CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

| | |
|---|---|
| Title | Practical programming for static average-case analysis: the MOQA investigation |
| Authors | Townley, Jacinta Maria |
| Publication date | 2013 |
| Original Citation | Townley, J. M. 2013. Practical programming for static average-case analysis: the MOQA investigation. PhD Thesis, University College Cork. |
| Type of publication | Doctoral thesis |
| Rights | © 2013, Jacinta Maria Townley - http://creativecommons.org/licenses/by-nc-nd/3.0/ |
| Download date | 2026-11-07 12:37:47 |
| Item downloaded from | https://hdl.handle.net/10468/1345 |



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh




Department of Computer Science
National University of Ireland, Cork

A thesis submitted for the degree of
Doctor of Philosophy

---

# Practical Programming for Static Average-Case Analysis: the MOQA Investigation

---

Jacinta Maria Townley

*Supervisor:* Dr. Joseph Manning

*Head of Department:* Prof. Barry O'Sullivan

September 2013

# Contents

**List of Figures** v

**List of Tables** viii

**Declaration** ix

**Acknowledgements** x

**Abstract** xi

**1 Introduction** 1

1.1 Problem Statement . . . . . 1

1.2 Aims and Objectives . . . . . 2

1.3 Thesis Layout . . . . . 2

**2 MOQA Background** 4

2.1 MOQA Theory . . . . . 5

2.2 MOQA Functions . . . . . 13

2.2.1 MOQA Product . . . . . 16

2.2.2 MOQA Split . . . . . 18

2.2.3 MOQA Deletion . . . . . 19

2.2.4 MOQA Projection . . . . . 21

2.3 Average-case Cost in MOQA . . . . . 21

2.4 MOQA Algorithms . . . . . 28

2.5 Chapter Summary . . . . . 28

**3 The MOQA Language** 30

3.1 The MOQA Language Implementation . . . . . 30

3.1.1 *MOQA-Java* Details . . . . . 32

3.1.2 *MOQA-Java* Functions . . . . . 40
3.2 The Cost of Some *MOQA-Java* Examples . . . . . 45
3.2.1 The Space Cost of Insertion-sort in *MOQA-Java* . . . . . 45
3.2.2 The Average-case Cost of Quicksort in *MOQA-Java* . . . 48
3.3 New MOQA Functions . . . . . 54
3.3.1 MOQA Top . . . . . 55
3.3.2 MOQA Bot . . . . . 55
3.3.3 MOQA Lift . . . . . 55
3.3.4 MOQA Insert . . . . . 56
3.4 MOQA and Reversibility . . . . . 57
3.5 Chapter Summary . . . . . 61

**4 Tracking Data Structure State 63**
4.1 Chapter Overview . . . . . 64
4.2 Program Control Flow . . . . . 67
4.3 Expanding the MOQA Theory . . . . . 68
4.4 Data Structures Represented by $P_\beta$ . . . . . 77
4.4.1 Fixed Po-structure . . . . . 77
4.4.2 Inductive Po-class . . . . . 79
4.4.3 Split Po-class and General Split Po-class . . . . . 91
4.4.4 Compound Structure . . . . . 93
4.5 The Average-case Cost of a MOQA Function . . . . . 98
4.5.1 The Average-case Cost of a MOQA Function Applied to $FPS_\beta$ . . . . . 98
4.5.2 The Average-case Cost of a MOQA Function Applied to $IPC_\beta$ . . . . . 98
4.5.2.1 The Average-case Cost of a MOQA Function Applied to a Finite $IPC_\beta$ . . . . . 98
4.5.2.2 The Average-case Cost of a MOQA Function Applied to an Infinite $IPC_\beta$ . . . . . 99
4.5.2.3 The Average-case Cost of a MOQA Function Applied to an Infinite $DIPC_\beta$ . . . . . 100
4.5.2.4 The Average-case Cost of a MOQA Function Applied to an Infinite $NDIPC_\beta$ . . . . . 141

4.5.3 The Average-case Cost of a MOQA Function Applied to $SPC_{\beta_{max},n}$ and $GSPC_{\beta_{max},Z}$ . . . 156
4.5.4 The Average-case Cost of a MOQA Function Applied to $CS_\beta$ . . . 156
4.6 The Number of Canonically-ordered Labelings on $IPC_\beta$ . . . 157
4.7 The Number of Canonically-ordered Labelings on $SPC_{\beta_{max},n}$ and $GSPC_{\beta_{max},Z}$ . . . 162
4.8 Chapter Summary . . . 162

**5 Duplicate Labels 164**
5.1 The Duplicate Label Question . . . 164
5.2 Random Selection . . . 166
5.3 Label Distribution . . . 173
5.4 Duplicate Labels in Insertion-sort . . . 176
5.5 Duplicate Labels in Quicksort . . . 182
5.6 Chapter Summary . . . 184

**6 Literature Review 185**
6.1 PL and EL . . . 185
6.2 Metric . . . 190
6.3 ACE . . . 196
6.4 COMPLEXA . . . 200
6.5 LUO . . . 205
6.6 Mishna . . . 217
6.7 Sarkar . . . 219
6.8 Other Related Research . . . 222
6.9 Randomness Preservation . . . 224
6.10 Chapter Summary . . . 231

**7 Conclusion 232**
7.1 MOQA Assessment . . . 232
7.2 Future Work . . . 239
7.3 Thesis Summary . . . 240

**A Treapsort Algorithm 242**

**B Algorithms LUO Can Analyse 244**

**Bibliography** **245**

# List of Figures


2.1 I) A total order, II) The unlabeled Hasse diagram of I) . . . . . 7
2.2 The unlabeled Hasse diagram $H^a$ . . . . . 8
2.3 A data structure that is not series-parallel . . . . . 10
2.4 An unlabeled Hasse diagram . . . . . 18
2.5 Figure 2.4 after a MOQA product function . . . . . 18
2.6 The four distinct Hasse diagrams that can result after the MOQA split function is applied to a discrete Hasse diagram whose size is four . . . . . 19
2.7 Figure 2.5 after a downwards MOQA deletion function . . . . . 21

3.1 The *MOQA-Java* class diagram . . . . . 33
3.2 Adding label values to MOQA data structures in *MOQA-Java* — the first argument to a NodeInfo's constructor is a Label object / label value and the second argument is the data associated with that Label object / label value . . . . . 35
3.3 LPO I, the *MOQA-Java* representation of a discrete partial order of size seven, becomes LPO II after the MOQA product function products elements a, b and c above elements d and e . 38
3.4 LPO II' is LPO II in Figure 3.3 without the SubLPO design . . 39
3.5 Insertion-sort in *MOQA-Java* . . . . . 46
3.6 Quicksort in *MOQA-Java* . . . . . 49
3.7 For the MOQA product function and $\beta_{max}$, half of the possible input pairs to product($x_i, y_i$) that result in the output of a total order of size four . . . . . 59

4.1 Inductive Po-Class Framework . . . . . 66
4.2 A simple example in *MOQA-Java*, program $p'_1$ . . . . . 78

4.3 One of these fixed po-structures will result after the MOQA deletion function in Figure 4.2 . . . . . 79
4.4 I) A fixed po-structure, II) A fixed po-structure and III) A condensed representation of I and II . . . . . 80
4.5 A simple example in *MOQA-Java*, program $p'_2$ . . . . . 82
4.6 Control flow graph of Figure 4.5 . . . . . 83
4.7 The five distinct BSTs of size three . . . . . 86
4.8 The sequence of production rules that constructed the far left fixed po-structure in Figure 4.7 . . . . . 87
4.9 A simple example in *MOQA-Java*, program $p'_4$ . . . . . 94
4.10 Control flow graph of Figure 4.9 . . . . . 94
4.11 $VLIST$ — a compound structure . . . . . 95
4.12 A $T$ production rule sequence that constructs a fixed po-structure of size four . . . . . 104
4.13 Another $T$ production rule sequence that constructs a fixed po-structure of size four . . . . . 105
4.14 II(a) - IV(a) show the labels with ranks from 1 to 3 that can swapped on to I(a) and $*$(b) shows the canonically-ordered labeling that $*$(a) can be reduced to . . . . . 127
4.15 A fixed po-structure in $A$'s infinite set . . . . . 146

5.1 For the label set $\{1, 2\}$ and $\beta_{max}$, all the distinct labelings of a discrete fixed po-structure of size three when 2 is assigned twice 165
5.2 For the label set $\{1, 2_a, 2_b\}$ and $\beta_{max}$, all the distinct labelings of a discrete fixed po-structure of size three . . . . . 165
5.3 Figure 5.1 just after the first node from left to right is connected above the next two nodes . . . . . 166
5.4 Figure 5.2 just after the first node from left to right is connected above the next two nodes . . . . . 167
5.5 New code for a random selection technique discussed in this section . . . . . 169
5.6 New code for Schellekens's random selection technique [63] . . . 171
5.7 The labelings of Figure 5.4 after adjustment by the MOQA product function . . . . . 172

6.1 Mergesort in *MOQA-Java* . . . . . 189

6.2 The unlabeled and labeled operations from which LUO's unlabeled and labeled combinatorial structures are respectively defined 206
6.3 The data structures $U$ and $V$ . . . . . 211
6.4 Four possible inputs for a product that involves $U$ and $V$ when they are unrelated structures and the relabelings that result when LUO's partitional product connects $U$ above $V$ for the leftmost input . . . . . 212
6.5 The five distinct BSTs of size three with labels . . . . . 225

# List of Tables


2.1 $L(H^a_{\beta_{max}})$ . . . . . 8
2.2 All distinct canonically-ordered labelings of Figure 2.4 . . . . . 18
2.3 Table 2.2's labelings after reorganisation by the MOQA product function depicted in Figure 2.5 . . . . . 18
2.4 All distinct canonically-ordered labelings of Figure 2.5 . . . . . 21
2.5 Table 2.4's labelings after reorganisation by the MOQA deletion function depicted in Figure 2.7 . . . . . 21

5.1 $L(D_{\beta_{max}}, K)$ when $N = 3$ and $K = \{1, 2\}$ . . . . . 174
5.2 $L(D_{\beta_{max}}, K)$ when $N = 3$ and $K = \{1, 2\}$ . . . . . 175

6.1 $L(U_{\beta_{max}} + V_{\beta_{max}})$ . . . . . 211
6.2 $L(U_{\beta_{max}})$ and $L(V_{\beta_{max}})$ . . . . . 212

# Declaration

This thesis is submitted to University College Cork, Ireland, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and theses presented herein are entirely the work of the author.

---

Jacinta Maria Townley

# Acknowledgements

*I am certain of nothing but of the holiness of the Heart's affections and the truth of Imagination.*

John Keats

I would first and foremost like to thank my supervisor Dr. Joseph Manning. His support and kind words never wavered and it seems safe to say that without him this work would not be in your hands right now. I will fondly remember the many conversations that we have had over the years and look forward to the ones to come.

Outside of college there have been those special people who loved and cared for me, who gave my life real meaning and purpose. These people have left a lasting impression on my heart. I cannot thank them enough and listing them here is but poor compensation: Maura, Sandy, Ilona, Stig, Joan, Rosy, Bert, Carole, Terry, Debbie and Shelly. You all gave me your best with no consideration of cost while Maura rescued me.

I would also like to show my appreciation to my other good friends, with whom I have enjoyed many moments. Last but far from least, I would like to thank my examiners Dr. Patrick Healy and Dr. Kieran Herley. I am sincerely grateful for the time they took out of their busy schedules to carefully read and consider this work; they have both my respect and sympathy!

# Abstract

This work considers the static calculation of a program's average-case time. The number of systems that currently tackle this research problem is quite small due to the difficulties inherent in average-case analysis. While each of these systems make a pertinent contribution, and are individually discussed in this work, only one of them forms the basis of this research. That particular system is known as MOQA.

The MOQA system consists of the MOQA language and the MOQA static analysis tool. Its technique for statically determining average-case behaviour centres on maintaining strict control over both the data structure type and the labeling distribution. This research develops and evaluates the MOQA language implementation, and adds to the functions already available in this language. Furthermore, the theory that backs MOQA is generalised and the range of data structures for which the MOQA static analysis tool can determine average-case behaviour is increased. Also, some of the MOQA applications and extensions suggested in other works are logically examined here. For example, the accuracy of classifying the MOQA language as reversible is investigated, along with the feasibility of incorporating duplicate labels into the MOQA theory. Finally, the analyses that take place during the course of this research reveal some of the MOQA strengths and weaknesses.

This thesis aims to be pragmatic when evaluating the current MOQA theory, the advancements set forth in the following work and the benefits of MOQA when compared to similar systems. Succinctly, this work's significant expansion of the MOQA theory is accompanied by a realistic assessment of MOQA's accomplishments and a serious deliberation of the opportunities available to MOQA in the future.

# Chapter 1

# Introduction

## 1.1 Problem Statement

The characteristics associated with various data structures and the algorithms that operate on them is a well-studied topic in modern computing. Hence, it is commonplace to consider an algorithm's best-case, average-case and worst-case behaviour. Of these three, best-case and worst-case behaviour establish the lower and upper bounds for program execution; this information is very useful to both soft and hard real-time systems. On the other hand, average-case behaviour may better reveal a program's nature because it involves all of the program's executions for some distribution. However, this latter behaviour is generally the most difficult of the three to establish because it first requires the selection of a meaningful distribution and then involves an analysis that covers many program executions. Accordingly, a variety of methodologies are currently in use in the area of average-case analysis, one example being Kolmogorov complexity [70].

The difficulties surrounding average-case analysis are further compounded when the move is made is obtain such information *statically*. One substantial concern is translating the intuitive reasonings performed during a hand-analysis for a particular approach into machine logic. This issue opens up an interesting research problem, which is how to calculate an algorithm's average-case behaviour automatically. Any static analysis tool that does so would be a valuable aid to system designers. It would release them from the, often complicated, mechanics involved in determining average-case behaviour and knowing this behaviour is helpful when it comes to allocating resources. Thus,

providing a reason, aside from intellectual curiosity, for the interest in this subject.

Therefore, there is a small collection of tools that attempt to automatically estimate an algorithm's average-case behaviour. One of these is MOQA [63] and the work reported in this thesis builds upon the MOQA research.

## 1.2 Aims and Objectives

The aims and objectives of this work revolve around MOQA (MOdular Quantative Analysis), which contributes to recent research into statically determining the average-case behaviour of computer programs. Published as a book [63], the MOQA theory aims to simplify the complexity normally associated with this field.

In closely examining the concepts behind MOQA, this work can be divided into four main areas:

- Implementing the MOQA language in a manner that encourages the programmer to adhere to its requirements.
- Expanding the functionality of the MOQA language and theory, along with identifying gaps yet to be filled.
- Evaluating the claims made about MOQA and its potential.
- Defining the boundaries of MOQA.

The last area is the most ambitious: to objectively examine the MOQA approach, determine the borders of its effectiveness and hence, clearly delineate its successes and limitations. This would allow MOQA to be carefully measured against existing research in the area, which is included in this work's objectives, and therefore assist in prioritising future work on MOQA. The author recognises that this is a considerable aim, difficult to achieve in its entirety, but hopes to bring some useful insights to light.

## 1.3 Thesis Layout

This thesis consists of five main chapters and one concluding chapter.

Chapter 2 introduces the reader to the MOQA theory and is a foundation chapter for each of those that follow.

Chapter 3's theme is the MOQA language. This chapter discusses the author's implementation of the MOQA language and considers the cost of certain algorithms written in it. It also presents new MOQA functions and investigates whether the MOQA language is a reversible language.

Chapter 4 concerns itself with broadening the MOQA theory. After doing so, it also provides new average-case formulas, which increase the power of the MOQA static analysis tool.

Chapter 5 deals with the issue of duplicate labels in MOQA and comes to an important decision regarding their inclusion in MOQA.

Chapter 6 is where the literature review takes place. This chapter often appears much earlier in similar bodies of work. However, it is the sixth chapter here because the reader will have the strongest grasp of the MOQA theory at this stage and this is the ideal when comparing MOQA to related systems.

Chapter 7 gives an overall assessment of MOQA and then finishes by identifying some future directions for MOQA.

# Chapter 2

# MOQA Background

The aim of MOQA [63] is to statically calculate the average-case cost of an algorithm. In MOQA, the average-case cost of an algorithm is measured by the average number of label-to-label comparisons that take place within that algorithm's data structures. If an algorithm is to be successfully analysed by MOQA, then it must adhere to a specific form; the functions used by the algorithm are those provided by MOQA and the algorithm follows certain control flow constraints. Such an algorithm is then parsed and evaluated by a MOQA static analysis tool. In general, the average-case cost that results is expressed as a recurrence relation. *MOQA-Java* is the current version of the MOQA language; see Chapter 3 for details. The current version of the MOQA static analysis tool is called *Distri-Track* [35].

MOQA functions control how an algorithm's data structures are accessed and modified. The intention of this regulation is 1), to remove any uncertainty statically about all the possible states of a data structure after a MOQA function is applied to it and 2), to ensure that these states follow a particular distribution. The average-case behaviour of functions that have these two properties will be easier to determine statically and such functions are known in MOQA as random bag preserving or random structure preserving. It is this concept that drives the MOQA theory. The definition of random bag/structure preservation is detailed in the following section and includes the anticipated distribution of states. (From this point onwards, random bag/structure preservation will be referred to as MOQA random bag/structure preservation so as to distinguish it from another related concept with a very similar name, discussed in Section 6.9.)

The sole purpose of this chapter is to concisely summarise over a hundred pages of core MOQA theory and therefore, will be a brief overview of the work rather than a direct reproduction. Hence, this recap will include definitions from works that are not quoted in the MOQA literature. Also, some of the MOQA-related lexicon of this chapter will differ to that of the other MOQA literature[1].

## 2.1 MOQA Theory

First, some useful preliminary definitions and concepts shall be introduced.

**Notation 1.** *Let $P$ denote a program.*

**Definition 1** (Composite variable)**.** *A composite variable is a variable in $P$ that refers to a data structure.*

In general, the data structures that will be discussed in this work are DAGs (Directed Acyclic Graphs).

**Notation 2.** *Let $c$ denote a composite variable.*

A program variable that refers to a data structure is identified here as a composite variable to differentiate the variable name from its possible states at any particular moment during its lifetime in the program.

**Definition 2** (A moment of a composite variable's lifetime)**.** *A moment of a composite variable's lifetime is an instant in time, with respect to a sufficiently coarse grain of time in which changing the value of the composite variable is an instantaneous operation [16].*

**Notation 3.** *Let $i_{P,c}$ denote a moment of $c$'s lifetime in $P$.*

**Notation 4.** *Let $F$ denote a function.*

For any average-case analysis tool, a prerequisite to statically determining the average-case behaviour of $F$ when applied to $c$ at $i_{P,c}$ is static awareness of all the various states that $c$ can take at $i_{P,c}$. There are different approaches

[1]The motivation behind this difference is an attempt to provide a MOQA notation of greater economy than the original, along with clarifying the occasional ambiguity that can be found in the latter.

to how these various states are represented statically, the study of which in Chapter 6 reveals how intertwined the representation and tracking of this information is. For now, just how MOQA captures this information is under consideration. As shall be seen shortly, MOQA expects $c$ to always refer to a Hasse diagram. So the **first step** is to identify all the unlabeled Hasse diagrams that $c$ can possibly take at $i_{P,\,c}$.

**Definition 3** (Reflexive edge)**.** *The edge (a, b) in a graph is reflexive if a and b are the same node.*

**Definition 4** (Transitive edge)**.** *The edge (a, b) in a graph is transitive if there is a path from a to b of length greater than one.*

**Definition 5** (Hasse diagram)**.** *A Hasse diagram is a DAG with no reflexive or transitive edges.*

For convenience of presentation, the term "Hasse diagram" will be used interchangeably in this work to mean either the underlying DAG or a drawing of it with all edges pointing upwards.

**Notation 5.** *Let $H$ denote an unlabeled Hasse diagram.*

In addition to statically knowing all the unlabeled Hasse diagrams that $c$ can possibly take at $i_{P,\,c}$, the **second step** involves statically knowing all of the possible values that can be stored within the data structures represented by these unlabeled Hasse diagrams. This motivates the introduction of a label ordering and a labeling. For these definitions, note that in MOQA the labels on a Hasse diagram are always selected from a totally-ordered set. So any two labels in such a set are comparable; the two labels $x$ and $y$ are comparable if $x \leq y$ or $y \leq x$.

**Definition 6** (Partial order)**.** *A partial order is a relation on a set of elements that is reflexive, transitive and antisymmetric.*

**Definition 7** (Label ordering)**.** *A label ordering is a method that assigns labels to the nodes of an unlabeled Hasse diagram according to some set of constraints.*

Note that such a partial order on the labels need bear no relationship with the direction of the edges in the unlabeled Hasse diagram.




Figure 2.1: I) A total order, II) The unlabeled Hasse diagram of I)

**Notation 6.** *Let $\beta$ denote a label ordering.*

An example of a label ordering would be the ordering of a heap or the ordering of a binary search tree. In other words, a label ordering describes how labels can be applied to an unlabeled Hasse diagram; it describes the rules for the pattern of arrangement, such as the label of a parent node must always be greater than or equal to the labels of its children nodes, which is known as *max-heap ordered*.

**Notation 7.** *Let $\beta_{max}$ denote the max-heap label ordering.*

**Notation 8.** *Let $H_\beta$ denote $H$ and some label ordering $\beta$ on it.*

**Definition 8** (Labeling)**.** *A labeling maps a label to each node in an unlabeled Hasse diagram according to a given label ordering.*

For the duration of this work, a labeling will always be assembled from a set of integers though any other totally-ordered data type, such as real numbers or strings, would be just as acceptable. Figure 2.1 provides an example of a labeling. In I the labels 4 and 7 are each associated with a node of II according to some label ordering. It is generally assumed, both here and by Schellekens [63], that a labeling will map $|H_\beta|$ *distinct* labels to the nodes in $H_\beta$; Chapter 5 discusses stepping away from this assumption of label distinctness.

So, if the average-case cost of $F$ when applied to $c$ at $i_{P,c}$ is to be considered, then it has just been stated that for the second step it is necessary to statically know the possible labelings, and precedently the $\beta$, of each distinct $H$ that $c$ can possible take at $i_{P,c}$. However, if the labels of a Hasse diagram are selected from an infinite set, then, for any label ordering, the diagram has an infinite number of distinct labelings from which its possible labelings are selected. Dealing with infinity is problematic from a static perspective and avoided where possible. Hence, it is common practise in average-case analysis

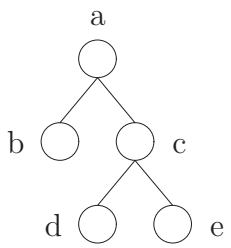



Figure 2.2: The unlabeled Hasse diagram $H^a$

| a | b | c | d | e |
|---|---|---|---|---|
| 5 | 4 | 3 | 2 | 1 |
| 5 | 4 | 3 | 1 | 2 |
| 5 | 3 | 4 | 2 | 1 |
| 5 | 3 | 4 | 1 | 2 |
| 5 | 2 | 4 | 3 | 1 |
| 5 | 2 | 4 | 1 | 3 |
| 5 | 1 | 4 | 3 | 2 |
| 5 | 1 | 4 | 2 | 3 |

Table 2.1: $L(H^a_{\beta_{max}})$

to instead consider a labeling in terms of the relative order between its labels, as opposed to the actual values of its labels, as relative order is bounded in terms of the number of labels. Therefore, in MOQA, the Hasse diagram's finite number of distinct canonically-ordered labelings for its $\beta$ is the set from which its possible labelings are selected.

**Definition 9** (Canonically-ordered labeling)**.** *A canonically-ordered labeling is a labeling that has its n distinct labels restricted to the values* $\{1, 2, \ldots, n\}$ *[42].*

Any labeling of a Hasse diagram can be reduced to a canonically-ordered labeling by mapping the $y^{th}$ smallest value in the labeling to the value $y$.

**Notation 9.** *Let* $L(H_\beta)$ *denote the set of all canonically-ordered labelings of* $H_\beta$*.*

Table 2.1 shows all the distinct canonically-ordered labelings of the unlabeled Hasse diagram in Figure 2.2 when $\beta$ is max-heap ordered.

The **third step** is to statically know, for each distinct $H_\beta$ that $c$ can take at $i_{P,c}$, not only which of its canonically-ordered labelings can occur but also how

often does each one of these labelings occur. In other words, for each distinct $H_\beta$ that $c$ can take at $i_{P,c}$, what is the distribution of the canonically-ordered labelings in $L(H_\beta)$? For example, imagine that the first labeling in Table 2.1 is a labeling of $H^a_{\beta_{max}}$ for four run-times, the next six labelings in the table are each a labeling of $H^a_{\beta_{max}}$ for two run-times and the last labeling in the table is never a labeling of $H^a_{\beta_{max}}$. For these sixteen run-times, the frequency of the first labeling is four and the frequency of each of the next six is two, or it can be said that the frequency of the first labeling is two and the frequency of each of the next six is one. (As the last labeling in Table 2.1 is never a possible labeling of $H^a_{\beta_{max}}$, its frequency is always zero.) The revised distribution still correctly reflects the labelings' comparative frequency, though the number of actual run-times involved is now lost. As the average number of label-to-label comparisons is of interest, and not the total, this loss of information, if it occurs, has no impact.

To summarise these three steps, MOQA needs to be statically aware of the following:

- Each distinct $H$ that $c$ can possibly take at $i_{P,c}$.
- For each distinct $H$ that $c$ can possibly take at $i_{P,c}$, what its $\beta$ is and which of its canonically-ordered labelings are possible values of $c$ at $i_{P,c}$.
- For each distinct $H$ that $c$ can possibly take at $i_{P,c}$, the distribution of its canonically-ordered labelings that are possible values of $c$ at $i_{P,c}$.

The MOQA static analysis tool uses the above information in conjunction with the operational semantics of $F$ to calculate the average-case cost of $F$ when applied to all of $c$'s possible states at $i_{P,c}$. Finally, the tool must compute how $F$ transforms these states, i.e. must determine the above information for input to the subsequent function.

In order to automatically calculate the average number of label-to-label comparisons that take place within an algorithm's data structures, MOQA tracks the above information by restricting both facets of the data organisation.

**Definition 10** (Data organisation)**.** *A data organisation is a "class of data structures together with the associated algorithms for operating on these structures" [42].*

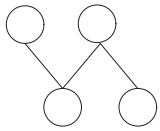


Figure 2.3: A data structure that is not series-parallel

A data organisation is also known as an ADT (Abstract Data Type), the latter term being more widely used.

So MOQA limits both the possible states of $c$ at $i_{P,c}$ and the type of function that can be applied to $c$. The latter restriction is that the set of functions currently allowed to operate on any class of data structures in MOQA must be a subset of those presented in Sections 2.2 and 3.3. These functions were designed for series-parallel data structures. This is one of the constraints that is always placed on each distinct $H_\beta$ that $c$ can possibly take at $i_{P,c}$. The following definition of a series-parallel data structure is a modified version of Stanley and Fomin's definition [69], which is used by Schellekens [63].

**Definition 11** (Series-parallel data structure)**.** *A series-parallel data structure is either empty or is obtained from one-node data structures through successive iterations of the operations of sequential and parallel composition.*

The operation of sequential composition connects one data structure above another. The operation of parallel composition places two data structures in parallel, i.e. side-by-side. In Section 2.2 it will become obvious that the MOQA product function is equivalent to the operation of sequential composition. To briefly counter illustrate, Figure 2.3 shows an example of a data structure that is not series-parallel. All other figures in this section show series-parallel data structures.

As well as MOQA specifying that the shape of each distinct $H_\beta$ that $c$ can possibly take at $i_{P,c}$ is series-parallel, it also specifies the distribution of the canonically-ordered labelings that can be on these $H_\beta$s. The MOQA rule is that, for each distinct series-parallel $H_\beta$ that $c$ can possibly take at $i_{P,c}$, a MOQA random structure is able to represent $H_\beta$ and the distribution of all of the canonically-ordered labelings that can be on $H_\beta$.

**Definition 12** (MOQA random structure)**.** *A MOQA random structure consists of a series-parallel $H_\beta$, a positive integer $M$ and a multiset containing $M$*

*copies of each canonically-ordered labeling in* $L(H_\beta)$[2]*.*

**Notation 10.** *Let* $S$ *denote a MOQA random structure.*

**Notation 11.** *Let* $n_S$ *denote the size of* $H_\beta$ *in* $S$*, i.e. the number of nodes in that* $H_\beta$*.*

**Definition 13** (Multiplicity of a MOQA random structure)**.** *The multiplicity of* $S$ *is the positive integer* $M$ *in Definition 12.*

**Notation 12.** *Let* $M^S$ *denote the multiplicity of* $S$*.*

So an unlabeled series-parallel Hasse diagram with label ordering $\beta$ can form the basis of a MOQA random structure if all its canonically-ordered labelings have equal likelihood of occurring, the likelihood being the multiplicity. This information is reflected in the multiset of a MOQA random structure, which contains all the possible labelings of the Hasse diagram. Like the series-parallel requirement for $H$, insisting that all the canonically-ordered labelings of $H_\beta$ have the same multiplicity is necessary because of how the average-case cost of a MOQA function is derived; see Section 2.3 for more detail.

When these restrictions apply, *all* of $c$'s possible states at $i_{P,c}$ can be stored in a MOQA random bag.

**Definition 14** (MOQA random bag)**.** *A MOQA random bag is a multiset of MOQA random structures.*

So the generalisation of the above MOQA rule is that a MOQA random bag is able to represent all of the states that $c$ can possible take at $i_{P,c}$. This means that the function applied next to $c$ must not leave it in such a condition where all of its possible states can no longer be stored in a MOQA random bag. Therefore, the function applied next must be a MOQA random structure preserving function.

**Definition 15** (A MOQA random structure preserving function)**.** *A function is MOQA random structure preserving if it maps a MOQA random structure to a multiset of one of more MOQA random structures.*

[2]The original definition of a MOQA random structure, see [63], does not include the positive integer $M$ as it is defined separately.

When a MOQA random structure preserving function is applied to each MOQA random structure in a MOQA random bag, then the results can be collected together in a MOQA random bag. Hence, a MOQA random structure preserving function is also known as a *MOQA random bag preserving function.*

**Definition 16** (MOQA-satisfying program)**.** *A MOQA-satisfying program is a program $P$ whose composite variables can store all of their possible states at any moment during $P$ in a MOQA random bag.*

In other words, all of the functions in a MOQA-satisfying program are MOQA random bag/structure preserving.

**Notation 13.** *Let $p$ denote a MOQA-satisfying program.*

An example of a program that is not MOQA-satisfying would be a program with a composite variable whose possible values at a particular moment are those of $H^a_{\beta_{max}}$ in the previous example on page 9. It would not be possible to represent those sixteen labelings with a MOQA random structure because the likelihood of each canonically-ordered labeling in $L(H^a_{\beta_{max}})$ occurring is not the same; there are three different likelihoods, i.e. zero, one or two, so there is no common multiplicity. Therefore, these states for that particular moment could not be stored in a MOQA random structure and hence, a MOQA random bag.

**Notation 14.** *Let $MRB_{p,\,c,\,\beta,\,i}$ denote the MOQA random bag that represents all of $c$'s possible states at $i_{p,\,c}$.*

**Notation 15.** *Let $\mathcal{M}$ denote $MRB_{p,\,c,\,\beta,\,i}$.*

**Notation 16.** *Let $MRB_{p,\,c,\,\beta}$ denote $c$'s MOQA random bag at the first moment that $c$ is referred to in $p$.*

What is in $MRB_{p,\,c,\,\beta}$? If $c$ refers to a variable that has been passed from another MOQA-satisfying program, then there may be any combination of MOQA random structures in $MRB_{p,\,c,\,\beta}$, depending on the behaviour of the other program. If $c$ does not refer to such a variable and $MRB_{p,\,c,\,\beta}$ was not provided along with $p$ as input to the MOQA static analysis tool, then the static assumption in Schellekens's work [63] is that $MRB_{p,\,c,\,\beta}$ contains one MOQA random structure. The $H_\beta$ of this MOQA random structure is a discrete Hasse diagram, i.e. a Hasse diagram with no edges, whose size is not fixed and its multiplicity is one. This will be discussed further in Section 4.3.

**Definition 17** (The size of a MOQA random bag)**.** *The size of a MOQA random bag is the number of MOQA random structures in it.*

**Notation 17.** *Let* $|\mathcal{M}|$ *denote the size of* $\mathcal{M}$*.*

The multiset of canonically-ordered labelings associated with a MOQA random structure does not need to be explicitly recorded for each $S$ in $\mathcal{M}$. If required, it would be possible to derive this information for any $S$ using its $H_\beta$ and $M^S$. The reader may also have observed that the notation for a MOQA random bag, $MRB_{p,c,\beta,i}$, includes $\beta$. That is not there to imply that all the MOQA random structures in a MOQA random bag *must* have the same label ordering. However, to date, this is what happens. So, for ease of notation, the common $\beta$ associated with each MOQA random structure in a MOQA random bag is recorded just once, in the notation for the bag itself. Hence, a MOQA random bag can be expanded to the following:

$$MRB_{p,c,\beta,i} \quad = \quad \{(S_1^{\mathcal{M}}, M^{S_1^{\mathcal{M}}}), (S_2^{\mathcal{M}}, M^{S_2^{\mathcal{M}}}), \ldots, (S_{|\mathcal{M}|}^{\mathcal{M}}, M^{S_{|\mathcal{M}|}^{\mathcal{M}}})\},$$

where $S_j^{\mathcal{M}}$ is the $j^{th}$ MOQA random structure in $\mathcal{M}$ and has a multiplicity of $M^{S_j^{\mathcal{M}}}$, $1 \leq j \leq |\mathcal{M}|$.

## 2.2 MOQA Functions

The terminology below is instrumental in defining the MOQA functions.

**Definition 18** (Minimal nodes)**.** *The minimal nodes in* $H$ *are the nodes in* $H$ *with no incoming edges.*

**Notation 18.** *Let* $m(H)$ *denote the set of minimal nodes in* $H$*.*

**Definition 19** (Maximal nodes)**.** *The maximal nodes in* $H$ *are the nodes in* $H$ *with no outgoing edges.*

**Notation 19.** *Let* $M(H)$ *denote the set of maximal nodes in* $H$*.*

**Notation 20.** *Let* $\lfloor x \rfloor$ *denote the set of all nodes that are directly below node* $x$ *in* $H$*.* [3]

[3] Any ambiguity with regard to the phrase " directly below" should be eliminated by the original definition of $\lfloor x \rfloor$: for a partial order $(X, \sqsubseteq)$ and an element $x \in X$, we define $\lfloor x \rfloor$ to be the set of all elements immediately and strictly below $x$ [63]. There is a similar original definition for $\lceil x \rceil$, which is to be discussed very shortly.

This is more informal definition than "for a partial order $(X, )$ and an element $x \in X$, we define $\lfloor x \rfloor$ to be the set of all elements immediately and strictly below $x$" [63].

**Notation 21.** *For any subgraph $I$ of $H$, let $\lfloor I \rfloor$ denote $\bigcup_{x \in I} \lfloor x \rfloor$.*

**Notation 22.** *Let $\lceil x \rceil$ denote the set of all nodes that are directly above node $x$ in $H$.*

**Notation 23.** *For any subgraph $I$ of $H$, let $\lceil I \rceil$ denote $\bigcup_{x \in I} \lceil x \rceil$.*

**Definition 20** (Isolated subset)**.** *A subgraph $I$ of $H$ is an isolated subset if it satisfies the following three conditions:*

1. $\lfloor I \setminus m(I) \rfloor \subseteq I$ *and* $\lceil I \setminus M(I) \rceil \subseteq I$.
2. $\forall x, y \in m(I) : \lfloor x \rfloor = \lfloor y \rfloor$.
3. $\forall x, y \in M(I) : \lceil x \rceil = \lceil y \rceil$.

Informally, the subgraph $I$ is an isolated subset if its minimal and maximal nodes are the only nodes in $I$ directly related to any nodes outside of $I$, and every minimal/maximal node of $I$ has the same set of nodes directly below/above it.

**Notation 24.** *$I\downarrow$ is the subgraph of $H$ that is composed of the nodes of subgraph $I$ of $H$ and those of $H$ that are below $I$, along with all the edges in $H$ that are between these nodes.*

**Notation 25.** *$I\uparrow$ is the subgraph of $H$ that is composed of the nodes of subgraph $I$ of $H$ and those of $H$ that are above $I$, along with all the edges in $H$ that are between these nodes.*

Note in the $I\downarrow/I\uparrow$ notation that the nodes of $H$ that are below/above $I$ are not *just* those that are *directly* below/above $I$.

**Definition 21** (Seam)**.** *A seam of $H$ is a pair $(A, B)$ of subgraphs of $H$ such that:*

1. *$A$ is completely below $B$, i.e. each node in $A$ is below all of the nodes in $B$.*

2. $(A\downarrow) \cup (B\uparrow) = H$.

**Definition 22** (Strictly isolated subset)**.** *The subgraph $I$ of $H$ is a strictly isolated subset if it satisfies the following three conditions:*

1. *$I$ is an isolated subset.*

2. *$(\lfloor m(I) \rfloor, m(I))$ is a seam of $H$.*

3. *$(M(I), \lceil M(I) \rceil)$ is a seam of $H$.*

Informally, the subgraph $I$ is a strictly isolated subset if it is an isolated subset and there are no nodes to either "side" of it. For example, the isolated subset comprised of $c$, $d$ and $e$ in Figure 2.2 is *not* strictly isolated because of the "side" node $b$. Note that the above seam and strictly isolated subset definitions aim to express the same concepts found in Schellekens's work [63] despite the differences in how they are formulated.

Note also that the empty subgraph is both isolated and strictly isolated, though the tendency is to assume that the subgraphs are not empty.

**Definition 23** (Connected component)**.** *The subgraph $I$ of $H$ is a connected component of $H$ if it is a maximal connected subgraph in $H$.*

**Notation 26.** *Let $A \,||\, B$ denote that the two disjoint Hasse diagrams represented by $A$ and $B$ are in parallel.*

Note that all connected components of $H$ are in parallel.

**Definition 24** (Label of rank $k$)**.** *The label of rank $k$ in a set of labels is the $k^{\text{th}}$ smallest label in that set of labels.*

Hence, the label of rank $k$ in the set of labels for a canonically-ordered labeling has the value $k$.

**Notation 27.** *Let $swap(x, y)$ denote the operation that swaps the labels of nodes $x$ and $y$.*

**Notation 28.** *Let $b(x)$ denote the label value on node $x$.*

**Notation 29.** *For the labeling $f$ on $H_\beta$, let $v(l_f, H_\beta)$ denote the node in $H_\beta$ that the label value $l_f$ is on.*

**Notation 30.** *For the labeling $f$ on $H_\beta$, let $\wedge(f, H_\beta)$ denote the minimum node in $H_\beta$, i.e. the node with the minimum label.*

**Notation 31.** *For the labeling $f$ on $H_\beta$, let $\vee(f, H_\beta)$ denote the maximum node in $H_\beta$, i.e. the node with the maximum label.*

The MOQA functions currently available are presented in the following subsections. Each one is MOQA random structure preserving. The *Extension Theorem* [63] proves that a MOQA random structure preserving function is such a function both when applied to an isolated subset of the $H_\beta$ represented by a MOQA random structure and when applied to the entire $H_\beta$ represented by a MOQA random structure.

However, the following subsections do not include all of the MOQA functions presented by Schellekens [63]. Three absentees, which are the MOQA top, bot and lift functions, were developed during the course of this research and therefore, are presented separately in Chapter 3 for the purpose of clearly delineating between existing work and this work.

### 2.2.1 MOQA Product

Let $H_\beta$ denote a series-parallel Hasse diagram with an isolated subset $I_\beta$ consisting of exactly two connected components, $A_\beta$ and $B_\beta$. The MOQA product function takes $A_\beta$ and $B_\beta$ and connects every minimal node of $A_\beta$ above every maximal node of $B_\beta$. Once this relationship has been established, it may be necessary to reorganise the labeling on $H_\beta$ so that it remains in accord with $\beta$. The MOQA product function assumes that $\beta$ is max-heap ordered, so this is achieved via the following sequence of steps:

1. For the labeling $f$ on $H_{\beta_{max}}$, let $min_f$ denote the smallest label in the set of labels on $m(A_{\beta_{max}})$, which of course will also be the smallest label in the set of labels on $A_{\beta_{max}}$, and let $max_f$ denote the largest label in the set of labels on $M(B_{\beta_{max}})$, which of course will also be the largest label in the set of labels on $B_{\beta_{max}}$. If $min_f < max_f$, then carry on to Step 2. Otherwise, stop because $f$ is consistent with $\beta_{max}$.

2. Swap the $min_f$ and $max_f$ labels between their nodes. In other words, `swap`$(v(min_f, I_{\beta_{max}}), v(max_f, I_{\beta_{max}}))$.

3. Apply the push-down logic of the following pseudo-code:

   `while` $\lfloor v(min_f, I_{\beta_{max}})\rfloor \subseteq I_{\beta_{max}}$ `and` $min_f < b(\vee(f, \lfloor v(min_f, I_{\beta_{max}})\rfloor))$
   `swap(`$v(min_f, I_{\beta_{max}})$`,` $\vee(f, \lfloor v(min_f, I_{\beta_{max}})\rfloor)$`)`

4. Apply the push-up logic of the following pseudo-code:

   `while` $\lceil v(max_f, I_{\beta_{max}})\rceil \subseteq I_{\beta_{max}}$ `and` $max_f > b(\wedge(f, \lceil v(max_f, I_{\beta_{max}})\rceil))$
   `swap(`$v(max_f, I_{\beta_{max}})$`,` $\wedge(f, \lceil v(max_f, I_{\beta_{max}})\rceil)$`)`

5. Go to Step 1.

Observe that the only labels reorganised by the MOQA product function are those on $I_{\beta_{max}}$. In other words, the MOQA product function does not need to adjust the labels on $H_{\beta_{max}} \setminus I_{\beta_{max}}$ to reconcile the entire labeling $f$, just those on $I_{\beta_{max}}$. This is because $I_{\beta_{max}}$ is an *isolated* subset of $H_{\beta_{max}}$. So the MOQA product function can view $I_{\beta_{max}}$ as independent of the portion of $H_{\beta_{max}}$ that surrounds it and this is why its "isolation" is advantageous.

**Notation 32.** *Let $A \otimes B$ denote that the Hasse diagram represented by $A$ has been producted, via the MOQA product function, above the Hasse diagram represented by $B$, when $A$ and $B$ are disjoint*[4]*.*

A MOQA product function example may be helpful at this point. In Figure 2.4, let $A$ be the Hasse diagram whose nodes are $a$ and $b$ and let $B$ be the Hasse diagram whose nodes are $c$ and $d$; note that nodes $c$ and $d$ have already been connected together by an earlier MOQA product function. Table 2.2 shows all the distinct canonically-ordered labelings of the overall Hasse diagram depicted in Figure 2.4. Figure 2.5 then shows the Hasse diagram $A \otimes B$ and Table 2.3 shows the labelings of Table 2.2 after they have been reorganised by this MOQA product function.

It is shown, see [63], that applying the MOQA product function to the isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure

[4]This is consistent with the implementation of the MOQA language in this work, *MOQA-Java*, and with the implementation of the MOQA static analysis tool, *Distri-Track* [35]. However, $A \otimes B$ means the reverse in the original MOQA theory [63], i.e. it means that the Hasse diagram represented by $B$ has been producted above the Hasse diagram represented by $A$.

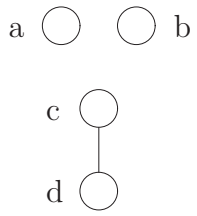



Figure 2.4: An unlabeled Hasse diagram

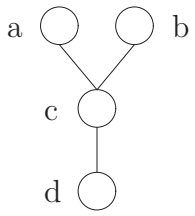



Figure 2.5: Figure 2.4 after a MOQA product function

| a | b | c | d |
|---|---|---|---|
| 1 | 2 | 4 | 3 |
| 1 | 3 | 4 | 2 |
| 1 | 4 | 3 | 2 |
| 2 | 1 | 4 | 3 |
| 2 | 3 | 4 | 1 |
| 2 | 4 | 3 | 1 |
| 3 | 1 | 4 | 2 |
| 3 | 2 | 4 | 1 |
| 3 | 4 | 2 | 1 |
| 4 | 1 | 3 | 2 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 2 | 1 |

Table 2.2: All distinct canonically-ordered labelings of Figure 2.4

| a | b | c | d |
|---|---|---|---|
| 4 | 3 | 2 | 1 |
| 4 | 3 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 4 | 3 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 3 | 4 | 2 | 1 |
| 4 | 3 | 2 | 1 |
| 4 | 3 | 2 | 1 |
| 4 | 3 | 2 | 1 |

Table 2.3: Table 2.2's labelings after reorganisation by the MOQA product function depicted in Figure 2.5

in the MOQA random bag $M$, results in the multiplication of $M^{S_j^{\mathcal{M}}}$ by:

$$\binom{|A_{\beta_{max}}| + |B_{\beta_{max}}|}{|B_{\beta_{max}}|}.$$

### 2.2.2 MOQA Split

Let $I_{\beta_{max}}$ denote a discrete isolated subset of the series-parallel $H_{\beta_{max}}$. Let $x$ denote a node of $I_{\beta_{max}}$. For any labeling on $H_{\beta_{max}}$, the MOQA split function 1), connects every node in $I_{\beta_{max}} \setminus x$ whose label is greater than the label on $x$ above $x$ and 2), connects every node in $I_{\beta_{max}} \setminus x$ whose label is smaller than the label on $x$ below $x$. The labeling then on $I_{\beta_{max}}$ has no need of modification because the nature of the MOQA split function ensures that it is correct. The motivation for $I_{\beta_{max}}$ being isolated has already been stated in Section 2.2.1.

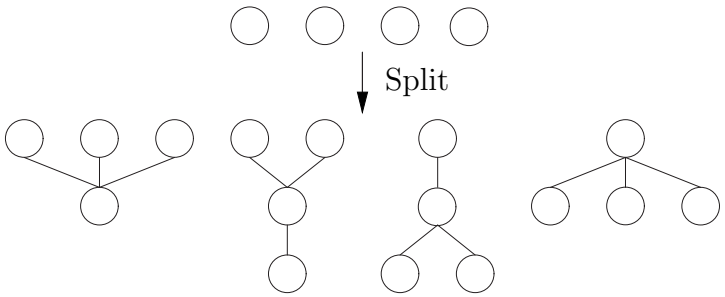



Figure 2.6: The four distinct Hasse diagrams that can result after the MOQA split function is applied to a discrete Hasse diagram whose size is four

For any labeling on $H_{\beta_{max}}$, one of $|I_{\beta_{max}}|$ distinct Hasse diagrams will result from the application of the MOQA split function to $I_{\beta_{max}}$: one for when the label on $x$ is the smallest in the set of labels on $I_{\beta_{max}}$, one for when the label on $x$ is the second smallest in the set of labels on $I_{\beta_{max}}$, etc. Figure 2.6 illustrates this by showing the four distinct Hasse diagrams that can result from the application of the MOQA split function to the discrete Hasse diagram of size four.

It is shown, see [63], that applying the MOQA split function to the discrete isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, results in the multiplication of $M^{S_j^{\mathcal{M}}}$ by:

$$\binom{|\lceil x \rceil| + |\lfloor x \rfloor|}{|\lfloor x \rfloor|}.$$

### 2.2.3 MOQA Deletion

Let $I_{\beta_{max}}$ denote a strictly isolated subset of the series-parallel $H_{\beta_{max}}$. Let the label of rank $k$ on $I_{\beta_{max}}$ denote the $k^{\text{th}}$ smallest label in the set of labels on $I_{\beta_{max}}$, $1 \leq k \leq |I_{\beta_{max}}|$. For any labeling on $H_{\beta_{max}}$, the MOQA deletion function deletes the label of rank $k$ from $I_{\beta_{max}}$ either upwards or downwards; this deletion of a label is accompanied by the deletion of a node. A distinctive characteristic of the MOQA deletion function is that, upon identification of the label to be deleted, the label to be deleted is viewed as either smaller, if the label is being pushed downwards, or larger, if the label is being pushed upwards, than any of the other labels on $I_{\beta_{max}}$. By this means the MOQA deletion function can move the label until it is on one of the extremal nodes in

$I_{\beta_{max}}$, which it then deletes. The following sequence of steps details how the label of rank $k$ on $I_{\beta_{max}}$ is deleted downwards:

1. For the labeling $f$ on $H_{\beta_{max}}$, let $min_f$ denote the $k^{\text{th}}$ smallest label in the set of labels on $I_{\beta_{max}}$ after it has been changed to some value less than $b(\wedge(f, I_{\beta_{max}}))$.

2. Apply the push-down logic of the following pseudo-code:

   `while` $\lfloor v(min_f, I_{\beta_{max}})\rfloor \subseteq I_{\beta_{max}}$ `and` $min_f < b(\vee(f, \lfloor v(min_f, I_{\beta_{max}})\rfloor))$
   `swap(`$v(min_f, I_{\beta_{max}})$`,` $\vee(f, \lfloor v(min_f, I_{\beta_{max}})\rfloor)$`)`

3. Now delete $v(min_f, I_{\beta_{max}})$, which is a minimal node.

The following sequence of steps details how the label of rank $k$ in $I_{\beta_{max}}$ is deleted upwards:

1. For the labeling $f$ on $H_{\beta_{max}}$, let $max_f$ denote the $k^{\text{th}}$ smallest label in the set of labels on $I_{\beta_{max}}$ after it has been changed to some value greater than $b(\vee(f, I_{\beta_{max}}))$.

2. Apply the push-up logic of the following pseudo-code:

   `while` $\lceil v(max_f, I_{\beta_{max}})\rceil \subseteq I_{\beta_{max}}$ `and` $max_f > b(\wedge(f, \lceil v(max_f, I_{\beta_{max}})\rceil))$
   `swap(`$v(max_f, I_{\beta_{max}})$`,` $\wedge(f, \lceil v(max_f, I_{\beta_{max}})\rceil)$`)`

3. Now delete $v(max_f, I_{\beta_{max}})$, which is a maximal node.

Informally, the MOQA deletion function deletes the label of rank $k$ from $I_{\beta_{max}}$ by pushing it downwards/upwards to a minimal/maximal node and then removing that node. $I_{\beta_{max}}$ being strictly isolated for MOQA deletion brings the same advantages as $I_{\beta_{max}}$ being isolated for MOQA product, i.e. the MOQA deletion function can ignore $H_{\beta_{max}} \setminus I_{\beta_{max}}$.

Table 2.4 shows all the distinct canonically-ordered labelings of the Hasse diagram depicted in Figure 2.5. Now consider the MOQA downward deletion of the label of rank four from this Hasse diagram and label set. The label of rank four is the fourth smallest label and so, in this example, is the largest label in the set. Figure 2.7 then shows the result of this MOQA deletion function and Table 2.5 shows the labelings of Table 2.4 after they have been reorganised by the function. Note that column $d$ is blank in Table 2.5 as this is the node that has just been deleted.

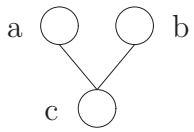



Figure 2.7: Figure 2.5 after a downwards MOQA deletion function

| a | b | c | d |
|---|---|---|---|
| 4 | 3 | 2 | 1 |
| 3 | 4 | 2 | 1 |

Table 2.4: All distinct canonically-ordered labelings of Figure 2.5

| a | b | c | d |
|---|---|---|---|
| 2 | 3 | 1 | - |
| 3 | 2 | 1 | - |

Table 2.5: Table 2.4's labelings after reorganisation by the MOQA deletion function depicted in Figure 2.7

It is shown, see [63], that applying the MOQA deletion function to the strictly isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, does not change the multiplicity of $M^{S_j^{\mathcal{M}}}$.

### 2.2.4 MOQA Projection

Let $I_\beta$ denote an isolated subset of the series-parallel $H_\beta$. For any labeling on $H_\beta$, the MOQA projection function makes a copy of $I_\beta$. As this function makes no changes to either the original $I_\beta$ or the copy $I_\beta$, the labeling on both is still in order, and multiplicity is unaffected. So the MOQA projection function never involves any label-to-label comparisons as it simply clones an isolated subset.

## 2.3 Average-case Cost in MOQA

Average-case behaviour is modular, or as Schellekens [63] puts it, it is IO-compositional. To illustrate, let $P$ denote a program that consists of just two independent calls, the first to program $Q$ and the second to program $R$. Average-case behaviour is said to be *IO-compositional* because the average-case behaviour of $P$ is the average-case behaviour of $Q$ plus the average-case behaviour of $R$. Unlike average-case behaviour, this is not always true for either best-case or worst-case behaviour; this point is further discussed in Chapter 6

in its review of related works that have come to the same conclusion.

All MOQA functions are MOQA random bag/structure preserving, i.e. they output a non-empty multiset of MOQA random structures when a MOQA random structure is the input. MOQA limits function input to a MOQA random structure so that its formula for determining the average number of label-to-label comparisons[5] can correctly assume that all the canonically-ordered labelings of the MOQA random structure's $H_\beta$ are equally likely. The formula operates by iterating once through the shape of $H_\beta$ to determine the average number of label-to-label comparisons, so combining this assumption with the additional assumption that $\beta$ is max-heap ordered enables a key formula expectation to be accurate. This expectation is that *the average-case formula is equally likely to take each path at any branching point reached during its traversal of* $H_\beta$ and this allows for the complexity of these MOQA formulas to be greatly reduced. Furthermore, the output of a MOQA function is arranged to be acceptable input for the next MOQA function. Hence, the restriction on MOQA function input and output simplifies the static analysis.

An important point just noted here and also earlier, which should be retained by the reader throughout this work, is that both the MOQA functions and the mathematical techniques employed by MOQA assume that $\beta$ is max-heap ordered; an assumption so ingrained that it is actually hard-coded into the MOQA methodology. Though the MOQA approach would also apply if $\beta$ is min-heap ordered, max-heap ordered is taken to be the default $\beta$ value. This will continue to be the default for the new research presented in coming chapters.

Deeper attention will now be given to how MOQA reckons the average-case cost of its functions for a MOQA random structure before moving on to the average-case cost of its functions for a MOQA random bag. When a MOQA function is applied to the $H_{\beta_{max}}$ of a MOQA random structure, then its MOQA formula for statically calculating the average number of label-to-label comparisons generally involves subsidiary equations, which are known as *composition laws*. These composition laws are divided into four groups, $\sigma$, $\kappa$, $\tau$ and $\Delta$, and the equations of each group measure a specific property. All of

[5]At future times the term "label-to-label comparisons" may be abbreviated to simply "comparisons". However, the context should still indicate that the comparisons being referred to are of type label-to-label.

these equations have a parameter in common, the series-parallel $H_{\beta_{max}}$.

Let $A_{\beta_{max}}$ and $B_{\beta_{max}}$ denote two disjoint and non-empty series-parallel $H_{\beta_{max}}$s. So, when the size of the series-parallel $H_{\beta_{max}}$ parameter for one of these composition laws is greater than one, then the parameter becomes binary, i.e. it is a function of two variables as it is equivalent to either $A_{\beta_{max}} \otimes B_{\beta_{max}}$ or $A_{\beta_{max}} \,||\, B_{\beta_{max}}$. Let $\bullet_{\beta_{max}}$ denote the series-parallel $H_{\beta_{max}}$ whose size is one. So, when the size of the series-parallel $H_{\beta_{max}}$ parameter for one of these composition laws is one, then $\bullet_{\beta_{max}}$ is the parameter.

The $\sigma_{up}$, $\kappa_{up}$ and $\tau_{up}$ (but not the $\Delta_{up}$) equations given below are all based on the assumption that the minimum label on the series-parallel $H_{\beta_{max}}$ parameter has just been replaced with a new label, which will be of rank $k$ in the set of labels now on $H_{\beta_{max}}$, $1 \leq k \leq |H_{\beta_{max}}|$. (Recall that the label of rank $k$ in a set of labels is the $k^{\text{th}}$ smallest label in that set of labels.) Such label replacement takes place, for example, during the MOQA product function.

There is now enough background to present equations from each of the four composition law groups. These equations are taken from Schellekens [63].

The first composition law group is $\boldsymbol{\sigma(H_{\beta_{max}})}$. $\sigma_{up}(H_{\beta_{max}})$ is the average number of label-to-label comparisons that it takes to push the new label on the series-parallel $H_{\beta_{max}}$ from a minimal node *up* to a maximal node when the new label is of rank $|H_{\beta_{max}}|$ and the average is taken over every canonically-ordered labeling in $L(H_{\beta_{max}})$, which is the set of all canonically-ordered labelings of the Hasse diagram $H$ that has the label ordering $\beta$ on it.

$$\sigma_{up}(A_{\beta_{max}} \,||\, B_{\beta_{max}}) = \frac{|A_{\beta_{max}}|.\sigma_{up}(A_{\beta_{max}}) + |B_{\beta_{max}}|.\sigma_{up}(B_{\beta_{max}})}{|A_{\beta_{max}}| + |B_{\beta_{max}}|} \quad (2.1)$$

$$\sigma_{up}(A_{\beta_{max}} \otimes B_{\beta_{max}}) = \sigma_{up}(A_{\beta_{max}}) + \sigma_{up}(B_{\beta_{max}}) + |m(A_{\beta_{max}})| \quad (2.2)$$

$$\sigma_{up}(\bullet_{\beta_{max}}) = 0 \quad (2.3)$$

The second composition law group is $\boldsymbol{\kappa(H_{\beta_{max}})}$. $\kappa_{up}(H_{\beta_{max}})$ is the average number of times that the new label on the series-parallel $H_{\beta_{max}}$ is pushed *up* as far as a maximal node. In other words, let $y_k$ denote the average number of times that the new label on $H_{\beta_{max}}$ is pushed up as far as a maximal node when the new label is of rank $k$ and the average is taken over every canonically-

ordered labeling in $L(H_{\beta_{max}})$. Therefore, $\kappa_{up}(H_{\beta_{max}}) = \sum_{k=1}^{|H_{\beta_{max}}|} y_k$.

$$\kappa_{up}(A_{\beta_{max}} \,||\, B_{\beta_{max}}) = \kappa_{up}(A_{\beta_{max}}) + \kappa_{up}(B_{\beta_{max}}) \tag{2.4}$$

$$\kappa_{up}(A_{\beta_{max}} \otimes B_{\beta_{max}}) = \kappa_{up}(A_{\beta_{max}}) \tag{2.5}$$

$$\kappa_{up}(\bullet_{\beta_{max}}) = 1 \tag{2.6}$$

The third composition law group is $\boldsymbol{\tau(H_{\beta_{max}})}$. $\tau_{up}(H_{\beta_{max}})$ is the average number of label-to-label comparisons that it takes to push the new label on the series-parallel $H_{\beta_{max}}$ from a minimal node *up* to its correct position. In other words, let $x_k$ denote the average number of label-to-label comparisons that it takes to push the new label on $H_{\beta_{max}}$ from a minimal node up to its correct position when the new label is of rank $k$ and the average is taken over every canonically-ordered labeling in $L(H_{\beta_{max}})$. Therefore, $\tau_{up}(H_{\beta_{max}}) = (\sum_{k=1}^{|H_{\beta_{max}}|} x_k)/|H_{\beta_{max}}|$.

$$\tau_{up}(A_{\beta_{max}} \,||\, B_{\beta_{max}}) = \frac{|A_{\beta_{max}}|.\tau_{up}(A_{\beta_{max}}) + |B_{\beta_{max}}|.\tau_{up}(B_{\beta_{max}})}{|A_{\beta_{max}}| + |B_{\beta_{max}}|} \tag{2.7}$$

$$\begin{aligned}\tau_{up}(A_{\beta_{max}} \otimes B_{\beta_{max}}) = {} & (|B_{\beta_{max}}|.\tau_{up}(B_{\beta_{max}}) + \kappa_{up}(B_{\beta_{max}}).|m(A_{\beta_{max}})| + \\ & |A_{\beta_{max}}|.(\tau_{up}(A_{\beta_{max}}) + |m(A_{\beta_{max}})| + \sigma_{up}(B_{\beta_{max}})))/ \\ & |A_{\beta_{max}}| + |B_{\beta_{max}}|\end{aligned} \tag{2.8}$$

$$\tau_{up}(\bullet_{\beta_{max}}) = 0 \tag{2.9}$$

The fourth composition law group is $\boldsymbol{\Delta(H_{\beta_{max}}, k)}$. $\Delta_{up}(H_{\beta_{max}}, k)$ is the average number of label-to-label comparisons that it takes to delete the label of rank $k$ on the series-parallel $H_{\beta_{max}}$ by pushing it *up* to a maximal node and then removing that node, when the average is taken over every canonically-ordered labeling in $L(H_{\beta_{max}})$.

$$\begin{aligned}\Delta_{up}(A_{\beta_{max}} \,||\, B_{\beta_{max}}, k) \;=\; & \Bigg(\Bigg(\sum_{i=1}^{|B_{\beta_{max}}|} \binom{k-1}{i-1}.\binom{|A_{\beta_{max}}|+|B_{\beta_{max}}|-k}{|B_{\beta_{max}}|-i}. \\ & \Delta_{up}(B_{\beta_{max}}, i)\Bigg) + \Bigg(\sum_{i=1}^{|A_{\beta_{max}}|} \binom{k-1}{i-1}. \\ & \binom{|A_{\beta_{max}}|+|B_{\beta_{max}}|-k}{|A_{\beta_{max}}|-i}.\Delta_{up}(A_{\beta_{max}}, i)\Bigg)\Bigg)/ \\ & \binom{|A_{\beta_{max}}|+|B_{\beta_{max}}|}{|B_{\beta_{max}}|} \end{aligned} \tag{2.10}$$

$$\Delta_{up}(A_{\beta_{max}} \otimes B_{\beta_{max}}, k) \;=\; \begin{cases} \Delta_{up}(A_{\beta_{max}}, k-|B_{\beta_{max}}|) & \text{if } k > |B_{\beta_{max}}| \\ \Delta_{up}(B_{\beta_{max}}, k) + |m(A_{\beta_{max}})| & \\ -1 + \Delta_{up}(A_{\beta_{max}}, |A_{\beta_{max}}|) & \text{if } k \leq |B_{\beta_{max}}| \end{cases}$$

$$\Delta_{up}(\bullet_{\beta_{max}}) \;=\; 0 \tag{2.11}$$

The $\sigma_{down}$, $\kappa_{down}$ and $\tau_{down}$ (but not the $\Delta_{down}$) equations are all based on the assumption that the maximum label on the series-parallel $H_{\beta_{max}}$ parameter has just been replaced with a new label, which will be of rank $k$ in the set of labels now on $H_{\beta_{max}}$. For example, $\sigma_{down}(H_{\beta_{max}})$ is the average number of label-to-label comparisons that it takes to push the new label on the series-parallel $H_{\beta_{max}}$ from a maximal node *down* to a minimal node when the new label is of rank 1 and the average is taken over every canonically-ordered labeling in $L(H_{\beta_{max}})$. Hence, the equations for $\sigma_{down}$, $\kappa_{down}$, $\tau_{down}$ and $\Delta_{down}$ will be similar to those above.

**Notation 33.** *Let $f$ denote a MOQA function.*

**Notation 34.** *Let $\overline{T}_f(H_{\beta_{max}})$ denote the average number of label-to-label comparisons that result when the MOQA function $f$ is applied to the series-parallel $H_{\beta_{max}}$, when the average is taken over every labeling in the canonically-ordered set $L(H_{\beta_{max}})$.*

So Schellekens's formula [63] for the average-case cost of the MOQA product function when applied to the $H_{\beta_{max}}$ of a MOQA random structure demon-

strates the application of these composition laws:

$$\begin{aligned}\overline{T}_{prod}(A_{\beta_{max}} \,||\, B_{\beta_{max}}) &= \frac{|A_{\beta_{max}}|.|B_{\beta_{max}}|}{|A_{\beta_{max}}| + |B_{\beta_{max}}|}.(\tau_{down}(B_{\beta_{max}}) + \tau_{up}(A_{\beta_{max}})) + \\ &\quad \left(\frac{|A_{\beta_{max}}|.|B_{\beta_{max}}|}{|A_{\beta_{max}}| + |B_{\beta_{max}}|} + 1\right).(|M(B_{\beta_{max}})| + \\ &\quad |m(A_{\beta_{max}})| - 1).\end{aligned}$$

Note that $A_{\beta_{max}} \,||\, B_{\beta_{max}}$ will have been transformed into $A_{\beta_{max}} \otimes B_{\beta_{max}}$ upon completion of the MOQA product function.

The formula for the average-case cost of the MOQA deletion function that deletes the label of rank $k$ upwards through the $H_{\beta_{max}}$ of a MOQA random structure is simply:

$$\overline{T}_{deleteUp}(H_{\beta_{max}}, k) = \Delta_{up}(H_{\beta_{max}}, k).$$

However, a MOQA function's average-case formula is not required to include the above composition laws, which has been established by Schellekens's average-case formula [63] for the MOQA split function. The formula for the average-case cost of the MOQA split function when applied to a node in the discrete $H_{\beta_{max}}$ of a MOQA random structure is:

$$\overline{T}_{split}(H_{\beta_{max}}) = |H_{\beta_{max}}| - 1.$$

In general, for each MOQA function $f$, the MOQA theory should provide the formula for $\overline{T}_f(H_{\beta_{max}})$ when $H_{\beta_{max}}$ is the appropriate series-parallel data structure. (Note that, when discussing the average-case cost of a MOQA function in this work, the term "formula" will refer to the total average-case cost and the term "equation" will refer to the composition laws involved.)

After presenting the means by which MOQA determines the average-case cost of a MOQA function when applied to the $H_{\beta_{max}}$ of a MOQA random structure, attention can be turned to how MOQA determines this average-case information when the MOQA function is applied to a MOQA random bag.

**Notation 35.** *Let $\overline{T}_f(S_j^{\mathcal{M}})$ denote $\overline{T}_f(H_{\beta_{max}})$ for the $H_{\beta_{max}}$ of that $S_j^{\mathcal{M}}$.*

Recall that $S_j^{\mathcal{M}}$ denotes the $j^{th}$ MOQA random structure in the MOQA

random bag $M$.

**Notation 36.** *Let $\overline{T}_f(\mathcal{M})$ denote the average number of label-to-label comparisons that result when the MOQA function $f$ is applied to the MOQA random bag $\mathcal{M}$.*

As shall be seen shortly, it is necessary to work out the relative frequency of each $S_j^{\mathcal{M}}$ occurring in $\mathcal{M}$ if the MOQA static analysis tool is to calculate $\overline{T}_f(\mathcal{M})$.

**Notation 37.** *Let $L(S_j^{\mathcal{M}})$ denote $L(H_{\beta_{max}})$ for the $H_{\beta_{max}}$ of that $S_j^{\mathcal{M}}$.*

**Notation 38.** *Let $l(\mathcal{M})$ denote the multiset union of each $S_j^{\mathcal{M}}$'s multiset of canonically-ordered labelings.*

Therefore, $|l(\mathcal{M})| = \sum_{j=1}^{|\mathcal{M}|} |L(S_j^{\mathcal{M}})|.M^{S_j^{\mathcal{M}}}$. So $\frac{|L(S_j^{\mathcal{M}})|.M^{S_j^{\mathcal{M}}}}{|l(\mathcal{M})|}$ is the relative frequency of $S_j^{\mathcal{M}}$ occurring in $\mathcal{M}$.

Though the $|L(S_j^{\mathcal{M}})|$ term could be established through generating $L(S_j^{\mathcal{M}})$ and then counting its size, a more efficient solution was devised. In like manner to the composition laws, this solution takes advantage of the fact that $H_{\beta_{max}}$ is in series-parallel and has the $\beta_{max}$ label ordering on it and therefore, only needs to iterate once through the shape of $H_{\beta_{max}}$ to get $|L(H_{\beta_{max}})|$. The equations for this approach, see [63], are as follows.

If $H_{\beta_{max}} = H_{\beta_{max},1} \otimes \ldots \otimes H_{\beta_{max},n}$, then:

$$|L(H_{\beta_{max}})| = \prod_{i=1}^{n} |L(H_{\beta_{max},i})|. \tag{2.12}$$

If $H_{\beta_{max}} = H_{\beta_{max},1} \parallel \ldots \parallel H_{\beta_{max},n}$, then:

$$|L(H_{\beta_{max}})| = \binom{|H_{\beta_{max}}|}{|H_{\beta_{max},1}| \ldots |H_{\beta_{max},n}|} . \prod_{i=1}^{n} |L(H_{\beta_{max},i})|, \tag{2.13}$$

where

$$\binom{|H_{\beta_{max}}|}{|H_{\beta_{max},1}| \ldots |H_{\beta_{max},n}|} = \binom{|H_{\beta_{max}}|}{|H_{\beta_{max},1}|} . \binom{|H_{\beta_{max}}| - |H_{\beta_{max},1}|}{|H_{\beta_{max},2}|} \ldots \binom{|H_{\beta_{max}}| - \sum_{i=1}^{n-1} |H_{\beta_{max},i}|}{|H_{\beta_{max},n}|}.$$

Accordingly, $\overline{T}_f(\mathcal{M})$ is an amalgamation of the average-case cost of $f$ when applied to each $S_j^{\mathcal{M}}$ in $\mathcal{M}$ and the relative frequency of each $S_j^{\mathcal{M}}$ occurring in $\mathcal{M}$. As a formula, this is:

$$\overline{T}_f(\mathcal{M}) = \sum_{j=1}^{|\mathcal{M}|} \frac{|L(S_j^{\mathcal{M}})|.M^{S_j^{\mathcal{M}}}}{|l(\mathcal{M})|}.\overline{T}_f(S_j^{\mathcal{M}}). \tag{2.14}$$

Note that when there is only one MOQA random structure in $\mathcal{M}$, then $\overline{T}_f(\mathcal{M})$ is simply $\overline{T}_f(S_1^{\mathcal{M}})$.

## 2.4 MOQA Algorithms

This exploration of the MOQA theory—realised through statically calculating the average-case behaviour of MOQA functions—concludes with listing the algorithms that it is capable of automatically analysing. There are four such algorithms presented in the parental MOQA research [63]: insertion-sort, quicksort/quickselect, mergesort and treapsort. Later discussions will again refer to these algorithms but, for now, their descriptions can be found at the following locations:

- Figure 3.5 on page 46 shows the insertion-sort algorithm, as implemented in *MOQA-Java*. (*MOQA-Java* is the MOQA language implementation developed during this research; it is introduced in Chapter 3.)
- Figure 3.6 on page 49 shows the quicksort algorithm, as implemented in *MOQA-Java*.
- Figure 6.1 on page 189 shows the mergesort algorithm, as implemented in *MOQA-Java*.
- Appendix A shows and explains the pseudo-code for the treapsort algorithm, whose inspiration is the heapsort algorithm.

## 2.5 Chapter Summary

This chapter presented a summary of the MOQA theory, which should be sufficient for comprehending the work to come. The language implementation

of this theory is dealt with in the following chapter. Note that, unless stated otherwise, the work from this point onwards has been independently developed by the author.

# Chapter 3

# The MOQA Language

This chapter concentrates on the MOQA language, with special attention being given to practicality. The design for the current implementation of the MOQA language is detailed, along with the raison d'être for many of the design choices. Next, an examination of implementation costs, specifically space and average-case cost, is carried out. Subsequently, new MOQA functions are described and their existence justified. This chapter then concludes with a study of MOQA language reversibility.

## 3.1 The MOQA Language Implementation

The MOQA language implementation discussed in this section is based on the theory [63] summarised in Chapter 2[1]. There is a clear delineation to be made between *MOQA-Java*, the current implementation of the MOQA language, and *Distri-Track* [35], the current implementation of the MOQA static analysis tool. The former is the language in which MOQA-satisfying programs are written and executed. The later is a tool whose input is a MOQA-satisfying program written in *MOQA-Java*, and which then parses and analyses this code to output its average-case behaviour. So the execution of a *MOQA-Java* program is simply one run-time of that *MOQA-Java* program whereas the execution of a *Distri-Track* program is one run-time of that *Distri-Track* program which considers all the run-times of a particular *MOQA-Java* program, thus motivating the MOQA random structure/bag methodology.

[1]Hence, the new data structure types in Chapter 4 are not part of the current MOQA language implementation.

So *Distri-Track* supplies the timing information for a *MOQA-Java* program without executing it. Generally, however, the purpose of writing code is to run it and an aim of MOQA is for programmers to write components of their real-world applications in *MOQA-Java* so that they can benefit from information about the behavioural nature of the code they have just written. Therefore, the *MOQA-Java* syntax should encourage programmers to write code that conforms to the MOQA theory so that the code can be completely and correctly analysed by *Distri-Track*. This was a strong motivation behind some of the *MOQA-Java* design choices.

*MOQA-Java* is in the form of a Java 5.0 package[2], thereby presenting a new paradigm in a familiar setting. A programmer will import the *MOQA-Java* package if s/he wishes to program in *MOQA-Java*; modifications to the *MOQA-Java* package are only undertaken by implementers of the MOQA language. However, *MOQA-Java* should not be seen as a Java extension, quite the opposite in fact, as the MOQA theory restricts some of the basic constructs in the core Java library. The theory limits the range of conditional expressions allowed in if statements to first-order and second-order conditional expressions; a *first-order conditional expression* compares label values and a *second-order conditional expression* compares the size of a data structure to a closed arithmetical expression. It also completely excludes while statements because statically determining how often they iterate can be problematic, which motivates other static analysis timing tools to bound the number of while loop cycles [34]. Though while statements are precluded from MOQA, Hickey [35] explores which instances could be open to analysis by a future MOQA static analysis tool.

Furthermore, only the classes made available by the *MOQA-Java* package should be present in the code submitted for evaluation to the MOQA static analysis tool. The sole purpose of the MOQA static analysis tool is to track the average-case cost of the MOQA functions that are applied to the MOQA data structure, i.e. the series-parallel data structure. This means that classes outside of the *MOQA-Java* package are outside of MOQA's remit. Therefore, while *MOQA-Java* code has Java's syntax and object-oriented design, it cannot be mixed up amongst other Java code if it is to statically analysed. So it may be more appropriate to view *MOQA-Java* as a modest language in the Java

[2]Java 5.0 was the latest version of Java at the time of development.

syntax. Thus, *MOQA-Java* code should be a stand-alone entity that can be employed by, but does not depend on, non-*MOQA-Java* code.

The above restrictions on control flow and classes are not enforced by the *MOQA-Java* package. Any breach of these regulations would have to be detected by the MOQA static analysis tool. Hickey [35] details the restrictions, control flow in particular, on the *MOQA-Java* code that *Distri-Track* currently requires for an analysis to successfully complete.

### 3.1.1 *MOQA-Java* Details

Figure 3.1 is the *MOQA-Java* class diagram. Its graphic notation follows the standard UML graphic notation and is as follows[3]. Let <<TypeName>> denote that TypeName is an interface. Let *TypeName* denote that TypeName is an abstract class. Let TypeName denote that TypeName is a standard class. Let the hexagon at one end of a solid line denote that there is a composition relationship between two classes. Let the number at one end of a solid line denote multiplicity. For example, in Figure 3.1 a NodeInfo instance will always have one Label instance. Inheritance is indicated by a solid line with an unfilled arrowhead pointing at the super class. A class that realises/implements the behaviour of an interface is indicated by a dotted line with an unfilled arrowhead pointing at the interface.

The label ordering on the MOQA data structure is hard-coded into *MOQA-Java* and, in line with the definitions of the MOQA functions, is the max-heap label ordering. (In the future, *MOQA-Java* could allow the programmer to specify whether the label ordering is min-heap or max-heap, which would then lead to the selection of either the min-heap or max-heap versions of the MOQA functions.) The classes in Figure 3.1 will now be explained.

The MOQA data structure is represented in *MOQA-Java* by an instance of the LPO (Labeled Partial Order) class and the subsets of a MOQA data structure, which are created and then returned by MOQA functions, are instances of the SubLPO class. So the SubLPO class, in conjunction with the Node class discussed below, represents part of the MOQA data structure and in particular, the part it represents is an isolated subset. An instance of the SubLPO class is only ever created to reflect a structural change within a MOQA data struc-

[3]The attributes and operations of each class in Figure 3.1 are excluded for ease of reading.

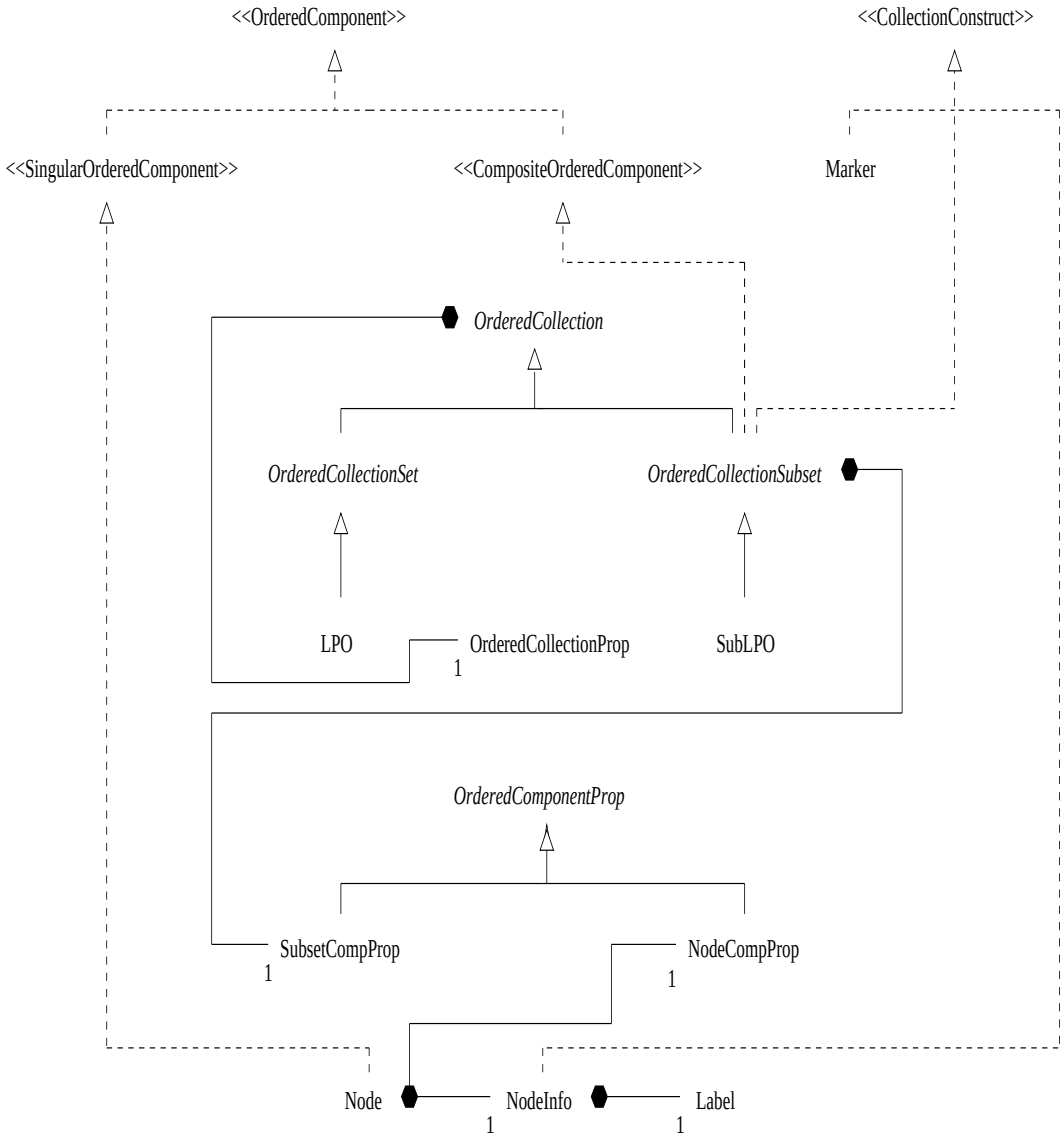



Figure 3.1: The *MOQA-Java* class diagram

ture/LPO instance. As can be seen from Figure 3.1, the LPO class extends the OrderedCollectionSet abstract class and the SubLPO class extends the OrderedCollectionSubset abstract class. Through the OrderedCollectionSubset class, the SubLPO class inherits the OrderedComponent interface. Both of these abstract classes extend the OrderedCollection abstract class, which represents a collection whose elements are of type OrderedComponent. In general, an instance of the SubLPO class will be directly contained within an OrderedCollection instance and it will be a collection of elements that had the same set of elements above them and the same set of elements below them within that OrderedCollection instance; an OrderedComponent instance is *directly contained* within an OrderedCollection instance when it is not nested within any other OrderedComponent instance within that OrderedCollection instance and yet is still within that OrderedCollection instance.

A point to consider is that a programmer in *MOQA-Java* can never instantiate a SubLPO. This class can only be instantiated internally within the *MOQA-Java* package and, as already stated, is done so during the execution of a MOQA function. Why is this not an option for a programmer in *MOQA-Java*? Chapter 2 states the MOQA requirements for statically determining the average-case behaviour of a function that is applied to a data structure. First, the data structure is a MOQA data structure whose states are restricted to a particular distribution and second, any function applied to the MOQA data structure must preserve this distribution and hence, be a MOQA random structure preserving function. So a MOQA data structure should be exclusively modified by the MOQA functions. Therefore, a programmer in *MOQA-Java* should not modify a LPO instance outside of the MOQA functions. As a SubLPO instance is only ever created to reflect a structural change within a LPO instance, a programmer in *MOQA-Java* should not bring one into being independently of the MOQA functions. This constraint is enforced by the design of *MOQA-Java* prohibiting a programmer from creating a SubLPO instance. In addition to being returned by MOQA functions, the SubLPO instances from which a LPO instance is composed can also be accessed via an Iterator; see Section 3.1.2 for method details.

It can be seen from the *MOQA-Java* examples in this work, beginning with Figure 3.2, that the addition of generics to Java 5.0 is used to specify the type of label value on the elements contained within a LPO instance and hence, on

```
LPO<Integer> lpo1 = new LPO<Integer>();
LPO<Integer> lpo2 = new LPO<Integer>();

NodeInfo<Integer> minusOne = new NodeInfo<Integer>(
    new Label<Integer>(-1), ``minusOne'');
NodeInfo<Integer> zero = new NodeInfo<Integer>(0, ``zero'');
NodeInfo<Integer> one = new NodeInfo<Integer>(1, ``one'');

lpo1.add(minusOne);
lpo1.add(zero);
lpo1.add(one);

lpo2.add(minusOne);
lpo2.add(zero);
```

Figure 3.2: Adding label values to MOQA data structures in *MOQA-Java* — the first argument to a NodeInfo's constructor is a Label object / label value and the second argument is the data associated with that Label object / label value

the elements contained within all of its SubLPO instances.

As indicated by Figure 3.2, the NodeInfo class is used to populate a LPO. In other words, labels values are added to the MOQA data structure by means of the NodeInfo class. So each instance of the NodeInfo class has exactly one instance of the Label class. A Label instance stores a label value. The Label instance that belongs to a NodeInfo instance is either supplied as an argument to the NodeInfo constructor or is created by the NodeInfo constructor for the label value that is supplied as an argument to it. Any data associated with the label value of the Label instance is stored in the NodeInfo instance. There are no restrictions on the associated data. For example, it could be the medical history of some patient whose unique identifier is the corresponding label value.

While the data associated with the label value can be modified at any time, the label value is constant because modifying it can introduce discord between the labeling on the MOQA data structure and the obligatory max-heap label ordering. What about implementing a function that readjusts a modified labeling? One impediment to this solution is the development of a MOQA average-case formula for such a function and without the formula there would be a cost that is not factored into the timing result generated by the MOQA static analysis tool. There is another serious impediment to consider, which is that all of the MOQA data structure's canonically-ordered labelings

may no longer be equally likely when a label value change necessitates labeling rearrangement. So rearrangement of the labeling could result in a distribution of MOQA data structure state that differs from the one dealt with by the MOQA average-case formulas and therefore, would result in the MOQA static analysis tool producing a faulty average-case recurrence when other MOQA functions follow the label value change. Hence, *MOQA-Java* denies label value modification after its initial assignment in a Label instance.

As demonstrated in Figure 3.2, the same NodeInfo instance can be added to one or more LPO instances, i.e. multiple LPO instances can store the same label value and data pair. (However, all new elements must be added to a MOQA data structure prior to applying the first MOQA function [63]. So, because *MOQA-Java* is built on the MOQA theory [63], a NodeInfo instance can be added to a LPO instance but not to a SubLPO instance, as it is the application of a MOQA function that triggers the instantiation of a SubLPO. Additionally, the NodeInfo instance can only be added to a LPO instance if the LPO instance is discrete, that is, it is at "starting" state. The adaptation of the MOQA functions in Section 3.3 would remove these restrictions, and hence the cost incurred from verifying LPO discreteness, from future MOQA language implementations.) The LPO instance creates a Node instance for each NodeInfo instance added to it so the Node class has a one-to-many relationship with the LPO class. The Node class has package-level visibility which means that code outside of the *MOQA-Java* package is unaware of the Node class. (The Node class has reduced visibility because the algorithms currently written in *MOQA-Java* can accomplish their tasks without access to this class. So the current version of the *MOQA-Java* package keeps the number of public classes to a minimum for the sake of simplicity. However, if later work concludes that programmers in *MOQA-Java* should have access to the Node class because such access would make programming in *MOQA-Java* more practical, then the visibility of the Node class should be increased.) Like the OrderedCollectionSubset class, the Node class inherits the OrderedComponent interface so it is of the type that can be contained within an OrderedCollection.

The purpose of the Node class is to record the series-parallel relationship between a Node instance and the other OrderedComponent instances that are also directly contained within the same OrderedCollection instance. As the purpose of the NodeInfo class is to record a label value and data pair, the Node

and NodeInfo design give a measure of separation between the series-parallel data structure and the labeling on it. Thus, while a NodeInfo instance can be in multiple LPO instances, a Node instance is unique to a single LPO instance because it represents one element in the MOQA data structure represented by that LPO instance. Note that a NodeInfo instance is not bound to its initial Node instance. The MOQA functions will swap NodeInfo instances between Node instances during their adjustment of the labeling on the MOQA data structure.

It has been observed that both the Node class and the SubLPO class inherit the OrderedComponent interface and that it is objects of this type that are collected within a LPO/SubLPO instance. The moment at which a Node instance comes into existence has just been explained but when exactly does a SubLPO instance come into existence? What structural change requires some of the OrderedComponent instances directly contained within an OrderedCollection instance to be moved into a new SubLPO instance for that OrderedCollection instance? (Recall that only the OrderedComponent instances that have the same set of elements directly above them and the same set of elements directly below them within that OrderedCollection instance can be moved into its new SubLPO instance.) *MOQA-Java* abides by its rule that the OrderedComponent instances directly contained within an OrderedCollection instance are always in parallel. There is one permissible exception to this rule, that of an OrderedCollectionSubset instance with exactly two OrderedComponent instances directly contained within it. Such OrderedComponent instances can also be in series. So a SubLPO instance is created during the execution of a MOQA function to ensure that the entire LPO instance continues to adhere to this *MOQA-Java* rule. Figure 3.3 gives an example of SubLPO instantiation; in this figure a LPO instance is represented by a solid line box and a SubLPO instance is represented by a dotted line box. In Figure 3.3, LPO I illustrates how a discrete partial order of size seven is represented in *MOQA-Java* and LPO II illustrates the *MOQA-Java* representation of LPO I after a MOQA product function involving five elements is applied to it. Note that the outermost SubLPO instance in LPO II has been added to maintain LPO parallelism and that its direct content can be in series because it consists of exactly two OrderedComponent instances.

One benefit from *MOQA-Java* representing the series-parallel nature of

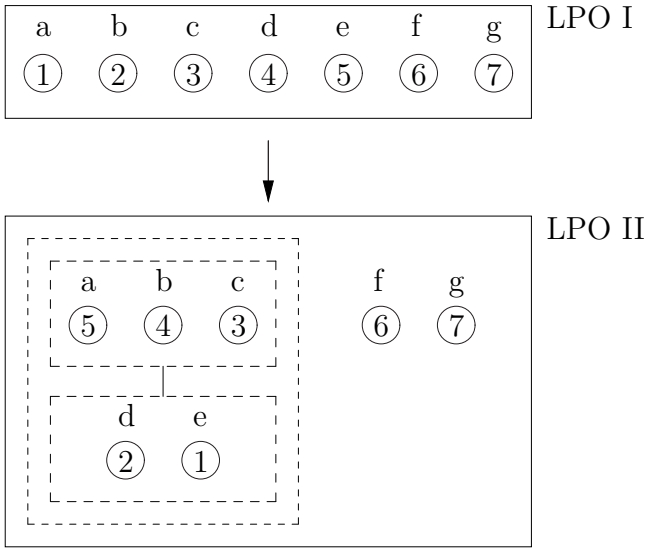



Figure 3.3: LPO I, the *MOQA-Java* representation of a discrete partial order of size seven, becomes LPO II after the MOQA product function products elements a, b and c above elements d and e

the MOQA data structure in this way is that every OrderedComponent instance has at most one OrderedComponent instance directly above it and at most one OrderedComponent instance directly below it. Figure 3.4 illustrates how explicitly recording all of the OrderedComponent instances directly above and below each OrderedComponent instance differs from the *MOQA-Java* approach. (Note that any saving in space from the links being implicit is overshadowed by the space that a SubLPO instance consumes. The impact of this is considered in Section 3.2.) This SubLPO design greatly simplifies the checks for correctness that must be carried out on the arguments supplied to each MOQA function. For example, let $A$ denote an OrderedComponent instance and let $B$ denote an OrderedComponent instance distinct from $A$. If the MOQA product function is requested to product $A$ above $B$, then *MOQA-Java* can determine whether $A$ and $B$ are legitimate arguments for the MOQA product function by simply checking whether they are both directly contained within the same OrderedCollection instance. A final minor characteristic of the SubLPO design is that it can give some indication of the sequence of MOQA functions that produced the LPO instance in question.

Returning to the remaining classes in Figure 3.1, a constant instance of the

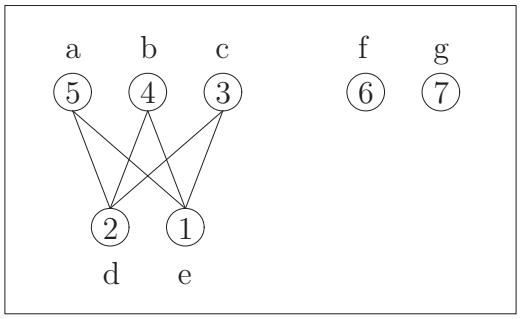



Figure 3.4: LPO II' is LPO II in Figure 3.3 without the SubLPO design

Marker class is used as an argument separator for some *MOQA-Java* functions. For example, the MOQA product function in the OrderedCollection class uses the Marker constant to split its variable number of arguments into two groups. The first group is the arguments that precede the Marker argument and the second group is the arguments that succeed the Marker argument. The first group of arguments is then producted above the arguments of the second group.

The OrderedCollectionSubset, NodeInfo and Marker class implement the CollectionConstruct interface. This interface is for classes that are visible outside of the *MOQA-Java* package and, as the name of the interface suggests, are involved in the construction of a LPO/SubLPO instance. For example, the variable number of arguments accepted by the MOQA product function in the OrderedCollection class are all of type CollectionConstruct.

Finally, the purpose of the OrderedCollectionProp/OrderedComponent-Prop class, which has package-level visibility, is to gather together all of the attributes and operations common to an OrderedCollection/OrderedComponent. For example, an OrderedComponentProp instance includes a reference to the OrderedComponent instance directly above and below its OrderedComponent instance and the operations for getting the minimal and maximal elements in its OrderedComponent instance.

Now that the overview of *MOQA-Java* has come to a conclusion, a synopsis of the core *MOQA-Java* functions can follow.

### 3.1.2 *MOQA-Java* Functions

This section outlines the principal methods available to a programmer in *MOQA-Java*. Recall that *MOQA-Java* makes use of generics to specify the type of label value on the elements contained within a LPO instance, in particular, the type variables $T$ and $L$ in the method signatures that follow.

The principal methods made public by the **OrderedCollection** class are:

1. **public static <T extends Comparable<** $T$ **>> OrderedCollectionSet<** $T$ **> copyOf(OrderedCollection<T> oc)**
   Returns a deep copy of the specified OrderedCollection. The deep copy does not include the NodeInfos, which are shallow copied.

2. **public int size()**
   Returns the total number of NodeInfos directly and indirectly contained within this OrderedCollection.

3. **public int directContentSize()**
   Returns the total number of OrderedComponents directly contained within this OrderedCollection, i.e. returns the total number of NodeInfos and SubLPOs directly contained within this OrderedCollection.

4. **public boolean isAtomic()**
   Returns true when every NodeInfo in this OrderedCollection is atomic[4].

5. **public boolean isSeries()**
   Returns true when all of the OrderedComponents directly contained within this OrderedCollection are in series.

6. **public boolean isParallel()**
   Returns true when all of the OrderedComponents directly contained within this OrderedCollection are in parallel.

7. **public OrderedCollectionSubset<L> product(CollectionConstruct... components)**
   Returns the OrderedCollectionSubset whose direct content is the result of producting the specified OrderedComponents preceding the specified

[4]In Schellekens's work [63] the term "atomic" is interchangeable with "discrete".

Marker above the specified OrderedComponents succeeding it. This method is the implementation of the MOQA product function.

8. **public OrderedCollection<L>[5] split(NodeInfo<L> nodeInfo)**
   Returns the OrderedCollection whose direct content is the result of a split on the specified NodeInfo. This method is the implementation of the MOQA split function.

9. **public NodeInfo<L> removeKBiggestDown(int k)**
   Returns the $k^{th}$ biggest NodeInfo in this OrderedCollection after removing it by pushing it down to one of the minimal elements in this OrderedCollection and then deleting that element. This method is an implementation of the MOQA deletion function[6].

10. **public NodeInfo<L> removeKSmallestDown(int k)**
    Returns the $k^{th}$ smallest NodeInfo in this OrderedCollection after removing it by pushing it down to one of the minimal elements in this OrderedCollection and then deleting that element. This method is an implementation of the MOQA deletion function.

11. **public NodeInfo<L> removeKBiggestUp(int k)**
    Returns the $k^{th}$ biggest NodeInfo in this OrderedCollection after removing it by pushing it up to one of the maximal elements in this OrderedCollection and then deleting that element. This method is an implementation of the MOQA deletion function.

12. **public NodeInfo<L> removeKSmallestUp(int k)**
    Returns the $k^{th}$ smallest NodeInfo in this OrderedCollection after removing it by pushing it up to one of the maximal elements in this OrderedCollection and then deleting that element. This method is an implementation of the MOQA deletion function.

13. **public NodeInfo<L> removeMinimum()**
    Returns the smallest NodeInfo in this OrderedCollection after removing

[5] An OrderedCollectionSet is only returned when this OrderedCollection is an OrderedCollectionSet that has no other discrete NodeInfos directly contained within it so there is nothing for this method to do. Otherwise, an OrderedCollectionSubset is always returned.

[6] *MOQA-Java* uses the term "remove" instead of "delete" so that its method signatures are more in conformance with those set down in the java.util.Collection interface.

it by deleting the minimum element in this OrderedCollection. This method is an implementation of the MOQA deletion function.

14. **public NodeInfo removeMaximum()**
    Returns the biggest NodeInfo in this OrderedCollection after removing it by deleting the maximum element in this OrderedCollection. This method is an implementation of the MOQA deletion function.

15. **public Iterator<NodeInfo<L>> iterator()**
    Returns a top-down Iterator over all of the NodeInfos directly and indirectly contained within this OrderedCollection. The returned Iterator does not support the remove operation.

16. **public Iterator<NodeInfo<L>> iteratorExcept(CollectionConstruct... excluding)**
    Returns a top-down Iterator over all of the NodeInfos directly and indirectly contained within this OrderedCollection minus those represented by the specified CollectionConstructs. The returned Iterator does not support the remove operation.

17. **public Iterator<NodeInfo<L>> getDirectNodeInfoIter()**
    Returns an Iterator over all of the NodeInfos directly contained within this OrderedCollection. The returned Iterator does not support the remove operation.

18. **public Iterator<OrderedCollectionSubset<L>> getDirectSubsetIter()**
    Returns an Iterator over all of the OrderedCollectionSubsets directly contained within this OrderedCollection. The returned Iterator does not support the remove operation.

The principal methods made public by the **OrderedCollectionSet** class are:

1. **public boolean add(NodeInfo<L> nodeInfo)**
    Returns true if the specified NodeInfo is successfully added to this OrderedCollectionSet. This method is successful when this OrderedCollec-

tionSet does not already contain a Label whose label value is equal to the label value of the Label of the specified NodeInfo[7].

2. **public boolean addLabelValueAndData(L labelValue, Object... data)**
   Returns true if the specified label value and data pair are successfully added to this OrderedCollectionSet. This method is successful when this OrderedCollectionSet does not already contain a label value equal to the specified label value; the footnote to the previous method also applies for this method.

The principal methods made public by the **OrderedCollectionSubset** class are:

1. **public Set<NodeInfo<L>> removeComplement()**
   Returns the Set of all the NodeInfos directly and indirectly contained within this OrderedCollectionSubset's OrderedCollectionSet minus the NodeInfos directly and indirectly contained within this OrderedCollectionSubset once this method has removed the NodeInfos to be returned from this OrderedCollectionSubset's OrderedCollectionSet. So any OrderedCollectionSubset within the complement of this OrderedCollectionSubset is emptied of all its NodeInfos and is no longer within an OrderedCollectionSet, which motivates the next method[8].

2. **public boolean inCollectionSet()**
   Returns true when this OrderedCollectionSubset is within an OrderedCollectionSet.

[7] Therefore, adding $n$ NodeInfos to a LPO has a total cost of $\frac{n^2-n}{2}$ label-to-label comparisons and hence, an average-case cost of $\frac{n-1}{2}$ label-to-label comparisons. *Distri-Track* does not yet analyse the code for "filling" a LPO with label value and data pairs and assumes any LPO that it receives is already initialised and populated by distinct label values only. The check for label value distinctness could be removed from the implementation on the hope that programmers in *MOQA-Java* will follow *Distri-Track*'s assumption and never add duplicate label values to a LPO. However, removing this check and following this assumption must come with the caveat that the analysed MOQA code can now execute over a data structure state not considered by the MOQA static analysis tool, which would make its average-case analysis incorrect. Chapter 5 considers the capability of the MOQA theory in handling duplicate label values.

[8] As programmers in *MOQA-Java* do not have access to the Node class, the Nodes that the removed NodeInfos belong to can be safely deleted without fear of there being dangling references to them outside of the *MOQA-Java* package.

3. **public boolean isStrictlyIsolated()**
   Returns true when this OrderedCollectionSubset is strictly isolated.

The principal methods made public by the **NodeInfo** class are:

1. **public Label<L> getLabel()**
   Returns the Label of this NodeInfo.

2. **public Set<NodeInfo<L>> removeComplement(OrderedCollection<L> inColl)**
   Returns the Set of all the NodeInfos directly and indirectly contained within the specified OrderedCollection minus this NodeInfo after this method has removed the NodeInfos to be returned from the specified OrderedCollection. Any OrderedCollectionSubset in the specified OrderedCollection that is completely emptied of all its NodeInfos is no longer within an OrderedCollectionSet.

3. **public void add(Object... newData)**
   Adds the specified Objects to this NodeInfo; the specified Objects are data to be associated with the Label of this NodeInfo.

4. **public boolean remove(Object data)**
   Returns true if the specified Object is in, and therefore is removed from, this NodeInfo; the specified Object is data associated with the Label of this NodeInfo.

5. **public boolean replace(Object oldData, Object newData)**
   Returns true if the first specified Object is in this NodeInfo and therefore, is replaced by the second specified Object; the specified Objects are data associated with the Label of this NodeInfo.

6. **public Iterator<Object> iterator()**
   Returns an Iterator over all of the Objects in this NodeInfo; these Objects are all the data associated with the Label of this NodeInfo.

So, in addition to implementing the MOQA functions in Section 2.2, it can be seen from above that *MOQA-Java* also provides many other functions for accessing and querying state. However, none of these *MOQA-Java* functions are primitive operations in the sense of a standard programming language

though they are the most basic functions available. The MOQA theory does not provide simpler constructs, which seems to defy the concept of having simple ideas that can be gathered together to form more complex ideas, as one definition of a powerful programming language requires [1]. So, considering the range of expressiveness that the MOQA functions display, *MOQA-Java* cannot be seen as a general-purpose programming language with all the corresponding capabilities of a commonly used language. Rather, it is a special-purpose programming language comprised from a suite of statically analysable functions and control flow rules.

It will now be of interest to look at some standard algorithms written in the *MOQA-Java* syntax and consider the various costs that accompany this implementation of the MOQA language.

## 3.2 The Cost of Some *MOQA-Java* Examples

The following examination of insertion-sort and quicksort in *MOQA-Java* has already been published [72].

### 3.2.1 The Space Cost of Insertion-sort in *MOQA-Java*

The pseudo-code [13] for this well-known algorithm, commonly used for its efficiency in sorting small data sets, is as follows:

$InsertionSort(A)$
    for $j \leftarrow 2$ to $length[A]$
        do $key \leftarrow A[j]$
            $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$
            $i \leftarrow j-1$
            while $i > 0$ and $A[i] > key$
                do $A[i+1] \leftarrow A[i]$
                    $i \leftarrow i-1$
            $A[i+1] \leftarrow key$

Figure 3.5 shows insertion-sort implemented in *MOQA-Java*.

After comparing the insertion-sort pseudo-code to Figure 3.5, it is clear that MOQA provides another level of abstraction. There is no explicit reference in the *MOQA-Java* code to the position of the next element to be inserted

```
/**
 * Insertion-sorts the specified OrderedCollection.
 * @param oc a discrete OrderedCollection to be sorted.
 */
public static <L extends Comparable<L>> void
    insertionsort(OrderedCollection<L> oc) {
    if (oc.size() > 1) {
        Iterator<NodeInfo<L>> ocNodeInfos =
            oc.getDirectNodeInfoIter();
        OrderedCollectionSubset<L> sorted =
            oc.product(ocNodeInfos.next(), ocNodeInfos.next());
        for (int i = 1; i < oc.size(); i++) {
            sorted = oc.product(ocNodeInfos.next(), sorted);
        }
    }
}
```

Figure 3.5: Insertion-sort in *MOQA-Java*

amongst the elements already sorted. Instead, there is an iterator over the OrderedCollection to be sorted and this returns the next element for insertion, whereas in the pseudo-code the variable $j$ is an explicit reference to the position of the next element to be inserted. The first two elements returned by this iterator are the parameters for the first MOQA product function. This function connects its first parameter's Node above its second parameter's Node and swaps their NodeInfos if the NodeInfo of the top Node is less than the NodeInfo of the bottom Node. In other words, the labels on the two Nodes will be swapped if they do not agree with the max-heap label ordering. The MOQA product function then creates a new OrderedCollectionSubset within oc and moves the two connected Nodes into this newly created OrderedCollectionSubset, which it then returns. For the MOQA product function in the body of the for loop, its first parameter's Node is connected above the OrderedCollectionSubset returned by the previous MOQA product function and the NodeInfos of these Nodes are swapped around until the max-heap label ordering is satisfied. Once satisfied, the MOQA product function then moves its first parameter's Node and the OrderedCollectionSubset now connected below it into a newly created OrderedCollectionSubset within oc, which is then returned.

Figure 3.5's insertion-sort does not throw a ConcurrentModificationException and from this it can be seen that a *MOQA-Java* iterator is *not fail-fast*;

to generalise, if an OrderedCollection is modified at any time after a *MOQA-Java* iterator over it is created, then the iterator does not fail. This is because the iterator is actually created over a shallow copy of the OrderedCollection's content. Furthermore, any iteration, whether partial or complete, over an OrderedCollection will always return the NodeInfos in the order that their Nodes are stored in the OrderedCollection. The Nodes are stored in the order that they were added to the OrderedCollection.

It is also clear that the in-place nature of the insertion-sort pseudo-code is lost in the *MOQA-Java* implementation as a new OrderedCollectionSubset is created for every element, apart from the first two, in oc; there is just one OrderedCollectionSubset created for the first two elements in oc. So the Node associated with the first parameter of each MOQA product function, along with the Node associated with the second parameter of the first MOQA product function, is removed from oc and added to a newly created OrderedCollectionSubset, that is in turn added to oc. This results in the creation of $n - 1$ OrderedCollectionSubsets, so *MOQA-Java*'s insertion-sort requires $(n - 1).x$ extra space, where $x$ is the space required by an OrderedCollectionSubset. This extra space applies when oc is an OrderedCollectionSet because the direct content of an OrderedCollectionSet must be connected components, i.e. must be Nodes and/or OrderedCollectionSubsets all in parallel. Otherwise, if oc is an OrderedCollectionSubset, then *MOQA-Java*'s insertion-sort requires $(n - 2).x$ extra space. This is because a new OrderedCollectionSubset is not created for the final MOQA product function. The function simply connects the Node above the OrderedCollectionSubset and adjusts the labeling; Section 3.1 explained this design choice.

So the requirement that the MOQA data structure is a partial order results in its traversal and manipulation being more intricate than that of an array. The current implementation is one way of representing this additional complexity and the insertion-sort example demonstrates its extra cost in storage space. While it is worthwhile to focus on the "real-estate" aspect, it is of greater interest to consider the average-case cost of an algorithm written in the current MOQA language implementation, i.e. what price does the MOQA theory carry when it comes to the average-case cost of an algorithm in *MOQA-Java*? It will be useful to examine this question with a sorting algorithm that is not so simplistic in its approach to sorting, the quicksort algorithm.

### 3.2.2 The Average-case Cost of Quicksort in *MOQA-Java*

Quicksort is one of the more interesting sorting algorithms due to the asymptotic difference between its average-case and worst-case behaviour so this section will consider the average-case behaviour of quicksort in the *MOQA-Java* implementation. This analysis is based on certain assumptions. The first being that input values are distinct. This assumption is "fundamental to the analysis of nearly all sorting programs, and it is very often realistic" [66]. Along with this, it is often assumed when analysing the behaviour of an algorithm that all permutations of the distinct input values are equally likely. This is the second of the assumptions and is in agreement with the MOQA theory. Another of the *MOQA-Java* quicksort assumptions is that the following costs take a fixed time:

- the initialisation of a variable,
- the assignment of a value to a variable,
- each arithmetic operator,
- a boolean comparison,
- adding an item to and accessing an item in a collection of items,
- the instanceof keyword,
- the new keyword. (This final fixed-time cost does not cover the cost of the operations within the constructor of the newly created object. If any operations are present in the constructor their cost is also calculated).

A further assumption is made regarding these fixed costs, that they are all equal. This closing assumption can be replaced by a more refined estimation at a later time.

The pseudo-code [13] for quicksort, which sorts the array $A[p \ldots r]$ from index $p$ to index $r$ inclusive, is:

```
/**
 * Quicksorts the specified OrderedCollection.
 * @param oc a discrete OrderedCollection to be sorted.
 */
public static <L extends Comparable<L>> void
    quicksort(OrderedCollection<L> oc) {
    if (oc.size() > 1) {
        NodeInfo<L> pivotNI = oc.getDirectNodeInfoIter().next();
        OrderedCollection<L> partition = oc.split(pivotNI);
        Iterator<OrderedCollectionSubset<L>> aboveAndBelow =
            partition.getDirectSubsetIter();
        if (aboveAndBelow.hasNext()) {
            quicksort(aboveAndBelow.next());
            if (aboveAndBelow.hasNext()) {
                quicksort(aboveAndBelow.next());
            }
        }
    }
}
```

Figure 3.6: Quicksort in *MOQA-Java*

*Quicksort*$(A, p, r)$
    if $p < r$
        then $q \leftarrow$ *Partition*$(A, p, r)$
            *Quicksort*$(A, p, q - 1)$
            *Quicksort*$(A, q + 1, r)$

*Partition*$(A, p, r)$
    $x \leftarrow A[r]$
    $i \leftarrow p - 1$
    for $j \leftarrow p$ to $r - 1$
        do if $A[j] \leq x$
            then $i \leftarrow i + 1$
                *exchange* $A[i] \leftrightarrow A[j]$
    *exchange* $A[i + 1] \leftrightarrow A[r]$
    return $i + 1$

Figure 3.6 shows quicksort implemented in *MOQA-Java*.

The most basic recurrence for this standard quicksort, which does not take advantage of the optimisation techniques presented by other works, see [45],

[37] and [67], is:

$$\overline{T}_{qs}(n) = (n-1) + \frac{2}{n}.\sum_{k=1}^{n}\overline{T}_{qs}(k-1), \qquad n > 1. \tag{3.1}$$

It has already been noted that the MOQA functions are not as primitive as those generally found in programming languages; the MOQA functions in *MOQA-Java* are composed of many Java primitive operations. The main MOQA function involved in Figure 3.6 is the MOQA split function and is at least as complex as the algorithm itself; clearly the MOQA split function is in essence partition, which is central to quicksort. The consequence of a MOQA split function on a non-empty partial order is a non-empty partial order above the pivot if there are elements larger than the pivot and a non-empty partial order below the pivot if there are elements smaller than the pivot. So how the labels are arranged is under the sole control of the MOQA split function and indeed this is true for all MOQA functions. Therefore, there is no direct handling of a data structure's labels in a MOQA algorithm and this is a marked difference between Figure 3.6 and the standard quicksort algorithm. So how does this difference affect the average-case cost of *MOQA-Java*'s quicksort?

The recurrence generated by *Distri-Track* [35] for the average-case cost of *MOQA-Java*'s quicksort is:

$$\begin{aligned}
&quicksort[n1_] := \texttt{Which}[\texttt{Greater}[n1, 1], \texttt{Plus}[-1, n1,\\
&\quad \texttt{Sum}[\texttt{Times}[\texttt{Power}[n1, -1], quicksort[\texttt{Plus}[-1, n1, \texttt{Times}[-1, r0]]]],\\
&\quad \{r0, 0, \texttt{Plus}[-1, n1]\}],\\
&\quad \texttt{Sum}[\texttt{Times}[\texttt{Power}[n1, -1], quicksort[r0]], \{r0, 0, \texttt{Plus}[-1, n1]\}]],\\
&\quad \texttt{True}, 0];\\
&method[n0_] := quicksort[n0];
\end{aligned}$$

This recurrence involves Mathematica functions [49], which are identified here by the `typewriter font`. Let $n1$ denote the size of quicksort's discrete partial order parameter and let $r0$ denote the number of elements that the MOQA split function connects *below* the pivot element. According to the above recurrence, $r0$ is equally likely to be any integer in the range $[0, n1-1]$.

However, the following scrutiny of *MOQA-Java*'s quicksort is the result of a hand analysis of both Figure 3.6 and the *MOQA-Java* implementation of the relevant MOQA functions. This results in a more detailed recurrence than the above *Distri-Track* recurrence, which does not attempt to include *MOQA-Java* implementation costs.

The average-case recurrence for *MOQA-Java*'s quicksort on a discrete partial order of size $n$ is:

$$\begin{aligned}\overline{T}_{mqs}(n) &= c_1.n + \overline{T}_{split}(n) + c_2.\left(\frac{3.(n-4)+10}{n}\right) + c_3.\left(\frac{2.(n-4)+4}{n}\right) + \\ & c_4 + \frac{2}{n}.\sum_{k=1}^{n}\overline{T}_{mqs}(k-1), \qquad n > 3, \end{aligned} \tag{3.2}$$

where the first term $c_1.n$ is the cost of getting an iterator over the NodeInfos in the OrderedCollection to be sorted, the third term $c_2.\left(\frac{3.(n-4)+10}{n}\right)$ is the cost of getting an iterator over the OrderedCollectionSubsets in oc after the MOQA split function and the fourth term $c_3.\left(\frac{2.(n-4)+4}{n}\right)$ is the cost of making the recursive calls. $c_1, c_2$ and $c_3$ are constants in these costs and $c_4$ is the other constant costs that occur in a call to *MOQA-Java*'s quicksort.

Equation 3.2 can be simplified to:

$$\begin{aligned}\overline{T}_{mqs}(n) &= c_1.n + \overline{T}_{split}(n) - \left(\frac{2.c_2 + 4.c_3}{n}\right) + c_4 + \\ & \frac{2}{n}.\sum_{k=1}^{n}\overline{T}_{mqs}(k-1), \qquad n > 3. \end{aligned} \tag{3.3}$$

The second term in the two previous recurrences is $\overline{T}_{split}(n)$, which is the average-case cost of the MOQA split function on $n$ discrete elements. Let $X$ denote an OrderedCollection that has $n$ discrete elements and let $p$ denote the pivot element which is one of these discrete elements. The following sequence of events is a broad description of the MOQA split function:

1. $Y = getDiscrete(X, p)$
   Records in collection $Y$ the discrete elements, excluding $p$, that are present in $X$. This allows $X$ be legal input for the MOQA split function even when it has connected components whose sizes are greater than one.

2. $A, B = relation(Y, p)$

Records in collection $A$ the discrete elements in $Y$ that are greater than $p$ and records in collection $B$ the discrete elements in $Y$ that are smaller than $p$, $|A| + |B| + 1 = n$.

3. *relocate*$(A, B, p)$
   If $|A| > 1$, then $A$'s elements are moved from $X$ into a new OrderedCollectionSubset that is directly (if $X$ is an OrderedCollectionSubset) or indirectly (if $X$ is an OrderedCollectionSet) added to $X$. If $|B| > 1$, then $B$'s elements are moved from $X$ into a new OrderedCollectionSubset that is also directly (if $X$ is an OrderedCollectionSubset) or indirectly (if $X$ is an OrderedCollectionSet) added to $X$.

4. *connect*$(A, p, B)$
   If $|A| > 0$, then the representation of its elements in $X$ is connected above $p$ and if $|B| > 0$, then the representation of its elements is connected below $p$.

Figure 3.6 gets its reference to pivotNI through an iterator. This iterator is over a new collection that contains the NodeInfos of all the Nodes directly contained in oc. The next line in Figure 3.6 calls the MOQA split function, whose first action is to collect together the NodeInfos of all the discrete Nodes, excluding $p$, directly contained in oc. As oc in Figure 3.6 is a discrete OrderedCollection, the remaining elements to be returned by the iterator are the same elements initially collected together by the MOQA split function. This duplication of information makes a case for the MOQA split function to have an additional optional parameter, which would refer to the set of discrete NodeInfos to be split around the pivot NodeInfo, i.e. would be the iterator. Tailoring the implementation of a MOQA function so as to increase its efficiency for a specific application is far from ideal but, because the definition of the MOQA split function shows it to be purpose-built for quicksort/quickselect, it would seem to be an acceptable concession in this instance.

Following the assumptions laid out earlier, the average-case cost of the above events is:

$$\begin{aligned} \overline{T}_{csplit}(n) &= 46.n + \left( \frac{\sum_{k=3}^{n-2} 32.(k-1) + 24.n + c_x}{n} \right) + \\ &\quad \left( \frac{192.n + 4.c_x + 73}{n} \right) + c_y, \qquad n > 3, \end{aligned} \tag{3.4}$$

where $c_x$ and $c_y$ are constants.

The storage space consumed by *MOQA-Java*'s insertion-sort depended on whether the OrderedCollection to be sorted was an OrderedCollectionSet or an OrderedCollectionSubset. The same principle holds for $\overline{T}_{split}(n)$ because its cost is also dependent on whether oc is an OrderedCollectionSet or an OrderedCollectionSubset. This difference in cost is again tied to the design decision that all of the direct content in an OrderedCollectionSet is in parallel. So Equation 3.4 is the common average-case cost of the MOQA split function regardless of oc's type. When oc is an OrderedCollectionSet, then $\overline{T}_{split}(n)$, which is the average-case cost of the MOQA split function on $n$ discrete elements, in Equation 3.3 is replaced by:

$$\overline{T}_{splitSet}(n) = \overline{T}_{csplit}(n) + \frac{64}{n} + c_r, \qquad n > 3, \tag{3.5}$$

where $c_r$ is a constant. When oc is an OrderedCollectionSubset, then $\overline{T}_{split}(n)$ in Equation 3.3 is replaced by:

$$\overline{T}_{splitSubset}(n) = \overline{T}_{csplit}(n) + c_s, \qquad n > 3, \tag{3.6}$$

where $c_s$ is a constant.

Though the average-case behaviour of *MOQA-Java*'s quicksort is still log-linear, its cost, as investigated just above, is clearly higher than the standard given in Equation 3.1. There is no doubt that refining the *MOQA-Java* implementation would reduce some of these extra costs. However, MOQA functions remove certain actions normally carried out directly within an algorithm and execute these actions internally. So they provide another level of abstraction between the programmer and the data structure, which comes at a price. Additionally, the MOQA functions have to be as general-purpose as possible because they are not geared towards one specific implementation of one specific algorithm and therefore, will require some input validation, which is another extra expense. Hence, this all means that it is unlikely that the constants for a MOQA version of quicksort will ever be reduced to those in Sedgewick's non-optimised version of quicksort [67].

In conclusion, there is an additional cost in space for insertion-sort in *MOQA-Java* and a tendency towards higher constant values than normal for

the average-case behaviour of quicksort in *MOQA-Java*, though the recurrence does not deviate from the standard trend of quicksort. To statically time algorithms, this work deems it acceptable to carry some extra expense when it does not change the asymptotic behaviour of the algorithm. Accordingly, while it would be advantageous to lower these costs to their minimum value, it can be expected, in order to statically determine the average-case cost of an algorithm, that the constants will be higher within the same order of growth as traditional variants.

## 3.3 New MOQA Functions

This section introduces new MOQA functions developed during the course of this research, all of which are MOQA random structure preserving.

When aiming to construct a well-designed language, the motivation for adding a function must extend beyond the benefit of extra functionality. An actual need for each function added should be perceived. Heapsort algorithm discussions led to the creation of the MOQA top function, the converse MOQA bot function and the MOQA lift function. Their introduction then paved the way for Schellekens's new treapsort algorithm [63], see Appendix A. (These functions were first presented in the MOQA book [63]. They are not present in *MOQA-Java* as they were conceived after work on the language implementation was concluded. Future work could include incorporating these functions into *MOQA-Java*.) The MOQA insert function contribution is given after its description below. Another new MOQA function is presented on page 213 in Section 6.5. This function improves on the MOQA product function but it is necessary to wait until Section 6.5 for the full details, as the section's context helps in explaining this function's inspiration, description and benefits.

Though not listed below, other MOQA random structure preserving functions were discovered during the course of this research. As some of these functions currently lack an application, they should not be included in any implementation of the MOQA language until a need for them arises. Leaving these functions aside, four of the new MOQA functions are now considered.

### 3.3.1 MOQA Top

Let $I_{\beta_{max}}$ denote an isolated subset of the series-parallel $H_{\beta_{max}}$. Let $x$ denote the maximum node in $I_{\beta_{max}}$, i.e. the label on $x$ is greater than the label on any other node in $I_{\beta_{max}}$. For any labeling on $H_{\beta_{max}}$, the MOQA top function connects $x$ above every other maximal node in $I_{\beta_{max}}$; all of the nodes below $x$ prior to the MOQA top function remain as they are. The labeling then on $I_{\beta_{max}}$ has no need of modification because the nature of the MOQA top function ensures that it is correct.

Therefore, the average-case cost of the MOQA top function when applied to the isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, is zero and the multiplicity of $M^{S_j^{\mathcal{M}}}$ is unaffected.

### 3.3.2 MOQA Bot

Let $I_{\beta_{max}}$ denote an isolated subset of the series-parallel $H_{\beta_{max}}$. Let $x$ denote the minimum node in $I_{\beta_{max}}$, i.e. the label on $x$ is smaller than the label on any other node in $I_{\beta_{max}}$. For any labeling on $H_{\beta_{max}}$, the MOQA bot function connects $x$ below every other minimal node in $I_{\beta_{max}}$; all of the nodes above $x$ prior to the MOQA bot function remain as they are. The labeling then on $I_{\beta_{max}}$ has no need of modification because the nature of the MOQA bot function ensures that it is correct.

Therefore, the average-case cost of the MOQA bot function when applied to the isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, is zero and the multiplicity of $M^{S_j^{\mathcal{M}}}$ is unaffected.

### 3.3.3 MOQA Lift

Let $I_{\beta_{max}}$ denote an isolated subset of the series-parallel $H_{\beta_{max}}$. The label on the maximum node in $I_{\beta_{max}}$ is pushed downwards as it would be by the MOQA deletion function until it reaches a minimal node in $I_{\beta_{max}}$, that minimal node is then deleted and a new node with the deleted label is connected above every maximal node in $I_{\beta_{max}}$.

This new MOQA lift function is plainly the MOQA deletion function aug-

mented by the reinsertion of the deleted node. Therefore, the average-case cost of the MOQA lift function when applied to the isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, is $\Delta_{down}(I_{\beta_{max}}, |I_{\beta_{max}}|)$ and the multiplicity of $M^{S_j^{\mathcal{M}}}$ is unaffected, as it would be by the MOQA deletion function.

### 3.3.4 MOQA Insert

Let $I_{\beta_{max}}$ denote an isolated subset of the series-parallel $H_{\beta_{max}}$. The MOQA insert function adds a new discrete node to $I_{\beta_{max}}$.

This is the first proper MOQA insertion function; the MOQA product function does not count because it adds edges not nodes. Recall that "each data-labeling is assumed to be priorly constructed via MOQA operations from a random input list" [63]. So Schellekens's work [63] does not allow a node to be added to a data structure during the execution of a MOQA program. This is functionality now provided by the MOQA insert function.

However, why does Schellekens [63] not allow a node to be added to a data structure during the execution of a MOQA program? In other words, is there a good reason why the MOQA insert function is not found in Schellekens's research [63]? According to this research, all of a data structure's $n$ nodes must be in place prior to MOQA program execution so that it can assume that the label of each node is equally likely to be any value in the set $\{1, 2, \ldots, n\}$ when no two nodes can have the same label, $n \geq 0$. (Note that none of Schellekens's functions [63] can be applied to an empty data structure because each one deals only with the nodes already present in the data structure. Therefore, in practise, $n$ will always be greater than zero because there is no sense in creating empty data structures that are never used.) Yet this label distribution is just assumed so why not always commence with an empty data structure and instead assume that the label of each node added to a data structure of size $d$ is equally likely to fall into any of the $d+1$ intervals defined by the $d$ values currently serving as labels on the data structure, $d \geq 0$? This latter assumption is interchangeable with the former and is the one backing the MOQA insert function, i.e. it is assumed that the label of the discrete node added to $I_{\beta_{max}}$ is equally likely to fall into any of the $|H_{\beta_{max}}|+1$ intervals defined by the $|H_{\beta_{max}}|$ values currently serving as labels on $H_{\beta_{max}}$. (This

latter assumption is also found in work by Knuth [42]. Section 6.9 studies this research by Knuth and the relation between these two assumptions is explored further there.) This assumption change does not negatively effect the rest of the MOQA theory and so, in answer to the questions just posed, there is no obstacle to the new MOQA insert function. It is acceptable because it is a proponent of just another way of viewing MOQA's initial label distribution.

In fact, there are positive outcomes from always commencing with an empty data structure and then using the MOQA insert function to add nodes. The scope of MOQA programs is increased because it is now no longer necessary to fix the number of nodes present in a data structure prior to executing any sequence of MOQA functions. Also, some of the MOQA theory can now be simplified. For example, $L_j^{\mathcal{M}'}$, which is a necessary extension to the current MOQA theory and is introduced in Section 4.4.4, would now become redundant. Finally, this approach is more consistent with how data structures are normally constructed.

The average-case cost of the MOQA insert function when applied to the isolated subset $I_{\beta_{max}}$ of $H_{\beta_{max}}$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, is zero and the multiplicity of $M^{S_j^{\mathcal{M}}}$ is unaffected.

## 3.4 MOQA and Reversibility

It is stated that "MOQA is close in spirit to reversible languages" [64] and that "reversibility refers to the fact that for any output it is possible to re-compute the input which gave rise to the output in question" [64]. The Janus language[9] is cited by Schellekens [64] as one of the few examples of a reversible language. The work of Yokoyama and Glück [79] formalises, and proves the reversibility of, Janus, and provides the following more formal definition of a reversible computing system.

**Definition 25** (A reversible computing system)**.** *A reversible computing system [71, 7, 28] has, at any time, at most a single previous computation state as well as a single next computation state, and thus a reversible computing system can run programs uniquely forward and backward by following the deterministic*

[9]Developed by C. Lutz for a Caltech course.

*trajectory of the computation.*

This is the definition of a *logically reversible computing system.* A *thermodynamically reversible computing system* is a physical computing system that generates almost no new physical entropy; *physical entropy* is the amount of energy that cannot be used to do work. Erasing information at a hardware level generates physical entropy because it requires transferring the heat that arises from grounding a circuit node that is holding a charge, with the charge representing the information to be erased. A logically reversible computing system can reduce or eliminate new physical entropy because reversible functions are less likely to throw away information so that any alteration they make to their input can be undone [56].

However, not all of the MOQA functions are naturally reversible. Figure 3.7 demonstrates this by showing, for the MOQA product function and the max-heap label ordering, half of the fourteen possible input pairs to product($x_i, y_i$) that result in a total order of size four as output. For the other seven possible input pairs, swap the x and y identifiers in each pair in Figure 3.7; so $x_a$ would now identify the total order of size one whose label is one and $y_a$ would now identify the total order of size three whose labels are four, three and two, and so on for the other six pairs. This example makes it clear that the standard MOQA language is not reversible because not all of its functions are reversible.

However, the MOQA product function becomes reversible when "indices are tracked during computation, where swaps between labels result in a corresponding swap between indices of elements" [64]. Recording this additional information logs the changes the MOQA product function makes to its input partial order. So when such erudite output from a MOQA product function is accompanied by an inverse MOQA product function, it is then possible to step backwards from a MOQA product function's output and uniquely determine its input. Does this mean that the MOQA language is close in spirit to reversible languages?

A language is irreversible when key information about the behaviour of a program in this language is lost during execution. Both Yokoyama and Glück's research [79] and Bennett's seminal work [6] identify these key areas of information loss to be data erasure and the paths taken in conditional branching. In fact, Yokoyama and Glück further clarify this by stating that the main difference between standard languages and those that are reversible is "re-

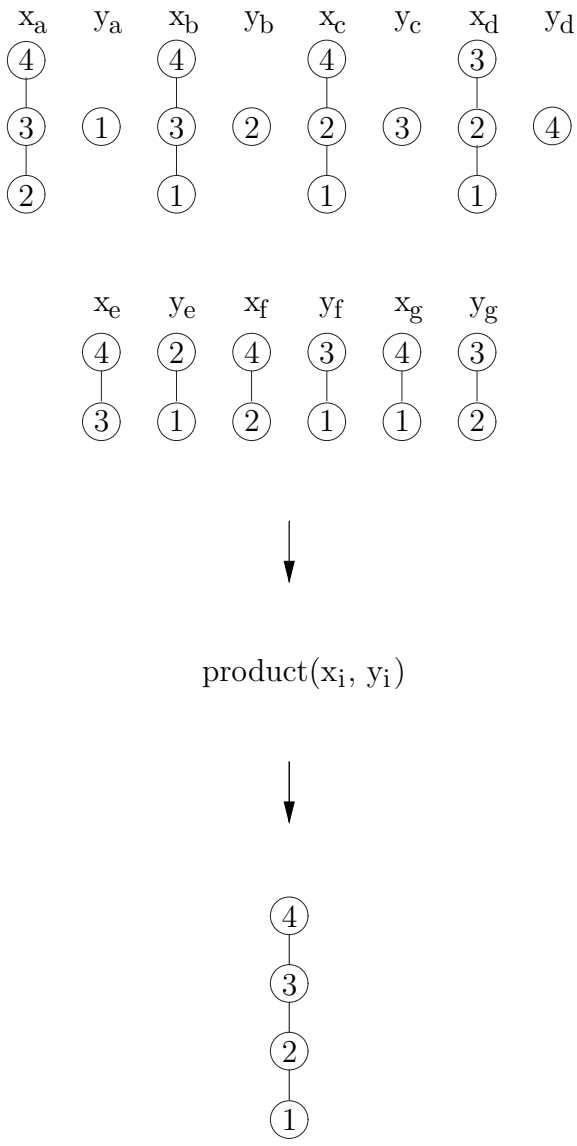



Figure 3.7: For the MOQA product function and $\beta_{max}$, half of the possible input pairs to product($x_i, y_i$) that result in the output of a total order of size four

versible assignments and control constructs, and the possibility of uncalling procedures" [79]. So tracking and swapping element indices allow for the possibility of uncalling the MOQA product function. This is also a possibility for the MOQA split function because it already has a one-to-one correspondence between input and output. However, as acknowledged by Schellekens [64], the MOQA delete function is not reversible. So, as there has been no consideration about how the MOQA deletion of data can be reversed, the MOQA language does not currently overcome one of the two main problems that reversible languages address. In addition to this, there has been no work on reversible assignments and reversible control constructs in the MOQA language. So this is the other main obstacle to reversibility that the MOQA language has not yet overcome and, as only the most basic of programs have no control constructs, it is one that cannot be avoided. Hence, it would seem to be far too early to claim reversibility for a language that has no provision for the reversal of its destructive operations and control flow statements despite the promising reversibility of its non-destructive functions.

However, it is still possible to reverse a MOQA program. Bennett [6] considers the logical reversion of computing systems whose functions are not necessarily bijective and proves that *any* Turing machine can be made logically reversible at every step. This is achieved by recording a computational history during the forward execution of the program that can be cleaned up during the backward execution of the program. The backward execution of the program is then given a copy of the forward program's output and with the computational history produces the forward program's input. This appears to be a common approach to reversibility and is one that enables the reversal of a program written in any programming language. However, Yokoyama and Glück [79] make it clear that a language which is reversible in nature has its own methodology. This methodology means that the deterministic forward and backward execution of a Janus program does not require a computational history.

So the MOQA language is not at present close in spirit to high-level reversible languages such as Janus. Its programs do not have the potential benefit of those which are written in such a language, i.e. "programs that are built from locally invertible primitives and control flow operators have the potential benefit of reversibility of the underlying reversible structures" [79]. This is be-

cause MOQA programs only have the underlying reversible structure feature and they only have this feature when there are no contractive MOQA functions in the program. Therefore, to logically reverse a MOQA program now would require the same technique required to logically reverse any program written in any other irreversible programming language. Hence, the reservation here regarding the claim that the MOQA language is a reversible language.

## 3.5 Chapter Summary

*MOQA-Java*, presented here in detail for the first time, is a working implementation of the MOQA language theory [63]. The central aim behind its design is to help *MOQA-Java* programmers naturally write MOQA-satisfying programs. However, there is still room to improve, as highlighted by the space costs revealed in Section 3.2.1. Examination of the *MOQA-Java* code at a later date also revealed areas that could do with further refactoring. For example, there are more areas in the code that would benefit from the application of polymorphism. However, these issues are not serious enough to prevent *MOQA-Java* from being the language in which programs submitted to *Distri-Track* are written and from continuing to maintain its status as the current implementation of the MOQA language.

The analysis of the quicksort algorithm presented in this chapter demonstrated the average-case cost of the *MOQA-Java* split function. This average-case cost included costs not factored into MOQA's average-case formulas, giving a fuller appreciation for the price paid when there is another layer of abstraction between the programmer and the machine. This chapter also considered the reversibility of the MOQA language and reasoned that additional work is required before the MOQA language can be truly be called reversible.

The MOQA language implementation was one of the first undertakings in this body of research. During the process, new MOQA functions were found, some of which are reported on in this chapter. In addition to these new functions, deeper underlying issues were also found in the MOQA theory. What these issues are and how they are overcome is the main subject matter of the remaining chapters. Hence, future work could retrofit *MOQA-Java* for some of the solutions presented in these chapters, such as the new data structure types, or it may be considered more appropriate to write a new implementation of

the MOQA language from scratch.

# Chapter 4

# Tracking Data Structure State

This chapter begins by expanding the MOQA theory and does so for two important reasons. First of all, there are concepts in the MOQA book [63] that are not allowed within the framework of its own theory. A key example of this incongruity is the MOQA random bag; Schellekens's definition of this [63] does not include some of the data structure types inferred to be in it in later chapters of that work. So this expansion of the MOQA theory will rectify some of these inconsistencies. Furthermore, the redefining of the MOQA theory will enable this work to discuss the theoretically capabilities of MOQA with greater confidence in Chapter 6 and Section 7.1. Another reason, though minor, is to tighten up some of the current MOQA definitions.

As well as expanding the MOQA theory, this chapter carefully defines various data structure types and considers their average-case cost from the MOQA stance, considering inductive po-class sub-categories in particular depth. New average-case equations for some of these types are then presented in this chapter, along with new equations that assist in calculating the relative frequency of these types occurring in the MOQA random bag. The result of this is that the MOQA static analysis tool will be able to determine the average-case behaviour of MOQA functions for additional data structure types and therefore, its application has the potential to be widened. (The phrase "the MOQA static analysis tool" is used throughout this work and refers to the ideal MOQA static analysis tool, i.e. there is no statement being made about the capabilities of the current version, which is *Distri-Track* [35].) Finally, the clear categorisation of the data structure types that takes place in this chapter has the added benefit of clarifying which ones have yet to incorporated into

the MOQA theory, thereby identifying areas open to future work.

## 4.1 Chapter Overview

In support of the new research laid out in this substantial chapter, there are many formal notations, definitions and proofs. Hence, it may aid the reader to first begin with an informal high-level overview of what is soon to be discussed in depth.

Before all else, recall two of the fundamental definitions found in the MOQA book [63] and restated in Chapter 2: the definition of a MOQA random structure and the definition of a MOQA random bag. These definitions can be briefly summarised as follows:

**MOQA random structure** A representation of a particular series-parallel finite partial order and all of its canonically-ordered labelings when all of its canonically-ordered labelings have *equal* likelihood of occurring.

**MOQA random bag** A representation of a collection of MOQA random structures and how often each one occurs in relation to the others.

These definitions restrict the MOQA theory to the static analysis of data structures whose sizes are fixed. This limitation cannot be easily ignored because it is unusual to statically determine an algorithm's average-case cost for a data structure of fixed size[1], e.g. determine quicksort's average-case cost for a data structure of size 5. Therefore, this chapter expands the above definitions to incorporate data structures whose sizes can vary, e.g. data structures of size $n$. These new definitions can be briefly summarised, for now, as follows:

**MOQA′ random structure** A representation of any one of the following series-parallel data structures and all of their canonically-ordered labelings[2]:

- **F**ixed **P**o-**S**tructure ($FPS$), i.e. a finite partial order.
- **I**nductive **P**o-**C**lass ($IPC$), i.e. an inductive type. An inductive po-class is the same concept as an inductive type, which Hickey [35]

[1] This is borne out by the literature reviewed in Chapter 6.

[2] The expected likelihood that these canonically-ordered labelings have of occurring will be detailed later on.

first introduced to MOQA. It is renamed here for the sole purpose of consistent terminology.

- **S**plit **P**o-**C**lass or **G**eneral **S**plit **P**o-**C**lass ($SPC$ or $GSPC$), i.e. the two categories of data structures that can result from the MOQA split function. The MOQA book [63] and Hickey [35] introduced these structures respectively.
- **C**ompound **S**tructure ($CS$), i.e. a combination in parallel and/or series of the above structures.

**MOQA′ random bag** A representation of a collection of MOQA′ random structures and how often each one occurs in relation to the others.

Note that the first three data structures listed in the MOQA′ random structure definition are themselves not new to this work. The last three data structures listed are new additions to the original MOQA random structure definition as the fixed po-structure is the only data structure that the original definition catered for.

Though the latter part of the MOQA book [63] does discuss the static analysis of data structures whose sizes can vary and the current implementation of the MOQA static analysis tool [35] can determine the average-case cost of certain algorithms for such data structures, the definitions in the MOQA book [63] are too narrow to accommodate this effort. So the new definitions given in this chapter, the above abridgements for example, serve the purpose of harmonising the MOQA theory with its application for the first-time. This is a very necessary adjustment and will prevent this work from subsequently finding itself in the awkward situation of examining and evaluating research that, in reality, is more powerful than its theory allows for.

Next, this chapter introduces further data structure types, each of which is a type of inductive po-class, that can be statically analysed by the MOQA methodology. In other words, the average-case cost of a MOQA function when applied to one of these new data structure types can now be resolved statically. To allow for this, composition laws and their requisite proofs are given for each new type; remember that composition laws are subsidiary equations that MOQA depends on when statically determining the average-case cost of its functions and that the four composition law groups are $\sigma$, $\kappa$, $\tau$ and $\Delta$. The

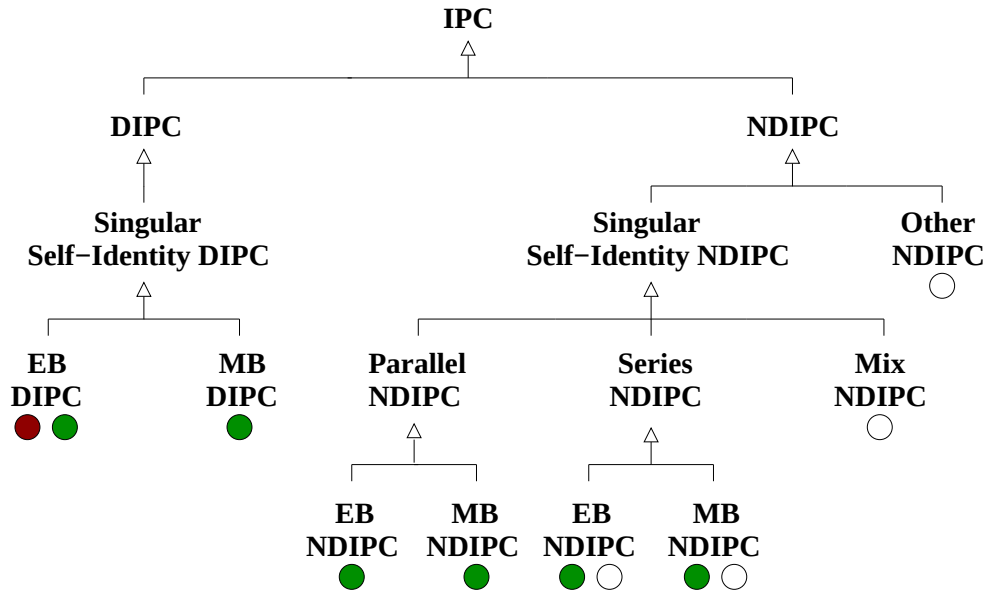



Figure 4.1: Inductive Po-Class Framework

remaining equations that must accompany the composition laws for the new MOQA data structure types are then provided towards the end of this chapter.

An inductive po-class framework is also introduced throughout this chapter. The central purpose of this framework is to identify the nature of inductive po-classes that do, and do not, lend themselves to being analysed by the MOQA approach. It is visually depicted in Figure 4.1, with each leaf of the tree denoting an inductive po-class type. The acronyms used in Figure 4.1 are as follows:

- EB = **E**mpty-**B**ase
- MB = **M**ulti-**B**ase
- DIPC = **D**eterministic **I**nductive **P**o-**C**lass
- NDPIC = **N**on-**D**eterministic **I**nductive **P**o-**C**lass

Figure 4.1 allows a clear distinction to be made between the inductive po-class type introduced by Hickey [35], which is the empty-base DIPC and is marked with a red and a green circle, and the seven types introduced by this work, which are marked with green and white circles. An inductive po-class type marked with just a green circle indicates that this research has established all

of its composition laws. An inductive po-class type marked with just a white circle indicates that all of its composition laws are yet unknown. An inductive po-class type marked with both a green and a white circle indicates that this research has established some but not all of its composition laws. Finally, the empty-base DIPC is marked with both a red and a green circle to indicate that all of its composition laws have been established by Hickey [35], with some fine-tuning of these laws taking place here. The benefits of having these extra MOQA data structure types has already been outlined in the introduction to this chapter, with the primary benefit being a future MOQA static analysis tool of possibly greater scope.

With the overview of this chapter complete, it is now time to focus on specifics. This shall commence with the groundwork being laid for the MOQA theory expansion.

## 4.2 Program Control Flow

The flow of control through a program can be represented by a directed graph. Each vertex in such a graph is a basic block. The following basic block definition is taken from the well-known dragon book [2].

**Definition 26** (Basic block)**.** *A basic block for program $P$ is a maximal sequence of consecutive instructions in $P$ with the following properties:*

- *The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.*
- *Control will leave the block without halting or branching, except possibly at the last instruction in the block.*

The directed graph that shows all the possible paths of execution through a program is known as a control flow graph. (It is being assumed that the program does not contain unreachable code so only execution paths that are possible are present in the control flow graph.)

**Definition 27** (Control flow graph)**.** *A control flow graph for program $P$ is a directed graph whose nodes are basic blocks corresponding to $P$'s code and*

*whose edges indicate which blocks can follow which other blocks, i.e. indicate potential flow of control between the nodes of $P$.*

**Definition 28** (Control flow path)**.** *A control flow path is an ordered sequence of adjacent nodes taken from a control flow graph, i.e.* $(n_1, n_2, \ldots, n_k)$.

Both of the above definitions are closely based on those given by Zeil [80].

**Definition 29** (Originating control flow path)**.** *A control flow path is an originating control flow path if the first node in its ordered sequence has no predecessor in the control flow graph from which it is taken.*

**Definition 30** (MOQA control flow graph)**.** *A MOQA control flow graph is the control flow graph of a MOQA-satisfying program.*

**Notation 39.** *Let $MCG_p$ denote the MOQA control flow graph of the MOQA-satisfying program $p$.*

**Notation 40.** *Let $mcf_{p,\,n_a,\,n_b}$ denote the control flow path $(n_a, \ldots, n_b)$ taken from $MCG_p$.*

**Notation 41.** *Let $mcf$ denote $mcf_{p,\,n_a,\,n_b}$ whenever a known $p$, $n_a$ and $n_b$ are being dealt with.*

**Notation 42.** *Let $mcf^o$ denote $mcf_{p,\,n_a,\,n_b}$ when it is an originating control flow path.*

**Notation 43.** *Let $\mathcal{M}^{mcf(\mathcal{M})}$ denote the MOQA random bag after program $p$ follows control flow path $mcf$ when $\mathcal{M}$ is program $p$'s MOQA random bag at the moment the control of flow enters node $n_a$.*

## 4.3 Expanding the MOQA Theory

Section 2.1 stated that, under certain conditions, $MRB_{p,\,c,\,\beta}$ contains a specific MOQA random structure. (Recall that $MRB_{p,\,c,\,\beta}$ denotes composite variable $c$'s MOQA random bag at the first moment that $c$ is referred to in program $p$.) The $H_\beta$ of this specific $S_1^{MRB_{p,\,c,\,\beta}}$ is a discrete Hasse diagram whose size is not fixed and the multiplicity of this $S_1^{MRB_{p,\,c,\,\beta}}$ is one, thus each labeling in $L(H_\beta)$ has a frequency of one. (Recall that $L(H_\beta)$ denotes the set of all canonically-ordered labelings of the Hasse diagram $H$ that has the label ordering $\beta$ on

it.) The conditions are 1), $MRB_{p,\,c,\,\beta}$ is not the output of another MOQA-satisfying program and 2), $MRB_{p,\,c,\,\beta}$ has not been provided as input to the MOQA static analysis tool.

So, when both of these conditions hold, the size of this discrete $H_\beta$ is not known, that is, $n_{S_1^{MRB_{p,\,c,\,\beta}}}$ is not known. Nonetheless, $n_{S_1^{MRB_{p,\,c,\,\beta}}}$ should have one stipulation placed on it; $n_{S_1^{MRB_{p,\,c,\,\beta}}}$ should be greater than or equal to the total number of distinct nodes involved in all of the MOQA functions applied to $c$ in $p$. (The number of nodes "involved" in a MOQA function is the number of nodes in the (strictly) isolated subset considered by that MOQA function. For example, there are five nodes "involved" in the MOQA product function that takes the two disjoint connected components of an isolated subset and products the one with three nodes above the one with two nodes.) This lower bound, having all of $c$'s required nodes from its moment of inception, ensures that $n_{S_1^{MRB_{p,\,c,\,\beta}}}$ is always large enough for any sequence of MOQA functions applied to $c$ in $p$. There is no upper bound on $n_{S_1^{MRB_{p,\,c,\,\beta}}}$.

However, a discrete series-parallel Hasse diagram of no fixed size cannot be represented by $S_1^{MRB_{p,\,c,\,\beta}}$. In keeping with Schellekens's definition [63] of a MOQA random structure, $H_\beta$ is a finite series-parallel Hasse diagram. So how does such a, as yet undetermined, initial size of $c$ integrate with the MOQA theory? To answer this, the following notation will be of use.

**Notation 44.** *Let $(F_{mcf,\,c})$ denote the sequence of MOQA functions applied to composite variable $c$ in control flow path $mcf$.*

$(F_{mcf,\,c})$ can be expanded to the following:

$$(f_{mcf,\,c,\,1},\ f_{mcf,\,c,\,2},\ \ldots,\ f_{mcf,\,c,\,n-1},\ f_{mcf,\,c,\,n}).$$

$(F_{mcf,\,c})$ is ordered chronologically; this order is inferred from $mcf$. For example, let $f_{mcf,\,c,\,x}$ and $f_{mcf,\,c,\,y}$ denote two MOQA functions in $(F_{mcf,\,c})$, $1 \leq x, y \leq n$ and $x \neq y$. If $x < y$, then $f_{mcf,\,c,\,x}$ is applied to $c$ before $f_{mcf,\,c,\,y}$.

**Notation 45.** *Let $I((F_{mcf,\,c}))$ denote the set of all nodes involved in the MOQA functions in $(F_{mcf,\,c})$.*

**Notation 46.** *Let $D((F_{mcf,\,c}))$ denote the set of all nodes deleted by the MOQA deletion functions in $(F_{mcf,\,c})$.*

So, when the MOQA static analysis tool does not have an exact value for $c$'s initial size because $c$ is the discrete series-parallel Hasse diagram just described, the sizes of the data structures constructed/modified by any $(F_{mcf^o,c})$ are what is of interest; these sizes can be inferred from $|I((F_{mcf^o,c}))| - |D((F_{mcf^o,c}))|$. If the MOQA static analysis tool can reduce $|I((F_{mcf^o,c}))| - |D((F_{mcf^o,c}))|$ to a fixed value, then the data structures which $(F_{mcf^o,c})$ constructs/modifies can be represented by finite series-parallel Hasse diagrams. Therefore, such a result of $(F_{mcf^o,c})$ can be represented by $\mathcal{M}^{mcf^o(\mathcal{M})}$ when $\mathcal{M}$ is the initial MOQA random bag under discussion here. Hence, it becomes safe to statically ignore the unbounded number of leftover nodes in $c$'s initial discrete series-parallel Hasse diagram, i.e. the nodes not involved in $(F_{mcf^o,c})$, and just track in $\mathcal{M}^{mcf^o(\mathcal{M})}$ the fixed number of nodes that result after $(F_{mcf^o,c})$. So, though Schellekens's definition [63] of a MOQA random structure does not allow $c$'s initial data structure to be a discrete Hasse diagram of no fixed size, this does not prevent the static calculation of average-case cost for this situation. It is just a minor discrepancy in the theory. In addition, if the new version of the MOQA product function in Section 3.3 replaces the current one, then this discrepancy disappears. Then $c$'s initial data structure under the two conditions stipulated above is an empty partial order, which can be represented by a Hasse diagram.

However, it is still necessary to extend Schellekens's definition [63] of a MOQA random structure and, consequently, the definition of a MOQA random bag. Not just to harmonise the current theory but, more importantly, to be able to represent the data structures that result when the MOQA static analysis tool cannot reduce $|I((F_{mcf^o,c}))| - |D((F_{mcf^o,c}))|$ to a fixed value. Hence, the MOQA theory is extended as follows.

**Definition 31** (Fixed po-structure)**.** *A fixed po-structure is a series-parallel Hasse diagram.*

**Notation 47.** *Let $FPS$ denote a fixed po-structure.*

**Notation 48.** *Let $FPS_\beta$ denote a fixed po-structure and some label ordering $\beta$ on it.*

So $FPS_\beta$ is equal to $H_\beta$ in Section 2.1.

**Definition 32** (Inductive po-class)**.** *An inductive po-class consists of an inductively defined set of rules and some set of fixed po-structures constructed by these rules.*

Inductive po-classes can represent data structures that do not have a fixed size, see Section 4.4.2. Generally, in this work the inductively defined set of an inductive po-class will not be empty.

**Notation 49.** *Let $IPC$ denote an inductive po-class.*

**Notation 50.** *Let $IPC_\beta$ denote an inductive po-class and some label ordering $\beta$ on its set of fixed po-structures.*

**Notation 51.** *Let $R(IPC_\beta)$ denote the zero or more size restrictions placed on $IPC_\beta$.*

In other words, $R(IPC_\beta)$ denotes the zero or more size restrictions placed on the fixed po-structures in $IPC_\beta$'s set. Restrictions that can be placed on $IPC_\beta$ include:

- $IPC_\beta$'s lower bound, i.e. the size of the smallest fixed po-structures in its set.
- $IPC_\beta$'s upper bound, i.e. the size of the largest fixed po-structures in its set.
- $IPC_\beta$'s size range. For example, let $IPC_\beta$'s size range be $[4, 13]$. This means that its set is finite because it would contain all of the fixed po-structures from size four to size thirteen that $IPC_\beta$ can represent.
- General $IPC_\beta$ size constraints. For example, a fixed po-structure is only allowed in $IPC_\beta$'s set if its size is even.

If $R(IPC_\beta)$ is empty, then $IPC_\beta$'s set is infinite; this is equivalent to $R(IPC_\beta)$ just representing that $IPC_\beta$'s lower bound is zero. So when it is unconditionally stated in this work that an $IPC_\beta$'s set is infinite, it means that it contains all the fixed po-structures of all the possible sizes that the $IPC_\beta$ can represent, potentially including the fixed po-structure of size zero. The discussion of inductive po-classes in Section 4.4.2 goes into further detail regarding finite and infinite sets but it is becoming more plausible why an inductive po-class is suitable for representing data structures that do not have a fixed size.

**Notation 52.** *Let $FPS_{\beta_{max},a,b}$ denote a fixed po-structure that is composed of three discrete fixed po-structures in series which, from the top, are of size $a$, 1 and $b$ respectively and the max-heap label ordering on it, $a, b \geq 0$.*

**Definition 33** (Split po-class)**.** *For a specific $n \geq 1$, a split po-class is the set*

$$\{FPS_{\beta_{max},0,n-1}, FPS_{\beta_{max},1,n-2}, \ldots, FPS_{\beta_{max},n-2,1}, FPS_{\beta_{max},n-1,0}\}.$$

The split po-class represents all of the fixed po-structures that can result after the MOQA split function is applied to a discrete fixed po-structure; $n$ is the specific size of the discrete fixed po-structure that the MOQA split function is applied to. Following Schellekens's definition [63] of a MOQA random structure, which states that it represents *one* fixed po-structure, there would have to be a MOQA random structure for each $FPS_{\beta,a,b}$ in a split po-class's set. The redefining of the MOQA random structure, which is to follow, can represent a split po-class so now just *one* of these new MOQA random structures can cover all of the fixed po-structures in a split po-class's set.

**Notation 53.** *Let $SPC_{\beta_{max},n}$ denote a split po-class, with $n$ as in Definition 33.*

**Definition 34** (General split po-class)**.** *A general split po-class is*

$$\bigcup_{n \in Z} SPC_{\beta_{max},n},$$

*where $Z \subseteq \{0, 1, 2, \ldots\}$.*

**Notation 54.** *Let $GSPC_{\beta_{max},Z}$ denote a general split po-class, with $Z$ as in Definition 34.*

**Notation 55.** *Let $GSPC_{\beta_{max},\infty}$ denote $GSPC_{\beta_{max},Z}$ when $Z = \{0, 1, 2, \ldots\}$.*

$GSPC_{\beta_{max},\infty}$ represents all of the fixed po-structures that can result after the MOQA split function is applied to a discrete inductive po-class with an infinite set. (Note that the MOQA split function does not modify input when it is an empty fixed po-structure or a fixed po-structure of size one.) $Z$ is to $GSPC_{\beta_{max},Z}$ what $R(IPC_\beta)$ is to $IPC_\beta$. More information regarding split and general split po-classes can be found in Section 4.4.3. However, it is important

to draw attention here to the fact that the general split po-class concept comes from research carried out by Hickey [35].

**Definition 35** (Compound structure)**.** *A compound structure is obtained from finite numbers of $FPS_\beta$s, $IPC_\beta$s, $SPC_{\beta_{max},n}$s, and $GSPC_{\beta_{max},Z}$s through successive iterations of the operations $||$ and $\otimes$.*

Note that the finite numbers in the above definition include zero. A compound structure can be used to represent all of the possible outputs from a MOQA function when these outputs cannot be represented by a fixed po-structure or by *one* of the above classes, i.e. by one inductive po-class or by one split po-class or by one general split po-class. The driver for this new data structure type is to allow MOQA functions involving both a fixed and an unknown number of nodes to be applied to the same composite variable during its lifetime in some program. Section 4.4.4 provides a more detailed explanation of compound structures, along with examples.

**Notation 56.** *Let $CS$ denote a compound structure.*

**Notation 57.** *Let $CS_\beta$ denote a compound structure and some label ordering $\beta$ on it.*

Each item from which a compound structure is obtained is associated with a label ordering. In MOQA all of these label orderings are the same, see Section 2.2. Hence, the label ordering that is common throughout the compound structure can be part of its notation.

**Notation 58.** *Let $P_\beta$ denote $CS_\beta$.*

Due to the compound structure definition, a salient point worth noting here is that when $CS_\beta$ represents just one $FPS_\beta$, $IPC_\beta$, $SPC_{\beta_{max},n}$ or $GSPC_{\beta_{max},Z}$, then $P_\beta$ can be said to denote that $FPS_\beta$, $IPC_\beta$, $SPC_{\beta_{max},n}$ or $GSPC_{\beta_{max},Z}$.

**Notation 59.** *Let $M'$ denote an expression whose variables are $FPS$ properties.*

The number of nodes in a $FPS$, or the depth of a $FPS$ if it is a tree, are both examples of $FPS$ properties.

**Notation 60.** *Let $M'(FPS)$ denote the value of $M'$ on a specific $FPS$.*

The background for the following definition is now in place. This definition pushes the MOQA theory outside of the realm of finite data structures for the first time.

**Definition 36** (MOQA′ random structure)**.** *A MOQA′ random structure consists of a $P_\beta$, an expression $M'$ that for each $FPS_\beta$ represented by $P_\beta$ evaluates to a positive integer and, for each $FPS_\beta$ represented by $P_\beta$, a multiset containing $M'(FPS)$ copies of each canonically-ordered labeling in $L(FPS_\beta)$.*

Note that the MOQA static analysis tool should also use $R(IPC_\beta)$ to assist in recording the $FPS_\beta$s that are represented by $P_\beta$ when $P_\beta$ is $IPC_\beta$. Though the tool may individually record each $FPS_\beta$ represented by $IPC_\beta$ when its set is finite, this option is no longer available when $IPC_\beta$'s set is conditionally or unconditionally infinite. $R(IPC_\beta)$ must then be used to statically record the infinite number of fixed po-structures in $IPC_\beta$'s set.

**Notation 61.** *Let $S'$ denote a MOQA′ random structure.*

**Notation 62.** *Let $n_{S'}$ denote the size of $P_\beta$ in $S'$.*

When $P_\beta$ is $FPS_\beta$, then the size of $P_\beta$ is the size of $FPS_\beta$, i.e. the number of nodes in $FPS_\beta$. When $P_\beta$ is $IPC_\beta$, then the size of $P_\beta$ can be the size of any $FPS_\beta$ in $IPC_\beta$'s inductively defined set. When $P_\beta$ is $SPC_{\beta_{max},n}$, then the size of $P_\beta$ is $n$. When $P_\beta$ is $GSPC_{\beta_{max},Z}$, then the size of $P_\beta$ can be the size of any $FPS_{\beta,a,b}$ in $GSPC_{\beta_{max},Z}$'s set. When $P_\beta$ is $CS_\beta$, then the size of $P_\beta$ can be the size of any $FPS_\beta$ represented by $CS_\beta$; see Section 4.4.4 for the $FPS_\beta$s that a $CS_\beta$ represents.

**Definition 37** (Multiplicity of MOQA′ random structure)**.** *The multiplicity of $S'$ is the expression $M'$ in Definition 36.*

**Notation 63.** *Let $M^{S'}$ denote the multiplicity expression of $S'$.*

An example in MOQA of $M'$ is the multiplicity expression for the MOQA split function; see Section 2.2.2. Note that it is not required for $M'$ to have variables. If $M'$ does not have variables, then every $M'(FPS)$ evaluates to the same positive integer and $FPS$ is irrelevant.

**Definition 38** (MOQA′ random bag)**.** *A MOQA′ random bag is a multiset of MOQA′ random structures.*

**Definition 39** (A MOQA′ random structure preserving function)**.** *A function is MOQA′ random structure preserving if it maps a MOQA′ random structure to a multiset of one of more MOQA′ random structures.*

A MOQA′ random structure preserving function is also known as a *MOQA′ random bag preserving function*.

**Definition 40** (MOQA′-satisfying program)**.** *A MOQA′-satisfying program is a program $P$ whose composite variables can store all of their possible states at any moment during $P$ in a MOQA′ random bag.*

**Notation 64.** *Let $p'$ denote a MOQA′-satisfying program.*

**Notation 65.** *Let $MRB_{p',c,\beta,i}$ denote the MOQA′ random bag that represents all of $c$'s possible states at $i_{p',c}$.*

**Notation 66.** *Let $\mathcal{M}'$ denote $MRB_{p',c,\beta,i}$.*

**Notation 67.** *Let $MRB_{p',c,\beta}$ denote $c$'s MOQA′ random bag at the first moment that $c$ is referred to in $p'$.*

**Definition 41** (The size of a MOQA′ random bag)**.** *The size of a MOQA′ random bag is the number of MOQA′ random structures in it.*

**Notation 68.** *Let $|\mathcal{M}'|$ denote the size of $\mathcal{M}'$.*

**Notation 69.** *Let $\mathcal{M}^{D'}$ denote a MOQA′ random bag with a MOQA′ random structure whose $P_\beta$ is $P_{\beta_{max}}$, which denotes a discrete inductive po-class with an infinite set, and whose multiplicity is one.*

As for a MOQA random structure, the multisets of canonically-ordered labelings associated with a MOQA′ random structure are not explicitly recorded for each $S'$ in $\mathcal{M}'$. While previously this information was not recorded by choice, now it will not be possible to record these multisets when $P_\beta$ of $S'$ represents an unbounded number of fixed po-structures.

A MOQA′ random bag can be expanded to the following:

$$MRB_{p',c,\beta,i} \quad = \quad \{(S_1^{\mathcal{M}'}, M^{S_1^{\mathcal{M}'}}), (S_2^{\mathcal{M}'}, M^{S_2^{\mathcal{M}'}}), \ldots, (S_{|\mathcal{M}'|}^{\mathcal{M}'}, M^{S_{|\mathcal{M}'|}^{\mathcal{M}'}})\},$$

where $S_j^{\mathcal{M}'}$ is the $j^{th}$ MOQA′ random structure in $\mathcal{M}'$ and has a multiplicity of $M^{S_j^{\mathcal{M}'}}$, $1 \leq j \leq |\mathcal{M}'|$.

**Notation 70.** *Let $L(P_\beta)$ denote the set of all canonically-ordered labelings of $P_\beta$.*

When $P_\beta$ is $FPS_\beta$, then $L(P_\beta)$ is $L(FPS_\beta)$, c.f. $L(H_\beta)$. When $P_\beta$ is $IPC_\beta$, $SPC_{\beta_{max},n}$, $GSPC_{\beta_{max},Z}$ or $CS_\beta$, then $L(P_\beta)$ can be the $L(FPS_\beta)$ of any $FPS_\beta$ in the collection represented by $P_\beta$.

**Notation 71.** *Let $L(S_j^{\mathcal{M}'})$ denote $L(P_\beta)$ for the $P_\beta$ of that $S_j^{\mathcal{M}'}$.*

**Notation 72.** *Let $l(\mathcal{M}')$ denote the multiset union of each $S_j^{\mathcal{M}'}$'s multiset of canonically-ordered labelings.*

**Notation 73.** *Let $\overline{T}_f(P_\beta)$ denote the average number of label-to-label comparisons that result when the MOQA function $f$ is applied to $P_\beta$, when the average is taken over every labeling in the canonically-ordered set $L(P_\beta)$.*

**Notation 74.** *Let $\overline{T}_f(S_j^{\mathcal{M}'})$ denote $\overline{T}_f(P_\beta)$ for the $P_\beta$ of that $S_j^{\mathcal{M}'}$.*

**Notation 75.** *Let $\overline{T}_f(\mathcal{M}')$ denote the average number of label-to-label comparisons that result when the MOQA function $f$ is applied to the MOQA$'$ random bag $\mathcal{M}'$.*

So the formula for the average-case cost of the MOQA function $f$ when applied to the MOQA$'$ random bag $\mathcal{M}'$ only differs from Formula 2.14 on page 28 notationally.

$$\overline{T}_f(\mathcal{M}') = \sum_{j=1}^{|\mathcal{M}'|} \frac{|L(S_j^{\mathcal{M}'})|.M^{S_j^{\mathcal{M}'}}}{|l(\mathcal{M}')|}.\overline{T}_f(S_j^{\mathcal{M}'}) \tag{4.1}$$

It has just been established that there may be multiple distinct $L(S_j^{\mathcal{M}'})$s when $P_\beta$ is $IPC_\beta$, $SPC_{\beta_{max},n}$, $GSPC_{\beta_{max},Z}$ or $CS_\beta$. When this is so, then there may be some confusion in the reader's mind regarding which set is used for $\overline{T}_f(P_\beta)$. Additionally, if at least two of these sets have distinct cardinalities, then that confusion may also arise regarding the value of $|L(S_j^{\mathcal{M}'})|$ in Formula 4.1. However, by the end of this chapter, it should be clear how this work addresses this issue.

**Definition 42** (MOQA$'$ control flow graph)**.** *A MOQA$'$ control flow graph is the control flow graph of a MOQA$'$-satisfying program.*

**Notation 76.** *Let $mcf'$ denote $mcf_{p', n_a, n_b}$ whenever a known $p'$, $n_a$ and $n_b$ are being dealt with.*

**Notation 77.** *Let $mcf'^o$ denote $mcf_{p', n_a, n_b}$ when it is an originating control flow path.*

**Notation 78.** *Let $\mathcal{M}^{mcf'(\mathcal{M}')}$ denote the MOQA′ random bag after program $p'$ follows $mcf'$ from the MOQA′ random bag $\mathcal{M}'$.*

Definitions $FPS_\beta$, $IPC_\beta$, $SPC_{\beta_{max}, n}$, $GSPC_{\beta_{max}, Z}$ and $CS_\beta$ have been introduced in this section. The next sections shall explore these data structure representations further. Particular attention is paid to $IPC_\beta$ as it is an important component in any average-case analysis technique and, with the extension here of the MOQA theory, its relation to MOQA can now be formally considered.

## 4.4 Data Structures Represented by $P_\beta$

### 4.4.1 Fixed Po-structure

When $MRB_{p', c, \beta}$ is $\mathcal{M}^{D'}$ and $|I((F_{mcf'^o, c}))| - |D((F_{mcf'^o, c}))|$ can be reduced by the MOQA static analysis tool to a specific integer, then each $P_\beta$ in $\mathcal{M}^{mcf'^o(\mathcal{M}^{D'})}$ is a fixed po-structure. Recall that all the distinct canonically-ordered labelings of each of these $P_\beta$s are equally likely.

An example of a MOQA′ random bag in which each $P_\beta$ is a fixed po-structure can be shown using the MOQA′ program $p'_1$ in Figure 4.2, which is written in the correct syntax of *MOQA-Java*; see Chapter 3. The code is a sequence of three MOQA functions: a MOQA product function involving three nodes, a MOQA deletion function involving one node, finishing with a MOQA product function involving three nodes, two that were involved in the previous MOQA product function and one new node. Let $c$ denote the composite variable `oc` and let $MRB_{p'_1, c, \beta}$ denote $\mathcal{M}^{D'}$. Let $mcf'^o_x$ denote $mcf_{p'_1, n_a, n_a}$, the only originating control flow path through $p'_1$ of length one when entry and exit blocks are ignored. The last instruction in this path's basic block $n_a$ is the final MOQA product function in $p'_1$.

There are two MOQA′ random structures in $\mathcal{M}^{mcf'^o_x(MRB_{p'_1, c, \beta})}$. The fact that there are two of them is due to $p'_1$'s deletion function. $p'_1$'s deletion

```
/**
 * A simple example.
 * @param oc a discrete OrderedCollection of size greater than
 * three.
 */
public <L extends Comparable<L>, G> void
    eg(OrderedCollection<L, G> oc) {
    Iterator<NodeInfo<L, G>> iter = oc.getDirectNodeInfoIter();
    OrderedCollectionSubset<L, G> insert1 =
        oc.product(iter.next(), iter.next(), MARKER, iter.next());
    NodeInfo<L, G> delete1 = insert1.removeMaximum();
    OrderedCollectionSubset<L, G> insert2 =
        oc.product(insert1, iter.next());
}
```

Figure 4.2: A simple example in *MOQA-Java*, program $p'_1$

function is only ever executed on a particular fixed po-structure, a v-shaped Hasse diagram of size three, and it selects one of the structure's two maximal nodes for deletion. Both these nodes have an equal chance of selection for deletion because the deletion function deletes the node with the largest label and all the distinct canonically-ordered labelings of the fixed po-structure in question are function input for a constant number of program run-times. So two fixed po-structures are equally likely to result from $p'_1$'s deletion function. These are shown in Figure 4.3, the v-shaped Hasse diagram when the rightmost maximal node has been deleted and the v-shaped Hasse diagram when the leftmost maximal node has been deleted. The final MOQA product function connects another node below these fixed po-structures to give two total orders of size three, the $P_\beta$s in $\mathcal{M}^{mcf'^o_x(MRB_{p'_1,c,\beta})}$. Both MOQA′ random structures in $\mathcal{M}^{mcf'^o_x(MRB_{p'_1,c,\beta})}$ have a multiplicity of one. So fixed po-structures are applicable when the number of nodes involved in all of a MOQA′ program's product and deletion functions can be statically determined as a fixed value, which is when this information is completely independent of the number of program run-times.

The current MOQA static analysis tool utilises a technique called condensed representation [35]. Its definition, slightly modified for consistency with the terminology here, is as follows.

**Definition 43** (Condensed representation)**.** *A condensed representation is a series-parallel partial order representation that contains a component which is*



Figure 4.3: One of these fixed po-structures will result after the MOQA deletion function in Figure 4.2

*an embedded MOQA random bag. It thus is a single representation for one or more MOQA random structures in a MOQA random bag, the number of which depends on the number of MOQA random structures in the MOQA random bag.*

The purpose of a condensed representation is to more efficiently record multiple fixed po-structures in the same MOQA random bag when they have a large amount of structural similarity. For example, assume that fixed po-structures I and II in Figure 4.4 are in the same MOQA random bag. Instead of individually representing each of these fixed po-structures in the MOQA random bag, Hickey [35] would represent them once, the condensed representation III of Figure 4.4 with its box symbolising the embedded MOQA random bag. Clearly, letting $H_\beta$ of $S$ be a condensed representation breaks Schellekens's definition [63] of a MOQA random bag though statically the result is that "there is a large saving in terms of the space required"[35]. Therefore, $P_\beta$ of $S'$ can be extended to also denote a condensed representation, allowing this useful implementation technique to be in harmony with the expansion of the MOQA theory in Section 4.3.

### 4.4.2 Inductive Po-class

As discussed in Section 4.3, there are times when the MOQA static analysis tool is not able to reduce $|I((F_{mcf'^o,c}))| - |D((F_{mcf'^o,c}))|$ to a specific integer. When this is so, then it may be that some or all of the data structures constructed/modified by $(F_{mcf'^o,c})$ can be represented by an inductive po-class. The potential value of inductive po-classes in this situation is that they would allow the MOQA static analysis tool to continue tracking the state and distribution of a program's data structures. Later on, careful consideration will be given to when inductive po-classes are actually suitable for use in the MOQA context.

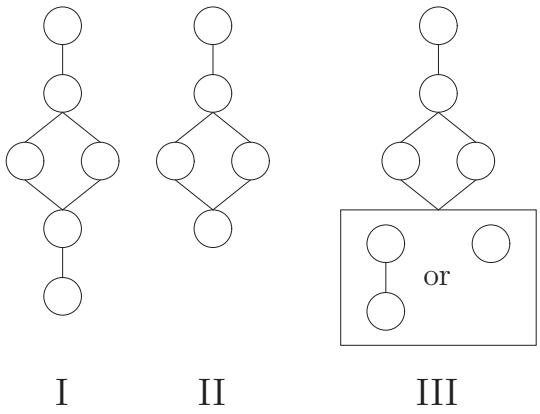



Figure 4.4: I) A fixed po-structure, II) A fixed po-structure and III) A condensed representation of I and II

Definition 32 described an inductive po-class, a set of fixed po-structures whose structural definition is in terms of itself. This work *always* uses its own extended version of the original Backus-Naur Form (BNF) notation to express structural definitions. This extended BNF notation is the original BNF notation plus parentheses. The motivation behind including parentheses is to indicate operator precedence and is purely syntactic sugar because adding parentheses to the original BNF notation does not increase its power of expression. Parentheses simply allow for fewer production rules, in the same way that the Kleene star does in other extended BNF notations. For example, the following structural definition of an inductive po-class is expressed in this work's extended BNF notation and makes use of parentheses.

$$\begin{aligned} < R > &::= () \\ < R > &::= ((< number > \otimes < number >) \otimes < number >) \otimes < R > \end{aligned}$$

Note that () in the first production rule is the empty string. This structural definition can also be expressed in just the original BNF notation:

$$\begin{aligned} < Q > &::= () \\ < Q > &::= < R > \otimes < Q > \\ < R > &::= < S > \otimes < number > \\ < S > &::= < number > \otimes < number > \end{aligned}$$

As standard, the production rules in a structural definition are written as *head* ::= *body*, where the left-hand side, the head, may be replaced by the right-hand side, the body.

Note that there is an important distinction between an inductive po-class and the structural definition of this inductive po-class. The first is a set of fixed po-structures that are potentially of different sizes and this set can be infinite. Hence, knowing that a fixed po-structure belongs to an inductive po-class does not mean that the number of elements in the partial order is known. The second is a set of structural rules used to construct each member of the inductive po-class. So this structural definition determines which fixed po-structures *can* be members of the inductive po-class but not which ones actually *are* members. It has no influence over member size or quantity. These attributes depend on the context in which the inductive po-class was formed.

Note also that when $P_\beta$ of $S_j^{\mathcal{M}'}$, which is the $j^{th}$ MOQA′ random structure in the MOQA′ random bag $\mathcal{M}'$, is an inductive po-class, each fixed po-structure in its inductively defined set has the same label ordering, which is $\beta$, and all the distinct canonically-ordered labelings of each possible fixed po-structure in that set are equally likely.

Inductive po-classes are similar to inductively defined types. These are commonly found in functional programming languages, such as Haskell [33] and ML [53], where they are known as algebraic types. They are also known as recursive data structures [39]. In the MOQA book [63] a total of three inductively defined types are mentioned: discrete, linear list and binary tree. Not considered in the MOQA book [63] is how inductively defined types fit into the MOQA theory, as the MOQA theory was not developed with them in mind, which was a strong motivation for the formal extension of the MOQA theory in Section 4.3. This work will show that there is a difference between the set of all inductively defined types and the set of all inductively defined types possible in MOQA. The other inductive po-class related definitions which occur later on in this work will assist in detailing the distinction between these two sets.

So to date one inductive po-class has already been seen, $\mathcal{M}^{D'}$, which is a discrete inductive po-class whose lower size bound is zero. A lower size bound of zero means that this inductive po-class represents all discrete partial orders including the empty discrete partial order. A simple example of a MOQA′

```
1  /**
2   * A simple example.
3   * @param oc a discrete OrderedCollection of any size.
4   */
5  public <L extends Comparable<L>, G> void
6      eg(OrderedCollection<L, G> oc) {
7      Iterator<NodeInfo<L, G>> iter = oc.getDirectNodeInfoIter();
8      for (int i = 1; i < oc.size()/2; i++)
9          insert = oc.product(insert, iter.next());
10 }
```

Figure 4.5: A simple example in *MOQA-Java*, program $p'_2$

random bag that contains another inductive po-class whose lower size bound is zero can be shown using the MOQA′ program $p'_2$ in Figure 4.5. Let $c$ denote the composite variable `oc` and let $\mathcal{M}_{p'_2,c,\beta}$ denote $\mathcal{M}^{D'}$. Figure 4.6 shows the MOQA′ control flow graph of $p'_2$. Let $mcf'^{o}_{x}$ denote $mcf_{p'_2,n_1,n_5}$, an originating control flow path through $p'_2$. Finally, let $q$ denote $n_{S_1^{\mathcal{M}^{D'}}}$ during the course of this example[3].

The length of $mcf'^{o}_{x}$ depends on the number of times the "yes" edge is traversed. For example, if this edge is traversed twice, then the length of $mcf'^{o}_{x}$ is eight, $(n_1, n_2, n_3, n_4, n_3, n_4, n_3, n_5)$. In turn, the number of times the "yes" edge is traversed/the number of times basic block $n_4$ is executed depends on $q$. This value cannot be resolved to a specific integer because what it depends on cannot be resolved to a specific integer; $q$ can be any value greater than or equal to zero so it can be any one of an infinite number of possible values. So an inductive po-class is used to represent the infinite number of data structures that can be constructed/modified by $(F_{mcf'^{o}_{x},c})$. This inductive po-class is the linear list inductive po-class with the label ordering $\beta_{max}$ on it. Its structural definition is:

$$\begin{aligned} < LIST > &::= () \\ < LIST > &::= < number > \otimes < LIST > \end{aligned}$$

The size of the linear list inductive po-class when it is the $P_\beta$ of the one MOQA′

[3]Notationally, the node $n_i$ in a control flow graph differs to $n_{S'}$, which denotes the size of $P_\beta$ in $S'$, as the former's subscript is an integer and the latter's subscript is a compound structure. Nonetheless, $q$ will denote $n_{S_1^{\mathcal{M}^{D'}}}$ in this example to eliminate any possible confusion that may arise between the two notations.

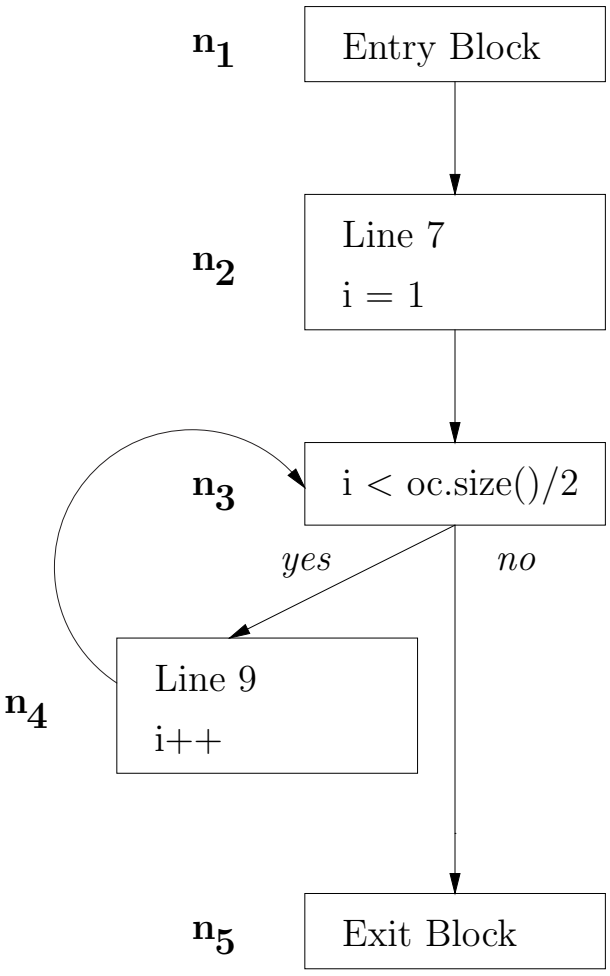



Figure 4.6: Control flow graph of Figure 4.5

random structure in $\mathcal{M}^{mcf'^{o}_{x}(\mathcal{M}_{p'_2,\,c,\,\beta})}$ is:

$$\lfloor \frac{q}{2} \rfloor.$$

Hence, $p'_1$ ignores

$$\lceil \frac{q}{2} \rceil$$

of the nodes in $S_1^{\mathcal{M}_{p'_2,\,c,\,\beta}}$.

However, could fixed po-structures have been used in this example instead of inductive po-classes? No. To explain why not, it is worthwhile answering a more general question first. What does the set of fixed po-structures associated with the definition of an inductive po-class epitomise? Recall that a MOQA random bag represents *all* the possible fixed po-structures that a composite variable can take at some moment. So when all program run-times are considered, the composite variable is each of these structures for at least one run-time. Likewise, when all program run-times are considered, the composite variable's size is each of the distinct sizes of these structures for at least one run-time. Therefore, the MOQA′ random structure whose $P_\beta$ is an inductive po-class can be viewed as a logically disjunctive series of distinct MOQA′ random bags nested within the overall MOQA′ random bag. Each MOQA′ random bag in the nested series contains only fixed po-structures of the same size, which are all the fixed po-structures of that size in the inductive po-class's set, and in the series there is a MOQA′ random bag for each distinct fixed po-structure size found in the inductive po-class's set. The key point is that only *one* of these MOQA′ random bags contains the possible fixed po-structures that the composite variable can take at that moment.

So replacing $LIST$ in the above example with fixed po-structures would yield an infinite series of MOQA′ random bags nested within $\mathcal{M}^{mcf'^{o}_{x}(\mathcal{M}_{p'_2,\,c,\,\beta})}$. It is clearly impossible to explicitly represent each individual element in an infinite set of fixed po-structures/MOQA′ random bags statically. So such an infinite number of fixed po-structures/MOQA′ random bags are concisely represented in a MOQA′ random bag by one $P_\beta$, an inductive po-class.

An inductive po-class can be further classified into one of two sub-types: deterministic or non-deterministic.

**Definition 44** (Deterministic inductive po-class)**.** *An inductive po-class is said to be a deterministic inductive po-class if, for every given size, there is at most one fixed po-structure of that size in the set.*

**Notation 79.** *Let $DIPC$ denote a deterministic inductive po-class.*

**Notation 80.** *Let $DIPC_\beta$ denote a deterministic inductive po-class and some label ordering $\beta$ on it.*

**Definition 45** (Non-deterministic inductive po-class)**.** *An inductive po-class is said to be a non-deterministic inductive po-class if there is at least one size for which there is more than one fixed po-structure of that size in the set.*

**Notation 81.** *Let $NDIPC$ denote a non-deterministic inductive po-class.*

**Notation 82.** *Let $NDIPC_\beta$ denote a non-deterministic inductive po-class and some label ordering $\beta$ on it.*

An example of a $DIPC_{\beta_{max}}$ has just been highlighted, the linear list inductive po-class with the label ordering $\beta_{max}$ on it. When $P_\beta$ of $S_j^{\mathcal{M}'}$ is a linear list inductive po-class, then there is exactly one fixed po-structure for any $n_{S_j^{\mathcal{M}'}}$. For a $NDIPC$, consider the following structural definition:

$$\begin{aligned} 1.\ &< BT > \ ::= \ () \\ 2.\ &< BT > \ ::= \ < number > \otimes (< BT > || < BT >) \end{aligned}$$

This is the structural definition of a *binary tree inductive po-class*. The fixed po-structures in the set belonging to a binary tree inductive po-class correspond to conventional binary trees with $< number >$ at the root and whose left and right $< BT >$ operands of $||$, respectively. So, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is a binary tree inductive po-class, then the number of distinct fixed po-structures for any $n_{S_j^{\mathcal{M}'}}$ depends on the value of $n_{S_j^{\mathcal{M}'}}$. For example, a binary tree of size two can be one of the two distinct fixed po-structures in Figure 4.3, whereas a binary tree of size three can be one of the five distinct fixed po-structures in Figure 4.7.

The fact that the fixed po-structures in the set belonging to a binary tree inductive po-class correspond to conventional binary trees raises an important

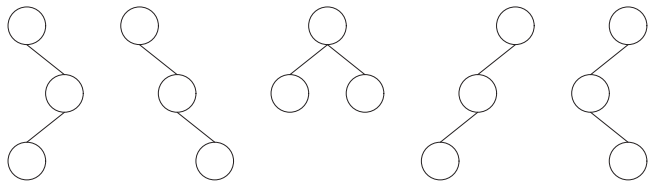


Figure 4.7: The five distinct BSTs of size three

new concept in this work — that a fixed po-structure in an inductive po-class's set is identified not only by shape, which of course is related to size, but also by the precise sequence of production rules which were used to derive it. For example, a DAG has no intrinsic concept of "left" or "right" children, yet by taking into account the actual derivation of the DAG, these concepts now become meaningful. Therefore, in general, *any two fixed po-structures constructed from the structural definition of an inductive po-class are distinct if they are constructed via distinct production rule sequences.* This can be expanded on a little further. Let $A \Rightarrow_i B$ denote that string $B$ is generated from string $A$ by applying production rule $i$ to the *leftmost* applicable non-terminal in $A$. Then Figure 4.8 shows the *unique* sequence of production rules that constructed the far left fixed po-structure in Figure 4.7; $BT$'s production rules are numbered in its structural definition above.

So the production rule sequence that constructed the far left fixed po-structure in Figure 4.7 is $[2, 1, 2, 2, 1, 1, 1]$. Whereas the production rule sequence that constructed the fixed po-structure second to the left in Figure 4.7 is $[2, 1, 2, 1, 2, 1, 1]$. This is why all four fixed po-structures of height two in Figure 4.7 are considered to be distinct, therefore eliminating the argument that all four of these fixed po-structures could be interpreted as the same linear list of length three. So it is this understanding of distinctness between fixed po-structures constructed from the structural definition of an inductive po-class that determines the elements of an inductive po-class's set. Also, it is the order of the terminals from left to right at each depth of nesting in a production rule sequence that is the left-to-right order of the nodes at that depth. So, for the binary tree case, this terminal order in a production rule sequence specifies the left and right children of a parent node.

Hence, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an inductive po-class, its structural definition determines the number of distinct fixed po-structures for each $n_{S_j^{\mathcal{M}'}}$. This

$$
\begin{aligned}
&< BT > && \Rightarrow_2 \\
&< number > \otimes (< BT > \,||\, < BT >) && \Rightarrow_1 \\
&< number > \otimes (() \,||\, < BT >) && \Rightarrow_2 \\
&< number > \otimes (() \,||\, (< number > \otimes (< BT > \,||\, < BT >))) && \Rightarrow_2 \\
&< number > \otimes (() \,||\, (< number > \otimes ((< number > \otimes \\
&\qquad (< BT > \,||\, < BT >)) \,||\, < BT >))) && \Rightarrow_1 \\
&< number > \otimes (() \,||\, (< number > \otimes ((< number > \otimes \\
&\qquad (() \,||\, < BT >)) \,||\, < BT >))) && \Rightarrow_1 \\
&< number > \otimes (() \,||\, (< number > \otimes ((< number > \otimes \\
&\qquad (() \,||\, ())) \,||\, < BT >))) && \Rightarrow_1 \\
&< number > \otimes (() \,||\, (< number > \otimes ((< number > \otimes \\
&\qquad (() \,||\, ())) \,||\, ())))
\end{aligned}
$$

Figure 4.8: The sequence of production rules that constructed the far left fixed po-structure in Figure 4.7

means that an inductive po-class is suitable not only when the size is not fixed but also when the number of distinct fixed po-structures for each size is not fixed.

It has been shown here how an inductive po-class captures some or all of the fixed po-structures that a composite variable can take at a particular moment. Now consider the multiplicity expression associated with an inductive po-class for the MOQA′ random structure definition. When $P_\beta$ of $S_j^{\mathcal{M}'}$ is an inductive po-class, then $M^{S_j^{\mathcal{M}'}}$ captures how often each fixed po-structure in $P_\beta$'s set occurs when it is among the possible fixed po-structures that $c$ can take at $i_{p',c}$. In other words, for each $FPS_\beta$ in this $P_\beta$'s set, $M^{S_j^{\mathcal{M}'}}(FPS)$ captures how often each labeling in $L(FPS_\beta)$, which is the set of all canonically-ordered labelings of $FPS_\beta$, occurs when $FPS_\beta$ is among the possible fixed po-structures that $c$ can take at $i_{p',c}$. For example, the multiplicity expression associated with a linear list inductive po-class captures how often a total order in its set occurs when it is a possible fixed po-structure; the expression captures how often the one canonically-ordered labeling of a total order in its set occurs when it is a possible fixed po-structure. Insertion-sort is an example of this in practise; see Section 3.2.1 to recall the insertion-sort algorithm.

Let $p'_3$ denote the insertion-sort program and let $c$ denote the composite

variable to which it is applied. Let $MRB_{p'_3,\,c,\,\beta}$ denote $\mathcal{M}^{D'}$. Let $mcf'^{\,o}_x$ denote $mcf_{p'_3,\,n_x,\,n_y}$, an originating control flow path through $p'_3$ with $n_x$ being the entry block and $n_y$ being the exit block. After insertion-sort is finished, there is one MOQA′ random structure in $\mathcal{M}^{mcf'^{\,o}_x(MRB_{p'_3,\,c,\,\beta})}$. The $P_\beta$ of this MOQA′ random structure is a linear list inductive po-class with the label ordering $\beta_{max}$ on it and its size is $n_{S_1^{\mathcal{M}^{D'}}}$. The multiplicity of this MOQA′ random structure is $n_{S_1^{\mathcal{M}^{D'}}}!$. But why is this the multiplicity of the only MOQA′ random structure in $\mathcal{M}^{mcf'^{\,o}_x(MRB_{p'_3,\,c,\,\beta})}$?

For any $FPS_\beta$ in the infinite set belonging to $P_\beta$ of $S_1^{MRB_{p'_3,\,c,\,\beta}}$, each of its $|FPS_\beta|!$ distinct canonically-ordered labelings is transformed by $p'_3$ into a sorted list when that $FPS_\beta$ is the fixed po-structure that $c$ takes at the commencement of $p'_3$. So, for any $FPS_\beta$ in the infinite set belonging to the $P_\beta$ of the only MOQA′ random structure in $\mathcal{M}^{mcf'^{\,o}_x(MRB_{p'_3,\,c,\,\beta})}$, its one distinct canonically-ordered labeling occurs $|FPS_\beta|!$ times when that $FPS_\beta$ is the fixed po-structure that $c$ takes at the completion of $p'_3$. Hence, the multiplicity expression of $n_{S_1^{\mathcal{M}^{D'}}}!$.

For this example, the above multiplicity expression is actually irrelevant because there is just one MOQA′ random structure in $\mathcal{M}^{mcf'^{\,o}_x(MRB_{p'_3,\,c,\,\beta})}$ and it is clear statically that its multiplicity expression is constant for each $FPS_\beta$ of the same size in the set belonging to its $P_\beta$. For this to be clear statically, without involving any powerful deduction techniques, the multiplicity expression is either a constant or the overall fixed po-structure size is the only variable in it. So a multiplicity expression other than one should only exist, and perhaps not even then, when there is at least one other distinct $P_\beta$ in the same MOQA′ random bag or when there is just one $P_\beta$ in the MOQA′ random bag but some of the fixed po-structures of the same size in its set occur at different frequencies. Therefore, the above multiplicity expression of the linear list inductive po-class can be changed to one. The example in Section 4.4.1 can be used to show how multiplicity expressions are dealt with when two or more of the $P_\beta$s in a MOQA′ random bag are judged to be the same. Both of the MOQA′ random structures in that $\mathcal{M}^{mcf'^{\,o}_x(MRB_{p'_1,\,c,\,\beta})}$ have a multiplicity expression of one. If the MOQA static analysis tool examined this bag and considered both of these MOQA′ random structures to be the same, then it could replace them with just one instance and sum together their multiplicity expressions, resulting in the instance having a multiplicity expression of two.

Now, with the multiplicity expression of the only MOQA′ random structure in the bag being a constant, the multiplicity expression can be safely reduced to one.

Now the focus is on multiplicity expressions associated with the more common type of inductive po-class, the non-deterministic inductive po-class. A multiplicity expression that has no variables or all of its variables are $|FPS_\beta|$ has just been discussed; so, for such a multiplicity expression, all the fixed po-structures of the same size in $IPC_\beta$'s set are equally likely to occur when that is the size of the composite variable. (Note that two distinct and equally likely fixed po-structures of the same size with the same label ordering may not have the same number of distinct canonically-ordered labelings. So, for example, one of these fixed po-structures could have four distinct canonically-ordered labelings and the other could have five, though they are both equally likely.) Attention is drawn to such multiplicity expressions because in this work they are the only kind dealt with for non-deterministic inductive po-classes. Observe that, regardless of the multiplicity expression, it is always true that all the fixed po-structures of the same size in a $DIPC_\beta$'s set are equally likely to occur when that is the size of the composite variable. This is because, for any size, there is always at most one fixed po-structure of that size in any $DIPC_\beta$'s set. This distribution pattern is also applied in a previous work that developed a framework for the automatic average-case analysis of algorithms. In Flajolet, Salvy and Zimmermann [24], inductive data types are referred to as decomposable data types and for any inductive data type "all input structures of a given size $n$ are taken equally likely". This is also known as the standard uniform tree model of combinatorics.

In all of the examples so far, an inductive po-class has been employed for representing an infinite number of fixed po-structures. However, it can also represent a finite number of fixed po-structures while still being in accord with its definition. For example, let $P_\beta$ of $S_j^{\mathcal{M}'}$ denote a binary tree inductive po-class that has seven fixed po-structures in its inductively defined set, all the binary trees of size two and three. (This $S_j^{\mathcal{M}'}$ can be viewed as two nested MOQA′ random bags; one with two MOQA′ random structures whose $P_\beta$s are the fixed po-structures of the two binary trees of size two and the other with five MOQA′ random structures whose $P_\beta$s are the fixed po-structures of the five binary trees of size three.) Clearly, this is a finite set.

Note that an inductive po-class that represents a finite number of fixed po-structures is not equivalent to a condensed representation. *Each* of the fixed po-structures represented by a condensed representation is among the possible fixed po-structures that a composite variable can take at some moment. Following the above reasoning, only *a subset* of the fixed po-structures represented by an inductive po-class are among the possible fixed po-structures that a composite variable can take at some moment. However, there is one situation where it is possible to convert an inductive po-class that represents a finite number of fixed po-structures into a condensed representation and vice versa. This is when there are only fixed po-structures of one particular size in the inductive po-class's set.

So, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an inductive po-class with either a finite or infinite set, the MOQA static analysis tool must also record the distinct sizes of the fixed po-structures in $P_\beta$'s set. For example, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an inductive po-class whose set is finite, then the extra information associated with $S_j^{\mathcal{M}'}$ may be a list of each of the distinct fixed po-structure sizes found in $P_\beta$'s set. Or, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an inductive po-class whose set is infinite, all that may be required is the lower size bound. (Clearly, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an inductive po-class whose set is infinite, the distinct fixed po-structure sizes in its set must be represented finitely.)

Finally, it is assumed that the structural definition of any inductive po-class is, if not already stated, tightly defined.

**Definition 46** (Tightly defined structural definition)**.** *The structural definition of an inductive po-class is tightly defined if it consists of the minimum number of production rules in which it is possible to define that inductive po-class and each of these production rule bodies consist of the minimum number of terminals and non-terminals in which it is possible to define that production rule body.*

Tightly defining the structural definition of an inductive po-class results in an *equivalent* set of rules; equivalent in the sense that both the original rules and the new rules construct exactly the same possible sets of fixed po-structures. For any given structural definition, it is clear that a tightly defined equivalent exists, though no claim is being made that the latter can be produced by an algorithm that takes the former. Recall that structural definitions

in this work are always expressed in the extended BNF notation defined at the start of this section. So tightly defining a structural definition may involve parentheses.

The above structural definition of a linear list inductive po-class exemplifies a tightly defined structural definition, unlike the following equivalent structural definition:

$$\begin{aligned} < LIST > &::= () \\ < LIST > &::= < number > \\ < LIST > &::= < number > \otimes < LIST > \end{aligned}$$

The production rule $< LIST > ::= < number >$ is redundant in this structural definition as it makes it unnecessarily verbose in capturing the description of a linear list inductive po-class. The next structural definition of a linear list inductive po-class illustrates how a production rule can be needlessly inflated.

$$\begin{aligned} < LIST > &::= () \\ < LIST > &::= (< LIST > \otimes < number >) \otimes < LIST > \end{aligned}$$

Therefore, stripping an inductive po-class's structural definition of any repetition is part of tightly defining it. The application of this definition can be seen in later sections.

### 4.4.3 Split Po-class and General Split Po-class

Section 2.2.2 showed the finite number of distinct fixed po-structures that result when a MOQA split function is applied to a discrete fixed po-structure whose multiplicity is one. Though each of the distinct fixed po-structures that result can be individually represented in the MOQA random bag via a MOQA random structure, the split po-class abbreviation was introduced in Section 4.3 so that just one MOQA′ random structure could collectively represent them in the MOQA′ random bag. In other words, instead of representing each distinct $FPS_{\beta_{max},a,b}$ and the result of $\binom{a+b}{b}$ for that $FPS_{\beta_{max},a,b}$ with the $H_\beta$ and $\mathcal{M}$ of a MOQA random structure, $SPC_{\beta_{max},n}$ and $\binom{a+b}{b}$ are represented with the $P_\beta$ and $\mathcal{M}'$ of just one MOQA′ random structure. As it is purely an abbreviation, all the distinct canonically-ordered labelings of each $FPS_{\beta_{max},a,b}$

in a split po-class's set are still equally likely. So a split po-class is a condensed representation; when the issue of efficient memory usage is left to one side, then it does not matter which representation is used by the MOQA static analysis tool.

However, there is no such freedom of choice when the MOQA split function is applied to a discrete inductive po-class whose set is infinite. This is because the output of the MOQA split function, due to the input, is infinite. Therefore, it is not statically possible to individually represent each of the fixed po-structures outputted. It is particularly relevant to consider the MOQA split function in this context because the function's inclusion in MOQA was triggered by the quicksort and quickselect algorithms. If the MOQA static analysis tool is to determine a general average-case cost for these algorithms, then being able to track the results of the MOQA split function when applied to a discrete inductive po-class whose set is infinite is compulsory. Hence, the definition of a general split po-class, i.e. the definition of $GSPC_{\beta_{max},\infty}$ in Section 4.3. This is the motivation behind the same concept minus $Z$ in Hickey's earlier research [35], which is defined there as $star[n]$. So if the MOQA split function is applied to a discrete inductive po-class with an infinite set whose multiplicity is one, then the resulting $GSPC_{\beta_{max},\infty}$ and $\binom{a+b}{b}$ are represented with the $P_\beta$ and $\mathcal{M}'$ of just one MOQA$'$ random structure. All the distinct canonically-ordered labelings of each $FPS_{\beta_{max},a,b}$ in a general split po-class's set are also equally likely.

In addition to motivation, a general split po-class has much in common with an inductive po-class. Like an inductive po-class, the set of a general split po-class can be finite; this will occur when $N_x$ is finite. Furthermore, only all the fixed po-structures of some unknown but specific size in the general split po-class's set are among the possible fixed po-structures that the composite variable can take at that moment. This principle is explained in detail for inductive po-classes in Section 4.4.2. Hence, a general split po-class is only ever equivalent to a condensed representation when the cardinality of $N_x$ is one. Therefore, with the exception of this case, the fixed po-structures in the finite set belonging to a general split po-class cannot be individually represented in the MOQA$'$ random bag via a MOQA$'$ random structure. While this may be technically possible when the set is finite, it is not technically accurate because not all of them can be possible values for the composite variable.

However, despite sharing traits, an inductive po-class cannot be considered as a replacement for a general split po-class because no such structural definition can represent the set of fixed po-structures represented by a general split po-class.

Note that the multiplicity expression associated with either a split po-class or a general split po-class will involve $\binom{a+b}{b}$. These variables are the sizes of strict subsets of the fixed po-structure selected from the set. Due to this, the result of $\binom{a+b}{b}$, and hence the entire multiplicity expression, will vary for fixed po-structures of the same size in the set. So all the fixed po-structures of the same size in the set belonging to either a split po-class or a general split po-class are not equally likely to occur. This is where a general split po-class deviates from the inductive po-classes discussed in this work.

As a final note, though both the split po-class and general split po-class are defined in terms of the max-heap label ordering, they could just as equally have been defined in terms of the min-heap label ordering.

### 4.4.4 Compound Structure

Section 4.3 introduced the compound structure. A compound structure is an amalgamation of data structure representations, selected from amongst fixed po-structures, split po-classes, general split po-classes and inductive po-classes, joined in parallel and/or series. It can be used for the $P_\beta$ of a MOQA′ random structure when all possible outputs of a MOQA function cannot be symbolised by just one of these data structure representations, as demonstrated by the following example.

The MOQA′ program $p'_4$ in Figure 4.9 is an extension of $p'_2$ in Section 4.4.2; lines 9 and 10 are the new additions. Let $c$ denote the composite variable `oc`. Let $\mathcal{M}_{p'_4,\,c,\,\beta}$ denote a MOQA′ random bag with one MOQA′ random structure when the multiplicity of this MOQA′ random structure is one and its $P_\beta$ is $P_{\beta_{max}}$, which denotes a discrete inductive po-class with an infinite set whose lower size bound is three. Figure 4.10 shows the MOQA′ control flow graph of $p'_4$. Let $mcf'^{\,o}_x$ denote $mcf_{p'_4,\,n_1,\,n_5}$, an originating control flow path through $p'_4$. Finally, let $q$ denote $n_{S_1^{\mathcal{M}D'}}$ during the course of this example[4].

[4]As noted in Section 4.4.2, the node $n_i$ in a control flow graph differs notationally to $n_{S'}$, which denotes the size of $P_\beta$ in $S'$, as the former's subscript is an integer and the latter's subscript is a compound structure. Nonetheless, $q$ will denote $n_{S_1^{\mathcal{M}D'}}$ in this example to

```
1  /**
2   * A simple example.
3   * @param oc a discrete OrderedCollection of size greater than
4   * two.
5   */
6  public <L extends Comparable<L>, G> void
7      eg(OrderedCollection<L, G> oc) {
8      Iterator<NodeInfo<L, G>> iter = oc.getDirectNodeInfoIter();
9      OrderedCollectionSubset<L, G> insert =
10         oc.product(iter.next(), MARKER, iter.next(), iter.next());
11     for (int i = 1; i < oc.size()/2; i++)
12         insert = oc.product(insert, iter.next());
13 }
```

Figure 4.9: A simple example in *MOQA-Java*, program $p'_4$

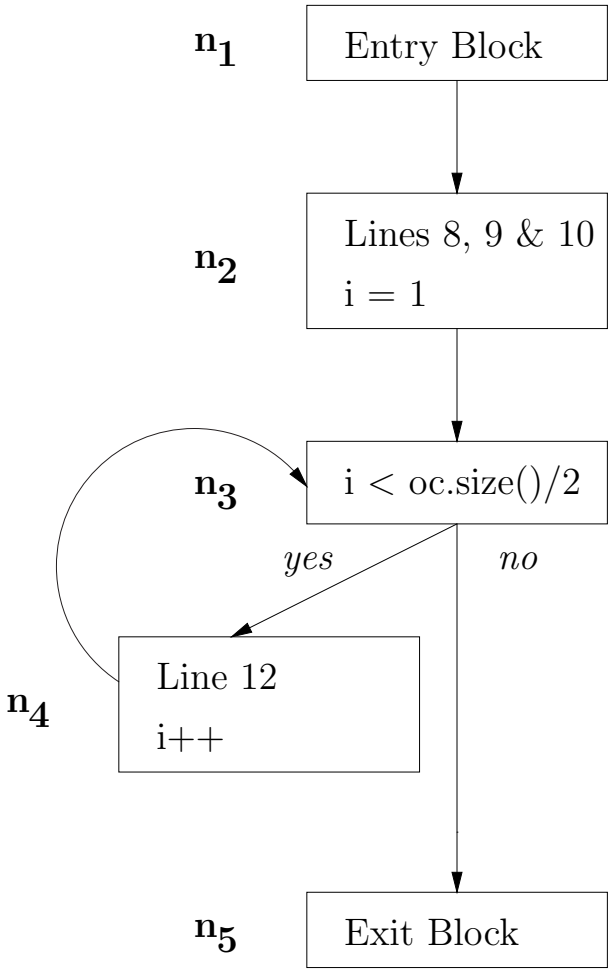



Figure 4.10: Control flow graph of Figure 4.9

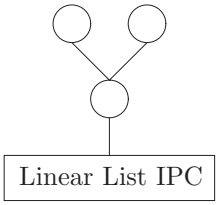



Figure 4.11: $VLIST$ — a compound structure

Lines 9 and 10 in Figure 4.9 construct a v-shaped fixed po-structure of size three with the label ordering $\beta_{max}$ on it. Producted below this fixed po-structure is the linear list inductive po-class constructed by the for loop at lines 11 and 12, which also has the label ordering $\beta_{max}$ on it. Let $VLIST_{\beta_{max}}$ denote this compound structure, the v-shaped fixed po-structure in series above the linear list inductive po-class, illustrated in Figure 4.11. The size of $VLIST_{\beta_{max}}$ when it is the $P_\beta$ of the one MOQA′ random structure in $\mathcal{M}^{mcf'^{o}_{x}(\mathcal{M}_{p'_4,\,c,\,\beta})}$ is:

$$3 + \lfloor \frac{q}{2} \rfloor .$$

Hence, $p'_4$ ignores

$$\lceil \frac{q}{2} \rceil - 3$$

of the nodes in $S_1^{\mathcal{M}_{p'_4,\,c,\,\beta}}$.

So this compound structure represents an infinite set of fixed po-structures. Each fixed po-structure in the set is a v-shaped fixed po-structure of size three producted above a linear list fixed po-structure of some size greater than or equal to zero and each fixed po-structure in the set has the label ordering $\beta_{max}$ on it. As the composition of $VLIST$ includes an inductive po-class, only all the fixed po-structures of the same but unknown size in that infinite set are among the possible fixed po-structures that $c$ can take at the completion of $p'_4$. In general, whether or not a compound structure represents a finite or infinite set of fixed po-structures and whether or not all of these fixed po-structures are among the fixed po-structures that a composite variable can take at some particular moment will depend on the composition of the compound structure.

eliminate any possible confusion that may arise between the two notations.

However, when $P_\beta$ of $S_j^{\mathcal{M}'}$, which is the $j^{th}$ MOQA′ random structure in the MOQA′ random bag $\mathcal{M}'$, is a compound structure, then it is always the case that all the distinct canonically-ordered labelings of each possible fixed po-structure that it represents are equally likely.

So compound structures, a concept not found in the MOQA book [63], will enable the current MOQA theory to represent the possible outputs of a MOQA function for additional situations. Aside from representing the possible outputs of a MOQA function, there is another situation where compound structures will prove useful. To explain, return to $p'_1$ of Figure 4.2 from Section 4.4.1. For this MOQA′ program let $\mathcal{M}_{p'_1,\,c,\,\beta}$ denote a MOQA′ random bag with one MOQA′ random structure when the multiplicity of this MOQA′ random structure is one and its $P_\beta$ is $P_{\beta_{max}}$, which denotes a discrete inductive po-class with an infinite set whose lower size bound is four. Now examine this $c$'s MOQA′ random bag after the first MOQA product function involving three nodes. It should contain one MOQA′ random structure. Its $P_\beta$ should represent the fixed po-structure of size three that is constructed by the MOQA product function and *also* what is remaining of the discrete inductive po-class from which the three nodes were selected, which is now a discrete inductive po-class with an infinite set whose lower size bound is one. The only data structure representation capable of capturing this information is a compound structure. So the $P_\beta$ of the MOQA′ random structure arising from the first MOQA product function should actually be the compound structure that is this fixed po-structure and discrete inductive po-class in parallel. The same logic applies to the example programs in Section 4.4.2. At any moment during their construction of a linear list inductive po-class, there will be one MOQA′ random structure in $c$'s MOQA′ random bag for that moment. For some or all of these moments, its $P_\beta$ should actually be the compound structure that is the linear list inductive po-class constructed so far in parallel with the discrete inductive po-class from which the linear list inductive po-class's nodes have been pulled. Therefore, continuing this reasoning, $VLIST_{\beta_{max}}$ in the example above is not the full compound structure of the $P_\beta$ of the one MOQA′ random structure in $\mathcal{M}^{mcf'^{\,o}_x(\mathcal{M}_{p'_4,\,c,\,\beta})}$. The full compound structure that this $P_\beta$ represents is $VLIST_{\beta_{max}}$ in parallel with a discrete inductive po-class that has the label ordering $\beta_{max}$ on it and whose lower size bound is zero.

So compound structures may also be required when a MOQA′ program

commences with a discrete data structure representation and then uses its nodes to construct data structures. The compound structures that may occur in this situation would represent what is constructed by the MOQA′ program in parallel with the leftover nodes of the initial discrete data structure representation, i.e. in parallel with the nodes in the initial discrete data structure representation that are ignored by the MOQA′ program.

**Notation 83.** *Let $L_j^{\mathcal{M}'}$ denote the discrete data structure representation in a compound structure from which "new" nodes are available when $P_\beta$ of $S_j^{\mathcal{M}'}$ represents a compound structure with such a discrete data structure representation.*

However, as already seen, the main focus of interest is the data structures constructed by the MOQA′ program. The discussion of the example in Section 4.4.1 honed in on the fixed po-structure of size three that was built by the first MOQA product function because this was where cost was incurred. The discrete inductive po-class in parallel with it just represents a source of new nodes. Hence, while $L_j^{\mathcal{M}'}$ should become part of the current MOQA theory so as to unify it, it is not necessary to continually reference its existence in this work. (Recall also that there is the option of the new MOQA functions in Chapter 3.3, which would allow $L_j^{\mathcal{M}'}$ to be completely dropped from the theory.) Therefore, from this point onwards, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is a $CS_\beta$ that includes $L_j^{\mathcal{M}'}$, the $L_j^{\mathcal{M}'}$ part of $CS_\beta$ will be ignored unless it is directly relevant to the discussion. Whatever is remaining in $CS_\beta$ is what $P_\beta$ shall be stated as representing, a useful though technically inaccurate abbreviation. So, returning to the example in Section 4.4.1, the $P_\beta$ of the MOQA′ random structure arising from the first MOQA function would become the fixed po-structure of size three. For the examples in Section 4.4.2, the $P_\beta$ of the one MOQA′ random structure in any MOQA′ random bag would become the linear list inductive po-class constructed so far. For the example given in this section, the $P_\beta$ of the one MOQA′ random structure in $\mathcal{M}^{mcf'^{o}_{x}(\mathcal{M}_{p'_4,\,c,\,\beta})}$ is still a compound structure but would become the one illustrated in Figure 4.11.

## 4.5 The Average-case Cost of a MOQA Function

$\mathcal{M}'$ captures all of $c$'s possible states at $i_{p',c}$. Statically calculating the average-case cost of the MOQA function applied next to $c$ involves statically calculating the MOQA function's average-case cost for each $S_j^{\mathcal{M}'}$ in $\mathcal{M}'$; see Formula 4.1. (Recall that $S_j^{\mathcal{M}'}$ denotes the $j^{th}$ MOQA$'$ random structure in the MOQA$'$ random bag $\mathcal{M}'$.) So the following sections examine when and how the average-case cost of a MOQA function can be determined for each representation of $S_j^{\mathcal{M}'}$'s $P_\beta$ that is discussed in this work: fixed po-structures, inductive po-classes, split and general split po-classes and compound structures.

### 4.5.1 The Average-case Cost of a MOQA Function Applied to $FPS_\beta$

The MOQA functions and their formulas for average-case behaviour are discussed in Section 2.2. According to how the MOQA function is defined, each MOQA function is applied to either an isolated subset or a strictly isolated subset of a series-parallel Hasse diagram. A fixed po-structure is a series-parallel Hasse diagram. Therefore, when $P_\beta$ of $S_j^{\mathcal{M}'}$, which is the $j^{th}$ MOQA$'$ random structure in the MOQA$'$ random bag $\mathcal{M}'$, is $FPS_{\beta_{max}}$, the MOQA static analysis tool can use the standard MOQA formulas to determine the average-case cost of a MOQA function when it is correctly applied to some subset of $P_\beta$.

### 4.5.2 The Average-case Cost of a MOQA Function Applied to $IPC_\beta$

Inductive po-classes whose sets are finite will be considered first, then inductive po-classes whose sets are infinite.

#### 4.5.2.1 The Average-case Cost of a MOQA Function Applied to a Finite $IPC_\beta$

Let $P_\beta$ of $S_j^{\mathcal{M}'}$, which is the $j^{th}$ MOQA$'$ random structure in the MOQA$'$ random bag $\mathcal{M}'$, denote an inductive po-class whose set is finite. Is it possible to statically determine the average-case cost of a MOQA function when it

is applied to such a $P_\beta$? Yes. One solution is to replace $S_j^{\mathcal{M}'}$ by the series of MOQA′ random bags it represents; recall Section 4.4.2. Let $k$ denote the finite number of distinct fixed po-structure sizes in the set of the original $P_\beta$. There would now be $k$ additional MOQA′ random bags nested within $\mathcal{M}'$. Each of these $k$ MOQA′ random bags contains a finite number of fixed po-structures. (This expanded $\mathcal{M}'$ is still an abbreviation for the fact that there are at least $k$ distinct, separate and equally likely MOQA′ random bags for $c$ at $i_{p',c}$; "at least $k$" because there may be other $S_j^{\mathcal{M}'}$s in $\mathcal{M}'$ that can be replaced by a nested series of MOQA′ random bags.) For each fixed po-structure in each of the $k$ new MOQA′ random bags nested within $\mathcal{M}'$, the MOQA function applied next to $c$ is applied to the relevant subset of that fixed po-structure and so its average-case cost can be determined using the standard MOQA formulas discussed in Section 2.2. Of course, the success of each average-case calculation still depends on whether the application of the MOQA function is in accord with its definition. When all of these calculations are successful, the result is $k$ possible average-case costs, one for each of the $k$ new MOQA′ random bags nested within $\mathcal{M}'$. These costs cannot be summed together as only *one* of these nested MOQA′ random bags contains the possible fixed po-structures that $c$ can take at $i_{p',c}$. The implication of arriving at multiple solutions when trying to determine the average-case cost of a MOQA function applied to some $P_\beta$ of $S_j^{\mathcal{M}'}$ is discussed in Section 4.5.2.3. For now, it is accurate to state that the MOQA static analysis tool can determine one or more average-case costs for a MOQA function that is applied to an inductive po-class whose set is finite and has the max-heap label ordering on it by determining the cost for each of the fixed po-structures that the inductive po-class represents.

#### 4.5.2.2 The Average-case Cost of a MOQA Function Applied to an Infinite $IPC_\beta$

It is more common in the average-case analysis field to evaluate algorithms for inductive po-classes whose sets are infinite than to evaluate them for inductive po-classes whose sets are finite. However, the approach advocated in Section 4.5.2.1 is not applicable when $P_\beta$ of $S_j^{\mathcal{M}'}$, which is the $j^{th}$ MOQA′ random structure in the MOQA′ random bag $\mathcal{M}'$, denotes an inductive po-class whose set is infinite. This is because the MOQA static analysis tool cannot apply the standard MOQA formulas discussed in Section 2.2 to every fixed po-structure

in the infinite set that such an inductive po-class represents. Therefore, a necessary and novel part of this work is to extend MOQA beyond the original theory [63] so as to statically determine the average-case cost of a MOQA function for inductive po-classes with infinite sets.

#### 4.5.2.3 The Average-case Cost of a MOQA Function Applied to an Infinite $DIPC_\beta$

An average-case formula for a MOQA function is a *general average-case inductive formula* if it can determine the average-case cost of the function when the function is separately applied to two or more inductive po-classes. So, a general average-case inductive formula for a MOQA function is *not* designed expressly for the unique attributes of one particular inductive po-class and therefore, unlikely to work for any other inductive po-class. It has a wider application.

There is no provision of general average-case inductive formulas for any of the MOQA functions in the MOQA book [63]. It only ever details the average-case formula of a MOQA function for a specific inductive po-class; specifically, it only ever details the average-case formula of a MOQA function for three examples.

The first example is the MOQA split function. Unlike the other MOQA functions, when the MOQA split function is applied to a fixed po-structure there is no timing reason to statically iterate through that fixed po-structure. This is because the MOQA split function must be applied to a discrete fixed po-structure only. (Depending on the implementation of the MOQA static analysis tool, there may be a correctness reason to statically iterate through the fixed po-structure that the MOQA split function is applied to — to verify that it is actually discrete.) As there is no choice regarding the shape of the fixed po-structure that the MOQA split function can be applied to, when the MOQA split function is applied to a fixed po-structure the only variable in its average-case formula is the size of that fixed po-structure. The formula is simply the size of the fixed po-structure minus one. With the other MOQA functions there is flexibility regarding both the shape and the size of the fixed po-structures that they are applied to. Therefore, the standard formulas for these MOQA functions need to iterate through the fixed po-structures when calculating average-case cost. So, heeding the definition of the MOQA split

function, the only inductive po-class that it can be applied to is the discrete inductive po-class. Hence, when $P_{\beta_{max}}$ of $S_j^{\mathcal{M}'}$, which is the $j^{th}$ MOQA′ random structure in the MOQA′ random bag $\mathcal{M}'$, denotes a discrete inductive po-class, the average-case formula for the MOQA split function applied to $P_{\beta_{max}}$ is $n_{S_j^{\mathcal{M}'}} - 1$. As the MOQA split function can be applied to only one inductive po-class, this average-case formula is not a general average-case inductive formula because it is custom-made for a specific class of data structures.

For the second example, the MOQA book [63] provides a tailor-made $\sigma_{up}$ and $\tau_{up}$ for another inductive po-class, the linear list inductive po-class and the max-heap label ordering on it. ($\sigma$ and $\tau$ are used in some MOQA average-case formulas, as detailed in Section 2.2.)

For the third example, the MOQA book [63] also provides a tailor-made $\sigma_{up}$ and $\tau_{up}$ for what it says is an inductive po-class, the complete binary tree and the max-heap label ordering on it. [5] In any case, providing hand-crafted equations for a specific class of data structures is not in the spirit of static algorithm analysis regardless of whether the class can be inductively defined or not. Such an approach would reduce the MOQA static analysis tool to nothing more than a complex book-keeping tool that does not scale well. The MOQA static analysis tool would be obtaining the average-case cost of an algorithm by performing arithmetic operations with the averages inputted for each class of data structures instead of obtaining these averages through mathematical techniques that cover multiple classes, techniques such as general average-case inductive formulas.

However, general average-case inductive formulas are found in Hickey's work [35]. The inductive po-classes covered by Hickey's general average-case inductive formulas fall under its "group structure" definition. A new exacting definition for these inductive po-classes is provided here presently, which is the empty-base $DIPC_\beta$ definition. First of all though, some preparatory definitions are needed. (Note that the rest of this section considers deterministic $IPC_\beta$s whose sets are infinite. Non-deterministic $IPC_\beta$s whose sets are infinite are considered in the following section. Like the empty-base $DIPC_\beta$ definition, this work developed the deterministic and non-deterministic $IPC_\beta$

[5] However, while a binary tree can be inductively defined, the same cannot be said for a *complete* binary tree. The structural definition of a binary tree cannot be reformulated to ensure that all tree leaves are always at the same depth.

definitions.)

**Definition 47** (Base-case production rule)**.** *The production rule $r$ in the structural definition of an inductive po-class is a base-case production rule when the body of $r$ does not contain the non-terminal that is the head of $r$ and the replacement of any non-terminal in the body of $r$ can never involve, either directly or indirectly, the non-terminal that is the head of $r$.*

**Definition 48** (Non base-case production rule)**.** *The production rule $r$ in the structural definition of an inductive po-class is a non base-case production rule when the body of $r$ contains at least one non-terminal that is the head of $r$ and/or the replacement of at least one non-terminal in the body of $r$ can involve, either directly or indirectly, the non-terminal that is the head of $r$.*

The tightly defined structural definition of the linear list inductive po-class in Section 4.4.2 helps illustrate the difference between base-case and non base-case production rules. It has one base-case production rule, the production rule whose body is empty. Its other production rule is a non base-case production rule because its body contains the non-terminal $LIST$, which is the non-terminal of its head.

Prior to the next definition, it is helpful to underscore the fact that an $IPC$'s structural definition describes not only how fixed po-structures are to be constructed but also the type of data to be stored in fixed po-structures that are constructed in this manner.

**Definition 49** (Self-identity $IPC$)**.** *An $IPC$ is a self-identity $IPC$ if, for each non base-case production rule $r$ in its structural definition, each non-terminal in the body of $r$ is either the non-terminal that is the head of $r$ or the non-terminal that represents the totally-ordered data type of the labelings on the fixed po-structures in $IPC$'s set.*

Again, the same structural definition of a linear list inductive po-class can be used as an example of a self-identity $IPC$. The body of its non base-case production rule contains two non-terminals. It has just been stated that one of these non-terminals is the non-terminal of the production rule's head. The other non-terminal is *number* and represents the totally-ordered data type of the labelings on the fixed po-structures in this linear list inductive po-class's set, i.e. each of these labelings is assembled from some set of numbers. Hence, $LIST$ is a self-identity $IPC$.

**Definition 50** (Empty-base $IPC$)**.** *An $IPC$ is an empty-base $IPC$ if its structural definition is tightly defined, it is a self-identity $IPC$ and the tightly defined structural definition has two production rules, one non base-case production rule and one base-case production rule whose body is empty.*

**Definition 51** (Multi-base $IPC$)**.** *An $IPC$ is a multi-base $IPC$ if its structural definition is tightly defined, it is a self-identity $IPC$ and the tightly defined structural definition has one non base-case production rule and either one base-case production whose body is not empty or two or more base-case production rules, one of which may have an empty body.*

**Proposition 1.** *A $DIPC$'s tightly defined structural definition has exactly one non base-case production rule.*

*Proof.* To be inductively defined is to be defined in terms of oneself. So the structural definition of any $IPC$ must have at least one production rule $r$ whose body contains a non-terminal that is the head of $r$ and/or a non-terminal whose replacement can involve, either directly or indirectly, the non-terminal that is the head of $r$. By definition, such a production rule is a non base-case production rule and so the structural definition of any $IPC$, whether it is tightly defined or not, must have at least one non base-case production rule.

One relevant characteristic of a structural definition that is tightly defined is that each production rule describes a distinct structural feature. In other words, a tightly defined structural definition does not contain multiple equivalent production rules. Now, consider the tightly defined structural definition of an $IPC$ with at least two non base-case production rules, $r_1$ and $r_2$. Also, let $[\ldots]*$ denote the production rule sequence $[\ldots]$ after all base-case production rules have been removed from it. For such an $IPC$, the production rule sequences $[r_1, r_2]*$ and $[r_2, r_1]*$ are guaranteed to be distinct because $r_1$ and $r_2$ are distinct due to tightly defining the $IPC$'s structural definition. These two distinct production rule sequences will construct two fixed po-structures of the same size when the original sequences have the same distribution of base-case production rules. For example, $[r_1, r_2, b_1, b_1]$ and $[r_2, b_1, r_1, b_1]$ will construct two fixed po-structures of the same size when $b_1$ denotes one of the $IPC$'s base-case production rules. Therefore, if the tightly defined structural definition of an $IPC$ has at least two non base-case production rules, then the $IPC$ is non-deterministic.

$$
\begin{aligned}
&< T > && \Rightarrow_2 \\
&< number > \otimes < T > && \Rightarrow_3 \\
&< number > \otimes && \\
&(((< number > || < number >) || < number >) \otimes < T >) && \Rightarrow_1 \\
&< number > \otimes && \\
&(((< number > || < number >) || < number >) \otimes ()) &&
\end{aligned}
$$

Figure 4.12: A $T$ production rule sequence that constructs a fixed po-structure of size four

For an $IPC$ to be deterministic, it must have at most one fixed po-structure of any given size in its set. So, for any fixed po-structure size that occurs in a $DIPC$'s set, there must be just one production rule sequence that can construct a fixed po-structure of that size. Therefore, there must be just *one way* to choose $s$ of the $n$ distinct non base-case production rules available to any production rule sequence that involves $s$ of these $n$ rules, $s, n \geq 1$. However, for this lower bound on $s$, $\binom{n}{s} = 1$ only when $n = 1$. Hence, if the tightly defined structural definition of an $IPC$ is to have at most one fixed po-structure of any given size in its set and accordingly be the tightly defined structural definition of a $DIPC$, then it must have exactly one non base-case production rule. □

The following structural definition illustrates the non-determinacy of an inductive po-class whose tightly defined structural definition has two non base-case production rules:

1. $< T > \ ::= \ ()$
2. $< T > \ ::= \ < number > \otimes < T >$
3. $< T > \ ::= \ ((< number > || < number >) || < number >) \otimes < T >$

The production rule sequence $[2, 3, 1]$ in Figure 4.12 and the production rule sequence $[3, 2, 1]$ in Figure 4.13 both have the same distribution of base-case production rules and both construct a fixed po-structure of size four. So $T$ is a $NDPIC$ because, for certain sizes, it has at least two fixed po-structures of the same size in its set.

$$
\begin{aligned}
&< T > && \Rightarrow_3 \\
&((< number > || < number >) || < number >) \otimes < T > && \Rightarrow_2 \\
&((< number > || < number >) || < number >) \otimes && \\
&(< number > \otimes < T >) && \Rightarrow_1 \\
&((< number > || < number >) || < number >) \otimes && \\
&(< number > \otimes ())
\end{aligned}
$$

Figure 4.13: Another $T$ production rule sequence that constructs a fixed po-structure of size four

*More than two* fixed po-structures of the same size can also be constructed when the tightly defined structural definition of an $IPC$ has at least two non base-case production rules, $r_1$ and $r_2$, i.e. when the $IPC$ is non-deterministic. For example, $[r_1, r_1, r_2, r_2]*$, $[r_1, r_2, r_1, r_2]*$, $[r_1, r_2, r_2, r_1]*$, $[r_2, r_2, r_1, r_1]*$, ..., all construct fixed po-structures of the same size when the original sequences have the same distribution of base-case production rules. Correspondingly, more than two fixed po-structures of the same size can also be constructed by distinct production rule sequences that involve more than two distinct non base-case production rules. For example, $[r_1, r_2, r_3]*$, $[r_1, r_3, r_2]*$, $[r_2, r_1, r_3]*$, $[r_2, r_3, r_1]*$, ..., all construct fixed po-structures of the same size when, again, the original sequences have the same distribution of base-case production rules.

**Proposition 2.** *The structural definition of a $DIPC$ has an equivalent set of rules that satisfy the self-identity $IPC$ requirements.*

*Proof.* Assume that the structural definition of $DIPC$ $D$ is tightly defined. It is safe to ignore the base-case production rules in $D$'s tightly defined structural definition because there are no restrictions placed on base-case production rules by the self-identity $IPC$ definition. Note that $D$'s tightly defined structural definition has exactly one non base-case production rule; Proposition 1. Note also that a non-terminal in the body of this production rule is accepted by the self-identity $IPC$ definition when it is either the non-terminal of the head or the non-terminal that represents the totally-ordered data type in question.

Consider the process behind constructing a fixed po-structure according to $D$'s structural definition. $D$'s determinism means that there is at most one fixed po-structure of any given size in $D$'s set. So, for any fixed po-

structure size that occurs in $D$'s set, there is only one way, according to $D$'s structural definition, to construct a fixed po-structure of that size. Obviously, it is $D$'s structural definition that enforces this singularness. So each of the fixed po-structures in $D$'s set is the result of $p$ applications of its only non base-case production rule and one application of one of its base-case production rules, whose application will then halt fixed po-structure construction, $p \geq 0$. Additionally, application of $D$'s only non base-case production rule will always affix the same fixed po-structure to the fixed po-structure under construction; note that affixing the same fixed po-structure for each application of $D$'s only non base-case production rule is an important contribution to $D$'s hold on its determinism. Hence, each fixed po-structure in $D$'s set is composed of $p$ repetitions of the fixed po-structure described by $D$'s only non base-case production rule and the fixed po-structure described by one of $D$'s non base-case production rules. So, $D$'s only non base-case production rule describes what this repeat fixed po-structure is and how it is to be affixed to the fixed po-structure under construction, the latter description involving the non-terminal $D$. $D$'s only non base-case production rule could capture this information with non-terminals other than those accepted by the self-identity $IPC$ definition. However, no matter how many other structural definitions this information is spread over or how convoluted these structural definitions are, they are still simply involved in describing a single fixed po-structure and its attachment to the fixed po-structure under construction. Therefore, it is always possible to replace $D$'s only non base-case production rule with an equivalent non base-case production rule that can describe this information using no other non-terminals than the non-terminal $D$ and the non-terminal that represents the totally-ordered data type in question. So, if $D$ is not already a self-identity $IPC$, then its structural definition has an equivalent set of rules that satisfy the self-identity $IPC$ requirements. □

For example, consider the following $DIPC\ U$:

$$\begin{aligned} < U > &::= () \\ < U > &::= < U > \otimes < NODE > \\ < NODE > &::= < number > \end{aligned}$$

$U$ is not a self-identity $IPC$ because the non-terminal $NODE$ is obviously not

$U$ and it is also not the totally-ordered data type of the labelings on the fixed po-structures in $U$'s set. The structural definition of $NODE$ shows that this data type is once again represented by the non-terminal *number*. However, $U$ is clearly equivalent to the linear list inductive po-class defined earlier, which is a self-identity $IPC$. Therefore, an equivalent structural definition of $U$, which satisfies the self-identity $IPC$ requirements, is as follows:

$$\begin{aligned} < U > &::= () \\ < U > &::= < U > \otimes < number > \end{aligned}$$

**Proposition 3.** *The structural definition of a $DIPC$ has an equivalent set of rules that satisfy the requirements of either an empty-base or a multi-base $IPC$.*

*Proof.* Assume that the structural definition of $DIPC$ $D$ satisfies the self-identity $IPC$ requirements; Proposition 2. Next, assume that this structural definition is tightly defined. This structural definition will be composed from base-case and non base-case production rules. It must have at least one base-case production rule because otherwise the construction of a data structure would be interminable. Moreover, it must have exactly one non base-case production rule; Proposition 1. Hence, $D$'s structural definition will satisfy the requirements of either an empty-base or a multi-base $IPC$. □

So a simplification of the above proposition, which the previous two propositions contributed to, is that a $DIPC$ can always be classified as either an empty-base or a multi-base $DIPC$.

Another way of defining a $DIPC$ is as follows. Let *op* denote either $\otimes$ or $||$. Let $FPS_x$ and $FPS_y$ denote two distinct fixed po-structures. For a specific *op* and a specific $FPS_x$, each fixed po-structure in a $DIPC$'s set is obtained from zero or more $FPS_x$s and up to one $FPS_y$ through successive iterations of *op*. $FPS_x$ is the fixed po-structure construed from the tightly defined $DIPC$'s non base-case production rule and $FPS_y$ is a fixed po-structure construed from any base-case production rule of the tightly defined $DIPC$ whose body is not empty. So, for the linear list inductive po-class example, $FPS_x$ is the fixed po-structure of size one and there is no $FPS_y$ because the body of this inductive po-class's only base-case production rule is empty. Note that $FPS_x$

is the same for all of the fixed po-structures in a $DIPC$'s set because a tightly defined $DIPC$ has exactly one non base-case production rule. This guarantee does not apply to $FPS_y$. This is because a $DIPC$ can be a multi-base $DIPC$ whose structural definition has at least two base-case production rules whose bodies are not empty. So, there could be a choice for $FPS_y$ among the fixed po-structures in such a $DIPC$'s set. $FPS_x$ is called a $DIPC$'s *repeat fixed po-structure* as every fixed po-structure in $DIPC$'s set includes zero or more $FPS_x$s. $FPS_y$ is called one of a $DIPC$'s *base-case fixed po-structures*.

**Notation 84.** *Let $SR_j^{\mathcal{M}'}$ denote $S_j^{\mathcal{M}'}$'s repeat fixed po-structure when $P_\beta$ of $S_j^{\mathcal{M}'}$ is a $DIPC_\beta$.*

**Notation 85.** *Let $p$ denote how often $SR_j^{\mathcal{M}'}$ is repeated within a fixed po-structure in $P_\beta$'s set, $p \geq 0$.*

**Notation 86.** *Let $C(S_j^{\mathcal{M}'})$ denote the set of all $S_j^{\mathcal{M}'}$'s base-case fixed po-structures when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an $IPC_\beta$.*

$P_\beta$ of $S_j^{\mathcal{M}'}$ is an $IPC_\beta$ rather than a $DIPC_\beta$ in the above notation. This generalisation is so that the notation will also hold for $NDIPC_\beta$.

**Notation 87.** *Let $C$ denote an arbitrary element of $C(S_j^{\mathcal{M}'})$.*

Note that $C(S_j^{\mathcal{M}'})$ will always be empty when $P_\beta$ of $S_j^{\mathcal{M}'}$ is an empty-base $DIPC_\beta$ and that $C(S_j^{\mathcal{M}'})$ will always be non-empty when $P_\beta$ of $S_j^{\mathcal{M}'}$ is a multi-base $DIPC_\beta$. Note also that there will never be two base-case fixed po-structures of the same size in $C(S_j^{\mathcal{M}'})$ because the inductive po-class $P_\beta$ of $S_j^{\mathcal{M}'}$ is deterministic.

The following structural definition is an example of an empty-base $DIPC$:

$$\begin{aligned} < V > &::= () \\ < V > &::= ((< number > || < number >) \otimes < number >) \otimes < V > \end{aligned}$$

Every fixed po-structure in $V$'s set can be obtained from $p$ v-shaped fixed po-structures of size three through successive iterations of $\otimes$; $V$'s repeat fixed po-structure is this v-shaped fixed po-structure of size three and its set of base-case fixed po-structures is empty. So the size of every fixed po-structure in $V$'s set is a multiple of three.

The following structural definition is an example of a multi-base $DIPC$:

$$\begin{aligned} < W > &::= () \\ < W > &::= < number > \\ < W > &::= ((< number > || < number >) \otimes < number >) || < W > \end{aligned}$$

Every fixed po-structure in $W$'s set can be obtained from $p$ v-shaped fixed po-structures of size three and up to one fixed po-structure of size one through successive iterations of $||$; $W$'s repeat fixed po-structure is also this v-shaped fixed po-structure of size three and its set of base-case fixed po-structures has one member, the fixed po-structure of size one. So the size of every fixed po-structure in $W$'s set is either a multiple of three or a multiple of three plus one.

Let $P_\beta$ of $S_j^{\mathcal{M}'}$ denote an empty-base $DIPC_\beta$ with an infinite set. So how is the general average-case inductive formula for a MOQA function applied to some subset of this $P_\beta$ determined? The answer involves the equations given immediately below, Equations 4.2 through 4.9 inclusive; recall that Section 2.2 explains $\sigma$, $\kappa$, $\tau$ and $\Delta$ for the fixed po-structure scenario. These equations originate in Hickey's research [35], which is where their proofs are also found, and must be applied under the following two conditions. First, $\beta$ still must be either max-heap or min-heap ordered, as discussed in Section 2.2, though max-heap ordered continues to be the default. Second, the subset of $P_\beta$ that the MOQA function is applied to must be obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of some specific $op$, $0 \leq r \leq p$. These equations can, where appropriate, replace the standard binary versions in a MOQA function's average-case formula and thus transform it into a general average-case inductive formula. Note that the $\sigma$, $\kappa$, $\tau$ and $\Delta$ equations are being used in the polymorphic sense, i.e. it is the number and the type of the arguments passed to an equation that determine which version is called. For example, the left-hand side of Equation 4.2 is called when three arguments are passed to $\sigma_{up}$, whereas the left-hand side of Equation 2.1 is called when just one argument is passed to $\sigma_{up}$ and that argument is not $\bullet_{\beta_{max}}$.

**Proposition 4.** *[35][$\sigma_{up}$ for an empty-base $DIPC_{\beta_{max}}$]*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, ||, r) = \sigma_{up}(SR_j^{\mathcal{M}'}) \tag{4.2}$$

$$\sigma_{up}(SR_j^{\mathcal{M}'}, \otimes, r) = r.\sigma_{up}(SR_j^{\mathcal{M}'}) + (r-1).|m(SR_j^{\mathcal{M}'})| \tag{4.3}$$

**Proposition 5.** *[35][$\kappa_{up}$ for an empty-base $DIPC_{\beta_{max}}$]*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, ||, r) = r.\kappa_{up}(SR_j^{\mathcal{M}'}) \tag{4.4}$$

$$\kappa_{up}(SR_j^{\mathcal{M}'}, \otimes, r) = \kappa_{up}(SR_j^{\mathcal{M}'}) \tag{4.5}$$

**Proposition 6.** *[35][$\tau_{up}$ for an empty-base $DIPC_{\beta_{max}}$]*

$$\tau_{up}(SR_j^{\mathcal{M}'}, ||, r) = \tau_{up}(SR_j^{\mathcal{M}'}) \tag{4.6}$$

$$\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, r) = (\tau_{up}(SR_j^{\mathcal{M}'}) + \sum_{i=1}^{r-1}(\tau_{up}(SR_j^{\mathcal{M}'}) + |m(SR_j^{\mathcal{M}'})| + \sigma_{up}(SR_j^{\mathcal{M}'}, \otimes, i)))/r \tag{4.7}$$

**Proposition 7.** *[35][$\Delta_{up}$ for deleting the label with rank $k$, i.e. the $k^{\text{th}}$ smallest label, from an empty-base $DIPC_{\beta_{max}}$]*

$$\Delta_{up}(SR_j^{\mathcal{M}'}, ||, r, k) = \frac{r.\sum_{i=1}^{|SR_j^{\mathcal{M}'}|}\binom{k-1}{i-1}.\binom{r.|SR_j^{\mathcal{M}'}|-k}{|SR_j^{\mathcal{M}'}|-1}.\Delta_{up}(SR_j^{\mathcal{M}'}, i)}{\binom{r.|SR_j^{\mathcal{M}'}|}{|SR_j^{\mathcal{M}'}|}} \tag{4.8}$$

$$\Delta_{up}(SR_j^{\mathcal{M}'}, \otimes, r, k) = \Delta_{up}(SR_j^{\mathcal{M}'}, k \bmod |SR_j^{\mathcal{M}'}|) + \left(r - \left\lceil \frac{k}{|SR_j^{\mathcal{M}'}|} \right\rceil\right). (|m(SR_j^{\mathcal{M}'})| - 1 + \Delta_{up}(SR_j^{\mathcal{M}'}, |SR_j^{\mathcal{M}'}|)) \tag{4.9}$$

Recall, from Section 2.2.3, the assumption that rank $k$ is always some

value between one and the size of the fixed po-structure under consideration. However, the $\Delta$ equations of both Hickey's work [35] and this work extend this assumption by allowing $k$ to take an additional value, which is zero. The cost of deleting a label with rank zero from any fixed po-structure will always be zero, regardless of whether the deletion is upwards or downwards. So now if the fixed po-structure under consideration is $I$, then $0 \leq k \leq |I|$.

Hickey [35] states that there are similar equations for $\sigma_{down}$, $\kappa_{down}$, $\tau_{down}$ and $\Delta_{down}$. Note that these equations can be applied to a linear list inductive po-class, thereby making the MOQA book [63] equations for this inductive po-class redundant.

So, even though an empty-base $DIPC_{\beta_{max}}$ can have an infinite number of fixed po-structures in its set, it is still possible to determine the average-case cost of the pertinent MOQA functions applied to such a $DIPC_{\beta_{max}}$ by iterating through the structure of the $DIPC_{\beta_{max}}$'s repeat fixed po-structure. But is it also possible to determine the average-case cost of a MOQA function when it is applied to a multi-base $DIPC_{\beta_{max}}$ with an infinite set?

Yes, this is possible with the following new equations that incorporate the base-case fixed po-structures of a multi-base $DIPC_{\beta_{max}}$. Now, let $P_\beta$ of $S_j^{\mathcal{M}'}$ denote a multi-base $DIPC_{\beta_{max}}$ with an infinite set. So there will be at least one base-case fixed po-structure in $C(S_j^{\mathcal{M}'})$. The other change is that the subset of this $P_\beta$ that the MOQA function is applied to must be obtained from $r$ $SR_j^{\mathcal{M}'}$s and up to one $C$ through successive iterations of some specific $op$, $0 \leq r \leq p$. So the new equations that are to follow can likewise, where appropriate, replace the standard binary versions in a MOQA function's average-case formula and thus transform it into a general average-case inductive formula.

When solving any one of these new formulas, the static selection of the correct equation from below may depend on where each fixed po-structure in $C(S_j^{\mathcal{M}'})$ is located in relation to the repetitions of $SR_j^{\mathcal{M}'}$. These base-case fixed po-structures always occur at a specific extremity of the relevant fixed po-structures in $P_\beta$'s set; the relevant fixed po-structures being those obtained from $p$ $SR_j^{\mathcal{M}'}$s and *one* $C$. The location of this specific extremity can be ascertained from $P_\beta$'s structural definition. This is illustrated by $X$'s structural

definition.

$$\begin{aligned} < X > &::= () \\ < X > &::= < number > \\ < X > &::= ((< number > || < number >) \otimes < number >) \otimes < X > \end{aligned}$$

For $X$, whose specific *op* is $\otimes$, all of its base-case fixed po-structures will occur at the "bottom" of the relevant fixed po-structures in its set. (In this example, all of $X$'s base-case fixed po-structures just amount to a total of one.) If $X$'s non base-case production rule is changed to:

$$< X > ::= < X > \otimes ((< number > || < number >) \otimes < number >),$$

then all of its base-case fixed po-structures will occur at the "top" of the relevant fixed po-structures in its set. Similarly, for a multi-base $DIPC$ whose specific *op* is $||$, its base-case fixed po-structures will either all occur to the "leftmost" or all occur to the "rightmost" of the relevant fixed po-structures in its set.

**Notation 88.** *Let $\underline{C(S_j^{\mathcal{M}'})}$ denote a set $C(S_j^{\mathcal{M}'})$ in which all of its base-case fixed po-structures occur at the bottom of the relevant fixed po-structures in the set belonging to $P_\beta$ of $S_j^{\mathcal{M}'}$.*

**Notation 89.** *Let $\overline{C(S_j^{\mathcal{M}'})}$ denote a set $C(S_j^{\mathcal{M}'})$ in which all of its base-case fixed po-structures occur at the top of the relevant fixed po-structures in the set belonging to $P_\beta$ of $S_j^{\mathcal{M}'}$.*

The new $\sigma_{up}$, $\kappa_{up}$, $\tau_{up}$ and $\Delta_{up}$ equations for a multi-base $DIPC_{\beta_{max}}$ can now be introduced.

**Proposition 8** ($\sigma_{up}$ for a multi-base $DIPC_{\beta_{max}}$ when *op* is $||$)**.** *If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ and $r > 0$, then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, ||, r) = \sigma_{up}(SR_j^{\mathcal{M}'}). \tag{4.10}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ and $r = 0$, i.e. $\sigma_{up}$ is applied to the empty fixed po-structure,*

*then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, ||, 0) \;=\; 0. \tag{4.11}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$, then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, C, ||, r) \;=\; \frac{r.|SR_j^{\mathcal{M}'}|.\sigma_{up}(SR_j^{\mathcal{M}'}) \;+\; |C|.\sigma_{up}(C)}{r.|SR_j^{\mathcal{M}'}| \;+\; |C|}. \tag{4.12}$$

*Proof of Equations 4.10 and 4.11.* The proof of Equation 4.2, which is Equation 4.10 but without the condition that $r > 0$, is supplied by Hickey [35]. This work introduces this extra condition because Equation 4.2 [35] should only be used when $r > 0$. When $r = 0$, then the result should be 0, as stated by Equation 4.11, not $\sigma_{up}(SR_j^{\mathcal{M}'})$. □

*Proof of Equation 4.12.* This proof is an adaptation of the $\sigma_{up}(A, ||, B)$ proof in the MOQA book [63]. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$. The average number of comparisons made in pushing up the label of one of $I$'s minimal nodes when that label is greater than any of $I$'s other labels depends only on the labeling of $I$ and not on the label set. Therefore, the average number of comparisons to push such a label up through $I$ is $\sigma_{up}(SR_j^{\mathcal{M}'})$ if the label is pushed up through one of the $r$ $SR_j^{\mathcal{M}'}$s and $\sigma_{up}(C)$ if it is pushed up through $C$. So Equation 4.12 is the weighted average of $r.\sigma_{up}(SR_j^{\mathcal{M}'})$ and $\sigma_{up}(C)$. □

**Proposition 9** ($\sigma_{up}$ for a multi-base $DIPC_{\beta_{max}}$ when *op* is $\otimes$)**.** *If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r > 0$, then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, \otimes, r) \;=\; r.\sigma_{up}(SR_j^{\mathcal{M}'}) + (r-1).|m(SR_j^{\mathcal{M}'})|. \tag{4.13}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r = 0$, i.e. $\sigma_{up}$ is applied to the empty fixed po-structure, then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, \otimes, 0) \;=\; 0. \tag{4.14}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ and $C \in \underline{C(S_j^{\mathcal{M}'})}$, then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r) \quad = \quad \sigma_{up}(C) + r.\sigma_{up}(SR_j^{\mathcal{M}'}) + r.|m(SR_j^{\mathcal{M}'})|. \tag{4.15}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r > 0$, then:*

$$\begin{aligned}\sigma_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r) \quad = \quad & r.\sigma_{up}(SR_j^{\mathcal{M}'}) + \sigma_{up}(C) + (r-1).|m(SR_j^{\mathcal{M}'})| + \\ & |m(C)|.\end{aligned} \tag{4.16}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r = 0$, then:*

$$\sigma_{up}(SR_j^{\mathcal{M}'}, C, \otimes, 0) \quad = \quad \sigma_{up}(C). \tag{4.17}$$

*Proof of Equations 4.13 and 4.14.* The proof of Equation 4.3, which is Equation 4.13 but without the condition that $r > 0$, is supplied by Hickey [35]. This work introduces this extra condition because Equation 4.3 [35] should only be used when $r > 0$. When $r = 0$, then the result should be 0, as stated by Equation 4.14, not $-|m(SR_j^{\mathcal{M}'})|$. □

*Proof of Equation 4.15.* This proof is an adaptation of the $\sigma_{up}(A, \otimes, B)$ proof in the MOQA book [63]. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \underline{C(S_j^{\mathcal{M}'})}$, i.e. $C$ is at the bottom of the series. Pushing a label up through $I$ consists of the following sequence of steps:

1. Push the label up through the bottom $C$.
2. Swap the label on to the above $SR_j^{\mathcal{M}'}$ if there is such a fixed po-structure, otherwise stop.
3. Push the label up through $SR_j^{\mathcal{M}'}$.
4. Go to Step 2.

For Step 2, the number of comparisons made in swapping from a maximal node of $C$ to a minimal node of $SR_j^{\mathcal{M}'}$ is $|m(SR_j^{\mathcal{M}'})|$, regardless of the labeling. There is one such swap when $r > 0$ but there is no such swap when $r = 0$. Likewise, the number of comparisons made in swapping from a maximal node of $SR_j^{\mathcal{M}'}$ to a minimal node of $SR_j^{\mathcal{M}'}$ is $|m(SR_j^{\mathcal{M}'})|$, regardless of the labeling. There are $r - 1$ such swaps when $r > 0$ but there are no such swaps when $r = 0$. Since, for Steps 1 and 3, the labelings of $C$ and of each of the $r$ $SR_j^{\mathcal{M}'}$s are all independent of one another, then the desired average for $I$ is simply the sum of the averages of these three separate steps. Hence, Equation 4.15. □

*Proof of Equations 4.16 and 4.17.* Both of these proofs closely follow that of Equation 4.15. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$, i.e. $C$ is at the top of the series. Pushing a label up through $I$ consists of the following sequence of steps:

1. If $r = 0$, then skip to Step 5.
2. Push the label up through $SR_j^{\mathcal{M}'}$.
3. Swap the label on to the above fixed po-structure. If the above fixed po-structure is $C$, then skip to Step 5.
4. Go to Step 2.
5. Push the label up through the top $C$.

For Step 3, the number of comparisons made in swapping from a maximal node of $SR_j^{\mathcal{M}'}$ to a minimal node of $SR_j^{\mathcal{M}'}$ is $|m(SR_j^{\mathcal{M}'})|$, regardless of the labeling. There are $r - 1$ such swaps when $r > 0$ but there are no such swaps when $r = 0$. The number of comparisons made in swapping from a maximal node of $SR_j^{\mathcal{M}'}$ to a minimal node of $C$ is $|m(C)|$, regardless of the labeling. There is one such swap when $r > 0$ but there is no such swap when $r = 0$. Since, for Steps 2 and 5, the labelings of each of the $r$ $SR_j^{\mathcal{M}'}$s and of $C$ are all independent of one another, then the desired average for $I$ when $r > 0$ is simply the sum of the averages of these three separate steps. Hence, Equation 4.16.

When $r = 0$, then only one step counts, Step 5. Therefore, the desired average for $I$ is simply the average of this step. Hence, Equation 4.17. □

Notice that all the new equations above and below ignore the location of the base-case fixed po-structure when the *op* in question is $||$. This is because, for any labeling of a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$, the label being pushed will be pushed through only *one* of these $r+1$ fixed po-structures. So how these $r+1$ fixed po-structures are ordered in parallel is irrelevant to any of the average-case equations considered in this work. In contrast, for any labeling of a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$, the label being pushed will be pushed through $i$ of these fixed po-structures, $1 \leq i \leq r+1$. How these $r+1$ fixed po-structures are ordered in series affects how many comparisons are required when the label is swapped on to the $2^{\text{nd}}$, $3^{\text{rd}}$, ... and $i^{\text{th}}$ of these fixed po-structures.

**Proposition 10** ($\kappa_{up}$ for a multi-base $DIPC_{\beta_{max}}$ when *op* is $||$)**.** *If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$, then:*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, ||, r) \quad = \quad r.\kappa_{up}(SR_j^{\mathcal{M}'}). \tag{4.18}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$, then:*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, C, ||, r) \quad = \quad r.\kappa_{up}(SR_j^{\mathcal{M}'}) + \kappa_{up}(C). \tag{4.19}$$

*Proof of Equation 4.18.* The proof of Equation 4.18, which is identical to Equation 4.4, is supplied by Hickey [35]. □

*Proof of Equation 4.19.* This proof is an adaptation of the $\kappa_{up}(A, ||, B)$ proof in the MOQA book [63]. Of later use in this proof is the fact that the average value of the $m^{\text{th}}$ smallest element in an $s$ element subset of the first $n$ positive integers is $m.\frac{n+1}{s+1}$. The proof for this is also given in the MOQA book [63].

The *push-up path for a labeling of fixed po-structure $A$* is the path in $A$ that starts at the node with the minimum label and moves from each node to the node directly above it with the smallest label — of course, there is the assumption here that labels are distinct. This is the path that any label being pushed all the way up through $A$ will follow for that labeling. Let $\overline{ML}(A)$ denote the average value of the largest label in $A$'s push-up path over all of

$A$'s canonically-ordered labelings. Likewise, let $\overline{ML}(SR_j^{\mathcal{M}'}, C, ||, r)$ denote this value when $A$ is formed from $r$ $SR_j^{\mathcal{M}'}$s in parallel with one $C$.

Let $t_{A,l,max}$ denote the value of the largest label in $A$'s push-up path for labeling $l$. For any canonically-ordered labeling $cl$ of $A$, any label swapped onto $A$ with rank $k > t_{A,cl,max}$ will be pushed up to the maximal node in $A$'s push-up path for $cl$. So, the number of ranks pushed up to this maximal node is $|A| + 1 - t_{A,cl,max}$. Averaging this over all of $A$'s canonically-ordered labelings gives $\kappa_{up}(A) = |A| + 1 - \overline{ML}(A)$. Likewise:

$$\kappa_{up}(SR_j^{\mathcal{M}'}, C, ||, r) = r.|SR_j^{\mathcal{M}'}| + |C| + 1 - \overline{ML}(SR_j^{\mathcal{M}'}, C, ||, r).$$

Now consider $\overline{ML}((SR_j^{\mathcal{M}'}, C, ||, r) \,|_1\, SR_j^{\mathcal{M}'})$, which is the same function as above in that it is the average value of the largest label in $SR_j^{\mathcal{M}'}$'s push-up path. The difference is the set of labelings over which it is averaged. The set is all the canonically-ordered labelings of $r$ $SR_j^{\mathcal{M}'}$s in parallel with one $C$ for which 1 is the label of some node in that individual $SR_j^{\mathcal{M}'}$ (so that the label being pushed up is pushed up through that individual $SR_j^{\mathcal{M}'}$). In other words, the set contains all the canonically-ordered labelings of each distinct label set, which will contain 1 and $|SR_j^{\mathcal{M}'}| - 1$ other labels selected from the positive integers between 2 and $r.|SR_j^{\mathcal{M}'}| + |C|$, that can be applied to the individual $SR_j^{\mathcal{M}'}$. The average of $t_{SR_j^{\mathcal{M}'},cl,max}$ when each of these distinct label sets is applied to the *same* canonically-ordered labeling $cl$ of this individual $SR_j^{\mathcal{M}'}$ is:

$$1 + \frac{r.|SR_j^{\mathcal{M}'}| + |C|}{|SR_j^{\mathcal{M}'}|}.(t_{SR_j^{\mathcal{M}'},cl,max} - 1).$$

This is derived using the above equation for the average value of the $m^{\text{th}}$ smallest element. So averaging over *all* of $SR_j^{\mathcal{M}'}$'s canonically-ordered labelings gives:

$$\overline{ML}((SR_j^{\mathcal{M}'}, C, ||, r) \,|_1\, SR_j^{\mathcal{M}'}) = 1 + \frac{r.|SR_j^{\mathcal{M}'}| + |C|}{|SR_j^{\mathcal{M}'}|}.(\overline{ML}(SR_j^{\mathcal{M}'}) - 1).$$

Therefore, $\overline{ML}(SR_j^{\mathcal{M}'}, C, ||, r)$ can now be expressed as the weighted average

of $r$ $\overline{ML}((SR_j^{\mathcal{M}'}, C, ||, r) \,|_1\, SR_j^{\mathcal{M}'})$s and $\overline{ML}((SR_j^{\mathcal{M}'}, C, ||, r) \,|_1\, C)$.

$$\begin{aligned}\overline{ML}(SR_j^{\mathcal{M}'}, C, ||, r) &= (r.|SR_j^{\mathcal{M}'}|.\overline{ML}((SR_j^{\mathcal{M}'}, C, ||, r) \,|_1\, SR_j^{\mathcal{M}'}) + \\ &\quad |C|.\overline{ML}((SR_j^{\mathcal{M}'}, C, ||, r) \,|_1\, C))/ \\ &\quad r.|SR_j^{\mathcal{M}'}| + |C|\end{aligned}$$

Substitute this into the previous equation for $\kappa_{up}(SR_j^{\mathcal{M}'}, C, ||, r)$ and the result can be obtained as follows.

$$\begin{aligned}&= r.|SR_j^{\mathcal{M}'}| + |C| + 1 - \overline{ML}(SR_j^{\mathcal{M}'}, C, ||, r) \\ &= r.|SR_j^{\mathcal{M}'}| + |C| + 1 - (r.\overline{ML}(SR_j^{\mathcal{M}'}) + \overline{ML}(C) - r) \\ &= r.(|SR_j^{\mathcal{M}'}| + 1 - \overline{ML}(SR_j^{\mathcal{M}'})) + (|C| + 1 - \overline{ML}(C)) \\ &= r.\kappa_{up}(SR_j^{\mathcal{M}'}) + \kappa_{up}(C)\end{aligned}$$

Hence, Equation 4.19. □

**Proposition 11** ($\kappa_{up}$ for a multi-base $DIPC_{\beta_{max}}$ when *op* is $\otimes$)**.** *If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r > 0$, then:*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, \otimes, r) = \kappa_{up}(SR_j^{\mathcal{M}'}). \tag{4.20}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r = 0$, i.e. $\kappa_{up}$ is applied to the empty fixed po-structure, then:*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, \otimes, 0) = 0. \tag{4.21}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$, $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $r > 0$, then:*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r) = \kappa_{up}(SR_j^{\mathcal{M}'}). \tag{4.22}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ and either $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $r = 0$ or $C \in$*

$\overline{C(S_j^{\mathcal{M}'})}$, *then:*

$$\kappa_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r) \quad = \quad \kappa_{up}(C). \tag{4.23}$$

*Proof of Equations 4.20 and 4.21.* The proof of Equation 4.5, which is Equation 4.20 but without the condition that $r > 0$, is supplied by Hickey [35]. This work introduces this extra condition because Equation 4.5 [35] should only be used when $r > 0$. When $r = 0$, then the result should be 0, as stated by Equation 4.21, not $\kappa_{up}(SR_j^{\mathcal{M}'})$. □

*Proof of Equation 4.22.* This proof is an adaptation of the $\kappa_{up}(A, \otimes, B)$ proof in the MOQA book [63]. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $r > 0$. For any canonically-ordered labeling of $I$, if a label of rank $k \leq |C| + (r-1).|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then after its push up is complete it will end up either in $I$'s $C$ or in one of $I$'s $r-1$ $SR_j^{\mathcal{M}'}$s not at the top of $I$'s series. The nodes of these fixed po-structures are mutually exclusive from $I$'s maximal nodes; the maximal nodes of $I$ must be a subset of its top $SR_j^{\mathcal{M}'}$. For any canonically-ordered labeling of $I$, if a label of rank $k > |C|+(r-1).|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then it will be pushed all the way up through $I$'s $C$, all the way up through the next $r-1$ $SR_j^{\mathcal{M}'}$s in the series and then finally swapped on to $I$'s top $SR_j^{\mathcal{M}'}$. As a result, this label now has a rank between 1 and $|SR_j^{\mathcal{M}'}|$ in the set of labels for the top $SR_j^{\mathcal{M}'}$. So, the average number of ranks between 1 and $|C| + r.|SR_j^{\mathcal{M}'}|$ that get to the top of $I$ will be precisely the average number of ranks between 1 and $|SR_j^{\mathcal{M}'}|$ that get to the top of $I$'s top $SR_j^{\mathcal{M}'}$. Hence, Equation 4.22. □

*Proof of Equation 4.23.* This proof closely follows that of Equation 4.22. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $r = 0$. So, because $r$ equals zero, $C$ is also at the top of $I$'s series. This means that the average number of ranks between 1 and $|C|$ that get to the top of $I$ will be precisely the average number of ranks between 1 and $|C|$ that get to the top of $C$. The same logic applies when $I$ denotes a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r = 0$. Hence, Equation 4.23 for both of these cases.

Now, let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r > 0$. For any canonically-ordered labeling of $I$, if a label of rank $k \leq r.|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then after its push up is complete it will end up in one of $I$'s $r$ $SR_j^{\mathcal{M}'}$s. The nodes of these fixed po-structures are mutually exclusive from $I$'s maximal nodes; the maximal nodes of $I$ must be a subset of its top $C$. For any canonically-ordered labeling of $I$, if a label of rank $k > r.|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then it will be pushed all the way up through $I$'s $r$ $SR_j^{\mathcal{M}'}$s and then swapped on to $I$'s top $C$. As a result, this label now has a rank between 1 and $|C|$ in the set of labels for the top $C$. So, the average number of ranks between 1 and $r.|SR_j^{\mathcal{M}'}| + |C|$ that get to the top of $I$ will be precisely the average number of ranks between 1 and $|C|$ that get to the top of $I$'s top $C$. Hence, Equation 4.23 for this case. □

**Proposition 12** ($\tau_{up}$ for a multi-base $DIPC_{\beta_{max}}$ when *op* is $||$)**.** *If $\tau_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ and $r > 0$, then:*

$$\tau_{up}(SR_j^{\mathcal{M}'}, ||, r) \quad = \quad \tau_{up}(SR_j^{\mathcal{M}'}). \tag{4.24}$$

*If $\tau_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ and $r = 0$, i.e. $\tau_{up}$ is applied to the empty fixed po-structure, then:*

$$\tau_{up}(SR_j^{\mathcal{M}'}, ||, 0) \quad = \quad 0. \tag{4.25}$$

*If $\tau_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$, then:*

$$\tau_{up}(SR_j^{\mathcal{M}'}, C, ||, r) \quad = \quad \frac{r.|SR_j^{\mathcal{M}'}|.\tau_{up}(SR_j^{\mathcal{M}'}) \; + \; |C|.\tau_{up}(C)}{r.|SR_j^{\mathcal{M}'}| \; + \; |C|}. \tag{4.26}$$

*Proof of Equations 4.24 and 4.25.* The proof of Equation 4.6, which is Equation 4.24 but without the condition that $r > 0$, is supplied by Hickey [35]. This work introduces this extra condition because Equation 4.6 [35] should only be used when $r > 0$. When $r = 0$, then the result should be 0, as stated

by Equation 4.25, not $\tau_{up}(SR_j^{\mathcal{M}'})$. □

*Proof of Equation 4.26.* This proof is an adaptation of the $\tau_{up}(A, ||, B)$ proof in the MOQA book [63]. For any canonically-ordered labeling $cl$ of fixed po-structure $A$, the label values in $A$'s push-up path for $cl$ will be some subsequence of the first $|A|$ positive integers. Let $n_{A,\,l}$ denote the number of nodes in $A$'s push-up path for labeling $l$. Let $t_{A,\,l,\,i}$ denote the label value of the $i^{\text{th}}$ node in $A$'s push-up path for labeling $l$, $1 \leq i \leq n_{A,\,l}$. Let $n^*_{A,\,l,\,i}$ denote the number of nodes directly above this $i^{\text{th}}$ node. Then the average number of comparisons, over all ranks from 1 to $|A|$, to push a label swapped on to $A$ up to its correct position for a canonically-ordered labeling $cl$ of $A$ is:

$$\frac{\sum_{i=1}^{n_{A,\,cl}}(|A| + 1 - t_{A,\,cl,\,i}).n^*_{A,\,cl,\,i}}{|A|}.$$

$\tau_{up}(A)$ is the average of this over all of $A$'s canonically-ordered labelings.

Now consider the average value of $t_{SR_j^{\mathcal{M}'},\,cl,\,i}$ when $r$ $SR_j^{\mathcal{M}'}$s are in parallel with one $C$ and 1 is the label of some node in one of the $SR_j^{\mathcal{M}'}$s (so that the label being pushed up is pushed up through that individual $SR_j^{\mathcal{M}'}$). In other words, the set of labelings that $t_{SR_j^{\mathcal{M}'},\,cl,\,i}$ is averaged over contains all the canonically-ordered labelings of each distinct label set, which will contain 1 and $|SR_j^{\mathcal{M}'}| - 1$ other labels selected from the positive integers between 2 and $r.|SR_j^{\mathcal{M}'}| + |C|$, that can be applied to the individual $SR_j^{\mathcal{M}'}$. Therefore, the average of $t_{SR_j^{\mathcal{M}'},\,cl,\,i}$ when each of these distinct label sets is applied to the *same* canonically-ordered labeling $cl$ of this individual $SR_j^{\mathcal{M}'}$ is:

$$1 + \frac{r.|SR_j^{\mathcal{M}'}| + |C|}{|SR_j^{\mathcal{M}'}|}.(t_{SR_j^{\mathcal{M}'},\,cl,\,i} - 1).$$

(This is derived using the equation for the average value of the $m^{\text{th}}$ smallest element in an $s$ element subset of the first $n$ positive integers, given in the proof of Equation 4.19.) So, the average number of comparisons, over all $r.|SR_j^{\mathcal{M}'}| + |C|$ ranks, to push a label swapped on to $SR_j^{\mathcal{M}'}$ up to its correct position for a canonically-ordered labeling $cl$ of $SR_j^{\mathcal{M}'}$ is:

$$\frac{\sum_{i=1}^{n_{SR_j^{\mathcal{M}'},\,cl}}\left(r.|SR_j^{\mathcal{M}'}| + |C| - \frac{r.|SR_j^{\mathcal{M}'}|+|C|}{|SR_j^{\mathcal{M}'}|}.(t_{SR_j^{\mathcal{M}'},\,cl,\,i} - 1)\right).n^*_{SR_j^{\mathcal{M}'},\,cl,\,i}}{r.|SR_j^{\mathcal{M}'}| + |C|}.$$

But, by cancelling above and below, this is simply:

$$\frac{\sum_{i=1}^{n_{SR_j^{\mathcal{M}'},cl}} \left(|SR_j^{\mathcal{M}'}| + 1 - t_{SR_j^{\mathcal{M}'},cl,i}\right).n^*_{SR_j^{\mathcal{M}'},cl,i}}{|SR_j^{\mathcal{M}'}|}.$$

Note that this is also the average number of comparisons, over all ranks from 1 to $|SR_j^{\mathcal{M}'}|$, to push a label swapped on to $SR_j^{\mathcal{M}'}$ up to its correct position for a canonically-ordered labeling $cl$ of $SR_j^{\mathcal{M}'}$. So, once again, $\tau_{up}(SR_j^{\mathcal{M}'})$ would be the average of this over all of $SR_j^{\mathcal{M}'}$'s canonically-ordered labelings. Therefore, the average number of comparisons to push a label up through $r$ $SR_j^{\mathcal{M}'}$s in parallel with one $C$ when the 1 is on one of the $SR_j^{\mathcal{M}'}$s is exactly the same as the average number of comparisons to push a label up through one $SR_j^{\mathcal{M}'}$. The same reasoning applies to the average number of comparisons to push a label up through $r$ $SR_j^{\mathcal{M}'}$s in parallel with one $C$ when the 1 is on $C$. Hence, Equation 4.26 is the average weighted by the probabilities of the 1 being on each of the $r$ $SR_j^{\mathcal{M}'}$s and $C$ respectively. □

**Proposition 13** ($\tau_{up}$ for a multi-base $DIPC_{\beta_{max}}$ when *op* is $\otimes$)**.** *If $\tau_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r > 0$, then:*

$$\begin{aligned}\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, r) \;=\; & \left(\tau_{up}(SR_j^{\mathcal{M}'}) + \frac{(r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|}{|SR_j^{\mathcal{M}'}|} + \right. \\ & \left. \sum_{i=1}^{r-1}(i.\sigma_{up}(SR_j^{\mathcal{M}'}) + i.|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'}))\right)/r.\end{aligned} \tag{4.27}$$

*If $\tau_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r = 0$, i.e. $\tau_{up}$ is applied to the empty fixed po-structure, then:*

$$\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, 0) = 0. \tag{4.28}$$

*If $\tau_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$*

*through successive iterations of* $\otimes$, $C \in \underline{C(S_j^{\mathcal{M}'})}$ *and* $r > 0$, *then:*

$$\begin{aligned}\tau_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r) &= (|C|.\tau_{up}(C) + \kappa_{up}(C).|m(SR_j^{\mathcal{M}'})| + |SR_j^{\mathcal{M}'}|. \\ &\quad \sum_{i=1}^{r}(\sigma_{up}(C) + (i-1).\sigma_{up}(SR_j^{\mathcal{M}'}) + i.|m(SR_j^{\mathcal{M}'})| \\ &\quad +\tau_{up}(SR_j^{\mathcal{M}'})) + (r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|)/ \\ &\quad r.|SR_j^{\mathcal{M}'}| + |C|. \end{aligned} \tag{4.29}$$

*If* $\tau_{up}$ *is applied to a fixed po-structure obtained from* $r$ $SR_j^{\mathcal{M}'}s$ *and one* $C$ *through successive iterations of* $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$ *and* $r > 0$, *then:*

$$\begin{aligned}\tau_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r) &= ((r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})| + |SR_j^{\mathcal{M}'}|. \\ &\quad \sum_{i=1}^{r}((i-1).\sigma_{up}(SR_j^{\mathcal{M}'}) + (i-1).|m(SR_j^{\mathcal{M}'})| + \\ &\quad \tau_{up}(SR_j^{\mathcal{M}'})) + \kappa_{up}(SR_j^{\mathcal{M}'}).|m(C)| + |C|. \\ &\quad (r.\sigma_{up}(SR_j^{\mathcal{M}'}) + (r-1).|m(SR_j^{\mathcal{M}'})| + |m(C)| + \\ &\quad \tau_{up}(C)))/r.|SR_j^{\mathcal{M}'}| + |C|. \end{aligned} \tag{4.30}$$

*If* $\tau_{up}$ *is applied to a fixed po-structure obtained from* $r$ $SR_j^{\mathcal{M}'}s$ *and one* $C$ *through successive iterations of* $\otimes$, $r = 0$ *and either* $C \in \underline{C(S_j^{\mathcal{M}'})}$ *or* $C \in \overline{C(S_j^{\mathcal{M}'})}$, *then:*

$$\tau_{up}(SR_j^{\mathcal{M}'}, C, \otimes, 0) = \tau_{up}(C). \tag{4.31}$$

Notice that Equation 4.27 differs from Equation 4.7 though they are both for the same function, which is $\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, r)$ when $r > 0$. (Once again, Equation 4.7 [35] should only be used when $r > 0$.) So first the proof for this work's version will be presented and this will help when then explaining the difference between the two.

*Proof of Equation 4.27.* This proof is an adaptation of the $\tau_{up}(A, \otimes, B)$ proof in the MOQA book [63]. First, Equation 4.27 is expanded to the following

equation, which is closer to the structure of this proof.

$$\begin{aligned}\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, r) &= (|SR_j^{\mathcal{M}'}|.\tau_{up}(SR_j^{\mathcal{M}'}) + |SR_j^{\mathcal{M}'}|. \\ &\quad \sum_{i=1}^{r-1}(i.\sigma_{up}(SR_j^{\mathcal{M}'}) + i.|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'})) + \\ &\quad (r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|)/r.|SR_j^{\mathcal{M}'}|\end{aligned}$$

Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ with $r > 0$. For any canonically-ordered labeling of $I$, if a label of rank $k \leq |SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then after its push up is complete it will end up in $I$'s bottom $SR_j^{\mathcal{M}'}$. So the number of comparisons involved will be exactly the same as if the $r-1$ $SR_j^{\mathcal{M}'}$s in series above $I$'s bottom $SR_j^{\mathcal{M}'}$ were not there. *For each of the $\kappa_{up}(SR_j^{\mathcal{M}'})$ ranks which are pushed up as far as the maximal nodes of $I$'s bottom $SR_j^{\mathcal{M}'}$, when $r > 1$ there will be an additional $|m(SR_j^{\mathcal{M}'})|$ comparisons to ensure that the label should not be pushed any further.* So, summing over all ranks $k \leq |SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons when $r > 1$ is:

$$|SR_j^{\mathcal{M}'}|.\tau_{up}(SR_j^{\mathcal{M}'}) + \kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|.$$

Otherwise, summing over all ranks $k \leq |SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons when $r > 1$ is $|SR_j^{\mathcal{M}'}|.\tau_{up}(SR_j^{\mathcal{M}'})$.

For any canonically-ordered labeling of $I$, if a label of rank $(i-1).|SR_j^{\mathcal{M}'}| < k \leq i.|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then after its push up is complete it will end up in $I$'s $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, $1 < i \leq r-1$. So, each such label must have passed through $i-1$ of $I$'s $SR_j^{\mathcal{M}'}$s, which is an average of $(i-1).\sigma_{up}(SR_j^{\mathcal{M}'})$ comparisons, and been swapped on to each of the preceding $(i-2)$ $SR_j^{\mathcal{M}'}$s and then on to the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, which requires a total of $(i-1).|m(SR_j^{\mathcal{M}'})|$ comparisons. Now the label has a rank from 1 to $|SR_j^{\mathcal{M}'}|$ in the set of labels on the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ and so the average number of comparisons in pushing it up through the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ must be $\tau_{up}(SR_j^{\mathcal{M}'})$. Finally, *for each of the $\kappa_{up}(SR_j^{\mathcal{M}'})$ ranks which are pushed up as far as the maximal nodes of the $i^{th}$ $SR_j^{\mathcal{M}'}$, there will be an additional $|m(SR_j^{\mathcal{M}'})|$ comparisons to ensure that the label should not be pushed*

*any further*. Therefore, summing over all ranks $|SR_j^{\mathcal{M}'}| < k \leq (r-1).|SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons is:

$$|SR_j^{\mathcal{M}'}|.\sum_{i=2}^{r-1}((i-1).\sigma_{up}(SR_j^{\mathcal{M}'}) + (i-1).|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'})) +$$
$$(r-2).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|.$$

For any canonically-ordered labeling of $I$, if a label of rank $k > (r-1).|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then after its push up is complete it will end up in $I$'s top $SR_j^{\mathcal{M}'}$. So, each such label must have passed through $r-1$ of $I$'s $SR_j^{\mathcal{M}'}$s, which is an average of $(r-1).\sigma_{up}(SR_j^{\mathcal{M}'})$ comparisons, and been swapped on to each of the preceding $(r-2)$ $SR_j^{\mathcal{M}'}$s and then on to the top $SR_j^{\mathcal{M}'}$, which requires a total of $(r-1).|m(SR_j^{\mathcal{M}'})|$ comparisons. Now the label has a rank from 1 to $|SR_j^{\mathcal{M}'}|$ in the set of labels on the top $SR_j^{\mathcal{M}'}$ and so the average number of comparisons in pushing it up through the top $SR_j^{\mathcal{M}'}$ must be $\tau_{up}(SR_j^{\mathcal{M}'})$. As this $SR_j^{\mathcal{M}'}$ is at the top of $I$, there is no need to ensure that the label should not be pushed any further. Therefore, summing over all ranks $k > (r-1).|SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons is:

$$|SR_j^{\mathcal{M}'}|.((r-1).\sigma_{up}(SR_j^{\mathcal{M}'}) + (r-1).|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'})).$$

Summing these three equations together and dividing by $r.|SR_j^{\mathcal{M}'}|$ to get an average over all ranks $1 \leq k \leq r.|SR_j^{\mathcal{M}'}|$ gives Equation 4.27. □

The discrepancy that exists between this work's $\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, r)$ when $r > 0$ and that of Hickey's [35] is now returned to. First, so as to bear a closer resemblance to Equation 4.27, Hickey's version is rearranged to:

$$\begin{aligned}\tau_{up}(SR_j^{\mathcal{M}'}, \otimes, r) &= (\tau_{up}(SR_j^{\mathcal{M}'}) + \\ &\quad \sum_{i=1}^{r-1}(i.\sigma_{up}(SR_j^{\mathcal{M}'}) + i.|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'})))/r.\end{aligned}$$

Missing from the numerator of this is $((r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|)/SR_j^{\mathcal{M}'}$, the motivation for which is italicised in the above proof. The impact of not

having this information in Equation 4.7 can be illustrated using Figure 4.14. I(a) shows a Hasse diagram of size three with some labeling. I(b) shows the canonically-ordered labeling that this labeling can be reduced to, which is also the only canonically-ordered labeling possible for such a Hasse diagram. Now consider the effect of replacing the minimum label in I(a)'s labeling with a label of each rank from one to three. These are the ranks considered in calculating $\tau_{up}$ for a fixed po-structure obtained from three $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ when $SR_j^{\mathcal{M}'}$ is a fixed po-structure of size one. In other words, these are the ranks considered in calculating $\tau_{up}$ for the Hasse diagram of size three in Figure 4.14. II(a) shows I(a)'s labeling just after its minimum label has been replaced by a label of rank one. II(b) shows the canonically-ordered version of this labeling. Now, only one comparison is required to establish that the new label is already in its correct position. III(a) shows I(a)'s labeling just after its minimum label has been replaced by a label of rank two. III(b) shows the canonically-ordered version of this labeling, which is still in flux. Now, pushing the new label up to its correct position requires two comparisons. IV(a) shows I(a)'s labeling just after its minimum label has been replaced by a label of rank three. IV(b) shows the canonically-ordered version of this labeling, which is still in flux. Again, pushing the new label up to its correct position requires two comparisons. Therefore, the average number of comparisons for this $\tau_{up}$ is $\frac{5}{3}$, which is also the result of Equation 4.27. On the other hand, Hickey's equation [35] returns an average of 1. Hence, any future MOQA static analysis tool should adopt the more accurate Equation 4.27 instead of Equation 4.7 [35].

*Proof of Equation 4.28.* Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ with $r = 0$. This $I$ is the empty fixed po-structure. A label cannot be swapped on to such a fixed po-structure and therefore, no comparisons take place. Hence, Equation 4.28. □

*Proof of Equation 4.29.* This proof closely follows that of Equation 4.27. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $r > 0$. For any canonically-ordered labeling of $I$, if a label of rank $k \leq |C|$ is swapped on to $I$, then after its push up is complete it will end up in $C$, which is at the bottom of $I$. So the number of comparisons involved will be exactly the same as if the $r$ $SR_j^{\mathcal{M}'}$s in series

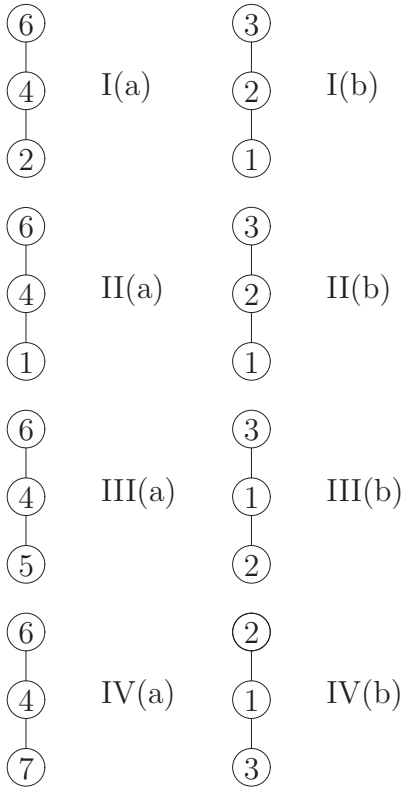



Figure 4.14: II(a) - IV(a) show the labels with ranks from 1 to 3 that can swapped on to I(a) and $*$(b) shows the canonically-ordered labeling that $*$(a) can be reduced to

above $C$ were not there. For each of the $\kappa_{up}(C)$ ranks which are pushed up as far as the maximal nodes of $C$, which is at the bottom of $I$, there will be an additional $|m(SR_j^{\mathcal{M}'})|$ comparisons to ensure that the label should not be pushed any further. So, summing over all ranks $k \leq |C|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons is:

$$|C|.\tau_{up}(C) + \kappa_{up}(C).|m(SR_j^{\mathcal{M}'})|.$$

For any canonically-ordered labeling of $I$, if a label of rank $(|C| + (i - 1).|SR_j^{\mathcal{M}'}|) < k \leq (|C| + i.|SR_j^{\mathcal{M}'}|)$ is swapped on to $I$, then after its push up is complete it will end up in $I$'s $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, $1 \leq i \leq r$. So, each such label must have passed through $I$'s $C$ and $i-1$ of $I$'s $SR_j^{\mathcal{M}'}$s, which is an average of $\sigma_{up}(C)$ and $(i-1).\sigma_{up}(SR_j^{\mathcal{M}'})$ comparisons, and been swapped on to each of the preceding $(i-1)$ $SR_j^{\mathcal{M}'}$s and then on to the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, which requires a total of $i.|m(SR_j^{\mathcal{M}'})|$ comparisons. Now the label has a rank from 1 to $|SR_j^{\mathcal{M}'}|$ in the set of labels on the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ and so the average number of comparisons in pushing it up through the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ must be $\tau_{up}(SR_j^{\mathcal{M}'})$. Finally, for each of the $\kappa_{up}(SR_j^{\mathcal{M}'})$ ranks which are pushed up as far as the maximal nodes of the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, when $i \leq (r-1)$ there will be an additional $|m(SR_j^{\mathcal{M}'})|$ comparisons to ensure that the label should not be pushed any further; when $i = r$, then the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ is at the top of $I$ so there is no need to ensure that the label should not be pushed any further. Therefore, summing over all ranks $|C| < k \leq r.|SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons is:

$$|SR_j^{\mathcal{M}'}|.\sum_{i=1}^{r}(\sigma_{up}(C) + (i-1).\sigma_{up}(SR_j^{\mathcal{M}'}) + i.|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'})) +$$
$$(r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})|.$$

Summing these two expressions together and dividing by $r.SR_j^{\mathcal{M}'} + C$ to get an average over all ranks $1 \leq k \leq (|C| + r.|SR_j^{\mathcal{M}'}|)$ gives Equation 4.29. □

*Proof of Equation 4.30.* Again, this proof closely follows that of Equation 4.27. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r > 0$. For any canonically-ordered labeling of $I$, if a label of rank $(i-1).|SR_j^{\mathcal{M}'}| < k \leq i.|SR_j^{\mathcal{M}'}|$ is

swapped on to $I$, then after its push up is complete it will end up in $I$'s $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, $1 \leq i \leq r$. So, each such label must have passed through $i-1$ of $I$'s $SR_j^{\mathcal{M}'}$s, which is an average of $(i-1).\sigma_{up}(SR_j^{\mathcal{M}'})$ comparisons, and been swapped on to each of the preceding $(i-2)$ $SR_j^{\mathcal{M}'}$s and then on to the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, which requires a total of $(i-1).|m(SR_j^{\mathcal{M}'})|$ comparisons. Now the label has a rank from 1 to $|SR_j^{\mathcal{M}'}|$ in the set of labels on the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ and so the average number of comparisons in pushing it up through the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$ must be $\tau_{up}(SR_j^{\mathcal{M}'})$. Finally, for each of the $\kappa_{up}(SR_j^{\mathcal{M}'})$ ranks which are pushed up as far as the maximal nodes of the $i^{\text{th}}$ $SR_j^{\mathcal{M}'}$, when $i \leq (r-1)$ there will be an additional $|m(SR_j^{\mathcal{M}'})|$ comparisons to ensure that the label should not be pushed any further; when $i = r$ there will be an additional $|m(C)|$ comparisons to ensure that the label should not be pushed any further. Therefore, summing over all ranks $k \leq r.|SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons is:

$$|SR_j^{\mathcal{M}'}|.\sum_{i=1}^{r}((i-1).\sigma_{up}(SR_j^{\mathcal{M}'}) + (i-1).|m(SR_j^{\mathcal{M}'})| + \tau_{up}(SR_j^{\mathcal{M}'})) +$$
$$(r-1).\kappa_{up}(SR_j^{\mathcal{M}'}).|m(SR_j^{\mathcal{M}'})| + \kappa_{up}(SR_j^{\mathcal{M}'}).|m(C)|.$$

For any canonically-ordered labeling of $I$, if a label of rank $k > r.|SR_j^{\mathcal{M}'}|$ is swapped on to $I$, then after its push up is complete it will end up in $C$, which is at the top of $I$. So, each such label must have passed through $r$ of $I$'s $SR_j^{\mathcal{M}'}$s, which is an average of $r.\sigma_{up}(SR_j^{\mathcal{M}'})$ comparisons, and been swapped on to each of the preceding $(r-1)$ $SR_j^{\mathcal{M}'}$s and then on to the top $C$, which requires a total of $(r-1).|m(SR_j^{\mathcal{M}'})| + |m(C)|$ comparisons. Now the label has a rank from 1 to $|C|$ in the set of labels on the top $C$ and so the average number of comparisons in pushing it up through the top $C$ must be $\tau_{up}(C)$. As this $C$ is at the top of $I$, there is no need to ensure that the label should not be pushed any further. Therefore, summing over all ranks $k > r.|SR_j^{\mathcal{M}'}|$ and averaging over all of $I$'s canonically-ordered labelings, the total number of comparisons is:

$$|C|.(r.\sigma_{up}(SR_j^{\mathcal{M}'}) + (r-1).|m(SR_j^{\mathcal{M}'})| + |m(C)| + \tau_{up}(C)).$$

Summing these two expressions together and dividing by $r.SR_j^{\mathcal{M}'} + C$ to get an average over all ranks $1 \leq k \leq (|C| + r.|SR_j^{\mathcal{M}'}|)$ gives Equation 4.30. $\square$

*Proof of Equation 4.31.* Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r = 0$. So, because $r$ equals zero, $I$ is obtained from just one $C$. This means that the average number of comparisons to push a label swapped on to the bottom of $I$ up to its correct position is exactly the same as the average number of comparisons to push a label swapped on to the bottom of $C$ up to its correct position. The same logic applies when $I$ denotes a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r = 0$. Hence, Equation 4.31. □

**Proposition 14** ($\Delta_{up}$ for deleting the label with rank $k$, i.e. the $k^{\text{th}}$ smallest label, from a multi-base $DIPC_{\beta_{max}}$ when *op* is $||$)**.** *If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ and $r > 0$, then:*

$$\Delta_{up}(SR_j^{\mathcal{M}'}, ||, r, k) = \frac{r.\sum_{i=1}^{|SR_j^{\mathcal{M}'}|}\binom{k-1}{i-1}.\binom{r.|SR_j^{\mathcal{M}'}|-k}{|SR_j^{\mathcal{M}'}|-i}.\Delta_{up}(SR_j^{\mathcal{M}'}, i)}{\prod_{x=1}^{r}\binom{x.|SR_j^{\mathcal{M}'}|}{|SR_j^{\mathcal{M}'}|}}. \tag{4.32}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ and $r = 0$, i.e. $\Delta_{up}$ is applied to the empty fixed po-structure, then:*

$$\Delta_{up}(SR_j^{\mathcal{M}'}, ||, 0, k) = 0. \tag{4.33}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$*

*through successive iterations of* $||$ *and* $r > 0$*, then:*

$$\begin{aligned}\Delta_{up}(SR_j^{\mathcal{M}'}, C, ||, r, k) \;=\; & \Big(r.\Big(\sum_{i=1}^{|SR_j^{\mathcal{M}'}|}\binom{k-1}{i-1}.\binom{r.|SR_j^{\mathcal{M}'}|+|C|-k}{|SR_j^{\mathcal{M}'}|-i}.\\ & \Delta_{up}(SR_j^{\mathcal{M}'}, i)\Big) + \\ & \sum_{i=1}^{|C|}\binom{k-1}{i-1}.\binom{r.|SR_j^{\mathcal{M}'}|+|C|-k}{|C|-i}.\Delta_{up}(C,i)\Big)/\\ & \prod_{x=1}^{r}\binom{x.|SR_j^{\mathcal{M}'}|+|C|}{|SR_j^{\mathcal{M}'}|}.\end{aligned} \tag{4.34}$$

*If* $\Delta_{up}$ *is applied to a fixed po-structure obtained from* $r$ $SR_j^{\mathcal{M}'}$*s and one* $C$ *through successive iterations of* $||$ *and* $r = 0$*, then:*

$$\Delta_{up}(SR_j^{\mathcal{M}'}, C, ||, 0, k) \;=\; \Delta_{up}(C, k). \tag{4.35}$$

Notice that Equation 4.32 differs from Equation 4.8 though they are both for the same function, which is $\Delta_{up}(SR_j^{\mathcal{M}'}, ||, r, k)$ when $r > 0$. (Once again, Equation 4.8 [35] should only be used when $r > 0$.) So first the proof for this work's version will be presented and this will help when then explaining the difference between the two.

*Proof of Equation 4.32.* This proof is an adaptation of the $\Delta_{up}(A, ||, B, k)$ proof in the MOQA book [63]. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ with $r > 0$. For each $i$ between 1 and $|SR_j^{\mathcal{M}'}|$, consider the number of ways to split the set of labels on $I$ such that the $k^{\text{th}}$ smallest label in the entire set becomes the $i^{\text{th}}$ smallest on *one* of the $r$ $SR_j^{\mathcal{M}'}$s that $I$ is obtained from. For such a split to take place, $i-1$ labels are chosen from the set of $k-1$ smallest labels to put on $SR_j^{\mathcal{M}'}$ and $|SR_j^{\mathcal{M}'}|-i$ labels are chosen from the set of $|I|-k$ largest labels to put on $SR_j^{\mathcal{M}'}$. So, for each $i$ between 1 and $|SR_j^{\mathcal{M}'}|$, select $i-1$ labels smaller than rank $k$, one label of rank $k$ and $|SR_j^{\mathcal{M}'}|-i$ labels greater than rank $k$ to put on $SR_j^{\mathcal{M}'}$, $(i-1)+1+(|SR_j^{\mathcal{M}'}|-i) = |SR_j^{\mathcal{M}'}|$. Since the two choices are independent, there are exactly $\binom{k-1}{i-1}.\binom{|I|-k}{|SR_j^{\mathcal{M}'}|-i}$ different splits of the set of labels in which

the $k^{\text{th}}$ smallest label in the full set becomes the $i^{\text{th}}$ smallest label in the set of labels on $SR_j^{\mathcal{M}'}$. The average number of comparisons for deleting the $i^{\text{th}}$ smallest label on $SR_j^{\mathcal{M}'}$ is simply $\Delta_{up}(SR_j^{\mathcal{M}'}, i)$. Summing over all $i$ for each of the $r$ $SR_j^{\mathcal{M}'}$s and taking an average over all the different possible splits of the set of labels on $I$ gives Equation 4.32. □

So why is Equation 4.32 different to that given by Hickey [35]? The first of the two discrepancies is between the second binomial coefficient in both numerators.

$$\text{Here}: \binom{r.|SR_j^{\mathcal{M}'}|-k}{|SR_j^{\mathcal{M}'}|-\boldsymbol{i}} \qquad [35]: \binom{r.|SR_j^{\mathcal{M}'}|-k}{|SR_j^{\mathcal{M}'}|-\mathbf{1}}$$

As explained in the above proof, this binomial coefficient selects labels greater than rank $k$ to put on $SR_j^{\mathcal{M}'}$. The number selected completes the set of labels on $SR_j^{\mathcal{M}'}$ when $i$ labels have already been selected for the set of labels on $SR_j^{\mathcal{M}'}$. So this binomial coefficient chooses $|SR_j^{\mathcal{M}'}| - i$ labels as $i + (|SR_j^{\mathcal{M}'}| - i) = |SR_j^{\mathcal{M}'}|$. However, the binomial coefficient in Hickey's work [35] chooses $|SR_j^{\mathcal{M}'}| - 1$ labels though $i + (|SR_j^{\mathcal{M}'}| - 1) = |SR_j^{\mathcal{M}'}|$ only holds when $i = 1$. Note that this may have been a typographical error.

The other discrepancy between the two equations can be found in the denominator. The denominator used in Hickey's work [35] does not cover, as it should, all of the different possible splits of the set of labels on the $r$ $SR_j^{\mathcal{M}'}$s in parallel. A simple example demonstrates this. Let $|SR_j^{\mathcal{M}'}| = 2$ and let $r = 3$. So there are six elements in the set of labels on the three $SR_j^{\mathcal{M}'}$s that are in parallel. The total number of different possible splits of this set is $\binom{6}{2}.\binom{4}{2}.\binom{2}{1}$. Though all of this information is captured by Equation 4.32's denominator, Hickey's denominator [35] just captures the first part, i.e. $\binom{6}{2}$. Therefore, any future MOQA static analysis tool should adopt the more accurate Equation 4.32 instead of Equation 4.8 [35].

*Proof of Equation 4.33.* Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $||$ with $r = 0$. This $I$ is the empty fixed po-structure. A label cannot be swapped on to such a fixed po-structure and therefore, no comparisons take place. Hence, Equation 4.33. □

*Proof of Equation 4.34.* This proof closely follows that of Equation 4.32. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$ with $r > 0$. The proof of Equation 4.32 considered

the number of ways to split the set of labels on $I$ such that the $k^{\text{th}}$ smallest label in the entire set becomes the $i^{\text{th}}$ smallest on *one* of these $r$ $SR_j^{\mathcal{M}'}$s. Now the number of ways in which to split the set of labels on $I$ such that the $k^{\text{th}}$ smallest label in the entire set becomes the $i^{\text{th}}$ smallest on $C$ also needs to be considered. Using the logic in the proof of Equation 4.32, there are exactly $\binom{k-1}{i-1}.\binom{|I|-k}{|C|-i}$ different splits of the set of labels in which the $k^{\text{th}}$ smallest label in the full set becomes the $i^{\text{th}}$ smallest label in the set of labels on $C$. Again, the average number of comparisons for deleting the $i^{\text{th}}$ smallest label on $C$ is simply $\Delta_{up}(C, i)$. Summing over all $i$ for each of the $r$ $SR_j^{\mathcal{M}'}$s and the one $C$ and taking an average over all the different possible splits of the set of labels on $I$ gives Equation 4.34. □

*Proof of Equation 4.35.* Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $||$ with $r = 0$. So, because $r$ equals zero, $I$ is obtained from just one $C$. This means there is only one set of labels possible on $C$, i.e. the set of labels on $I$. Therefore, the $k^{\text{th}}$ smallest label in the set of labels on $I$ will always be the $k^{\text{th}}$ smallest label in the set of labels on $C$. So the average number of comparisons for deleting the $k^{\text{th}}$ smallest label on $I$ is simply $\Delta_{up}(C, k)$ and averaging this over the only possible split of the set of labels on $I$ gives Equation 4.35. □

**Proposition 15** ($\Delta_{up}$ for deleting the label with rank $k$, i.e. the $k^{\text{th}}$ smallest label, from a multi-base $DIPC_{\beta_{max}}$ when *op* is $\otimes$)**.** *If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r > 0$, then:*

$$\begin{aligned}\Delta_{up}(SR_j^{\mathcal{M}'}, \otimes, r, k) \;=\; & \Delta_{up}(SR_j^{\mathcal{M}'}, k \bmod |SR_j^{\mathcal{M}'}|) + \\ & \left(r - \left\lceil \frac{k}{|SR_j^{\mathcal{M}'}|} \right\rceil\right).(|m(SR_j^{\mathcal{M}'})| - 1 + \Delta_{up}(SR_j^{\mathcal{M}'}, 1)).\end{aligned} \tag{4.36}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ and $r = 0$, i.e. $\Delta_{up}$ is applied to the empty structure, then:*

$$\Delta_{up}(SR_j^{\mathcal{M}'}, \otimes, 0, k) \;=\; 0. \tag{4.37}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}s$ and one $C$ through successive iterations of $\otimes$, $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $k \leq |C|$, then:*

$$\begin{aligned}\Delta_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r, k) &= \Delta_{up}(C, k) + \\ &\quad r.(|m(SR_j^{\mathcal{M}'})| - 1 + \Delta_{up}(SR_j^{\mathcal{M}'}, 1)). \quad (4.38)\end{aligned}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}s$ and one $C$ through successive iterations of $\otimes$, $C \in \underline{C(S_j^{\mathcal{M}'})}$, $k > |C|$ and $r > 0$, then:*

$$\begin{aligned}\Delta_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r, k) &= \Delta_{up}(SR_j^{\mathcal{M}'}, (k - |C|) \bmod |SR_j^{\mathcal{M}'}|) + \\ &\quad \left(r - \left\lceil \frac{k - |C|}{|SR_j^{\mathcal{M}'}|} \right\rceil\right).(|m(SR_j^{\mathcal{M}'})| - 1 + \\ &\quad \Delta_{up}(SR_j^{\mathcal{M}'}, 1)). \quad (4.39)\end{aligned}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}s$ and one $C$ through successive iterations of $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$, $k \leq r.|SR_j^{\mathcal{M}'}|$ and $r > 0$, then:*

$$\begin{aligned}\Delta_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r, k) &= \Delta_{up}(SR_j^{\mathcal{M}'}, k \bmod |SR_j^{\mathcal{M}'}|) + \\ &\quad \left(r - \left\lceil \frac{k}{|SR_j^{\mathcal{M}'}|} \right\rceil\right).(|m(SR_j^{\mathcal{M}'})| - 1 + \\ &\quad \Delta_{up}(SR_j^{\mathcal{M}'}, 1)) + |m(C)| - 1 + \\ &\quad \Delta_{up}(C, 1). \quad (4.40)\end{aligned}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}s$ and one $C$ through successive iterations of $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$ and either $k \leq r.|SR_j^{\mathcal{M}'}|$ and $r = 0$ or $k > r.|SR_j^{\mathcal{M}'}|$, then:*

$$\Delta_{up}(SR_j^{\mathcal{M}'}, C, \otimes, r, k) = \Delta_{up}(C, k \bmod |SR_j^{\mathcal{M}'}|). \quad (4.41)$$

Once again, notice that Equation 4.36 differs from Equation 4.9 though they are both for the same function, which is $\Delta_{up}(SR_j^{\mathcal{M}'}, \otimes, r, k)$ when $r > 0$. (Once again, Equation 4.9 [35] should only be used when $r > 0$.) So first the proof for this work's version will be presented and this will help when then explaining the difference between the two.

*Proof of Equation 4.36.* This proof is an adaptation of the $\Delta_{up}(A, \otimes, B, k)$

proof in the MOQA book [63]. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ with $r > 0$. It is known that all of the labels on $I$'s $w^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom are greater than all of the labels on $I$'s $(w-1)^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom, $2 \leq w \leq r$. Therefore, a label has rank $k$ in the set of labels on $I$'s $w^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom if and only if it has rank $k + (w-1).|SR_j^{\mathcal{M}'}|$ in the set of labels on all $w$ of $I$'s bottom $SR_j^{\mathcal{M}'}$s, which is the same as it having rank $k + (w-1).|SR_j^{\mathcal{M}'}|$ in the set of labels on $I$.

Let $J$ denote $I$'s $\lceil k/|SR_j^{\mathcal{M}'}|\rceil^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom. The label of rank $k$ in the set of labels on $I$ is in the set of labels on $J$ and has rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ in this set of labels. (Note that $(k \bmod |SR_j^{\mathcal{M}'}|)$ equals zero when the label of rank $k$ in the set of labels on $I$ has rank $|SR_j^{\mathcal{M}'}|$ in the set of labels on $J$. This is acceptable however because a label with rank $|SR_j^{\mathcal{M}'}|$ in the set of labels on $SR_j^{\mathcal{M}'}$ is already on one of $SR_j^{\mathcal{M}'}$'s maximal nodes so the cost of deleting the label up through $SR_j^{\mathcal{M}'}$ is zero, which is also the cost of deleting a label with rank zero up through $SR_j^{\mathcal{M}'}$.) So deleting a label with rank $k$ up through $I$ consists of the following sequence of steps:

1. Delete the label of rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ from $J$ by pushing it up through $J$.

2. Swap the label on to the above $SR_j^{\mathcal{M}'}$ if there is such a fixed po-structure, otherwise go to Step 5.

3. Delete the label, which is now of the smallest rank in the set of labels on the current $SR_j^{\mathcal{M}'}$, from the current $SR_j^{\mathcal{M}'}$ by pushing it up through $SR_j^{\mathcal{M}'}$.

4. Go to Step 2.

5. Delete the maximal node that the label has been pushed up to in the current $SR_j^{\mathcal{M}'}$.

For Step 1, as the average number of comparisons made in deleting a label up through $J$ is independent of the labels on the surrounding $(r-1)$ $SR_j^{\mathcal{M}'}$s, the average number of comparisons made in deleting the label of rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ from $J$ by pushing it up through $J$ is $\Delta_{up}(SR_j^{\mathcal{M}'}, k \bmod |SR_j^{\mathcal{M}'}|)$. For Step 2, when the label being deleted has been pushed up through $SR_j^{\mathcal{M}'}$, then

the number of comparisons made in swapping that label on to the above $SR_j^{\mathcal{M}'}$, if there is such a fixed po-structure, is $|m(SR_j^{\mathcal{M}'})| - 1$. (It is one less than the corresponding number of comparisons for a Push-Up because in this context it is not necessary to compare the smallest label on the above $SR_j^{\mathcal{M}'}$ to the label being deleted, since it will always be larger.) Finally, for Step 3, as the average number of comparisons made in deleting a label up through $SR_j^{\mathcal{M}'}$ is independent of the labels on the surrounding $(r-1)$ $SR_j^{\mathcal{M}'}$s, the average number of comparisons made in deleting the smallest label from $SR_j^{\mathcal{M}'}$ by pushing it up through $SR_j^{\mathcal{M}'}$ is $\Delta_{up}(SR_j^{\mathcal{M}'}, 1)$. Steps 2 and 3 are repeated $r - \lceil k/|SR_j^{\mathcal{M}'}| \rceil$ times as this is the number of $SR_j^{\mathcal{M}'}$s above $J$ in $I$. Note that Step 5 does not involve any comparisons. Hence, Equation 4.36. □

The difference between Equation 4.36 and Equation 4.9 is now returned to. According to Equation 4.9 [35], once the label being deleted has been swapped on to the above $SR_j^{\mathcal{M}'}$ it has the largest rank in the set of labels on that $SR_j^{\mathcal{M}'}$. (Leaving aside the use of $\Delta_{up}$ throughout Equation 4.9 [35], the fact that the number of comparisons to swap between $SR_j^{\mathcal{M}'}$s is $|m(SR_j^{\mathcal{M}'})| - 1$ shows the intent to swap the label being deleted upwards.) However, as stated in the above proof, all of the labels on any $SR_j^{\mathcal{M}'}$ are greater than all of the labels on any $SR_j^{\mathcal{M}'}$ that has been producted below it. Therefore, the label being deleted upwards *must* have the smallest rank, not the largest, in the set of the labels on the $SR_j^{\mathcal{M}'}$ that it has just been swapped on to. Therefore, any future MOQA static analysis tool should adopt the more accurate Equation 4.36 instead of Equation 4.9 [35].

*Proof of Equation 4.37.* Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s through successive iterations of $\otimes$ with $r = 0$. This $I$ is the empty fixed po-structure. A label cannot be swapped on to such a fixed po-structure and therefore, no comparisons take place. Hence, Equation 4.37. □

*Proof of Equation 4.38.* This proof applies concepts explained in the proof of Equation 4.36. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \underline{C(S_j^{\mathcal{M}'})}$. If a label of rank $k \leq |C|$ is to be deleted up through $I$, then this label has rank $k$ in the set of labels on $C$ because $C$ is at the bottom of $I$. Therefore, the average number of comparisons made in deleting the label of rank $k$ from $C$ by pushing it up through $C$ is $\Delta_{up}(C, k)$. Steps 2 to 5 in the proof of Equation 4.36 finish

deleting the label up through the remainder of $I$. Steps 2 and 3 are repeated $r$ times as this is the number of $SR_j^{\mathcal{M}'}$s above $C$ in $I$. Hence, Equation 4.38. □

*Proof of Equation 4.39.* This proof applies concepts explained in the proof of Equation 4.36. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r > 0$. Let $J$ denote $I$'s $\lceil k-|C|/|SR_j^{\mathcal{M}'}|\rceil^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom. If a label of rank $k > |C|$ is to be deleted up through $I$, then this label is in the set of labels on $J$ and has rank $((k-|C|) \bmod |SR_j^{\mathcal{M}'}|)$ in this set of labels. (Recall the assumption that rank $k$ is some value between zero and the size of $I$. So, if rank $k$ is greater than the size of $C$, then $r$ must be greater than zero for this assumption to hold. Hence, deleting the label of rank $k > |C|$ up through an $I$ whose $C$ is at the bottom is only considered when $r > 0$.) Therefore, the average number of comparisons made in deleting the label of rank $((k-|C|) \bmod |SR_j^{\mathcal{M}'}|)$ from $J$ by pushing it up through $J$ is $\Delta_{up}(SR_j^{\mathcal{M}'}, (k-|C|) \bmod |SR_j^{\mathcal{M}'}|)$. Steps 2 to 5 in the proof of Equation 4.36 finish deleting the label up through the remainder of $I$. Steps 2 and 3 are repeated $r - \lceil k-|C|/|SR_j^{\mathcal{M}'}|\rceil$ times as this is the number of $SR_j^{\mathcal{M}'}$s above $J$ in $I$. Hence, Equation 4.39. □

*Proof of Equation 4.40.* This proof applies concepts explained in the proof of Equation 4.36. Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r > 0$. Let $J$ denote $I$'s $\lceil k/|SR_j^{\mathcal{M}'}|\rceil^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom. If a label of rank $k \leq r.|SR_j^{\mathcal{M}'}|$ is to be deleted up through $I$, then this label is in the set of labels on $J$ and has rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ in this set of labels. Therefore, the average number of comparisons made in deleting the label of rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ from $J$ by pushing it up through $J$ is $\Delta_{up}(SR_j^{\mathcal{M}'}, k \bmod |SR_j^{\mathcal{M}'}|)$. Steps 2 to 4 in the proof of Equation 4.36 continue deleting the label by pushing it up through the $SR_j^{\mathcal{M}'}$s that are above $J$ in $I$. Steps 2 and 3 are repeated $r - \lceil k/|SR_j^{\mathcal{M}'}|\rceil$ times as this is the number of $SR_j^{\mathcal{M}'}$s above $J$ in $I$. Finally, after the label being deleted has been pushed up through $I$'s $r^{\text{th}}$ $SR_j^{\mathcal{M}'}$ from the bottom, it is then swapped on to the above $C$, pushed up through it and the maximal node in $C$ that it arrives at is deleted. The average number of comparisons made in swapping the label on to the above $C$ is $|m(C)| - 1$ and the average number of comparisons made in deleting the label from $C$ by pushing it up through $C$ is $\Delta_{up}(C, 1)$. Hence, Equation 4.40. □

*Proof of Equation 4.41.* Let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $r = 0$. This $I$ is the fixed po-structure $C$ and if a label of rank $k \leq r.|SR_j^{\mathcal{M}'}|$ is to be deleted up through this $I$, then the rank $k$ is zero. Therefore, Equation 4.41 should return zero because this is the cost of deleting a label with rank zero up through any fixed po-structure. As $(k \bmod |SR_j^{\mathcal{M}'}|) = (0 \bmod |SR_j^{\mathcal{M}'}|) = 0$, Equation 4.41 becomes $\Delta_{up}(C, 0)$ and so will return zero.

Now let $I$ denote a fixed po-structure obtained from $r$ $SR_j^{\mathcal{M}'}$s and one $C$ through successive iterations of $\otimes$ with $C \in \overline{C(S_j^{\mathcal{M}'})}$. If a label of rank $k > r.|SR_j^{\mathcal{M}'}|$ is to be deleted up through $I$, then this label is in the set of labels on $C$ and has rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ in this set of labels. The average number of comparisons made in deleting the label of rank $(k \bmod |SR_j^{\mathcal{M}'}|)$ from $C$ by pushing it up through $C$ is $\Delta_{up}(C, k \bmod |SR_j^{\mathcal{M}'}|)$. As $C$ is at the top of $I$, all that remains is to delete the maximal node in $C$ that the label being deleted has been pushed up to. This does not involve any comparisons. Hence, Equation 4.41. □

There would be similar $\sigma_{down}$, $\kappa_{down}$, $\tau_{down}$ and $\Delta_{down}$ equations for a multi-base $DIPC_{\beta_{max}}$.

So the multiple possible solutions for these new equations can lead to multiple average-case results for the algorithm under static analysis. The code below demonstrates this. For the program $p'$ that this code is a snippet from, let c denote composite variable $c$. Let $\mathcal{M}'$ denote $c$'s MOQA$'$ random bag when the for loop below is reached. Let there be one MOQA$'$ random structure in this $\mathcal{M}'$ and let its $P_{\beta_{max}}$ be the following inductive po-class $Y$ whose set is infinite.

$$
\begin{aligned}
< Y > &::= () \\
< Y > &::= < number > \\
< Y > &::= < Y > \otimes ((< number > || < number >) \otimes < number >)
\end{aligned}
$$

builtSoFar in the code below references $c$ for the initial and subsequent loop iterations and iter is an iterator whose underlying collection is $L_1^{\mathcal{M}'}$.

```
for (int i = 0; i < n; i++)
  builtSoFar = c.product(builtSoFar, c.product(
      iter.next(), iter.next(), MARKER, iter.next()));
```

After each loop iteration, there is one MOQA′ random structure in $c$'s MOQA′ random bag for that moment and the $P_\beta$ of this MOQA′ random structure is $Y_{\beta_{max}}$. It is still $Y_{\beta_{max}}$ because the outer MOQA product function conforms to $Y$'s structural definition. Now let $A_{\beta_{max}}$ denote the fixed po-structure of size two whose two nodes are in parallel, let $B_{\beta_{max}}$ denote the fixed po-structure of size one and let $C_{\beta_{max}}$ denote the v-shaped fixed po-structure of size three. So the average-case cost of the above for loop is:

$$n.\overline{T}_{prod}[A_{\beta_{max}} \otimes B_{\beta_{max}}] + n.\overline{T}_{prod}[Y_{\beta_{max}} \otimes C_{\beta_{max}}].$$

$\overline{T}_{prod}[A_{\beta_{max}} \otimes B_{\beta_{max}}]$ can be solved by using the standard binary equations for $\tau_{up}(A_{\beta_{max}})$ and $\tau_{down}(B_{\beta_{max}})$; see Section 2.2 for the average-case formula for the MOQA product function and these standard binary equations. Likewise, $\overline{T}_{prod}[Y_{\beta_{max}} \otimes C_{\beta_{max}}]$ can be solved with the standard binary equation for $\tau_{down}(C_{\beta_{max}})$ but $\tau_{up}(Y_{\beta_{max}})$ must be selected from one of the appropriate multi-base $DIPC_{\beta_{max}}$ equations. The appropriate equations are Equations 4.27, 4.28, 4.30 and 4.31. (Equation 4.29 does not apply here as it is clear from the structural definition of $Y$ that all its base-case fixed po-structures will be located at the top of the relevant fixed po-structures in $Y$'s set.) Therefore, because of these four appropriate equations, there are four possible solutions to $\overline{T}_{prod}[Y_{\beta_{max}} \otimes C_{\beta_{max}}]$ and so, there are a total of four possible average-case costs for the above for loop. (If $Y$'s set of base-case fixed po-structures had a cardinality of two instead of one, then there would be a total of six possible average-case costs for this loop because both Equation 4.30 and Equation 4.31 would have two possible solutions, one for each base-case fixed po-structure. And so on as the cardinality of $Y$'s set of base-case fixed po-structures increases.)

This example confirms an important fact, which is that these new multi-base $DIPC_{\beta_{max}}$ equations may result in there being more than one average-case solution. Similarly, when $P_{\beta_{max}}$ of $S_j^{\mathcal{M}'}$ denoted an inductive po-class whose set is finite, it was established that multiple average-case solutions could result from applying a MOQA function to such a $P_{\beta_{max}}$. Moreover, Hickey's

empty-base $DIPC_{\beta_{max}}$ equations [35] may also have this effect as it has been identified here, on pages 125, 132 and 136, that some of them do not cover the case when $r$ equals zero, thereby requiring this work to supply additional equations for such cases. (In the MOQA book [63], binary $\sigma$, $\kappa$, $\tau$ and $\Delta$ each have additional "base-case" equations also; see page 23 to page 25. These equations are applied when the size of the fixed po-structure is zero or one. However, there is always exactly one average-case solution because the binary equations for $\sigma$, $\kappa$, $\tau$ and $\Delta$ are applied to *finite* partial orders and therefore, are always resolved to some rational number.) Yet, it is necessary that such empty-base and multi-base $DIPC_{\beta_{max}}$ equations exist because their presence will enable a MOQA static analysis tool to analyse data structures whose sizes are not fixed, while still adhering to the mathematical approach advocated by Schellekens [63]. A MOQA static analysis tool would not be of much practical use if it only examined fixed po-structures. Nonetheless, any system that statically calculates any algorithmic behaviour should aim to provide its user with a single solution. So the MOQA static analysis tool will often fall short in this regard for both the current and the new $DIPC_{\beta_{max}}$ equations. This is a serious failing because it is not the desired resolution to the problem that the MOQA book [63] proclaims to solve and so, it is not a flaw to be easily dismissed. However, it could be argued that Hickey's equations [35] and the new equations presented here are still an advance over the MOQA static analysis tool furnishing no average-case solution at all and perhaps the multiple outputs will still supply some useful sense of algorithmic behaviour to the end-user.

In conclusion, existing equations have been corrected and new ones developed so that the MOQA static analysis tool can provide average-case results for algorithms that involve $DIPC_{\beta_{max}}$s whose sets are infinite[6]. These equations reveal that Schellekens's mathematical approach is weak when it comes to providing a single average-case solution for data structures whose sizes are not fixed, i.e. it is weak for the general case. Such obscurity negatively impacts on how useful the MOQA approach is, an issue that is returned to later on in this work. For now, the next point of order to consider is $NDIPC_{\beta}$s whose sets are infinite.

[6]Note that these equations will also apply for the min-heap label ordering. Recall that any other label ordering is not considered by MOQA.

#### 4.5.2.4 The Average-case Cost of a MOQA Function Applied to an Infinite $NDIPC_\beta$

Initially, some specific $IPC$ types shall be defined.

**Definition 52** (Singular self-identity $IPC$)**.** *An $IPC$ is a singular self-identity $IPC$ if it is a self-identity $IPC$ and, for each non base-case production rule $r$ in its structural definition, there is exactly one non-terminal that is the head of $r$ in the body of $r$.*

**Definition 53** (Empty-base singular $IPC$)**.** *An $IPC$ is an empty-base singular $IPC$ if its structural definition is tightly defined, it is a singular self-identity $IPC$ and the tightly defined structural definition has two or more non base-case production rules and one base-case production rule whose body is empty.*

**Definition 54** (Multi-base singular $IPC$)**.** *An $IPC$ is a multi-base singular $IPC$ if its structural definition is tightly defined, it is a singular self-identity $IPC$ and the tightly defined structural definition has two or more non base-case production rules and either one base-case production rule whose body is not empty or two or more base-case production rules, one of which may have an empty body.*

**Definition 55** (Singular $IPC$)**.** *A singular $IPC$ is either an empty-base or a multi-base singular $IPC$.*

As an aside, it may be of interest to the reader to note that the tightly defined structural definition of a $DIPC$ also meets the requirements of a singular self-identity $IPC$. It was not necessary to explicitly state this fact earlier as it was already covered by a $DIPC$'s determinism. If the structural definition of an $IPC$ has one non base-case production rule and the body of this production rule has two or more head non-terminals, then there would be at least one size for which there is more than one fixed po-structure of that size in the $IPC$'s set. Such an $IPC$ would therefore be non-deterministic. Hence, a $DIPC$'s tightly defined structural definition has exactly one head non-terminal in the body of its one non base-case production rule.

Returning to topic, a singular $NDIPC$ can be defined as follows. As in the previous section, let *op* denote either $\otimes$ or $||$. Now let $FPS_x$ denote a finite multiset of $s$ fixed po-structures, $\{FPS_{x_1}, FPS_{x_2}, \ldots, FPS_{x_s}\}$, with

$s \geq 2$. $FPS_y$ denotes a fixed po-structure. For a specific $FPS_x$, each fixed po-structure in a singular $NDIPC$'s set is obtained from zero or more $FPS_{x_1}$s, zero or more $FPS_{x_2}$s, ..., zero or more $FPS_{x_s}$s and up to one $FPS_y$ through successive iterations of *op*. Each $FPS_{x_i}$ is a fixed po-structure construed from one of the tightly defined singular $NDIPC$'s non base-case production rules, $1 \leq i \leq s$. So $s$ is the number of non base-case production rules in a singular $NDIPC$'s tightly defined structural definition. Hence, $s$ is always greater than or equal to two. $FPS_y$ is still a fixed po-structure construed from any base-case production rule of the tightly defined singular $NDIPC$ whose body is not empty. $FPS_{x_i}$ is called one of a singular $NDIPC$'s *multi-repeat fixed po-structures* as every fixed po-structure in the singular $NDIPC$'s set includes zero or more $FPS_{x_i}$s. $FPS_y$ is, as before, called one of a singular $NDIPC$'s base-case fixed po-structures.

**Notation 90.** *Let $MR(S_j^{\mathcal{M}'}) = \{M_1, M_2, \ldots, M_s\}$ denote the multiset of all $S_j^{\mathcal{M}'}$'s multi-repeat fixed po-structures when $P_\beta$ of $S_j^{\mathcal{M}'}$ is a singular $NDIPC_\beta$.*

Recall that $S_j^{\mathcal{M}'}$ denotes the $j^{th}$ MOQA′ random structure in the MOQA′ random bag $\mathcal{M}'$.

**Notation 91.** *Let $p_i$ denote the number of times that $M_i$ is repeated within a fixed po-structure in $P_\beta$'s set, $p_i \geq 0$.*

Why is $MR(S_j^{\mathcal{M}'})$, and therefore $FPS_x$, defined above as a multiset rather than a set? The definition of a singular $NDIPC$ includes inductive po-classes such as $Z$.

$$
\begin{aligned}
< Z > &::= () \\
< Z > &::= < number > \otimes < number > \\
< Z > &::= (< number > || < number >) || < Z > \\
< Z > &::= (< number > || < number >) \otimes < Z >
\end{aligned}
$$

Two identical fixed po-structures are construed from the two non base-case production rules in this tightly defined structural definition, i.e. the fixed po-structure of size two whose nodes are in parallel is construed twice. However, when $P_\beta$ of $S_j^{\mathcal{M}'}$ is the above $Z_\beta$, both instances of this fixed po-structure are separately represented in $MR(S_j^{\mathcal{M}'})$ (and therefore in $FPS_x$). The motivation for this duplication is to have a one-to-one relationship between $Z$'s

multi-repeat fixed po-structures and $Z$'s non base-case production rules. Accordingly, for any singular $NDIPC$, each multi-repeat fixed po-structure is uniquely associated with a specific non base-case production rule and vice versa. Therefore, $MR(S_j^{\mathcal{M}'})$ and $FPS_x$ are multisets.

There is not much flexibility in the structural arrangement of a fixed po-structure from a $DIPC$'s set. Such a fixed po-structure is composed of $p$ copies of the $DIPC$'s repeat fixed po-structure and, at a specific extremity of these $p$ structures, there is one non-empty fixed po-structure when the set containing all of the $DIPC$'s base-case fixed po-structures is not empty. The specific extremity is established by the $DIPC$'s tightly defined structural definition, as previously explained in Section 4.5.2.3 with the aid of inductive po-class $X$. So, for an empty-base $DIPC$, $p$ is the only variable in the structural arrangement of any fixed po-structure selected from its set. For a multi-base $DIPC$, an additional variable is introduced when its tightly defined structural definition has two or more base-case production rules. This variable represents the fixed po-structure, empty or otherwise, which has been placed at the specific extremity of the fixed po-structure selected from the multi-base $DIPC$'s set. However, for a fixed po-structure in a singular $NDIPC$'s set, there is not only choice with regard to how often each of the $NDIPC$'s multi-repeat fixed po-structures are repeated within that fixed po-structure but there is also choice with regard to the location of these multi-repeat fixed po-structures relative to one another within that fixed po-structure. This is because the structural definition of a singular $NDIPC$ does not impose a relative order between its multi-repeat fixed po-structures. For example, the structural definition of a singular $NDIPC$ does not stipulate that one of its multi-repeat fixed po-structures must always be above another of its multi-repeat fixed po-structures. While a multi-base singular $NDIPC$ has the same structural arrangement choice as a multi-base $DIPC$ when both tightly defined structural definitions have two or more base-case production rules, all multi-base singular $NDIPC$'s have a further structural arrangement choice when their tightly defined structural definitions allow a fixed po-structure, empty or otherwise, to be placed at *multiple* extremities. The following $IPC$, $A$, is a multi-base singular $NDIPC$ whose tightly defined structural definition introduces this extra alternative. $A$'s only base-case fixed po-structure of size one can be located either at the bottom or

to the left of a fixed po-structure from $A$'s set.

$$\begin{aligned} < A > &::= < number > \\ < A > &::= ((< number > \,||\, < number >) \,||\, < number >) \otimes < A > \\ < A > &::= < A > \,||\, (< number > \otimes < number >) \end{aligned}$$

So a singular $NDIPC$ has two additional structural options. Of the two, let us first of all introduce notation to represent the distribution of and the relative ordering between a singular $NDIPC$'s multi-repeat fixed po-structures within a fixed po-structure from the singular $NDIPC$'s infinite set.

Let $P_\beta$ of $S_j^{\mathcal{M}'}$ denote a singular $NDIPC_\beta$ with an infinite set. For the positive integer $x$, let $R_x$ denote a set of $n$ positive integers whose sum is $x$, i.e. denote a partition of $x$, $R_x = \{p_{x,1},\ p_{x,2}, \ldots,\ p_{x,n}\}$ and $1 \leq n \leq x$. Let $R_0$ denote the set whose only element is zero, $R_0 = \{0\}$. So $R_x$ will never contain the element zero, only $R_0$ contains this element. Here the $x$ of $R_x$ represents $p_i$ when it is a positive integer and the 0 of $R_0$ represents $p_i$ when it is zero; recall that $p_i$ denotes how often $M_i$ is repeated within a fixed po-structure in $P_\beta$'s set. The $p_i$ repetitions of $M_i$ do not have to be grouped all together within a fixed po-structure from $P_\beta$'s set, i.e. there do not have to be $p_i$ successive iterations of *op* involving $M_i$. So $R_{p_i}$ represents one possible way the $p_i$ repetitions of $M_i$ can be arranged into separate groups within a fixed po-structure from $P_\beta$'s set by representing the number and the size of these separate groups. Note the implicit assumption that there will be at least one other $M_i$ between any two of these groups, without which they would not be separate. Let $H_\infty$ denote the set of all $R_x$ sets for every $x$ and the set $R_0$. Hence $H_\infty$ is an infinite set, as indicated by its notation. $H_\infty$ represents *all* the possible ways that *all* $p_i$ repetitions of $M_i$ can be separated into groups within a fixed po-structure in $P_\beta$'s set. Now, let the $s$ sets $R_{p_1}, R_{p_2}, \ldots, R_{p_s}$ each denote some set in $H_\infty$, $R_{p_i} = \{p_{p_i,1},\ p_{p_i,2}, \ldots,\ p_{p_i,n}\}$ or $R_{p_i} = \{0\}$ and $1 \leq i \leq s$. (This $s$ is the same $s$ that represents how many multi-repeat fixed po-structures belong to $P_\beta$; $s$ is a constant value for a specific $P_\beta$.) So, for each $M_i$ in $MR(S_j^{\mathcal{M}'})$, there is a selection of *one possible way* from $H_\infty$ that the $p_i$ repetitions of $M_i$ can be arranged into separate groups within a fixed po-structure in $P_\beta$'s set. Note that two or more of these $s$ selections can refer to the same set in $H_\infty$. Let $P$ denote the multiset union of $R_{p_1}, R_{p_2}, \ldots, R_{p_s}$. Let $d$ denote one permutation of $P$'s

elements. $d$ represents one distribution of and relative ordering between $P_\beta$'s multi-repeat fixed po-structures within a fixed po-structure from $P_\beta$'s set. Let $d'$ denote permutation $d$ when adjacent elements from the same $R_{p_i}$ are replaced by their sum and when all zero elements are removed. $d'$ merges adjacent elements when these elements represent groups of the same $M_i$ because these groups are not separated by at least one other $M_i$ and therefore, are not actually separate groups. $d'$ drops all zero elements because each of these represent an empty group, a $M_i$ that is not present in the fixed po-structure from $P_\beta$'s set. So $d'$, more succinctly than $d$, represents one distribution of and relative ordering between the $p_1$ $M_1$s, $p_2$ $M_2$s, ... and $p_s$ $M_s$s within a fixed po-structure in $P_\beta$'s set.

Note that $R_{p_i}$ is selected from an infinite set. Hence the multiset $P$ can be any one of an infinite number of multisets and hence, the permutation $d'$ can be any one of an infinite number of permutations.

Inductive po-class $A$ provides a concrete example of this notation. Let $E$ denote a fixed po-structure of size three whose nodes are in parallel and let $F$ denote a fixed po-structure of size two whose nodes are in series. So $\{E, F\}$ shall be $A$'s multiset of multi-repeat fixed po-structures. Now let $R_{p_1}$, one possible way the $p_1$ repetitions of $E$ can be arranged into separate groups within a fixed po-structure in $A$'s infinite set, be $\{p_{p_1,1},\ p_{p_1,2},\ p_{p_1,3}\}$ and let $R_{p_2}$, one possible way the $p_2$ repetitions of $F$ can be arranged into separate groups within a fixed po-structure in $A$'s infinite set, be $\{p_{p_2,1},\ p_{p_2,2}\}$. Therefore:

$$P = \{p_{p_1,1},\ p_{p_1,2},\ p_{p_1,3},\ p_{p_2,1},\ p_{p_2,2}\}.$$

One permutation of $P$ is:

$$d = [p_{p_1,3},\ p_{p_1,1},\ p_{p_2,2},\ p_{p_1,2},\ p_{p_2,1}].$$

Note that none of the elements in $d$ are zero elements as both $R_{p_1}$ and $R_{p_2}$ have a cardinality greater than one. Hence:

$$d' = [(p_{p_1,3} + p_{p_1,1}),\ p_{p_2,2},\ p_{p_1,2},\ p_{p_2,1}].$$

This $d'$ still represents an infinite number of possible arrangements between the $p_1$ $E$s and $p_2$ $F$s within a fixed po-structure from $A$'s infinite set. It is an

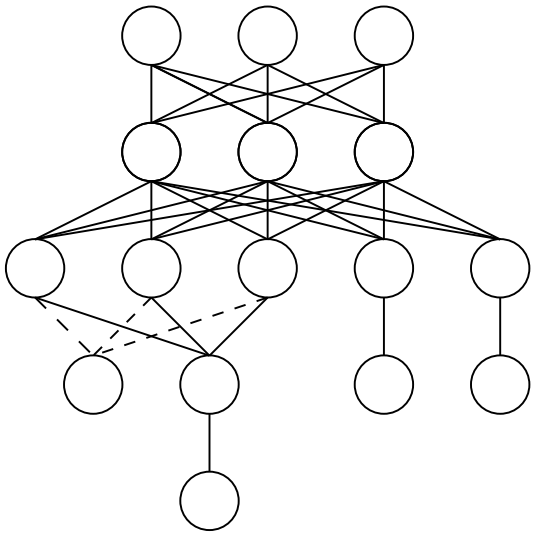


Figure 4.15: A fixed po-structure in $A$'s infinite set

infinite amount because, though this $d'$ specifies a relative ordering between $A$'s multi-repeat fixed po-structures, their distribution is still unknown; each variable in this $d'$ has yet to be replaced by some positive integer, of which there is of course an infinite number. So, for the next step, let $(p_{p_1,3} + p_{p_1,1}) = 2$, $p_{p_2,2} = 2$, $p_{p_1,2} = 1$ and $p_{p_2,1} = 1$. With these fixed values, $d'$ now represents one distribution of and relative ordering between the $p_1$ $E$s and $p_2$ $F$s within a fixed po-structure from $A$'s infinite set. In this work, a $d'$ permutation is always read from left to right. Figure 4.15 shows the fixed po-structure that the example $d'$ represents when it is read from left to right. For completeness, $A$'s base-case fixed po-structure is also included in Figure 4.15 and is indicated by the dotted lines. Therefore, Figure 4.15 is one of the fixed po-structures in $A$'s infinite set.

Now consider the second of the two additional structural options, which is only an option for a multi-base singular $NDIPC$. This is the possibility that the empty fixed po-structure and/or the $NDIPC$'s base-case fixed po-structures can be placed at multiple extremities of the fixed po-structures in the $NDIPC$'s set. For a fixed po-structure in the set of a multi-base singular $NDIPC$ whose structural definition allows for such a choice, the specific extremity that the empty fixed po-structure/one of the $NDIPC$'s base-case fixed po-structures is located at within that fixed po-structure depends on the distribution of and the relative ordering between the $NDIPC$'s multi-repeat

fixed po-structures within that fixed po-structure. This was demonstrated in the last example. The specific extremity that $A$'s base-case fixed po-structure is located at in Figure 4.15, which is to the left of the overall fixed po-structure as opposed to the bottom of it, is determined by the relative ordering represented by that $d'$.

With a clearer understanding of the extra structural variations available to a singular $NDIPC$, the equations involved in calculating the average-case cost of a MOQA function when applied to a singular $NDIPC_{\beta_{max}}$ with an infinite set can be developed. The main challenge presented by a singular $NDIPC$ with an infinite set is that there is an infinite number of ways to arrange its multi-repeat fixed po-structures, in contrast to the one way of arranging a $DIPC$'s repeat fixed po-structure. However, recall the discussion on page 116, which is in Section 4.5.2.3, about pushing a label up through the labeling of a fixed po-structure obtained from $y$ fixed po-structures in parallel, $y \geq 0$. The conclusion was that the relative ordering between these $y$ fixed po-structures is irrelevant when determining the average number of comparisons involved. So in this situation it is safe to ignore how a $NDIPC_{\beta_{max}}$'s multi-repeat fixed po-structures are relatively ordered within a fixed po-structure and instead be concerned only by how often each one occurs in the overall fixed po-structure.

**Definition 56** (Parallel singular $NDIPC$)**.** *A singular $NDIPC$ is a parallel singular $NDIPC$ if $||$ is the only operation that can connect any two of its multi-repeat fixed po-structures.*

Though $A$ on page 144 is not a parallel singular $NDIPC$, $B$ is.

$$\begin{aligned} < B > &::= < number > \\ < B > &::= ((< number > || < number >) || < number >) || < B > \\ < B > &::= < B > || (< number > \otimes < number >) \end{aligned}$$

$B$ shows that being a parallel singular $NDIPC$ does not necessarily exclude $\otimes$ from the production rules; $\otimes$ can still be used in the formation of the $NDIPC$'s base-case and multi-repeat fixed po-structures as the structural definition of $B$ makes evident in one of its non base-case production rules.

Now, let $P_\beta$ of $S_j^{\mathfrak{M}'}$ denote a parallel singular $NDIPC_{\beta_{max}}$ with an infinite set. The subset of this $P_\beta$ that the MOQA function is applied to must be

obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and up to one $C$ through successive iterations of $||$, $0 \leq r_1 \leq p_1$, $0 \leq r_2 \leq p_2, \ldots, 0 \leq r_s \leq p_s$. Under this condition, the following are the new $\sigma_{up}$, $\kappa_{up}$, $\tau_{up}$ and $\Delta_{up}$ equations for a parallel singular $NDIPC_{\beta_{max}}$. These new equations can, where appropriate, replace the standard binary versions in a MOQA function's average-case formula and thus transform it into a general average-case inductive formula. Their proofs are excluded because they are closely related to those given in Section 4.5.2.3.

**Proposition 16** ($\sigma_{up}$ for a parallel singular $NDIPC_{\beta_{max}}$)**.** *If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1s$, $r_2$ $M_2s$, ..., $r_s$ $M_ss$ through successive iterations of $||$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\frac{\sum_{i=1}^{s} r_i.|M_i|.\sigma_{up}(M_i)}{\sum_{i=1}^{s} r_i.|M_i|}. \tag{4.42}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1s$, $r_2$ $M_2s$, ..., $r_s$ $M_ss$ through successive iterations of $||$ and $\sum_{i=1}^{s} r_i = 0$, i.e. $\sigma_{up}$ is applied to an empty fixed po-structure, then its value is:*

$$0. \tag{4.43}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1s$, $r_2$ $M_2s$, ..., $r_s$ $M_ss$ and one $C$ through successive iterations of $||$, then its value is:*

$$\frac{(\sum_{i=1}^{s} r_i.|M_i|.\sigma_{up}(M_i)) + |C|.\sigma_{up}(C)}{(\sum_{i=1}^{s} r_i.|M_i|) + |C|}. \tag{4.44}$$

**Proposition 17** ($\kappa_{up}$ for a parallel singular $NDIPC_{\beta_{max}}$)**.** *If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1s$, $r_2$ $M_2s$, ..., $r_s$ $M_ss$ through successive iterations of $||$, then its value is:*

$$\sum_{i=1}^{s} r_i.\kappa_{up}(M_i). \tag{4.45}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1s$, $r_2$ $M_2s$, ..., $r_s$ $M_ss$ and one $C$ through successive iterations of $||$, then its value is:*

$$\left(\sum_{i=1}^{s} r_i.\kappa_{up}(M_i)\right) + \kappa_{up}(C). \tag{4.46}$$

**Proposition 18** ($\tau_{up}$ for a parallel singular $NDIPC_{\beta_{max}}$)**.** *If $\tau_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $||$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\frac{\sum_{i=1}^{s} r_i.|M_i|.\tau_{up}(M_i)}{\sum_{i=1}^{s} r_i.|M_i|}. \tag{4.47}$$

*If $\tau_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $||$ and $\sum_{i=1}^{s} r_i = 0$, i.e. $\tau_{up}$ is applied to an empty fixed po-structure, then its value is:*

$$0. \tag{4.48}$$

*If $\tau_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and one $C$ through successive iterations of $||$, then its value is:*

$$\frac{(\sum_{i=1}^{s} r_i.|M_i|.\tau_{up}(M_i)) + |C|.\tau_{up}(C)}{(\sum_{i=1}^{s} r_i.|M_i|) + |C|}. \tag{4.49}$$

To help improve the readability of the following proposition, let $L$ denote the set of labels on the entire fixed po-structure that the equation is considering. Therefore, $|L| = \sum_{i=1}^{s} r_i.|M_i|$ in Equation 4.50 and $|L| = (\sum_{i=1}^{s} r_i.|M_i|) + |C|$ in Equation 4.52.

**Proposition 19** ($\Delta_{up}$ for deleting the label with rank $k$, i.e. the $k^{\text{th}}$ smallest label, from a parallel singular $NDIPC_{\beta_{max}}$)**.** *If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $||$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\frac{\sum_{i=1}^{s} r_i. \sum_{x=1}^{|M_i|} \binom{k-1}{x-1}.\binom{|L|-k}{|M_i|-x}.\Delta_{up}(M_i, x)}{\prod_{i=1}^{s}\prod_{y=1}^{r_i} \binom{|L|-(\sum_{z=1}^{i-1} r_z.|M_z|)-(y-1).|M_i|}{|M_i|}}. \tag{4.50}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $||$ and $\sum_{i=1}^{s} r_i = 0$, i.e. $\Delta_{up}$ is applied to an empty fixed po-structure, then its value is:*

$$0. \tag{4.51}$$

*If $\Delta_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$*

*$M_s$s and one $C$ through successive iterations of $||$, then its value is:*

$$\left(\left(\sum_{i=1}^{s} r_i . \sum_{x=1}^{|M_i|} \binom{k-1}{x-1} . \binom{|L|-k}{|M_i|-x} . \Delta_{up}(M_i, x)\right) + \sum_{x=1}^{|C|} \binom{k-1}{x-1} . \binom{|L|-k}{|C|-x} . \Delta_{up}(C, x)\right) / \prod_{i=1}^{s} \prod_{y=1}^{r_i} \binom{|L| - (\sum_{z=1}^{i-1} r_z . |M_z|) - (y-1).|M_i|}{|M_i|}. \tag{4.52}$$

There would be similar $\sigma_{down}$, $\kappa_{down}$, $\tau_{down}$ and $\Delta_{down}$ equations for a parallel singular $NDIPC_{\beta_{max}}$.

Attention can now be turned to the series singular $NDIPC$.

**Definition 57** (Series singular $NDIPC$)**.** *A singular $NDIPC$ is a series singular $NDIPC$ if $\otimes$ is the only operation that can connect any two of its multi-repeat fixed po-structures.*

So now let $P_\beta$ of $S_j^{\mathcal{M}'}$ denote a series singular $NDIPC_{\beta_{max}}$ with an infinite set. The subset of this $P_\beta$ that the MOQA function is applied to must be obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and up to one $C$ through successive iterations of $\otimes$, $0 \leq r_1 \leq p_1$, $0 \leq r_2 \leq p_2, \ldots, 0 \leq r_s \leq p_s$. Under this condition, the following are the new $\sigma_{up}$ and $\kappa_{up}$ equations for a series singular $NDIPC_{\beta_{max}}$. Again, these new equations can, where appropriate, replace the standard binary versions in a MOQA function's average-case formula and thus transform it into a general average-case inductive formula. As before, their proofs are excluded because they are closely related to those given in Section 4.5.2.3.

**Proposition 20** ($\sigma_{up}$ for a series singular $NDIPC_{\beta_{max}}$)**.** *Let $\underline{M_x}$ denote that $M_x$ is the fixed po-structure that occurs at the bottom of the entire fixed po-structure considered by the equation, $1 \leq x \leq s$.*

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ...,*

*$r_s$ $M_s$s through successive iterations of $\otimes$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\left(\sum_{i=1}^{s} r_i.\sigma_{up}(M_i)\right) + \left(\sum_{i=1}^{x-1} r_i.|m(M_i)|\right) + (r_x - 1).|m(\underline{M_x})| + \sum_{i=x+1}^{s} r_i.|m(M_i)|. \tag{4.53}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $\otimes$ and $\sum_{i=1}^{s} r_i = 0$, i.e. $\sigma_{up}$ is applied to an empty fixed po-structure, then its value is:*

$$0. \tag{4.54}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and one $C$ through successive iterations of $\otimes$ and $C \in \underline{C(S_j^{\mathcal{M}'})}$, then its value is:*

$$\sigma_{up}(C) + \sum_{i=1}^{s} r_i.(\sigma_{up}(M_i) + |m(M_i)|). \tag{4.55}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and one $C$ through successive iterations of $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\left(\sum_{i=1}^{s} r_i.\sigma_{up}(M_i)\right) + \sigma_{up}(C) + \left(\sum_{i=1}^{x-1} r_i.|m(M_i)|\right) + (r_x - 1).|m(\underline{M_x})| + \left(\sum_{i=x+1}^{s} r_i.|m(M_i)|\right) + |m(C)|. \tag{4.56}$$

*If $\sigma_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and one $C$ through successive iterations of $\otimes$, $C \in \overline{C(S_j^{\mathcal{M}'})}$ and $\sum_{i=1}^{s} r_i = 0$, then its value is:*

$$\sigma_{up}(C). \tag{4.57}$$

Until Proposition 20, all the new equations in this work could be solved for a specific $IPC$ up to the point where $r$, or each $r_i$, and perhaps $k$ were

the only remaining variables in each equation. Now Proposition 20 introduces another variable that cannot be uniquely solved for a specific $IPC$. This variable is $\underline{M_x}$, which is a placeholder for the multi-repeat fixed po-structure at the bottom of the fixed po-structure when $C$ is not at the bottom of or even present in the fixed po-structure. (Knowing what is at the bottom of the fixed po-structure allows the average number of comparisons involved in swapping the label up onto the fixed po-structure to be excluded from the equation; the equation should only include the average number of comparisons that take place within the fixed po-structure.) While there is an infinite number of possible substitutions for $r$/each $r_i$, there are just $s$ possible substitutions for $\underline{M_x}$. Therefore, Equations 4.53 and 4.56 can be calculated for each of these $s$ substitutions. However, despite the strong likelihood of there being only a few non base-case production rules in the tightly defined structural definition of a series singular $NDIPC$, having additional results for each of these two equations further contributes to the total number of average-case solutions for a MOQA function applied to such a $NDIPC_{\beta_{max}}$ when the function's average-case formula involves Equations 4.53 and/or 4.56.

**Proposition 21** ($\kappa_{up}$ for a series singular $NDIPC_{\beta_{max}}$)**.** *Let $\overline{M_x}$ denote that $M_x$ is the fixed po-structure that occurs at the top of the entire fixed po-structure considered by the equation, $1 \leq x \leq s$.*

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $\otimes$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\kappa_{up}(\overline{M_x}). \tag{4.58}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s through successive iterations of $\otimes$ and $\sum_{i=1}^{s} r_i = 0$, i.e. $\kappa_{up}$ is applied to an empty fixed po-structure, then its value is:*

$$0. \tag{4.59}$$

*If $\kappa_{up}$ is applied to a fixed po-structure obtained from $r_1$ $M_1$s, $r_2$ $M_2$s, ..., $r_s$ $M_s$s and one $C$ through successive iterations of $\otimes$, $C \in \underline{C(S_j^{\mathcal{M}'})}$ and $\sum_{i=1}^{s} r_i > 0$, then its value is:*

$$\kappa_{up}(\overline{M_x}). \tag{4.60}$$

*If* $\kappa_{up}$ *is applied to a fixed po-structure obtained from* $r_1$ $M_1 s$, $r_2$ $M_2 s$, ..., $r_s$ $M_s s$ *and one* $C$ *through successive iterations of* $\otimes$ *and either* $C \in \underline{C(S_j^{\mathcal{M}'})}$ *and* $\sum_{i=1}^{s} r_i = 0$ *or* $C \in \overline{C(S_j^{\mathcal{M}'})}$*, then its value is:*

$$\kappa_{up}(C). \tag{4.61}$$

In the same way that extra average-case solutions can arise because of $\underline{M_x}$ in Proposition 20, extra average-case solutions can also arise because of $\overline{M_x}$ in Proposition 21.

There would also be similar $\sigma_{down}$ and $\kappa_{down}$ equations for a series singular $NDIPC_{\beta_{max}}$.

The $\tau_{up}$ and $\Delta_{up}$ equations for a series singular $NDIPC_{\beta_{max}}$ are not provided by this work. Recall, in Section 4.5.2.3, the $\tau_{up}$ and $\Delta_{up}$ proofs for a multi-base $DIPC_{\beta_{max}}$ when *op* is $\otimes$. It is simple to apply the logic used in these proofs to the case where the fixed po-structure is instead selected from a series singular $NDIPC_{\beta_{max}}$'s infinite set. Adapting the $\tau_{up}$ logic for such a fixed po-structure means that it would be necessary to determine the average number of comparisons that result when the label being pushed up through the fixed po-structure ends up in each instance of each multi-repeat fixed po-structure. Determining the average number of comparisons for each instance involves all of the multi-repeat fixed po-structure instances below it. Overall, this amounts to knowing not only the distribution of but also the relative ordering between the multi-repeat fixed po-structures within the selected fixed po-structure. Adapting the $\Delta_{up}$ logic for such a fixed po-structure means that, in determining the average number of comparisons, it would be necessary to determine the initial multi-repeat fixed po-structure instance that the label of rank $k$ is on. It would also be necessary to determine all of the multi-repeat fixed po-structure instances above this initial instance. So, once again, both the distribution of and the relative ordering between the multi-repeat fixed po-structures within the selected fixed po-structure is required information.

However, the equations in this work are realised because either 1), they safely ignore the relative ordering within the selected fixed po-structure or 2), there is a set number of possible relative orderings within the selected fixed po-structure and an equation is then provided for each of these orderings. This is why there are no difficulties in providing equations for both $DIPC_{\beta_{max}}$ types;

a fixed po-structure from a $DIPC$'s set does not have much flexibility in its structural arrangement/relative ordering, as detailed earlier in this section. So this equation format is not designed to deal with an unknown number of relative orderings within the selected fixed po-structure. (Besides $k$, distribution is the only unknown that the format can deal with, via the $r$ and $r_i$ variables. Hence, the presence of the above $\sigma_{up}$ and $\kappa_{up}$ equations for a series singular $NDIPC_{\beta_{max}}$ is due to the fact that distribution is the only unknown factor in both calculations.) Therefore, as neither $\tau_{up}$ nor $\Delta_{up}$ for a series singular $NDIPC_{\beta_{max}}$ with an infinite set is suited to this style of equation, because both calculations consider the relative ordering within the selected fixed po-structure and there is an infinite number of these possible, their equations are missing from this work.

Finally, the remaining tightly defined $NDIPC$s can be discussed. Such a $NDIPC$ is either a singular $NDIPC$ that is not in parallel or series or a $NDIPC$ that is not a singular $NDIPC$. For the former of these two $NDIPC$ categories, two operations can now connect the $NDIPC$'s multi-repeat fixed po-structures; $A$ above is an example of this. To determine $\sigma$, $\kappa$, $\tau$ or $\Delta$ for a fixed po-structure in this singular $NDIPC_{\beta_{max}}$'s infinite set, it would be necessary to know the sequence of operations that connect the multi-repeat fixed po-structures within the fixed po-structure because this knowledge is needed by all of the equations presented both here and in the MOQA book [63]. This is akin to knowing the distribution of and the relative ordering between the multi-repeat fixed po-structures within the selected fixed po-structure. Why make this correlation? Each production rule associated with each multi-repeat fixed po-structure instance within the selected fixed po-structure specifies what the adjoining operation is. Therefore, the full sequence of operations can be established from the selected fixed po-structure's $d'$ by reading this $d'$ from left to right. Hence, the requirement of knowing exact distribution and relative ordering. This means a return to the situation where access is needed to an infinite body of information that cannot be incorporated into the style of equations presented in this work. (The extra operation also broadens the set of choices for the specific extremity that the $NDIPC$'s base-case fixed po-structure, if it has at least one, can be located at. For a parallel singular $NDIPC$, the set of choices is $\{left, right\}$. For a series singular $NDIPC$, the set of choices is $\{bottom, top\}$. Now for a singular $NDIPC$ that is neither of

these, the set of choices is $\{left, right, bottom, top\}$. As before, the structural definition of the $NDIPC$ will decide the subset of choices available to the fixed po-structures in its set.)

A tightly defined $NDIPC$ in the second category, that is, a tightly defined $NDIPC$ that is not a singular $NDIPC$, is not a singular self-identity $IPC$ and maybe not even a self-identity $IPC$. The lack of the former characteristic would mean that there is more than one head non-terminal in the body of at least one non base-case production rule in the $NDIPC$'s structural definition; see $BT$'s structural definition in Section 4.4.2. The lack of the latter characteristic would mean that the body of at least one non base-case production rule in the $NDIPC$'s structural definition references another non-trivial structural definition which may or may not be inductive. The added complexity introduced by a non singular $NDIPC$ is left for unravelling by future work, though Chapter 7 points to more fundamental issues that future work should address first. However, it seems safe to conclude that some of these $NDIPC_{\beta_{max}}$s, particularly those that are not singular self-identity $IPC$s, will run into the above relative ordering difficulties if the above equation techniques are applied in determining $\sigma$, $\kappa$, $\tau$ or $\Delta$.

Therefore, when $P_\beta$ of $S_j^{\mathcal{M}'}$ represents a $NDIPC_{\beta_{max}}$ in either of these two final categories or when it represents a series singular $NDIPC_{\beta_{max}}$ and the average-case formula for the MOQA function applied to $P_\beta$ involves $\tau$ or $\Delta$, then, at this point in time, the average-case cost of the MOQA function applied to $P_\beta$ must be supplied to the MOQA static analysis tool by a user. However, when $P_\beta$ of $S_j^{\mathcal{M}'}$ represents a parallel singular $NDIPC_{\beta_{max}}$ or when it represents a series singular $NDIPC_{\beta_{max}}$ and the average-case formula for the MOQA function applied to $P_\beta$ involves $\sigma$ or $\kappa$, then new equations that give rise to general average-case inductive formulas have been developed[7], thereby enabling the MOQA static analysis tool to provide average-case behaviours for certain MOQA functions when they are applied to certain groups of $NDIPC_{\beta_{max}}$s whose sets are infinite.

[7]Note that these equations will also apply for the min-heap label ordering.

### 4.5.3 The Average-case Cost of a MOQA Function Applied to $SPC_{\beta_{max},n}$ and $GSPC_{\beta_{max},Z}$

As the average-case cost of the MOQA split function has already been examined, the focus of this section is the average-case cost of a MOQA function when applied to the result of a MOQA split function.

First of all, suppose the result is a split po-class. This is a condensed representation. Therefore, the average-case cost of a MOQA function when applied to a split po-class can be separated into the average-case cost of the MOQA function when applied to each of the fixed po-structures represented by the split po-class, which has already been considered in Section 4.5.1.

Second of all, suppose the result is a general split po-class so let $P_\beta$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, denote $GSPC_{\beta_{max},Z}$. When $GSPC_{\beta_{max},Z}$'s set is finite, then the average-case cost of the MOQA function applied to this $P_\beta$ can be determined in the same manner as outlined in Section 4.5.2.1 for $IPC_\beta$s whose sets are finite. When $GSPC_{\beta_{max},Z}$'s set is infinite, then the average-case cost of the MOQA function is its average-case cost when applied to $FPS_{\beta_{max},a,b}$, where $a = n_{S_j^{\mathcal{M}'}} - 1 - b$ and $0 \leq b \leq n_{S_j^{\mathcal{M}'}} - 1$. This $FPS_{\beta_{max},a,b}$ can be viewed as a compound structure. The compound structure interpretation is a discrete inductive po-class of size $n_{S_j^{\mathcal{M}'}} - 1 - b$ in series with a fixed po-structure of size one in series with a discrete inductive po-class of size $b$. As the set belonging to $GSPC_{\beta_{max},Z}$ is infinite, the sets belonging to both of these discrete inductive po-classes are infinite. So can the average-case cost of a MOQA function that is applied to such a composite variable be statically determined? This is the subject of the following section. However, the conclusion taken from Section 4.5.4 is "yes", because it is possible to statically determine the average-case cost of a MOQA function when correctly applied to both a discrete inductive po-class whose set is infinite and a fixed po-structure.

### 4.5.4 The Average-case Cost of a MOQA Function Applied to $CS_\beta$

A compound structure consists of a finite number of the data structure representations discussed in this chapter. Therefore, the MOQA static analysis tool

can determine the average-case cost of the MOQA function applied to some subset of a compound structure if it can determine the average-case cost of the function applied to each data structure representation in that subset. The previous sections have just detailed the representations for which such timing information can be statically obtained, along with how.

## 4.6 The Number of Canonically-ordered Labelings on $IPC_\beta$

Knowing the total number of distinct canonically-ordered labelings on each $P_\beta$ in the MOQA′ random bag is also part of statically deriving the complete average-case cost of the MOQA function applied to that bag; see Section 2.2.

There are already equations for the total number of distinct canonically-ordered labelings on $P_\beta$ when it represents some fixed po-structure, c.f. Equations 2.12 and 2.13. Corresponding equations are also needed for when $P_\beta$ represents an $IPC_{\beta_{max}}$ with an infinite set. However, such equations are only needed for $IPC_{\beta_{max}}$s for which the average-case cost of a MOQA function can be statically calculated. Currently, these are the $IPC_{\beta_{max}}$s of Section 4.5.2.3 and 4.5.2.4. There is no need to deal with other $IPC_{\beta_{max}}$s because, when the MOQA static analysis tool cannot resolve the average-case cost of a MOQA function that is applied to an $IPC_{\beta_{max}}$, an impasse is reached whether or not it can calculate the relative frequency of that $IPC_{\beta_{max}}$ occurring in the MOQA′ random bag.

If $P_\beta$ represents an **empty-base $DIPC_{\beta_{max}}$** and the structural definition operation that connects its repeat fixed po-structure is ||, then either Equation 4.62 or Equation 4.63 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is an empty fixed po-structure, then:

$$|L(FPS_\beta)| = 0. \tag{4.62}$$

Recall that $L(FPS_\beta)$ denotes the set of all canonically-ordered labelings of the fixed po-structure $FPS$ that has the label ordering $\beta$ on it.

If $FPS_\beta$ is selected from a $DIPC_{\beta_{max}}$'s set, $r > 0$ and

$$FPS_\beta = \overbrace{SR_j^{\mathcal{M}'} \;||\; \ldots \;||\; SR_j^{\mathcal{M}'}}^{r},$$

then Hickey [35] states that:

$$|L(FPS_\beta)| = \left( \prod_{i=1}^{r} \binom{i.|SR_j^{\mathcal{M}'}|}{|SR_j^{\mathcal{M}'}|} \right) . |L(SR_j^{\mathcal{M}'})|^r. \tag{4.63}$$

A relevant aside is that, in addition to Equation 4.62, there should also be the following simple base-case equation for both the equations in this section and Equations 2.12 and 2.13.

If $FPS_\beta$ is a fixed po-structure of size one, then:

$$|L(FPS_\beta)| = 1. \tag{4.64}$$

On the other hand, if $P_\beta$ represents an **empty-base $\boldsymbol{DIPC_{\beta_{max}}}$** and the structural definition operation that connects its repeat fixed po-structure is $\otimes$, then either Equation 4.62 or Equation 4.65 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from a $DIPC_{\beta_{max}}$'s set, $r > 0$ and

$$FPS_\beta = \overbrace{SR_j^{\mathcal{M}'} \;\otimes\; \ldots \;\otimes\; SR_j^{\mathcal{M}'}}^{r},$$

then Hickey [35] states that:

$$|L(FPS_\beta)| = \prod_{i=1}^{r} |L(SR_j^{\mathcal{M}'})| = |L(SR_j^{\mathcal{M}'})|^r. \tag{4.65}$$

Note this work's minor addendum of placing a restriction on $r$'s range in both of Hickey's equations, Equations 4.63 [35] and 4.65 [35]. Note also that Equation 4.63 has been rearranged in this work.

The remaining equations shall now be provided. If $P_\beta$ represents an **multi-base $\boldsymbol{DIPC_{\beta_{max}}}$** and the structural definition operation that connects its repeat fixed po-structure is $||$, then Equation 4.62, 4.63 or 4.66 will provide

the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from a multi-base $DIPC_{\beta_{max}}$'s set and

$$FPS_\beta = \overbrace{SR_j^{\mathcal{M}'} \;||\; \ldots \;||\; SR_j^{\mathcal{M}'}}^{r} \;||\; C,$$

then it can be shown that:

$$|L(FPS_\beta)| = \left(\prod_{i=1}^{r} \binom{i.|SR_j^{\mathcal{M}'}| + |C|}{|SR_j^{\mathcal{M}'}|}\right).|L(SR_j^{\mathcal{M}'})|^r.|L(C)|. \tag{4.66}$$

If $P_\beta$ represents an **multi-base $\boldsymbol{DIPC_{\beta_{max}}}$** and the structural definition operation that connects its repeat fixed po-structure is $\otimes$, then Equation 4.62, 4.65 or 4.67 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from a multi-base $DIPC_{\beta_{max}}$'s set and

$$FPS_\beta = \overbrace{SR_j^{\mathcal{M}'} \;\otimes\; \ldots \;\otimes\; SR_j^{\mathcal{M}'}}^{r} \;\otimes\; C,$$

then it can be shown that:

$$|L(FPS_\beta)| = |L(SR_j^{\mathcal{M}'})|^r.|L(C)|. \tag{4.67}$$

If $P_\beta$ represents a **parallel empty-base singular $\boldsymbol{NDIPC_{\beta_{max}}}$**, then either Equation 4.62 or Equation 4.68 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from a parallel singular $NDIPC_{\beta_{max}}$'s set, $\sum_{i=1}^{s} r_i > 0$ and

$$FPS_\beta = \overbrace{M_1 \;||\; \ldots \;||\; M_1}^{r_1} \;||\; \ldots \;||\; \overbrace{M_s \;||\; \ldots \;||\; M_s}^{r_s},$$

then it can be shown that:

$$\begin{aligned} |L(FPS_\beta)| \;=\; & \left(\prod_{i=1}^{s}\prod_{y=1}^{r_i} \binom{|FPS_\beta| - (\sum_{z=1}^{i-1} r_z.|M_z|) - (y-1).|M_i|}{|M_i|}\right). \\ & \prod_{i=1}^{s} |L(M_i)|^{r_i}, \end{aligned} \tag{4.68}$$

where

$$|FPS_\beta| = \sum_{i=1}^{s} r_i.|M_i|.$$

If $P_\beta$ represents a **series empty-base singular $\boldsymbol{NDIPC_{\beta_{max}}}$**, then either Equation 4.62 or Equation 4.69 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from a series singular $NDIPC_{\beta_{max}}$'s set, $\sum_{i=1}^{s} r_i > 0$ and

$$FPS_\beta = \overbrace{M_1 \otimes \ldots \otimes M_1}^{r_1} \otimes \ldots \otimes \overbrace{M_s \otimes \ldots \otimes M_s}^{r_s},$$

then it can be shown that:

$$|L(FPS_\beta)| = \prod_{i=1}^{s} |L(M_i)|^{r_i}. \tag{4.69}$$

If $P_\beta$ represents a **parallel multi-base singular $\boldsymbol{NDIPC_{\beta_{max}}}$**, then Equation 4.62, 4.68 or 4.70 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from a parallel multi-base singular $NDIPC_{\beta_{max}}$'s set and

$$FPS_\beta = \overbrace{M_1 \parallel \ldots \parallel M_1}^{r_1} \parallel \ldots \parallel \overbrace{M_s \parallel \ldots \parallel M_s}^{r_s} \parallel C,$$

then it can be shown that:

$$\begin{aligned} |L(FPS_\beta)| \;=\; & \prod_{i=1}^{s} \prod_{y=1}^{r_i} \binom{|FPS_\beta| - (\sum_{z=1}^{i-1} r_z.|M_z|) - (y-1).|M_i|}{|M_i|}. \\ & (\prod_{i=1}^{s} |L(M_i)|^{r_i}).|L(C)|, \end{aligned} \tag{4.70}$$

where

$$|FPS_\beta| = (\sum_{i=1}^{s} r_i.|M_i|) + |C|.$$

If $P_\beta$ represents a **series multi-base singular $\boldsymbol{NDIPC_{\beta_{max}}}$**, then Equation 4.62, 4.69 or 4.71 will provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set.

If $FPS_\beta$ is selected from series multi-base singular $NDIPC_{\beta_{max}}$'s set and

$$FPS_\beta = \overbrace{M_1 \otimes \ldots \otimes M_1}^{r_1} \otimes \ldots \otimes \overbrace{M_s \otimes \ldots \otimes M_s}^{r_s} \otimes C,$$

then it can be shown that:

$$|L(FPS_\beta)| = (\prod_{i=1}^{s} |L(M_i)|^{r_i}).|L(C)|. \tag{4.71}$$

The particular extremity at which $C$ is located in the $FPS_\beta$ of the above germane equations has no affect on their result. Hence, the placement of $C$ at the end of the chain in the notations for these $FPS_\beta$s is a random choice of no significance. Likewise, where each instance of each $M_i$ is located within the $FPS_\beta$ of the above germane equations does not affect their result. So, once again, their sequential placement in the notations for these $FPS_\beta$s is a random choice of no significance.

So clearly there are multiple equations for counting the total number of distinct canonically-ordered labelings on each type of $IPC_{\beta_{max}}$ that the above $P_\beta$ represented. (Note that these equations will also apply for the min-heap label ordering.) To illustrate these multiple equations, we just saw that when $P_\beta$ represents a series multi-base singular $NDIPC_{\beta_{max}}$, then Equations 4.62, 4.69 and 4.71 provide the distinct canonically-ordered labeling count on a fixed po-structure selected from its set. Similarly, there are multiple average-case equations for each of these $IPC_{\beta_{max}}$ types in Sections 4.5.2.3 and 4.5.2.4. There is a correspondence between these two equation categories: the average-case equation involved in obtaining the average-case cost of a MOQA function applied to one of these $IPC_{\beta_{max}}$ types determines which of the above equations is involved in obtaining the relative frequency of that $IPC_{\beta_{max}}$ occurring in the MOQA random bag. For example, if $P_\beta$ represents a parallel multi-base singular $NDIPC_{\beta_{max}}$ and Equation 4.44 is part of determining the average-case of a MOQA function applied to $P_\beta$, then Equation 4.70 is part of determining the relative frequency of $P_\beta$ occurring in the MOQA random bag.

## 4.7 The Number of Canonically-ordered Labelings on $SPC_{\beta_{max},n}$ and $GSPC_{\beta_{max},Z}$

If a non-empty $FPS_\beta$ is selected from a $SPC_{\beta_{max},n}$'s set or a $GSPC_{\beta_{max},Z}$'s set, then it is simple to infer from the MOQA book [63] that the total number of distinct canonically-ordered labelings on it are:

$$\frac{(a+b)!}{\binom{a+b}{b}}.$$

Equation 4.62 gives the total when the selected fixed po-structure is empty.

When $H_\beta$ of $S_j^{\mathcal{M}}$, which is the $j^{th}$ MOQA random structure in the MOQA random bag $M$, denotes the fixed po-structure $FPS_{\beta_{max},a,b}$, the MOQA book [63] makes the point that $|L(S_j^{\mathcal{M}})|.M^{S_j^{\mathcal{M}}}$, which is the relative frequency of $S_j^{\mathcal{M}}$ occurring in $M^{S_j^{\mathcal{M}}}$, can be reduced to $(a+b)!$. (Recall that $L(S_j^{\mathcal{M}})$ denotes the set of all canonically-ordered labelings of the $j^{th}$ MOQA random structure in the MOQA random bag $M$ when that structure has the label ordering $\beta$ on it.) This is because:

$$|L(S_j^{\mathcal{M}})|.M^{S_j^{\mathcal{M}}} = \frac{(a+b)!}{\binom{a+b}{b}}.\binom{a+b}{b} = (a+b)!.$$

Therefore, when $P_\beta$ of $S_j^{\mathcal{M}'}$ denotes $SPC_{\beta_{max},n}$, $|L(S_j^{\mathcal{M}'})|.M^{S_j^{\mathcal{M}'}}$ can be reduced to $(n-1)!$; recall that $a+b=n-1$. Similarly, when $P_\beta$ of $S_j^{\mathcal{M}'}$ denotes $GSPC_{\beta_{max},Z}$, then $|L(S_j^{\mathcal{M}'})|.M^{S_j^{\mathcal{M}'}}$ can be reduced to $(n_{S_j^{\mathcal{M}'}}-1)!$.

## 4.8 Chapter Summary

The MOQA approach to statically determining average-case behaviour centres on iterating through the shape of a fixed po-structure. By adhering to the essence of MOQA, this work has added to the data structure types that can be included in the MOQA$'$ random bag. While Chapter 6 presents other mechanisms for statically attaining the average-case behaviour of data structure types, one aim of this work, and specifically of this chapter, is to push the current boundaries of MOQA. Hence, the style of the new equations developed here for $\sigma$, $\kappa$, $\tau$, $\Delta$ and $L(S_j^{\mathcal{M}'})$. Due to these equations, there may be more

than one average-case solution for the algorithm under static analysis[8] and, when this occurs, a user may not be able to extract meaningful average-case information from these multiple solutions. However, this chapter has shown that this is not a challenge new to MOQA and the MOQA approach may still have the benefit of being able to provide a unique insight into the formulation of average-case behaviour for particular algorithms. Also achieved in this chapter was the further augmentation of the general MOQA theory through the expansion of its definitions and additional categorisations.

[8] An example of an algorithm whose MOQA static analysis would yield more than one average-case solution is given on page 138.

# Chapter 5

# Duplicate Labels

All the labels of a canonically-ordered labeling are distinct by definition and a MOQA/MOQA′ random structure only deals with canonically-ordered labelings. Though it is not unusual to take the stance that a data structure's values are distinct, this chapter considers whether duplicate labels can be integrated into MOQA's approach to average-case analysis.

## 5.1 The Duplicate Label Question

To aid with this discussion, this section will begin with a duplicate label example. Figure 5.1 shows, for the label set $\{1, 2\}$ and the max-heap label ordering, all the distinct labelings of a discrete fixed po-structure of size three when the label value 2 is assigned twice. It is possible to distinguish between the two identical label values in each labeling of Figure 5.1 by identifying one of them as $2_a$ and the other as $2_b$. However, if their new identities resulted in these label values being viewed as distinct, then Figure 5.1 would no longer correctly show all of the fixed po-structure's distinct labelings. Rather, Figure 5.2 would show, for the label set $\{1, 2_a, 2_b\}$ and the max-heap label ordering, all the distinct labelings of a discrete fixed po-structure of size three.

Now consider the average-case cost of a MOQA function when duplicate labels are involved. Continuing with the discrete fixed po-structure of size three example, give specific consideration to this for the MOQA product function that products from left to right the first node above the next two node. For each labeling in Figure 5.1, Figure 5.3 shows the result of this MOQA product function when the connections between minimal and maximal nodes have been

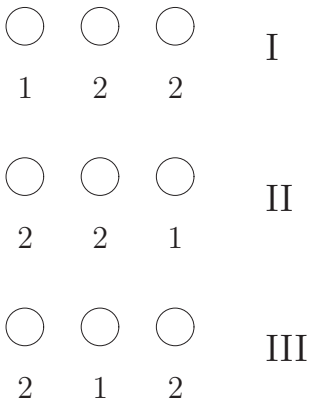



Figure 5.1: For the label set $\{1, 2\}$ and $\beta_{max}$, all the distinct labelings of a discrete fixed po-structure of size three when 2 is assigned twice

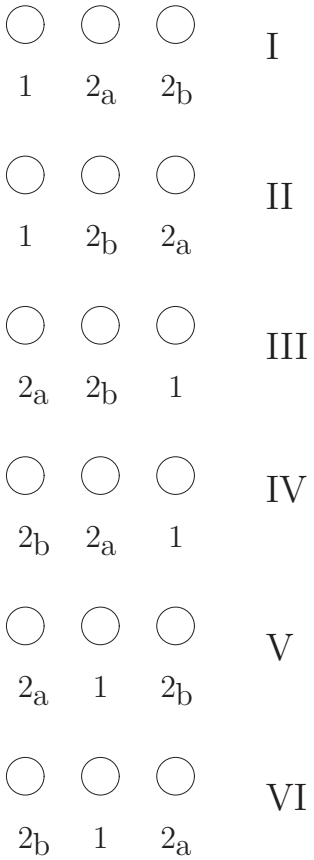



Figure 5.2: For the label set $\{1, 2_a, 2_b\}$ and $\beta_{max}$, all the distinct labelings of a discrete fixed po-structure of size three

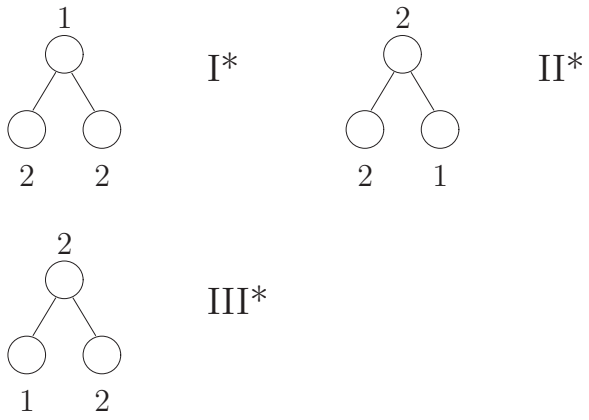



Figure 5.3: Figure 5.1 just after the first node from left to right is connected above the next two nodes

made but, as of yet, the function has not broached adjusting the labeling. For each labeling in Figure 5.2, Figure 5.4 shows likewise.

During the process of ensuring that a labeling is consistent with its label ordering, the MOQA product and deletion functions will select the node with the smallest/largest label in some isolated subset of nodes; see Section 2.2. If there is more than one node with the smallest/largest label in that subset, then these duplicate labels introduce a further choice; which node is the minimum/maximum node in that subset? The following section will look at some ways in which this decision can be made and, through the examples just given, discuss their impact on average-case cost.

## 5.2 Random Selection

When a MOQA function is faced with choosing between nodes whose label values are equal, then one approach to this problem is that the MOQA function randomly selects one of these nodes. For the sake of simplicity, it shall be naively assumed that the implementer of any random selection method discussed in this section does not sin[1] and therefore, each node in some group of nodes from which there is a random selection is equally likely to be selected.

The push-down and push-up logic of the MOQA product function will find

[1] "Anyone who uses arithmetic methods to produce random numbers is in a state of sin." - John von Neumann

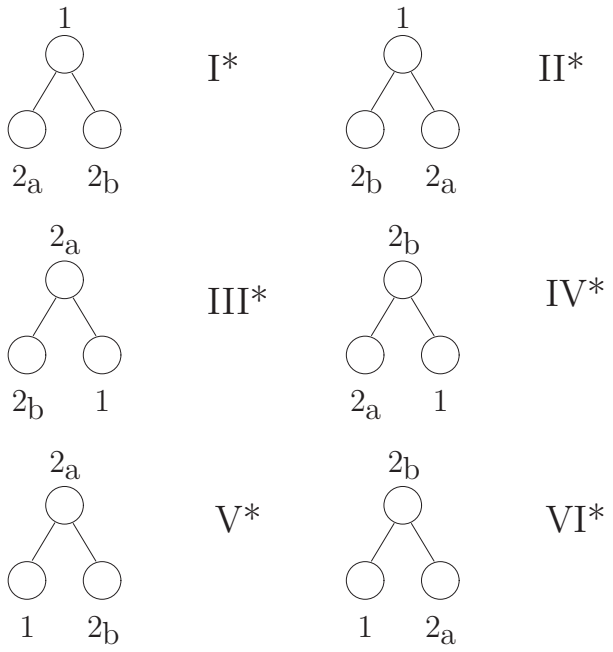



Figure 5.4: Figure 5.2 just after the first node from left to right is connected above the next two nodes

that $II^*$ and $III^*$ in Figure 5.3 are already consistent with the max-heap label ordering. Only $I^*$ in Figure 5.3 will need adjustment. Applying the approach just suggested, let the leftmost minimal node of $I^*$ be the node randomly selected for the swap operation. So, after that swap operation and thus the MOQA product function completes, $I^*$ becomes $III^*$. The other equally likely possibility is that the rightmost minimal node of $I^*$ is the node randomly selected for the swap operation. If so, then, after that swap operation and thus the MOQA product function completes, $I^*$ becomes $II^*$.

Hence, the MOQA product function will be deterministic for inputs II and III of Figure 5.1 but non-deterministic for input I of Figure 5.1 due to random selection. Though the MOQA product function will always produce three outputs for these three inputs, it is not sure whether these three outputs will be $\{II^*, III^*, III^*\}$ or $\{II^*, II^*, III^*\}$. In other words, the three outputs will be $\{II^*, III^*, (II^* \mid III^*)\}$. So what is the average number of label-to-label comparisons for this random selection MOQA product function during its examination and possible adjustment of the labelings in Figure 5.3? Figure 5.5 illustrates the new code that could be used for this random se-

lection[2]. Its *getMaximumElement* would replace the code for $\vee(f, I_{\beta_{max}})$ in the standard MOQA product function. There would be a corresponding *getMinimumElement*, which would replace the code for $\wedge(f, I_{\beta_{max}})$ in the standard MOQA product function. Otherwise, the standard MOQA product function as described in Section 2.2 would remain untouched. Let $T_{prod}(X)$ denote the *total* number of label-to-label comparisons for this random selection MOQA product function when it examines and possibly adjusts the labeling on $X$, when $X$ is a fixed po-structure with some labeling on it. Therefore, the average-case cost of this random selection MOQA product function when applied to the labelings in Figure 5.3 is:

$$\frac{1}{6}.T_{prod}(I^*) + \frac{1}{6}.T_{prod}(I^*) + \frac{1}{3}.T_{prod}(II^*) + \frac{1}{3}.T_{prod}(III^*).$$

The first $T_{prod}(I^*)$ is for when the MOQA product function randomly selects the leftmost minimal node of $I^*$ for the swap operation and the second $T_{prod}(I^*)$ is for when it randomly selects the rightmost minimal node. Using the code in Figure 5.5, this average-case cost can be evaluated to:

$$\left(\frac{1}{6}\right).10 + \left(\frac{1}{6}\right).12 + \left(\frac{1}{3}\right).6 + \left(\frac{1}{3}\right).4 \quad = \quad 7. \tag{5.1}$$

Note that Equation 5.1 ignores the number of comparisons involved in the random selection of a node and in iterating through a set of nodes because only label-to-label comparisons are under scrutiny here. Though MOQA's domain is *exact average-case cost*, this example shows that the non-determinism of this random selection technique causes a shift into the area of *expected average-case cost*.

Duplicate labels are briefly mentioned in the MOQA book [63], which states that they are "allowed in MOQA" when the technique for handling them is random selection. However, Schellekens [63] advocates random selection via a different approach. Prior to the execution of *every* MOQA program, he suggests that a unique subscript from a totally-ordered set of elements is randomly assigned to each label in the labeling on the input series-parallel $H_\beta$; so the size of this totally-ordered set must be at least the number of nodes in the input

[2]The only aim in presenting this code is to clearly demonstrate some of the involved comparisons. Hence, it is not necessary for this code, and that which follows, to agree with the *MOQA-Java* syntax.

```
/**
 * Returns the maximum element in the specified set.
 * @param set a non-empty set of elements.
 */
public Element getMaximumElement(Set<Element> set) {
    List<Element> maximumElements = getMaximumElements(set);
    if (maximumElements.size() > 1) {
        // Return element randomly selected from maximumElements.
    }
    return maximumElements.get(0);
}

/**
 * Returns a list of the maximum elements in the specified set.
 * @param set a non-empty set of elements.
 */
private List<Element> getMaximumElements(Set<Element> set) {
    List<Element> maximumElements = new ArrayList<Element>();
    Iterator<Element> iter = set.iterator();
    maximumElements.add(iter.next());
    Element currentElement; int compare;
    while (iter.hasNext()) {
        currentElement = iter.next();
        compare = maximumElements.get(0).
            compareTo(currentElement);
        if (compare < 0) {
            maximumElements.clear();
            maximumElements.add(currentElement);
        } else if (compare == 0) {
            maximumElements.add(currentElement);
        }
    }
    return maximumElements;
}

public class Element implements Comparable<Element> {
    ...

    /**
     * Returns a negative integer, zero, or a positive integer
     * when this element is less than, equal to, or greater than
     * the specified element.
     */
    public int compareTo(Element element) {
        return (this.label < element.label ? -1 :
                (this.label == element.label ? 0 : 1));
    }
}
```

Figure 5.5: New code for a random selection technique discussed in this section

$H_\beta$. Then, when comparisons between two nodes reveal their label values to be equal, a further comparison takes place between their label subscript values to determine the smaller/larger of the two. Therefore, nodes whose label values were deemed to be equal before subscript assignment are now no longer equal due to their values having distinct subscripts. Though this point is not stated in the MOQA book [63], its random selection technique again causes a domain shift into the arena of expected average-case cost because $|H_\beta| - 1$ nodes are randomly selected during the subscript assignment that takes place at the beginning of every MOQA program. This differs from the other random selection technique just discussed because for that technique a node is randomly selected only when the required node has multiple possibilities, which has the minor advantage of only engaging in random selection when duplicate labels are actually present.

Figure 5.2 displays how the MOQA book's random selection technique [63] would transform the input labelings of Figure 5.1. So what is the average number of label-to-label comparisons for the MOQA book's random selection MOQA product function during its examination and possible adjustment of the labelings in Figure 5.4, when $a < b$ is taken to be the ordering of the label subscripts? Figure 5.6 illustrates the new code that could be used for Schellekens's random selection [63]. Its $getMaximumElement$ would replace the code for $\vee(f, I_{\beta_{max}})$ in the standard MOQA product function. There would be a corresponding $getMinimumElement$, which would replace the code for $\wedge(f, I_{\beta_{max}})$ in the standard MOQA product function. Additionally, the push-down logic of the standard MOQA product function would have to be modified to include subscript comparison and, therefore, would become:

```
while ⌊v(min_f, I_βmax)⌋ ⊆ I_βmax and
    v(min_f, I_βmax).compareTo(∨(f, ⌊v(min_f, I_βmax)⌋)) < 0
    swap(v(min_f, I_βmax), ∨(f, ⌊v(min_f, I_βmax)⌋))
```

There would have to be a similar modification to the push-up logic of the standard MOQA product function. Hence, using the code in Figure 5.6, the average number of label-to-label comparisons, which includes label subscript comparisons, in rearranging the labelings in Figure 5.4 so that they become

```
/**
 * Returns the maximum element in the specified set.
 * @param set a non-empty set of elements.
 */
public Element getMaximumElement(Set<Element> set) {
    Iterator<Element> iter = set.iterator();
    Element maximumElement = iter.next();
    Element currentElement;
    while (iter.hasNext()) {
        currentElement = iter.next();
        if (maximumElement.compareTo(currentElement) < 0) {
            maximumElement = currentElement;
        }
    }
    return maximumElement;
}

public class Element implements Comparable<Element> {
    ...

    /**
     * Returns a negative integer or a positive integer when this
     * element is less than or greater than the specified element.
     */
    public int compareTo(Element element) {
        return (this.label < element.label ?
                -1 : (this.label == element.label ?
                (this.labelSubscript < element.labelSubscript ?
                -1 : 1) : 1));
    }
}
```

Figure 5.6: New code for Schellekens's random selection technique [63]

the labelings in Figure 5.7 is:

$$\left(\frac{1}{6}\right).13 + \left(\frac{1}{6}\right).12 + \left(\frac{1}{6}\right).14 + \left(\frac{1}{6}\right).7 + \left(\frac{1}{6}\right).12 + \frac{1}{6} \quad = \quad \frac{32}{3}. \qquad (5.2)$$

The first fraction is the average number of label-to-label comparisons in rearranging $I^*$ to $I^{**}$, the second fraction is the average number of label-to-label comparisons in rearranging $II^*$ to $II^{**}$, etc. Note that Equation 5.2 ignores the number of comparisons involved in the random selection of a node, i.e. the $|H_\beta| - 1$ nodes randomly selected prior to each execution of a MOQA program, and in iterating through a set of nodes for the same reason that these comparisons are ignored by Equation 5.1.

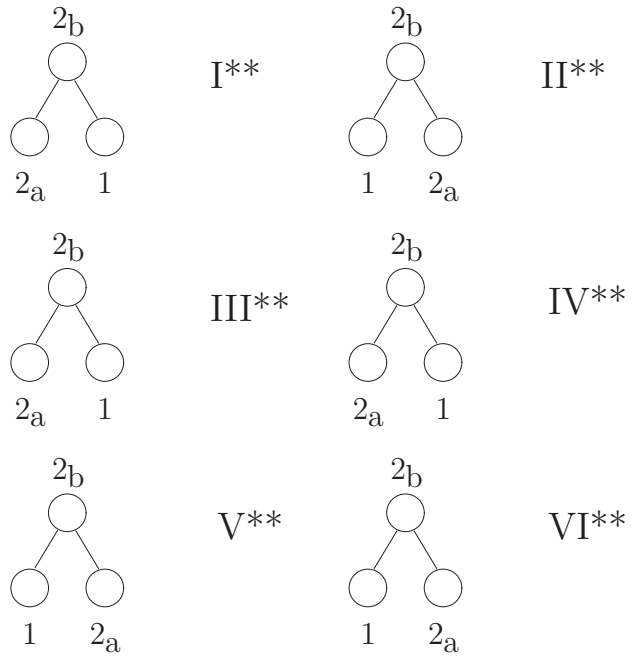



Figure 5.7: The labelings of Figure 5.4 after adjustment by the MOQA product function

However, the MOQA book [63] asserts that its random selection "amounts to considering all labels distinct and hence our analysis, which is carried out on states and under the assumption of distinct labels, will yield the correct result". Is this hypothesis of equivalence correct? Does the average-case cost of this random selection technique really never differ from the average-case cost that would normally be determined by MOQA? The above example for the MOQA book's random selection technique [63] makes this claim simple to test as its average-case cost can be compared to the average-case cost that would result from the MOQA static analysis tool. So, for a discrete fixed po-structure of size three, what is the average-case cost of the standard MOQA product function that products from left to right the first node above the next two nodes? The MOQA static analysis tool uses the average-case formula for the standard MOQA product function in Section 2.3 to give an answer of 10/3. Clearly, this is not the result of Equation 5.2. Therefore, Schellekens's supposition [63] does not hold for this relatively simple example. Intuitively this makes sense because the extra comparisons introduced by the MOQA book's random selection technique [63] would have to result in a higher average;

there are extra comparisons even when prior to subscript assignment there are no duplicate labels in the labeling on the input $H_\beta$. Hence, this work rejects the assumption that the standard MOQA average-case solution applies for this random selection technique [63]. This lack of interchangeability is also true for the other random selection technique/randomised algorithm, as evidenced by Equation 5.1 not being equal to $\frac{10}{3}$.

## 5.3 Label Distribution

An important point that has not yet been explicitly discussed is the distribution of duplicate labels on a fixed po-structure. This would seem to be relevant information as "knowing a distribution on the inputs can help us to analyze the average-case behavior of an algorithm" [13].

**Notation 92.** *Let $D_{\beta_{max}}$ denote a discrete fixed po-structure with label ordering $\beta_{max}$ on it and let $N$ denote its size.*

**Notation 93.** *Let $K$ denote a finite non-empty label set.*

**Definition 58** (Label collection)**.** *A label collection on $D_{\beta_{max}}$ with $K$ is a label multiset of size $N$ whose elements are members of $K$.*

**Definition 59** (Unique label collection)**.** *A label collection on $D_{\beta_{max}}$ with $K$ is a unique label collection if all of its elements are distinct, hence $N \leq |K|$.*

**Definition 60** (Duplicate label collection)**.** *A label collection on $D_{\beta_{max}}$ with $K$ is a duplicate label collection if its elements are not all distinct.*

Figure 5.1 showed, for the label set $\{1, 2\}$ and the max-heap label ordering, all the distinct labelings of a discrete fixed po-structure of size three when the label value 2 is assigned twice. However, $\{1, 2, 2\}$ is just one of the duplicate label collections possible on this fixed po-structure with this label set. Instead of this collection, the label value 1 could have been assigned twice; $\{1, 1, 2\}$ would have resulted in the subsequent MOQA product function being deterministic hence returning to the exact average-case behaviour that MOQA normally investigates. The other possible duplicate label collections are $\{1, 1, 1\}$ and $\{2, 2, 2\}$. (The random selection technique used by the MOQA book [63] does not need to be concerned with how duplicate labels are distributed on a

| a | b | c |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 1 | 1 | 2 |
| 1 | 2 | 1 |
| 2 | 1 | 1 |
| 2 | 2 | 1 |
| 2 | 1 | 2 |
| 1 | 2 | 2 |

Table 5.1: $L(D_{\beta_{max}}, K)$ when $N = 3$ and $K = \{1, 2\}$

fixed po-structure because its solution to duplicate labels is to remove them, thereby side-stepping the problem but at an additional cost.) So average-case cost can be calculated over one duplicate label collection on a discrete fixed po-structure with some label set as happened for the example of Figure 5.1 onwards. It can also be calculated over all distinct label collections on a discrete fixed po-structure with some label set. (Attention is being given to the distribution of labels on a *discrete* fixed po-structure because, as discussed in Section 4.3, this is normally the type of structure that a MOQA′-satisfying program receives as input.)

**Notation 94.** *Let $L(D_{\beta_{max}}, K)$ denote the union of the set of all labelings for each distinct label collection possible on $D_{\beta_{max}}$ with $K$.*

As duplicate label collections can be among the label collections possible on $D_{\beta_{max}}$ with $K$, $L(D_{\beta_{max}}, K)$ has a cardinality of $|K|^N$ since the label of each node in $D_{\beta_{max}}$ can be chosen in $|K|$ ways. So $L(D_{\beta_{max}}, K)$ includes all the labelings of $D_{\beta_{max}}$ for each distinct duplicate label collection possible on $D_{\beta_{max}}$ with $K$. For example, when $N = 3$ and $K = \{1, 2\}$, then Table 5.1 shows all of the $|K|^N = 2^3 = 8$ labelings in $L(D_{\beta_{max}}, K)$. When $N \leq |K|$, $L(D_{\beta_{max}}, K)$ includes all the $N!$ labelings of $D_{\beta_{max}}$ for each distinct unique label collection possible on $D_{\beta_{max}}$ with $K$. For example, when $N = 3$ and $K = \{1, 2, 3\}$, then Table 5.2 shows all of the $|K|^N = 3^3 = 27$ labelings in $L(D_{\beta_{max}}, K)$.

Note that $L(D_{\beta_{max}}, K)$ is in stark contrast to $L(D_{\beta_{max}})$. Recall that the latter is the set of all canonically-ordered labelings on $D_{\beta_{max}}$ so there are no duplicate labels in any of these canonically-ordered labelings. In addition,

| a | b | c |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 1 | 1 | 2 |
| 1 | 2 | 1 |
| 2 | 1 | 1 |
| 1 | 1 | 3 |
| 1 | 3 | 1 |
| 3 | 1 | 1 |
| 2 | 2 | 1 |
| 2 | 1 | 2 |
| 1 | 2 | 2 |
| 2 | 2 | 3 |
| 2 | 3 | 2 |

| a | b | c |
|---|---|---|
| 3 | 2 | 2 |
| 3 | 3 | 1 |
| 3 | 1 | 3 |
| 1 | 3 | 3 |
| 3 | 3 | 2 |
| 3 | 2 | 3 |
| 2 | 3 | 3 |
| 1 | 2 | 3 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |
| 3 | 2 | 1 |

Table 5.2: $L(D_{\beta_{max}}, K)$ when $N = 3$ and $K = \{1, 2\}$

$L(D_{\beta_{max}})$ has a cardinality of $N!$ as opposed to $|K|^N$ because it just contains the $N!$ labelings of $D_{\beta_{max}}$ for a unique label collection possible on $D_{\beta_{max}}$.

Finally, in this work the uniform probability distribution is always defined on the sample space $L(D_{\beta_{max}}, K)$.

**Definition 61** (Uniform probability distribution (UPD))**.** *There is a uniform probability distribution on the finite or countably infinite sample space $S$ when every elementary event $s \in S$ has probability*

$$Pr\{s\} = \frac{1}{|S|}$$

*and, for any event $A$,*

$$Pr\{A\} = \sum_{s \in A} Pr\{s\},$$

*since elementary events, specifically those in $A$, are mutually exclusive [13].*

The uniform probability distribution on $S$ can be described as "picking an element of $S$ at random" [13].

## 5.4 Duplicate Labels in Insertion-sort

The average number of comparisons for either of Section 5.2's random selection examples is over *one* of the duplicate label collections possible on a discrete fixed po-structure of size three with label set {1, 2} rather than over *all* of the distinct label collections possible on that fixed po-structure with that label set. Hence, the argument could be made that the conclusions drawn from comparing these averages to the average number of comparisons for the standard MOQA product function example in that section are not definitive because the random selection averages were not over all distinct label collections. These averages also depend to a certain degree upon their implementations. Hence, it could also be argued that the implementations in Section 5.2 are particularly unwieldy — a claim the author would not try and dispute because no attempt was ever made to fine tune these implementations — and therefore, the discrepancy between these averages could be reduced. Perhaps even removed? In fact, certain works, see [67], [45] and [65], show that appropriately modifying insertion-sort and quicksort implementations can improve their efficiency/reduce the number of comparisons involved. Additionally, research by Wegner [77] shows that the average number of comparisons for various quicksort derivatives over distributions involving duplicate labels also varies.

There is no consideration given in the MOQA book [63] to the label collections over which the label-to-label comparison count should be averaged when duplicate labels are involved or to implementation details for duplicate labels beyond what is repeated in Section 5.2. So both of these arguments are speculative defences of Schellekens's concept [63] that there is a way to view the distribution of repeated input so that it does not prevent MOQA in its current form from correctly determining the average-case behaviour of a MOQA-satisfying program. This works's response to the above arguments and to this concept is that there is no technique, which does not significantly alter algorithm behaviour, that can guarantee that the average-case behaviour of an algorithm will be unaffected by how its input is distributed when the algorithm's behaviour is determined by its very input.

This response is demonstrated by 1), determining the average number of swaps for MOQA's insertion-sort on the input $D_{\beta_{max}}$ of size $N$ when all the possible labelings of $D_{\beta_{max}}$ are represented by $L(D_{\beta_{max}}, K)$ and then 2), com-

paring this average to the average number of swaps for MOQA's insertion-sort on the input $D_{\beta_{max}}$ of size $N$ when $L(D_{\beta_{max}})$ represents all the possible labelings of $D_{\beta_{max}}$; insertion-sort was chosen because it is one of the MOQA book's [63] few key application examples. First of all, this comparison between averages addresses the label collection issue raised by the first objection. Second of all, the average number of swaps, not comparisons, for MOQA's insertion-sort is of interest because while its average number of comparisons may be inherently linked to implementation detail, its average number of swaps is not. Therefore, if this comparison between averages shows that the average number of swaps for MOQA's insertion-sort depends upon the label collections under consideration, then it cannot be claimed that there is some implementation where the average number of comparisons have no such dependence. Hence, such evidence would overcome the second objection and show the fallacy of the MOQA book's [63] dismissal of distributions that involve duplicate labels.

**Notation 95.** *Let $T_{\beta_{max}}$ denote a linear fixed po-structure of size $n-1$, $n \geq 2$.*

**Notation 96.** *Let $e_{\beta_{max}}$ denote a discrete fixed po-structure of size one.*

In accomplishing the first aim, consider the MOQA product function that products $e_{\beta_{max}}$ below $T_{\beta_{max}}$ when these fixed po-structures are disjoint connected components of the same isolated subset. Assume that $T_{\beta_{max}}$ is the result of $n-2$ MOQA product functions on an initially discrete fixed po-structure of size $n-1$. Prior to these $n-2$ functions, let $L(D_{\beta_{max}}, K)$ represent all the possible, and thus equally likely, labelings over the $N$ nodes comprised from these $n-1$ discrete nodes, $e_{\beta_{max}}$ and the $N-n$ other discrete nodes in the same isolated subset, $2 \leq n \leq N$. After the $n-2$ MOQA product functions, whatever labels are on $T_{\beta_{max}}$ will be sorted in order according to the max-heap label ordering but once the subsequent MOQA product function connects $e_{\beta_{max}}$ below $T_{\beta_{max}}$, then the labeling on the linear fixed po-structure that is now of size $n$ will have to be examined and possibly adjusted. The expressions that immediately follow form Expression 5.9, which is the average number of swaps that it takes for the MOQA product function to push the label of the newly attached bottom node up to its correct position in this linear fixed po-structure of size $n$, when the average is taken over every labeling in $L(D_{\beta_{max}}, K)$ *after* the $n-2$ MOQA product functions that formed $T_{\beta_{max}}$ have been applied to each of these $|K|^N$ labelings. Note that the totals of Expressions 5.3 to 5.8

are also over every labeling in $L(D_{\beta_{max}}, K)$ *after* the $n-2$ MOQA product functions that formed $T_{\beta_{max}}$ have been applied to each of these $|K|^N$ labelings. Note also that $0^0$ is taken to be 1 in this section.

**Notation 97.** *Let $i$ denote the label value of $e_{\beta_{max}}$.*

Let $j$ denote the total number of nodes in $T_{\beta_{max}}$ that have the largest label in the set of labels on $T_{\beta_{max}}$, $1 \leq j \leq n-1$; these $j$ sequential nodes will be located from $T_{\beta_{max}}$'s top node downwards. Therefore, when $n > 1$ and $1 \leq j < n-1$, the total number of swaps that occur for all possible values of $i$ when $i$ is equal to the label values of the top $j$ sequential nodes in $T_{\beta_{max}}$ is:

$$(n-1-j).\sum_{i=1}^{|K|-1} i^{(n-1-j)}.|K|^{N-n}.\binom{n-1}{j}. \tag{5.3}$$

In other words, this expression is for the case when the label value of $e_{\beta_{max}}$ is equal to the label values of exactly $j$ sequential nodes in $T_{\beta_{max}}$, $j \geq 1$ and these $j$ nodes are located from $T_{\beta_{max}}$'s top node downwards though never reaching far enough to include its bottom node — as there are no swaps when $j = n-1$ it is safe to ignore the number of swaps for this value of $j$. If $j = 0$ in the above expression, then Expression 5.3 will also apply for another scenario. In this scenario, when $n > 1$ and $j = 0$, Expression 5.3 is the total number of swaps that occur for all possible values of $i$ when $i$ is greater than the label value of $T_{\beta_{max}}$'s top node. Summing these two scenarios together for all possible values of $j$ gives:

$$\sum_{j=0}^{n-2}(n-1-j).\sum_{i=1}^{|K|-1} i^{(n-1-j)}.|K|^{N-n}.\binom{n-1}{j}. \tag{5.4}$$

Next, let the node of rank $r$ in $T_{\beta_{max}}$ denote $T_{\beta_{max}}$'s $r^{\text{th}}$ node from the bottom upwards, $1 \leq r \leq n-1$. So, the node of rank one in $T_{\beta_{max}}$ is $T_{\beta_{max}}$'s bottom node, the node of rank two in $T_{\beta_{max}}$ is $T_{\beta_{max}}$'s second node from the bottom up and so on, until the node of rank $n-1$ in $T_{\beta_{max}}$ is $T_{\beta_{max}}$'s top node. Let $s$ denote the total number of nodes in $T_{\beta_{max}}$ whose label values are equal to $i$, $0 \leq s \leq n-1$; these $s$ nodes will be sequential in $T_{\beta_{max}}$. Therefore, when $n > 3$, $2 \leq r \leq n-2$ and $1 \leq s \leq n-1-r$, the total number of swaps that occur for all possible values of $i$ when $i$ is equal to the label values of $s$ nodes

of rank $r$ to $(r+s-1)$ in $T_{\beta_{max}}$ is:

$$(r-1).\sum_{i=1}^{|K|-2} i^{r-1}.(|K|-1-i)^{(n-r-s)}.|K|^{N-n}.\binom{n-1}{r-1+s}.\binom{r-1+s}{r-1}. \quad (5.5)$$

In other words, this expression is for the case when the label value of $e_{\beta_{max}}$ is equal to the label values of exactly $s$ sequential nodes in $T_{\beta_{max}}$, $s \geq 1$ and the ranks of these $s$ nodes never encompass $T_{\beta_{max}}$'s top or bottom nodes — Expression 5.3 is for when $i$ is equal to the label value of $T_{\beta_{max}}$'s top node and it has already been noted that there are no swaps when $i$ is equal to the label value of $T_{\beta_{max}}$'s bottom node. Hence, for all possible values of $r$ and $s$, Expression 5.5 becomes:

$$\sum_{r=1}^{n-3} r.\sum_{s=1}^{(n-2-r)}\sum_{i=1}^{(|K|-2)} i^{r}.(|K|-1-i)^{(n-r-1-s)}.|K|^{N-n}.\binom{n-1}{r+s}.\binom{r+s}{r}. \quad (5.6)$$

Finally, when $n > 2$ and $1 \leq r \leq n-2$, the total number of swaps that occur for all possible values of $i$ when $i$ is greater than the label value of $T_{\beta_{max}}$'s node of rank $r$ and less than the label value of $T_{\beta_{max}}$'s node of rank $r+1$ is:

$$r.\sum_{i=1}^{|K|-2} i^{r}.(|K|-1-i)^{n-r-1}.|K|^{N-n}.\binom{n-1}{r}. \quad (5.7)$$

Hence, for all possible values of $r$, Expression 5.7 becomes:

$$\sum_{r=1}^{n-2} r.\sum_{i=1}^{|K|-2} i^{r}.(|K|-1-i)^{n-r-1}.|K|^{N-n}.\binom{n-1}{r}. \quad (5.8)$$

Therefore, dividing the sum of Expressions 5.4, 5.6 and 5.8 by $|K|^N$ gives the average number of swaps that it takes for the MOQA product function to push the label of the newly attached bottom node $e_{\beta_{max}}$, which was selected from $N-n+1$ discrete nodes, up to its correct position in $T_{\beta_{max}}$, when the average is taken over every labeling in $L(D_{\beta_{max}}, K)$ after the $n-2$ MOQA product functions that formed $T_{\beta_{max}}$ have been applied to each of these $|K|^N$ labelings. By merging Expression 5.8 into Expression 5.6 and rearranging

Expression 5.4, this division is:

$$\left(\left(\sum_{r=1}^{n-2} r.\sum_{s=0}^{(n-2-r)}\sum_{i=1}^{(|K|-2)} i^r.(|K|-i-1)^{n-r-1-s}.|K|^{N-n}.\binom{n-1}{r+s}.\binom{r+s}{r}\right)\right.$$
$$\left.+\sum_{j=1}^{n-1} j.\sum_{i=1}^{|K|-1} i^j.|K|^{N-n}.\binom{n-1}{j}\right)/|K|^N. \quad (5.9)$$

**Notation 98.** *Let $A_{K,N,n}$ denote the numerator of Expression 5.9.*

Empirical evidence suggests that $A_{K,N,n}$ is equal to:

$$(n-1).|K|^{N-2}.\frac{(|K|-1).|K|}{2}.$$

This equivalence[3] has first been tested with a Python script for $|K| = [1,\ldots,8]$, for each value of $|K|$, $N = [1,\ldots,|K|^3]$ and, for each value of $N$, $n = [1,N]$; this is a total of $224,130$ individual tests. It has then been tested for larger values of $|K|$, e.g. for $|K| = 33$, $N = 4324$ and $n = 4223$.

As $A_{K,N,2} = |K|^{N-2}.\frac{(|K|-1).|K|}{2}$, Expression 5.9 can be rewritten as:

$$\frac{(n-1).A_{K,N,2}}{|K|^N}.$$

The solution to this section's initial aim can now be provided. Insertion-sort in *MOQA-Java* is the code of Figure 3.5. The average number of swaps for its first MOQA product function, which products $e_{\beta_{max}}$ below a $T_{\beta_{max}}$ of size one, is:

$$\frac{1.A_{K,N,2}}{|K|^N} = \left(\frac{1}{2} - \frac{1}{2.|K|}\right).$$

For $3 \leq y \leq n$ and $n = N$, the average number of swaps for each of its subsequent MOQA product functions, which product $e_{\beta_{max}}$ below the $T_{\beta_{max}}$ of size $y-1$, is:

$$\frac{(y-1).A_{K,N,2}}{|K|^N}.$$

[3]This equivalence was initially identified during the manual examination of sample datasets.

So the average number of swaps for MOQA's insertion-sort on the $D_{\beta_{max}}$ of size $N$ when $L(D_{\beta_{max}}, K)$ represents all the possible labelings of $D_{\beta_{max}}$ is the average of the sum of the average number of swaps for these $N-1$ MOQA product functions, which is:

$$\begin{aligned}\left(\frac{A_{K,N,2}}{|K|^N} + \sum_{y=3}^{N} \frac{(y-1).A_{K,N,2}}{|K|^N}\right)/(N-1) &= \\ \left(\sum_{y=1}^{N-1} \frac{y.A_{K,N,2}}{|K|^N}\right)/(N-1) &= \\ \frac{1}{4}.N.\left(1 - \frac{1}{|K|}\right). &\end{aligned} \tag{5.10}$$

Now attention can be turned to the second of the two aims laid out at the start of this section, which is the comparison of Expression 5.10 to the average number of swaps for MOQA's insertion-sort over all permutations of the one unique label collection possible on the $D_{\beta_{max}}$ of size $N$ when $|K| = N$. This set of $N!$ permutations is of course the only set of possible labelings on $D_{\beta_{max}}$ that is addressed by the MOQA theory.

As $K$ becomes larger and larger, the probability of selecting the same label more than once from $K$ approaches zero because the number of possible labels greatly exceeds the number of labels to be selected/sorted [66]. So, for very large values of $|K|$, $\frac{1}{|K|}$ in Expression 5.10 becomes $\frac{1}{\infty}$, which tends to zero. Hence, Expression 5.11 is the average number of swaps for MOQA's insertion-sort on the $D_{\beta_{max}}$ of size $N$ when $L(D_{\beta_{max}})$ represents all the possible labelings of $D_{\beta_{max}}$.

$$\frac{1}{4}.N \tag{5.11}$$

So, though Expression 5.10 asymptotically equals Expression 5.11, their difference in constants refutes any argument that the exact number of average swaps for MOQA's insertion-sort over $L(D_{\beta_{max}})$ and over $L(D_{\beta_{max}}, K)$ are the same. This further supports this work's rebuttal of the assertion by Schellekens [63] that the MOQA approach, in addition to determining the average-case behaviour of a MOQA-satisfying program over one unique label collection, is also able to determine a MOQA-satisfying program's average-case behaviour over distributions that involve duplicate labels.

## 5.5 Duplicate Labels in Quicksort

The conclusions being drawn in this chapter regarding the MOQA book's [63] assessment of duplicate labels are finally evidenced by contrasting quicksort's average-case behaviour over distinct labels to its average-case behaviour over duplicate labels using the MOQA measure for evaluating average-case behaviour, which is clearly defined to be the average number of label-to-label comparisons. Quicksort was chosen because it, along with quickselect, is also one of the MOQA book's [63] few key application examples.

This contrast for an array-based implementation of quicksort has already been detailed by Sedgewick [66][4]. The average-case behaviour of quicksort when all $N!$ labelings for a unique label collection on the $D_{\beta_{max}}$ of size $N$ are equally likely is:

$$2.(N+1).\left(H_{N+1} - \frac{4}{3}\right),$$

where $H_{N+1}$ denotes the $(N+1)^{\text{th}}$ harmonic number, $H_N \leq ln(N) + 1$ for all $N \geq 1$.

Continuing with $N \geq 2$, Sedgewick [66] then examines quicksort's average-case behaviour for two other distributions of input. The first is when all $N!$ labelings for a duplicate label collection on the $D_{\beta_{max}}$ of size $N$ are equally likely. For this situation, the average-case behaviour of quicksort for duplicate label collection $\{x_1 \cdot 1, \ldots, x_{|K|} \cdot |K|\}$ is at least:

$$N - |K| + 2. \sum_{1 \leq h < j \leq |K|} \frac{x_h.x_j}{x_h + \ldots + x_j},$$

where $x_i$ is how often label $i$ is duplicated, $x_1 + \ldots + x_{|K|} = N$ and $j$ is the pivot label. The upper bound given by Sedgewick [66] for the average-case behaviour of quicksort in this situation is also not in closed-form because it too has a dependence on the values of $x_1, \ldots, x_{|K|}$. This upper bound is:

$$2. \sum_{4 \leq h+3 \leq j \leq |K|} \frac{x_h.x_j}{1 + x_{h+1} + \ldots + x_{j-1}} + I(x_1, \ldots, x_{|K|}).$$

[4]Chapter 3 presents the *MOQA-Java* implementation and gives the extra overhead that quicksort in *MOQA-Java* carries in comparison to an array-based implementation. Hence, Sedgewick [66] considers a version of quicksort that is more efficient than *MOQA-Java*'s.

The $I(x_1, \ldots, x_{|K|})$ of this equation is defined as:

$$I(x_1, \ldots, x_{|K|}) = \sum_{1 \leq h \leq |K|-2} C(x_k, x_{k+1}, x_{k+2}) - \sum_{1 \leq h \leq |K|-3} C(x_{k+1}, x_{k+2}),$$

where $C(x_1, \ldots, x_{|K|})$ is the maximum number of comparisons needed on average to sort a permutation of the multiset $\{x_1 \cdot 1, \ldots, x_{|K|} \cdot |K|\}$.

The next distribution of input examined by Sedgewick [66] is when all $|K|^N$ labelings in $L(D_{\beta_{max}}, K)$ are equally likely to be on the $D_{\beta_{max}}$ of size $N$. For this situation, the average-case behaviour of quicksort is at least:

$$2.N.\left(1 + \frac{1}{|K|}\right).H_{|K|} - 3.(N + |K|)$$

or, for large values of $|K|$, at least:

$$2.(N+1).H_N - 4.N + 2.\left(\frac{N}{|K|}\right).(H_N - 1) + O\left(\frac{N^3}{|K|^2}\right).$$

The upper bound for the average-case behaviour of quicksort in this situation is:

$$2.N.\left(1 - \frac{1}{|K|}\right).H_{|K|} - 3.N + 2.\frac{N}{|K|} - 9.\left(\frac{N}{|K|}\right)^2 - 7.\frac{N}{|K|^2}$$

or, for large values of $|K|$, is:

$$2.N.(H_N + 1) - 2 + O\left(\frac{N^2}{|K|}\right).$$

So clearly duplicate labels effect the average-case behaviour of quicksort and, more specifically, the very model describing how repeated input is distributed will influence average-case behaviour. (Note that the quicksort derivatives in Wegner's research [77], which are over the $N!$ labelings for a duplicate label collection, aim for the above lower bounds established by Sedgewick [66]; a goal in which some are successful.) Hence, there can be no further doubt about the inaccuracy of Schellekens's statement [63] that no serious amendments to the current MOQA theory are required for it to correctly obtain the average-case behaviour of a MOQA-satisfying program when duplicate labels are involved. As the MOQA formulas for the average-case cost of its functions

have been crafted on the assumption that all canonically-ordered labelings of the specified isolated subset have equal likelihood of occurring, these formulas cannot subsume, and hence generate the correct answer for, any other distribution of labelings. Therefore, new formulas are required if MOQA is to ever precisely calculate the average-case cost of its functions for any other distribution of labelings on the specified isolated subset. Like their predecessors, i.e. those given in the MOQA book [63] and Chapter 4, these new formulas would have to be created by hand.

## 5.6 Chapter Summary

The MOQA theory was designed with distinct labels in mind so this chapter reasoned whether the current theory can also encompass duplicate labels. In comparison to the unique label distribution, it was shown that extra work, and accordingly cost, is generated on average by the MOQA product function for some of the distributions involving duplicate labels that are considered in this chapter; the focus was on the MOQA product function because it is the core MOQA function. So the MOQA book's [63] ungrounded assertion that its theory can correctly determine the average-case cost of each MOQA function for any distribution of input was plainly exposed to be erroneous. This enhancement was also supported by other quicksort-related research, which demonstrated how average cost fluctuates according to the distribution of labelings. The final point made in this chapter was that the creation of new formulas for the average-case cost of MOQA functions over a specific distribution involving duplicate labels would allow the MOQA static analysis tool to provide timing information for that distribution. Justifying which distributions to develop these new formulas for, along with their actual development, is left to future work.

# Chapter 6

# Literature Review

This chapter examines research in the field of automated average-case analysis and compares it to the MOQA principles. Such a detailed study highlights some of the contributions made by the MOQA theory. It also identifies areas in which the MOQA theory is comparatively weaker and thereby opens up these areas for improvement in the future.

This literature review is not limited to systems that only consider the average-case behaviour of a program. Tools that address other behaviours, such as best-case and worst-case, are also discussed here for the purpose of providing a more complete view of the range of techniques that can be employed when it comes to evaluating program complexity.

Concepts related to those found in MOQA's body of work are also explored in this chapter. As a thorough contextualisation of MOQA has not yet been carried out, it is hoped that this chapter may be the beginning of a valuable research contribution. It is recommended that prior to reading this chapter the reader has fully acquainted themselves with the MOQA theory in Chapter 2.

## 6.1 PL and EL

Cohen and Zuckerman [11] present one of the first works to attempt the automatic estimation of an algorithm's behaviour. To do so, Cohen and Zuckerman [11] proposed two languages and developed their processors.

The first of these languages is the programming language PL (Programming Language). PL is Algol 60 [55] with restrictions, a principle one being the

prohibition of recursive algorithms. The PL processor takes a PL program as input and then outputs that program's time-formula, which is a symbolic formula for the time it takes to execute the input program. The PL processor translates the input program into a time-formula through the syntactic rules that it has at its disposal. The most important of these syntactic rules are listed by Cohen and Zuckerman [11]. One of them is as follows:

$$if\ \ b\ then\ \ s \rightarrow (+\ \ @IFOH\ \ b\ \ (\#IFn\ \ s))$$

So there are $\#IFn$, $\#SIGMA$ and $\#WHILEn$ operators involved in the PL processor's translation of if, for and while statements, respectively; the number $n$ being the order in which the statement appears in the PL program. Operators such as $PROCDEC$ and $PROCCALL$ are used when the PL processor translates procedure declarations and calls. This syntax-directed translation of the input program also replaces operations such as $*$, $-$, assignment and variable declaration by their time-variables; hence, $*$ would be replaced by $@MULTI$, $-$ by $@SUBI$, assignment by $@ASSIGN$ and variable declaration by $@TYPED$. A *time-variable* denotes operation cost and can be replaced by an actual cost later on. So $@IFOH$ in the above syntactic rule example is the time-variable for the overhead of an if statement. All of the PL processor's operators and time-variables are listed by Cohen and Zuckerman [11].

The next language is EL (Evaluation Language), which is an interactive language of commands. These commands are inputted to, and then executed by, the EL processor. The EL commands enable a user to provide additional information about the time-formula produced by the PL processor. In other words, they enable a user to assist in solving the execution time of the PL input program. Two of these EL commands are:

**retrieve** <**file**> : retrieves the time-formula file <file> outputted by the PL processor.

**bind** <**variable**> <**number**> <**arithmetic expression**> : replaces each occurrence of variable <variable> in block number <number> with arithmetic expression <arithmetic expression>.

One application of the EL bind command is to replace a time-variable by a constant based on the specific compiler-machine architecture on which the input

PL program is to be executed. In total, there are twenty-two EL commands. Other EL commands allow the user to specify the probability of a conditional expression, the number of times a while statement is executed, and the type of execution behaviour that the time-formula is to be evaluated for, which can be actual, best-case or worst-case execution time[1]. Another set of EL commands plot actual/best-case/worst-case execution time graphs for the time-formula that the EL processor currently has to hand when there is only one variable in that time-formula. (A user can replace a time-formula's variables with constants via the appropriate EL commands and, when necessary, it is this mechanism that will reduce the number of variables in a time-formula to one.) So when the type of execution behaviour and a set of possible values for that remaining variable is user-specified, the EL processor will determine and then plot the time-formula's actual/best-case/worst-case execution time for each of these possible values. Note that this useful visual aid could be an interesting extension to a future MOQA static analysis tool. The most complex of the EL commands is the eval command, which attempts to further resolve the time-formula through the symbolic simplification techniques then available [18], [78].

The main distinction that is obvious between this research [11] and MOQA is the algorithmic behaviours that they examine. Cohen and Zuckerman [11] address three algorithmic behaviours, which are an algorithm's actual, best-case and worst-case execution time, and it appears that their estimation of these behaviours is fairly accurate based on the examples provided. MOQA addresses one algorithmic behaviour, which is the average number of label-to-label comparisons that take place within an algorithm's data structures. Yet, despite this important difference, the two works still have much in common.

PL is comparable to *MOQA-Java* in that it is the language in which programs to be statically analysed are written. Both PL and *MOQA-Java* drop some features from their parent languages, which are Algol 60 and Java, respectively, so that each language then matches the interpretive abilities of the tool that statically analyses programs written in it.

Like the PL processor, the current MOQA static analysis tool, *Distri-Track*,

[1]Heeded by Cohen and Zuckerman [11] is the fact, as is heeded by Schellekens [63], that neither best-case nor worst-case behaviour is IO-compositional and they state that this is something that an EL user will have to bear in mind when supplying time-formula information.

will parse its input program to provide a formula for the behaviour of that program. Interestingly, the formulas produced by *Distri-Track* [35] are similar in structure to those produced by the PL processor. However, the complexity of the techniques used by Hickey [35] in acquiring these formulas supersedes the complexity of those used by Cohen and Zuckerman [11]. *Distri-Track* needs to be more sophisticated than the PL processor because 1), of the type of behaviour it tracks, 2), it tracks data structure state, something never considered by Cohen and Zuckerman [11], and 3), the MOQA book [63] allows for recursive algorithms so *Distri-Track* must be able to tackle such algorithms, whereas the PL processor will never encounter recursive algorithms due to their exclusion from PL.

Chapter 4 details each data structure type that can have the average-case cost of a MOQA function that is applied to it determined by a MOQA static analysis tool; some of these types are new additions by this work. A MOQA static analysis tool should be initialised with the average-case equations for each of these data structure types and therefore, the average-case cost of a MOQA function applied to any one of these data structure types can be statically determined without any outside help. However, if *Distri-Track* is to determine the average-case behaviour of the entire *MOQA-Java* algorithm, then there are other tasks that it may need external assistance with. In fact, the MOQA book states that "in general, some user input is required to guide the analysis" [63]. Inductive po-classes for which *Distri-Track* cannot determine the average-case cost of MOQA functions is one area where external assistance is required. So a *Distri-Track* user can specify the average-case formula for a function applied to such an inductive po-class. Another area where external assistance is required is establishing that an algorithm's integer parameter is actually the size of the data structure to which one of the algorithm's MOQA functions is applied. This identification will result in that variable being part of the static average-case calculations, as it should be. For example, the mergesort algorithm in Figure 6.1 has the integer parameter noOfNodes. This variable is actually the size of the data structure to which mergesort's MOQA product function is applied. Therefore, it should be part of *Distri-Track*'s recurrence relation for mergesort but this knowledge would have to be supplied by a *Distri-Track* user. Furthermore, for any conditional expression other than a first-order or second-order conditional expression, *Distri-Track* will definitely

```
/**
 * Mergesorts the specified OrderedCollection.
 * @param oc a discrete OrderedCollection of size greater than
 * zero.
 */
public static <L extends Comparable<L>> CollectionConstruct
    mergesort(OrderedCollection<L> oc, int noOfNodes,
    Iterator<NodeInfo<L>> ocNodeInfos){
    if (noOfNodes == 1){
        return ocNodeInfos.next();
    }
    int mid = noOfNodes/2;

    return oc.product(mergesort(oc, mid, ocNodeInfos),
        mergesort(oc, noOfNodes - mid, ocNodeInfos));
}
```

Figure 6.1: Mergesort in *MOQA-Java*

require a user-specified probability if it is not to remain unknown.

So *Distri-Track* does not provide an interactive language like EL but it too allows for users to interact with it through Java annotations in the *MOQA-Java* code. A full list of the areas in which *Distri-Track* allows for user assistance, and the syntax of their Java annotations, is available [63]. Hence, *Distri-Track* is akin to EL and the PL and EL processor because Java annotations/EL commands can supplement the behavioural deductions made by *Distri-Track*/the PL and EL processor. Of course, any MOQA static analysis tool with this feature has the same limitation as the EL processor, which is "the quality of the results obtained by the EL user in manipulating time-formulas is directly proportional to his competence" [11].

In summary, both Cohen and Zuckerman [11] and Schellekens [63] follow related paths in their consideration of how to statically determine algorithmic behaviour though, as Cohen and Zuckerman [11] acknowledge, average-case behaviour requires far more ingenuity. So, overall, there is little question that MOQA is the more accomplished work of the two. However, one of the works that followed PL and EL attempted to move beyond reliance on user interaction and also included average-case behaviour in its automatic estimation of an algorithm's behaviour. Hence, PL and EL can be seen as a precursor of this system, which is called Metric and which is reviewed in the next section.

## 6.2 Metric

**Definition 62** (Repetitive algorithm)**.** *A repetitive algorithm relies on recursion and/or on at least one unbounded iterative statement.*

**Definition 63** (Non-repetitive algorithm)**.** *A non-repetitive algorithm relies only on straight-line code and/or on bounded iterative statements, i.e. it is an algorithm that is not repetitive.*

Metric [76] is a system that statically analyses Lisp programs and derives, by means of recurrence relations where necessary, closed-form expressions for their behaviour. Closed-form expressions are derived for the best-case, worst-case and average-case behaviour of repetitive and non-repetitive algorithms. For both these algorithm types, closed-form expressions are also derived for the variance of average-case behaviour. In comparison, the MOQA book [63] only derives closed-form expressions for the average-case behaviour of non-repetitive algorithms. When considering the average-case behaviour of non-repetitive algorithms, MOQA never goes further than recurrence relations.

Of the differences between Metric and MOQA, the one that has the greatest impact is the measure(s) (or metric(s)) that they use to calculate algorithmic behaviour. MOQA *always* uses the average number of comparisons that take place within an algorithm's data structures to calculate algorithmic behaviour. Therefore, the measure tracked by MOQA is an internal data structure characteristic.

**Definition 64** (Internal data structure characteristic)**.** *An internal data structure characteristic is a data structure property that depends on the actual values stored within the data structure.*

**Definition 65** (External data structure characteristic)**.** *An external data structure characteristic is a data structure property that is independent of the actual values stored within the data structure.*

Metric on the other hand is designed for a choice of measures, one of which is selected by the user for the algorithm that they wish to analyse. (There is obviously no such option for users of the MOQA static analysis tool because it is designed for one and only one measure.) Metric's analysis of the algorithm may involve measures that are in no way associated with the algorithm's data

structures but that all depends on the choice of user-specified measure. However, any Metric measure that is associated with an algorithm's data structure is an external data structure characteristic. Size is an example of an external list characteristic and is one of the Metric measures. At times, Metric may find it appropriate to analyse a portion of the algorithm for a measure other than the one specified. This is commonplace when the user-specified measure is time as the timing of many algorithms will be linked to their data structures' external characteristics, external characteristics such as list size.

Hence, the fact that the MOQA measure relies on the comparisons that take place within a data structure, i.e. relies on an *internal* data structure characteristic, means that its system deals with *labeled* data structures. Conversely, Metric will only ever deal with *unlabeled* data structures because the only data structure characteristics that it takes into account are *external*.

Metric data types are limited to lists, whereas MOQA data types are limited to series-parallel data structures. Structural equations are used by Wegbreit [76] to define each measure that Metric can track for a list. For example, the structural equations for the size of list $L$ are:

```
size(L)  =  if null(L) then 0
            else 1 + size(car(L)) + size(cdr(L))
```

In addition to these structural equations, Metric stores the time costs for a subset of Lisp's primitive functions; this subset includes the functions car, cdr and cons. Metric also stores the time costs of language overhead activities, such as function calls and references to variables and constants. These time costs can be either symbolic or explicit costs. In a similar vein, the MOQA static analysis tool stores the average-case costs for MOQA's "primitive" functions, which are the MOQA functions in Sections 2.2 and 3.3. These average-case costs can also be described as structural equations because they too are generally defined in terms of how the data is structured. Both Metric and the MOQA static analysis tool have their respective structural equations supplied to them, with Metric also having the time costs for the Lisp functions and language overheads supplied to it.

A simple example illustrates how the choice of measure can impact static analysis. The purpose of this example is twofold: to show how the measure

selected can influence the static analysis process and to emphasise how the selection of two different measures can result in two different average-case costs for the same algorithm. So, due to tracking different measures, two static analysis systems can produce two distinct results that are respectively *correct* when determining the average-case behaviour of the same algorithm. As such a disparity is shared by Metric and MOQA, it is useful to demonstrate it here while comparing and evaluating these two systems.

Consider the very common concept of adding an item to the start of a list. Let $L$ denote a list and let $n$ denote the size of $L$. What is the most basic algorithm that Metric can analyse which performs this task? In Lisp, the function cons adds an item to the start of a list. So Metric would analyse the one-line algorithm that, by means of cons, adds the algorithm's first parameter to the start of $L$, which is the algorithm's second parameter. Metric would find the behaviour of this algorithm to be fixed; the behaviour of an algorithm is fixed when the algorithm's best-case and worst-case behaviour, and hence the algorithm's average-case behaviour and its variance, are the same. So, if the user-specified Metric measure is time, then the fixed cost of this algorithm is the time to execute one cons. If the user-specified Metric measure is, say, list size, then the fixed cost is $n + 1$. Now consider the MOQA version of this algorithm. Its single parameter will be $L$ and the cons function will be replaced by the MOQA product function. (The MOQA version of this algorithm can keep the second parameter, which is the item to be added to $L$, when the new MOQA insert function in Section 3.3 is the replacement MOQA function.) However, in addition to adding a new item to the start of $L$, the MOQA product function may also have to reorganise the labeling now on $L$. (Recall that this reorganisation is required when, due to the label of the newly added node, the labeling now on $L$ is no longer in accord with the max-heap label ordering.) So the MOQA static analysis tool considers the average-case behaviour of this algorithm to be the average number of comparisons that it takes to push the new label into its correct position in $L$. The MOQA static analysis tool will use the following average-case formula for the MOQA product

function to deduce this average.

$$\begin{aligned}
\overline{T}_{prod}(\bullet_{\beta_{max}} \,||\, L_{\beta_{max}}) &= \frac{|\bullet_{\beta_{max}}|.|L_{\beta_{max}}|}{|\bullet_{\beta_{max}}|+|L_{\beta_{max}}|}.(\tau_{down}(L_{\beta_{max}})+\tau_{up}(\bullet_{\beta_{max}}))+ \\
&\quad (\frac{|\bullet_{\beta_{max}}|.|L_{\beta_{max}}|}{|\bullet_{\beta_{max}}|+|L_{\beta_{max}}|}+1).(|M(L_{\beta_{max}})|+ \\
&\quad |m(\bullet_{\beta_{max}})|-1) \\
&= \frac{1.n}{1+n}.(\tau_{down}(L_{\beta_{max}})+0)+(\frac{1.n}{1+n}+1).(1+1-1) \\
&= \frac{n}{1+n}.(\tau_{down}(\bullet_{\beta_{max}},\otimes,n))+\frac{n}{1+n}+1 \\
&= \frac{n}{1+n}.\frac{n-1}{2}+\frac{n}{1+n}+1 \\
&= \frac{n+2}{2}
\end{aligned}$$

The equations that produce $(n-1)/2$ for the above $\tau_{down}(\bullet_{\beta_{max}},\otimes,n)$ come from Equation 4.7 and the equations that it references when every *up* that occurs in them is replaced by *down*. So this simple example clearly demonstrates that the measure tracked by a static analysis tool has a significant impact on the result.

Metric performs up to three steps to produce a closed-form expression for algorithmic behaviour. In the first step, another algorithm is derived from the original. This algorithm computes the complexity of the original for the measure under consideration. A complexity reference in the transformed algorithm is replaced by a closed-form expression when 1), it is a measure of an algorithm other than the current algorithm or 2), it is a measure of the current algorithm but the measure differs from the one currently under consideration. These closed-form expressions are obtained by Metric recursively calling itself for the algorithm and measure in question. The logic behind implementing this step is quite detailed as examples demonstrate, see [76]. However, from this brief description, it is clear that Metric can never obtain closed-form expressions for mutually recursive algorithms. If the transformed algorithm has a complexity reference to itself for the measure under consideration after all possible closed-form substitutions have taken place, then and only then does Metric move onto the second step, which is recursion analysis. Commencing with the result from the first step, some of the main features of the second step include:

- determining which variables are irrelevant to the measure under consideration or are constant and therefore can be ignored,
- determining how the recursion variables change from one call to the next call,
- mapping the recursion variables that are lists onto integers by replacing the lists with some suitable abstraction, such as their size.

By the end of the second step Metric has a difference equation to describe the original algorithm; a difference equation being a specific type of recurrence relation. The third and final step that Metric undertakes is solving the difference equation for the four asymptotic behaviours. When it is determined by Metric that the best-case and worst-case behaviour differ, then the average-case behaviour and its variance are solved with generating functions. Metric even goes as far as manipulating and simplifying each closed-form expression to emphasise its dependence on the algorithm's parameter(s).

So both Metric, during its first step, and the MOQA static analysis tool get the performance of a non-repetitive algorithm through the composition of the algorithm's local costs. How recurrence relations for repetitive algorithms written in *MOQA-Java* are obtained by *Distri-Track* is described in Hickey's work [35]. While the techniques used by *Distri-Track* in determining these recurrence relations differ from those used by Metric, *Distri-Track* needs to deduce much of the knowledge that Metric does during its second step, in particular all of the above itemised points. However, some of this knowledge still has to be user-specified to *Distri-Track*, knowledge such as the first two itemised points above. (The data type of a MOQA recursion variable is not limited to lists, whereas in Metric it is, because of Lisp. Nonetheless, the current MOQA static analysis tool has only constructed recurrence relations for recursive algorithms whose recursion variables are discrete partial orders. So while it may be possible for a MOQA static analysis tool to correctly construct recurrence relations for recursive algorithms with other recursion variable types, this has yet to be demonstrated in practice. Therefore, it is still too early to categorically state that MOQA recursion variable range is broader than Metric recursion variable range.) A comparison between Metric's third step and the MOQA static analysis tool is not possible because the latter

halts once a recurrence relation for the algorithm's average-case behaviour is obtained.

It is worthwhile to underscore the difference between the recurrence relations produced by the two static analysis tools. The measure tracked by the MOQA static analysis tool during algorithm analysis is actually more than a data structure property. It is a *particular behaviour* of that data structure property, i.e. the *average* number of comparisons. This means that *DistriTrack*'s recurrence relations already reflect the average-case behaviour. The measures tracked by Metric during algorithm analysis are data structure properties. Metric needs to further analyse its recurrence relations if it is to produce a closed-form expression for each of the four asymptotic behaviours that it considers. Though this approach is more work-intensive than MOQA's, whose recurrence relations are already in the correct format, the benefit is a wider range of results. It is noted by Wegbreit [76], and by Schellekens [63], that obtaining the best-case or worst-case behaviour of an algorithm through the composition of local best-case or worst-case behaviours does not necessarily lead to the correct overall answer. For example, the worst-case behaviour of an algorithm derived in this manner would be incorrect if two algorithms that it relies upon have worst-case behaviours that can never occur during the same run-time. Rephrasing this in the MOQA book [63] terminology, best-case and worst-case behaviour is not guaranteed to be IO-compositional.

In conclusion, Metric [76] and MOQA [63] have a meaningful amount of overlap. They can both determine the average-case behaviour of certain algorithms with the measure(s) that they track and they both use recurrence relations when the algorithm is repetitive. However, Metric only considers unlabeled data structures and MOQA only considers labeled data structures. More specifically, Metric and MOQA track different algorithmic measures with Metric tracking external data structure characteristics, see Definition 65, and MOQA tracking an internal data structure characteristic, see Definition 64. In general, the difference between Metric's measures and MOQA's measure means that these static analysis tools will never both give the same average-case behaviour for the same algorithm. Hence, in general, each will provide an average-case solution that is accurate for its own measure/context but not for the other's; this is illustrated on page 192. Based on the examples presented by Wegbreit [76], this divergence between the two tools results in Metric being

capable of analysing more recursive algorithms than the current MOQA static analysis tool. This is mainly due to the measure that MOQA tracks and, as a consequence, the other restrictions that MOQA places on the algorithms that it analyses, one of the main ones being that every function in an algorithm is MOQA random structure preserving.

## 6.3 ACE

The system ACE (Automatic Complexity Evaluator) is the work of Le Métayer [51]. ACE statically evaluates the worst-case behaviour of programs written in the FP language [3]. Hence, an obvious and major difference between ACE and MOQA is ACE's consideration of worst-case, rather than average-case, behaviour. However, ACE is still of interest due to its method for statically determining worst-case behaviour, a method that relies on FP's powerful algebra of programs.

> "This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws and *are carried out in the same language in which programs are written* [emphasis added]. Combining forms are chosen not only for their programming power but also for the power of their associated algebraic laws. General theorems of the algebra give the detailed behaviour and termination conditions for large classes of programs" [3].

The fact that the FP language is such a mathematical model enables ACE to use FP axioms and recursive definitions to convert the input FP function into a non-recursive FP function that evaluates to the input function's worst-case behaviour. So, to complete its task, ACE relies fully on FP. This is clearly very different to MOQA, which provides mathematical formulas for the average-case cost of its functions but has to rely on standard static analysis techniques when analysing program construction; see [35] for details beyond those given in this work. Hence, it is worthwhile to consider the ACE system and what it accomplishes.

The ACE system can be divided into two parts. The first part derives a FP recursive function $Cf$ from the input FP recursive function $f$; $Cf$ evaluates to the worst-case complexity of $f$. This derivation is achieved by applying ACE's syntax-directed worst-case rules for a particular type of complexity to $f$, with the possible complexity types being time, length and size[2]. For example, one of the syntax-directed worst-case rules for time-complexity is:

$$T(E1 \rightarrow E2; E3) = E1 \rightarrow +o[T(E1), T(E2)]; +o[T(E1), T(E3)],$$

where

$$(E1 \rightarrow E2; E3) : x = \begin{cases} E2 : x & \text{if } E1 : x = T \\ E3 : x & \text{if } E1 : x = F \\ \perp & \text{otherwise} \end{cases},$$

$(f_1 o f_2) : x = f_1 : (f_2 : x)$ and $[f_1, \ldots, f_n] : x = \langle f_1 : x, \ldots, f_n : x \rangle$.

The second part attempts to transform $Cf$ into a non-recursive FP function. (Fait accompli if $Cf$ is already a non-recursive function due to $f$ being a non-recursive function.) The following sequence of steps describe the ACE system in its entirety, with steps 3 to 6 detailing this transformation process.

1. The function $Cf$ is derived from the function $f$.

2. The user-defined functions in $Cf$ are identified and ACE is recursively called for each one, i.e. ACE proceeds to Step 1 for each one. After which, ACE proceeds to Step 3.

3. Application of the recursion induction principle: the McCarthy recursion induction principle[3] is the theory behind this key transformation step, which tries to match $Cf$ to one of the patterns in ACE's library of FP recursive definitions. If a match is found, then the equivalent non-recursive FP function for that pattern is used to transform $Cf$ into a non-recursive FP function, whereupon ACE has successfully completed its analysis. If no match is found, then ACE proceeds to Step 4.

[2] $length\langle x_1, \ldots, x_n \rangle = n$ and $size\langle x_1, \ldots, x_n \rangle = /+ : \langle size : x_1, \ldots, size : x_n \rangle$, with $length : \varnothing = size : \varnothing = 0$.

[3] "The McCarthy recursion induction principle can be stated as follows [50]: 'two functions which verify the same recursive equation are equivalent over the domain of the function defined by this equation'" [51].

4. Factorisation: ACE attempts to put $Cf$ in the form $C\,o\,m$, where $m$ denotes a measure. If the current complexity is either length or size, then $m$ represents that complexity. Otherwise, $m$ represents a part of the argument modified in the recursive call. ACE next verifies whether the value selected for $m$ is actually correct and if it is, then ACE proceeds to Step 2; the verification technique is outlined by Le Métayer [51], along with its proof of correctness. If none of the values that can be selected for $m$ turn out to be correct, then ACE proceeds to Step 5.

5. Splitting: when $Cf$ is of the form:

$$Cf = P1 \rightarrow E1; P2 \rightarrow E2; \ldots Pn \rightarrow En; E'(Cf),$$

ACE splits it into $n$ equations:

$$\begin{aligned} Cf1 &= P1 \rightarrow E1; E'(Cf1) \\ &\vdots \\ Cfn &= Pn \rightarrow En; E'(Cfn). \end{aligned}$$

ACE is then recursively called for each of these $n$ equations. If $Cf1 = \ldots = Cfn$, then $Cf = Cfi$ and ACE proceeds to Step 2; the proof of correctness for this technique is outlined by Le Métayer [51]. Otherwise, ACE proceeds to Step 6.

6. Substitution: ACE tries to prove that $H = E(H)$ when $Cf$ is of the form:

$$Cf = p \rightarrow H; E(Cf).$$

The proof of correctness for this step is outlined by Le Métayer [51]. If this step succeeds, then $Cf = H$ and ACE has successfully completed its analysis. Otherwise, ACE has failed.

Each of the above steps will also use FP axioms to further simplify $Cf$.

So ACE clearly takes advantage of the mathematical properties associated with FP. (Note that these mathematical properties could be similarly applied to any other purely functional language, though the transformation process

would not be as smooth.) The wide variety of programs for which ACE can calculate worst-case behaviour include sorting programs, such as quicksort, insertion-sort and selection-sort, numerical programs, graph programs, search programs and a parser. Adding more FP axioms and recursive definitions to the ACE library would further increase its capabilities.

In comparison, MOQA is one level of abstraction above ACE because it is centred on the mathematical properties of its functions as opposed to being centred on the mathematical properties of the language in which these functions are defined. So the MOQA theory does not depend on the language in which MOQA programs are written; at most, the theory prohibits certain language features, such as while statements. While there may be some advantages to this independence — it would be reasonable to highlight such universality as an advantage, tethering the MOQA theory to a purely functional language would expose it to the powerful algebra that is an inherent attribute of languages in this class. Hence, future work should seriously consider this previously ignored path.

Le Métayer's research [51] also takes a step back and nicely breaks the general problem of automatically obtaining a recursive algorithm's asymptotic behaviour into two sub-problems. The first sub-problem is establishing the structural property that the algorithm's asymptotic behaviour depends upon and the second sub-problem is expressing this behaviour as a non-recursive equation. When working towards worst-case time, ACE's manipulation of $f$ may involve the structural properties length and/or size and their involvement is determined by $f$'s construction. By contrast, for average-case time, the MOQA book [63] addresses the first sub-problem by decreeing that the structural property is always the average number of label-to-label comparisons that take place within the algorithm's data structures. For the second sub-problem, the steps above show how ACE produces a non-recursive equation but this sub-problem falls outside of MOQA's purview. Once the MOQA static analysis tool determines a recursive algorithm's recurrence relation, it is then deemed the responsibility of some other tool, e.g. Mathematica [49], to further transform that recurrence relation. So, contemplating Le Métayer's analysis of the general problem domain leads to the conclusion that MOQA handles the first sub-problem by fixing the structural property and ignores the second sub-problem, which is a non-trivial problem, as Le Métayer states [51]. It is

a serious handicap for any static analysis tool in this domain, regardless of the specific asymptotic behaviour(s) it examines, to evade such an important part of its job. Hence, Le Métayer's reasoning supports the contention that a significant limitation of MOQA is that it halts too early on.

In comparing itself to Metric, which was the only other related system available at that time and is discussed in Section 6.2, Le Métayer's research [51] recognises that ACE has fewer hurdles to overcome because it can overlook certain issues that systems statically considering average-case behaviour cannot. For example, the probability of a conditional expression is acknowledged by Le Métayer as irrelevant when contemplating worst-case behaviour, yet it is a "tricky question" [51] whose answer is essential when establishing average-case behaviour. (Hence, the efforts of the MOQA book [63] to provide probabilities for a narrow range of conditional expressions.) So, in general, the worst-case behaviour of a program is easier to statically deduce than its average-case behaviour. Accordingly, it is very important to never lose sight of this fact when comparing any worst-case static analysis tool, such as ACE, to any average-case static analysis tool, such as MOQA. In other words, worst-case static analysis tools take less effort and can accomplish more.

## 6.4 COMPLEXA

COMPLEXA [81, 82] is an extension of Metric [76], which is reviewed in Section 6.2. Like Metric, COMPLEXA considers the best-case, worst-case and average-case behaviour of algorithms. It introduces data types other than lists which extends Metric to typed algorithms, other data types such as binary search trees. Each new data type is represented by a set of constructor terms. This set corresponds to the structural definition of an inductive poclass; see Section 4.4.2. In harmony with Metric, each external data structure characteristic that COMPLEXA tracks for a new data type is also defined by structural equations. Including data types other than lists leads COMPLEXA to generalise certain operations within Metric's three step process for producing a closed-form expression for an algorithm's behaviour. For example, COMPLEXA generalises the mapping of recursion variables onto integers to include the mapping of recursion variables that are not lists onto integers.

Qualified difference equations is one specific Metric area that greatly ben-

efits from the COMPLEXA extension. Here is a qualified difference equation example produced by Metric's second step, where $a_0$, $a_1$ and $a_2$ are constant values.

$$\begin{aligned} F(0) &= a_0 \\ F(n+1) &= \begin{cases} a_1 & \text{if } x = car(y) \\ a_2 + F(n) & \text{otherwise} \end{cases} \end{aligned}$$

As Metric does not know the probability of $x = car(y)$ for this qualified difference equation, it represents the unknown probability with a variable. How Metric solves this and other such difference equations for the various asymptotic behaviours is then detailed by Wegbreit [76]. COMPLEXA has a different approach to qualified difference equations. While COMPLEXA does not have any additional ability over Metric's to determine the probability of conditional expressions evaluating to true, Zimmermann [82] demonstrates that it is possible to extract non-qualified information regarding two qualified difference equations when they have the same conditional expression. This new COMPLEXA feature means that the asymptotic behaviours that it produces will have fewer variables than those produced by Metric for certain algorithms. Hence, COMPLEXA will be more precise in its timing information for these algorithms.

If COMPLEXA is to determine non-qualified information regarding two qualified difference equations, then they must have the following structures, where *cond* is some conditional expression.

$$\begin{aligned} a_0 &= c_0 \\ a_{n+1} &= \begin{cases} p_1(n+1) + c_1(n)a_n & \text{if } cond \\ p_2(n+1) + c_1(n)a_n & \text{otherwise} \end{cases} \\ b_0 &= d_0 \\ b_{n+1} &= \begin{cases} q_1(n+1) + d_1(n)b_n & \text{if } cond \\ q_2(n+1) + d_1(n)b_n & \text{otherwise} \end{cases} \end{aligned}$$

If, in addition to the above equation structures,

$$d_1(n)(p_1(n+1) - p_2(n+1))(q_2(n) - q_1(n)) =$$
$$c_1(n)(q_2(n+1) - q_1(n+1))(p_1(n) - p_2(n)),$$

then COMPLEXA can perform a dependency analysis between $a_{n+1}$ and $b_{n+1}$ that is independent of *cond*.

How is this new COMPLEXA feature integrated into Metric's system? As COMPLEXA examines dependency between recurrence relations, it needs to gather together the *recurrence relations* that form the composition of the algorithm it is analysing. However, recall that Metric's first step substitutes a closed-form expression for each complexity reference that is both a measure of an algorithm and meets the specifications outlined in Section 6.2. So Metric gathers together the *closed-form expressions* that form the composition of the algorithm it is analysing. Therefore, Metric's first step is modified in this regard as follows. In COMPLEXA, each complexity reference that is both a measure of an algorithm and meets the specifications outlined in Section 6.2 is replaced by its unsolved difference equation instead. This adjustment gathers together all the recurrence relations from which the algorithm being analysed is composed. If any two recurrence relations summed together in this composition meet the above requirements for COMPLEXA's dependency analysis, then COMPLEXA will calculate a replacement recurrence relation that represents their sum. Importantly, this new replacement recurrence relation will not be qualified, unlike the recurrence relations of which it is the sum. COMPLEXA will then solve the recurrence relations that now form the composition of the algorithm it is analysing.

The new COMPLEXA feature may be of further assistance when it is analysing a recursive algorithm with one or more parameters. Specifically, it will be of assistance when 1), two or more distinct code segments[4] determine the recursive algorithm's argument(s) and 2), the difference equations for the complexity of these distinct code segments meet the above requirements for COMPLEXA's dependency analysis; the complexity of each distinct code segment will rest on a measure and this measure is the one to which the recursive algorithm's arguments are mapped. In this situation, COMPLEXA will uncover a non-qualified dependency between recursion variables that are individually influenced by some qualification. The following illustration will

[4]So these code segments are not identical line-for-line though they may have lines in common.

help to clarify this further. Let $L$ denote a list and let $n$ denote the size of $L$. Let COMPLEXA analyse a recursive algorithm whose single argument is $L$, which is mapped to the measure length during COMPLEXA's analysis of the algorithm. Let this algorithm contain exactly two calls to itself. So the context of these two calls, i.e. the code segments that lead to these two calls, separately determine a value for the algorithm's argument. Now assume that these code segments comply with the two specific conditions just described here. In other words, for the measure length, the difference equations for the complexity of these two distinct code segments meet the above requirements for COMPLEXA's dependency analysis. Despite the presence of *cond* in both of these difference equations, COMPLEXA's dependency analysis can express the dependency between the two recursion variables without any reference to *cond*. So perhaps COMPLEXA concludes that the length of these two recursion variables added together is equal to $n - 1$. Therefore, COMPLEXA's new approach enables it to concisely express the relationship between recursion variables in terms of the relevant measure. Metric would not be able to extract such succinct information because it has no technique for analysing conditional expressions that are not directly related to external data structure characteristics.

COMPLEXA's dependency analysis between recursion variables reveals indeterminacy. For example, the dependency analysis between the two recursion variables in the last example showed that the length of each could be satisfied by more than one value. Hence, in step three COMPLEXA may have to solve a recurrence relation that does not fix the measure to which at least two recursion variables are mapped. If this is the case, then COMPLEXA factors in the indeterminacy to conclude the best-case and worst-case behaviour of the algorithm it is analysing. This is consistent with the Metric method. When it comes to determining the average-case behaviour, COMPLEXA makes the uniform distribution assumption regarding the indeterminacy. So, for the last example, COMPLEXA would assume that the $n+1$ distinct possible solutions to the dependency between the length of the two recursion variables are equally likely. It is noted by Wegbreit [76] that Metric does not consider indeterminacy so the introduction of COMPLEXA's dependency analysis means that COMPLEXA can reveal indeterminacy where Metric cannot and then resolve it.

So COMPLEXA allows for more data types than Metric does and this, along with its dependency analysis, means that COMPLEXA augments the set of algorithms that Metric can analyse. COMPLEXA's dependency analysis proves to be of particular use for divide-and-conquer algorithms with an intelligent conditional expression, e.g. the quicksort algorithm; a conditional expression is here deemed to be intelligent when establishing exactly how it affects program flow is beyond the abilities of the static analysis tool. While neither Metric nor COMPLEXA can figure out the probability of an intelligent conditional expression evaluating to true, COMPLEXA can produce asymptotic behaviours free of probability variables for a divide-and-conquer algorithm with an intelligent conditional expression. Metric cannot drop such probability variables. Accordingly, it can be said that COMPLEXA is a solid advancement on Metric.

Now to compare COMPLEXA to MOQA. Clearly, Section 6.2's comparisons between Metric and MOQA also hold for COMPLEXA because it extends Metric. However, the new features of COMPLEXA allow it to analyse algorithms that Metric cannot but MOQA can, algorithms such as quicksort and quickselect. COMPLEXA now matches the MOQA static analysis tool in that it is able to deal with a range of data types, though COMPLEXA is not restricted to series-parallel data types as the MOQA theory is. While COMPLEXA can on occasions ignore conditional expressions, and hence on these occasions the issue of their probability becomes moot, it, like Metric, can never calculate the probability of a conditional expression evaluating to true. In this regard, the MOQA static analysis tool is more powerful than both Metric and COMPLEXA because on occasions it can calculate this probability; more specifically, the MOQA static analysis tool can calculate the probability of certain conditional expressions that query data structure characteristics.

An interesting COMPLEXA consideration is the point at which the uniform distribution assumption comes into play. This assumption is applied at a relatively late stage in the process, when COMPLEXA has already constructed the difference equations. By contrast, the assumption of uniform distribution is completely embedded in the MOQA theory from the start. This would indicate that COMPLEXA has greater flexibility in its design than the design found in the MOQA book [63]. Other advantages COMPLEXA has over MOQA include statically determining the best-case and worst-case behaviour of algorithms in

addition to their average-case behaviour, not relying on user-interaction and not requiring the syntax of algorithms to be reformatted with new language constructs such as the MOQA product function.

Overall, it is this work's conclusion that COMPLEXA's aims are very close in nature to MOQA's.

A considerable COMPLEXA weakness is that its dependency analysis "is not decidable in the general case" [81]. So COMPLEXA's expansion of Metric is targeted at solving the complexity of algorithms that fall into a specific and narrow category, that is, divide-and-conquer algorithms. (As will be brought out later in this work, targeting a specific and narrow category of algorithms is also one of MOQA's limitations. However, the category of algorithms that MOQA is restricted to is not the same category that COMPLEXA is restricted to.) Hence, the Zimmermanns [81] point to another average-case static analysis system as offering a more general solution. This system is called LUO and is discussed in Section 6.5.

## 6.5 LUO

The LUO (Lambda–Upsilon–Omega, which is also denoted as $\Lambda\Upsilon\Omega$) system statically calculates the average-case behaviour of algorithms that are applied to combinatorial structures, i.e. countable discrete structures. Both the theory behind LUO and its implementation are thoroughly discussed, see [19], [22], [23] and [24].

LUO accepts combinatorial structure definitions and algorithms which are written in its own language Adl (Algorithm Description Language). "Adl is a language whose primitives correspond closely to the type structuring mechanisms" [23] and its expressions are Lisp-like. LUO converts the structural definitions of combinatorial structures into counting generating functions; the coefficient of the $n^{th}$ term giving the number of combinatorial structures of size $n$. For an algorithm that is applied to combinatorial structures, LUO converts the algorithm's structural specification into generating functions of average-case costs; the coefficients giving the average-case cost of the algorithm over random data of size $n$. These latter generating functions are known as *complexity descriptors*. After gathering together the algorithm's complexity descriptors and the relevant counting generating functions, the well-characterised an-

| **Unlabeled** | **Labeled** |
|---|---|
| union | union |
| cartesian product | partitional product |
| sequence | partitional sequence |
| cycle | partitional cycle |
| set | partitional set |
| multiset | |

Figure 6.2: The unlabeled and labeled operations from which LUO's unlabeled and labeled combinatorial structures are respectively defined

alytic properties that exist for such generating functions are then used by LUO to determine the asymptotic growth of their coefficients. Hence, more simply put, LUO determines an algorithm's average-case behaviour by applying advanced analysis techniques to the generating functions that it has associated with the input algorithm and its data structures.

The LUO theory tightly defines the combinatorial structures and language features for which it can automatically provide generating functions, and hence average-case behaviour. First, consider the combinatorial structures that LUO accepts. Typical examples include words, permutations, trees and graphs. These combinatorial structures can be either unlabeled or labeled. LUO admits an unlabeled combinatorial structure when it is defined in terms of the unlabeled operations listed in Figure 6.2. Likewise for a labeled combinatorial structure. The most basic component of an unlabeled structure is the *atom* primitive. This primitive represents a single element of size one, e.g. a letter or a node. Similarly, the most basic component of a labeled structure is the *latom* primitive. The LUO theory provides the set of rules that translate unlabeled combinatorial structures into ordinary generating functions and labeled combinatorial structures into exponential generating functions.

The key LUO attribute of any labeled structure of size $n \geq 1$ is that its $n$ latoms have distinct integer labels from 1 to $n$. However, the cartesian product of two such labeled structures will result in some ordered pairs with the same integer value for both components. So to avoid pairs of duplicate integers, which nullify the key LUO attribute of any labeled structure, partitional product replaces cartesian product when the structures involved are labeled. "The partitional product of labeled structures $U$ and $V$ consists of forming ordered

pairs $(u, v)$ from $U \times V$ and relabeling them in all possible [well-labeled] ways that preserve the order of the labels in $u$ and $v$"[74]. (In combinatorics, an object of size $n$ is *well-labeled* if each of its atoms has a distinct label from the set $\{1, 2, \ldots, n\}$. This concept is the same as Knuth's canonically-ordered labeling [42].) The number of possible relabelings is $\binom{|U|+|V|}{|V|}$. An example of partitional product follows later. Partitional product is a standard concept in combinatorial analysis and is fully detailed in various works, see [12], [27] and [30]. The other partitional operations in Figure 6.2 are partitional in the same sense and for the same reason. Note that the union operation definition ensures label value distinctness and that the nature of the multiset operation disqualifies it for labeled structures.

The structural definitions of LUO's combinatorial structures can be either iterative or recursive. To illustrate, the following is an iterative structural definition of all non-empty binary strings with alphabet $\{a, b\}$, which is an unlabeled combinatorial structure.

$$\begin{aligned} Word &= sequence(Letter); \\ Letter &= union(a, b); \\ a, b &= atom; \end{aligned}$$

Next, consider the language features accepted by LUO, i.e. consider Adl. All of its basic primitives are mechanisms for traversing through the combinatorial structures. One example is the *Test on Union* primitive:

$$\begin{aligned} A &= union(B, C) \\ P[a : A] &= \text{if } a \in B \text{ then } Q[a] \text{ else } R[a], \end{aligned}$$

where $P[a : A]$ denotes that the argument of procedure $P$ is $a$ and that the type of $a$ is $A$, and similarly, $Q[b : B]$ and $R[c : C]$. Another example is the *Component Iteration* primitive:

$$\begin{aligned} A &= cycle(C) \\ P[a : A] &= \text{forall } b \text{ in } a \text{ do } Q[b], \end{aligned}$$

where $Q[c : C]$. For more primitive details, see [23].

Specifically, the LUO system is composed of three parts:

1. Algebraic Analyser (ALAS): translates combinatorial structure definitions and their algorithms into counting generating functions and complexity descriptors respectively. ALAS does this by using formal translation rules that map the Adl language onto generating functions.

2. Solver: attempts to derive closed-form expressions for the generating functions outputted by ALAS.

3. Analytic Analyser (ANANAS): tries to extract average-case behaviour on the coefficients of the generating functions outputted by the Solver. Its manipulation and analysis of the generating functions relies upon "an extensive collection of routines" [24].

The LUO theory is taken from research in the fields of combinatorial analysis, applied mathematics and analytical number theory. So, it is clearly driven by strong mathematical techniques and therefore, the LUO engine can always be further strengthened as new results in these areas become available. In fact, a more modern version of LUO is now available as the combstruct package [21], which is part of the Algolib library [47]. (Some of the combstruct package's online examples model series and parallel circuits. Adapting the MOQA theory for this field is an active area of current research. Therefore, it may be appropriate for this continuing research to also examine LUO from the hardware angle.)

*The LUO system is the most successful average-case static analysis tool encountered by this work.* This is clearly demonstrated by the sheer range of algorithms for which it can deduce average-case behaviour. These algorithms can be loosely organised into three categories: 1), regular languages and finite automata, 2), context-free languages, terms and symbolic manipulation algorithms and 3), combinatorial problems. Many examples of algorithms in all three categories are given [23] and some of these examples are listed in Appendix B.

It is recognised in the MOQA book [63] that "the use and incorporation of generating functions in the MOQA context" should be investigated in the future and this research was responsible for commissioning a detailed study of LUO [14], which includes a helpful introduction to generating functions.

(However, the present work is the first to give proper attention to how the two systems compare.) Therefore, any future work that contemplates adding generating functions to the MOQA theory will have to seriously consider the following two questions. The first question is how would their addition impact on MOQA's originality? Techniques that produce recurrence relations for an algorithm's average-case behaviour are mentioned by Flajolet and Sedgewick [25] and they state that these "recurrence relations are either solved directly — whenever they are simple enough — or by means of ad hoc generating functions, introduced as a mere technical artifice." So, despite the fact that generating functions can be used to solve the recurrence relations produced by the MOQA static analysis tool and that this would supplement the MOQA theory without negatively effecting its originality, it is not an approach that incorporates generating functions *into* the MOQA theory. Besides, supplementing the MOQA theory in such a way is not innovative, as evidenced by the comment made by Flajolet and Sedgewick. On the other hand, if the MOQA theory is just replaced by the theory of generating functions, then there may be difficulty in distinguishing this new theory from LUO's. Hence, it is important that an original way of amalgamating generating functions with the MOQA theory is found. This prompts the second question. What advances would this amalgamation bring to LUO's area of expertise, whose intent is to automatically establish an algorithm's average-case behaviour via analytic combinatorics? This question is especially relevant since this is an ongoing area of research, see Section 6.6, that continues to be dominated by one of LUO's architects, Philippe Flajolet. For example, there is a recently published book on the topic of analytic combinatorics [25]. Therefore, any future contribution by MOQA to this field would have a lot of ground to cover to produce state of the art research.

Although LUO and MOQA take markedly different paths to computing the average-case behaviour of an algorithm, new vistas are opened up when their respective data structure operations are examined and contrasted, as the reader shall now see.

Product stands out when LUO's *labeled* operations are compared to the MOQA functions[5] because LUO's partitional product and MOQA's product

[5]LUO's unlabeled operations are not compared to the MOQA functions because they are separate universes. Section 6.2 discusses the disparity between unlabeled and labeled data

introduce the same relationship between the nodes of the two structures that they product together; note that this likeness hinges on the nodes, not on the labels of the nodes. In addition to this similarity, the number of relabelings performed by LUO's partitional product is equal to the factor by which MOQA's product increases structure multiplicity, i.e. $\binom{|U|+|V|}{|V|}$ when $U$ and $V$ denote the two structures being producted together. So it is worthwhile to further compare these two products.

Recall that MOQA's function products together two isolated components from the same isolated subset, which has the $\beta_{max}$ label ordering on it. In other words, the MOQA product function operates within one labeled structure. It then relabels the newly connected sub-structure so that the labeling on it is still in accord with $\beta_{max}$. This relabeling leads to the output of exactly one labeling. By contrast, LUO's function products together two unrelated structures and then relabels the newly connected structure to preserve the label order that was on each structure. So each structure's labeling is still in accord with the label ordering that was on that structure, whatever this label ordering is. This relabeling leads to the output of one or more labelings.

Though both functions safeguard the label ordering on each of the two structures that they product together, their methodology differs. The MOQA product function preserves the label orderings by simply insisting that the label ordering on both structures is max-heap (or alternatively min-heap) ordered. It then relabels to satisfy that particular label ordering. On the other hand, LUO's partitional product does not make any demands about the type of label ordering on either of the two structures. It just relabels to remove duplicate labels while ensuring that neither of the pre-product label orderings are disturbed.

Another crucial difference between the two products is that MOQA, unlike LUO, prevents its function from connecting two unrelated structures. The MOQA product function never connects two unrelated structures because the introduction of duplicate labels is disruptive to the current MOQA theory[6]. While duplicate labels have the same negative impact on the LUO theory, its partitional product can connect two unrelated structures because the prod-

structures.

[6]A simple modification to the MOQA theory is proposed in the MOQA book [63], with the claim that it can then handle duplicate labels. This proposal is shown to be ineffective in Chapter 5.




Figure 6.3: The data structures $U$ and $V$

| a | b | c | d |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 4 | 3 |
| 1 | 3 | 2 | 4 |
| 1 | 3 | 4 | 2 |
| 1 | 4 | 2 | 3 |
| 1 | 4 | 3 | 2 |
| 2 | 1 | 3 | 4 |
| 2 | 1 | 4 | 3 |
| 2 | 3 | 1 | 4 |
| 2 | 3 | 4 | 1 |
| 2 | 4 | 1 | 3 |
| 2 | 4 | 3 | 1 |

| a | b | c | d |
|---|---|---|---|
| 3 | 1 | 2 | 4 |
| 3 | 1 | 4 | 2 |
| 3 | 2 | 1 | 4 |
| 3 | 2 | 4 | 1 |
| 3 | 4 | 1 | 2 |
| 3 | 4 | 2 | 1 |
| 4 | 1 | 2 | 3 |
| 4 | 1 | 3 | 2 |
| 4 | 2 | 1 | 3 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 1 | 2 |
| 4 | 3 | 2 | 1 |

Table 6.1: $L(U_{\beta_{max}} + V_{\beta_{max}})$

uct's relabeling process removes all duplicate labels. The following example illustrates this distinction. When the structures $U$ and $V$ in Figure 6.3 are two isolated components from the same isolated subset, which has the $\beta_{max}$ label ordering on it, then Table 6.1 gives all the canonically-ordered labelings of $U$ and $V$. So clearly there is no risk of duplicate labels when the MOQA product function involves two isolated components from the same isolated subset. Now assume that the structures $U$ and $V$ in Figure 6.3 are unrelated and that both structures have the $\beta_{max}$ label ordering on them. In this case, Table 6.2 gives all the canonically-ordered labelings of $U$ and, likewise, of $V$. From these labelings, Figure 6.4 then depicts the four possible inputs for any product that involves this $U$ and $V$. While the duplicate labels in each possible input categorically rules it out as input for MOQA's product, each one is acceptable input for LUO's product. For example, Figure 6.4 goes on to depict the six relabelings that result when partitional product connects $U$ above $V$ for the leftmost of these four possible inputs.

So MOQA's product is less flexible than LUO's due to its constraint on

| a | b |
|---|---|
| 1 | 2 |
| 2 | 1 |

| c | d |
|---|---|
| 1 | 2 |
| 2 | 1 |

Table 6.2: $L(U_{\beta_{max}})$ and $L(V_{\beta_{max}})$

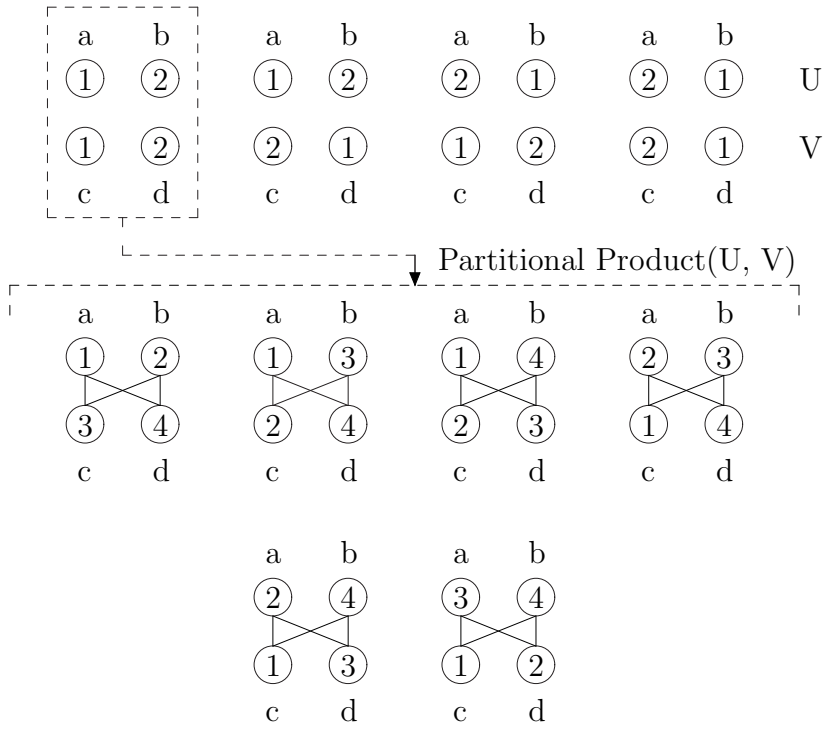



Figure 6.4: Four possible inputs for a product that involves $U$ and $V$ when they are unrelated structures and the relabelings that result when LUO's partitional product connects $U$ above $V$ for the leftmost input

label ordering type and its inability to work with duplicate labels. However, this is not unexpected as MOQA's theory is also less flexible than LUO's. This is evidenced by LUO being able to analyse a larger range of algorithms. The types of algorithms LUO can analyse are classified above and Appendix B gives fifteen examples. In comparison, MOQA can only analyse the class of algorithms whose average-case behaviour is asymptotically equivalent to the average number of *comparisons* that take place within the algorithm's data structures. The MOQA book [63] presents just four examples that MOQA is capable of analysing, which are listed in Section 2.4. The greater diversity and complexity of the algorithms that LUO can handle in comparison to MOQA is evidenced when comparing the fifteen LUO examples, which are not the full

set of LUO examples, against the four MOQA examples.

The last example can be used to illustrate the next point too. Take the twenty-four relabelings that result when LUO's partitional product that connects $U$ above $V$ is applied in turn to each of the four inputs depicted in Figure 6.4. Furthermore, apply the MOQA assumption regarding the uniform distribution of input and assume that the four inputs for this partitional product are equally likely. As a consequence, these twenty-four relabelings are also equally likely. Next, consider the MOQA product function at the point where it has just finished connecting $U$'s nodes above $V$'s for each of the labelings in Table 6.1 but before it has performed any push-ups or push-downs. The twenty-four equally likely labelings resulting from LUO's partitional product are equal to the twenty-four equally likely labelings that this MOQA product function currently has to hand. In other words, the twenty-four labelings that result from LUO's partitional product are equal to the labelings given in Table 6.1. So this example supports the following notable point: over all canonically-ordered input uniformly distributed, LUO's product, after it is applied to any two unrelated series-parallel structures that both have the $\beta_{max}$ label ordering on them, is equivalent to MOQA's product at the point just specified when MOQA's product is applied to the same structures, except that they now are isolated components of the same isolated subset. (The MOQA product function then takes the extra and final step of ensuring that the labeling over the newly-connected structure also satisfies $\beta_{max}$; this may require push-ups and push-downs.)

After identifying the correspondence that exists between these two products, this work was inspired to develop a new MOQA function. The new MOQA function is an enhanced version of the current MOQA product function, which is improved by the addition of partitional product's logic; the average-case formula for the current MOQA product function will have to be adjusted accordingly. So, when the two structures being producted together are isolated components from the same isolated subset, then the new MOQA product function behaves just as before. However, when the two structures being producted together are unrelated but otherwise adhere to the MOQA specifications, then the new MOQA product function follows partitional product's relabeling technique either before or after it introduces the new connections. It then performs the required push-ups and push-downs. This new MOQA

random structure preserving function now offers the best of both worlds because it can product within and across structures without any duplicate label complications, although the MOQA constraint on max-heap or min-heap label ordering still applies. Nonetheless, this is an important new function because it increases the range of a core MOQA function, which should certainly motivate its use in the future.

Now, observe that the LUO language is without deletion operations. This is because "no general method is known in order to analyse intrinsically *dynamic* algorithms that repeatedly modify a structure" [24]. LUO's discussion of this matter is referred to early on in the MOQA book [63], which states that "this led to the consideration of the redesign of standard data structuring operations and general novel language design to address the problem", i.e. led to MOQA. Yet, despite any suggestion that MOQA overcomes this LUO limitation, *MOQA does not supply a general method for analysing dynamic algorithms*. What is the basis for such a strong statement? Is there not a MOQA deletion function? Well, contemplate the MOQA deletion function. Like the other MOQA functions, its average-case formula when applied to a fixed po-structure is given in the MOQA book [63]. This formula correctly calculates the average-case cost of deleting a node from a fixed po-structure and the MOQA static analysis tool is able to keep account of each fixed po-structure that can result from this deletion because there is a finite number of them. However, this MOQA deletion function must be employed exclusively in algorithms that construct *specific fixed po-structures* because it has been designed for application to specific fixed po-structures; see Section 4.4.1 for an illustrative example that involves this MOQA deletion function. Algorithms that construct specific fixed po-structures have limited use and their average-case behaviour is a reasonably trivial affair because it is averaged over the few canonically-ordered labelings of their specific fixed po-structures. Hence, in addition to calculating such average-case behaviour statically, it is also easy to calculate it by hand or empirically. It is for such algorithms that the core MOQA theory [63] has been developed.

So data structure *classes* are not admitted in the domain of the above MOQA deletion function. No average-case static analysis tool calculates the average-case behaviour of an algorithm whose structures are defined by class by calculating the average-case behaviour of the algorithm for just one specific

fixed po-structure of that class. This would not yield the algorithm's average-case behaviour simply because it is not averaged over the general class. However, it is algorithms of this calibre that any serious average-case static analysis tool should be interested in timing. Accordingly, Hickey [35] and Chapter 4 extended Schellekens's average-case formulas [63] so that the MOQA static analysis tool can determine average-case behaviour when MOQA functions are applied to certain inductive po-classes. Although the new average-case formulas for the MOQA deletion function correctly calculate its cost for these inductive po-classes, there is now the issue of whether the MOQA static analysis tool is able to keep account of each fixed po-structure that can result from this deletion. To explain, for the inductive po-class $I$ whose set is infinite, let $Z$ denote the multiset of fixed po-structures of size $n-1$ that result after the MOQA deletion function is applied to each fixed po-structure of size $n$ in $I$'s set, $n \geq 1$. The next two properties are expected to hold true for $Z$: 1), each of $Z$'s fixed po-structures are equally likely and 2), $Z$'s set of fixed po-structures is equal to the set of fixed po-structures of size $n-1$ in $I$'s set. If at least one of these properties is not true for at least one value of $n$, then it is no longer possible for $I$ to represent the fixed po-structures that can result after the MOQA deletion function is applied to $I$. *So, in this case, the MOQA deletion function cannot be applied to $I$ because the MOQA static analysis tool has no means of representing the fixed po-structures that can result.* (This deficiency is acceptable only when this MOQA deletion function is the very last under analysis, as the infinite number of fixed po-structures that can result is then irrelevant.) For that reason, the MOQA static analysis tool needs to be informed by a user when it is safe for an inductive po-class to have the MOQA deletion function applied to it. Then, and only then, can the MOQA deletion function be applied to that inductive po-class.

Clearly, there is no general MOQA tactic for analysing dynamic algorithms because Schellekens's MOQA deletion function [63] only works when applied to specific fixed po-structures and the new MOQA deletion functions given by Hickey [35] and Chapter 4 only work when applied to some strict subset of the inductive po-classes that Chapter 4 permits, e.g. the discrete inductive po-class is in this strict subset. These are tailor-made solutions and therefore, they can be firmly rejected as a "general method". This is emphasised by the fact that the lack of any general approach for analysing dynamic algorithms leaves

LUO "helpless" [24] when it comes to typical algorithms such as heapsort and balanced trees. MOQA is just as helpless for these algorithms too[7] and so, has made no advances with regard to this key "bottleneck" [24].

As well as struggling with the static analysis of dynamic algorithms, LUO and MOQA share two other common traits. One of these traits is that writing a program that LUO or MOQA can statically analyse requires a solid understanding of their respective theories. It should be apparent at this point that LUO is designed for programs whose style is functional and MOQA is designed for programs whose style is object-oriented. However, once a programmer is comfortable with these language styles, writing programs in either syntax will not be challenging. It will require effort though to gain a knowledge of the mathematical concepts that LUO/MOQA rely on so as to write programs for which an average-case solution can then be automatically determined by LUO/MOQA. In other words, it is necessary to grasp what it is that makes LUO/MOQA capable of handling particular algorithms and then apply these principles when it comes to writing new programs for their analysis. This is not a simple task. Once a programmer has mastered the LUO/MOQA theory, LUO/MOQA should produce an average-case solution that is asymptotically correct for the specified program. There can be confidence in the average-case solution produced because 1) both LUO and MOQA have verified the results that they have already produced by comparing these results against the literature and, more importantly, 2) both LUO and MOQA have established theory reliability via proofs. So, the asymptotic accuracy of their results is another trait that is shared by LUO and MOQA.

In summary, LUO's recursively defined labeled combinatorial structures are akin to MOQA's inductive po-classes in that both use their admissible operations when recursively defining a class of data structures; LUO's admissible operations for labeled combinatorial structures are those listed in the second column of Figure 6.2 and MOQA's admissible operations are the functions described in Sections 2.2 and 3.3. However, the mathematical foundation on

[7]It is also accepted by the MOQA book [63] that it fails to statically determine the average-case cost of its own version of heapsort, which is called percolating heapsort. The treapsort algorithm is another of the MOQA algorithms, see Appendix A, and is presented alongside heapsort; it takes advantage of the MOQA top function from this work. The MOQA static analysis tool does succeed at statically determining the average-case cost of the treapsort algorithm but, because it has a worst-case time of $O(n^2)$, it is not actually a genuine variant of heapsort.

which LUO rests gives "results of sweeping generality" [25] as indicated by the variety of algorithms referred to in Appendix B, whereas it is a struggle to find useful algorithms that the MOQA static analysis tool can analyse in addition to the few sorting algorithms outlined in the MOQA book [63] and itemised in Section 2.4. In fact, Flajolet, Salvy and Zimmermann [24] succinctly gets to the heart of the matter after introducing the LUO framework:

> "This specification of a precise mathematical level of expertise also ensures that our 'automatic theorems' actually represent automatic results (and not a haphazard collection of ad hoc recipes put into a large programme!)."

As the MOQA theory has been customised for very specific algorithms and situations, it seems fair to state that the MOQA system is a closer match to the above "haphazard collection of ad hoc recipes" than to a " specification of a precise mathematical level of expertise". For example, the MOQA split function was developed for the quicksort and quickselect algorithms, the MOQA top, bot and lift functions were developed for the treapsort algorithm and the MOQA deletion function can only be applied to the small group of structures defined above. An unavoidable consequence of this difference in aptitude is that MOQA is outperformed by LUO when it comes to the number of algorithms for which it can derive average-case behaviour.

## 6.6 Mishna

Mishna [52] presents a technique for advancing the LUO system; LUO is surveyed in the previous section. This technique relies upon attribute grammars and a specification for their conversion into generating functions. (Attribute grammars were originally conceived by Knuth [41].) Mishna [52] uses attribute grammars because, as she points out, "an attribute describes the number of steps (however that is defined) an algorithm requires when a given structure is input". Therefore, Mishna pairs an attribute grammar definition with the structural definition of a LUO combinatorial structure, with the attribute grammar definition adhering to the form given by the structural definition. Take, for example, the following structural definition of a LUO combinatorial

structure:

$$\begin{aligned} T &= \epsilon \\ T &= product(atom, set(T)) \end{aligned}$$

This is the structural definition of an unlabeled tree. One of Mishna's examples then pairs this structural definition with the following attribute grammar definition for the internal pathlength of such a tree:

$$\begin{aligned} ipl(T) &= 0 \\ ipl(T) &= set(ipl(T)) + size(T) - 1 \end{aligned}$$

So attribute grammars are used by Mishna's research [52] to describe LUO combinatorial structure properties because this type of knowledge is often helpful when establishing an algorithm's average-case behaviour.

While Mishna notes that it is not unusual to describe algorithms with attribute grammars[8], it is the relationship that exists between attribute grammars and generating functions that makes these grammars so amenable for integration into LUO; this relationship has been established by earlier works, e.g. see [15]. (Recall that generating functions are a key part of the LUO theory because they reveal information about averages.) A noteworthy feature of the generating functions introduced by attribute grammars is that they are multivariate whereas the generating functions in LUO are univariate. So, this additional characteristic enables Mishna's work [52] to determine the average-case behaviour of certain algorithms for which LUO cannot. Mishna's work [52] also formally proves that it can determine the average-case behaviour of any algorithm for which LUO can. For example, both LUO and Mishna's extension can analyse quicksort but Mishna's extension can also handle quickselect.

Finally, Mishna [52] adds a new operation to those listed in Figure 6.2 in Section 6.5. This operation is known as the box or min label operator and it assigns the minimum label to a component. It was originally developed by Green [31], who also provided the generating function relationship. This new operation enables Mishna's extension to model increasing trees; an *increasing tree* is a tree of size $n$ labeled by distinct integers from the set $\{1, 2, \ldots, n\}$

[8]This point is demonstrated by Metric's use of structural equations; see Section 6.2.

with the restriction that the label on any node is greater than the label on its parent. Therefore, Mishna's structural definition of an increasing binary tree $I$ is:

$$\begin{aligned} I &= \epsilon \\ I &= product(min(atom), I, I) \end{aligned}$$

This min label operation is of particular interest because it would make it possible to now model in the LUO syntax a MOQA inductive po-class that has the min-heap label ordering on it. If the corresponding max label operation and generating function relationship was also added to the LUO system, then it would become possible to model in the LUO syntax a MOQA inductive po-class that has the max-heap label ordering on it. Consequently, the MOQA inductive po-classes discussed in Chapter 4 can have their structural definitions readily translated into the LUO syntax. (Doing a similar translation for MOQA's fixed po-structures is rather lacking in purpose for the reason given on page 214 in Section 6.5.) Hence, for future work, it may be of interest to obtain the enhanced LUO implementation, i.e. the LUO system with Mishna's additions, to see if it can successfully analyse insertion-sort, mergesort and treapsort; these are the three other algorithms, in addition to quicksort, that the MOQA book [63] considers. If this analysis was favourable and then supplemented with a formal MOQA-to-enhanced-LUO translation or simulation, then a proof could be given for whether or not this LUO extension has the ability to determine the average-case behaviour of any algorithm for which the MOQA static analysis tool can determine average-case behaviour.

In summary, the work of Mishna [52] strengthens the potency and potential of LUO and Mishna's capacity to model increasing trees is closer to how MOQA represents its data structures.

## 6.7 Sarkar

Research by Sarkar and Hennessy [62] and solely Sarkar [61] considers the automatic partitioning of a program for the purpose of scheduling it over multiple processors, with an estimation of program behaviour determining how it is carved up. This led to Sarkar presenting a framework for determining

the average-case behaviour of a program and its variance [60]. The central tenet of this latter research by Sarkar is that such information can be extracted from intelligently monitoring one or more executions of the program. *So the average-case behaviour of a program and its variance, as determined in Sarkar's work [60], is extrapolated from run-time information.*

The first step Sarkar [60] takes towards his goal is to construct a control flow graph for the program under analysis; a program's control flow graph is defined in Section 4.2. (Note that this is also *Distri-Track*'s first step.) The loop cycles in this control flow graph are next identified; the loop cycles in a control flow graph are also known as the *interval structure* of the control flow graph. The control flow graph is then extended according to Sarkar's specifications [60]; most of the new nodes and edges added to the control flow graph further emphasise its interval structure. This extended control flow graph is finally converted into a *forward control dependence graph* by ignoring all of its back edges. The control flow graph extension means that valuable loop cycle information is not lost when back edges are disregarded.

Sarkar [60] next adds counter variables to the compiled program code to track the frequency with which segments of it are executed. These counter variables are incremented during program execution and are stored in a program database at the end of *each* program execution. The program's forward control dependence graph is used to locate the counter variables in the compiled program code and this is a more sophisticated approach, which results in it being more efficient than simply having a counter variable for each node in the program's control flow graph. For example, this advancement assigns a counter variable per control condition, thereby eliminating the duplicate counter variables that would arise when there is a counter variable for each node in the program's control flow graph and multiple nodes depend alike on a single control condition. So it is the storing of this variable information that enables the average execution frequency of each node in the program's forward control dependence graph to be estimated. Of course, the average execution frequency of each node should become more and more accurate when averaged over more and more distinct program executions. The average execution frequency of each node, in conjunction with its cost, is then used to estimate the total average-case behaviour of the program. Variance, which arises from conditional branching and involves formulas of greater complexity than those

given for average-case behaviour, is also estimated in Sarkar's framework [60].

So Sarkar [60] collects run-time information about a program and then uses average-case and variance formulas to interpret this data. The result is an execution profile of the program. This is substantially different to any of the other methods that have been considered up until now because none of these methods required actual program execution. Therefore, Sarkar's technique [60], which *estimates* a program's average-case behaviour and its variance through the profiling of program execution, is quite dissimilar to that of the MOQA book [63], which *calculates* a program's average-case behaviour solely through *static* techniques; recall that a program's average-case behaviour according to Sarkar relies on the frequency with which program statements are executed whereas a program's average-case behaviour according to MOQA relies on the number of comparisons that take place within the program's data structures. Hence, it is reasonable to state that these two bodies of work have little theoretical overlap.

How useful is Sarkar's approach? He reckons that *statically* calculating conditional branch probabilities and the number of loop cycles is feasible for "only a few restricted cases" [60], of which some examples are given. It turns out that Sarkar's examples are a generalisation of when statically calculating conditional branch probabilities and the number of loop cycles is deemed feasible in the MOQA book [63] and in Hickey's research [35]; it is feasible in these for certain first-order and second-order conditional expressions and bounded loop cycles. So Sarkar advocates analysing a program with his execution profile technique when such static calculations are not feasible for part of the program in question. (Note that Sarkar's work [60] does takes advantage of bounded loop cycles. The counter variable for a bounded loop cycle is simply set to the number of loop iterations instead of being incrementing by one for each loop iteration.) Hence, Sarkar provides a way of estimating a program's average-case behaviour and its variance when there is no other means of doing so. Therefore, Sarkar's approach [60] to average-case analysis allows it to analyse a wide variety of programs because, after identifying suitable places of observation within the program, it monitors program behaviour at run-time and then analyses the results.

## 6.8 Other Related Research

A selection of systems that analyse program behaviour has just been carefully considered. Each of these systems is scrutinised because it is among those closest in nature to MOQA and/or it successfully determines average-case behaviour. For example, Sarkar [60] successfully determines average-case behaviour, although its observational approach is quite different to that of the others. However, these are not the only systems to analyse program behaviour and so this section will give a synopsis of those that remain.

Ramshaw [59] presents a formal frequency system that firstly, calculates a program's average-case time-formula and then secondly, evaluates that time-formula, which is in the form of a difference equation. This frequency system rests on the theory used in program correctness verification systems and in the determination of loop invariants. Specifically, it rests on the work of Floyd [26] and Hoare [38], although Ramshaw's complexity assertions differ from theirs. The complexity assertions of Floyd [26] and Hoare [38] describe properties satisfied by program variable values at certain points in the program, whereas Ramshaw's complexity assertions [59] describe properties satisfied by the *distribution* of program variable values at certain points in the program. Ramshaw's work [59] is also based on Kozen's denotational semantics for probabilistic programs [46]. Kozen [46] considers programs as linear operators on Banach spaces of measures, with Banach spaces being a well-used mathematical technique in functional analysis. While Ramshaw [59] does supply programs that his system is capable of analysing, there is no implementation of this system. This is a later accomplishment of Hickey and Cohen [36], for the FP language. However, despite Hickey and Cohen's successful implementation and analysis of some non-trivial programs, the subsequent opinion of one of its authors is that "unfortunately, Ramshaw's approach is formally enticing but practically ineffectual" [9]. One reason given for this opinion is that Ramshaw's approach struggles with programs containing arrays.

Hickey and Cohen [36] deal with recurrence relations, as this work realises Ramshaw's [59], but they state that "the problem of solving the resulting equations may be complex". MOQA faces this problem too because the MOQA static analysis tool also produces the same type of equation, which reaffirms the fact that the MOQA book's [63] avoidance of this complex issue is an

unfortunate deficiency; this point was initially made in Section 6.3.

Certain works divide the analytical analysis of algorithms into two categories: macroanalysis and microanalysis. One such work by Cohen [10] defines these categories as follows:

> "The macroanalysis of algorithms consists of choosing a dominant operation of an algorithm and expressing execution time as a function of the number of times this operation is used. In contrast, the microanalysis of programs consists of expressing the execution time as a function of the time needed to execute *each* of the operations in the program."

Cohen [10] goes on to summarise the field of microanalysis. Some of these works have been examined here in Section 6.2, Section 6.3 and, in the case of Ramshaw, in the preceding paragraphs. The other relevant works reviewed by Cohen [10] are Ramamoorthy [58] and Beizer [5]. Both of these systems use a discrete Markov model to statically determine average-case behaviour. So, if Ramamoorthy [58] has 1), a program's control flow graph, 2), the constant probability of taking each branch in that control flow graph and 3), the execution time of each basic block in that control flow graph, then he can calculate the program's average-case behaviour and its variance. The work of Beizer [5] is similar to that of Ramamoorthy [58]; the main difference between the two is that the latter gears itself towards programs that perform multiplication at the hardware level.

The worst-case static analysis tool ACE [51], see Section 6.3, is the only system of those surveyed up until now which does not consider average-case behaviour. It is fitting that each of these other systems study average-case behaviour because such behaviour is at the heart of the system which underpins this work, i.e. MOQA [63]. An additional reason for the focus on average-case static analysis tools is that average-case behaviour is generally far more challenging to resolve than worst-case behaviour, as concluded in Section 6.3. Therefore, worst-case static analysis tools frequently rely on theoretical concepts that would be lacking in information if average-case behaviour was sought instead, which means that only a limited amount of their theory would be suitable for average-case analysis. (This reasoning also holds for any best-case static analysis tool.) However, as the implementor of the current

MOQA static analysis tool *Distri-Track*, Hickey [35] does discuss worst-case static analysis tools like Gustafsson, Lisper, Sandberg and Bermudo [32], Liu and Gomez [48] and Puschner and Schedl [57] because average-case and worst-case static analysis tools often have mechanical details in common, such as program transformation techniques. Nonetheless, as this work is primarily interested in establishing and extending the MOQA theory and not in the finer details of the system for its delivery, as Hickey [35] is, ACE [51] is the only static analysis tool designed exclusively for worst-case analysis that merited close attention here.

There is one final MOQA attribute left to explore. This is the MOQA random structure preserving feature of any MOQA function and the following section examines how it relates to existing research.

## 6.9 Randomness Preservation

The MOQA functions are MOQA random structure preserving and this attribute is an important factor in MOQA's success. Knuth also examines functions that preserve data structure randomness because he too finds the average-case analysis of such functions easier. Though the aim of both Knuth and Schellekens is to simplify average-case analysis for certain problems, the meaning behind Knuth's preservation of randomness [42] differs from the meaning found in the MOQA book [63].

In exploring this difference, the following definitions will be useful. Let $\alpha$ denote a data structure family and the label ordering on that data structure family. For example, let $\alpha_{bst}$ denote the binary tree family and the label ordering that requires the label of a parent node to be greater than the label of its left child and smaller than the label of its right child[9]. Let $I_\alpha(x)$ denote the $\alpha$ insertion function that inserts label $x$. Let $D_\alpha(x)$ denote the $\alpha$ deletion function that deletes label $x$. Let $S_\alpha(x_1, x_2, \ldots, x_n)$ denote the data structure of type $\alpha$ and size $n$ that is constructed from the sequence $I_\alpha(x_1), I_\alpha(x_2), \ldots, I_\alpha(x_n)$ when applied to an initially empty data structure. So $S_{\alpha_{bst}}(1, 3, 2)$ is binary tree I in Figure 6.5. Note that $S_\alpha$ can represent the same data structure for two or more distinct label insertion permutations of the same length. To illustrate, $S_{\alpha_{bst}}(2, 1, 3)$ and $S_{\alpha_{bst}}(2, 3, 1)$ both represent binary tree III in Figure 6.5.

[9]This label ordering assumes that labels are distinct.

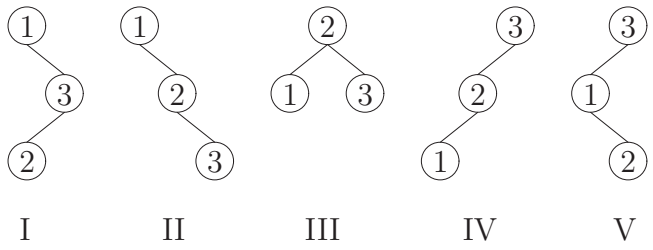



Figure 6.5: The five distinct BSTs of size three with labels

Let $X_n$ denote the set of $n!$ distinct permutations of the set $\{1, 2, \ldots, n\}$ when $n \geq 1$ and let $X_0 = \{\}$. Let $S_{X_n,\alpha}$ denote the multiset of data structures that is $S_\alpha(p)$ for each permutation $p$ in $X_n$, i.e. $S_{X_n,\alpha} = \sum_{p \in X_n} S_\alpha(p)$, $n \geq 0$. ($S_{X_n,\alpha}$ is a multiset as opposed to a set because two or more of the $X_n$ label insertion permutations for $S_\alpha$ may result in the same data structure.) For example, $S_{X_3,\alpha_{bst}}$ is the multiset of data structures in Figure 6.5 when binary tree III has a multiplicity of two and the others have a multiplicity of one. Let $A(S_{X_n,\alpha})$ denote the set of elements in $S_{X_n,\alpha}$, i.e. multiple repetitions of the same element in $S_{X_n,\alpha}$ are reduced to one membership in $A(S_{X_n,\alpha})$.

Both Knuth and Schellekens assume that only distinct labels are inserted into a data structure and that all insertion permutations of these labels are equally likely though the two differ in how this is actually accomplished. Knuth [42] simply assumes that only distinct labels are inserted into an initially empty composite variable, that these labels are selected from the set $\{1, 2, \ldots, n\}$ and that all of these label insertion permutations as represented by $X_n$ are equally likely. Schellekens [63] assumes that a composite variable is initialised as a discrete partial order of size $n$, that its possible labelings are represented by $X_n$ and that all of these canonically-ordered labelings are equally likely. So $X_n$ in the context of Knuth's work [42] represents the possible label insertion permutations on the initially empty composite variable and $X_n$ in the context of Schellekens's work [63] represents the possible canonically-ordered labelings on the initial discrete composite variable of size $n$. So MOQA's "insertion" functions, such as the MOQA top function, actually rearrange an existing data structure as they insert edges rather than nodes whereas Knuth allows for labels/nodes to be really inserted during program run-time. There is another way of viewing Schellekens's assumption that all data structure labels/nodes are already present prior to any sequence of MOQA functions: that $n$ insertion

functions which do nothing more than add a label/node and are in accord with Knuth's assumptions above have already been applied to the composite variable prior to any sequence of MOQA functions[10]. It is clear that both works are only concerned with the relative order of labels.

There is now enough background to examine randomness preservation as presented by Knuth.

**Definition 66** (Knuth's randomness preservation [42])**.** *At a particular moment in its lifetime, a composite variable of type $\alpha$ and size $n$ is randomness preserving if all of its possible states at that moment are equal to $S_{X_{n,\alpha}}$ after the multiplicity of every distinct element in the multiset of its possible states has been divided/multiplied by some common divisor/multiplier*[11]*.*

So, when a sequence of functions is applied to an initially empty composite variable and that sequence consists of $n$ insertion functions that are in agreement with Knuth's assumptions enumerated above, then the resulting composite variable of size $n$ is *always* randomness preserving according to this definition. (For this case, the common divisor, or equally it could be the common multiplier, for the frequency of every distinct state that can possibly result is simply one.) As Knuth states: "Occasionally an analysis of mixed insertions and deletions turns out to be workable because it is possible to prove some sort of invariance property; if we can show that deletions preserve "randomness" of the structure, in some sense, the analysis reduces to a study of structures built by random insertions" [42]. So $D_\alpha(x)$ preserves Knuth's randomness when it is applied to a randomness preserving composite variable of type $\alpha$ and size $n$ and the resulting possible states are equivalent from the average-case standpoint to the possible states after $n-1$ insertion functions on an initially empty composite variable.

Why are these two state multisets equivalent from the average-case standpoint? While the set of distinct states that can possibly result after this randomness preserving deletion function is equal to $A(S_{X_{n-1,\alpha}})$, the multiplicity of each distinct state possible will be some fixed multiple larger/smaller than its multiplicity in $S_{X_{n-1,\alpha}}$. For example, the multiset of data structures whose elements are ten instances of binary tree III in Figure 6.5 and five instances of

[10]Some of the new MOQA functions in Section 3.3 apply Knuth's viewpoint instead.

[11]This definition is not directly found in Knuth's paper [42] but does capture his intent.

each of the other binary trees in Figure 6.5 is equivalent from the average-case standpoint to $S_{X_3,\alpha_{bst}}$ because the multiplicity of each distinct state in the former multiset is five times larger than its multiplicity in the later. So though a distinct state's multiplicity may not be the same in both of the multisets, its comparative frequency to any of the other distinct states within either of the multisets is the same. Hence, both state multisets are considered equivalent from the average-case analysis perspective. (This was also briefly discussed in Section 2.1.)

Consider the initially empty composite variable that has had applied to it a sequence of functions consisting of $i$ insertion functions, which automatically preserve Knuth's randomness, and $d$ deletion functions that also preserve Knuth's randomness, $0 \leq d \leq i$ and $i-d=n$. What is the average-case cost of the function, i.e. the $(\mathrm{i}+\mathrm{d}+1)^{\mathrm{th}}$ function, next applied to the composite variable when that function too preserves Knuth's randomness? It is the sum of the function cost when applied to each state in $S_{X_{i-d},\alpha}$ divided by $|S_{X_{i-d},\alpha}|$. So effectively these $d$ deletion functions can be ignored when calculating average-case cost. However, if some of the deletion functions in the sequence of $i+d$ functions did not preserve randomness and/or the $(\mathrm{i}+\mathrm{d}+1)^{\mathrm{th}}$ function is a deletion function that does not preserve randomness, then $S_{X_{i-d},\alpha}$ is no longer guaranteed to correctly reflect the possible data structure states. *So, according to Knuth's definition of randomness, only deletion functions can destroy data structure randomness.*

Knuth's work [42] also categorises different types of insertion and deletion functions and proves which combinations preserve the randomness of a composite variable, i.e. proves the conditions under which deletion insensitivity is maintained.

Schellekens's definition of randomness preservation is now cast into the syntax of this section to allow for an easier comparison between the two definitions.

**Definition 67** (Schellekens's randomness preservation [63])**.** *At a particular moment in its lifetime, a composite variable of type $\alpha$ and size $n$ is randomness preserving if all of its possible states at that moment are equal to $A(S_{X_n,\alpha})$ after the multiplicity of every distinct element in the multiset of its possible states has been divided by the highest common divisor.*

In other words, the set of possible states for such a composite variable is equal to $A(S_{X_n,\alpha})$ and all of these states are equally likely to occur. Hence, while Knuth's insertion functions never destroy what he considers to be data structure randomness, there is no such guarantee for insertion functions under Schellekens's definition of data structure randomness. *So, according to Schellekens's definition of randomness, both insertion and deletion functions can destroy data structure randomness.*

It should now be clear that data structure randomness in the MOQA book [63] is more fragile than it is in Knuth's research [42] because the former can be jeopardised by a greater range of functions. This is why MOQA functions, the edge insertion functions in particular, are so restrictive in comparison to the functions that Knuth allows. For example, consider the data structure of size four in Figure 2.3 when it results from just insertion functions. This data structure can have five distinct canonically-ordered labelings on it for the max-heap label ordering. The number of label insertion permutations on the initially empty data structure or the number of canonically-ordered labelings on the initial discrete data structure of size four which map to any one of these five canonically-ordered labelings depends on the type of insertion function applied; assuming, of course, that the type of insertion function applied maintains max-heap label ordering. However, following Knuth, the data structure in Figure 2.3 with the max-heap label ordering is *always random* regardless of the insertion function applied because insertion functions intrinsically preserve Knuth's randomness. On the other hand, following Schellekens, the data structure in Figure 2.3 with the max-heap label ordering is *never random* regardless of the insertion function applied. This is a strong statement to make, yet simple to verify. The total number of distinct canonically-ordered labelings possible on the initial discrete data structure of size four divided by the total number of distinct canonically-ordered labelings possible on the data structure in Figure 2.3, i.e. $4!/5$, never resolves to a whole number. Therefore, it is never possible for the five distinct canonically-ordered labelings on this data structure to be equally likely when all of the distinct canonically-ordered labelings on the initial discrete data structure are equally likely. Hence, Schellekens's randomness can never be obtained for the data structure in Figure 2.3 when the max-heap label ordering is on it.

Note that $\alpha$ in the MOQA theory [63] can change throughout the composite

variable's lifetime. Initially $\alpha$ will refer to the discrete partial order family and the max-heap label ordering. MOQA functions, such as the MOQA product and split function, can then change $\alpha$'s data structure family though they will not change its label ordering. By contrast, $\alpha$ in the Knuth theory [42] is constant throughout the composite variable's lifetime.

The major commonality between these definitions is that they both require the composite variable of type $\alpha$ and size $n$ to be stationary. Flajolet, Françon and Vuillemin [20] define a composite variable of type $\alpha$ and size $n$ as *stationary* when the average-case cost of the $\alpha$ function next applied to the composite variable can be correctly calculated knowing only the composite variable's size. Hence, the sequence of functions previously applied to a stationary composite variable is irrelevant when it comes to determining the average-case cost of the function next applied.

As Knuth's definition of randomness allows for a wider range of data organisations, does it make sense to consider it as a replacement for the definition used by MOQA? As already covered in detail, every MOQA function has its own specific formula that calculates its average-case cost when it is applied to a fixed po-structure by iterating through the shape of that fixed po-structure. These formulas are then supplied to the MOQA static analysis tool. To change the meaning of randomness in MOQA would require discarding the current average-case formulas for fixed po-structures and supplying to the MOQA static analysis tool a new average-case formula for each of the functions that are now chosen for analysis. The functions chosen for analysis will be some subset of the functions that the tool is now capable of analysing, which due to the randomness change include any $I_\alpha(x)$ and any $D_\alpha(x)$ that preserves Knuth's randomness. So the key question is, for each freshly chosen function, can a formula that calculates its average-case cost when it is applied to a fixed po-structure be provided to the MOQA static analysis tool? (The average-case formulas for MOQA functions when they are applied to inductive po-classes is properly addressed for the first time by Hickey [35] and here. The MOQA book's theory [63] is for functions when they are applied to fixed po-structures. Therefore, this is the context for which the randomness swap is considered.) As the current MOQA average-case formulas step through the shape of a fixed po-structure, there would appear to be no reason why not, if the new formulas continue to adhere to MOQA's literal approach to average-

case derivation. In fact, in the worst-case scenario, the average-case cost that arises from applying a function that preserves Knuth's randomness to a data structure of type $\alpha$ and size $n$ can be calculated as follows: generate each data structure in $S_{X_n,\alpha}$, obtain function cost for each data structure by stepping through its shape, sum together all of these function costs and then divide the sum by $|S_{X_n,\alpha}|$. Admittedly, the average-case formulas for the current MOQA functions are not as intensive as this. Only once do they need to step through the shape of the fixed po-structure to which they are applied due to the shape being in series-parallel, instead of stepping through the shape for every canonically-ordered labeling on the fixed po-structure. Nonetheless, either way, there is a finite number of data structures being iterated through.

It is worth acknowledging that there will be instances of the worst-case scenario where generating each of the data structures in the finite multiset $S_{X_n,\alpha}$ is too large a problem to solve in practice, no matter how much computational power is thrown at the problem, because of the magnitude of $n$ in conjunction with the specific $\alpha$. However, there will be many instances of the worst-case scenario where the number of data structures in $S_{X_n,\alpha}$ is modest enough for them to be individually generated statically and so, in these cases, the worst-case scenario is solvable in practice. It is also worth noting that not much attention is generally given to the efficiency of non real-time static analysis tools; the focus is generally on their accuracy. So the number of hours/days taken by the MOQA static analysis tool to complete its analysis has never been of particular concern because it is a one-off event, whereas it is probable that the program being analysed will be executed many times.

Though at times the worst-case scenario will be the only solution for some of the functions chosen for analysis, it may be that the average-case formulas for the other functions chosen can also take shortcuts in their calculations because of the data structure family, label ordering and distribution of canonically-ordered labelings that they specifically deal with. It could be decided that only functions with a level of abstraction in their average-case formulas are to be included in the new repertoire. So the conclusion here is that this change to Schellekens's randomness appears to be achievable. If implemented, then the result would be a more powerful MOQA static analysis tool because the tool is now able to get the average-case cost of far more insertion and deletion functions. The main foreseen downside to this change is that the MOQA book

[63] would suffer from a loss of novelty as its uniqueness rests on it being able to statically analyse algorithms that comply with its own definition of randomness.

## 6.10 Chapter Summary

Alan Turing proved that the halting problem is undecidable [73] and, as a consequence, a general tool that determines the average-case behaviour of any program cannot exist, since some of the programs will not halt. Yet, it is still reasonable to aim for a tool that can determine average-case behaviour for a range of diverse algorithms. So this chapter examined the progress made by others towards this aspiration. While the majority of systems discussed here contributed in some way towards the automation of average-case analysis, the system that clearly made the greatest progress is LUO, which was subsequently improved upon by Mishna's extension.

As well as reviewing the current literature, this chapter carefully compares each key work against MOQA. This had not been done before in any significant detail and the benefits of such attention are plain. Firstly, the comparison revealed how MOQA distinguishes itself from other tools, e.g. Metric tracks unlabeled data structures whereas MOQA tracks labeled data structures. Secondly, it identified various features that could be assimilated into future versions of MOQA. For example, MOQA could take advantage of the mathematical properties inherent in functional languages as ACE does if it switched to a functional language or it is likely that MOQA could be strengthened from replacing its definition of randomness preservation with Knuth's. Indeed, the development of a new MOQA function resulted from contrasting LUO's product function with MOQA's product function. So, all in all, this chapter concretely shows how MOQA profits from a thorough comparison against existing research.

# Chapter 7

# Conclusion

This chapter commences with an overview of MOQA for the purpose of delimiting its scope. Next, there is a summary of paths that the research can take in the future. The chapter then concludes with a synopsis of this work's contributions.

## 7.1 MOQA Assessment

A variety of MOQA theory evaluations are dispersed throughout this work and occur relative to context. The aim of this section is to gather together the most important of these conclusions and to cultivate some of them further. This unification will provide a more coherent and comprehensive assessment of MOQA.

For ease of presentation, this assessment is separated into the following five topics.

#### Recurrence relations

MOQA aims to express the average-case cost of a recursive algorithm as a recurrence relation. If a closed-form solution is desired, then the suggestion is that this recurrence relation is plugged into some existing automated solver, such as Mathematica [49].

The first drawback with this is that MOQA cannot claim to be among the leading average-case static analysis systems if it can only produce *closed-form* expressions for *straight-line* code; static analysis systems that succeed

in determining closed-form expressions for the average-case cost of recursive algorithms are detailed in Chapter 7. The second drawback is that it neatly sidesteps the fact that automatically solving recurrence relations can be difficult and, in some cases, may result in closed-form expressions from which asymptotic growth is "unpleasant" to identify [81].

Another concern is that MOQA lags behind the latest methods in its field, which revolve around generating functions because they can represent recurrence relations and they are "easier to handle with a computer algebra system" [81]. Note, however, that generating functions do not eliminate the need for a sophisticated automated solver. For example, LUO devotes over $5,000$ lines of its code to solving such equations; LUO is reviewed in Section 6.5.

Therefore, *MOQA is not yet a complete solution* and so it is paramount that this issue is resolved regardless of the manner in which closed-form expressions for MOQA recurrence relations are actually derived, i.e. regardless of whether MOQA itself is extended or the pipeline to some other system is implemented.

### Label ordering

It is assumed that the label ordering on any data structure that can be represented in a MOQA′ random bag is max-heap ordered[1]. This allows label ordering to be an implicit constant in MOQA's average-case formulas, as opposed to being an explicit variable. In other words, this assumption allows the MOQA calculations to fuse the label ordering onto the data structure. Nailing down the label ordering severely restricts MOQA's extensibility in this regard because the addition of a new label ordering, such as the binary search tree label ordering, to the MOQA theory would require each of the MOQA function's average-case formula(s) for each acceptable data structure type to be recalculated by hand for this label ordering. For example, the MOQA product function's average-case formula(s) for a fixed po-structure and for each of the inductive po-class categories defined in Chapter 4 would have to be redone from scratch for each new label ordering.

Is this level of effort always to be expected when a system that automatically determines average-case behaviour has a label ordering added to it? Consider the addition of Green's box/min label operator to the LUO gram-

[1]The other possibility is that the label ordering is min-heap ordered. Either of these assumptions are acceptable because MOQA's average-case formulas apply in either scenario.

mar; this is discussed in Section 6.6 and the average-case static analysis system LUO is discussed in Section 6.5. This operator assigns the minimum label to a component of a data structure, which allows a LUO user to define not only a data structure but also the min-heap label ordering on it. (A similar max label operator can be introduced for the max-heap label ordering.) Due to the design of LUO, it is only necessary to update LUO's engine with the generating function for the new box operator. So a label ordering is added to LUO by layering the label ordering definition over the data structure definition and this removes the need to rework the generating functions that already exist in LUO's engine. Hence, LUO can translate a structural definition that involves the box operator into a generating function after a straightforward system extension and this ease is due to the *compositional* nature of LUO's translation process. So it would be just as simple to add yet another label order operator to LUO's grammar once the generating function for that operator is determined. Therefore, LUO's system and Mishna's provision of separate operators for label order allows average-case behaviour to be calculated without "baking" label ordering into every operator, as MOQA does. Hence, this label ordering approach is far more extensible than MOQA's, which shows that MOQA's awkwardness in this matter is due to how it own theory has been developed, rather than being due to a theoretical limitation.

### Data structure representation

In Artificial Intelligence, an environment can be modelled either iconically or logically.

- An *iconic representation* has a strong structural resemblance to the environment it is representing.
- A *logical representation* describes the environment it is representing but there is no necessity for it to have a strong structural resemblance to the environment.

According to this classification, MOQA represents its data structures iconically. Such a form of representation is required because of MOQA's average-case formulas, which usually iterate over the *shape* of a data structure[2]. How-

[2]The new average-case formulas developed by this work for inductive po-classes, and presented in Chapter 4, behave similarly because they are an extension of the MOQA system.

ever, any formula that needs to explicitly iterate over the shape of a data structure takes a very literal path in deriving its result. So, as MOQA's average-case formulas mainly rely on there being some concrete representation of data structure shape, it is evident that MOQA introduces little abstraction between the actual data structures and its modelling of them. Hence, it is argued here that this approach costs MOQA because more conceptual data structure representations tend to give more scope for analysis and crucially, may result in closed-form expressions for average-case cost, which would be a marked improvement on the recurrence relations that MOQA produces.

To demonstrate the power of abstraction when it comes to data structure representation, take, once again, the example of the more successful LUO; see Section 6.5 for details. The LUO system represents its data structures with generating functions and hence, represents them logically. Although it may not be obvious at first glance what type of data structure a particular generating function represents, it is this abstraction from data structure shape to equation that lends LUO much of its strength because data structures are now represented in a conceptual form that can be mined for average-case data via existing mathematical techniques, and this results in closed-form expressions for average-case cost. Therefore, this evidence is supportive of this work's conclusion, which is that MOQA seriously impedes its own success by representing data structures iconically and thus, by its literalness when it comes to calculating average-case cost.

### The complexity of average-case analysis

Consider the following observation:

> "Systems simple enough to be understandable are not complicated enough to behave intelligently; systems complex enough to behave intelligently are not simple enough to understand [17]."

This paradox makes the interesting, and somewhat intuitive, point that simple systems cannot manage the intricacy associated with complex problems; such intricacies are very apparent in large-scale systems that predict the weather or diagnose space shuttle faults in real-time. Though fully detailed in Chapter 2, the MOQA solution to average-case analysis can be summed up as follows: apply one of the limited number of MOQA functions to a series-parallel data

structure whose canonically-ordered labelings are all equally likely and then iterate over the shape of that data structure to obtain that function's average-case cost. Clearly, MOQA is a simple system. Yet it is well-known that determining average-case behaviour is often a complex task, whether doing so by hand or automatically. Wegbreit states that "the analysis of many algorithms requires considerable mathematical expertise; an expert system would necessarily include all the techniques in the monumental work of Knuth" [76]. This is illustrated by Jonassen and Knuth in a paper aptly title "A simple algorithm whose analysis isn't" [40], which shows that a "surprisingly intricate analysis" is required to determine the performance of the standard insertion, deletion and search operations on binary search trees of size *three*. Therefore, while MOQA can be lauded for its simplicity, the main repercussion is that it is unable to analyse algorithms whose average-case behaviour is difficult to establish. In other words, MOQA's simplicity prevents it from being "intelligent" enough to handle the usual complications that arise in average-case analysis. This is further emphasised by considering the algorithms that MOQA *can* analyse.

A MOQA function is always applied to a series-parallel data structure whose canonically-ordered labelings are all equally likely and consequently, the function will always return a series-parallel data structure whose canonically-ordered labelings are all equally likely. In the field of automated average-case analysis, it is novel indeed to rely on functions that can guarantee such a uniform distribution of output, as the MOQA book [63] highlights. However, it is the class of algorithms that actually use, and only use, such functions which is of real import. The algorithms in this class only consider series-parallel data structures and continuously maintain the uniform distribution of their data structures' canonically-ordered labelings, i.e. no matter the modification made by one of these algorithms to one of its series-parallel data structures, all of the data structure's canonically-ordered labelings are still equally likely when considered over all possible executions of the algorithm. So MOQA can only analyse algorithms with this specific behaviour, e.g. insertion-sort, quicksort and mergesort. Such algorithms are generally found towards the start of textbooks that explain algorithm analysis, with "The Big Book (of Algorithms)" [13] being an example of one such book, because it is not too hard to reason out their average-case behaviour and so is a gentle introduction to the field.

Thus, MOQA can only analyse an algorithm whose very predictability ensures that its average-case behaviour is relatively simple to ascertain. Hence, MOQA is unable to statically analyse a considerable swath of algorithms because its theory is too basic to manage the level of intricacy normally associated with average-case analysis.

**Algorithms that can be analysed**

All the following requirements must hold true if MOQA is to successfully analyse an algorithm:

1. each of the algorithm's data structures must be in series-parallel throughout the algorithm,

2. each of the algorithm's data structures must have either the max-heap or the min-heap label ordering on it throughout the algorithm,

3. for each of the algorithm's data structures, the initial uniform distribution of its canonically-ordered labelings must never be disrupted by the algorithm regardless of the modifications the algorithm makes to it,

4. it must be possible to express the algorithm solely in terms of the MOQA functions and the reduced range of if statements, for statements and other program constructs that Hickey [35] allows for, and finally,

5. it must be possible to express the algorithm's average-case cost solely in terms of the average number of comparisons that take place *within* its data structures.

Additionally, these requirements were composed for algorithms whose data structures can be statically represented with fixed po-structures. However, it is atypical for there to be interest in the analysis of such algorithms. To illustrate why, consider the algorithm that applies some sequence of functions to the data structure which initially is the fixed po-structure illustrated in Figure 2.2. The average-case cost of this algorithm is averaged over the costs of just eight run-times; eight being the number of distinct canonically-ordered labelings of that fixed po-structure whose size is five. Hence, such a scenario is not usually considered in average-case analysis because the fixed number of distinct algorithm executions means that determining the average-case cost

of an algorithm whose data structure size remains constant is a frivolous enterprise[3]. Instead, it is far more conventional to consider the average-case cost of an algorithm whose data structure size is the variable $n$, with $n \geq 0$. However, the MOQA theory was specifically designed for fixed po-structures and this constraint motivated Section 4.3's expansion of the MOQA theory; the updated theory admits inductive po-classes and inductive po-classes can represent data structures of size $n$. (The MOQA fixed po-structure design also contributed to why MOQA selected such a literal data structure representation, which was discussed in an earlier topic.) So, this new version of the MOQA theory permits an algorithm's data structure to be statically represented with the inductive po-class $I$. When this is the case, then the third requirement above becomes the following: all of $I$'s fixed po-structures of size $n$ must be equally likely, all the canonically-ordered labelings of each of these fixed po-structure must be equally likely and these two initial properties of $I$ must never be disrupted by the algorithm regardless of the modifications the algorithm makes to $I$.

This is a very specific set of requirements; the previous topic examined how the third requirement, on its own, rules out the MOQA analysis of many algorithms. Therefore, while an algorithm may satisfy some of these requirements, it is not common to satisfy all of them. Take the heapsort algorithm as an example. While it is a comparison-based algorithm with the max-heap/min-heap label ordering on its series-parallel data structure[4], it does not maintain the required uniform distribution of output after the initial extraction of the largest/smallest label from the heap. The binary search algorithm demonstrates an algorithm whose average-case cost cannot be expressed in terms of the MOQA measure because each of its comparisons take place between the label of a node within the sorted list and the label of an item *outside* of the sorted list.

Therefore, MOQA's rigid design prevents it from being categorised as a general average-case static analysis tool. Instead, it falls into the same category

[3]There is also a programmer's concern about code smell when encountering a data structure whose size is hard-coded.

[4]While a heap is in series-parallel, there is still a static analysis problem when it comes to representing all heaps of size $n$ with a definition that is inductively defined. This problem stems from the fact that such a structural definition cannot represent a complete or nearly complete tree, which is part of the definition of a heap. This dilemma, in the MOQA context, was first identified by Hickey [35].

of systems that COMPLEXA does, see Section 6.4, because it is a purpose-built system for a small tightly defined range of algorithms.

To summarise this section, the MOQA average-case solution is often expressed as a recurrence relation and therefore, is often an incomplete solution. The MOQA average-case solution is also limited when it comes to expansion opportunities. This is due to the literal, and hence inflexible, approach that it takes in calculating average-case cost and the restrictions that it places on the algorithms that it can analyse. These constraints are particularly unfriendly when considering the complexity regularly inherent in average-case analysis.

However, the MOQA concept could prove to be a useful teaching tool; it could be a helpful way of introducing undergraduate students to the average-case analysis of common sorting algorithms. So, this is an area where the novel MOQA random structure preserving functions could be a boon.

## 7.2 Future Work

Certain key tasks for the future were identified during the course of this research and they are as follows:

- The new MOQA functions discussed in Section 3.3 could be added to the current MOQA language *MOQA-Java* and thereupon, incorporated into the current MOQA static analysis tool *Distri-Track*.
- *Distri-Track* could have its erroneous average-case formulas, which are detailed in Section 4.5.2.3, replaced by the correct versions, which are also detailed in the same section.
- *Distri-Track* could be augmented with Chapter 4's new inductive po-class types and their average-case formulas.
- The MOQA language could become functional, instead of being object-oriented, to take advantage of the mathematical properties associated with such languages; see Section 6.3 for further explanation.
- The MOQA static analysis tool could be extended to derive closed-form expressions, in place of recurrence relations, for average-case behaviour. As difficult recurrence relations can be solved via generating functions, it may be helpful to explore current developments in this area.

## 7.3 Thesis Summary

This thesis revolves around MOQA, whose objective is "to present a new approach to the Average-Case Analysis of Algorithms" [63].

Initially, this research developed an implementation of the MOQA language. This implementation, which is known as *MOQA-Java*, is designed to assist programmers in writing code that adheres to the requirements specified in the MOQA book [63]. After carefully designing and implementing this Java package, a few algorithms, whose translation into *MOQA-Java* is straightforward, were analysed by hand. The aim of this examination was to look at the extra overhead that the MOQA functions introduce because of their very definitions and therefore, to compare the behaviour of algorithms when written in these functions to their usual behaviour. The study revealed that the *MOQA-Java* implementation does result in extra costs. However, these extra costs are not significant enough to cause the asymptotic behaviour of each analysed algorithm to deviate from what is expected.

The *MOQA-Java* development led this work to create new helper functions and, far more importantly, to create new MOQA functions. These new MOQA functions increase the potential for MOQA to implement and analyse more algorithms. For example, some of the MOQA functions that originated in this work enabled MOQA to statically analyse the treapsort algorithm described in Appendix A. Moreover, consider the new MOQA product function described in Section 6.5. This function products together two distinct data structures and is the first of the MOQA functions to be applied across separate data structures.

In scrutinising the MOQA language, this work found the MOQA claim regarding the reversibility of its language to be lacking and detailed why. It also discovered a gap between how the MOQA theory is defined and what the MOQA book [63] envisions will be accomplished by the MOQA theory, which is explored in Chapter 4. Firstly, this chapter redefines and expands upon the MOQA theory. As a result, the theory can now encompass data structures other than those of a fixed size and shape. In other words, it can now encompass data structures of size $n$, e.g. inductive po-classes. This was a crucial evolution because tracking the average-case cost of an algorithm for a fixed data structure offers little reward. Secondly, this chapter carries on

the research that Hickey [35] began; Hickey developed general average-case MOQA formulas for one type of inductive po-class. In doing so, this work constructed an inductive po-class framework, into which Hickey's type [35] was placed after corrections were made to some of the formulas developed by that work. Next, general average-case MOQA formulas were developed for some of the other inductive po-class types in this framework. These new MOQA data structure types and their corresponding formulas, like the new MOQA functions, further increase the potential for MOQA to implement and analyse more algorithms. Additionally, the inductive po-class categorisation facilitated this work in identifying those that present a significant challenge when it comes to generating general average-case MOQA formulas for them.

Attention was given to whether MOQA can handle duplicate labels although it appeared to Schellekens that duplicate labels could be safely introduced into MOQA with negligible effort [63]. However, the work here showed this initial reasoning to be erroneous. This finding was also supported by examples; one of the examples that dispelled the notion that it would be simple to add duplicate labels to MOQA calculated the average number of swaps for MOQA's insertion-sort for a particular distribution of input that involved duplicate labels. These examples helped in cementing this work's verdict on the matter, which is that admitting duplicate labels to the MOQA theory would require it to be altered considerably.

Finally, a thorough literature review was performed. Although much of this literature was referenced in earlier works on MOQA, there had never previously been an intensive comparison between MOQA and the other systems in its field. This investigation revealed areas in which MOQA fell short and also areas in which it flourished. Moreover, it gave support to the MOQA assessment that followed, which perceives MOQA to be a system with narrow applicability despite some of its creative ideas for static average-case analysis.

To speak *ad rem*, the author trusts that these contributions meet the aims and objectives laid out at this work's commencement.

# Appendix A

# Treapsort Algorithm

The treap data structure was first introduced by Seidel and Aragon [68]. A treap is a binary search tree in which each node has both a key and a priority. As for a binary search tree, the inorder traversal of the treap's nodes returns the keys of the treap in ascending order. Furthermore, the treap's priorities are max-heap ordered.

For the treapsort pseudo-code below, let $A$ denote a discrete partial order. This partial order is first converted into a treap by the treapgen algorithm. This treap is then converted into a total order by $|A|-1$ consecutive iterations of the MOQA lift function. Let $A_{treap,\,i}$ denote the treap that results from treapgen after $i$ nodes have been deleted from it by the MOQA lift function, $0 \leq i \leq |A| - 2$.

$TreapSort(A)$:
    $TreapGen(A)$
    for $i = 0$ to $|A| - 2$ do
        $Lift(A_{treap,\,i})$

For the treapgen pseudo-code below, let $A$ denote a discrete partial order whose nodes are randomly ordered as $a_1, \ldots, a_{|A|}$. When the MOQA top function determines that the node $a_j$ is the maximum node in $A$, let $A_{left}$ denote the discrete partial order that is $a_1, \ldots, a_{j-1}$ and let $A_{right}$ denote the discrete partial order that is $a_{j+1}, \ldots, a_{|A|}$, $1 \leq j \leq |A|$.

$TreapGen(A)$:
    if $|A| > 1$ then
        $Top(A)$
        $TreapGen(A_{left})$
        $TreapGen(A_{right})$

*Distri-Track*'s analysis of the MOQA book's treapsort algorithm [63] is detailed in Hickey's research [35], where the algorithm is known as HOTsort.

# Appendix B

# Algorithms LUO Can Analyse

The LUO system is designed for the purpose of statically calculating an algorithm's average-case cost and this system is discussed in Section 6.5. The following list of algorithms/problem types are just some of those for which LUO can statically determine average-case cost. The LUO Cookbook [23] thoroughly details the analysis process for each example below, as well as for other examples whose descriptions are too complex to briefly summarise here.

1. Computation of exponential $x^c$ in group structure $G$, e.g. the integers modulo a prime, using the standard binary method.

2. Computation of exponential $x^c$ in group structure $G$ using the two heuristics developed by Morain and Olivos [54].

3. A concurrent access problem along the lines of the problems detailed by Beauquier et al [4] and Geniet [29], where the problem is described in terms of finite automata and each transition is associated to a procedure to which a cost is attached.

4. Given two rows consisting of $n$ points each, how many ways are there of drawing straight lines between them so that (I) from each point there is a line, (II) no lines cross, and (III) a line from point $i$ in row $a$ can only go to points $i-1$, $i$, $i+1$ in row $b$, and vice versa? This combinatorial problem was introduced by Steven Bird in the transaction *1107@epistemi.ed.ac.uk* of the newsgroup *Sci.math*.

5. A differentiation procedure as typically encountered in computer algebra

systems and a modified version of this procedure where its subtrees are copied instead of shared.

6. Computation of higher order derivatives using iterated differentiation.

7. Partial analysis of the *left-distributivity* rule, which is used classically in the *expand* primitive of computer algebra systems.

8. A term rewriting system with mutually recursive operators, as introduced by Choppy et al [8].

9. Three distinct ways of shuffling a binary tree, as introduced by Choppy et al [8].

10. Determining the collection of letters that can occur as initial letters in a regular language generated by a regular expression using an example provided by Vivares [75].

11. Estimating the expected number of connected components in a random labeled 2-regular graph of size $n$, where an undirected graph is said to be 2-regular if each node has degree 2.

12. Pollard's rho-method for integer factorisation, as described in [43].

13. The variance of internal pathlength in binary trees.

14. The number of partitions of $n$ into $k$ parts, when $k$ is fixed and $n \to \infty$.

15. Banach's matchbox problem, which is a particular case of the toilet paper problem [44].

# Bibliography


[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs (2nd Edition)*. MIT Press, 1996.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[3] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[4] J. Beauquier, B. Bérard, and L. Thimonnier. On a concurrency measure. Technical Report 306, Laboratoire de Recherche en Informatique, 1986.

[5] Boris Beizer. *Micro Analysis of Computer System Performance*. John Wiley & Sons, 1978.

[6] Charles H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[7] Charles H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1):16–23, 1988.

[8] C. Choppy, S. Kaplan, and M. Soria. Algorithmic complexity of term rewriting systems. In *Proceedings of the 2nd Rewriting Techniques and Applications Conference*, pages 256–273. Springer, 1987.

[9] Jacques Cohen. *Automatic Analysis of Programs (Microanalysis)*. `http://pages.cs.brandeis.edu/~jc/automatic_analysis_of_programs.html` .

[10] Jacques Cohen. Computer-assisted microanalysis of programs. *Communications of the ACM*, 25(10):724–733, 1982.

[11] Jacques Cohen and Carl Zuckerman. Two languages for estimating program efficiency. *Communications of the ACM*, 17(6):301–308, 1974.

[12] Louis Comtet. *Advanced Combinatorics*. Reidel Publishing Company, 1974.

[13] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd Edition)*. MIT Press, 2001.

[14] Paidi Creed. *Generating Functions and their Application to the Average-case Time Complexity of Algorithms*. 2004 Final Year Project Report, `http://www.ceol.ucc.ie/public.php` .

[15] Maylis P. Delest and Jean Marc M. Fedou. Attribute grammars are useful for combinatorics. *Theoretical Computer Science*, 98(1):65–76, 1992.

[16] E.W. Dijkstra. *Letter from E.W.Dijkstra to C.A.R.Hoare, 1970, EWD 292*. `http://www.cs.utexas.edu/users/EWD` .

[17] George B. Dyson. *Darwin Among The Machines: The Evolution Of Global Intelligence*. Helix Press, 1997.

[18] J.P. Fitch. On algebraic simplification. *The Computer Journal*, 16(1):23–27, 1973.

[19] Philippe Flajolet. Analytic analysis of algorithms. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 186–210. Springer, 1992.

[20] Philippe Flajolet, Jean Françon, and Jean Vuillemin. Computing integrated costs of sequences of operations with application to dictionaries. In *STOC '79: Proceedings of the 11th ACM Symposium on Theory of Computing*, pages 49–61. ACM Press, 1979.

[21] Philippe Flajolet and Bruno Salvy. Computer algebra libraries for combinatorial structures. *Journal of Symbolic Computation*, 20(5–6):653–671, 1995.

[22] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Lambda-Upsilon-Omega: an assistant algorithms analyzer. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, 357:201–212, 1989.

[23] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Lambda-Upsilon-Omega: the 1989 cookbook. Research Report 1073, Institut National de Recherche en Informatique et en Automatique, 1989.

[24] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 79(1):37–109, 1991.

[25] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[26] Robert W. Floyd. Assigning meaning to programs. In *Proceedings of the AMS Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[27] Dominique Foata. *La Série Génératrice Exponentielle dans les Problèmes d'Énumération*. Presses de l'Université de Montréal, 1974.

[28] Michael P. Frank. Introduction to reversible computing: motivation, progress and challenges. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 385–390. ACM Press, 2005.

[29] D. Geniet. *Automaf, un système de construction d'automates synchronisés et de mesure de parallélisme*. PhD thesis, University of Paris-Sud, Orsay, France, 1989.

[30] Ian P. Goulden and David M. Jackson. *Combinatorial Enumeration*. John Wiley & Sons, 1983.

[31] Daniel Green. *Formal Languages and their Uses*. PhD thesis, Stanford University, CA, USA, 1985.

[32] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 106–112. IEEE Press, 2003.

[33] Haskell. `http://www.haskell.org` .

[34] Christopher Healy, Mikael Sjödin, Viresh Rustagi, and David Whalley. Bounding loop iterations for timing analysis. In *Proceedings of the IEEE Real-Time Applications Symposium*, pages 12–21. IEEE Press, 1998.

[35] David Hickey. *Tracking Data Structures for Automated Average Time Analysis*. PhD thesis, University College Cork, Cork, Ireland, 2008.

[36] Timothy Hickey and Jacques Cohen. Automating program analysis. *Journal of the ACM*, 35(1):185–220, 1988.

[37] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–16, 1962.

[38] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[39] C.A.R. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, 1975.

[40] Arne T. Jonassen and Donald E. Knuth. A trivial algorithm whose analysis isn't. *Journal of Computer and System Sciences*, 16:301–322, 1978.

[41] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[42] Donald E. Knuth. Deletions that preserve randomness. *IEEE Transactions on Software Engineering*, 3(5):351–359, 1977.

[43] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms, Volume 2 (2nd Edition)*. Addison-Wesley, 1981.

[44] Donald E. Knuth. The toilet paper problem. *The American Mathematical Monthly*, 91(8):465–470, 1984.

[45] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3 (2nd Edition)*. Addison-Wesley, 1998.

[46] Dexter Kozen. Semantics of probabilistic programs. In *Proceedings of the 20th Symposium on Foundations of Computer Science*, pages 101–114. IEEE Press, 1979.

[47] Algolib Library. `http://algo.inria.fr/libraries/` .

[48] Yanhong A. Liu and Gustavo Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 31–40. Springer-Verlag, 1998.

[49] Mathematica. `http://www.wolfram.com/mathematica` .

[50] John McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.

[51] Daniel Le Métayer. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[52] Marni Mishna. Attribute grammars and automatic algorithm analysis. *Advances in Applied Mathematics*, 30(1–2):189–207, 2003.

[53] ML. `http://www.smlnj.org` .

[54] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. Research Report 983, Institut National de Recherche en Informatique et en Automatique, 1989.

[55] Peter Naur. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[56] University of Florida Reversible & Quantum Computing Research Group. `http://www.cise.ufl.edu/research/revcomp` .

[57] Peter Puschner and Anton Schedl. Computing maximum task execution times - a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, 1997.

[58] Chittoor V. Ramamoorthy. Analysis of computational systems: discrete Markov analysis of computer programs. In *Proceedings of the 1965 20th National Conference*, pages 386–392. ACM Press, 1965.

[59] Lyle H. Ramshaw. *Formalizing the Analysis of Algorithms*. PhD thesis, Stanford University, CA, USA, 1979.

[60] Vivek Sarkar. Determining average program execution times and their variance. *SIGPLAN Notices*, 24(7):298–312, 1989.

[61] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

[62] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and Functional Programming*, pages 202–211. ACM Press, 1986.

[63] Michel Schellekens. *A Modular Calculus for the Average Cost of Data Structuring*. Springer, 2008.

[64] Michel Schellekens. MOQA; unlocking the potential of compositional static average-case analysis. *Journal of Logic and Algebraic Programming*, 79(1):61–83, 2010.

[65] Robert Sedgewick. *Quicksort*. PhD thesis, Stanford University, CA, USA, 1975.

[66] Robert Sedgewick. Quicksort with equal keys. *SIAM Journal on Computing*, 6(2):240–268, 1977.

[67] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[68] Raimund Seidel and Cecilia Aragon. Randomized search trees. In *Proceedings of the 30th Symposium on Foundations of Computer Science*, pages 540–545. IEEE Press, 1989.

[69] Richard P. Stanley and Sergey Fomin. *Enumerative Combinatorics Volume 2*. Cambridge University Press, 2001.

[70] Jiang Tao, Li Ming, and Paul M. B. Vitányi. Average-case analysis of algorithms using Kolmogorov complexity. *Journal of Computer Science and Technology*, 15(5):402–408, 2000.

[71] Tommaso Toffoli. Reversible computing. *Automata, Languages and Programming, Lecture Notes in Computer Science (LNCS)*, 85:632–644, 1980.

[72] Jacinta Townley, Joseph Manning, and Michel Schellekens. Sorting algorithms in MOQA. *Electronic Notes in Theoretical Computer Science*, 225(C):391–404, 2009.

[73] Alan M. Turing. On computable numbers with an application to the Entscheidungs problem. In *Proceedings of the London Mathematical Society*, pages 230–265. LMS, 1937.

[74] Jeffrey Scott Vitter and Philippe Flajolet. Average-case analysis of algorithms and data structures. In *Handbook of Theoretical Computer Science, Volume A*, chapter 9. MIT Press, 1990.

[75] F. Vivarès. *Contribution à la modélisation de méthodes. Application aux méthodes de Jackson.* PhD thesis, École Nationale Supérieure de l'Aéronautique et de l'Espace, Toulouse, France, 1991.

[76] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[77] Lutz Michael Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, 34(4):362–367, 1985.

[78] D. Wooldridge. An algebraic simplify in LISP. Memo 11, Stanford Artificial Intelligence Laboratory, 1963.

[79] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the ACM SIGPLAN / Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 144–153. ACM Press, 2007.

[80] Steven J. Zeil. Selectivity of data flow and control-flow path criteria. In *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Verification and Analysis*, pages 216–222. IEEE Press, 1988.

[81] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. Research Report 1149, Institut National de Recherche en Informatique et en Automatique, 1989.

[82] Wolf Zimmermann. How to mechanize complexity analysis. Technical Report, University of Karlsruhe, 1988.