	2007.538	2760.011	3524.406
$\overline{\tau}_{n,\mathcal{B}}$	0.034508	0.074488	0.116927	0.158456	0.201269
BF of $\overline{\mu}_{n,\mathcal{B}}$	$0.6610\ldots\ n\log n + 1.1584\ldots\ n$				
BF of $\overline{\tau}_{n,\mathcal{B}}$	$6.4661\ldots\cdot 10^{-5}\,n\log n + 4.4947\ldots\cdot 10^{-4}\,n$				
BF of $\overline{\mu}'_{n,\mathcal{B}}$	$\overline{\mu}'_{n,\mathcal{B}} \approx 2\,\overline{\mu}_{n,\mathcal{B}} + 4.5\,n - 5$				

Table A.6: Boustrophedonic unlabelled binary trees.

n	50	100	150	200	250
$\mu_{n,\mathcal{M}}$	167.199	482.704	895.439	1386.898	1946.358
$\overline{\mu}_{n,\mathcal{M}}$	167.727	480.817	887.258	1375.210	1923.120
$\overline{\mu}'_{n,\mathcal{M}}$	846.181	2135.451	3704.774	5518.630	7512.360
$\epsilon_{n,\mathcal{M}}$	0.31 %	0.39 %	0.91 %	0.84 %	1.19 %
$\overline{\tau}_{n,\mathcal{M}}$	0.033307	0.090251	0.166150	0.267887	0.448058
$\mu_{n,\mathcal{M}}$	$0.5116\ldots\ n\sqrt{n} - 0.3333\ldots\ n - 0.1918\ldots\ n + 0.1666\ldots + \mathrm{o}(1)$				
BF of $\overline{\mu}_{n,\mathcal{M}}$	$0.4969\ldots\ n\sqrt{n} - 0.1621\ldots\ n$				
$\mathrm{AQE}_{\mathcal{M}}$	$149.4744\ldots$				
BF of $\overline{\tau}_{n,\mathcal{M}}$	$1.5740\ldots\cdot 10^{-4}\,n\sqrt{n} - 7.6565\ldots\cdot 10^{-4}\,n$				
BF of $\overline{\mu}'_{n,\mathcal{M}}$	$\overline{\mu}'_{n,\mathcal{M}} = 3\,\overline{\mu}_{n,\mathcal{M}} + 7\,n - 7$				

Table A.7: Lexicographic labelled Motzkin trees.

n	50	100	150	200	250
$\mu_{n,\mathcal{M}}$	167.199	482.704	895.439	1386.898	1946.358
$\overline{\mu}_{n,\mathcal{M}}$	167.032	481.179	891.513	1383.262	1977.649
$\overline{\mu}'_{n,\mathcal{M}}$	579.064	1457.358	2528.026	3761.524	5200.298
$\epsilon_{n,\mathcal{M}}$	0.10 %	0.32 %	0.44 %	0.26 %	1.61 %
$\overline{\tau}_{n,\mathcal{M}}$	0.033148	0.071102	0.111910	0.154252	0.195330
$\mu_{n,\mathcal{M}}$	$0.5116\ldots\ n\sqrt{n} - 0.3333\ldots\ n - 0.1918\ldots\ n + 0.1666\ldots + \mathrm{o}(1)$				
BF of $\overline{\mu}_{n,\mathcal{M}}$	$0.5356\ldots\ n\sqrt{n} - 0.5946\ldots\ n$				
$\mathrm{AQE}_{\mathcal{M}}$	$202.0279\ldots$				
BF of $\overline{\tau}_{n,\mathcal{M}}$	$1.1875\ldots\cdot 10^{-5}\, n\sqrt{n} + 5.9719\ldots\cdot 10^{-4}\, n$				
BF of $\overline{\mu}'_{n,\mathcal{M}}$	$\overline{\mu}'_{n,\mathcal{M}} = 2\,\overline{\mu}_{n,\mathcal{M}} + 5\,n - 5$				

Table A.8: Lexicographic unlabelled Motzkin trees.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{M}}$	157.917	358.855	575.015	801.364	1039.888
$\overline{\mu}'_{n,\mathcal{M}}$	865.751	1868.565	2917.045	3996.092	5111.664
$\overline{\tau}_{n,\mathcal{M}}$	0.028631	0.064164	0.103350	0.145266	0.197342
BF of $\overline{\mu}_{n,\mathcal{M}}$	$0.6251\ldots\ n\log n + 0.7029\ldots\ n$				
BF of $\overline{\tau}_{n,\mathcal{M}}$	$1.5703\ldots\cdot 10^{-4}\, n\log n - 8.8742\ldots\cdot 10^{-5}\, n$				
BF of $\overline{\mu}'_{n,\mathcal{M}}$	$\overline{\mu}'_{n,\mathcal{M}} = 3\,\overline{\mu}_{n,\mathcal{M}} + 8\,n - 8$				

Table A.9: Boustrophedonic labelled Motzkin trees.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{M}}$	156.424	357.601	573.128	802.540	1038.399
$\overline{\mu}'_{n,\mathcal{M}}$	519.223	1132.722	1774.902	2444.846	3127.341
$\overline{\tau}_{n,\mathcal{M}}$	0.032367	0.069811	0.110248	0.152287	0.196612
BF of $\overline{\mu}_{n,\mathcal{M}}$	$0.6372\ldots\ n\log n + 0.6346\ldots\ n$				
BF of $\overline{\tau}_{n,\mathcal{M}}$	$9.3343\ldots\cdot 10^{-5}\, n\log n + 2.6921\ldots\cdot 10^{-4}\, n$				
BF of $\overline{\mu}'_{n,\mathcal{M}}$	$\overline{\mu}'_{n,\mathcal{M}} \approx 2\,\overline{\mu}_{n,\mathcal{M}} + 4.2\,n$				

Table A.10: Boustrophedonic unlabelled Motzkin trees.

n	50	100	150	200	250
$\mu_{n,\mathcal{F}}$	342.946	866.549	1509.209	2248.274	3070.613
$\overline{\mu}_{n,\mathcal{F}}$	336.605	869.189	1497.514	2251.323	3018.220
$\overline{\mu}'_{n,\mathcal{F}}$	1588.283	3747.171	6191.075	9008.054	11867.418
$\epsilon_{n,\mathcal{F}}$	1.85 %	0.30 %	0.77 %	0.13 %	1.71 %
$\overline{\tau}_{n,\mathcal{F}}$	0.048730	0.125259	0.232866	0.359177	0.566123
$\mu_{n,\mathcal{F}}$	$0.6266\ldots\, n\sqrt{n} + 2.3333\ldots\, n + 0.6266\ldots\, \sqrt{n} + 0.2916\ldots + \mathrm{o}(1))$				
BF of $\overline{\mu}_{n,\mathcal{F}}$	$0.5910\ldots\, n\sqrt{n} + 2.7807\ldots\, n$				
$\mathrm{AQE}_{\mathcal{F}}$	$12.9481\ldots$				
BF of $\overline{\tau}_{n,\mathcal{F}}$	$1.7405\ldots \cdot 10^{-4}\, n\sqrt{n} - 5.5017\ldots \cdot 10^{-4}\, n$				
BF of $\overline{\mu}'_{n,\mathcal{F}}$	$\overline{\mu}'_{n,\mathcal{F}} \approx 3\,\overline{\mu}_{n,\mathcal{F}} + 11.2\, n$				

Table A.11: Lexicographic (labelled) functional graphs.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{F}}$	238.812	578.368	968.713	1402.059	1866.077
$\overline{\mu}'_{n,\mathcal{F}}$	1470.275	3221.821	5119.778	7148.806	9267.418
$\overline{\tau}_{n,\mathcal{F}}$	0.044865	0.100596	0.168831	0.252680	0.347581
BF of $\overline{\mu}_{n,\mathcal{F}}$	$1.7981\ldots\, n\log n - 2.4923\ldots\, n$				
BF of $\overline{\tau}_{n,\mathcal{F}}$	$3.9558\ldots \cdot 10^{-4}\, n\log n - 8.1434\ldots \cdot 10^{-4}\, n$				
BF of $\overline{\mu}'_{n,\mathcal{F}}$	$\overline{\mu}'_{n,\mathcal{F}} \approx 3\,\overline{\mu}_{n,\mathcal{F}} + 14.5\, n$				

Table A.12: Boustrophedonic (labelled) functional graphs.

n	50	100	150	200	250
$\mu_{n,\mathcal{S}_\mathsf{L}}$	100	200	300	400	500
$\overline{\mu}_{n,\mathcal{S}_\mathsf{L}}$	100	200	300	400	500
$\overline{\mu}'_{n,\mathcal{S}_\mathsf{L}}$	820.23	1640.87	2463.91	3277.41	4099.15
$\epsilon_{n,\mathcal{S}_\mathsf{L}}$	0%	0%	0%	0%	0%
$\overline{\tau}_{n,\mathcal{S}_\mathsf{L}}$	0.025027	0.052787	0.081367	0.112710	0.149673
$\mu_{n,\mathcal{S}_\mathsf{L}}$	$2\,n$				
BF of $\overline{\mu}_{n,\mathcal{S}_\mathsf{L}}$	$2\,n$				
$\mathrm{AQE}_{\mathcal{S}_\mathsf{L}}$	0				
BF of $\overline{\tau}_{n,\mathcal{S}_\mathsf{L}}$	$6.1843\ldots \cdot 10^{-4}\, n - 8.4517\ldots \cdot 10^{-3}$				
BF of $\overline{\mu}'_{n,\mathcal{S}_\mathsf{L}}$	$\overline{\mu}'_{n,\mathcal{S}_\mathsf{L}} \approx 8.2\,\overline{\mu}_{n,\mathcal{S}_\mathsf{L}}$				

Table A.13: Lexicographic labelled surjections.

n	50	100	150	200	250
$\mu^*_{n,\mathcal{S}_U}$	137	274.5	412	549.5	687
$\overline{\mu}_{n,\mathcal{S}_U}$	124.379	249.501	374.589	499.087	624.628
$\overline{\mu}'_{n,\mathcal{S}_U}$	616.266	1228.294	1839.621	2454.673	3064.567
$\epsilon^*_{n,\mathcal{S}_U}$	9.21%	9.11%	9.08%	9.17%	9.08%
$\overline{\tau}_{n,\mathcal{S}_U}$	0.024055	0.048500	0.074818	0.099662	0.127071
BF of $\mu^*_{n,\mathcal{S}_U}$	$2.75\,n - 0.5 + o(1)$				
BF of $\overline{\mu}_{n,\mathcal{S}_U}$	$2.5002\ldots\; n - 0.5884\ldots$				
$\mathrm{AQE}^*_{\mathcal{S}_U}$	$1723.1119\ldots$				
BF of $\overline{\tau}_{n,\mathcal{S}_U}$	$5.1439\ldots\cdot 10^{-4}\,n - 2.3370\ldots\cdot 10^{-3}$				
BF of $\overline{\mu}'_{n,\mathcal{S}_U}$	$\overline{\mu}'_{n,\mathcal{S}_U} \approx 4.9\,\overline{\mu}_{n,\mathcal{S}_U}$				

Table A.14: Lexicographic unlabelled integer compositions.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{S}_L}$	143.105	288.969	434.646	580.373	725.869
$\overline{\mu}'_{n,\mathcal{S}_L}$	1048.625	2105.380	3163.091	4216.207	5268.916
$\overline{\tau}_{n,\mathcal{S}_L}$	0.029814	0.064011	0.099753	0.137962	0.184334
BF of $\overline{\mu}_{n,\mathcal{S}_L}$	$2.9138\ldots\; n - 2.4872\ldots$				
BF of $\overline{\tau}_{n,\mathcal{S}_L}$	$7.6598\ldots\cdot 10^{-4}\,n - 1.1722\ldots\cdot 10^{-2}$				
BF of $\overline{\mu}'_{n,\mathcal{S}_L}$	$\overline{\mu}'_{n,\mathcal{S}_L} \approx 7.24\,\overline{\mu}_{n,\mathcal{S}_L} + 14$				

Table A.15: Boustrophedonic labelled surjections.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{S}_U}$	176.538	357.967	539.189	720.081	901.762
$\overline{\mu}'_{n,\mathcal{S}_U}$	701.707	1407.031	2112.719	2822.500	3524.755
$\overline{\tau}_{n,\mathcal{S}_U}$	0.028540	0.060395	0.097300	0.156227	0.211000
BF of $\overline{\mu}_{n,\mathcal{S}_U}$	$3.6251\ldots\; n - 4.6612\ldots$				
BF of $\overline{\tau}_{n,\mathcal{S}_U}$	$9.2150\ldots\cdot 10^{-4}\,n - 2.7533\ldots\cdot 10^{-2}$				
BF of $\overline{\mu}'_{n,\mathcal{S}_U}$	$\overline{\mu}'_{n,\mathcal{S}_U} \approx 3.9\,\overline{\mu}_{n,\mathcal{S}_U} + 11$				

Table A.16: Boustrophedonic unlabelled integer compositions.

n	50	100	150	200	250
$\mu^*_{n,\mathcal{P}_1}$	91.588	169.799	244.862	318.178	390.326
$\overline{\mu}_{n,\mathcal{P}_1}$	86.506	162.047	234.537	307.651	378.644
$\overline{\mu}'_{n,\mathcal{P}_1}$	361.148	665.168	955.342	1250.304	1531.132
$\epsilon^*_{n,\mathcal{P}_1}$	5.55%	4.56%	4.22%	3.31%	2.99%
$\overline{\tau}_{n,\mathcal{P}_1}$	0.013550	0.026316	0.042631	0.062979	0.088587
BF of $\overline{\mu}_{n,\mathcal{P}_1}$	$1.4598\ldots\, n + 14.9130\ldots$				
$\mathrm{AQE}^*_{\mathcal{P}_1}$	$87.9625\ldots$				
BF of $\overline{\tau}_{n,\mathcal{P}_1}$	$3.7347\ldots \cdot 10^{-4}\, n - 9.2085\ldots \cdot 10^{-3}$				
BF of $\overline{\mu}'_{n,\mathcal{P}_1}$	$\overline{\mu}'_{n,\mathcal{P}1} \approx 4\,\overline{\mu}_{n,\mathcal{P}_1} + 17$				

Table A.17: Lexicographic (unlabelled) integer partitions.

n	50	100	150	200	250
$\mu^*_{n,\mathcal{P}_2}$	122.797	232.061	338.034	442.183	545.116
$\overline{\mu}_{n,\mathcal{P}_2}$	113.165	217.059	318.449	422.416	521.776
$\overline{\mu}'_{n,\mathcal{P}_2}$	357.099	644.006	913.500	1185.329	1443.830
$\epsilon^*_{n,\mathcal{P}_2}$	7.84%	6.46%	5.79%	4.47%	4.28%
$\overline{\tau}_{n,\mathcal{P}_2}$	0.006049	0.010345	0.016944	0.029783	0.039157
BF of $\overline{\mu}_{n,\mathcal{P}_2}$	$2.0451\ldots\, n + 11.7993\ldots$				
$\mathrm{AQE}^*_{\mathcal{P}_2}$	$327.3795\ldots$				
BF of $\overline{\tau}_{n,\mathcal{P}_2}$	$1.7131\ldots \cdot 10^{-4}\, n - 5.2406\ldots \cdot 10^{-3}$				
BF of $\overline{\mu}'_{n,\mathcal{P}_2}$	$\overline{\mu}'_{n,\mathcal{P}2} \approx 2.65\,\overline{\mu}_{n,\mathcal{P}_2,+}64$				

Table A.18: Lexicographic (unlabelled) integer partitions.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{P}_1}$	144.586	285.524	420.132	561.740	691.782
$\overline{\mu}'_{n,\mathcal{P}_1}$	450.411	854.352	1232.867	1630.327	1995.301
$\overline{\tau}_{n,\mathcal{P}_1}$	0.017986	0.034150	0.055118	0.080435	0.111770
BF of $\overline{\mu}_{n,\mathcal{P}_1}$	$2.7412\ldots\, n + 9.5704\ldots$				
BF of $\overline{\tau}_{n,\mathcal{P}_1}$	$4.6771\ldots \cdot 10^{-4}\, n - 1.0264\ldots \cdot 10^{-2}$				
BF of $\overline{\mu}'_{n,\mathcal{P}_1}$	$\overline{\mu}'_{n,\mathcal{P}1} \approx 2.8\,\overline{\mu}_{n,\mathcal{P}_1} + 54$				

Table A.19: Boustrophedonic (unlabelled) integer partitions.

n	50	100	150	200	250
$\overline{\mu}_{n,\mathcal{P}_2}$	126.516	243.274	352.588	466.985	570.953
$\overline{\mu}'_{n,\mathcal{P}_2}$	334.233	602.910	845.249	1096.710	1324.036
$\overline{\tau}_{n,\mathcal{P}_2}$	0.007011	0.012088	0.022683	0.036652	0.048369
BF of $\overline{\mu}_{n,\mathcal{P}_2}$	$2.2252\ldots n + 18.2877\ldots$				
BF of $\overline{\tau}_{n,\mathcal{P}_2}$	$2.1456\ldots \cdot 10^{-4}\, n - 6.8234\ldots \cdot 10^{-3}$				
BF of $\overline{\mu}'_{n,\mathcal{P}_2}$	$\overline{\mu}'_{n,\mathcal{P}2} \approx 2.2\,\overline{\mu}_{n,\mathcal{P}_2} + 66$				

Table A.20: Boustrophedonic (unlabelled) integer partitions.

A.2 Plots

In this second section, we have plotted some results for the empirically mesured average cost to unrank an object of $\mathcal{A}_n$ ($\overline{\mu}_{n,\mathcal{A}}$) and the corresponding average CPU time (in seconds) ($\overline{\tau}_{n,\mathcal{A}}$). Data points for labelled cases are marked with "△", while those for unlabelled cases are marked with "□". On the other hand, we use solid lines "—" to plot the data for lexicographic unranking and dashed lines "- -" to plot the data for boustrophedonic unranking. X-axis denotes the size and the Y-axis denotes the experimental data ($\overline{\mu}_{n,\mathcal{A}}$ or $\overline{\tau}_{n,\mathcal{A}}$).

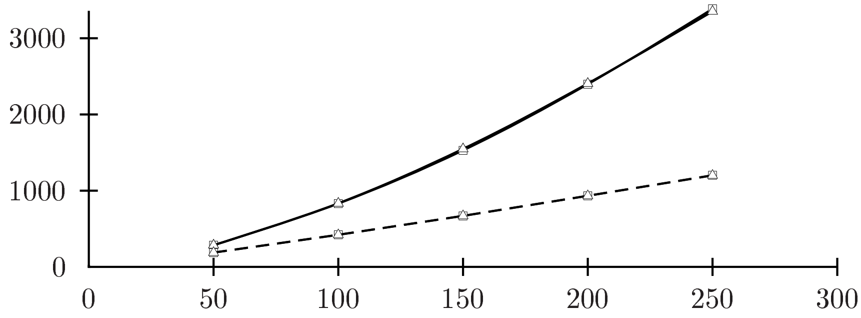


Figure A.1: Experimental average cost $\overline{\mu}_{n,\mathcal{B}}$ for labelled and unlabelled binary trees.

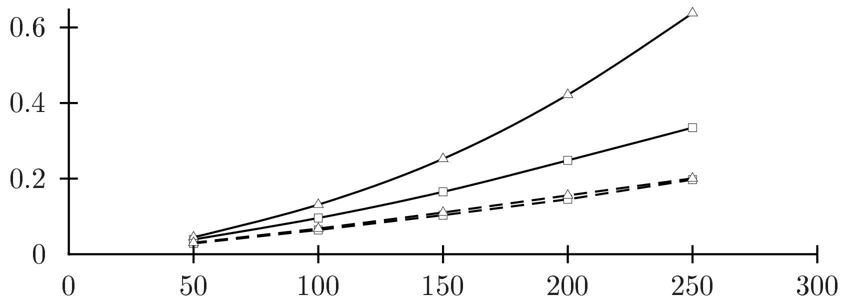


Figure A.2: CPU time $\overline{\tau}_{n,\mathcal{B}}$ for labelled and unlabelled binary trees (seconds).

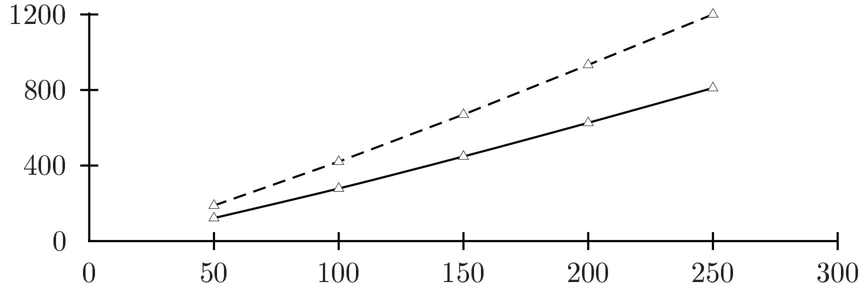


Figure A.3: Experimental average cost $\overline{\mu}_{n,\mathcal{B}}$ for labelled binary trees: Boustrophedonic order (upper line) v.s. (lexicographic) boxed heuristic (lower line).

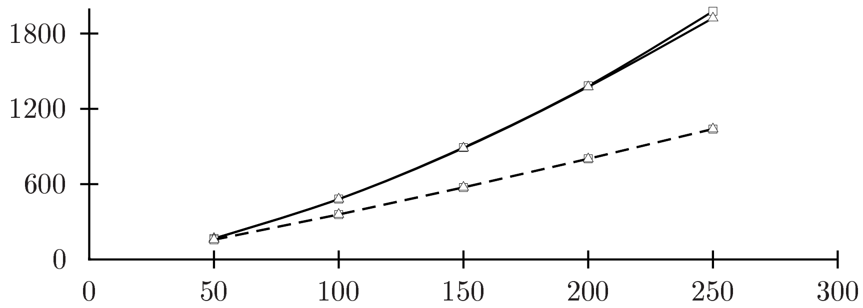


Figure A.4: Experimental average cost $\overline{\mu}_{n,\mathcal{M}}$ for labelled and unlabelled Motzkin trees.

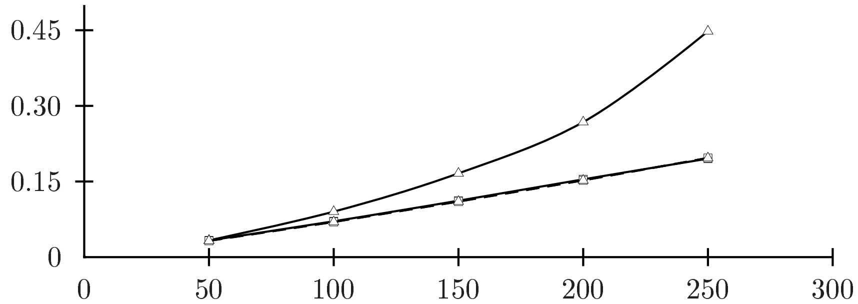


Figure A.5: CPU time $\overline{\tau}_{n,\mathcal{M}}$ for labelled and unlabelled Motzkin trees (seconds)[1].

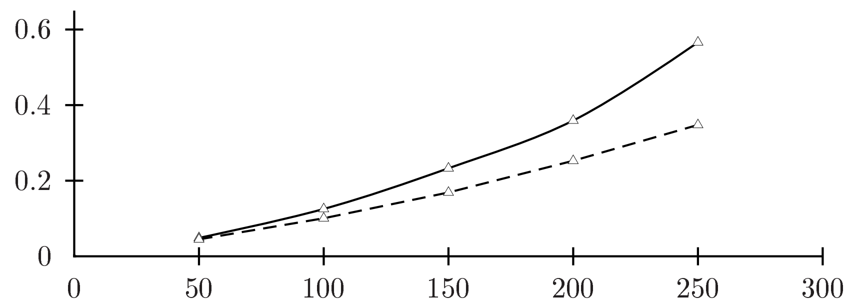


Figure A.6: CPU time $\overline{\tau}_{n,\mathcal{F}}$ for functional graphs (seconds).

[1] There is almost no difference for unlabelled unranking (boustrophedonic or lexicographic).

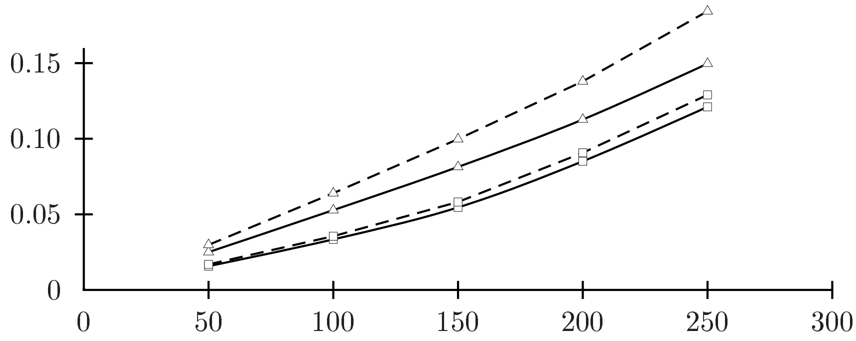


Figure A.7: CPU time $\overline{\tau}_{n,\mathcal{S}}$ for labelled surjections and unlabelled integer compositions (seconds).

Bibliography

[1] M. Abramowitz and I.A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publ., New York, 1964.

[2] S. Bacchelli, E. Barcucci, E. Grazzini, and E. Pergola. Exhaustive generation of combinatorial objects by ECO. *Acta Informatica*, 40(8):585–602, 2004.

[3] E. Barcucci, A. Del Lungo, and E. Pergola. Random generation of trees and other combinatorial objects. *Theoretical Computer Science*, 218(2):219–232, 1999.

[4] E. Barcucci, A. Del Lungo, E. Pergola, and R. Pinzani. ECO: a general methodology for the enumeration of combinatorial objects. *J. Difference Equations and Applications*, 5:435–490, 1999.

[5] C.M. Bender and S.A. Orszag. *Advanced Mathematical Methods for Scientists and Engineers*. McGraw-Hill, 1978.

[6] E.A. Bender. Asymptotic methods in enumeration. *SIAM Review*, 16(4):485–515, 1974.

[7] B.W. Char. MAPLE *V Library Reference Manual*. Springer-Verlag, 1993.

[8] MuPAD-Combinat – algebraic combinatorics package for MuPAD. URL: http://mupad-combinat.sourceforge.net/.

[9] Algolib. URL: http://algo.inria.fr/libraries/#combstruct.

[10] The Combinatorial Object Server. Designed and maintained by Frank Ruskey. URL: http://www.theory.cs.uvic.ca/~cos/.

[11] I. Dutour. *Grammaires d'objects: énumération, bijections et génération aléatoire*. PhD thesis, Université Bordeaux I, 1996.

[12] M.C. Er. A note on generating well-formed parenthesis strings lexicographically. *The Computer Journal*, 26(3):205–207, 1983.

[13] S. Even. *Combinatorial Algorithms*. MacMillan, New York, 1973.

[14] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 2. J. Wiley, 1968.

[15] P. Flajolet. Mathematical methods in the analysis of algorithms and data structures. *Trends in Theoretical Computer Science*, pages 225–304, 1988.

[16] P. Flajolet. Singular combinatorics. *Theoretical Computer Science*, 132(1):1–35, 1994.

[17] P. Flajolet. Singular combinatorics. In *International Congress of Mathematicians 2002*, volume III, pages 561–571, World Scientific, 2002.

[18] P. Flajolet and A.M. Odlyzko. Singularity analysis of generating functions. *SIAM J. on Discrete Mathematics*, 3(2):216–240, 1990.

[19] P. Flajolet and B. Salvy. Computer algebra libraries for combinatorial structures. *J. Symbolic Computation*, 20:653–671, 1995.

[20] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science, Series A*, 79(1):37–109, 1991.

[21] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Complex asymptotics and generating functions. Technical Report 2026, INRIA, 1993.

[22] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Counting and generating functions. Technical Report 1888, INRIA, 1993.

[23] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Saddle point asymptotics. Technical Report 1888, INRIA, 1994.

[24] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Multivariate asymptotics and limit distributions. Technical Report 3162, INRIA, 1997.

[25] P. Flajolet and J.S. Vitter. Average-case Analysis of Algorithms and Data Structures. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9. North-Holland, 1990.

[26] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of combinatorial structures. *Theoretical Computer Science*, 132(1):1–35, 1994.

[27] K. Gehrs and F. Postel. *MuPAD - A Practical Guide. Mathematics made anew. Tools and Texts for Computer Aided Learning*, volume 1. SciFace Software (SciFace), Paderborn, 2003.

[28] J. Gerhard, W. Oevel, F. Postel, and S. Wehmeier. *MuPAD Tutorial. A version and platform independent introduction.* Springer Verlag, 2000.

[29] L.A. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures.* Cambridge University Press, 1993.

[30] L.A. Goldberg and D.M. Jackson. *Combinatorial Enumeration.* John Wiley & Sons, 1983.

[31] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, M.A., 1989.

[32] D.H. Greene. Labelled formal languages and their uses. Technical Report 982, PhD Thesis, Stanford University, STAN-CS-83-982, 1983.

[33] D.H. Greene and D.E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhauser, Boston, 2nd edition, 1982.

[34] G.H. Hardy. *Divergent Series.* Oxford University Press, 1949.

[35] B. Harris and L. Schoenfeld. Asymptotic expansions for the coefficients of analytic functions. *Illinois J. Math.*, 12:264–277, 1968.

[36] W.K. Hayman. A generation of Stirling's formulas. *J. reine Angew. Math.*, 196:67–95, 1956.

[37] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM J. of Computing*, 12(4):645–655, 1983.

[38] F. Hivert and N.M. Thiéry. MuPAD-combinat, an open-source package for research in algebraic combinatorics. *Séminaire Lotharingien de Combinatoire*, 51:Article B51z, 70pp., 2004. URL: http://www.mat.univie.ac.at/~slc/wpapers/s51thiery.html.

[39] T.C. Hu. *Combinatorial Algorithms.* Addison-Wesley, 1982.

[40] K. Jensen and N. Wirth. *PASCAL User Manual and Report*, volume 18. Springer, 1974.

[41] R. Kemp. Generating words lexicographically: An average-case analysis. *Acta Informatica*, 35(1):17–89, 1998.

[42] D.E. Knuth. *The Art of Computer Programming, vol. 1: Fundamental Algorithms*, volume 1. Addison-Wesley, 1968.

[43] D.E. Knuth. *The Art of Computer Programming, vol. 2*, volume 2. Addison-Wesley, 1969.

[44] D.E. Knuth. *The Art of Computer Programming, vol. 3: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[45] D.E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms.* Addison-Wesley, 2007. In preparation. Available at http://www-cs-faculty.stanford.edu/~knuth.

[46] D.L. Kreher and D.R. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search.* CRC Press LLC, 1999.

[47] L. Kucera. *Combinatorial Algorithms.* Adam Hilger, 1990.

[48] Z. Li and E.M. Reingold. Solution of a divide-and-conquer maximin recurrence. *SIAM J. of Computing*, 18(6):1188–1200, 1989.

[49] J. Liebehenschel. Ranking and unranking of lexicographically ordered words: An average-case analysis. *J. of Automata, Languages and Combinatorics*, 2(4):227–268, 1997.

[50] J. Liebehenschel. Ranking and unranking of a generalized dyck language and the application to the generation of random trees. In *The Fifth International Seminar on the Mathematical Analysis of Algorithms*, Bellaterra (Spain), 1999.

[51] J.M. Lucas, D.R. van Baronaigien, and F. Ruskey. On rotations and the generation of binary trees. *J. of Algorithms*, 15:343–366, 1993.

[52] A. Del Lungo, A. Frosini, and S. Rinaldi. ECO method and the exhaustive generation of convex polyominoes. In C.S. Calude, M.J. Dinneen, and V. Vajnovszki, editors, *Proc. 4th Int. Conf. Discrete Mathematics and Theoretical Computer Science (DMTCS)*, volume 2731 of *LNCS*, pages 129–140, 2003.

[53] M.L. Majewski. *MuPAD Pro Computing Essentials*. Springer Verlag, Berlin; New York, 2002.

[54] C. Martínez and X. Molinero. Unranking of labelled combinatorial structures. In D. Krob, A. A. Mikhalev, and A. V. Mikhalev, editors, *Formal Power Series and Algebraic Combinatorics: 12th International Conference; Procedings / FPSAC'00*, pages 288–299. Springer-Verlag, 2000.

[55] C. Martínez and X. Molinero. Generic algorithms for the exhaustive generation of labelled objects. In *Proc. Workshop on Random Generation of Combinatorial Structures and Bijective Combinatorics (GASCOM' 01)*, pages 53–58. Università degli Studi di Siena & Gruppo Nazionale per il Calcolo Scientifico, 2001.

[56] C. Martínez and X. Molinero. A generic approach for the unranking of labeled combinatorial classes. *Random Structures & Algorithms*, 19(3-4):472–497, 2001.

[57] C. Martínez and X. Molinero. Generic algorithms for the generation of combinatorial objects. In Branislav Rovan and Peter Vojtáš, editors, *Mathematical Foundations of Computer Science 2003 (28th International Symposium, MFCS 2003; Bratislava, Slovakia, August 2003; Proceedings)*, volume 2747 of *LNCS*, pages 572–581. Springer, 2003.

[58] C. Martínez and X. Molinero. An efficient generic algorithm for the generation of unlabelled cycles. In Gardy D. Drmota M., Flajolet P. and Gittenberger B., editors, *Mathematics and Computer Science III: Algorithms, Trees, Combinatorics and Probabilities*, volume XV, pages 187–197. Birkhäuser, 2004.

[59] C. Martínez and X. Molinero. An experimental study of unranking algorithms. In Simone L. Martins Celso C. Ribeiro, editor, *LNCS: III Workshop on Efficient and Experimental Algorithms*, volume 3059 of *LNCS*, pages 326–340. Springer, 2004.

[60] C. Martínez and X. Molinero. Efficient iteration in admissible combinatorial classes. *Theoretical Computer Science*, 2005. To appear.

[61] A. Meir and J.W. Moon. On the altitude of nodes in random trees. *Canad. J. Math.*, 5(30):997–1015, 1978.

[62] A. Meir and J.W. Moon. On an asymptotic method in combinatorics. *J. Combinatorial Theory, Series A*, 51:77–89, 1989.

[63] A. Meir and J.W. Moon. The asymptotic behaviour of coefficients of powers of certain generating functions. *European J. Combinatorics*, 11:581–587, 1990.

[64] X. Molinero. Iterating through labelled combinatorial structures. In *Proc. Workshop on EuroConference on Combinatorics, Graph Theory and Applications (COMB' 01)*, pages 229–232, 2001.

[65] MuPAD: The open computer algebra system. URL: http://www.mupad.de/.

[66] P. Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[67] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms: For Computers and Calculators.* Academic Press, Inc., 1978.

[68] A.M. Odlyzko and L.B. Richmond. Asymptotic expansions for the coefficients of analytic generating functions. *Aequationes Math.*, 44:50–63, 1985.

[69] J.M. Pallo. Enumerating, ranking and unranking binary trees. *The Computer Journal*, 29(2):171–175, 1986.

[70] E.M. Reingold, J. Nievergelt, and N.Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[71] D. Rotem and Y.L. Varol. Generating of binary trees from ballot sequences. *J. of the ACM*, 25(3):396–404, 1978.

[72] F. Ruskey and T.C. Hu. Generating binary trees lexicographically. *SIAM J. of Computing*, 6:745–758, 1977.

[73] F. Ruskey and T.C. Hu. Generating binary trees lexicographically. *SIAM J. of Computing*, 7(4):492–509, 1978.

[74] F. Ruskey and A. Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11:68–84, 1990.

[75] F. Ruskey and J. Sawada. An efficient algorithm for generating necklaces with fixed density. *SIAM J. of Computing*, 29(2):671–684, 1999.

[76] J. Sawada. A fast algorithm to generate necklaces with fixed content. *Theoretical Computer Science*, 301(1-3):477–489, 2003.

[77] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9(3):137–164, 1977.

[78] R. Sedgewick. Mathematical analysis of combinatorial algorithms. *Probability Theory and Computer Science*, pages 124–205, 1983.

[79] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, Reading, MA, 1996.

[80] L. Takács. A Bernoulli excursion and its various applications. *Advances in Applied Probability*, 23:557–585, 1991.

[81] L. Takács. Conditional limit theorems for branching processes. *J. of Applied Mathematics and Stochastic Analysis*, 4(4):263–292, 1991.

[82] L. Takács. On a probability problem connected with railway traffic. *J. of Applied Mathematics and Stochastic Analysis*, 4(1):1–27, 1991.

[83] E.C. Titchmarsh. *The Theory of Functions.* Oxford University Press, 1939.

[84] M.B. Wells. Generation of permutations by transposition. *Mathematics of Computation*, 15:192–195, 1961.

[85] H.S. Wilf. *Combinatorial Algorithms: An Update.* SIAM, 1989.

[86] H.S. Wilf. East side, west side ... an introduction to combinatorial families- with Maple programming. Technical report, 1999. URL: http://www.cis.upenn.edu/~wilf/lecnotes.html.

[87] P. Zimmermann. *Séries Génératrices et Analyse Automatique d'Algorithmes.* PhD thesis, École Polytechnique, Palaiseau, 1991.

[88] P. Zimmermann. Random generation of unlabelled combinatorial structures. Technical report, INRIA, Lorraine, 1993. It is part of INRIA research report RR 2381.

[89] P. Zimmermann. Gaïa: A package for the random generation of combinatorial structures. 1(1):38–46, 1994.

[90] P. Zimmermann. Uniform random generation for the powerset construction. Technical report, INRIA, Lorraine, 1995.