


Title	Modular average case analysis: Language implementation and extension
Author(s)	Gao, Ang
Publication date	2013
Original citation	Gao, A. 2013. Modular average case analysis: Language implementation and extension. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	© 2013, Ang Gao http://creativecommons.org/licenses/by-nc-nd/3.0/ 
Embargo information	No embargo required
Item downloaded from	http://hdl.handle.net/10468/1089

Downloaded on 2017-02-12T13:10:15Z



UCC

University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Modular Average Case Analysis: Language Implementation And Extension



UCC

Coláiste na hOllscoile Corcaigh, Éire
University College Cork, Ireland

ANG GAO

A thesis submitted to the National University of Ireland, Cork
for the degree of Doctor of Philosophy

Department of Computer Science
National University of Ireland, Cork

Supervisor: Prof. Michel Schellekens

Head of Department: Prof. Barry O'Sullivan

April 2013

Contents

Contents	i
List of Figures	vi
List of Tables	ix
List of Listings	x
Declaration	xii
Dedication	xiii
Acknowledgements	xiv
Abstract	xvi
1 Introduction	1
1.1 Introduction	2
1.2 Research topics	4
1.2.1 <i>MOQA</i> language design and interpreter implementation	4
1.2.2 Parallel Extension of <i>MOQA</i>	5
1.2.3 Exploring <i>MOQA</i> applications	6
1.3 Research contributions	8
1.4 Dissertation structure	9
2 Background	11
2.1 Introduction	12

2.2	Static Average Case Analysis of Algorithms	12
2.2.1	Static Program Analysis	12
2.2.1.1	Data Flow Analysis	12
2.2.1.2	Abstract Interpretation	13
2.2.1.3	Symbolic Analysis	14
2.2.2	Time Analysis of Programs	14
2.2.3	<i>MOQA</i> Approach to Static Average Case Analysis	15
2.3	A Review of <i>MOQA</i> Theory	16
2.3.1	Timing Compositionality	16
2.3.1.1	Worst-Case	17
2.3.1.2	Average-Case	18
2.3.2	<i>MOQA</i> Data Structures	18
2.3.2.1	Partial Orders and Labelled Partial Orders	19
2.3.2.2	Random Structure/Random Bag	22
2.3.2.3	Randomness Preserving	25
2.3.3	<i>MOQA</i> Basic Operations	28
2.3.3.1	<i>MOQA</i> Split	28
2.3.3.2	<i>MOQA</i> Product	31
2.3.3.3	<i>MOQA</i> Top/Bot	36
2.3.3.4	<i>MOQA</i> Percolation	37
2.4	Related work	39
2.5	Summary	43
3	<i>MOQA</i> Language Design	44
3.1	Introduction	46
3.2	<i>MOQA</i> Language Overview	46
3.2.1	Motivations and Design Goals	47
3.2.2	Domain Specific Language	48
3.2.3	Why Python	49
3.2.4	Running <i>MOQA</i>	50
3.2.4.1	Execution Mode	51
3.2.4.2	Analysis Mode	52
3.3	<i>MOQA</i> Language Syntax	54

3.3.1	Lexical Conventions	55
3.3.1.1	Comments	55
3.3.1.2	Identifiers	55
3.3.1.3	Keywords	55
3.3.1.4	Primitive Types	56
3.3.2	Labelled Partial Order	56
3.3.3	<i>MOQA</i> Language Grammar	57
3.3.3.1	Structure of a <i>MOQA</i> Program	58
3.3.3.2	Variable Declarations	58
3.3.3.3	Function Definition	59
3.3.3.4	Expression	60
3.3.4	Scopes	67
3.3.5	<i>MOQA</i> Language Syntax Specification	67
3.4	<i>MOQA</i> Language Type System	69
3.4.1	Type Environments	69
3.4.2	Type Checking Rules	71
3.5	<i>MOQA</i> Language Semantics	78
3.5.1	<i>MOQA</i> Execution Semantics	79
3.5.2	<i>MOQA</i> Analysis Semantics	88
3.6	<i>MOQA</i> Programming Restrictions	97
3.7	Summary	97
4	<i>MOQA</i> Language Implementation	99
4.1	Python Lex-Yacc	100
4.2	<i>MOQA</i> Interpreter Architecture	101
4.3	Lexer	104
4.4	Parser	104
4.5	Environment Implementation	108
4.6	Type Checker	109
4.7	Execution Engine	111
4.8	Analysis Engine	115
4.8.1	Dynamic programming in ACETAnalysis	117
4.9	Summary	127

5	<i>MOQA</i> Programming Examples and Evaluation	128
5.1	Insertionsort	129
5.2	Quicksort	131
5.3	Quickselect	133
5.4	Mergesort	136
5.5	TreapGen	137
5.6	TreapSort	141
5.7	Heapify	143
5.8	Summary	147
 6	 Parallel Extension of <i>MOQA</i>	 148
6.1	Introduction	149
6.2	Related Work	150
6.3	Overview of Multithreaded Program Analysis	151
6.4	<i>MOQA</i> Theory Extension	153
6.5	Experimentation and Evaluating a Practical Example	155
6.5.1	Experiment Result	157
6.6	Summary	160
 7	 Applications of <i>MOQA</i>	 162
7.1	Random Bag Preservation for Heap Algorithms	163
7.1.1	Binary Heap	164
7.1.2	Leonardo Heap	165
7.1.3	Skew Heap	169
7.1.4	Min-Max Heap	172
7.2	Complexity and Entropy based on <i>MOQA</i>	176
7.2.1	Average Case Analysis of Treap Insertion	176
7.2.2	Entropy Analysis based on Random Bags	180
7.2.3	Tracking Entropy Changes in Sorting Algorithms	183
7.3	Reversible Computing for <i>MOQA</i>	189
7.3.1	Background	189
7.3.2	Reversible Split Operation	190
7.3.3	Reversible Quicksort	196

7.4	Smoothed Analysis for \mathcal{MOQA}	200
7.5	Summary	204
8	Conclusions and Future Work	207
8.1	Conclusions	208
8.2	Future Work	210
8.2.1	From An Application Perspective	210
8.2.2	From A Theoretical Perspective	212
A	\mathcal{MOQA} Language	213
A1	\mathcal{MOQA} Language Syntax Specification	214
A2	\mathcal{MOQA} Language Lexer	216
A3	\mathcal{MOQA} Language Parser	219
B		227
B1	Approximation Timing for the Merge Operation	228
	References	230

List of Figures

2.1	An example of a Hasse diagram with node set $X = \{a, b, c, d, e\}$	20
2.2	An example of SP-order	21
2.3	An example of <i>LPO</i> with labelling function $\mathcal{F} = \{a \mapsto 5, b \mapsto 3, c \mapsto 4, d \mapsto 2, e \mapsto 1\}$	22
2.4	An example of order-isomorphic data-labellings with the label set $L = \{1, 2, 3\}$	23
2.5	An example of random structure with the label set $L = \{1, 2, 3, 4, 5\}$	24
2.6	A size four heap random structure with label set $L = \{1, 2, 3, 4\}$	26
2.7	Resulting size three heap random structure	26
2.8	Isolated subsets example	27
2.9	<i>MOQA Split</i> over a <i>LPO</i>	29
2.10	<i>MOQA Split</i> over a discrete random structure $R(\Delta_3)$ with label set $L = \{1, 2, 3\}$	29
2.11	<i>Split</i> on list of size n	31
2.12	Product of two partial orders	32
2.13	Illustration of steps involved in executing the <i>MOQA Product</i> operation	33
2.14	An example of the <i>MOQA Top</i> operation	37
2.15	An example of the <i>MOQA Perc^M</i> operation	38
3.1	sample.moqa output 1	52
3.2	sample.moqa output 2	53
3.3	sample.moqa ACET output	54
3.4	<i>MOQA</i> Language Type System	58

LIST OF FIGURES

4.1	<i>MOQA</i> Interpreter Architecture	102
4.2	<i>MOQA</i> abstract syntax tree class hierarchy	105
4.3	Abstract syntax tree for <i>MOQA</i> Quicksort	106
4.4	Interpreter analysis engine Flowchart	116
4.5	<i>MOQA</i> analysis library class diagram	122
4.6	Series SPPO creation example	124
5.1	<i>MOQA</i> Treap creation example	139
5.2	Treapgen algorithm maps $R(\Delta_3)$ to random bag $TREAP(3)$. . .	140
5.3	Treap sort example on a four element treap.	142
5.4	Heapify example to create a four node heap.	144
6.1	A dag representation of a multithreaded program execution. Each vertex is a strand, edges represent strand dependencies.	152
6.2	A comparison of average time execution between different core amounts and how ACEST and ACEWT bounds the times. Green represents the sequential execution time.	159
6.3	A comparison of the obtained speedup differences for core amounts compared to sequential execution and the theoretically obtained values	160
7.1	Leonardo tree examples: Lt_0, Lt_1, Lt_2, Lt_3	167
7.2	Eight nodes Leonardo Heap example	167
7.3	Leonardo Heap creation example with input List $L = [3, 1, 4, 2]$. .	169
7.4	Leonardo Heap creation: random preserving contour example . . .	169
7.5	Skew Heap creation example: length three random list	172
7.6	Min-Max heap built from input list $[3, 5, 1, 2, 6, 4]$	173
7.7	<i>MOQA Min-Max heapify</i> on input list $[3, 5, 1, 2, 6, 4]$	174
7.8	One possible partial order generated by <i>Min-Max heapify</i>	175
7.9	Result of random insertion on <i>MOQA TREAP(3)</i>	178
7.10	Bottom-up Mergesort example: first five merge operations	185
7.11	Mergesort recursion tree on list size eight	186
7.12	Forward and reverse split on random list $[x, f, m, g, q, s, b, t, p, z]$.	195

LIST OF FIGURES

7.13 <i>MOQA</i> Smoothed Analysis of Quicksort with input list size 50 on different degrees of perturbation	205
---	-----

List of Tables

2.1	worst-case IO-compositionality counter example	17
3.1	<i>MOQA</i> language <i>LPO</i> operations	57
5.1	Insertionsort: comparing the theoretical result with the interpreter result	131
5.2	Quicksort: comparing the theoretical result with the interpreter result	133
5.3	Quickselect: comparing the theoretical result with the interpreter result	136
5.4	Mergesort: comparing the theoretical result with the interpreter result	138
5.5	Treapsort: comparing the theoretical result with the interpreter result	143
5.6	Heapify: comparing the theoretical result with the interpreter result	146
B.1	Times for merging two lists of length n and $n + 1$	228

List of Listings

3.1	Python List Comprehensions	50
3.2	Simple <i>MOQA</i> Code: sample.moqa	51
3.3	Simple <i>MOQA</i> Code: variable declarations	59
4.1	<i>MOQA</i> Language Lexer Fragment	104
4.2	Let expression grammar rule in PLY	107
4.3	<i>MOQA</i> binary operation rule in PLY	107
4.4	Python decorator used for code decomposition	109
4.5	<i>MOQA</i> type checker: Let expression rule	110
4.6	Simplified LPO class Code Fragment	112
4.7	Code for evaluating LPONode	113
4.8	Code for evaluating ReturnExprNode and FuncUseNode	114
4.9	Entry point of analysis engine	116
4.10	doAnalysis Implementation on FuncDefNode	117
4.11	Python memorising decorator	118
4.12	ACETAnalysis implementation	119
4.13	Code Fragment: simplified moqaProcess method on FuncUseNode	120
4.14	Create Discrete Random Structure in Analysis mode	123
4.15	<i>MOQA</i> product operation in analysis mode	125
4.16	<i>MOQA</i> product timing function implementation	126
5.1	<i>MOQA</i> Insertionsort	129
5.2	<i>MOQA</i> Quicksort	131
5.3	<i>MOQA</i> Quickselect	133
5.4	<i>MOQA</i> Mergesort	136
5.5	<i>MOQA</i> Treapgen	138
5.6	<i>MOQA</i> Treaport	141

LIST OF LISTINGS

5.7	<i>MOQA</i> Code: Heapify	143
A.1	<i>MOQA</i> Language Lexer (<code>moqa_tokens.py</code>)	216
A.2	<i>MOQA</i> Language Parser (<code>moqa_grammar.py</code>)	219
A.3	<i>MOQA</i> interpreter environment implementation	225

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not been submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced.

Ang Gao
April 2013.

To my loving parents —Fujin Gao and Ruiling Li
my dear fiancée —Xue Li

Acknowledgements

I would like to express my gratitude to the many people that I have worked with during my PhD program. Thank you for making the time working on my PhD a fulfilling and unforgettable experience.

First of all, I am deeply grateful to my supervisor Michel Schellekens. It's been a great pleasure to work with him. I appreciate all his help, support and guidance throughout the duration of my PhD. His encouragement enabled me to work on diverse exciting projects, his invaluable guidance helped me all through the research and writing of this thesis.

Thanks to my PhD examiners: Peter Puschner and Emanuel Popovici for their time, insight, and helpful comments.

I have been surrounded by wonderful colleagues. The members of the Centre for Efficiency Oriented Languages (CEOL), both past and present, have made our lab a great place to work. Thank you for providing insightful discussions and collaborations on a number of interesting projects. Thanks to Kenneth Rea for collaborating on parallel computing experiments and Diarmuid Early for collaborating on reversible computing projects. Thanks to Caitriona Walsh, who has been very effective in helping with all the administrative work. In particular, I would like to thank Aoife Hennessy, for her support and help with proof reading this thesis. My sincere thanks also goes to Dilip Vasudevan, Prasad Chebolu, Bichen Shi and Guojun Qin for their support and the enjoyable discussions about life.

Also, I am grateful to Derek Bridge for introducing me to the world of

research and helping me publish my first paper during my BSc study in UCC. I thank everybody who in any way supported me during my PhD study.

I would like to thank my fiancée Xue Li for her love, encouragement and faithful support. Last but not least, I thank my parents who raised me to have a keen interest in computer science and research. Thanks to their endless love, encouragement and support throughout my life.

Abstract

Motivated by accurate average-case analysis, *MOdular Quantitative Analysis* (*MOQA*) is developed at the *Centre for Efficiency Oriented Languages* (CEOL). In essence, *MOQA* allows the programmer to determine the average running time of a broad class of programmes directly from the code in a (semi-)automated way. The *MOQA* approach has the property of randomness preservation which means that applying any operation to a random structure, results in an output isomorphic to one or more random structures, which is key to systematic timing.

Based on original *MOQA* research, we discuss the design and implementation of a new domain specific scripting language based on randomness preserving operations and random structures. It is designed to facilitate compositional timing by systematically tracking the distributions of inputs and outputs. The notion of a labelled partial order (*LPO*) is the basic data type in the language. The programmer uses built-in *MOQA* operations together with restricted control flow statements to design *MOQA* programs. This *MOQA* language is formally specified both syntactically and semantically in this thesis. A practical language interpreter implementation is provided and discussed.

By analysing new algorithms and data restructuring operations, we demonstrate the wide applicability of the *MOQA* approach. Also we extend *MOQA* theory to a number of other domains besides average-case analysis. We show the strong connection between *MOQA* and parallel computing, reversible computing and data entropy analysis.

Chapter 1

Introduction

Contents

1.1	Introduction	2
1.2	Research topics	4
1.2.1	<i>MOQA</i> language design and interpreter implementation	4
1.2.2	Parallel Extension of <i>MOQA</i>	5
1.2.3	Exploring <i>MOQA</i> applications	6
1.3	Research contributions	8
1.4	Dissertation structure	9

“It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. [108]” – Alan Turing

1.1 Introduction

Since the outset of computing, people are interested in studying the performance of programs. In 1947 Turing suggested a convenient way to measure the amount of work or performance of a program, by studying the most expensive basic operation(s) involved in a computation, and realistically that’s what we do nowadays. Rather than taking account of every little detail, we focus on the most expensive basic operation(s) and use the estimation result as a proxy for the real running time, essentially, making the hypothesis that the running time will grow as a constant times this estimation result. The work presented in this thesis is based on this idea. *MOQA* [85, 87], developed by Prof M. Schellekens, is a theory and tool to study the average cost of a program by analysing the average number of comparisons involved in a computation. In our research we focus on comparison-based algorithms. This type of analysis can form the basis for a fine-tuned analysis, taking into account other primitive operations such as swaps and assignments [32].

A fundamental problem in computer science is that of writing efficient algorithms and studying their performance. Algorithm analysis is a core computer science area which provides insights in the design of new algorithm and applications. Technological revolution normally comes along with new algorithm design and performance speed up. For example, the Discrete Fourier Transform [104] is used to decompose waveform of N samples into periodic components, the design

of a Fast Fourier transform algorithm improved brute force algorithm, from N^2 steps to $N\log(N)$ steps. As a consequence, it enables new technologies such as DVD, JPEG, MRI, etc [95].

There are three classical types of algorithm analysis: *best-case*, *average-case*, and *worst-case* analysis. In our context, we will focus on *average-case execution time* (ACET) analysis, and the time we refer to is the average number of comparisons made during the execution of an operation. Currently, worst-case analysis is the most widely used methodology to analyse algorithms [94].

However, when the worst case is extremely rare, a worst-case view can be problematic. An algorithm might have acceptable performance on typical inputs while exhibiting poor worst-case time complexity. There are a number of algorithms which are widely used in practice in spite of a very poor worst-case time complexity, notably including Quicksort and the Simplex method [26, 63, 95]. Thus average-case analysis is an important complement to worst-case analysis. Such complementary information can potentially aid better budgeting of resources in a Real-Time context [67]. Smoothed analysis provides an alternative to measure complexity [100]. It is a hybrid of worst-case and average-case analyses, measuring the tipping point at which average-case turns to worst-case behaviour in terms of perturbations of inputs. This in turn enables one to quantify the unlikelihood of worst-case behaviour. We will return to it in Section 7.4.

To get a better sense of a ‘typical’ running time, the average running time over all instances of a problem is often considered. Average-case analysis is particularly relevant in applications with a high degree of randomization in the input data [64].

Average-case time analysis is a notoriously difficult area of Computer Science, not to say measuring average-case execution timing (ACET) automatically. There are a variety of techniques to carry out average-case analysis, but typically they do not allow for automation [34, 53]. Currently algorithms will be analysed on a case-by-case basis and various bottle-neck problems have been highlighted in the literature and some well-known algorithms escape analysis, such as Heapsort and Shellsort [26, 57]. In view of the status of the field, the ultimate aim to provide a unified foundation for average-case analysis motivated the work of many authors including [57, 77, 94, 110].

There are some analysis techniques already developed to tackle automatic

average-case analysis, e.g. [35], but these tend to be quite complex involving many difficult mathematical techniques. *MOQA* [85, 87] is a new method used to obtain average-case timing information, where the underlying mathematical techniques are less complicated than previous approaches. The crucial property of *MOQA* is that operations in this model are randomness preserving, which was captured by the notions of random structures and random bags. This ensures that the data structures in *MOQA* method are traceable, meaning that the probability of each structure occurring at any step during program execution is predictable. Random bags contribute a visual way to represent data and their distributions, which in turn facilitates the modular derivation of ACETs of algorithms.

1.2 Research topics

In this thesis we examine and extend different aspects of *MOQA* from both a practical and a theoretical point of view.

1.2.1 *MOQA* language design and interpreter implementation

The *MOQA* approach consists of designing a new domain specific scripting language based on randomness preserving operations and random structures. It is designed to facilitate compositional timing by systematically tracking the distributions of inputs and outputs. The basic data type in the language is a labelled partial order and the programmer designs an algorithm by applying built-in *MOQA* operations to this partial order, combined with restricted control flow statements. The syntax and semantics of the language are defined in this thesis. This follows closely the first introduced *MOQA* (modular quantitative analysis) language, a so-called efficiency-oriented language defined in the book [85] using mathematical notations.

In this thesis we will study and implement an interpreter for *MOQA* language. Comparing with original *MOQA* language defined in the book [85] the syntax of current *MOQA* language is more similar to an interpreted language

and close to practical applications.

When we evaluate a program using the *MOQA* interpreter, there are two modes built into the interpreter: execution mode and analysis mode. If execution mode is invoked, the designed *MOQA* interpreter will interpret the program by applying *MOQA* operations to the labelled partial order and keep tracking the changes of structure in the execution environment. The result of this type of evaluation is to apply the algorithm to one instance of the problem.

In contrast, analysis mode requires the user to provide an initial partial order size. The interpreter will first build the abstract syntax tree (AST), then will interpret the AST. Using the results from *MOQA* theory research, it will keep track of applying operations over the random structure and of the probability for each structure. By using random structures as representations of output data, it tracks all possible instances of the problem, thus enabling us to derive the average-case execution time (ACET) of the algorithm. The cost of each *MOQA* operation, the resulting random structures and probabilities is a central topic in *MOQA* research. We will discuss this aspect in detail in later chapters. The ACET of one algorithm is derived by accumulating the cost for each operation and the probabilities of the data-occurrences. Finally the interpreter will report back the number of comparisons needed to execute the algorithm. *MOQA* is the first language of its scope to allow for (semi-)automated average-case analysis.

1.2.2 Parallel Extension of *MOQA*

The second topic of this thesis is to extend the usage of *MOQA* to the parallel field.

With the advancement of multi-core processors, parallel algorithms and especially multi-threaded algorithms are of increasing importance to developers. The requirement for parallel algorithm analysis also has increased. In this thesis, we present a new way to analyse fork-join multi-threaded algorithms. These results were reported in [39]. Our approach is based on *MOQA* operations, where we extend the theory to a parallel field and use the traceability of the *MOQA* data structures. We show that multi-threaded algorithms which satisfy *MOQA* theory can be easily analysed. Parallel Quicksort is shown as an example for which

we derived equations for its work and span, something that cannot be achieved by current tools. In this thesis, we also validate our claims by running the parallel program and by comparing the bound predicted by our analysis with real speedup from the experiments. We show that our method is much more accurate than asymptotic analysis [39].

1.2.3 Exploring *MOQA* applications

To show the applicability of the *MOQA* method, we investigate several applications.

The usage of random bags and random bag preservation plays a key role in the research of *MOQA*. Each *MOQA* operation has the property of randomness preservation, which means that applying any operation to a random structure results in an output isomorphic to one or more random structures, which is the key to systematic timing. The notation of a random bag serves as a unifying model to represent data structures and their data distribution. To constructively track the distribution during computations we need to ensure random bag preservation, but this property is not available to all data structuring operations.

Several data constructing operations have been analysed with respect to random bag preservation, such as the construction of binary heaps and binary trees. In this thesis we will study several new type of heap data structures, for instance skew heap, min-max heap, and we will check their random bag preservation properties.

Next, we introduce a new operation: treap insertion. The average-case running time of the operation is discussed ¹.

Because of the usage of random structures to represent *MOQA* operations' inputs and outputs, the analysis of other data structure properties becomes easier. We present some other interesting properties that are not directly related to the running time.

For example, D. Early examined in his thesis the usage of *MOQA* to track second moments [32]. In this thesis we add the capacity to track entropy.

The entropy function is used as a measurement of randomness [69]. It is

¹Joint work with Dr. P. Chebolu

widely applied in information retrieval, machine learning etc [25, 52]. We adapt the definition of entropy and redefine it over a random bag. *MOQA* provides the ability to track the data distribution during computations. We extend this capability by tracking entropy changes during the computation and illustrate this with sorting algorithms such as Insertionsort, Treapsort etc.

We recall that randomness preservation is key to ensuring *MOQA*'s modular timing derivation. A degree of reversibility in turn is a key aspect to ensure randomness preservation. All operations of the *MOQA* language can be made reversible with minimal additional bookkeeping. A challenge in achieving this encoding in a frugal way is to ensure subsets of data can be stored without excessive overheads. This thesis focuses on the joint work [33] to illustrate such an encoding for the case of the Quicksort algorithm. D. Early provided an efficient encoding to reverse the split of a list into two sublists. We study the code for reversible Quicksort and provide a simplified explanation and an example illustrating the algorithm's reverse execution.

The last topic of this thesis is to present how to integrate 'smoothed complexity' in the *MOQA* interpreter. Smoothed analysis is a recent approach to measure how unusual the worst-case running time is [100]. Smoothed analysis considers inputs that are subject to some random perturbation.

The average running time of the algorithm over the perturbations of one input instance is called the smoothed measure of the algorithm over that input instance. The smoothed complexity of the algorithm is the worst smoothed measure of the algorithm on any input instance [100].

The parameter σ is used to measure the degree of perturbation. When σ becomes large, the perturbations on the input become significant, and the smoothed complexity tends towards the average-case running time. On the other hand, as σ becomes small, the perturbations become insignificant on the original instance, and the smoothed complexity tends towards the worst-case running time. Thus, the smoothed complexity is a function of σ which interpolates between the worst-case and average-case running times. The dependence on σ gives a sense of how unusual an occurrence of the worst-case input actually is, thus it can be used to explain why the Simplex method is so efficient [101]. Prior work from M. Schellekens and D. Early [88] has shown the fruitful links between *MOQA* and

smoothed complexity analysis.

In this thesis, we use a smoothed complexity result for Quicksort based on *MOQA* research by M. Schellekens and A. Hennessy [89], and present how to integrate this theoretical result into *MOQA* interpreter and experimentally derive smoothed complexity results using our interpreter analyzer, for different magnitudes of input perturbations.

1.3 Research contributions

This thesis presents a number of contributions involving the development of the *MOQA* language; the interpreter and the theory extension. The most important of these are as follows:

- The *MOQA* scripting language/interpreter is the first practical language/interpreter that provides tracking data structures and probability throughout computations. It is the first language of its scope to allow for (semi-) automated average-case analysis.
- The *MOQA* language syntax and semantics illustrates that *MOQA* research not only provides valuable theoretical results, but also exhibits potential to explore practical implications.
- The interpreter has the ability to show *MOQA* structure status during execution. It might be used as a learning tool for beginners to introduce *MOQA* operations.
- The design and implementation of a *MOQA* interpreter combines a number of Python language tricks, interpreter implementation methods and program analysis techniques.
- We analyse randomness preserving properties of several Heap-algorithms, expanding *MOQA* research to new data structures.
- We present a new way to analyse a multi-threaded fork-join program based on the *MOQA* theory. We expand the *MOQA* modularity theory to

a fork-join model, and show that multi-threaded algorithms which satisfy *MOQA* theory can be easily analysed.

- We investigate the average-case running time of a new operation on the *MOQA* treap data structure, namely, the treap insertion.
- We explore the first automated approach to carry out entropy analysis of various sorting algorithms, potentially linking to other research areas.
- We examine the frugal encoding underpinning the reversible *MOQA* operations, where the encoding can be achieved through the bookkeeping of a single number. The applicability of the encoding has been demonstrated via reversible Quicksort. We provide a simplified explanation and an example illustrates the algorithm's reverse execution.
- We explore the smoothed complexity analysis in the *MOQA* context and integrate the theoretical result with the *MOQA* interpreter.

1.4 Dissertation structure

We provide a brief overview of the structure of this thesis.

Chapter 2. Introduces the *MOQA* theory in some detail, covering the data structures, the means of tracking distributions, details of the *MOQA* operations and their costs, and introduces a modular derivation of ACETs of algorithms.

Chapter 3. Focuses on formally designing the *MOQA* language. The language syntax and operational semantics are presented. The capacity and limitations of the language are also discussed.

Chapter 4. Briefly presents our practical implementation of the *MOQA* language interpreter in Python, especially highlighting methods and tricks we used to program the interpreter.

Chapter 5. Lists several common algorithms implemented in the *MOQA* language. For each algorithm, a theoretical analysis is derived and the interpreter analyzer output is evaluated.

Chapter 6. Expands the *MOQA* modularity theory to a fork-join model, analysing multithreaded fork-join programs based on *MOQA*.

Chapter 7. Discusses several applications related to *MOQA* research, including: exploring new data structures and their randomness preserving property, investigating average-case complexity of treap insertion, tracking entropy changes during sorting algorithms' execution, designing reversible *MOQA* operations and algorithms, giving a brief overview of smoothed analysis in *MOQA* and incorporates it in the interpreter.

Chapter 8. Summarises the results presented in this thesis and discusses the benefits and challenges of *MOQA* research. It also examines directions for future work.

Chapter 2

Background

Contents

2.1	Introduction	12
2.2	Static Average Case Analysis of Algorithms	12
2.2.1	Static Program Analysis	12
2.2.2	Time Analysis of Programs	14
2.2.3	<i>MOQA</i> Approach to Static Average Case Analysis	15
2.3	A Review of <i>MOQA</i> Theory	16
2.3.1	Timing Compositionality	16
2.3.2	<i>MOQA</i> Data Structures	18
2.3.3	<i>MOQA</i> Basic Operations	28
2.4	Related work	39
2.5	Summary	43

2.1 Introduction

In this chapter we present the necessary background information on related research topics. We start with a brief discussion on static average-case analysis, after that some important *MOQA* concepts are introduced. Details of *MOQA* operations' timing functions are recalled.

In Section 2.2 some existing static program analysis techniques are presented and challenges involved in the static average-case analysis of algorithms are discussed. Next, in Section 2.3, we give a review of *MOQA* research, including the definition of a random structure, a random bag, the concept of the randomness preservation property and the *MOQA* modularity theory. Then we introduce some basic *MOQA* operations and their timing functions. Finally we give a short discussion on the *MOQA* language and a chapter summary.

2.2 Static Average Case Analysis of Algorithms

The main applications of computer program analysis are program optimization and program correctness. Computer program analysis is the process of analysing the behaviour and properties of a program. It has two main approaches: static program analysis and dynamic program analysis. In this thesis we focus on static program analysis, that is to analyse the program without executing it in order to derive useful information. Presently, static program analysis is a thoroughly studied area, yet still there are a number of open questions in the field and lots of researchers are involved in tackling them [50, 71, 82, 84].

2.2.1 Static Program Analysis

There are several techniques that are widely used to carry out static program analysis.

2.2.1.1 Data Flow Analysis

Data flow analysis is a technique for gathering information about the possible set of values for variables and expressions which are calculated at various points in

a computer program. It is used for program transformations such as dead-code elimination, common sub-expression elimination and register allocation [61, 71, 103].

In any data flow analysis, the *control flow graph* [71] plays a key role. It is used to determine how far an assignment to a variable could propagate in a program. The compiler often uses this information to do program optimization. A canonical example of a data flow analysis is the live variable analysis for register allocation [13, 15].

Data flow analysis is a very efficient and feasible technique. It is mainly used in compilers to create optimised code. Because it does not use the semantic of the programming language's operators, many interesting program properties cannot be gathered.

2.2.1.2 Abstract Interpretation

Abstract Interpretation is a theory of semantic approximation. It gains semantic information about a program and can be viewed as a partial execution of computer programs [27]. Each programming language has an associated concrete semantics, which defines the effect of each statement and expression as a program is being evaluated. The idea of abstract interpretation is to create a new semantics of the programming language, called abstract semantics, which must be a safe approximation of a concrete semantics and normally is defined to consider only the behaviour relevant to the particular analysis being undertaken. For example, using abstract semantics one can detect some possible semantic errors, such as division by zero [27, 71, 114]. In some sense, our approach to *MOQA*' language analyzer adapts this method. We define two semantics for the *MOQA* language, the execution semantics for the execution mode, and the analysis semantics used in the interpreter analysis mode. The execution semantics can be viewed as concrete semantics, and the analysis semantics plays the role of abstract semantics. The execution semantics is based on labelled partial orders, while the analysis semantics is grounded in random structures and bags.

2.2.1.3 Symbolic Analysis

Symbolic analysis is another technique to achieve static program analysis. It can be viewed as a compiler that translates a program into a different language that consists of symbolic expressions and symbolic recurrences. The symbolic expressions are used to denote the values of a program's computations and variables. It is a technique to derive a precise mathematical characterisation of program properties, and computer algebra systems (such as Mathematica, Maple) play an important role in this technique. Some applications of this type of analysis can be found in [51, 80, 97, 114].

2.2.2 Time Analysis of Programs

There are various approaches to obtain the time complexity of programs. [112] provides a good overview of the techniques used and challenges involved in the timing analysis of programs.

The first and simplest approach is to empirically analyse the time complexity of programs. By choosing a set of sample inputs and executing the program on a specific platform, one estimates the time complexity of the program based on their performance. However this approach has a number of drawbacks. The most obvious one is that the results are platform dependent. For example, one algorithm might perform better than another algorithm only because it used an operation which is optimized for this particular platform. There is no guarantee that this algorithm will outperform the other algorithms on a new platform. In terms of ACET (average-case execution times), empirical analysis normally requires generating a large sample of inputs. It thus might take a significant amount of time to obtain a reasonable approximation of the time complexity. Still there are a number of profilers for the empirical analysis of applications [5, 7].

Another interesting approach is to execute a program in a simulation environment that models the architecture of a particular system. This overcomes the platform dependency problem. An example of using modelling and simulation for WCET (worst-case execution times) analysis can be found in [56].

Instead of relying on these dynamic analysis approaches, static analysis is an alternative method to derive the timing behaviour of a program. Static worst-

case timing techniques have been successfully developed and there are a variety of results ranging from academic approaches [55, 58, 96] to commercial applications such as AbsInt’s WCET analyzers [1]. As discussed in [85], the key principle which lies at the heart of the current development of static worst-case timing tools is a partial compositionality principle. We will review this principle in Section 2.3.1.

In this thesis we are concerned with the automatic derivation of the ACETs of programs. As will be shown, there are lots of challenges involved, some of which are well known [34, 57] and are encountered in existing WCET and ACET tools, and others which are specific to applying our approach to calculating ACETs automatically.

2.2.3 *MOQA* Approach to Static Average Case Analysis

At this stage there are no widely used static average-case analysis tools available. Industry relies on simulation (empirical analysis of programs on large sample input sets) to derive information about the average-case behaviour of their products. The drawbacks are obvious. First of all, this approach suffers from imprecision as sample spaces are not necessarily representative of the whole possible input space. A second issue is that simulation steps take a long time because of the size of the sample space. Normally we need to invest more time and hence higher cost to gain a higher precision.

This approach affects both software and hardware analysis. *MOQA* provides a basis for novel modular static analysis to address this issue. The *MOQA* approach is based on average-case timing compositionality, which means that the average-case timing of a program is simply the sum of the times of the parts. This is a very helpful advantage for static analysis, something which is not available in current languages. But this average-case summation property does not “come for free”. We need to be able to track data and their distribution throughout computations. In a nutshell, this tracking is achieved through a representation of the distribution combined with a data structure representation, referred to as a *random bag* (a multiset). Through a careful design of the basic operations one ensures that such representations are preserved throughout the computations. This property is referred to as *random bag preservation*.

2.3 A Review of \mathcal{MOQA} Theory

In this section, we give a brief overview of \mathcal{MOQA} theory, the background presented in this section is based on [32, 85].

\mathcal{MOQA} is a data-structuring language and purposely designed to enable a (semi-)automated derivation of the average-case execution time. Average-case timing compositionality lies at the heart of this approach. It can rightfully be referred to as the “golden key” to static analysis, witnessed by its central role in static worst-case time analysis. We begin by introducing this concept.

2.3.1 Timing Compositionality

Recall that in this thesis we are focusing on comparison-based algorithms. Static timing in our context reflects the number of comparisons during the execution of comparison-based algorithms. For a comparison-based algorithm P , $T_P(I)$ refers to the timing (number of comparisons) of algorithm P executed on input instance I . $P(I)$ denotes the output for this algorithm with input I . Considering an input set \mathcal{I} , $P(\mathcal{I})$ is the multiset (in \mathcal{MOQA} we refer to it as a bag) of outputs produced by P , acting on each element of \mathcal{I} .

Definition 2.1. The *worst-case time* of P for inputs from \mathcal{I} , denoted by $T_P^W(\mathcal{I})$ is defined by:

$$T_P^W(\mathcal{I}) = \max\{T_P(I) \mid I \in \mathcal{I}\}.$$

The *average-case time* of P for inputs from \mathcal{I} , denoted by $\bar{T}_P(\mathcal{I})$ is defined by:

$$\bar{T}_P(\mathcal{I}) = \frac{\sum_{I \in \mathcal{I}} T_P(I)}{|\mathcal{I}|}.$$

Definition 2.2. Assume that we have two algorithms, P_1 and P_2 , for which the sequential execution $P_1; P_2$ is carried out.

Given a timing complexity T (which may be of worst-case, average-case), for input set \mathcal{I} . We say that T is **IO-compositional** if we have:

$$T_{P_1; P_2}(\mathcal{I}) = T_{P_1}(\mathcal{I}) + T_{P_2}(P_1(\mathcal{I}))$$

We say that T is lower (upper) IO-compositional if in the equation above $T_{P_1;P_2}(\mathcal{I})$ is less (greater) than or equal to the right-hand side.

2.3.1.1 Worst-Case

Clearly the worst-case timing measure T^W is lower IO-compositional, because the time taken to sequentially execute $P_1; P_2$ cannot take longer than worst case time for P_1 plus the worst case time for P_2 . However we will use the following counter example to show that worst-case is not IO-compositional.

Remark 2.1. A function or algorithm P with domain (input) X and codomain (output) Y is denoted by $P : X \rightarrow Y$.

Example 2.1. Suppose algorithm $P_1 : X \rightarrow Y$, $P_2 : Y \rightarrow Z$, where $Y \subseteq Z$. In the table below we list input/output pairs for both algorithms and for the composed algorithm with their time costs.

input	output	time
a	γ	1
b	α	6
c	β	3
d	γ	1

(a) Algorithm P_1

input	time
α	2
β	8
γ	9
δ	10

(b) Algorithm P_2

input	time
a	10
b	8
c	11
d	10

(c) Algorithm $P_1; P_2$

Table 2.1: worst-case IO-compositional counter example

In our example, $X = \{a, b, c, d\}$ and $Y = \{\alpha, \beta, \gamma, \delta\}$. Clearly it can be seen from the example that $T_{P_1}^W(X) = 6$, $T_{P_2}^W(Y) = 10$, $T_{P_1;P_2}^W(X) = 11$, thus $T_{P_1;P_2}^W(X) < T_{P_1}^W(X) + T_{P_2}^W(Y)$. If we restrict P_2 to outputs from P_1 , we get $T_{P_2}^W(P_1(X)) = 9$, which still gives $T_{P_1;P_2}^W(X) < T_{P_1}^W(X) + T_{P_2}^W(P_1(X))$. The reason for this is because when P_1 reaches a worst-case time (on input b), P_2 will execute quite fast on the output generated by P_1 . In fact, the input which gives the worst-case running time for $P_1; P_2$ (on input c) is not a worst-case input for P_1 , nor is $P_1(c) = \beta$ a worst-case input for P_2 (even among the set of outputs from P_1).

In order to make the equality hold in the equation, we need that when P_1 reaches a worst-case on a particular input, say x , then P_2 also reaches its worst-case on the output $P_1(x)$. As illustrated by the example, this is not the case in general.

Remark 2.2. Notice that the work by Burns and Puschner [76] does explore a compositional approach to worst-case time in a real-time context. Their work forces the time to be constant by forcing conditional statements to execute on both branches. As a result, they establish a restricted real-time language with respect to which the worst-case time is IO-compositional. However, as the authors of [76] point out, this approach can suffer from a drastic increase in execution time.

2.3.1.2 Average-Case

Theorem 2.1. *The average case timing measure is IO-compositional.*

We refer the reader to [85] for the proof of this theorem. Here we illustrate this theorem building on Example 2.1.

Example 2.2. The outputs for P_1 are $P_1(X) = \{\gamma, \alpha, \beta, \gamma\}$, the average running time for P_1 is $\bar{T}_{P_1}(X) = \frac{1+6+3+1}{4} = \frac{11}{4}$. Restricting P_2 to outputs of P_1 , the average running time is $\bar{T}_{P_2}(P_1(X)) = \frac{9+2+8+9}{4} = \frac{28}{4}$. The average running time for sequentially executing P_1 and P_2 is $\bar{T}_{P_1;P_2}(X) = \frac{10+8+11+10}{4} = \frac{39}{4}$, thus $\bar{T}_{P_1;P_2}(X) = \bar{T}_{P_1}(X) + \bar{T}_{P_2}(P_1(X)) = \frac{39}{4}$.

The property of IO-compositionality may seem to make average-case analysis even easier than worst-case analysis. However it is not immediately helpful in practice, because the average time for the second algorithm P_2 depends on the outputs and the distributions of outputs of the first algorithm P_1 . Generally, without execution, we do not have knowledge of the outputs from P_1 , not to say the associated probability distribution. *MOQA* addresses this issue by introducing random bags and the randomness preservation property. We will examine these concepts in the coming sections.

2.3.2 *MOQA* Data Structures

As mentioned earlier, *MOQA* is a data-structuring language designed to facilitate compositional timing by systematically tracking the distributions of inputs and outputs. This means that the average running time of any *MOQA* program

can be expressed as a linear combination of the average running times of a handful of basic operations. With the help of this modularity feature, *MOQA* allows for the automated extraction of expected running times (in terms of number of comparisons) directly from the source code. In Chapter 3 and Chapter 4, we will discuss the design and implementation details of the *MOQA* language and interpreter.

MOQA operations act on objects known as labelled partial orders. *MOQA* operations transfer each labelled partial order to a new labelled partial order.

2.3.2.1 Partial Orders and Labelled Partial Orders

Definition 2.3. A *partially ordered set* (or poset) is a pair (X, \sqsubseteq) consisting of a set X ¹ and a binary relation \sqsubseteq between elements of X . If $(a, b) \in \sqsubseteq$, we write $a \sqsubseteq b$, and we refer to a as a descendant of b , and b as an ancestor of a . Such that the relation is:

- Reflexive: $\forall x \in X, x \sqsubseteq x$
- Transitive: if $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$
- Anti-symmetric: if $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$

The elements of X are called the nodes of the partial order [29, 85]. For example if we have a three elements set $X = \{a, b, c\}$, and a binary relation $\sqsubseteq = \{(a, a), (b, b), (c, c), (a, c), (b, c)\}$, then we can say (X, \sqsubseteq) is a partial order.

A partially ordered set is generally represented by a *Hasse diagram*. A Hasse diagram is a directed graph whose nodes are the nodes of the poset. We draw a line segment that goes upward from x to y whenever $x \sqsubseteq y$ and there is no z such that $x \sqsubseteq z \sqsubseteq y$ (in this thesis we omit arrows in the line). In short, the Hasse diagram of a partial order is a digraph representation of its transitive reduction which discards all reflexive pairs and pairs that can be inferred by transitivity from \sqsubseteq . Later, we will use this diagram to represent labelled partial orders.

Example 2.3. We give an example of partially ordered set (X, \sqsubseteq) where $X = \{a, b, c, d, e\}$ and $\sqsubseteq = \{(a, a), (b, b), (c, c), (d, d), (e, e), (b, a), (c, a), (d, a), (e, a), (d, c), (e, c)\}$. And we show its Hasse diagram in Figure 2.1.

¹In our context we assume that all the sets are finite

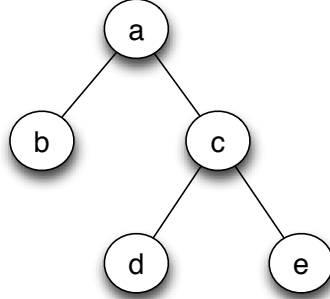


Figure 2.1: An example of a Hasse diagram with node set $X = \{a, b, c, d, e\}$

Currently, all the partial orders involved in \mathcal{MOQA} research are *Series-Parallel Partial Orders* (or SP-orders). Even though \mathcal{MOQA} operations have been defined over general partial orders [85], we focus on SP-orders because \mathcal{MOQA} operations’ timing functions are recursively defined over SP-orders [85]. SP-orders form a well-known class of computationally tractable data structures [70] and include normal data structures such as Tree, Heap, List, etc. A series-parallel partial order (SP-order) is a partial order that can be recursively constructed by applying the functions “series” and “parallel” starting with a single node [102]. We introduce these operations below.

Definition 2.4. Given two disjoint SP-orders (A, \sqsubseteq_1) and (B, \sqsubseteq_2) .

The series operation produces the partial order $A \otimes B$ on $A \cup B$ such that $x \sqsubseteq y$ in $A \otimes B \Leftrightarrow [x, y \in A \text{ and } x \sqsubseteq_1 y] \text{ or } [x, y \in B \text{ and } x \sqsubseteq_2 y], \text{ or } [x \in A \text{ and } y \in B]$

The parallel operation produces the partial order $A || B$ on $A \cup B$ such that $x \sqsubseteq y$ in $A || B \Leftrightarrow [x, y \in A \text{ and } x \sqsubseteq_1 y] \text{ or } [x, y \in B \text{ and } x \sqsubseteq_2 y]$.

Note: $A \otimes B$ is the result of applying the series operation to (A, B) , which is the same as the partial order created by applying the \mathcal{MOQA} Product operation to (A, B) [85, 86]. This operation places A below B and will be introduced in Section 2.3.3.2.

Example 2.4. The SP-order $(b || (f \otimes (d || e) \otimes c)) \otimes a$ is shown in Figure 2.2.

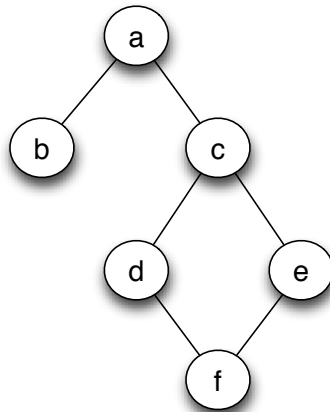


Figure 2.2: An example of SP-order

Theorem 2.2. *SP-order preservation [32, 85]: all the current MOQA operations are SP-preserving; that is, if a MOQA operation acts on inputs whose underlying posets are in SP-order then the outputs will also have underlying posets in SP-order.*

This result holds since we generally assume that any MOQA program starts from a *discrete partial order* where no two distinct nodes have a relation (referred by Δ , which is in SP-order), and all MOQA programs rely on basic MOQA operations to build up a data structure from the initial discrete partial order. Thus while programming in MOQA, all data structures must be in SP-order. So, it is sufficient to determine the running times based on inputs from SP-orders, and we can specify the running times of MOQA operations in terms of series and parallel recursion over the SP-orders.

Definition 2.5. A *Labelled Partial Order* (or *LPO*) is a triple $(X, \sqsubseteq, \mathcal{F})$, where (X, \sqsubseteq) is a poset and \mathcal{F} is an increasing function from X to some totally ordered set L (referred as the *label set*). For any node $x \in X$, $\mathcal{F}(x)$ is called *the label* on the node x . \mathcal{F} is called the labelling of the poset [32, 85].

Remark 2.3. As usual in the analysis of algorithms, to simplify the analysis, we assume that there are no repeated labels in the *label set*. The techniques for dealing with repeated labels in MOQA are discussed in [32, 85].

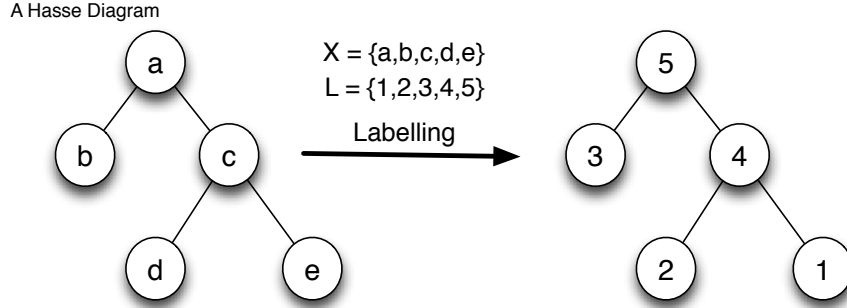


Figure 2.3: An example of *LPO* with labelling function $\mathcal{F} = \{a \mapsto 5, b \mapsto 3, c \mapsto 4, d \mapsto 2, e \mapsto 1\}$

Remark 2.4. *MOQA* computations transform *LPOs* to new *LPOs*.

The *LPOs* in *MOQA* are represented by Hasse diagrams, the value appearing in each node is the label assigned to it according to the labelling function \mathcal{F} . A *LPO* is simply an assignment of a finite number of values to each node of a partial order (data structure).

The requirement that the labelling function increases is equivalent to requiring that any directed links of the data structure are respected, i.e. if there is a directed link from a node x to a node y , then the label assigned to x must be less than the label assigned to y (place x below y). These labels can be any value, e.g. natural or real numbers, words, other data structures containing data such as trees, etc. Any two labels need to be comparable with respect to a given order on labels. For instance, the order on natural number labels typically is the usual order on natural numbers.

There is an example of a data labelling in Figure 2.3. The partial order is represented by a Hasse diagram, where the label set $L = \{1, 2, 3, 4, 5\}$, the labelling function $\mathcal{F} = \{a \mapsto 5, b \mapsto 3, c \mapsto 4, d \mapsto 2, e \mapsto 1\}$

2.3.2.2 Random Structure/Random Bag

A partial order (data structure) may have infinitely many labellings because the label set is infinite. For example, consider a discrete partial order with 3 nodes, where no two distinct nodes have a relation. We represent it using Δ_3 . Even though there can be infinitely many data-labellings, Δ_3 has finitely many states.

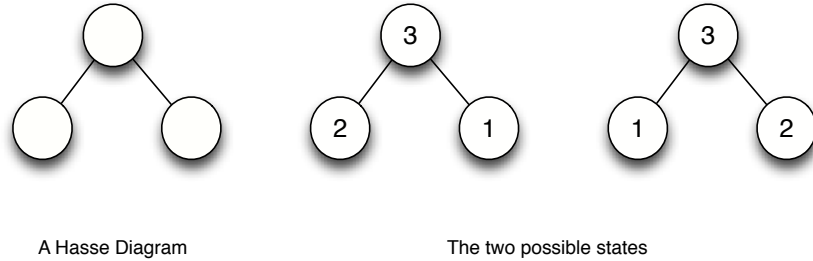


Figure 2.4: An example of order-isomorphic data-labellings with the label set $L = \{1, 2, 3\}$

Definition 2.6. A state is a representative from a collection of *order-isomorphic data-labellings*, i.e. data-labellings whose labels are arranged in the same relative order within the partial order (data structure).

Example 2.5. We illustrate this further with the data-labellings for the 3 element wedge-shaped partial order (\wedge) in Figure 2.4. If we use three distinct values, say 1, 2, 3 to represent the states then we have only two possible states as displayed in the figure.

To aid the analysis of average-case complexity, generally we fix the label set. In our Δ_3 example, if we fix the label set $L = \{1, 2, 3\}$, Δ_3 will have $3!$ possible labellings: $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, $(3, 2, 1)$. And we refer to this set of all possible *LPOs* as a *Random Structure*.

Definition 2.7. Given a partial order (X, \sqsubseteq) , the random structure over a given label set L is $R_L(X)$ which is the set of all possible *LPOs* $(X, \sqsubseteq, \mathcal{F})$ with respect to the order-isomorphic data-labellings.

Remark 2.5. For simplicity, sometimes we refer to a partial order as a random structure without specifying a labelling set. In that case, the labelling default set is $L = \{1, 2, 3, \dots, n\}$ where n is the number of nodes in the partial order.

Example 2.6. The random structure over Δ_3 is presented as:

$$R_L(\Delta_3) = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$$

where $L = \{1, 2, 3\}$. Sometimes we simply write $R(\Delta_3)$.

For the partial order shown in Figure 2.3, the random structure with the label set $L = \{1, 2, 3, 4, 5\}$ is shown in Figure 2.5.

Notice the result of one data restructuring operation may result in multiple random structures, thus a *Random Bag* is used.

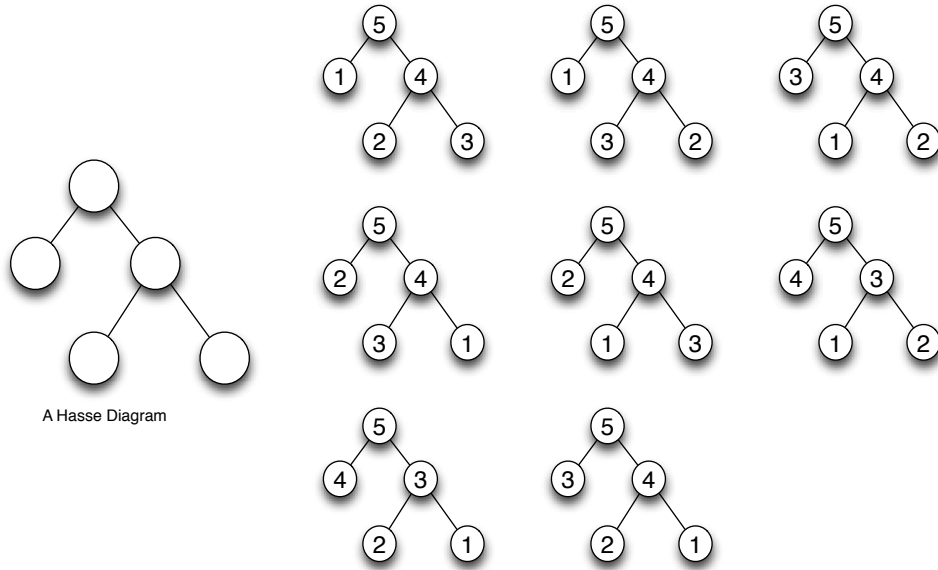


Figure 2.5: An example of random structure with the label set $L = \{1, 2, 3, 4, 5\}$

Definition 2.8. A random bag is a multiset of random structures, it generally represents by $\{(R_1, K_1), \dots, (R_n, K_n)\}$. It consists of finitely many random structures, R_1, \dots, R_n , each of which has a *multiplicity* K_i , where $i \in 1, \dots, n$.

Remark 2.6. The multiplicity is a natural number used in the calculation of the probabilities involved in the distribution.

With the help of random bags, we can calculate the probability of each *LPO* in the bag. This information enables us to derive the average-case complexity of a program.

Theorem 2.3. A *LPO* F in bag R_i , where $R = \{(R_1, K_1), \dots, (R_n, K_n)\}$ has the following probability:

$$Prob[F \in R_i] = \frac{K_i |R_i|}{\sum_{i=1}^n K_i |R_i|} = \frac{K_i |R_i|}{|R|}$$

where $F \in R_i$ indicates a specific LPO in the bag belonging to the random structure R_i , and $|R_i|$ is the number of LPOs in R_i . $|R|$ is the number of all LPOs in the bag.

2.3.2.3 Randomness Preserving

MOQA focuses on a special kind of operation, that is *Randomness Preserving* (or random bag preserving).

Definition 2.9. An operation is random bag preserving if and only if the operation maps input random bags to output random bags.

We refer the reader for a more formal definition in [85]. Operations which are random bag preserving enable the tracking of data structures and their distributions, which in turn is directly linked with the capacity to generate recurrence equations expressing the average-case number of basic operations. The multiplicities of the random bags are the key to the calculation of the ACETs. Notice that not every data restructuring operation is random bag preservation, e.g. the delete operation in Heapsort.

Example 2.7. Here we show that not all data restructuring operations are random bag preserving. We use the heap delete operation as an example. Say we have a four nodes heap as shown in Figure 2.6. The left most structure represents the partial order of the four node heap, the other three LPOs together form a random structure $R(H_4)$ with label set $L = \{1, 2, 3, 4\}$. We apply the standard Heapsort delete maximum operation to these LPOs, that is we swap the maximum element with the last element, after which the last element will be at the top, and the maximum element will be placed last. Then we remove the maximum element and call a push down at the top of the heap.

The resulting three-nodes LPO is shown in Figure 2.7. The first structure is the resulting partial order, and the remaining three are LPOs. As one can see, the first two LPOs form a three nodes heap random structure, however the remaining LPO on its own does not form a random structure. Thus the output of this operation cannot form a valid random bag. The delete operation is not random bag preserving. In [91] an algorithm called percolating heapsort is

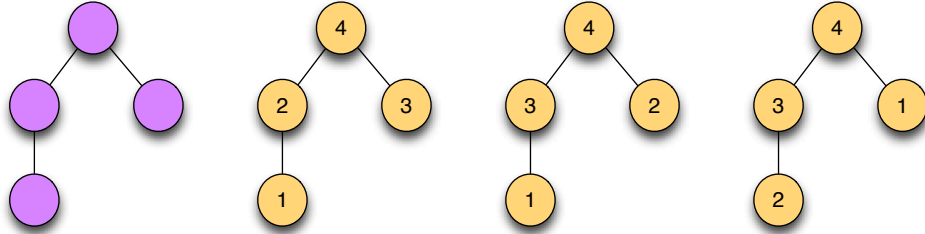


Figure 2.6: A size four heap random structure with label set $L = \{1, 2, 3, 4\}$

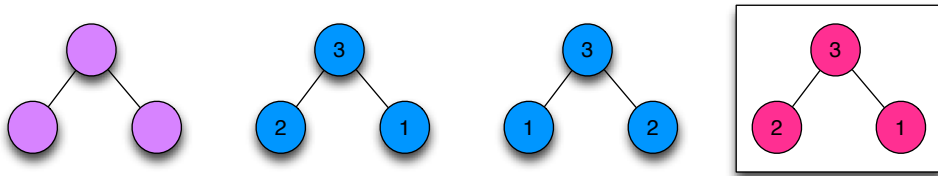


Figure 2.7: Resulting size three heap random structure

presented which has a specially designed delete operation, which has the random bag preserving property.

We also note that random bag structures are preserved on certain substructures. We refer to these substructures as *Isolated subsets*.

Definition 2.10. With technical omission, an informal definition of an *Isolated subset* I of X , for a given random structure $R(X, \sqsubseteq)$, is that, a nonempty subset I of X is isolated iff $\forall x \in X - I$ exactly one of three conditions holds:

- x is below (\sqsubseteq) every element of I .
- x is above (\supseteq) every element of I .
- x is independent (not \supseteq or \sqsubseteq) of every element in I .

Remark 2.7. This is a simpler but exact definition of Isolated subsets. We refer the reader to [32, 85] for the formal definition.

It has been proven in [85] that the restriction of all data-labelings of a random structure to an isolated subset forms a new random structure.

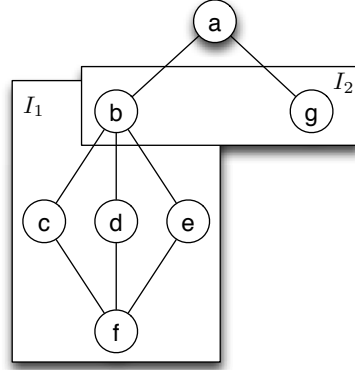


Figure 2.8: Isolated subsets example

Example 2.8. In Figure 2.8, I_1 is an example of an isolated subset, whereas I_2 is not an isolated subset. For the set I_2 , the elements c, d, e are all below b , but none of these are below the other I_2 element g .

The backbone theorem in \mathcal{MOQA} research is the *Linear-Compositionality Theorem* [85]. We outline it in the following theorem. It states two facts. Firstly, the average time of the sequential composition of two random bag preserving programs can be expressed as the sum of the individual parts. Secondly, the average time of a random bag preserving program on a random bag is the summation of the average times over the random structures in the random bag. The cardinality and multiplicity of a random structure together determines its probability. We will extend this theorem to the parallel field in Chapter 6.

Theorem 2.4. Consider random bag preserving programs/operations P and Q , such that we execute P on a random bag R , producing random bag R' .

- The ACET of executing P following by Q is:

$$\bar{T}_{P,Q}(R) = \bar{T}_P(R) + \bar{T}_Q(R')$$

- Consider random bag $R = \{(R_1, K_1), \dots, (R_n, K_n)\}$, then:

$$\bar{T}_P(R) = \sum_{i=1}^n Prob_i \times \bar{T}_P(R_i)$$

where $Prob_i = Prob[F \in R_i]$ which is defined in Theorem 2.3.

- For the particular case where $R = \{(R_1, K_1)\}$, the previous equality reduces to:

$$\overline{T}_P(R) = \overline{T}_P(R_1).$$

2.3.3 \mathcal{MOQA} Basic Operations

In the following section, we give a brief overview of basic \mathcal{MOQA} operations. For the details and theoretical proofs for the randomness preservation and timing functions we refer the reader to [32, 85].

The operations we present in this thesis are enough to cover most of the algorithms in the current \mathcal{MOQA} research. All these operations will be discussed again in later chapters to explore their application in algorithms and implementations in the \mathcal{MOQA} language/interpreter. For each operation, we will consider both application on a single LPO and on a random structure. The first case is used in interpreter execution mode, to get results for one particular input instance. The second case is used by the interpreter analyzer to derive the ACET. Notice that there are other \mathcal{MOQA} operations, such as *Projection*, \overline{Del} and \underline{Del} . They are currently not available to the \mathcal{MOQA} interpreter, thus we will not cover them in this thesis.

2.3.3.1 \mathcal{MOQA} Split

Firstly, we focus on a simple ‘randomness preserving’ operation: *Split*. The classical algorithms Quicksort and Quickselect are both based on a *Split* operation, which takes a list and a pivot (which is an element of the list) as arguments. We use a simpler version to reduce technicalities. The pivot for split is chosen to be the first element of the list. This choice is irrelevant. Other choices will result in similar random structures with minor technical modifications.

Split proceeds on a list of size n by comparing, in left to right order and starting at the second element, each label of the i^{th} element, $i \in \{2 \dots n\}$, with the pivot label. In cases where the label of the i^{th} element is greater than the pivot label, these elements and their labels are placed above the pivot. Otherwise

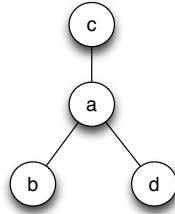


Figure 2.9: *MOQA Split* over a *LPO*.

they are placed below the pivot. In fact, the classical *Split* puts the i^{th} element to the left or the right of the pivot. The *MOQA Split* however puts it below or above the pivot, a minor technical difference. Due to the explicit representation of the order via a Hasse diagram.

Example 2.9. For example, if we have a random list a, b, c, d , one possible *LPO* for this list is shown in Figure 2.9. In this example $a > b$, $a > d$ and $a < c$.

Remark 2.8. In the following context, we will call the upper part Y_1 (consisting of the elements above the pivot), the middle pivot Y_2 and bottom part Y_3 (consisting of the elements below the pivot).

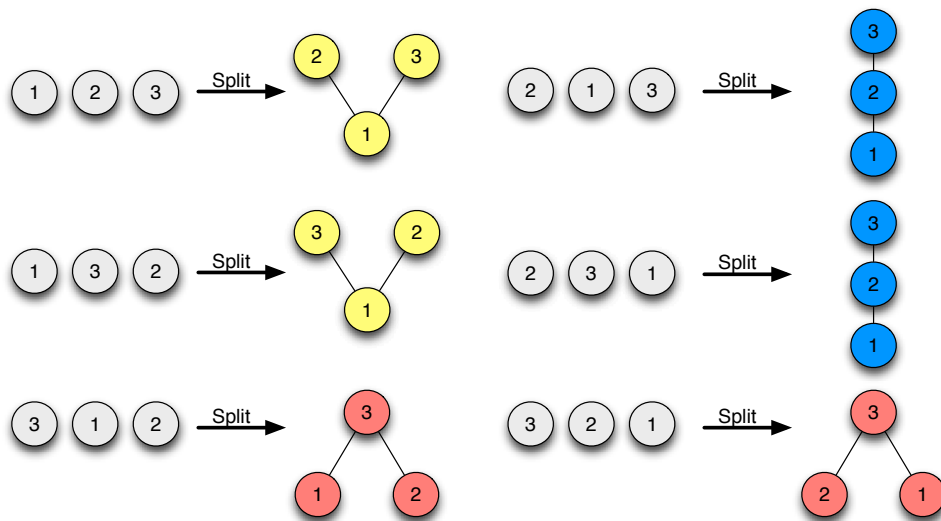


Figure 2.10: *MOQA Split* over a discrete random structure $R(\Delta_3)$ with label set $L = \{1, 2, 3\}$

Example 2.10. A more complete example is shown in Figure 2.10. It applies *Split* over a discrete random structure $R(\Delta_3)$, where we use the label set $L = \{1, 2, 3\}$ to simplify the example.

It is clear from the above example that when *Split* is executed on the random structure over the discrete partial order of size 3, i.e. $R(\Delta_3)$, where *Split* is executed over the $3! = 6$ random lists, the result is a random bag consisting of three new random structures: the 3 element V-shaped partial order, denoted by \vee_3 ; the linear order of size 3, denoted by \mathcal{S}_3 and the 3 element wedge-shaped partial order, denoted by \wedge_3 . Thus *Split* transforms a labelling over Δ_3 into a labelling over \vee_3, \mathcal{S}_3 or \wedge_3 . We conclude that:

$$R(\Delta_3) \mapsto \{(R(\vee_3), 1), (R(\mathcal{S}_3), 2), (R(\wedge_3), 1)\}$$

Hence *Split* is a random bag preserving operation over the random structure $R(\Delta_3)$.

Remark 2.9. We remark at this stage that there is a clear visual nature to the partial orders (data structures) associated with the random bag. Indeed, “star-like” objects are being created with the pivot as the central element, and for each case a collection of elements above the pivot and below the pivot.

The result of the operation can be generalized to n elements as follows. The partial order $P[i, j]$ over $i + j + 1$ elements is defined to be the structure which has one central pivot element, i elements below the pivot and j elements above the pivot, as shown in Figure 2.11.

Theorem 2.5.

$$Split : R(\Delta_n) \mapsto \{(R(P[0, n-1]), K_{n-1}), \dots, (R(P[n-1, 0]), K_0)\}$$

and where $K_i = \binom{n-1}{i}$ for $i \in \{0, \dots, n-1\}$

$$\overline{T}_{split}(R(\Delta_n)) = n - 1$$

For the details of the proof we refer the reader to the Springer book [85].

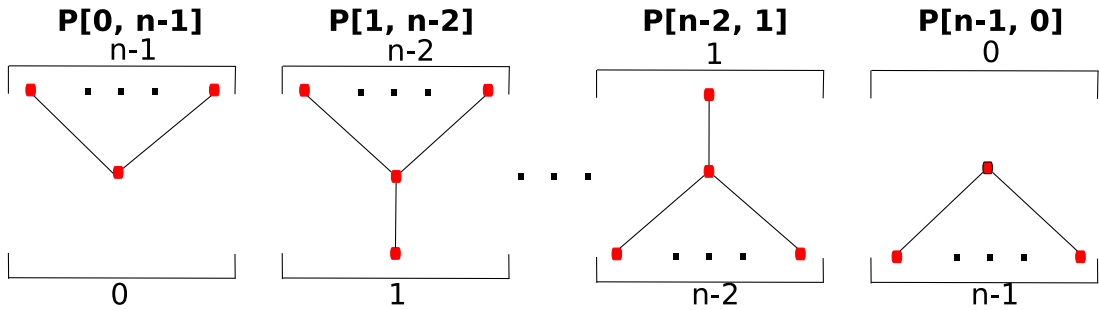


Figure 2.11: *Split* on list of size n

2.3.3.2 *MOQA Product*

The *Product* operation has two operands. If one of the operands is a single element, it plays the role of an insertion of a single element into a data structure. This operation also plays a crucial role to merge two data structures into a larger structure.

Given two *LPOs*, the binary *Product* operation places the first data structure below a second. By using *Push-Up* and *Push-Down* it makes all elements of the first order strictly below all elements of the second. The operation proceeds as follows [85]

- create a new partial order consisting of the union of the elements of the original two orders
- create all possible directed links from the maximal elements of the first order to the minimal elements of the second order.
- respect the new order by reorganizing labels via traditional *Push-Downs* and *Push-Ups*.

The *Push-Up* operation repeatedly swaps a label with the smallest label on the nodes immediately above it until all the nodes above the current node have a label larger than the current node. The formal definition of this operation can be found in [85]. *Push-Down* is the dual of *Push-Up*, which swaps a label with the largest label on the nodes immediately below it until all the labels on the nodes immediately below the current node are smaller.

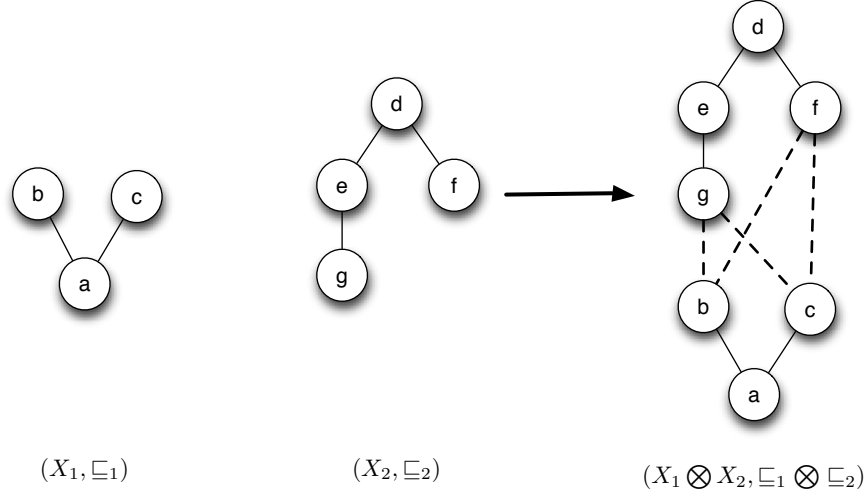


Figure 2.12: Product of two partial orders

Firstly, we define the *Product* of two finite partial orders and provide an example.

Definition 2.11. Given two finite *disjoint* partial orders (X_1, \sqsubseteq_1) and (X_2, \sqsubseteq_2) , i.e. partial orders for which $X_1 \cap X_2 = \emptyset$.

The set $X_1 \otimes X_2$ is defined to be the union of the disjoint sets X_1 and X_2 . The relation $\sqsubseteq_1 \otimes \sqsubseteq_2$ is defined to be the least partial order on $X_1 \otimes X_2$ containing \sqsubseteq_1 and \sqsubseteq_2 and $X_1 \times X_2$ [85].

Example 2.11. If we consider the sets $X_1 = \{a, b, c\}$ and $X_2 = \{d, e, f, g\}$ then $X_1 \otimes X_2 = \{a, b, c, d, e, f, g\}$. We use dashed lines to indicate the new pairs of relations added in the Hasse diagram via the operation \otimes , and the example is shown in Figure 2.12.

Next, we introduce the *Product* of two *LPOs* as an introduction towards the definition of the random product of two random structures.

Let F_1, F_2 be *LPOs* on finite partial orders (X_1, \sqsubseteq_1) and (X_2, \sqsubseteq_2) respectively. We present the result of *Product* F_1, F_2 by $F_1 \otimes F_2$.

The details of this operation and the proof of the following lemma follows via technical verification from the *MOQA Product* operation. We omit the details and again refer the reader to [85]

Lemma 2.6. *If F_1 and F_2 are disjoint LPOs then $F_1 \otimes F_2$ is a LPO.*

Example 2.12. Figure 2.13, taken from [85], illustrates the product of two LPOs F_1 and F_2 . Their partial orders are displayed at the top of the figure. This example illustrates the steps involved in executing the *MOQA Product* operation over two LPOs.

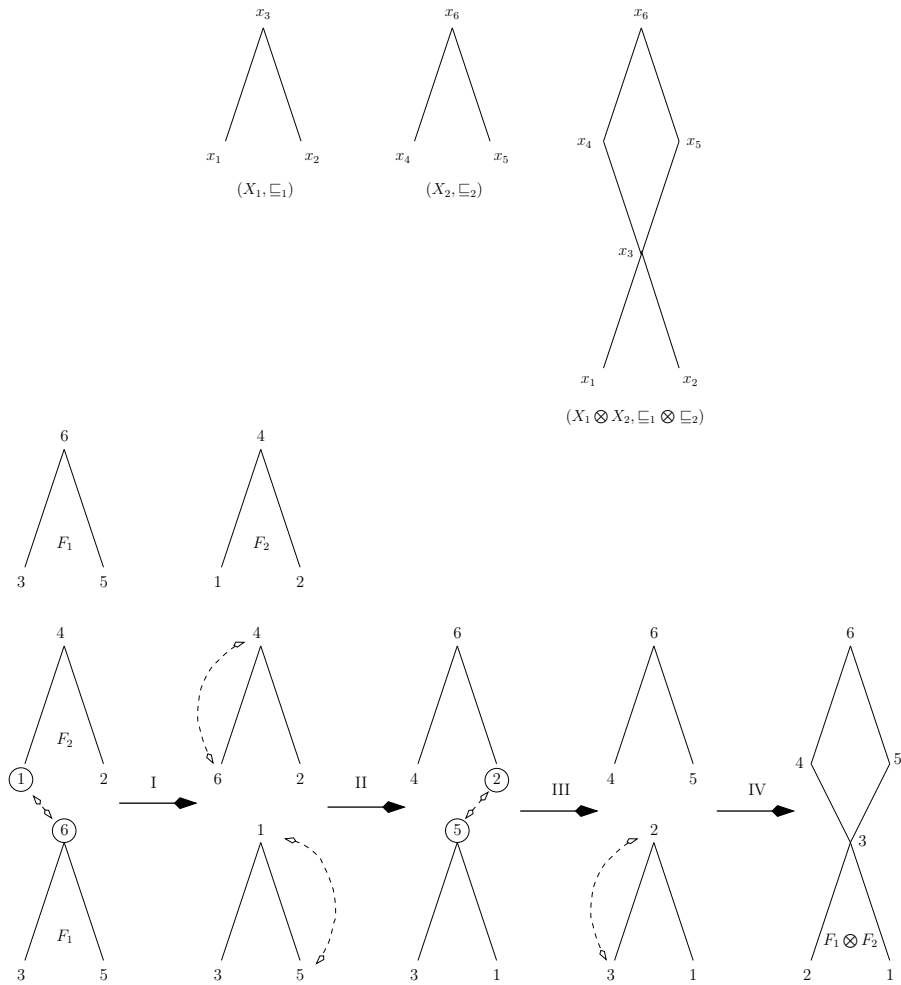


Figure 2.13: Illustration of steps involved in executing the *MOQA Product* operation

Finally, with omission of detail of the *MOQA* theory of extending the *MOQA* labelling *Product* to a product over random structures, we introduce the following definition,

Definition 2.12. Let $R_{L_1}(X_1, \sqsubseteq_1)$ and $R_{L_2}(X_2, \sqsubseteq_2)$ be two disjoint random structures. We define the *binary MOQA Product*, $R_{L_1}(X_1, \sqsubseteq_1) \otimes R_{L_2}(X_2, \sqsubseteq_2)$, by $R_{L_1 \cup L_2}(X_1 \otimes X_2, \sqsubseteq_1 \otimes \sqsubseteq_2)$.

Before we exit this section, we want to explore the ACET timing function for the *MOQA Product* operation, and we will use these formulas in our code analyzer to derive the ACET of *MOQA* codes. The work presented here reports research results from D. Early and M. Schellekens. We refer the reader to [32, 85] for technical details.

Theorem 2.7. *The average running time of the MOQA Product operation on the partial orders A and B is*

$$\bar{T}[A \otimes B] = \frac{|A||B|}{|A| + |B|} (\tau_{down}(A) + \tau_{up}(B)) + \left(\frac{|A||B|}{|A| + |B|} + 1 \right) (|A_{max}| + |B_{min}| - 1).$$

Proof. Here we adapt the original proof and give an intuitive proof on why this formula works. The details of original proof can be found in [85]. There are two helper functions in the formula: $\tau_{down}(A)$ and $\tau_{up}(B)$. We will introduce them in the following context. Recall that the *Product* operation makes all elements of the first order (A) strictly below (or smaller than) all elements of the second (B).

The operation is executed as follows: first it compares the minimal label in B with the maximum label in A . If the minimal label in B is greater than the maximum label in A , then the operation finishes. Otherwise, we swap the minimum label in B with the maximum label in A , and, following the swap operation, we execute Push-Up on the minimal node in B and Push-Down on the maximal node in A to move the swapped new label to the proper location. And we repeat this step until all elements of the first order (A) are strictly below (or smaller than) all elements of the second (B).

$\tau_{down}(A)$ is the average number of comparisons to Push-Down from the node with minimum label after swapping a new label (randomly). And $\tau_{up}(B)$ is the dual of $\tau_{down}(A)$, which calculates the average number of comparisons to Push-Up from the node with maximum label after swapping a new label (randomly). $\frac{|A||B|}{|A|+|B|}$ in the formula defines the average number of this minimum-maximum

label swap which occurs, thus $\frac{|A||B|}{|A|+|B|}(\tau_{down}(A) + \tau_{up}(B))$ is the average number of comparisons made by Push-Up and Push-Down in *Product* operation. Note that there are an extra $(|A_{max}| + |B_{min}| - 1)$ comparisons each time we do a swap, because we need to find the maximum and minimum label in both partial orders. A_{max} means the set of maximal elements in A , B_{min} means the set of minimal elements in B . $|A_{max}|, |B_{min}|$ calculate their cardinality respectively. Thus $\left(\frac{|A||B|}{|A|+|B|} + 1\right) (|A_{max}| + |B_{min}| - 1)$ is the total number of comparisons made by comparing the maximal label in A with the minimal label in B , where $\frac{|A||B|}{|A|+|B|}$ is the average occurrence of swaps. To confirm the operation is complete, one more minimum-maximum label comparison is needed.

After combining these two parts we get the desired formula. \square

We recall that an introduction to series-parallel orders (SP-orders) is included in Section 2.3.2.1. The timing function for τ is defined in terms of SP-orders [85].

Theorem 2.8. *Series-Parallel Composition Laws for the τ function:*

The trivial composition laws for $|A|$, $|A_{min}|$ and $|A_{max}|$ are as follows:

1. $|(A \otimes B)| = |(A||B)| = |A| + |B|$
2. $|(A \otimes B)_{min}| = |A_{min}|$
3. $|(A \otimes B)_{max}| = |B_{max}|$
4. $|(A||B)_{min}| = |A_{min}| + |B_{min}|$
5. $|(A||B)_{max}| = |A_{max}| + |B_{max}|$

Using only the values of these functions applied to A and B , τ functions for a SP-order can be calculated in terms of its values for the constituent parts as follows, where κ and σ are helper functions [85].

1. $\tau_{up}(A \otimes B) = \frac{|A|\tau_{up}(A) + \kappa_{up}(A)|B_{min}| + |B|(\tau_{up}(B) + |B_{min}| + \sigma_{up}(A))}{|A| + |B|}$
2. $\sigma_{up}(A \otimes B) = \sigma_{up}(A) + \sigma_{up}(B) + |B_{min}|$
3. $\kappa_{up}(A \otimes B) = \kappa_{up}(B)$

$$4. \tau(A||B) = \frac{|A|\tau(A) + |B|\tau(B)}{|A| + |B|}$$

$$5. \sigma(A||B) = \frac{|A|\sigma(A) + |B|\sigma(B)}{|A| + |B|}$$

$$6. \kappa(A||B) = \kappa(A) + \kappa(B)$$

The first three rules are stated only for the ‘up’ cases, but the ‘down’ ones are similarly obtained. The last three rules are symmetrical for the ‘up’ and ‘down’ versions, and the subscripts have been omitted.

Using these rules, it is possible to determine the τ function for a SP-order using only its series-parallel composition. The base case values for these functions are listed below:

$$1. \tau_{up}(\bullet) = \tau_{down}(\bullet) = 0$$

$$2. \sigma_{up}(\bullet) = \sigma_{down}(\bullet) = 0$$

$$3. \kappa_{up}(\bullet) = \kappa_{down}(\bullet) = 1$$

$$4. |\bullet| = |\bullet_{min}| = |\bullet_{max}| = 1$$

Example 2.13. To illustrate how all these timing functions work together, we compute $\bar{T}[A \otimes B]$, where $A = \bullet$, $B = (\bullet||\bullet) \otimes \bullet$. A is a single element, B is a \wedge shape random structure. $\tau_{down}(A) = \tau_{down}(\bullet) = 0$. Because $\tau_{up}(\bullet||\bullet) = \frac{|\bullet|\tau_{up}(\bullet) + |\bullet|\tau_{up}(\bullet)}{|\bullet| + |\bullet|} = 0$, $\kappa_{up}(\bullet||\bullet) = \kappa_{up}(\bullet) + \kappa_{up}(\bullet) = 2$, $\sigma_{up}(\bullet||\bullet) = \frac{|\bullet|\sigma_{up}(\bullet) + |\bullet|\sigma_{up}(\bullet)}{|\bullet| + |\bullet|} = 0$ thus $\tau_{up}((\bullet||\bullet) \otimes \bullet) = \frac{|\bullet||\bullet|\tau_{up}(\bullet||\bullet) + \kappa_{up}(\bullet||\bullet)|\bullet_{min}| + |\bullet|(\tau_{up}(\bullet) + |\bullet_{min}| + \sigma_{up}(\bullet||\bullet))}{|(\bullet||\bullet)| + |\bullet|} = \frac{0+2+1}{3} = 1$.

1. According to Theorem 2.7:

$$\begin{aligned} \bar{T}[A \otimes B] &= \frac{|A||B|}{|A|+|B|}(\tau_{down}(A) + \tau_{up}(B)) + \left(\frac{|A||B|}{|A|+|B|} + 1\right)(|A_{max}| + |B_{min}| - 1) \\ &= \frac{3}{1+3}(0 + 1) + \left(\frac{3}{1+3} + 1\right)(1 + 2 - 1) = \frac{17}{4} \end{aligned}$$

2.3.3.3 *MOQA* Top/Bot

Besides *Split* and *Product* operation, *MOQA* incorporates a number of other operations, such as *Top* and *Bot* to determine the minimum and maximum labels in a discrete order, all of which are random bag preserving.

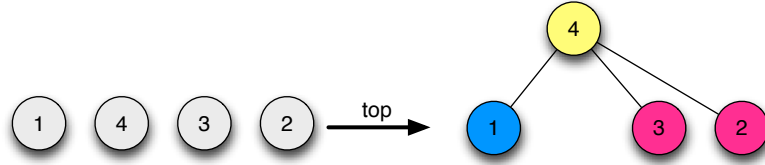


Figure 2.14: An example of the $MOQA$ *Top* operation

Motivated by the desire to create max/min treaps, the $MOQA$ *Top/Bot* operation is introduced. A brief introduction to the $MOQA$ treap is presented in Section 5.5.

Top/Bot takes a single LPO as an argument and turns the node with largest/smallest label into a maximum/minimum node. Like the *Split* operation, the average-case times for these two operations are $n - 1$ if the input structure is Δ_n .

To aid the creation of treaps, *Top/Bot* also keeps track of the node order where the maximum/minimum label occurs. For a discrete order of size n , it puts the node i containing the maximum/minimum label above/below all other nodes, and the rest of the nodes are put into two components in parallel: one containing the nodes $1 \dots (i - 1)$ (left component) and the other contains $(i + 1) \dots n$ (right component).

We will discuss the usage of this operation in Chapter 5, especially how to use it to create treaps in the $MOQA$ language. Here we use the *Top* operation as an example in Figure 2.14 to illustrate the usage of these two operations in practice. The two parallel components are coloured differently.

2.3.3.4 $MOQA$ Percolation

The last $MOQA$ operation we will introduce is called *Percolation*. During this thesis we will only focus on the $Perc^M$ operation. It acts on the maximum label in an LPO . We refer the reader to [85] for the symmetry operation; $Perc^m$ which acts on the minimum label.

The $Perc^M$ operation is executed as follows:

- Find the maximum label in an LPO , flag that label as minimum label over the label set. (eg: add a '-' sign)

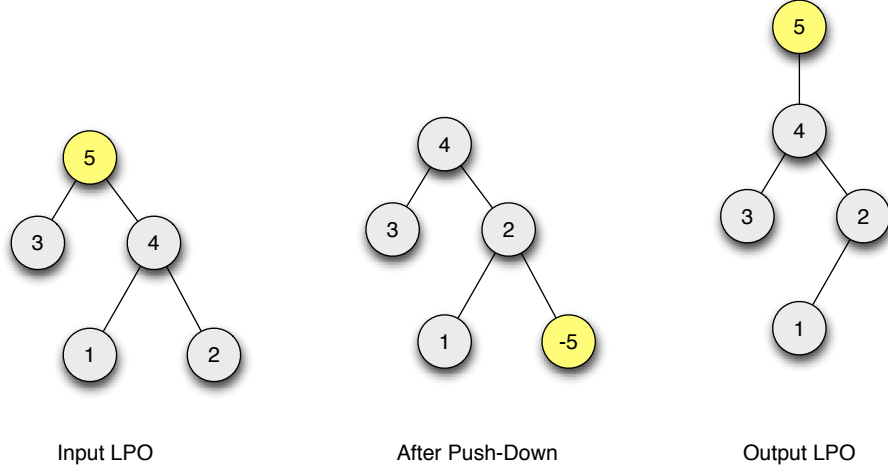


Figure 2.15: An example of the $MOQA Perc^M$ operation

- Call push-down on this node, which will move it to a minimal node.
- Remove the flagged node (eg: has a '-' sign).
- Reinsert the node with the original maximum label, placing the rest of the nodes below it.

Theorem 2.9. *The average number of comparisons $\Omega(A, k)$ required in deleting a k -th smallest label from one LPO, by percolating it to a minimum node and then removing this node (corresponding to Del in [85]) is defined over SP-orders (in [85] Ω is denoted as Δ):*

1.
$$\Omega(A \otimes B, k) = \begin{cases} \Omega(A, k) & \text{for } k \leq |A| \\ \Omega(B, k - |A|) + |A_{max}| - 1 + \Omega(A, |A|) & \text{for } k > |A| \end{cases}.$$
2.
$$\Omega(A \otimes B) = \frac{|A|\Omega(A) + |B|(\Omega(B) + |A_{max}| - 1 + \Omega(A, |A|))}{|A| + |B|}.$$
3.
$$\Omega(A \| B, k) = \frac{\sum_i \binom{k-1}{i-1} \binom{|A|+|B|-k}{|A|-i} \Omega(A, i) + \sum_i \binom{k-1}{i-1} \binom{|A|+|B|-k}{|B|-i} \Omega(B, i)}{\binom{|A|+|B|}{|A|}}.$$
4.
$$\Omega(A \| B) = \frac{|A|\Omega(A) + |B|\Omega(B)}{|A| + |B|}.$$
5.
$$\Omega(\bullet) = 0.$$

Remark 2.10. The average number of comparisons made by $Perc^M$ is the same as the number of comparisons made by Del in deleting the largest label. In that case k is equal to the size of LPO (since the two operations are identical except $Perc^M$ has re-insertion at the end).

Theorem 2.10. *The average number of comparisons needed to perform $Perc^M$ in a size n $MOQA$ treap data structure is:*

$$h(n) = h(n - 1) + \frac{2}{n} = 2H_n - 2$$

where H_n is the n^{th} Harmonic number [85].

Theorem 2.11. *The $MOQA$ $Perc^M$ operation, when executed on all treaps from a n element random treap $TREAP(n)$, returns n copies of the random structure $TREAP(n - 1) \otimes \bullet$ [85].*

In this section we introduced all the $MOQA$ operations we will use later in our $MOQA$ language. Other issues related to language structures, such as control flow, data types etc will be discussed in detail in Chapter 3 and Chapter 4.

2.4 Related work

In this section, we provide a brief overview and discussion of various prior approaches to automated complexity analysis, and a short discussion on motivation for a stand alone $MOQA$ language.

Automated program complexity analysis has undergone active research. Several approaches have been proposed to tackle this problem. In the functional programming languages area, a good number of complexity analysis frameworks have been devised, the main ones of which we discuss below.

ACE (automatic complexity evaluator) proposed in [65], uses a set of rewrite rules to transform functional language FP programs in order to derive the complexity function. A library of known recursions is used to derive closed forms for some recursion equations.

Abstract interpretation and program transformation techniques are used in [83], which proposes a system to derive a time bound function (or worst-case

complexity) in a first-order subset of Lisp programs.

An algorithm is introduced in [45] to extract cost recurrences from Dependent ML (DML) programs. A cost recurrence describes an upper bound for the running time of a program in terms of the size of its input. DML is an extension of ML, that provides dependent types. With DML types, data can be associated with size information, thus describing a possible abstraction.

In [81], dynamic parallelization decisions of a program are made with guidance from the inferred cost estimates. It uses an effect type system for automatically inferring cost estimates of functional programs. The system mainly focuses on the programs written with combinators such as map and fold. The cost system produces symbolic cost expressions that contain free variables describing the size and cost of the program's input. At run-time, the system dynamically computes a cost estimate from the statically determined cost expression combined with run-time cost and size information.

Besides these solutions, several other good approaches are also reported in [18, 76, 111] to cover imperative languages automated analysis field.

[76] provides a “compositional” approach to worst-case time in the real-time context. They remove non-determinism by forcing a single-path programming style, that is, by forcing conditional statements to execute on both branches. As a result, they establish a restricted real-time language with respect to which the worst-case time is IO-compositional. However, as the authors of [76] point out, predictability is achieved at the cost of performance.

A framework for providing portable WCET analysis for the Java platform is given in [18]. The portable WCET analysis is achieved by analysing Java Byte Code, not high-level Java source code. Thus other JVM languages could be analysed in this framework. The WCET analysis is separated into a machine independent part and a machine dependent part and they are staged in three steps: a Java virtual machine platform dependent (low-level) analysis, a software dependent (high-level) analysis and an on-line integration step.

The paper [111] uses integer linear programming techniques to estimate the WCET for programs by determining their worst-case path. The abstract interpretation is also used to predict the system's behaviour on the underlying processor's components.

Most of the work we have discussed so far focused on WCET. Comparing with WCET tools, there are fewer frameworks/tools for the automated ACET analysis of programs. Generally, this is because ACET analysis is even more complicated than WCET and partly as a result, the demand for ACET is not as high as to determine upper bounds for hard real-time systems.

The system Lambda-Upsilon-Omega(LUO) [35, 109] obtains average-case complexity functions for functional programs. Their approach involves generating functions [43]. It enables the automatic derivation of the average-case complexity of large classes of algorithms by producing generating functions while skipping the intermediate step of generating recurrences. LUO code is programmed in a restricted programming language called the Algorithm Description Language (ADL). The system can produce generating functions or calculate a final result with the help of the Maple system [3]. A related and generalized approach involves the use of attribute grammars [66]. This approach generalized univariate generating functions produced by LUO to multivariate generating functions.

The Performance Compiler tool developed in [49] demonstrates the work on semantics of probabilistic programs in [59] and correctness of performance annotated programs in [78] which can be used to automate the average-case analysis of simple programs. The programs that this system could analyse contain language construct such as assignments, conditionals, and loops. Their work could handle recursion and complex data structures, and the distributions on complex data structures are captured by using attributed probabilistic grammars (APGs).

The *MOQA* approach to automated average-case analysis differs from these prior work in that *MOQA* provides a novel compositionality for the average-case analysis. It also provides the ability to handle dynamic data structuring unlike LUO [35, 109]. The usage of random structure and random bag preserving operations in *MOQA* simplifies the average-case time analysis. The program data structure distributions are tracked throughout the computations.

We discuss the existing work in our research group that is closely related to this thesis. The first approach to a *MOQA* language implementation as a Java library was discussed in [107]. The package implements most of the *MOQA* data structures and basic operations that are specified in [85].

A static automated average-case analysis tool Distritrack has been developed

in [48], based on the Soot package. It statically analyses the *MOQA* codes written in those packages. It allows for sophisticated tracking of data structures over programs. With the help of Mathematica this tool can automatically derive average running times for a variety of *MOQA* programs.

And the pure theoretical work from [85] and [32] lays out a solid foundation for the *MOQA* research. They provide a detailed analysis of *MOQA* operations and their timing functions.

This thesis consists of three main parts. In Chapters 3 – 5, we will present a prototype of the *MOQA* language with new syntax and design purpose. The language relies on *MOQA* theory and supports (semi-)automated average-case analysis.

Chapter 6 focuses on an extension of *MOQA* in the parallel computing field and presents a new way to analyze fork-join multithreaded algorithms.

We explore several applications related to *MOQA* research in Chapter 7. The chapter starts with several new types of heap creation algorithms and focuses on their randomness preservation. The new *MOQA* treap insert operation is also discussed and the timing function is derived. Besides these, and building upon random bag preserving operations and random structures, reversible computing and entropy analysis are discussed. Finally, we briefly mention recent research on smoothed complexity and how it links with *MOQA* and can be incorporated in our language interpreter.

The design of a new syntax and making *MOQA* a stand alone language is not simply an integration work of language package [107] and analyzer [48]. We approach the work from a different angle and take advantage of the fact that we have the full control of the interpreter and language syntax, which, as a result, makes automated average-case analysis more straightforward.

The eagerness for the design of a stand alone *MOQA* language can be seen from the following two quotes in [48] and [32]:

“...more accuracy would be guaranteed for all programs and a simpler analysis would be possible if MOQA had its own dedicated language. [48]”

“Ideally, we can imagine a situation where a MOQA compiler

would read a piece of code, analyse the running times of the operations,.... [32]”

2.5 Summary

In this chapter we discussed several concepts related to this thesis and provided a brief overview of *MOQA* theory. We started with a general introduction to static average case analysis and some commonly used technologies. Then we briefly discussed how the *MOQA* approach provides a means for automating average-case analysis.

After that we presented an introduction to *MOQA* theory, mainly focusing on key ideas such as: random structure, random bag, randomness preservation and the *MOQA* modularity theory. We also discuss the basic *MOQA* operations that we implemented in our language.

Finally, several related work that provide automated WCET and ACET analysis were discussed. The *MOQA* approach, based on tracking random structures, is quite different from prior approaches. In later chapters we will show how *MOQA* works. Besides these, we briefly talked about other work that has been done in our research group and the motivation for the new work contained in this thesis.

Chapter 3

MOQA Language Design

Contents

3.1	Introduction	46
3.2	<i>MOQA</i> Language Overview	46
3.2.1	Motivations and Design Goals	47
3.2.2	Domain Specific Language	48
3.2.3	Why Python	49
3.2.4	Running <i>MOQA</i>	50
3.3	<i>MOQA</i> Language Syntax	54
3.3.1	Lexical Conventions	55
3.3.2	Labelled Partial Order	56
3.3.3	<i>MOQA</i> Language Grammar	57
3.3.4	Scopes	67
3.3.5	<i>MOQA</i> Language Syntax Specification	67
3.4	<i>MOQA</i> Language Type System	69
3.4.1	Type Environments	69
3.4.2	Type Checking Rules	71
3.5	<i>MOQA</i> Language Semantics	78
3.5.1	<i>MOQA</i> Execution Semantics	79

3.5.2	<i>MOQA</i> Analysis Semantics	88
3.6	<i>MOQA</i> Programming Restrictions	97
3.7	Summary	97

3.1 Introduction

In this chapter we discuss the design of the domain specific language *MOQA*. We provide the specification for the *MOQA* language in a formal way. In Chapter 4, based on this specification, a Python implementation is discussed.

We start with a general introduction to the *MOQA* language in Section 3.2. In this section, we first discuss the motivation and design goals of our language, then introduce basic concepts of domain specific languages and of our implementation language Python. Then a short overview on the capability of the current *MOQA* interpreter is presented. The purpose of this section is to give the reader a general idea on the *MOQA* language and its syntax.

Section 3.3 focuses on syntax aspects of the language, such as lexical conventions and language constructs etc. We discuss the language grammar in this section and how a *LPO* is represented in our language.

Next, in Section 3.4, we show the design of the type system in our language and provide typing rules as the basis for implementing a type checker.

Then in Section 3.5, we focus on two semantics of the *MOQA* language, one for execution mode, and one for analysis mode. In this section we also present how timing functions are integrated into our analyzer.

In Section 3.6, we discuss some general restrictions on our language to enable automated average-case analysis, e.g. discard while statement, restricted control flow etc. Finally we give a short summary of our language design.

3.2 *MOQA* Language Overview

MOQA is a static type-checked, imperative programming language, which supports automated average-case analysis. It can be viewed as a domain specific language, specially designed for (semi-)automated average-case analysis.

In this section, we give an overview of our design of the domain specific *MOQA* language. Practical examples will demonstrate the capability and usage of this language.

3.2.1 Motivations and Design Goals

Prior projects within CEOL group showed the running time of any *MOQA* programs can be determined in a static way in terms of the running times of basic *MOQA* operations.

[107] implements *MOQA* structures and basic operations as a Java library, and [48] provides a static automated average-case analysis tool *Distritrack* that analyses the *MOQA* codes written in those packages. These two projects work well together, but there are several problems:

- The *MOQA* library provides programmers with a suite of data structures and operations, but there is no way to prevent arbitrary control flow or if-expressions that do not obey *MOQA* restrictions.
- *Distritrack* relies on the design of a *MOQA* Java library. Adding features to any part needs updating of the other components. It makes the tools hard to extend, e.g. adding new data structure representations.
- *Distritrack* can only analyse algorithms that involve recursive data structures, e.g. *Heapify* is not analysable in *Distritrack*.
- The time is output as a function to a *Mathematica* package by *Distritrack*. It is a double-edge sword. Users could obtain a general timing equation, but it makes low interactivity between users and the tools. Users don not get instant feedback. Also some timing equations are hard to read and in practice, future algorithms might produce an equation that is hard to express in *Distritrack* or for which there is no solution for the recursion.
- The original code analysis phase is complex and there is no formal description.

Attempting to overcome these problems, we develop a new domain specific language. The aim to design *MOQA* as a domain specific language for automated average-case analysis, based on *MOQA* randomness preserving operations and structures. We summarize a set of core language design goals for the *MOQA* language:

-
- *MOQA* theory abstraction
 - The *MOQA* language provides a toolbox to manipulating *LPOs* using built-in *MOQA* basic operations together with restricted control flow constructs.
 - The back-end *MOQA* analysis is hidden from the programmer.
 - The usage of the *MOQA* programming language minimizes the time needed from *MOQA* concept to concrete working programs.
 - Automated average-case analysis
 - Based on the usage of randomness preserving operations over abstract data types (labelled partial orders), and the tracking of data distributions throughout computation.
 - The language supports automated average-case analysis for a wide range of algorithms, such as sorting or searching.
 - High level of readability
 - The *MOQA* language aims to be a clear and concise programming language.
 - Its syntax is inspired by modern dynamic languages, e.g.: Python, Ruyg, JavaScript etc.
 - User friendly
 - It is designed to be picked up easily by a programmer who knows basic imperative languages such as Java, Python etc.
 - It can be used as a tool for programmers to practice and learn *MOQA* theory.

3.2.2 Domain Specific Language

A Domain Specific Language (DSL) is a small but expressive programming language that is custom designed for a specific task [37]. In our case, the *MOQA*

language is a domain specific language that is specially designed for automated average-case analysis.

Like other DSLs, *MOQA* does not aim to be as powerful as a general-purpose language like C or Java. It is limited in scope and capabilities. A DSL is generally simple and concise, as its name suggest. It is designed to focus on a certain type of problem or domain. *MOQA*, based on randomness preserving operations, is aimed to facilitating average-case analysis. The *MOQA* language can be interpreted as a suite of data restructuring operations operating on labelled partial orders together with specifically designed control flow language constructs.

The key feature that makes *MOQA* distinct from its predecessor [35, 49, 66, 109], is that *MOQA* provides a novel timing compositionality, rooted in the notion of random bag preservation. The foundation for the new approach are abstract data types (labelled partial orders) and their associated random bag preserving operations. *MOQA* also provides the ability to handle traditional data structures, such as lists, binary trees and heaps, while other approaches generally do not support these high level structures. As a result, *MOQA* solution to automated average-case analysis stays more closely to traditional programming practice.

As we will also show in the later sections, the usage of random structures and random bag preserving operations in *MOQA* simplifies the way to approach average-case time analysis. By walking over the abstract syntax tree and tracking data structure distributions throughout the computations, the ACET for a program is derived in a more straightforward manner.

3.2.3 Why Python

In our interpreter implementation we choose the Python programming language. Python is an object oriented, imperative, dynamically-typed language. It was originally developed by Guido van Rossum at CWI in the Netherlands in the 1980s. It is widely used in different computing areas, such as dynamic web applications, natural language processing tasks, or scientific computing etc [12].

In this language, we do not have to deal with the overhead of static typing, while in a statically-typed language eg. in Java, we have to pay special attention

to the type system. As a result, Python is a proper choice for system prototyping.

Besides this, Python has a powerful and wide range of built-in data types such as: lists, tuples, sets and dictionary etc. Also Python is famous for its clear and expressive syntax. Here we use Python list comprehensions to illustrate these features.

Listing 3.1: Python List Comprehensions

```
1 even = [x for x in range(10) if x % 2 == 0]
2 # even = [0, 2, 4, 6, 8]
```

List comprehensions are used in Python to construct lists. As can be seen from Listing 3.1, the first line builds a list containing even numbers from 0 to 9 inclusive, in a very natural and easy way. The syntax is like mathematical notation, while in other languages, such as Java, we have to write a for loop to iterate and accumulate the result.

Similarly with other programming languages, Python is not perfect and it has drawbacks. It is said to be slow, especially in a parallel field due to the global interpreter lock [11]. In our context, we are only concerned with CPython, a Python language implemented in C. JVM-based Python, such as Jython, does not have this problem. Since we do not use parallel features in Python, this problem does not affect our implementation.

In our project, because we are more concerned with productivity i.e. the capacity to write concise code and short development cycle than efficiency at the current stage, we choose Python as our implementation language for rapid prototyping. We also use Python features to build our interpreter, but the concepts and methods are compatible with other languages and can be converted to Java or C/C++ if needed.

3.2.4 Running *MOQA*

As mentioned earlier, there are two modes in the *MOQA* interpreter. The first mode is called execution mode. It behaves like a normal interpreter, evaluates source code and produces a value or performs side effect (print or showing *LPO*)

for one particular instance of a problem. The second mode is called analysis mode. This mode first extracts all the functions defined in the source code, then invokes the built-in analyzer to do automated average-case analysis for these functions. With augmented initial partial order size supplied by the user, the interpreter produces a summary of ACETs for functions defined in the source code.

The program you write should be placed in a plain text file. By convention, the `.moqa` suffix is used to denote programs written in valid *MOQA* syntax. Let's start with execution mode.

3.2.4.1 Execution Mode

This section only provides a glimpse of *MOQA* programming, the language specification is given in Section 3.3, which will explain in sufficient detail along with the main concepts of the language. After that, more practical examples are given in Chapter 5, including examples of well-known algorithms for sorting, searching, data structures and their average-case analysis by the interpreter analyzer.

In *MOQA* programming, the basic building block is the *LPO*. In a *LPO*, each node has an associated label. Currently, we support string, integer and floating point as label type.

Listing 3.2: Simple *MOQA* Code: `sample.moqa`

```
1 let lpo = {2,1,3,5,0,7,6} /* create initial LPO */
2
3 def fun1(X) /* define a dummy function */
4   let pivot = X[0] /* store first element in pivot */
5   X >< pivot /* split X using variable pivot */
6   show(X) /* visualize LPO */
7   return X /* return resulting LPO */
8 end
9
10 show(lpo)
11 fun1(lpo) /* function call */
12 print(lpo) /* textual presentation of a LPO */
```

In Listing 3.2 we give a simple *MOQA* program, called `sample.moqa`. It involves function definition, *LPO* creation, function call etc.

In this example, the first line creates a discrete partial order *LPO* labelled with $\{2, 1, 3, 5, 0, 7, 6\}$. The following lines 3 – 8 define a function called `fun1`. This function applies the *Split* operation to the input *LPO*, with the first element as pivot, then shows the resulting *LPO*. The next three lines, 10 – 12, execute function calls to `show`, `fun1`, and `print` respectively.

Consider the following command:

```
1 - $ python moqa.py sample.moqa
```

It invokes the *MOQA* interpreter execution mode on `sample.moqa`. The script file `moqa.py` is our interpreter entry point. In a Unix/Linux system, if we set `moqa.py` executable, we can even simplify our command to:

```
1 - $ ./moqa.py sample.moqa
```

Once the command is executed, the interpreter first builds an abstract syntax tree (AST) from the source code, then it walks over the type-checked AST. The output for program `sample.moqa` has three parts, illustrated in Figure 3.1 and Figure 3.2, and one textual output:

```
[[[3.0], [5.0], [7.0], [6.0]], [2.0], [[1.0], [0.0]]]
```

Two visual representations of the *LPO* are created: one before execute `fun1`, and one after. The function call to `print` generates a textual representation of the final *LPO*.

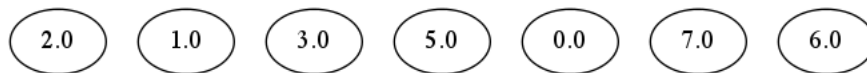


Figure 3.1: `sample.moqa` output 1

3.2.4.2 Analysis Mode

The second mode in the *MOQA* interpreter is called analysis mode. This is where the *MOQA* language differs most from other languages. Based on built-in *MOQA* random bag preservation operations, the interpreter keeps track of

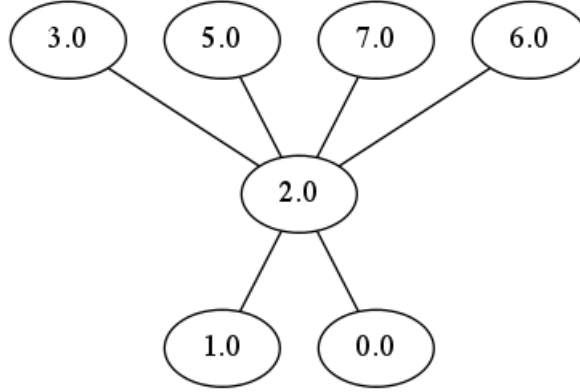


Figure 3.2: sample.moqa output 2

random bags for each variable. Because of the linear compositionality of *MOQA* programs, the ACET for a program is simply the sum of the ACET of individual parts.

To calculate the ACET for a particular algorithm, the interpreter does need to walk over the abstract syntax tree (AST). But the semantics for each operation is different compared with the execution mode. Instead of a single *LPO* as the input for each operation, in analysis mode, the input for each operation is a random bag, and the output is also a random bag. And, *MOQA* timing functions for each operation are used to calculate the time cost for the transformation. In contrast, in execution mode, the output is a single *LPO*, and there is no timing information. The details of both semantics are presented in Section 3.3.

Consider the following command:

```
1 $ python moqa.py -T20 sample.moqa
```

It invokes the *MOQA* interpreter analysis mode on `sample.moqa`. Since we generally assume that any *MOQA* program starts from a discrete partial order, `-T20` tells the interpreter to use analysis mode (`-T`) with initial partial order size 20. The analyzer then analyses the code in `sample.moqa`, one function is found and named `fun1`. The analyzer uses the provided initial partial order size (20) and derives the number of comparisons for this function, which for this case is 19. This result can be verified according to Theorem 2.5, where $\bar{T}_{split}(R(\Delta_n)) = n - 1$, and where in this example $n = 20$. Figure 3.3 is a screen-shot from the *MOQA*

interpreter output.

```
% python moqa.py -T20 sample.moqa
MOQA Code Analyzer found 1 function defined in sample.moqa
=====
| function name | discrete partial order size | number of comparisons
| fun1         | 20                          | 19
```

Figure 3.3: sample.moqa ACET output

Besides `-T`, there are two extra parameters that the user can provide to the *MOQA* interpreter.

- `-S` is used to switch interpreter analysis mode from automated average-case analysis to smoothed analysis. We will cover how to integrate smoothed analysis in Chapter 7.
- `-D` is a helper feature, it produces a visual representation for an abstract syntax tree. It can help debug the *MOQA* program and is especially useful for future interpreter extensions. We will discuss *MOQA* code abstract syntax tree in Section 4.4.

Remark 3.1. The analyzer analyses code at the level of function definitions. It is a general practice in *MOQA* programming to define an algorithm inside a function definition. With this method, syntactically, we force the algorithm to take a discrete partial order as its input, since the form of a function definition in *MOQA* programming is restricted. We will return to this restriction in later sections. The statements written outside a function definition are not evaluated or analysed by the analyzer.

3.3 *MOQA* Language Syntax

In this section we give the specification for the *MOQA* language. This section mainly focuses on defining the language formally. In Chapter 4 we present a practical implementation for these formal concepts.

At the current stage, *MOQA* is a static type-checked, imperative programming language. A program is static type-checked before being interpreted. The

designed type system can prevent simple type mismatches, such as to invoke the *MOQA Split* operation on a series data structure. We provide a specification of the *MOQA* language syntax in Section 3.3.3. Before we dive into the details, we start with lexical conventions first.

3.3.1 Lexical Conventions

A *MOQA* program consists of a single file. At global scope (outside any function definition, i.e. at the “top-level” of your program), a programmer could create an initial *LPO*, define functions or invoke function calls. Execution begins with the statements at global scope, line by line.

3.3.1.1 Comments

Block comments are introduced with `/*` and terminated with `*/`. Like the C family of languages, nesting comments is not permitted. Comments are in general ignored by the interpreter.

3.3.1.2 Identifiers

An identifier is a string that starts with an alphabetic letter followed by a sequence of letters, digits and underscore. Identifiers are case-sensitive in the *MOQA* language. The regular expression for an identifier is: `[A-Za-z][0-9A-Za-z_]*`

3.3.1.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<code>def</code>	<code>if</code>	<code>let</code>	<code>return</code>	<code>for</code>
<code>else</code>	<code>true</code>	<code>false</code>	<code>print</code>	<code>show</code>
<code>end</code>	<code>do</code>	<code>and</code>	<code>or</code>	<code>to</code>
<code>xor</code>	<code>Merge</code>	<code>downto</code>	<code>PercM</code>	

3.3.1.4 Primitive Types

There are three primitive types in *MOQA*. These are generally used as label values in a *LPO* node.

- *Numeric*: A numeric type must be in the form of an optional minus sign followed by either an integer, i.e. one or more digits, or a floating-point number, i.e. one or more digits followed by a period followed by zero or more digits. For example:

123 162.6 -6.7 -81 57.

A regular expression for numeric type is: `-?[0-9]+(\.[0-9]*)?`

- *String*: A string is a sequence of double quotes characters, that also allows escaped sequences. For example:

"cork" "a\"bc" A regular expression for string type is: `"([\\"\\]|(\\.))*"`

- *Boolean*: Boolean type can take either `true` or `false`.

3.3.2 Labelled Partial Order

Besides primitive data types, the second category of types in the *MOQA* language is the labelled partial order type. Based on the structure of the underlying partial order, it can be divided into three types:

- parallel labelled partial order (*PLPO*): a data structure built with parallel composition.
- series labelled partial order (*SLPO*): a data structure built with series composition.
- discrete parallel labelled partial order (*DLPO*): a sub-type of parallel labelled partial order, where each parallel component is a single node.

From a programmer's point of view these data types are invisible. The programmer only deals with *LPO* objects, where these types can be thought of as states or structures of a *LPO*. It is the programmer's responsibility to know how the shape of a data structure is evolved. The interpreter also tracks data types in order to support type checking and timing calculation.

All \mathcal{MOQA} programs take discrete labelled partial orders as inputs. The programmer does not need to declare a data type in the program. The job is to use \mathcal{MOQA} built-in operations plus control-flow constructs to guide the transformation of this initial LPO . The transformations are defined by \mathcal{MOQA} operations and the details are presented in Section 2.3.3.

Here we present a list of operations applied to the LPO type in Table 3.1. Notice that some operations are restricted to a particular LPO subtype (structure shape).

\mathcal{MOQA} language operator symbol	Operation name	Restriction
$><$	<i>Split</i> operation	$DLPO$
$<>$	<i>Product</i> operation	LPO
\wedge	<i>Top</i> operation	$DLPO$
\sim	<i>Bot</i> operation	$DLPO$
PercM	Percolation operation (Perc^M)	$SLPO$

Table 3.1: \mathcal{MOQA} language LPO operations

Similar to commonly used programming languages, such as Java, Python, all the types in our language inherit from a basic type object. The type system in \mathcal{MOQA} language can be seen in Figure 3.4. Notice that the type *Object* is not directly used by a programmer. It is only used as return type of control-flow constructs to help type checking. The formal definitions for all \mathcal{MOQA} language types and typing rules are presented in Section 3.4.

3.3.3 \mathcal{MOQA} Language Grammar

In this section, we do not use pure Backus-Naur Form (BNF) [13, 15] to specify the \mathcal{MOQA} language syntax. For convenience, some regular expression notations are also used.

Specifically, $A?$ means A is optional; A^* means zero or more A s in succession; A^+ means one or more A s. Double brackets $[[\]]$ are used to show association of grammar symbols and they are not part of \mathcal{MOQA} ; they are used in the grammar as a meta-symbol (e.g. $a[[bcd]]^+$ means a followed by one or more bcd triples).

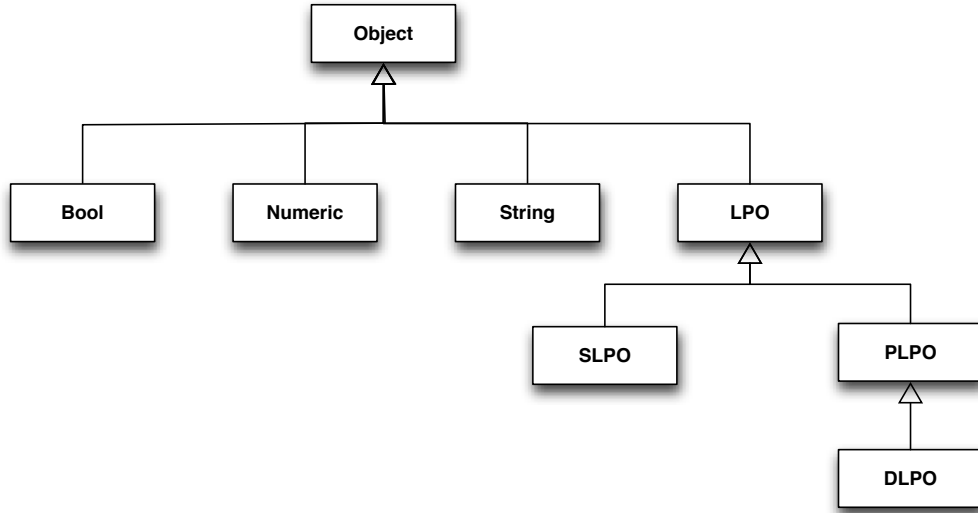


Figure 3.4: *MOQA* Language Type System

In the following section, for each BNF rule, non-terminals are in italics, while terminals use all-capitals. Commonly used symbols are: (, - [etc.

3.3.3.1 Structure of a *MOQA* Program

Every *MOQA* program is a sequence of *elements*. An *element* can be a function definition or an expression or a *LPO* builder. By placing *LPO* creation at the same level as function definition, syntactically, we can prevent the user from defining a new *LPO* inside a function definition. The job for a user defined function in *MOQA* programming, is to only manipulate an input *LPO* and to return the processed *LPO* or a component of it.

Grammar in Backus-Naur form

program \rightarrow *element* +

element \rightarrow *expr* | *defFun* | *lpoBuilder*

3.3.3.2 Variable Declarations

Variables are introduced by the `let` expression. Generally, in *MOQA* programming, this expression is used to bind a variable to an initially discrete labelled

partial order (*DLPO*), or to bind the result of an expression to a named variable. *lpoBuilder* can be viewed as a fixed form `let` expression. It encloses all node labels in curly braces.

Listing 3.3: Simple *MOQA* Code: variable declarations

```
1 let var1 = {2,1,3,5,0,7,6} /* LPO variable var1 */
2 let var2 = 10 /* Numeric variable var2, value 10 */
```

The first line builds a 7 nodes discrete labelled partial order (*DLPO*) called `var1`. The second line declares a numeric variable `var2` with value 10, but in current *MOQA* programming we seldom declare numeric or string type variables directly. This type of data is commonly used as a *LPO* label.

Grammar in Backus-Naur form

lpoBuilder \rightarrow `let` IDENTIFIER = { *nodelist* }

nodelist \rightarrow *node* [[, *node*]] *

node \rightarrow NUM | STRING

expr \rightarrow `let` IDENTIFIER = *expr*

3.3.3.3 Function Definition

Function definitions in *MOQA* are similar to those found in other programming languages, such as Java or C/C++,

Function definitions begin with the keyword `def` followed by a valid identifier (see Section 3.3.1.2), a possible empty list of formal parameters.

Currently, any number of formal parameters are syntactically correct, while, because of limitations on *MOQA*'s theory, the analyzer only processes functions with inputs of one *LPO* and one optional integer ¹.

The return type of a *MOQA* function is also fixed to be a *LPO* type. This decision makes type checking and code analysis easier. Recall that *MOQA*

¹As a theory simplification, *MOQA* deals only with single *LPO* input, with a necessary extension, the theory should also be applicable to multiple input *LPOs*.

operations are all SP-order preserving (see Section 2.3.2.1), thus the user defined functions in *MOQA* programming are always a mapping from a discrete partial order, with an optional integer, to a SP-order.

Grammar in Backus-Naur form

$defFun \rightarrow \text{def IDENTIFIER } (\text{optparams}) \text{ expr+ end}$

$optparams \rightarrow \text{IDENTIFIER } [[, \text{IDENTIFIER}]] ^*$

3.3.3.4 Expression

Expressions are the largest syntactic category in *MOQA* language.

Constants

The simplest expressions are constants. There are three types of constants. These were first introduced in Section 3.3.1.4.

- The boolean constants are true and false.
- Numeric constants are integer or real numbers.
- String constants are sequences of characters enclosed in double quotes.

Grammar in Backus-Naur form

$expr \rightarrow \text{NUM} \mid \text{BOOL} \mid \text{STRING}$

Built-in Functions

We provide two helper built-in functions: `print` and `show`. They take the following forms:

Grammar in Backus-Naur form

$expr \rightarrow \text{print } (\text{expr})$
 $\mid \text{show } (\text{expr})$

Both functions use side-effects to print or visualize data structures or values. The return type of built-in functions is *Object*.

`print` function provides printing to the standard output. It can be applied to primitive types or *LPO* types. The `show` function visualises *LPOs* using Graphviz's Dot language [9]. It is restricted to *LPO* types only. An example is shown in Section 3.2.4.1 (page 51).

Variable Assignment

In *MOQA* programming, there are two types of variable assignments, used to update the value associated with a variable. They take the following forms:

Grammar in Backus-Naur form

$$expr \rightarrow \text{IDENTIFIER } indexes? = expr$$
$$indexes \rightarrow [expr] \mid [expr? : expr?]$$

The *indexes* are optional. When there are no *indexes*, this expression updates the value associated to an identifier. And the identifier must be declared before hand by the `let` expression. The identifier also must have type *LPO*.

If the *indexes* exist, syntactically, both forms of indexes are valid, but in the typing rule we enforce only the first form. That is, variable assignment only works with a single numeric index. Also, the target identifier must be a variable with type *PLPO* or its subtype *DLPO*. This form is used to update a component of a indexable *LPO*. We discuss the types of *LPO* objects that are indexable in the next paragraph.

LPO Index and Slice

Slice and *index* application take the following forms:

Grammar in Backus-Naur form

$$expr \rightarrow \text{IDENTIFIER } indexes?$$
$$indexes \rightarrow [expr] \mid [expr? : expr?]$$

These two operations have similar semantics to operations in Python. Indices are 0 indexed and *slice* operations takes two integers separated by a colon. If the first integer is not specified, the system use 0 as default. If the second integer is missing, the number of components is used.

We extend these operations to *LPO* type and include a new restriction, used to extract components of a *LPO* type variable. The indices must have type numeric.

Because in our language we do not separate integer and floating numbers, the system always casts numeric type to integer by default.

Index operations are applied to *DLPO* variables or to the result of the `split`, `top`, `bot` or `PercM` operation, while *slice* operation only works on *DLPO* type variables.

Example 3.1. If a variable `X` binds to a *DLPO* shown in Figure 3.1 (page 52):

- `X[5]` means a *DLPO* with only one node labelled 7.0.
- `X[2:5]` means a *DLPO* with sliced labels {3.0, 5.0, 0.0}.
- `X[3:]` means a *DLPO* with sliced labels {5.0, 0.0, 7.0, 6.0}.

Index operations not only work on *DLPO* type variables, they are also applied to the result of `split`, `top`, `bot` or `PercM` operations. For these cases, index values are restricted to {0, 1, 2}. In particular, the result of `PercM` operation is restricted to {0, 1}.

Example 3.2. If a variable `X` binds to a *SLPO*, shown in Figure 3.2 (page 53), this *SLPO* is the result of applying the `split` operation.

- `X[0]` means a *DLPO* with labels greater than the pivot, that is {3.0, 5.0, 7.0, 6.0}.
- `X[1]` means a *DLPO* with only one node labelled with pivot 2.0.
- `X[2]` means a *DLPO* with labels less than pivot, that is {1.0, 0.0}.

If a variable `X` binds to a *SLPO* shown in Figure 2.14 (page 37), this *SLPO* is the result of applying the `top` operation.

-
- $X[0]$ means a *DLPO* with labels occurs before maximum label, that is $\{1\}$.
 - $X[1]$ means a *DLPO* with only one node labelled with maximum label $\{4\}$.
 - $X[2]$ means a *DLPO* with labels occurs after maximum label, that is $\{3, 2\}$.

If a variable X binds to a *SLPO* shown in Figure 2.15 (page 38), this *SLPO* is the result of applying the `PercM` operation.

- $X[0]$ means a *DLPO* with only one node labelled with maximum label 5.
- $X[1]$ means a *SLPO* with four elements $\{4, 3, 2, 1\}$. It is also marked as the result of applying a `PercM` operation.

Return Expression and Function Call

Like other programming languages, the *MOQA* language has return expressions and function calls, as follows:

Grammar in Backus-Naur form

$expr \rightarrow \mathbf{return} \ expr \mid \mathit{functionCall}$

$\mathit{functionCall} \rightarrow \text{IDENTIFIER} \ (\ \mathit{optargs}? \)$

$\mathit{optargs} \rightarrow \mathit{expr} \ [[, \ \mathit{expr} \] \]^*$

In order to support automated average-case analysis, the *MOQA* language differs from other languages in the following ways:

- return expressions only return *LPO* type values.
- User defined functions are restricted (see Section 3.3.3.3), thus a function call either inputs a *DLPO* or a *DLPO* plus an integer.

Arithmetical Expressions

Like other languages, *MOQA* provides commonly used arithmetic operations, it has the following forms:

Grammar in Backus-Naur form

$expr \rightarrow arithExpr$

$arithExpr \rightarrow expr + expr$

| $expr - expr$

| $expr * expr$

| $expr / expr$

| $expr \% expr$

To evaluate an arithmetical expression, the first *expr* is evaluated, then the second *expr*. The result of the expression is to apply the arithmetic operator to both sub-expressions.

Notice that sub-expressions in arithmetical expressions must have numeric type. While syntactically it doesn't matter, the typing rule we present later will enforce this. The return type for an arithmetic expression is numeric type.

Logic Expressions

Due to branching and loops, static time analysis is complicated. Both types of expressions are complicated to analyse because of their dependence on boolean expressions. Even worse, while-loops are not analysable in full generality due to non-decidability of termination, i.e. the halting problem [98]. To derive the average-case cost of an **if** expression, it is necessary to determine the probability of executing the **then** branch and the probability of executing the **else** branch.

Grammar in Backus-Naur form

$expr \rightarrow logicExpr$

$logicExpr \rightarrow moqaCond \text{ and } moqaCond$

| $moqaCond \text{ or } moqaCond$

| $moqaCond \text{ xor } moqaCond$

| $\text{not } moqaCond$

| $moqaCond$

$lpoSize \rightarrow | \text{IDENTIFIER} |$

$$\begin{aligned} moqaCond &\rightarrow lpoSize > expr \\ &| lpoSize \geq expr \\ &| lpoSize < expr \\ &| lpoSize \leq expr \end{aligned}$$

In *MOQA* programming we are interested in the determination of specific classes of boolean expressions for which it can be guaranteed that the probability can be statically derived. In BNF, we call this kind of boolean expression *moqaCond*, which only allows comparing the size of a *LPO* to a numeric value.

Like other programming languages, logic expressions are composed by *moqaCond* using logic operators, such as **and**, **or** etc. The type of a logic expression is boolean.

If Statement

An **if** statement in the *MOQA* language has the following form; where the predicate is a *MOQA* logic expression to support automated average-case analysis:

Grammar in Backus-Naur form

$$\begin{aligned} expr &\rightarrow ifStatement \\ ifStatement &\rightarrow ifStat elseStat? \mathbf{end} \\ ifStat &\rightarrow \mathbf{if} logicExpr \mathbf{do} expr+ \\ elseStat &\rightarrow \mathbf{else} \mathbf{do} expr+ \end{aligned}$$

The semantics of conditionals in the *MOQA* language are standard. The predicate (*logicExpr*) is evaluated first. If the predicate is true, the **then** branch is evaluated. If the predicate is false, then the **else** branch is evaluated. The **else** branch is optional.

The beginning and ending of conditionals are marked by keywords **do**, **end**, where Java uses curly braces, Python uses indentation. The value of the conditional is the value of the evaluated branch. Also the predicate must have type **bool**.

For Statement

In *MOQA* programming, a `for` statement basically has two flavours: loops from a numeric type variable from a smaller value to a larger value, or loops from a larger value down to a smaller value. These take the following form:

Grammar in Backus-Naur form

$expr \rightarrow forStatement$

$forStatement \rightarrow \text{for IDENTIFIER} = expr \text{ [[to | downto]] } expr \text{ do } expr + \text{end}$

In both cases, the loops guarantee to terminate and make deriving the average-case cost easier. When the loops starts, it introduces a new binding. It binds the variable name that the user specifies with the initial value. In each iteration, this variable either increases or decreases by one depending on the loop's type. Notice that to coordinate 0 indexing in *LPO* type, we follow the convention of other programming languages. The loop is stopped one value before the termination value. e.g. `for i = 1 to 6 do print(i) end` will only print values from 1 to 5.

MOQA Expressions

The last expression type is called *MOQA* expression, the most distinct feature of the *MOQA* language. It is not available in other programming languages and has the following forms:

Grammar in Backus-Naur form

$expr \rightarrow moqaExpr$

$moqaExpr \rightarrow expr \langle \rangle expr$

| $expr \rangle \langle expr$

| $\wedge expr$

| $\sim expr$

| Merge ($expr$, $expr$)

| PercM ($expr$)

They include all the basic *MOQA* operations provided in the current language. These operations were introduced in Section 2.3.3. For convenience, in our language we use symbols to present some basic *MOQA* operations, such as $\langle \rangle$, meaning the *Product* operation in *MOQA*. The details for these operations are discussed in Section 3.3.2, and we also discuss type restriction on each operation in Table 3.1.

3.3.4 Scopes

The topic of scope is important in any programming language. It defines how to match identifiers declarations with their usage. A variable x might have multiple definitions in the program. We need to know which definition we are talking about in any place of the program. The scope of an identifier defines the portion of a program in which the identifier is accessible.

There are generally two types of scoping rules, one named static scope, the other dynamic scope [13, 15]. Few languages are dynamically scoped such as the old version of Lisp [41], or SNOBOL [44], while most languages are statically scoped. Like C/C++, Java, Python etc, *MOQA* is statically scoped, that is, scope depends on program text not run-time behaviour. Generally speaking, most identifiers follow the most-closely nested rule. That is a variable binds to the definition most closely enclosing it. This should be familiar to most C/C++, Java or Python programmers.

3.3.5 *MOQA* Language Syntax Specification

In this section we present the full *MOQA* language syntax specification. More readily understood syntax diagrams are presented in Appendix A1.

$program \rightarrow element +$

$element \rightarrow expr \mid defFun \mid lpoBuilder$

$lpoBuilder \rightarrow let IDENTIFIER = \{ nodelist \}$

$nodelist \rightarrow node \llbracket [, node \rrbracket \ast$

```

node → NUM | STRING

expr → let IDENTIFIER = expr
      | IDENTIFIER indexes? = expr
      | print ( expr ) | show ( expr ) | return expr | ( expr )
      | IDENTIFIER indexes?
      | functionCall | ifStatement | forStatement
      | arithExpr | logicExpr | moqaExpr | lpoSize
      | NUM | BOOL | STRING

lpoSize → | IDENTIFIER |

indexes → [ expr ] | [ expr? : expr? ]

moqaExpr → expr <> expr | expr >< expr
          | ^ expr | ~ expr | Merge ( expr , expr ) | PercM ( expr )

arithExpr → expr [[+ | - | * | / | %]] expr

logicExpr → moqaCond [[and| or | xor]] moqaCond | not moqaCond | moqaCond

moqaCond → lpoSize [[> | >= | < | <=]] expr

functionCall → IDENTIFIER ( optargs? )

optargs → expr [[, expr ]] *

ifStatement → ifStat elseStat? end

ifStat → if logicExpr do expr+

elseStat → else do expr+

forStatement → for IDENTIFIER = expr [[to | downto]] expr do expr+ end

defFun → def IDENTIFIER ( optparams ) expr+ end

optparams → IDENTIFIER [[, IDENTIFIER ]] *

IDENTIFIER ::= [A-Za-z][0-9A-Za-z_]*
NUM ::= -?[0-9]+(\.[0-9]*)?
BOOL ::= true | false
STRING ::= "([^\\"|(\\"\.))*"

```

3.4 *MOQA* Language Type System

Currently the type system in *MOQA* programming is restricted and simple, but sufficient to support all the operations defined in *MOQA* theory. As shown earlier in Figure 3.4, all the types in *MOQA* programming are inherited from the type *Object*. Because the language does not support user defined types at the moment, the typing rule we illustrate here is used to prevent common type mismatch, such as to apply the *Product* operation between a *LPO* object and an integer.

In *MOQA* programming, the programmer does not need to declare the type of a variable. The type system will infer the type. There are two categories of types in the *MOQA* language: the primitive type and the *LPO* types.

For the primitive type, the inference rules are easy, based on the literal value we can infer the variable type directly, eg. variable assigned to number 5 has type *Numeric*. In terms of *LPO* types, because the input and output types for each *MOQA* operation are well defined, we use typing rules to track the type for each variable. Furthermore, restrictions on function definitions in *MOQA* programming (see Section 3.3.3.3), make our type inference even simpler, since we can assume the functions must have one of the following signatures:

- $DLPO \mapsto LPO$
- $DLPO \times Num \mapsto LPO$

Thus in our language, we do not apply the Hindley-Milner type inference algorithm [13]. Instead we employ typing rules to directly infer types.

3.4.1 Type Environments

The typing rules define the type of every *MOQA* expression in a given context. The context is the type environment, which assigns types to the free identifiers appearing in an expression. A variable is free in an expression if it is not defined within the expression. To type check an expression, we need extra type information on identifiers that is not defined within the current expression. This information is provided by the type environment.

During type checking, we apply recursive descent to an abstract syntax tree (AST). The type environment is passed down the AST from the root towards the leaves. In order to compute the type of an expression, say e , we first compute the types for e 's sub-expressions, then, based on these types, compute the type of e .

Before we dive into the *MOQA* language type checking rules, we introduce the relevant notations.

The type environment Γ is a function from identifiers to types. It has the following features:

- $\Gamma(x) = T$ means variable identifier x has type T in type environment Γ .
- $\Gamma(f) = T1 \mapsto T2$ means function identifier f maps type $T1$ to type $T2$ in type environment Γ .
- Every expression e is type checked in a type environment. Because some expressions might introduce a new identifier, their sub-expressions need to be type checked in a modified environment, e.g. `for`, `def` expression.
 - $\Gamma(T/y)$: A modified environment Γ , and identifier y has type T .
 - $\Gamma(T/y)(y) = T$
 - $\Gamma(T/y)(x) = \Gamma(x)$ if $x \neq y$.

As introduced earlier (see Figure 3.4, page 58), there is a type hierarchy in *MOQA* types, similar to other programming languages. In a function call or expression, if a value of type T is expected, then any value of subtype U may be used instead. In other words, if type U inherits from type T , either directly or indirectly, whenever a value of T is needed, we could substitute it with a value of type U .

In programming languages this is called *U conforms to T*, and represented by $U \leq T$ (U is a subtype of T). We give the formal definition below.

Definition 3.1. Assume three types A, B, C .

- $A \leq A$ for all types A
- if B inherits from A , then $B \leq A$.

-
- if $C \leq B$ and $B \leq A$, then $C \leq A$.

It can be seen in our type hierarchy in Figure 3.4 (page 58), for all types X we have $X \leq Object$.

To statically type check all *MOQA* expressions, we need to introduce one more operator: \sqcup . This operator applies to two types, say X and Y . It obtains the least common ancestor for X and Y in the type hierarchy.

Example 3.3. For example, the *MOQA* type hierarchy in Figure 3.4 (page 58)

- $DLPO \sqcup SLPO = LPO$
- $DLPO \sqcup PLPO = PLPO$
- $SLPO \sqcup Numeric = Object$

This operator is introduced to deal with the return type of conditional expressions. For example, let T and F be the types of the branches for a conditional expression, then the type for this conditional expression is $T \sqcup F$.

3.4.2 Type Checking Rules

In this section we present type checking rules for the *MOQA* programming language. A type checking rule has the general form:

$$\frac{\Gamma \vdash h_1 \cdots \Gamma \vdash h_n}{\Gamma \vdash e:T}$$

The statements above the horizontal bar represents hypotheses, the statement below the bar is the conclusion. If the hypotheses are satisfied, then the conclusion is true. This rule can be read as follows: in the type environment Γ , given hypotheses $h_1 \cdots h_n$ are all true, then it is provable that the expression e has type T .

Sometimes, an expression may not only evaluate to a type but also have a side-effect on the type environment, such as to introduce a new identifier-type binding. We write such a rule as: $\Gamma \vdash e:T, \text{set } \Gamma(id) = T$. It means that expression e has type T , but it also modifies the type environment by introducing identifier id with type T . The `let` expression is one such example.

The typing rules for constants are the easiest.

$$\begin{array}{c}
\frac{}{\Gamma \vdash \mathbf{true} : Bool} \quad [\text{True}] \\
\frac{}{\Gamma \vdash \mathbf{false} : Bool} \quad [\text{False}] \\
\frac{\mathbf{n} \text{ is a number literal}}{\Gamma \vdash \mathbf{n} : Numeric} \quad [\text{Numeric}] \\
\frac{\mathbf{s} \text{ is a string literal}}{\Gamma \vdash \mathbf{s} : String} \quad [\text{String}]
\end{array}$$

The rule for identifiers simply returns the type associated with the identifier in type environment.

$$\frac{\begin{array}{c} id \text{ is identifier literal} \\ \Gamma(id) = T \end{array}}{\Gamma \vdash id : T} \quad [\text{Var}]$$

From the syntax specification (see Section 3.3.5) we can see that *MOQA* programs can be thought of as a list of expressions, where the type for this sequencing is the type of the last expression.

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : T1 \\ \Gamma \vdash e_2 : T2 \\ \vdots \\ \Gamma \vdash e_n : TN \end{array}}{\Gamma \vdash e_1 e_2 \dots e_n : TN} \quad [\text{Sequence}]$$

The *LpoBuilder* rule introduces a *DLPO* type variable. Syntax rules guarantee that *nodelist* sub-expression e has a proper format, that is, all elements are a constant label type value, either *Numeric* or *String*. Thus we skip type checking for it, and just modify the type environment to make sure id has type *DLPO*. In the execution runtime, once the interpreter matches this node, it creates the *DLPO* object to hold all labels contained in *nodelist* sub-expression e .

$$\frac{}{\Gamma \vdash \mathbf{let } id = \{ e \} : Object, \mathbf{set } \Gamma(id) = DLPO} \quad [\text{LpoBuilder}]$$

Let expression binds a variable to a type checked expression e with type T .

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{let } id = e : \text{Object}, \text{set } \Gamma(id) = T} \quad [\text{Let}]$$

MOQA expressions are the most interesting part of the typing rules. Different operations have different restrictions, and the types that are applicable are different. Some operations might also change the type of the original variable because of *MOQA* data structure transformations.

For the *Product* operation, the two operands $e1$ and $e2$ must have type *LPO* or one of its subtypes.¹

$$\frac{\Gamma \vdash e1 : T1 \quad \Gamma \vdash e2 : T2 \quad T1 \leq LPO \quad T2 \leq LPO}{\Gamma \vdash e1 \langle \rangle e2 : SLPO} \quad [\text{Product}]$$

For the *Split* operation, the typing rule is a bit complicated, because we want to tag the identifier as the result of split operation. This will help future type checking of the *index* or *slice* operation.

Operand $e1$ must be an identifier and have type *DLPO*, $e2$ must be a single node *DLPO*. During runtime, the system will also check to make sure $e2$ is an element of $e1$.

If all hypotheses are satisfied, we update the type environment to set $e1$ with type *SLPO* and tag $e1$ as the result of a split operation by setting $\Gamma(_e1) = \>\langle$. Because $_e1$ (append ‘ $_$ ’ before $e1$) is not a valid identifier, it is safe to do so.

$$\frac{\begin{array}{l} e1 \text{ is identifier literal} \\ \Gamma(e1) = DLPO \\ \hline \Gamma \vdash e1 : DLPO \end{array} \quad \Gamma \vdash e2 : DLPO \quad \Gamma \vdash |e2| = 1}{\Gamma \vdash e1 \>\langle e2 : SLPO, \text{set } \Gamma(e1) = SLPO, \text{set } \Gamma(_e1) = \>\langle} \quad [\text{Split}]$$

Similar to the *Split* operation, *Top* and *Bot* operations only apply to variables with *DLPO* type, and we need to tag these variables to support type checking on index operations.

¹The two operands for the *Product* operation also need to be two disjoint random structures or two isolated subsets of the same random structure. In our language, only one *LPO* object is manipulated and because of the restriction on accessing structure components (through the *Index* or *Slice* operation), the components obtained are guaranteed to be isolated.

$$\frac{\frac{\frac{id \text{ is identifier literal}}{\Gamma(id) = DLPO}}{\Gamma \vdash id: DLPO}}{\Gamma \vdash \hat{\sim} id: SLPO, \text{set } \Gamma(id) = SLPO, \text{set } \Gamma(_id) = \hat{\sim}} \quad [\text{Top}]$$

$$\frac{\frac{\frac{id \text{ is identifier literal}}{\Gamma(id) = DLPO}}{\Gamma \vdash id: DLPO}}{\Gamma \vdash \sim id: SLPO, \text{set } \Gamma(id) = SLPO, \text{set } \Gamma(_id) = \sim} \quad [\text{Bot}]$$

The **PercM** operation only applies to *SLPO* type variables. This operation has no side-effect to change the original variable type, but to support the *index* operation, we tag the target variable as well.

$$\frac{\frac{\frac{id \text{ is identifier literal}}{\Gamma(id) = SLPO}}{\Gamma \vdash id: SLPO}}{\Gamma \vdash \text{PercM}(id):SLPO, \text{set } \Gamma(_id) = \text{PercM}} \quad [\text{PercM}]$$

To help implement merge sort (see Section 5.4), the **Merge** operation is introduced. It is not a standard *MOQA* operation but the average-case cost of this operation is known [73]. The two operands for the **Merge** operation must both have type *SLPO* and must be in sorted order. The sorted order is not checkable statically, thus we need to employ runtime checking for this operation as well.

$$\frac{\frac{\frac{e1 \text{ is identifier literal}}{\Gamma(e1) = SLPO}}{\Gamma \vdash e1: SLPO} \quad \frac{\frac{e2 \text{ is identifier literal}}{\Gamma(e2) = SLPO}}{\Gamma \vdash e2: SLPO}}{\Gamma \vdash \text{Merge}(e1, e2) : SLPO} \quad [\text{Merge}]$$

Index operations only apply to values of type *PLPO*, *DLPO* or the special *SLPO* type, i.e. the result of a *Split*, *Top/Bot* or **PercM** operation. The type of expression *e* must have type *Numeric* and will be cast to an integer.

$$\frac{\left(\frac{T \in \{PLPO, DLPO\}}{\Gamma \vdash \Gamma(id) : T} \vee \frac{\Gamma \vdash e : \text{Numeric}}{\Gamma \vdash \Gamma(_id) : \oplus \quad \oplus \in \{\langle \rangle, \rangle \langle, \wedge, \sim, \text{PercM}\}} \right)}{\Gamma \vdash id[e] : DLPO} \quad [\text{Index}]$$

The *Slice* operation is used to obtain part of elements from a *DLPO* type variable. Both indices $e1$ and $e2$ are optional and have type *Numeric*. During runtime the indices will be cast to integers. Example 3.1 (page 62) illustrates this operation.

$$\frac{\frac{id \text{ is identifier literal}}{\Gamma(id) = DLPO} \quad \frac{e1 \text{ exists}}{\Gamma \vdash e1 : \text{Numeric}} \quad \frac{e2 \text{ exists}}{\Gamma \vdash e2 : \text{Numeric}}}{\Gamma \vdash id[e1 : e2] : DLPO} \quad [\text{Slice}]$$

The assignment operation is only applicable to *LPO* type variables, and is used to update the associated value to an identifier. The type of the identifier is updated to the type derived for the right hand side expression.

$$\frac{\Gamma \vdash id : T1 \quad \Gamma \vdash e : T2 \quad T1 \leq LPO \quad T2 \leq LPO}{\Gamma \vdash id = e : \text{Object}, \text{set} \Gamma(id) = T2} \quad [\text{Assign}]$$

The indexed assignment is a special assignment operation, currently only designed for the Heapify algorithm to build a binary heap out of a *DLPO* type variable. The usage details of this operation is discussed in Section 5.7. The type of id is restricted to *DLPO* and *PLPO*. This operation will transform a *DLPO* type variable to a *PLPO* one. Also, if the resulting *PLPO* variable size is 1, the type of variable will change to *SLPO*.

$$\frac{\frac{T \in \{DLPO, PLPO\}}{\Gamma \vdash \Gamma(id) : T} \quad \Gamma \vdash e1 : \text{Numeric} \quad \Gamma \vdash e2 : SLPO}{\Gamma \vdash id[e1] = e2 : \text{Object}, \text{set} \Gamma(id) = SLPO \text{ if } |id| = 1 \text{ else } \Gamma(id) = PLPO} \quad [\text{IndexedAssign}]$$

The **for** expression iterates an integer variable from value $e1$ to $e2$ with step

size 1 or iterates reversely in a `downto-for` loop with step size -1 . The type of the entire `for` loop is always *Object*, the type checker recursively type checks loop body *es* in new type environment $\Gamma[\text{Numeric}/id]$.

$$\frac{\Gamma \vdash e1 : \text{Numeric} \quad \Gamma \vdash e2 : \text{Numeric} \quad \Gamma[\text{Numeric}/id] \vdash es : T \quad T \leq \text{Object}}{\Gamma \vdash \text{for } id = e1 \text{ [[to|downto]] } e2 \text{ do } es \text{ end} : \text{Object}} \quad [\text{For}]$$

MOQA condition compares the size of a *LPO* type variable with numeric type expression *e*.

$$\frac{\oplus \in \{>, >=, <, \Leftarrow\} \quad \Gamma \vdash \Gamma(id) : T \quad T \leq \text{LPO} \quad \Gamma \vdash e : \text{Numeric}}{\Gamma \vdash |id| \oplus e : \text{Bool}} \quad [\text{MoqaCond}]$$

The type checking rules for arithmetic and logic expressions are standard, similar to other programming languages.

$$\frac{\Gamma \vdash e1 : \text{Numeric} \quad \Gamma \vdash e2 : \text{Numeric} \quad \oplus \in \{+, -, *, /, \%\}}{\Gamma \vdash e1 \oplus e2 : \text{Numeric}} \quad [\text{ArithExpr}]$$

$$\frac{\Gamma \vdash e1 : \text{Bool} \quad \Gamma \vdash e2 : \text{Bool} \quad \oplus \in \{\text{and}, \text{or}, \text{xor}\}}{\Gamma \vdash e1 \oplus e2 : \text{Bool}} \quad [\text{LogicExpr}]$$

$$\frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{not } e : \text{Bool}} \quad [\text{Not}]$$

The `if` expression rules are straightforward. It depends on whether the expression has an `else` branch or not, so we present it by two rules. One for `if-then`, the other for `if-then-else`. The predicate must have type *Bool*. The return type uses operators we introduced earlier (see Section 3.4.1).

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash es : T \quad T \leq \text{Object}}{\Gamma \vdash \text{if } e \text{ do } es \text{ end} : T} \quad [\text{If-then}]$$

$$\frac{\begin{array}{c} \Gamma \vdash e: Bool \\ \Gamma \vdash es1: T1 \\ \Gamma \vdash es2: T2 \\ T1 \leq Object \quad T2 \leq Object \end{array}}{\Gamma \vdash \text{if } e \text{ do } es1 \text{ else do } es2 \text{ end} : T1 \sqcup T2} \quad [\text{If-then-else}]$$

In \mathcal{MOQA} programming, to facilitate automated average-case analysis, currently only two types of function definition are allowed: the single input function or the double inputs function (see Section 3.3.3.3). The first type of function has one $DLPO$ variable as input, the other type of function has two input variables: one $DLPO$ and one Num variable. In both cases, the typing rule checks the body of the function in an environment Γ , where Γ is extended with new bindings of formal parameters to their types. The return type of a user defined function is always a LPO type.

$$\frac{\begin{array}{c} \Gamma[DLPO/p] \vdash es: T \\ T \leq LPO \end{array}}{\Gamma \vdash \text{def id (p) es end} : LPO, \text{ set } \Gamma(id) = DLPO \mapsto LPO} \quad [\text{DefFun}]$$

$$\frac{\begin{array}{c} \Gamma[DLPO/p1, Numeric/p2] \vdash es: T \\ T \leq LPO \end{array}}{\Gamma \vdash \text{def id (p1 , p2) es end} : LPO, \text{ set } \Gamma(id) = (DLPO, Numeric) \mapsto LPO} \quad [\text{DefFun}]$$

The rules for function calls in \mathcal{MOQA} programming is not complicated because of the restriction on function definitions. Once the signature for the target function is retrieved, the number of actual parameters is verified to make sure the function signature has the same number of formal parameters. Then, each actual parameter is type checked, to make sure it conforms to the corresponding formal parameter.

$$\frac{\begin{array}{c} \Gamma \vdash \Gamma(id) = DLPO \mapsto LPO \\ \Gamma \vdash \Gamma(a) = DLPO \end{array}}{\Gamma \vdash id (a) : LPO} \quad [\text{FunctionCall-One}]$$

$$\frac{\begin{array}{l} \Gamma \vdash \Gamma(id) = (DLPO, Numeric) \mapsto LPO \\ \Gamma \vdash \Gamma(a1) = DLPO \\ \Gamma \vdash \Gamma(a2) = Numeric \end{array}}{\Gamma \vdash id (a1 , a2) : LPO} \quad [\text{FunctionCall-Two}]$$

Because **return** expressions are only used inside the definition of a function the returned expression must have type *LPO*.

$$\frac{\Gamma \vdash e : LPO}{\Gamma \vdash \text{return } e : LPO} \quad [\text{Return}]$$

The **print** built-in function is used to print either primitive type variables or *LPO* type variables to standard output, thus it is applicable to all types in *MOQA*. The return type for this expression is always type *Object*. The function uses side-effect to print out values.

$$\frac{\begin{array}{l} \Gamma \vdash e : T \\ T \leq Object \end{array}}{\Gamma \vdash \text{print}(e) : Object} \quad [\text{Print}]$$

Unlike **print** built-in function, the **show** function is only applicable to a *LPO* type variable. It is used to visualize a *LPO* data structure.

$$\frac{\begin{array}{l} \Gamma \vdash e : T \\ T \leq LPO \end{array}}{\Gamma \vdash \text{show}(e) : Object} \quad [\text{Show}]$$

3.5 *MOQA* Language Semantics

Defining a computer language consists of two main parts: syntax and semantics. Syntax describes the actual structure of a program, while semantics describes the meaning of programs. In this section, we define how *MOQA* programs execute by giving a formal semantics for it.

Currently, there are three main approaches for describing formal semantics [113], and some more suitable to various tasks than others.

- Axiomatic semantics: also called Floyd-Hoare Logic, based on formal logic. It studies how logical properties of the program state change as a program is executed. It is generally used to formally prove a property of the state after

execution of a program by assuming another pre-condition. For example, if we know $a > b$ and $c > d$ (pre-condition) then after executing the statement $x = a + b$, we also know that $x > b + d$.

- Denotational semantics: constructs a mapping from programs into equivalent mathematical functions. It is useful for proving properties of programs.
- Operational semantics: it can be essentially thought of as an interpreter written out in mathematical notations, and is by far the most common way to describe programming language semantics. Similar to type derivation it is a rule based method. It describes how a program evaluates via execution rules on an abstract machine. Operational semantics are classified into big-step semantics (natural semantics) and small-step semantics (structural operational semantics). Big-step semantics focus on how the overall result is obtained while small-step semantics concern individual step computation. In this thesis we focus on big-step semantics, because it is closer to our recursive descent interpreter implementation.

Unlike other programming languages, a *MOQA* program has two types of interpretation, an execution semantics and an analysis semantics. These are used in different modes, and they are both based on big-step operational semantics.

3.5.1 *MOQA* Execution Semantics

Execution semantics is used in the interpreter execution mode. It deals with normal *MOQA* program execution, at the level of a single *LPO*.

Evaluation rules in execution semantics also use logic rules of inference, similar to type checking. In type judgement, we give a context to determine the type of an expression. Similarly, in evaluation rules, we give a context to evaluate an expression for its value. Evaluation rules have the following general form:

$$\frac{c_1 \Downarrow v_1 \cdots c_n \Downarrow v_n}{c \Downarrow v}$$

The statements above the horizontal bar present hypotheses, the statement below the bar is the conclusion. If the hypotheses are satisfied, then the conclusion

is true. c, c_1, \dots, c_n are configurations holding program fragments together with an execution environment σ .

Different from typing rules, execution environments map identifiers to their values, e.g. $\sigma = \{x = 1, y = 0\}$, $\sigma(x) = 1$. In particular, the environment also maps a function name identifier to a function definition.

$(e, \sigma) \Downarrow v$ can be read as: in the execution environment σ , expression e evaluates to a value v . Sometimes, expressions do not compute to a value. Instead they change the environment. In this case the form of rule is:

$$(e, \sigma) \Downarrow \sigma'$$

We write $\sigma[val/x]$ for updating environment σ with a new binding from x to val . For example, the *Let* expression binds a variable to an expression e with value v .

$$\frac{(e, \sigma) \Downarrow v}{(\mathbf{let} \ id = e, \sigma) \Downarrow \sigma[val/id]}$$

There are three configuration types for the value returned.

- $(e, \sigma) \Downarrow v$: expression e evaluated to a value v in environment σ .
- $(e, \sigma) \Downarrow \sigma[val/id]$: evaluating expression e in σ causes a side-effect and changes environment to $\sigma[val/id]$.
- $(e, \sigma) \Downarrow (v, \sigma[val/id])$: evaluating expression e in environment σ produces a value v and causes a side-effect and changes environment to $\sigma[val/id]$.

In the following, we give a formal operational semantics for evaluating *MOQA* programs in execution mode. This describes how pieces of a program can be evaluated and it is closely related to real interpreter implementation. Generally, each evaluation rule will be mapped to a corresponding abstract syntax tree node processing method. We refer the reader to Section 4.7 for details of implementation for these rules.

When we introduce a new helper function, we will follow the form: $TYPE_{name}$ where $TYPE$ is the type of an object, and $name$ is the operation name associated to that type, e.g. $LPO_{product}$. In the *MOQA* interpreter implementation, we have

a runtime object library which implements a *LPO* object and all the necessary operations associated to it. We use these helper functions to calculate values for *MOQA* expressions.

Like typing rules, evaluating rules for constants are the easiest.

$$\frac{b \in \{\mathbf{true}, \mathbf{false}\}}{(b, \sigma) \Downarrow b} \quad [\text{Bool}]$$

$$\frac{\mathbf{n} \text{ is a number literal}}{(n, \sigma) \Downarrow n} \quad [\text{Num}]$$

$$\frac{\mathbf{s} \text{ is a string literal}}{(s, \sigma) \Downarrow s} \quad [\text{String}]$$

These rules say that a Boolean true/false, string or an integer evaluate to the expected value in any execution environment σ .

The rule for identifiers simply returns the value associated with the identifier in the execution environment. If *id* is not bound in the environment, then this rule doesn't apply and the interpreter should issue an error.

$$\frac{\begin{array}{l} id \text{ is identifier literal} \\ id \in \sigma \end{array}}{(id, \sigma) \Downarrow \sigma(id)} \quad [\text{Var}]$$

MOQA programs can be thought as a list of expressions (e_1, e_2, \dots, e_n) , where we evaluate expressions in order. We evaluate first expression e_1 with initial environment σ , then e_2 with resulting environment σ_1 , until we reach the last expression e_n . The value returned is the value of the last expression.

$$\frac{\begin{array}{l} (e_1, \sigma) \Downarrow (v_1, \sigma_1) \\ (e_2, \sigma_1) \Downarrow (v_2, \sigma_2) \\ \vdots \\ (e_n, \sigma_{n-1}) \Downarrow (v_n, \sigma_n) \end{array}}{(e_1 e_2 \dots e_n, \sigma) \Downarrow (v_n, \sigma_n)} \quad [\text{Sequence}]$$

The *LpoBuilder* rule builds object of type *DLPO*. Here we introduce a helper function $DLPO_{\text{FromLabels}}$. It takes a list of labels (e_1, e_2, \dots, e_n) and creates a *DLPO* object. In the interpreter implementation this directly maps to a function

call in our runtime library.

$$\frac{(\mathbf{let} \ id = \{ e_1, e_2 \dots e_n \}, \sigma) \Downarrow \sigma(DLPO_{FromLabels}(e_1, e_2 \dots e_n)/id)}{[\text{LpoBuilder}]}$$

As stated earlier, the *Let* expression evaluates expression e to a value v , then binds a variable id to the current environment σ .

$$\frac{(e, \sigma) \Downarrow v}{(\mathbf{let} \ id = e, \sigma) \Downarrow \sigma[v/id]} \quad [\text{Let}]$$

For the *Product* operation, the two operands $e1$ and $e2$ are evaluated first. After their values $v1$ and $v2$ are obtained, the $DLPO_{product}$ function is applied to get the final result.

$$\frac{(e1, \sigma) \Downarrow (v1, \sigma1) \quad (e2, \sigma1) \Downarrow (v2, \sigma2)}{(e1 \langle \rangle e2, \sigma) \Downarrow (DLPO_{product}(v1, v2), \sigma2)} \quad [\text{Product}]$$

For the *Split* operation, similar to the *Product* operation, two operands are evaluated, the objects $v1$ and $v2$ are obtained, where $v2$ is the pivot. Then the environment is updated with $e1$, which binds to the result of the *Split* operation (invoke $DLPO_{split}$ function).

Remark 3.2. In the original \mathcal{MOQA} language, we generally assume the pivot is the first element of the discrete partial order. In our language, the user can specify any element in the partial order as the pivot. Theoretically, this generalization does not affect the output random bags and the timing function. The \mathcal{MOQA} runtime system will also check whether $v2$ is an element of $v1$. Otherwise the system will raise an error.

$$\frac{\frac{e1 \text{ is identifier literal}}{(e1, \sigma) \Downarrow v1} \quad (e2, \sigma) \Downarrow v2 \quad v2 \in v1}{(e1 \succ \langle e2, \sigma) \Downarrow \sigma[DLPO_{split}(v1, v2)/e1]} \quad [\text{Split}]$$

Similar to the *Split* operation, the *Top* and *Bot* operations are operations associated to a $DLPO$ type object, and they all update environment bindings for the target identifier id .

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v1}}{(\sim id, \sigma) \Downarrow \sigma[DLPO_{top}(v1)/id]} \quad [\text{Top}]$$

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v1}}{(\sim id, \sigma) \Downarrow \sigma[DLPO_{bot}(v1)/id]} \quad [\text{Bot}]$$

The **PercM** operation can be applied to *SLPO* type objects. It also updates the environment binding for the target identifier *id*.

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v1}}{(\text{PercM}(id), \sigma) \Downarrow \sigma[SLPO_{PercM}(v1)/id]} \quad [\text{PercM}]$$

As stated earlier, the merge operation is not a standard *MOQA* operation but it is introduced here to help implementing Mergesort. It has the following semantics. The runtime system will check the operands to make sure they are all in sorted order.

$$\frac{\frac{e1 \text{ is identifier literal}}{(e1, \sigma) \Downarrow v1} \quad \frac{e2 \text{ is identifier literal}}{(e2, \sigma) \Downarrow v2}}{(\text{Merge}(e1, e2), \sigma) \Downarrow merge(v1, v2)} \quad [\text{Merge}]$$

Depending on the type of the target object, the *index* operation gets an element or a component of a *LPO*. *id* is evaluated first to get the target object. Then expression *e* is evaluated to get the index value *v2*, and the system will automatically cast it to an integer. The returned value for this expression is an element or a component which is placed at position *v2* in target object *v1*.

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v1} \quad (e, \sigma) \Downarrow v2}{(id[e], \sigma) \Downarrow v1[v2]} \quad [\text{Index}]$$

Next, the *slice* operation is an operation associated to *DLPO* objects. Both indices *e1* and *e2* are optional and are evaluated first. The value returned by this expression is a new *DLPO* object constructed by slicing the original object *v*.

$$\frac{\text{if } e1 \text{ exists } (e1, \sigma) \Downarrow i1 \quad \text{if } e2 \text{ exists } (e1, \sigma) \Downarrow i2 \quad \frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v}}{(id[e1 : e2], \sigma) \Downarrow v[i1 : i2]} \quad [\text{Slice}]$$

The assignment rule first validates identifier id existing in the environment, then it evaluates the right hand side expression e to a value v with possible modified environment $\sigma2$. Finally, the environment is updated with a new binding from id to value v .

$$\frac{id \text{ is identifier literal} \quad id \in \sigma \quad (e, \sigma) \Downarrow (v, \sigma2)}{(id = e, \sigma) \Downarrow \sigma2[v/id]} \quad [\text{Assign}]$$

The indexed assignment rule first evaluates identifier id to get the target object from the execution environment, then expression $e1$ is evaluated to get the index value. Next, the right hand side expression $e2$ is evaluated and might change the execution environment to $\sigma2$. Finally, the modified environment $\sigma2$ is returned.

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v1} \quad (e1, \sigma) \Downarrow i \quad (e2, \sigma) \Downarrow (v2, \sigma2)}{(id[e1] = e2, \sigma) \Downarrow \sigma2[v1[i] = v2/id]} \quad [\text{IndexedAssign}]$$

There are two types of `for` loops in \mathcal{MOQA} programming. One loops an integer variable from a smaller value to a bigger value, the other loops from a bigger value to a smaller value. Informally, the `for` loops work as follows. When entering the loop: `for $id = e1$ to/downto $e2$ do es end`, the expression $e1$ is evaluated to an integer $n1$ and the expression $e2$ is evaluated to an integer $n2$. If $n1 \geq n2$ ($n1 \leq n2$ in `downto` loop), the loop is terminated and the loop body is not evaluated. If $n1 < n2$ ($n1 > n2$ in `downto` loop), the body es is executed $n2 - n1$ ($n1 - n2$ in `downto` loop) times with id which binds the value $n1 + i - 1$ ($n1 - i + 1$ in `downto` loop) at the i th loop iteration.

Remark 3.3. To enable automated average-case analysis, updates to id inside the loops are not allowed, and this is generally achieved by our typing rule that only allows an update assignment to a LPO type object.

$$\frac{\begin{array}{c} (e1, \sigma) \Downarrow n1 \\ (e2, \sigma) \Downarrow n2 \\ n1 < n2 \\ (es, \sigma[n1/id]) \Downarrow \sigma2 \\ (\text{for } id = n1+1 \text{ to } e2 \text{ do } es \text{ end}, \sigma2) \Downarrow \sigma3 \end{array}}{(\text{for } id = e1 \text{ to } e2 \text{ do } es \text{ end}, \sigma) \Downarrow \sigma3} \quad [\text{For}]$$

$$\frac{\begin{array}{c} (e1, \sigma) \Downarrow n1 \\ (e2, \sigma) \Downarrow n2 \\ n1 > n2 \\ (es, \sigma[n1/id]) \Downarrow \sigma2 \\ (\text{for } id = n1-1 \text{ downto } e2 \text{ do } es \text{ end}, \sigma2) \Downarrow \sigma3 \end{array}}{(\text{for } id = e1 \text{ downto } e2 \text{ do } es \text{ end}, \sigma) \Downarrow \sigma3} \quad [\text{DowntoFor}]$$

The *MOQA* condition rule first evaluates *id* to target *LPO* object, then evaluates the numeric expression *e*. Finally it invokes *LPO* built-in function *size* to compare the size of a *LPO* object with a numeric value. The boolean result is returned.

$$\frac{\oplus \in \{>, >=, <, \Leftarrow\} \quad \frac{id \text{ is identifier literal}}{(id, \sigma) \Downarrow v} \quad (e, \sigma) \Downarrow n}{(|id| \oplus e, \sigma) \Downarrow LPO_{size}(v) \oplus n} \quad [\text{MoqaCond}]$$

The evaluation rules for arithmetic and logic expressions are standard and similar to other programming languages.

$$\frac{\begin{array}{c} (e1, \sigma) \Downarrow v1 \\ (e2, \sigma) \Downarrow v2 \\ \oplus \in \{+, -, *, /, \%\} \end{array}}{(e1 \oplus e2, \sigma) \Downarrow v1 \oplus v2} \quad [\text{ArithExpr}]$$

$$\frac{\begin{array}{c} (e1, \sigma) \Downarrow v1 \\ (e2, \sigma) \Downarrow v2 \\ \oplus \in \{and, or, xor\} \end{array}}{(e1 \oplus e2, \sigma) \Downarrow v1 \oplus v2} \quad [\text{LogicExpr}]$$

$$\frac{(e, \sigma) \Downarrow \text{false}}{(\text{not } e, \sigma) \Downarrow \text{true}} \quad \frac{(e, \sigma) \Downarrow \text{true}}{(\text{not } e, \sigma) \Downarrow \text{false}} \quad [\text{Not}]$$

The **if** expression evaluation rules are straightforward. The predicate is evaluated first, and depending on its boolean value, the **then** branch or the optional **else** branch is executed.

$$\frac{(e, \sigma) \Downarrow true \quad (es, \sigma) \Downarrow (v, \sigma 1)}{(if\ e\ do\ es\ end, \sigma) \Downarrow (v, \sigma 1)} \quad \frac{(e, \sigma) \Downarrow false}{(if\ e\ do\ es\ end, \sigma) \Downarrow \sigma} \quad [If-then]$$

$$\frac{(e, \sigma) \Downarrow true \quad (e_1, \sigma) \Downarrow (v, \sigma 1)}{(if\ e\ do\ e_1\ else\ do\ e_2\ end, \sigma) \Downarrow (v, \sigma 1)} \quad \frac{(e, \sigma) \Downarrow false \quad (e_2, \sigma) \Downarrow (v, \sigma 1)}{(if\ e\ do\ e_1\ else\ do\ e_2\ end, \sigma) \Downarrow (v, \sigma 1)} \quad [If-then-else]$$

Function definitions are not evaluated. They are stored as a closure in execution environment, which associates to the function name. The function body is evaluated when the function is invoked with actual parameters.

Definition 3.2. A closure consists of a mapping from a sequence of variables (the input variables) to an expression (the function body) and an environment where the function is defined. We write a closure in our execution environment as follows:

$$\langle (p_1, \dots, p_n) \mapsto exp, \sigma \rangle$$

So, we define a function in environment σ , its input formal parameters are p_1, \dots, p_n and its function body is exp . We can associate a closure to a function name in the execution environment.

Example 3.4. $\sigma[\langle (p) \mapsto exp, \sigma \rangle / qsort]$ means we have a function named $qsort$ in environment σ . $qsort$ is also defined in environment σ , it has one input variable p , and the function body is exp .

$$(\text{def id } (p_1, \dots, p_n) \text{ es end, } \sigma) \Downarrow \sigma[\langle (p_1, \dots, p_n) \mapsto es, \sigma \rangle / id] \quad [DefFun]$$

Remark 3.4. In our execution environment we treat functions as first-class values, similar to other dynamic programming languages, such as Python or JavaScript.

As a result, we should be able to define higher-order functions (functions taking other functions as input or output) [46], but in the current status of *MOQA* research, those functions are not allowed. Also nested function definitions are not allowed in our context.

To facilitate automated average-case analysis, currently only two types of function definition are allowed (see Section 3.3.3.3 for details).

The rule for function call is also similar to other programming languages. We begin by evaluating the receiver *id* to get the associated function definition closure from environment. After making sure the number of actual parameters are equal to the number of formal parameters, we then evaluate the actual parameters a_1 through to a_n in order to get their values. Finally, we evaluate the function body with possible modified environment σ_2 , that has bindings from formal parameters to actual parameters. Whatever is returned is the value of the function call.

$$\frac{\frac{\textit{id} \text{ is identifier literal}}{(id, \sigma) \Downarrow \langle (p_1, \dots, p_n) \mapsto \textit{body}, \sigma_2 \rangle} \quad k = n}{\begin{array}{c} (e_1, \sigma) \Downarrow a_1 \quad \dots \quad (e_k, \sigma) \Downarrow a_k \\ (\textit{body}, \sigma_2[a_1/p_1, \dots, a_n/p_n]) \Downarrow v \end{array}}{(id (e_1, \dots, e_k), \sigma) \Downarrow v} \quad \text{[FunctionCall]}$$

The remaining three expressions are straightforward, we list them below:

$$\frac{(e, \sigma) \Downarrow v}{(\text{return } e, \sigma) \Downarrow \textit{stop}(v)} \quad \text{[Return]}$$

$$\frac{(e, \sigma) \Downarrow v}{(\text{print}(e), \sigma) \Downarrow \textit{print}(v)} \quad \text{[Print]}$$

$$\frac{(e, \sigma) \Downarrow v}{(\text{show}(e), \sigma) \Downarrow \textit{LPO}_{\textit{show}}(v)} \quad \text{[Show]}$$

Remark 3.5. In the return expression, there is a special function *stop*. Remember that after a return statement we do not want to continue executing the statements that come after it. We want to jump back to the caller. We use this helper function to indicate that. In a real implementation, it is implemented with exceptions (see Section 4.7).

The `show` expression, invokes a *LPO* built-in method `show`, while the `print` expression uses Python’s built-in method `print`.

3.5.2 *MOQA* Analysis Semantics

As stated earlier, there are two semantics for *MOQA* programs, one used in execution mode, the other used in analysis mode. In the following, we discuss the *MOQA* analysis semantics. When the *MOQA* interpreter analyses the average-case complexity of a user defined function, it can be viewed as an abstract evaluation of the function using our analysis semantics.

The main differences between analysis semantics and execution semantics are as follows:

- Execution semantics is applied to the whole program, while analysis mode is only applied to expressions or statements that are inside a function definition. That is to say, analysis semantics are used to automatically calculate average cost for a user defined function in a *MOQA* program.
- Execution semantics uses *MOQA* operations at the level of a single *LPO* object, while analysis semantics uses operations at the level of random bags.
- Environment in execution mode maps identifiers to their values, where the value type could be a primitive type, a *LPO* type or a function closure. In analysis mode, environment maps identifiers to objects of a primitive type or of a random bag type. Random bag objects are used to help calculate cost for a particular *MOQA* operation (using Theorem 2.4).
- Besides the environment, analysis semantics also keeps track of how much cost has been made and the value is stored in a special variable ϵ .
- Because the input parameters for a user defined function are always a discrete partial order together with an optional number, we specify the size of discrete partial order using \mathcal{N} and an optional number, denoting for instance the rank of an element, using \mathcal{K} .

Remark 3.6. The parameter \mathcal{N} and \mathcal{K} are provided by the user to the interpreter analyzer at command line.

Analysis rules in analysis semantics also use logic rules of inference, quite similar to execution semantics' rules. We also use σ to represent our environment, and configurations are changed from a pair to a triple, e.g. (e, σ, ϵ) .

$(e, \sigma, \epsilon) \Downarrow (v, \sigma[val/id], \epsilon')$ can be read as: in the analysis environment σ with accumulated ϵ number of comparisons, expression e evaluates to a value v , and has a side-effect that changes the environment to $\sigma[val/id]$. The number of comparisons made so far after evaluation is ϵ' . Notice that in the configuration returned, the first two parts are both optional, but the last one, ϵ' , is required.

In the following, we give formal operational semantics for the analysis of *MOQA* programs in analysis mode. This describes how the interpreter analyzer analysis average-case cost for a user defined function. We refer the reader to Section 4.8 for details of the implementation for these rules.

As before, we use $TYPE_{name}$ to represent helper functions. Analysis mode uses a different runtime library. This time, the input and output for the functions are both random bags, e.g.

$$DLPO_{split}(R(\Delta_3)) \mapsto \{(R(\vee_3), 1), (R(S_3), 2), (R(\wedge_3), 1)\}$$

Recall from Theorem 2.4, given a random bag: $R = \{(R_1, K_1), \dots, (R_n, K_n)\}$, we use following equation to derive the average-case cost for an operation P :

$$\bar{T}_P(R) = \sum_{i=1}^n Prob_i \times \bar{T}_P(R_i)$$

where $Prob_i = Prob[F \in R_i] = \frac{K_i|R_i|}{\sum_{i=1}^n K_i|R_i|} = \frac{K_i|R_i|}{|R|}$. And $Prob_i$ can be viewed as the probability associated to the i^{th} random structure.

In our implementation, this probability is not calculated on the fly. To reduce overhead, once a random bag is created. Each random structure will be tagged with its probability. Thus the random bag is extended to : $R = \{(R_1, K_1, P_1), \dots, (R_n, K_n, P_n)\}$ where P_i is i^{th} random structure's probability. We then use the same equation to derive the average-cost of an operation over a random bag.

Let's start with the rule for defining a function. It is the entry point to analyse the average-cost of a user defined function.

Remark 3.7. $DLPO_{create}$ is a library function used to create a random bag with a single random structure $R(\Delta_{\mathcal{N}})$.

$$\frac{\text{set } \epsilon = 0 \quad (es, \sigma[DLPO_{create}(\mathcal{N})/p], \epsilon) \Downarrow (v, \epsilon')}{(\text{def id } (p) \text{ es end}, \sigma, \epsilon) \Downarrow (v, \epsilon')} \quad [\text{DefFun-One}]$$

$$\frac{\text{set } \epsilon = 0 \quad (es, \sigma[DLPO_{create}(\mathcal{N})/p_1, \mathcal{K}/p_2], \epsilon) \Downarrow (v, \epsilon')}{(\text{def id } (p_1, p_2) \text{ es end}, \sigma, \epsilon) \Downarrow (v, \epsilon')} \quad [\text{DefFun-Two}]$$

In both type of function definitions, we first reset ϵ before evaluating the average-case cost for this function. Then the function body es is evaluated under a new environment, where formal parameter p_1 binds to random structure $R(\Delta_{\mathcal{N}})$ and p_2 binds to number \mathcal{K} if p_2 exists. Finally, the function body es is evaluated to a value v with total number of comparisons ϵ' .

Most rules in analysis semantics are quite close to execution semantics rules, but with additional parameter ϵ . We first list all the rules that do not change this additional parameter, that is to say, the language constructs that do not contribute to the number of comparisons.

$$\frac{b \in \{\mathbf{true}, \mathbf{false}\}}{(b, \sigma, \epsilon) \Downarrow (b, \epsilon)} \quad [\text{Bool}]$$

$$\frac{\mathbf{n} \text{ is a number literal}}{(n, \sigma, \epsilon) \Downarrow (n, \epsilon)} \quad [\text{Num}]$$

$$\frac{\mathbf{s} \text{ is a string literal}}{(s, \sigma, \epsilon) \Downarrow (s, \epsilon)} \quad [\text{String}]$$

$$\frac{\begin{array}{l} id \text{ is identifier literal} \\ id \in \sigma \end{array}}{(id, \sigma, \epsilon) \Downarrow (\sigma(id), \epsilon)} \quad [\text{Var}]$$

The rules for constants and identifiers are not making additional comparisons, thus ϵ is not changed in these rules.

The expression list simply passes environment σ and accumulates average-case

cost ϵ from the first expression to the last one.

$$\frac{\begin{array}{c} (e_1, \sigma, \epsilon) \Downarrow (v_1, \sigma_1, \epsilon_1) \\ (e_2, \sigma_1, \epsilon_1) \Downarrow (v_2, \sigma_2, \epsilon_2) \\ \vdots \\ (e_n, \sigma_{n-1}, \epsilon_{n-1}) \Downarrow (v_n, \sigma_n, \epsilon_n) \end{array}}{(e_1 e_2 \dots e_n, \sigma, \epsilon) \Downarrow (v_n, \sigma_n, \epsilon_n)} \quad [\text{Sequence}]$$

Most rules do not directly contribute to the number of comparisons. Instead, they are dependent on evaluating their sub-expressions and keeping track of changes on ϵ . In comparison with execution semantics, these rules simply add an additional parameter ϵ to their configurations and recursively update their values.

$$\frac{(e, \sigma, \epsilon) \Downarrow (v, \epsilon')}{(\text{let } id = e, \sigma, \epsilon) \Downarrow (\sigma[v/id], \epsilon')} \quad [\text{Let}]$$

$$\frac{(e, \sigma, \epsilon_1) \Downarrow (v_2, \epsilon_2) \quad \frac{id \text{ is identifier literal}}{(id, \sigma, \epsilon) \Downarrow (v_1, \epsilon_1)}}{(id[e], \sigma, \epsilon) \Downarrow (v_1[v_2], \epsilon_2)} \quad [\text{Index}]$$

$$\frac{\frac{\text{if } e1 \text{ exists}}{(e1, \sigma, \epsilon) \Downarrow (i1, \epsilon_1)} \quad \frac{\text{if } e2 \text{ exists}}{(e1, \sigma, \epsilon_1) \Downarrow (i2, \epsilon_2)} \quad \frac{id \text{ is identifier literal}}{(id, \sigma, \epsilon_2) \Downarrow (v, \epsilon_3)}}{(id[e1 : e2], \sigma, \epsilon) \Downarrow (v[i1 : i2], \epsilon_3)} \quad [\text{Slice}]$$

$$\frac{id \text{ is identifier literal} \quad id \in \sigma \quad (e, \sigma, \epsilon) \Downarrow (v, \sigma_2, \epsilon_1)}{(id = e, \sigma, \epsilon) \Downarrow (\sigma_2[v/id], \epsilon_1)} \quad [\text{Assign}]$$

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma, \epsilon) \Downarrow (v_1, \epsilon_1)} \quad (e1, \sigma, \epsilon_1) \Downarrow (i, \epsilon_2) \quad (e2, \sigma, \epsilon_2) \Downarrow (v_2, \sigma_2, \epsilon_3)}{(id[e1] = e2, \sigma, \epsilon) \Downarrow (\sigma_2[v_1[i] = v_2/id], \epsilon_3)} \quad [\text{IndexedAssign}]$$

$$\frac{\begin{array}{l} (e1, \sigma, \epsilon) \Downarrow (v1, \epsilon) \\ (e2, \sigma, \epsilon) \Downarrow (v2, \epsilon) \\ \oplus \in \{+, -, *, /, \%\} \end{array}}{(e1 \oplus e2, \sigma, \epsilon) \Downarrow (v1 \oplus v2, \epsilon)} \quad [\text{ArithExpr}]$$

$$\frac{\begin{array}{l} (e1, \sigma, \epsilon) \Downarrow (v1, \epsilon_1) \\ (e2, \sigma, \epsilon_1) \Downarrow (v2, \epsilon_2) \\ \oplus \in \{\text{and}, \text{or}, \text{xor}\} \end{array}}{(e1 \oplus e2, \sigma, \epsilon) \Downarrow (v1 \oplus v2, \epsilon_2)} \quad [\text{LogicExpr}]$$

$$\frac{(e, \sigma, \epsilon) \Downarrow (\mathbf{false}, \epsilon_1)}{(\mathbf{not} \ e, \sigma, \epsilon) \Downarrow (\mathbf{true}, \epsilon_1)} \quad \frac{(e, \sigma, \epsilon) \Downarrow \mathbf{true}, \epsilon_1}{(\mathbf{not} \ e, \sigma, \epsilon) \Downarrow (\mathbf{false}, \epsilon_1)} \quad [\text{Not}]$$

$$\frac{(e, \sigma, \epsilon) \Downarrow (v, \epsilon_1)}{(\mathbf{return} \ e, \sigma, \epsilon) \Downarrow (\mathit{stop}(v), \epsilon_1)} \quad [\text{Return}]$$

$$\frac{(e, \sigma, \epsilon) \Downarrow (v, \epsilon_1)}{(\mathbf{print}(\ e), \sigma, \epsilon) \Downarrow (\mathit{print}(v), \epsilon_1)} \quad [\text{Print}]$$

$$\frac{(e, \sigma, \epsilon) \Downarrow (v, \epsilon_1)}{(\mathbf{show}(\ e), \sigma, \epsilon) \Downarrow (\mathit{LPO}_{\mathit{show}}(v), \epsilon_1)} \quad [\text{Show}]$$

In the *MOQA* condition rule, besides comparisons from sub-expressions, the expression itself also contributes one comparison. Thus the final result is the accumulated comparisons (ϵ_2) plus one.

$$\frac{\oplus \in \{>, >=, <, \Leftarrow\} \quad \frac{\textit{id} \text{ is identifier literal}}{(id, \sigma, \epsilon) \Downarrow (v, \epsilon_1)} \quad (e, \sigma, \epsilon_1) \Downarrow (n, \epsilon_2)}{(|id| \oplus e, \sigma, \epsilon) \Downarrow (\mathit{LPO}_{\mathit{size}}(v) \oplus n, \epsilon_2 + 1)} \quad [\text{MoqaCond}]$$

The comparisons contributed by *MOQA* expressions are defined by timing functions associated to each *MOQA* basic operations (see Chapter 2). We refer to these timing functions as \overline{T}_{name} , where *name* is the operation name. All timing functions take a random bag as input and produce the average number of comparisons needed after applying the operation to the input random bag. e.g. $\overline{T}_{\mathit{split}}(R(\Delta_{\mathcal{N}})) = \mathcal{N} - 1$.

$$\frac{(e1, \sigma, \epsilon) \Downarrow (v1, \sigma1, \epsilon1) \quad (e2, \sigma1, \epsilon1) \Downarrow (v2, \sigma2, \epsilon2)}{(e1 \ltimes e2, \sigma, \epsilon) \Downarrow (LPO_{product}(v1, v2), \sigma2, \epsilon2 + \overline{T}_{product}(v1, v2))} \quad [\text{Product}]$$

$$\frac{\frac{e1 \text{ is identifier literal}}{(e1, \sigma, \epsilon) \Downarrow (v1, \epsilon1)} \quad (e2, \sigma, \epsilon1) \Downarrow (v2, \epsilon2)}{(e1 \times e2, \sigma, \epsilon) \Downarrow (\sigma[DLPO_{split}(v1, v2)/e1], \epsilon2 + \overline{T}_{split}(v1))} \quad [\text{Split}]$$

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma, \epsilon) \Downarrow (v1, \epsilon1)}}{(\sim id, \sigma, \epsilon) \Downarrow (\sigma[DLPO_{top}(v1)/id], \epsilon1 + \overline{T}_{top}(v1))} \quad [\text{Top}]$$

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma, \epsilon) \Downarrow (v1, \epsilon1)}}{(\sim id, \sigma, \epsilon) \Downarrow (\sigma[DLPO_{bot}(v1)/id], \epsilon1 + \overline{T}_{bot}(v1))} \quad [\text{Bot}]$$

$$\frac{\frac{id \text{ is identifier literal}}{(id, \sigma, \epsilon) \Downarrow (v1, \epsilon1)}}{(\text{PercM}(id), \sigma, \epsilon) \Downarrow (\sigma[SLPO_{PercM}(v1)/id], \epsilon1 + \overline{T}_{PercM}(v1))} \quad [\text{PercM}]$$

To analyse for-loops in \mathcal{MOQA} programs, we accumulate the average-case cost in the loop body for each iteration. Notice that we do not count loop conditional checking for consistency with the original \mathcal{MOQA} theory.

$$\frac{\begin{array}{l} (e1, \sigma, \epsilon) \Downarrow (n1, \epsilon1) \\ (e2, \sigma, \epsilon1) \Downarrow (n2, \epsilon2) \\ n1 < n2 \\ (es, \sigma[n1/id], \epsilon2) \Downarrow (\sigma2, \epsilon3) \end{array}}{(\text{for } id = n1+1 \text{ to } e2 \text{ do } es \text{ end}, \sigma2, \epsilon3) \Downarrow (\sigma3, \epsilon4)} \quad [\text{For}]$$

$$\frac{}{(\text{for } id = e1 \text{ to } e2 \text{ do } es \text{ end}, \sigma, \epsilon) \Downarrow (\sigma3, \epsilon4)}$$

$$\frac{
\begin{array}{c}
(e1, \sigma, \epsilon) \Downarrow (n1, \epsilon_1) \\
(e2, \sigma, \epsilon_1) \Downarrow (n2, \epsilon_2) \\
n1 > n2 \\
(es, \sigma[n1/id], \epsilon_2) \Downarrow (\sigma 2, \epsilon_3) \\
(\text{for } id = n1-1 \text{ downto } e2 \text{ do } es \text{ end}, \sigma 2, \epsilon_3) \Downarrow (\sigma 3, \epsilon_4)
\end{array}
}{
(\text{for } id = e1 \text{ downto } e2 \text{ do } es \text{ end}, \sigma, \epsilon) \Downarrow (\sigma 3, \epsilon_4)
} \quad [\text{DowntoFor}]$$

The **if-expression** rule is the most complicated rule in our analysis semantics. To calculate the average-case cost, for a random bag used in an **if** expression we need to know the probability that the predicate evaluates to true. In other words, how many random structures fall into the **then** branch, and how many belongs to the **else** branch.

In *MOQA* programming, the sub-expression e must be a *MOQA* conditional expression. In analysis mode, it involves comparing the size of a random structure with a numeric number. Here we introduce a helper function *filter*. It takes a *MOQA* conditional expression e and a environment σ . The helper function is used to separate the random structures in the random bag into two groups, where one group contains random structures that satisfy predicate e , the other group contains structures that do not satisfy predicate e .

Consider a random bag $R = \{(R_1, K_1, P_1), \dots, (R_n, K_n, P_n)\}$ involved in expression e . After $filter(e, \sigma)$, the result will be (X, R^t, R^f) where X is the identifier which binds to the random bag R , R^t is a random bag containing all random structures satisfying predicate e , R^f consists of random structures that do not satisfy predicate e .

Example 3.5. Given a random bag: $R = \{(\emptyset, 1, \frac{1}{3}), (\bullet, 2, \frac{1}{3}), (\bullet\bullet, 1, \frac{1}{3})\}$ bound to variable X .

The first random structure is an empty structure, the second structure is a single node, the last structure is a two node discrete random structure $R(\Delta_2)$. Every random structure has a probability $\frac{1}{3}$.

If boolean condition e is **if** $|X| \leq 1$ **do**, then $filter(e, \sigma) = (X, R^t, R^f)$, where $R^t = \{(\emptyset, 1, \frac{1}{3}), (\bullet, 2, \frac{1}{3})\}$, $R^f = \{(\bullet\bullet, 1, \frac{1}{3})\}$. In this example, we can see that the probability to execute the **then-branch** is $\frac{1}{3} + \frac{1}{3} = \frac{2}{3}$, probability to execute the **else-branch** is $\frac{1}{3}$.

To derive the average-case cost for an **if-expression**, we need to combine the cost both in **then-branch** and **else-branch**, together with their probabilities. We accomplish this by the following steps:

- Rebind X to R^t , evaluating the **then-branch** in the new environment.
- In case **then-branch** modifies R^t , bind $_X$ to current R^t (similar tricks are used in type checking), rebind X to R^f , evaluating the **else-branch** in the new environment.
- Rebind X to the union of current R^t and R^f .
- The final returned random bag should unite the random bags produced by both branches, and the returned final cost is the accumulate cost ϵ plus one (in order to count in the comparison made by the **if-predicate**).

Each random structure has a probability associated with it. By limiting random structures involved in evaluating both branches, we derive the cost for each branch separately. After evaluating both branches, we restore random structures (random bag) by rebinding it to X , and continue with the other evaluation steps.

Remark 3.8. If there is no **else-branch**, just skip the second step.

The formal semantics rules for **if-expressions** are shown below. We present an application of this expression in the Quickselect algorithm (see Section 5.3).

$$\frac{filter(e, \sigma) = (X, R^t, R^f) \quad (es, \sigma[R^t/X], \epsilon) \Downarrow (v, \sigma 1, \epsilon_1)}{(\mathbf{if } e \mathbf{ do } es \mathbf{ end}, \sigma, \epsilon) \Downarrow (v, \sigma 1[\sigma 1[X] \cup R^f/X], \epsilon_1 + 1)} \quad [\mathbf{If-then}]$$

$$\frac{\begin{array}{l} filter(e, \sigma) = (X, R^t, R^f) \\ (e_1, \sigma[R^t/X], \epsilon) \Downarrow (v_1, \sigma 1, \epsilon_1) \\ (e_2, \sigma 1[R^f/X, R^t/_X], \epsilon_1) \Downarrow (v_2, \sigma 2, \epsilon_2) \end{array} \quad [\mathbf{If-then-else}]}{(\mathbf{if } e \mathbf{ do } e_1 \mathbf{ else do } e_2 \mathbf{ end}, \sigma, \epsilon) \Downarrow (v_1 \cup v_2, \sigma 2[\sigma 2[X] \cup \sigma 2[_X]/X], \epsilon_2 + 1)}$$

The last rule is a function call. Recall that the entry point to time analysis is a function definition rule. The user needs to provide \mathcal{N} , the size for the initial discrete partial order, and an optional number \mathcal{K} .

In our implementation, the analyzer is implemented by a procedure called $\mathcal{MOQA}_{process}$, then $\mathcal{MOQA}_{process}(fun, \mathcal{N}, \mathcal{K})$ produces average-case cost for a function fun with numbers \mathcal{N} and \mathcal{K} .

Theorem 3.1. *Given a function call $fun(X, k)$, where X is a random bag $R = \{(R_1, K_1, P_1), \dots, (R_n, K_n, P_n)\}$. The average-case cost for this function call is:*

$$\bar{T}(fun(X, k)) = \sum_{i=1}^n P_i \times \mathcal{MOQA}_{process}(fun, |R_i|, k)$$

where $|R_i|$ is the size of random stricture R_i .

Remark 3.9. When the function only has one input, just ignore the second parameter k .

To help present our semantics rule, we introduce a helper function def which maps function name to its definition. In our interpreter, it is implemented by first walking over the abstract syntax tree to obtain all the mappings from the function name to the function definition.

$$\frac{\begin{array}{c} (e, \sigma, \epsilon) \Downarrow (a, \epsilon_1) \\ a = \{(R_1, K_1, P_1), \dots, (R_n, K_n, P_n)\} \\ \mathcal{MOQA}_{process}(def(a), \mathcal{N} = |R_1|) \Downarrow (v_1, \epsilon_2) \\ \epsilon_1 + = P_1 * \epsilon_2 \\ \vdots \\ \mathcal{MOQA}_{process}(def(a), \mathcal{N} = |R_n|) \Downarrow (v_n, \epsilon_{n+1}) \\ \epsilon_1 + = P_n * \epsilon_{n+1} \end{array}}{(id (e), \sigma, \epsilon) \Downarrow (\cup_{i=1}^n v_i, \epsilon_1)} \quad [\text{FunctionCall-One}]$$

$$\frac{\begin{array}{c} (e_1, \sigma, \epsilon) \Downarrow (a, \epsilon_1) \\ (e_2, \sigma, \epsilon_1) \Downarrow (b, \epsilon_2) \\ a = \{(R_1, K_1, P_1), \dots, (R_n, K_n, P_n)\} \\ \mathcal{MOQA}_{process}(def(a), \mathcal{N} = |R_1|, \mathcal{K} = b) \Downarrow (v_1, \epsilon_3) \\ \epsilon_2 + = P_1 * \epsilon_3 \\ \vdots \\ \mathcal{MOQA}_{process}(def(a), \mathcal{N} = |R_n|, \mathcal{K} = b) \Downarrow (v_n, \epsilon_{n+2}) \\ \epsilon_2 + = P_n * \epsilon_{n+2} \end{array}}{(id (e_1, e_2), \sigma, \epsilon) \Downarrow (\cup_{i=1}^n v_i, \epsilon_2)} \quad [\text{FunctionCall-Two}]$$

3.6 *MOQA* Programming Restrictions

As stated earlier, there are several restrictions to *MOQA* programming. We list these below:

- Only two types of function definitions are allowed.
- Inside a function definition, *LPO* builder is not available, that is to say, there is only one partial order. The programmer uses *MOQA* basic operations together with restricted control flows to define an algorithm that manipulates the initial partial order.
- No nested function definitions are allowed.
- Higher order functions are not supported.
- User defined type is not available.
- Programming with restricted `for` loops and `if` expression.
- No `while` construct.
- Because no new partial order is created inside a function, recursive call is over parallel or series components of the initial partial order. Formally this is called parallel-recursion or series-recursion, and is defined in [85].

It may seem like a lot of restrictions, but we will show later that most of the sorting and searching algorithms can be implemented by our language. Furthermore, it supports accurate automated average-case analysis. We would like to continue expanding the usability and application domain for our language in future research.

3.7 Summary

In this chapter we presented the *MOQA* language specification in a formal way. The design of the language stays close in spirit to the normal imperative language. In addition, by adapting clear and expressive syntax, our language is

more concise and elegant compared with the original *MOQA* library implementation [107]. Based on the design of two semantics, we propose a new approach to automated average-case analysis differing from our old approach [48]. The analysis is effective and easier. All the formal work we presented here lays out a solid foundation for practical implementation, and makes it possible to be implemented in any programming language. In Chapter 4 we provide a sample prototype implementation using Python.

Chapter 4

MOQA Language Implementation

Contents

4.1	Python Lex-Yacc	100
4.2	<i>MOQA</i> Interpreter Architecture	101
4.3	Lexer	104
4.4	Parser	104
4.5	Environment Implementation	108
4.6	Type Checker	109
4.7	Execution Engine	111
4.8	Analysis Engine	115
4.8.1	Dynamic programming in ACETAnalysis	117
4.9	Summary	127

In this chapter, we look at the *MOQA* language from a practical point of view, and its associated interpreter implementation details and architecture are discussed. In particular, we focus on how it is implemented in the Python Lex-Yacc framework. The work presented in this section depends on Python language features, but implementing the *MOQA* language with other programming languages should be similar. All the implementations should follow the formal specification we gave in Chapter 3.

At the beginning, in Section 4.1, we give a brief introduction on Python Lex-Yacc and general parsing techniques. Next, in Section 4.2, the *MOQA* interpreter architecture is discussed and the role of each component is explained. In the following two sections, we present how the front-end of the interpreter is implemented. The focus for these two sections is to convert a source code to an internal abstract syntax tree (AST) representation. After that, the environment implementation is discussed in Section 4.5. Based on the environment implementation, the remaining three components are discussed in the following sections. They are mainly focused on converting the formal rules we defined in Chapter 3 to real code and on how automated time analysis is achieved with the help of dynamic programming. Finally, we present a short summary in Section 4.9.

4.1 Python Lex-Yacc

To implement an interpreter, the parser is one of the most important components in the overall structure. The job of a parser is to check for syntax correctness in the input program, and to build an abstract syntax tree out of the input tokens.

Given a language grammar specification to the parser, it needs to determine if and how the input tokens will be derived from the start symbol of the language. Based on the ways to parse input tokens, there are generally two major parsing approaches [13, 15]:

- Top-down parsing: begin with the start symbol and apply BNF rules by replacing left-hand side non-terminal to right-hand side strings until one arrives at the input string. Generally, it generates a *leftmost derivation*.
- Bottom-up parsing: starts with the input string, and tries to reduce the in-

put string back to the start symbol. It uses BNF rules in reverse order, that is it replaces right-hand side strings to their left-hand side non-terminal. Generally, it generates a *rightmost derivation* (usually in reverse).

LL parsers and recursive-descent parsers are both examples of top-down parsers, while LR, LALR parsers and CYK parsers are examples of bottom-up parsers.

For efficiency and maintenance, sometimes these parsers are hand written, e.g. GCC [2], but there are parser generators. These could generate a parser if we provided grammar specifications to them. Most of time, these generated parsers are good enough (Ruby uses a parser generator [6]).

ANTLR [75] is one example of parser generator. It can be used in several languages (such as C, Python, Java etc). It uses a modified LL parsing algorithm called LL(*). Because of the drawbacks with LL parsing, the user has to rewrite their grammar to eliminate left recursion in the BNF rules.

Besides ANTLR, there are traditional C languages with Lex-Yacc tool combination to implement the interpreter or compiler. Lex is used to generate Lexer, while Yacc is used to generate the LALR parser. In our project, we use Python Lex-Yacc or in short PLY. It is an implementation of Lex and Yacc parsing tools for Python [4]. Unlike the original C version, it doesn't need a separate explicit parser generation step. By heavily relying on reflection, it allows on-the-fly parser generation, thus enabling a shorter implementation to testing cycles and suits our prototype creation demands.

4.2 *MOQA* Interpreter Architecture

In Figure 4.1, we show the architecture of the *MOQA* interpreter. Similar with other interpreters, it consists of several standard components: a lexer, a parser, and a typechecker. Different from others, instead of a single evaluator, the *MOQA* interpreter has two engines, which implement the two execution modes for a *MOQA* program.

The first three components, lexer, parser and typechecker are generally called front-end in a compiler or interpreter system. The main job for this part is to produce a verified and concise tree based representation of the input program,

and it is usually called Abstract Syntax Tree (AST).

In a general compiler architecture, there is also a part called back-end. The back-end is used to perform several code analyses and optimizations before finally generating code into the target language. For a common interpreter, the back-end is used to directly execute the program by recursive descent interpreting the AST generated by the front-end. Our interpreter not only provides direct program execution, it also provides an automated average-case analysis mode by abstract interpreting a *MOQA* program based on *MOQA* theory.

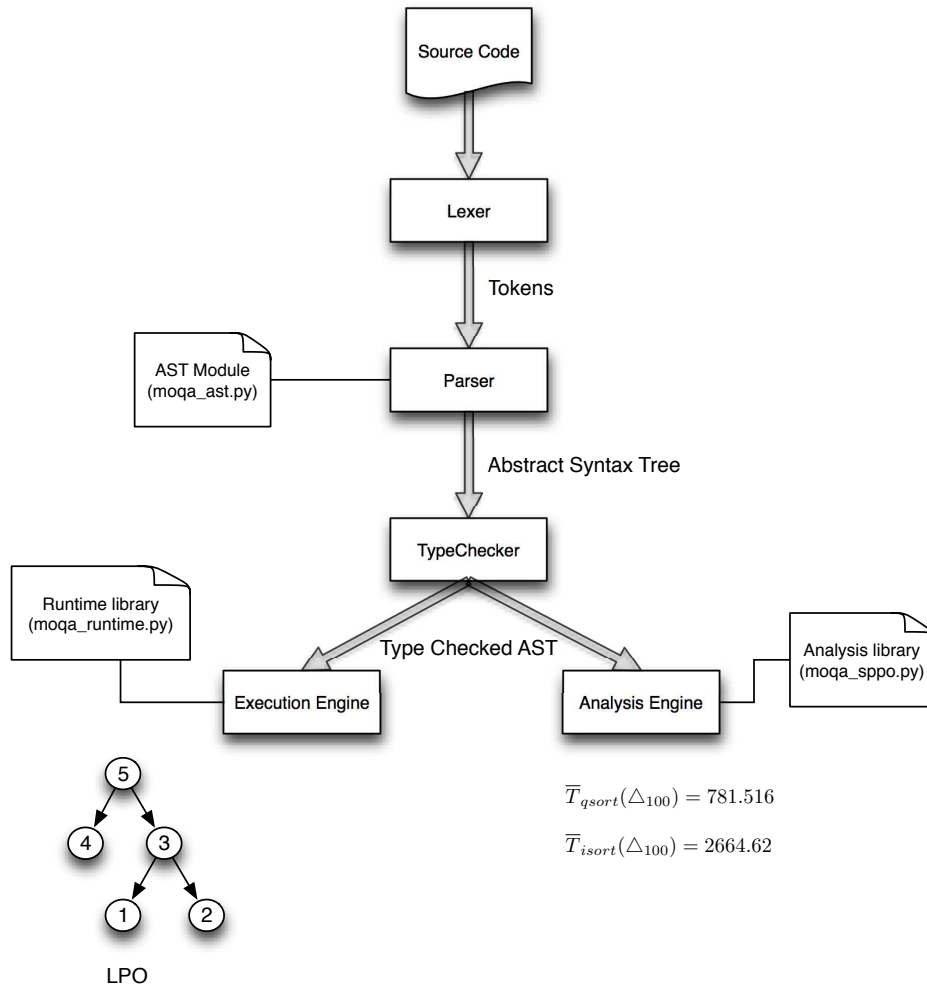


Figure 4.1: *MOQA* Interpreter Architecture

In our implementation, we adapt the method suggested by Matthieu at the

Python Conference 2010 (PyCon 2010) [14]. The decomposition of an interpreter is achieved by the Python decorator feature. As a result, our implementation has a clear separation between different interpreter components. Each component in the architecture is implemented by a related Python module (simply a `.py` file).

Here we list the role for each component in our interpreter architecture and their corresponding Python module implementations.

- Lexer takes a *MOQA* source code and transforms the sequence of characters into a sequence of tokens, where a token is specified using a regular expression. The corresponding Python module is `moqa_tokens.py`.
- Parser performs context-free syntax analysis to the input sequence of tokens. It identifies the grammatical structure of an input program by building an abstract syntax tree. All the related tree classes are defined in the AST module (`moqa_ast.py`). The corresponding Python module for the parser is `moqa_grammar.py`.
- Typechecker performs type checking to the input program by a recursive descent type check of the AST. Using the typing rules we defined earlier, it prevents common type errors, such as to apply the *MOQA* operations on a *non-LPO* object. The output after this step is a type checked abstract syntax tree. The corresponding Python module is `moqa_typecheck.py`.
- The execution engine behaves similar to an evaluator, relying on the *MOQA* runtime library (`moqa_runtime.py`), it executes statements and expressions in a program directly. The corresponding Python module is `moqa_interp.py`.
- The analysis engine is a special component in the *MOQA* interpreter. Based on the *MOQA* analysis library (`moqa_sppo.py`), it abstractly interprets *MOQA* programs in terms of basic *MOQA* operations and random bags. It keeps tracking data structures and their associated probabilities. Timing functions defined in basic *MOQA* operations are used to help the automated average-case analysis of input programs. The corresponding Python module is `moqa_analyzer.py`.

4.3 Lexer

In the following, we define four tokens in the *MOQA* language. These are discussed in the code fragment below. As expected, tokens are specified using regular expressions. Following naming conventions in PLY, function names start with `t`, followed by the token name. For example, token `NUMBER` is defined by function `t_NUMBER`. For simple tokens, we can even write one line of code by just providing their regular expressions (see Listing 4.1 lines 11–12). Other tokens are recognized by functions and the token values can be manipulated before returning to the caller, for example getting rid of `"` for a string.

Listing 4.1: *MOQA* Language Lexer Fragment

```
1 def t_NUMBER(t):
2     r' -?[0-9]+(\.[0-9]*)?'
3     t.value = float(t.value)
4     return t
5
6 def t_STRING(t):
7     r'"([\\"\\]|(\\.))*"'
8     t.value = t.value[1:-1] # strip off "
9     return t
10
11 t_GT = r'>'
12 t_GE = r'>='
```

Here we only list a small fraction of the *MOQA* lexer. We refer the reader to Appendix A Listing A.1 for full details.

4.4 Parser

In module `moqa_grammar.py`, each grammar rule we defined in Section 3.3.5 is implemented by one or several corresponding functions. The job for the parser is to produce an abstract syntax tree for a valid *MOQA* program or to reject it if

there are any syntax errors.

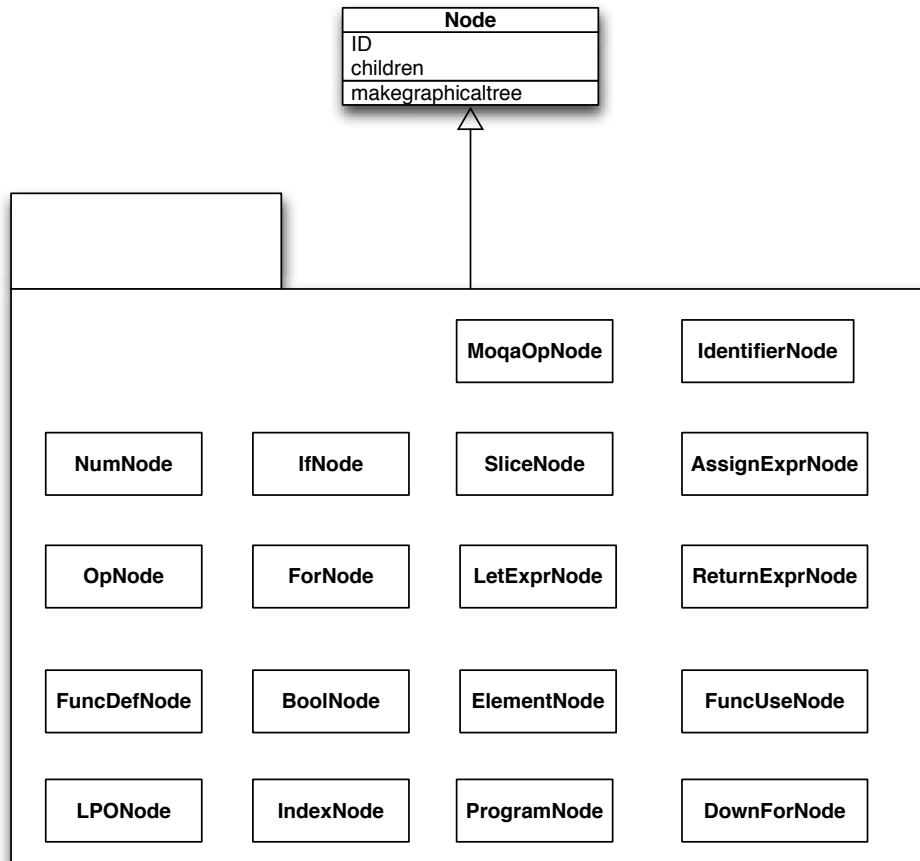


Figure 4.2: *MOQA* abstract syntax tree class hierarchy

In Figure 4.2, we show the class hierarchy of the abstract syntax tree module. All the classes are implemented in module `moqa_ast.py`. The core class of AST module is the `Node` class. It is simply a container for a list of children nodes with a unique ID for each object. The remaining classes are specialized node classes. They all extend the base class `Node`. Different subclasses represent different types of code fragments.

The abstract syntax tree module provides a graphical representation of an AST by employing Graphviz [9] software. We access Graphviz by using its python binding pydot [10].

Recall from Section 3.2.4.2, that the user can provide a `-D` argument to our

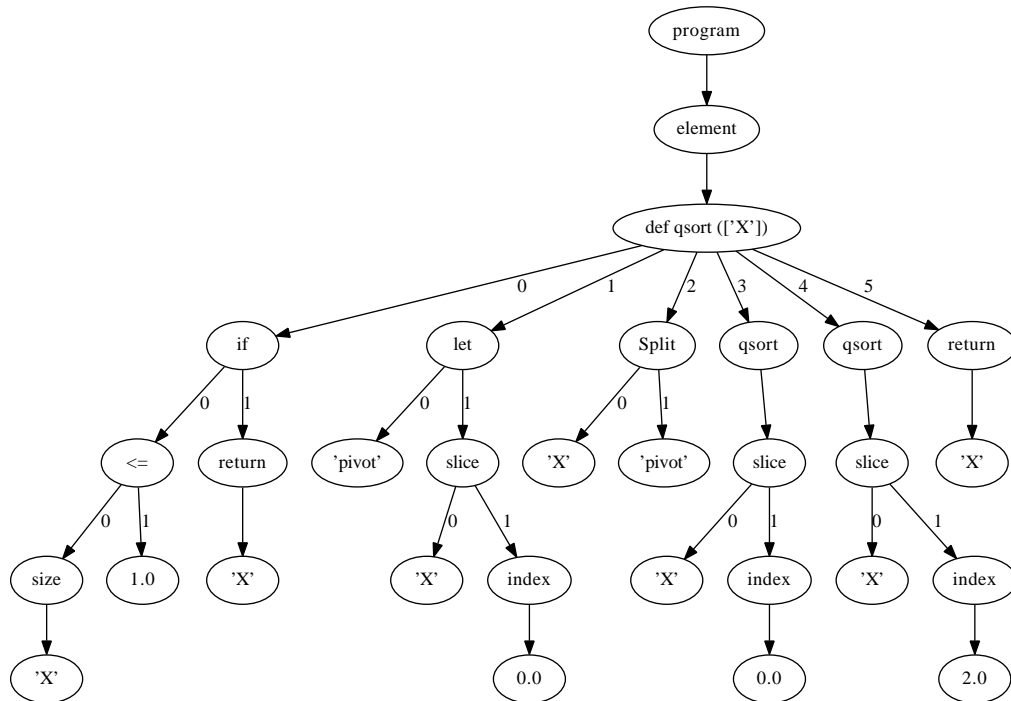


Figure 4.3: Abstract syntax tree for *MOQA* Quicksort

interpreter. In this situation, the interpreter will output an AST for the input program without evaluating it. This mode is helpful when we debug our *MOQA* program for syntax errors or it can be used to help us identify bugs resident in our interpreter. In Figure 4.3, we show an example abstract syntax tree for *MOQA* Quicksort.

It can be seen in Figure 4.3, that the source code contains only one function definition. Thus the program is made by one element node, which is a function definition. In the Quicksort function definition, its formal parameter is *X*. The function consists of six expressions: one base case conditional checking, one `let` pivot binding, one *MOQA Split* function call, followed by two recursive calls and one `return` expression.

We use several examples to show how to translate our grammar rules defined in Section 3.3.5 to real PLY Python codes. Recall the `let` expression grammar rule:

$expr \rightarrow \text{let IDENTIFIER} = expr$

Listing 4.2: Let expression grammar rule in PLY

```
1 # expr -> let Identifier = expr ;
2 def p_expr_let(p):
3     'expr : LET IDENTIFIER ASSIGN expr'
4     idf = moqa_ast.IdentifierNode(p[2])
5     p[0] = moqa_ast.LetExprNode([idf, p[4]])
```

Here we define a function `p_expr_let` for the `let` expression rule (see Listing 4.2). We follow the general naming convention in PLY: a function name is made with a list of words and joined by the separator `_`. Function names starting with `p`, indicates that is a parse function. Next, followed by the left-hand side non-terminal, in our example is the word `expr`. Finally, one or more words represent the right-hand side strings or their meaning. In our example this is the word `let`.

In a parsing function (see e.g. lines 2-5), the first line is always a docstring showing which grammar rule this function is implemented for. In the body of the function, a list-like parameter `p` is used to help construct the abstract syntax tree. `p[0]` corresponds to the value of the left-hand side nonterminal (e.g. `expr`), that is the returned parse tree. `p[1]` is the next symbol in the rule, etc.

For our example, a `let-expression` node consists of two components: one identifier node, one sub-expression node. We construct the `LetExprNode` by first building an `IdentifierNode` with value in `p[2]`, then combining it with whatever structure the sub-tree `p[4]` has, to construct the returned parse tree. Notice that useless terminals are discarded, e.g. `LET`, `ASSIGN`.

As a second example, the `MOQA` binary operation rule, is shown below. The `MoqaOpNode` constructor needs two arguments. The first one is a `MOQA` function name, the second argument is a list of operands for that `MOQA` function.

$$moqaExpr \rightarrow expr \langle \rangle expr \mid expr \rangle \langle expr$$

Listing 4.3: `MOQA` binary operation rule in PLY

```
1 # moqaExpr -> expr op expr
```

```

2 def p_moqa_expr_binop(p):
3     '''
4     moqaExpr : expr PRODUCT expr
5               | expr SPLIT expr
6     '''
7     p[0] = moqa_ast.MoqaOpNode(p[2], [p[1], p[3]])

```

During parsing, simple shift-reduce conflicts might occur [13]. We solved these problems by precedence rules. Here we only scratch the surface of our parser. The details of the implementation are shown in Appendix A Listing A.2.

4.5 Environment Implementation

In our implementation, the environment is implemented as chained environments. Each environment is a tuple: (`parent-pointer`, `{name:value}`). The first part is a reference to the parent environment. The second part is a dictionary that maps each variable name to its value in the current scope. The outer-most environment in our source code is called “global environment”. It has no parent environment, thus `parent-pointer=None`. Upon a function call, a new environment is created. Its parent is the current environment. The created new environment will bind each formal parameter with its corresponding actual parameter value. The function body will be evaluated in the new environment.

For our implementation, there are three specific environments. The exact meaning for their dictionary values are different:

- Type environment: used by the type checker, maps an identifier to its associated type.
- Execution environment: employed by the execution engine, keeps track of object values for a specific identifier.
- Analysis environment: maps an identifier to its random bag. It helps the analysis engine to keep track of random bags, thus enabling automated average-case analysis.

Remark 4.1. In our implementation, all these environments share the same implementation, because of dynamic features of Python.

Setting a name-value binding in our environment is simple. Say we have a environment `env`, then `(env[1])[x]=v` binds identifier `x` with value `v`, `env[1]` references the current scope's name-value dictionary.

Besides the set operation, the environment implementation also supports two other basic operations: “look up a value for a particular variable name”, “update a binding to a particular variable”. Both functions try to first look up the identifier in the current scope. If the function cannot find the target name, it recursively searches for the target name in the parent environment. The details of environment implementation can be found in Appendix A Listing [A.3](#).

4.6 Type Checker

As stated earlier, we implemented our interpreter in a decomposition approach. Each component is resident in a separate module file. This approach is achieved by using Python decorator. Recall in a classical C++ or Java interpreter, the codes for semantics analysis and evaluation will be scattered over different node types of AST classes. All the logic for either evaluating or type checking a tree node will be in the same AST module file. We separate codes for evaluation and type checking into different modules with a decorator-based solution.

Listing 4.4: Python decorator used for code decomposition

```
1 # To add a method to a given class (taken from Matthieu)
2 def addToClass(cls):
3     '''
4     Reference: AST.py v0.2, 2008-2009, Matthieu Amiguet
5     '''
6     def decorator(func):
7         setattr(cls, func.__name__, func)
8         return func
9     return decorator
```

This solution was first proposed by Matthieu at Python Conference 2010 (PyCon 2010) [14]. It is very simple but quite powerful. A simple decorator is suggested and shown in Listing 4.4.

A decorator is based on the fact that functions in Python can be passed around similarly to any other object. This decorator allows us to easily add a new method to a class from outside the class definition, e.g. from another module. Thus it enables us to put different semantic and evaluation logics into different modules.

To show how the decorator works, we present the implementation of two type checking rules in Listing 4.5, which is extracted from the `moqa_typecheck.py` module. The codes implement type checking for the `let` expression and the *LPO builder* expression. The environment in this setting is a type environment.

Remark 4.2. In our AST implementation, both the *LPO builder* expression and the `let-expression` are presented by `LetExprNode`. They differ by their right-hand side value. In a *LPO builder* expression, the right-hand side is a `LPONode`.

Listing 4.5: *MOQA* type checker: Let expression rule

```
1 #           LPONode           #
2 @addToClass(moqa_ast.LPONode)
3 def typecheck(self, env):
4     return Types.DLPO
5
6 #           LetExprNode       #
7 @addToClass(moqa_ast.LetExprNode)
8 def typecheck(self, env):
9     vname = self.children[0]
10    rhs = self.children[1]
11    (env[1])[vname.tok] = rhs.typecheck(env)
12    return Types.Object
```

Notice that there is an extra line before each function definition. All these functions are called “decorated functions”. They all decorated by the `addToClass` decorator. The name inside parentheses is a class name. It specifies which class

the decorated method is attached to. In our case, we add a `typecheck` method to both `moqa_ast.LPONode` class and `moqa_ast.LetExprNode` class.

The code we present here exactly maps the typing rules defined in Section 3.4.2. The remaining typing rules can be implemented in a similar manner. Most of them are simple and straight forward. By recursive descent type checking an AST, we can have a valid and type checked AST. Because our focus is automated average-case analysis, we won't cover them in full details.

4.7 Execution Engine

The execution engine evaluates a *MOQA* program and executes the program directly based on a runtime library (`moqa_runtime.py`). This component implements the execution semantics we defined in Section 3.5.1.

The core of this component implementation lies in the runtime library. This library implements all basic *MOQA* operations and *LPO* data structures. We start by describing the design and implementation of this runtime library.

There are three classes in our *MOQA* runtime library implementation:

- **Node**: A class that represents the most basic component stored in a **NodeSet**. Each **Node** has a label, which is the user-supplied information about this **Node**. It also stores the sets of nodes above and below it.
- **NodeSet**: A class that represents a collection of **Node** objects with an ordering between these nodes. Each **NodeSet** is a partial order. No duplicates are allowed in a **NodeSet**, which follows *MOQA* theory. A **NodeSet** is implemented by an **OrderedSet**.
- **LPO**: A class that represents a labelled partial order, which contains a **NodeSet** that stores all **Nodes** in this **LPO** and a components list that stores all **LPO** components (only works for **DLPO** and **SLPOs** that are the result of `split`, `product`, `top`, `bot`, `PercM`).

Remark 4.3. We do not have separate classes for *DLPO*, *SLPO* or *PLPO* types. They all shares the same implementation class **LPO**. To identify a specific *LPO* type, we have a type attribute in the **LPO** class to tag the type for an **LPO** object.

During the execution of a *MOQA* program, the type attribute for one LPO object might change, e.g. after a `split`, the object should change its type attribute from DLPO to SLPO.

All *MOQA* operations are defined as a function outside any class definition. These operations are attached to a LPO object dynamically based on LPO type attribute, e.g. DLPO object should have `split`, `top`, `bot` and `product` operations.

In the code fragment in Listing 4.6, we show a simplified LPO class to illustrate the idea of dynamic method attachment.

Listing 4.6: Simplified LPO class Code Fragment

```
1 class LPO:
2     def __init__(self):
3         self.nodes = NodeSet()
4         self.type = LPO_Type.Empty
5
6     def toDLPO(self):
7         # Once this LPO changed to DLPO, add DLPO methods to it
8         self.split = types.MethodType(split, self)
9         self.top = types.MethodType(top, self)
10        self.bot = types.MethodType(bot, self)
11        self.product = types.MethodType(product, self)
12
13    def toSLPO(self):
14        # Once this LPO changed to SLPO, add SLPO methods to it
15        self.split=None;del self.split
16        self.top=None;del self.top
17        self.bot=None;del self.bot
18        self.product = types.MethodType(product, self)
19        self.percM = types.MethodType(percM, self)
20
21    def fromLabels(self, labels):
```

```

22     # LPO from a set of labels, resulting a DLPO
23     assert self.type == LPO_Type.Empty
24     self.type = LPO_Type.DLPO
25     self.toDLPO()
26     for label in labels:
27         node = Node(label)
28         self.nodes.add(node)

```

The constructor `__init__` makes an empty `NodeSet()`, and tags the new LPO object as empty. The method `fromLabels` is used to make a *DLPO* from a set of labels. It only applies to an empty LPO object. Once `fromLabels` is invoked, a LPO object is attached to *DLPO* methods by calling `toDLPO`. Another method `toSLPO` is useful when a *LPO* object changes to a *SLPO* type, e.g. after applying `split` to a *DLPO* variable. For implementation details of all basic *MOQA* operations we refer the reader to the interpreter source code, and for the algorithm to [85].

We implement the execution engine by translating *MOQA* language execution semantics to real code, that is to attach the `execute` method to each AST node.

Recall that `LPONode` objects store labels for a LPO builder expression (e.g. 1,2,3). After evaluation, a *DLPO* object is created based on the label set stored in the `self.lpo` attribute. We give the code fragment for this evaluation rule in Listing 4.7.

Listing 4.7: Code for evaluating `LPONode`

```

1 #           LPONode           #
2 @addToClass(moqa_ast.LPONode)
3 def execute(self, env):
4     dlpo = moqa_runtime.LPO()
5     dlpo.fromLabels(self.lpo)
6     return dlpo

```

In Listing 4.8 the `return` expression is implemented by first evaluating target `expr` (see lines 4-5) and then raising a `MOQAReturn` exception (attached with return value, see line 6).

Because `return` is used inside a user defined function, the evaluation for the `FuncUseNode` will catch the exception and stop evaluating the rest of the statements (see lines 25 – 30). The lines 13 – 23 show how to make a new environment with formal parameters bound to actual parameter values for a function call.

Listing 4.8: Code for evaluating `ReturnExprNode` and `FuncUseNode`

```
1 #         ReturnExprNode         #
2 @addToClass(moqa_ast.ReturnExprNode)
3 def execute(self, env):
4     expr = self.children[0]
5     retval = expr.execute(env)
6     raise MOQAReturn(retval)
7
8 #         FuncUseNode         #
9 @addToClass(moqa_ast.FuncUseNode)
10 def execute(self, env):
11     # omit other details...
12     else:
13         fparams = fvalue[1]
14         fbody = fvalue[2]
15         fenv = fvalue[3]
16         if len(fparams) <> len(args):
17             print "ERROR: wrong number arguments to " + fname
18         else:
19             # Make a new environment frame
20             newenv = (fenv, {})
21             for i in range(len(args)):
22                 argval = args[i].execute(env)
23                 (newenv[1])[fparams[i]] = argval
24             # evaluate the body in the new frame
25             try:
26                 for stmt in fbody:
27                     stmt.execute(newenv)
```

```
28         return None
29     except MOQAReturn as r:
30         return r.retval
```

By supplying an `execute` method for each AST node, our execution engine can recursively evaluate the whole program and produce a valid answer. Other execution semantics can be implemented in a similar manner.

4.8 Analysis Engine

The core feature for our interpreter is implemented by the analysis engine. The flowchart for this component is shown in Figure 4.4. The entry point for the analysis engine is a method called `staticAnalysis`. Two helper sub-procedures are invoked inside `staticAnalysis` to automate average-case analysis. The main role for each method is listed below.

- **staticAnalysis**: entry point of analysis engine. This method takes an AST as input and consists of two phases: the pre-process and analysis phase. Pre-process traverses an AST to extract all function definitions and stores them in a dictionary. The stored dictionary maps a function name to the function definition. The analysis phase takes functions in a dictionary one by one, automatically analysing their average-case complexities in terms of the initial *DLPO* size N given by the user. Finally, the gathered average-case times for each function are printed to standard output.
- **doAnalysis**: pre-process method, added to each type of AST node, used to recursively traverse an AST. When traversing an AST, most nodes do nothing but transfer environment (dictionary stores mappings from function name to function definition). One exception is `FuncDefNode`. It stores function information into environment.
- **ACETAnalysis**: the key method in the analysis engine. It is responsible for the analysis phase. It analyses the average number of comparisons needed for a function with initial *DLPO* size N . This method is still implemented

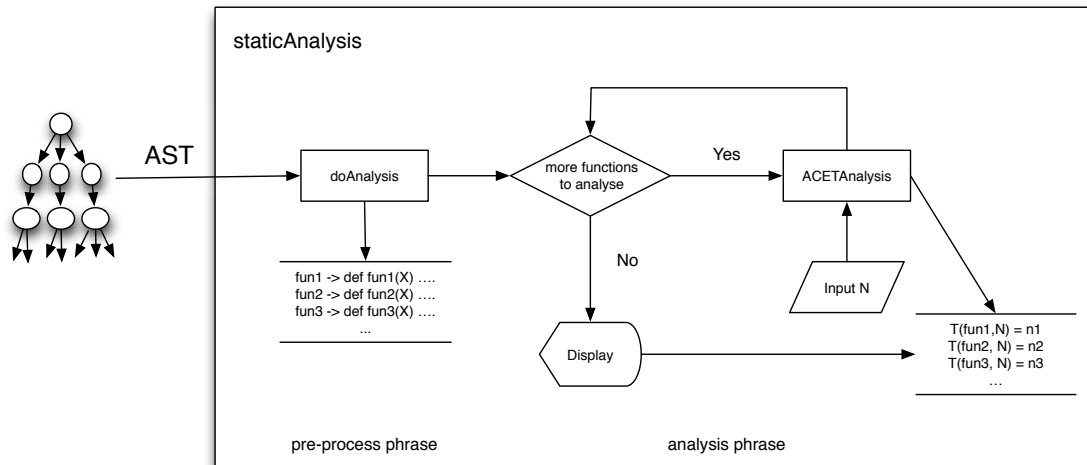


Figure 4.4: Interpreter analysis engine Flowchart

by the interpreter pattern, by attaching analysis methods to each type of AST node. With the help of the *MOQA* timing function and by keeping track of random bags throughout computation, it abstractly interprets the target function and automates average-case analysis.

Let's start with `staticAnalysis`, the entry point function (see Listing 4.9). This function takes three inputs: the abstract syntax tree, the user's input to the interpreter (e.g. size of initial *DLPO* size) and the source code file name.

Listing 4.9: Entry point of analysis engine

```

1 def staticAnalysis(ast, arg, filename):
2     global_env = (None, {})
3     ast.doAnalysis(global_env)
4     ACET = {}
5     for funName in global_env[1].keys():
6         ACET[funName] = ACETAnalysis(global_env, funName, arg)[1]
7     # omit output codes

```

It first makes a `global_env` in order to hold all mappings from the function name to the function definition (line 2). The `doAnalysis` is invoked on AST

object, by recursively walking over the AST, all user defined functions and related information are stored in `global_env`. Next, lines 4 – 6 are the analysis phase. Each function is analysed by the method `ACETAnalysis`. The results are gathered into the dictionary `ACET`. Notice that each function is analysed under `global_env`. It contains all user defined functions. As a result, without building a call graph, `ACETAnalysis` can handle invoking a user defined function inside the function definition. Finally the analyzer produces a report (in the console) for the user. We omit the code here.

In order to implement the pre-process phase, most of the `doAnalysis` methods attached to AST nodes are passing environments and recursively invoke children's `doAnalysis` methods. One exception is when the method is executed over a `FuncDefNode` (see Listing 4.10). In that case, it first extracts necessary information from the `FuncDefNode` object (see lines from 4 – 7), then the function information is stored in the environment and keyed with a function name (line 8).

Listing 4.10: `doAnalysis` Implementation on `FuncDefNode`

```
1 #           FuncDefNode           #
2 @addToClass(moqa_ast.FuncDefNode)
3 def doAnalysis(self, env):
4     fname = self.funcName
5     fparams = self.funcParams
6     fbody = self.children
7     fvalue = ("function", fparams, fbody)
8     env[1][fname] = fvalue
```

4.8.1 Dynamic programming in `ACETAnalysis`

In order to obtain the average timing for a function, we keep calling `ACETAnalysis` by providing all function definitions (`global_env`), target function names (`funName`), and user interpreter parameters (`arg`). But in some situations, e.g. Quicksort, in order to know its average-case time, there are lots of repeated computations.

Example 4.1. If we want to know $\overline{T}_{qsort}(R(\Delta_3))$, recall the result of split operation: $DLPO_{split}(R(\Delta_3)) \mapsto \{(R(\vee_3), 1), (R(S_3), 2), (R(\wedge_3), 1)\}$.

Thus $\overline{T}_{qsort}(R(\Delta_3)) = 2 + \overline{T}_{qsort}(\{(R(\vee_3), 1), (R(S_3), 2), (R(\wedge_3), 1)\})$. In order to know $\overline{T}_{qsort}(R(\vee_3))$, we need to compute the timing for two recursive calls, which sort upper component and lower component: $\overline{T}_{qsort}(R(\Delta_2))$ and $\overline{T}_{qsort}(R(\Delta_0))$.

Similarly, $\overline{T}_{qsort}(R(S_3))$ and $\overline{T}_{qsort}(R(\wedge_3))$ require repeated computations on $\overline{T}_{qsort}(R(\Delta_2))$, $\overline{T}_{qsort}(R(\Delta_1))$ and $\overline{T}_{qsort}(R(\Delta_0))$.

Listing 4.11: Python memorising decorator

```

1 def memo(f):
2     """Decorator that enables dynamic programming.
3     For each call to f(args[1:]) it caches the
4     returned value. When f called again with
5     same args[1:], cached value is returned
6     """
7     cache = {}
8     def wrap(*args):
9         try:
10            return cache[args[1:]]
11        except KeyError:
12            cache[args[1:]] = result = f(*args)
13            return result
14        except TypeError:
15            # some element of args can't be a dict key(mutable)
16            return f(*args)
17    return wrap

```

In our analysis engine, `ACETAnalysis` behaves like a mathematical function \overline{T} . To make our analysis efficient, we introduce a memorising decorator (see Listing 4.11). It decorates the function with a cache property, to help the function remember old computed results. If a repeated computation occurs, the cached value is returned without recomputation (the way dynamic programming be-

haves).

With memorising decorator our `ACETAnalysis` function is defined in Listing 4.12.

Listing 4.12: `ACETAnalysis` implementation

```
1 @memo
2 def ACETAnalysis(global_env, funcName, arg):
3     funcDef = global_env[1][funcName]
4     newenv = (global_env, {})
5     fbody = funcDef[2]
6     LPO_name = funcDef[1][0]
7     if len(funcDef[1]) > 1:
8         opt_arg = funcDef[1][1]
9         newenv[1][opt_arg] = arg[1]
10        rds = CreateDiscreteRandomStructure(arg[0])
11    else:
12        rds = CreateDiscreteRandomStructure(arg)
13    newenv[1][LPO_name] = rds
14    comparation = 0
15    try:
16        for stmt in fbody:
17            val, c = stmt.moqaProcess(newenv)
18            comparation += c
19    except MOQAReturn as e:
20        comparation += e.retval[1]
21        return e.retval[0], comparation
22    return None, comparation
```

The first line decorates `ACETAnalysis` with the memorization feature. Notice that in our memo implementation, the same function call is defined in terms of the same `funcName` and `arg`, because `global_env` is not changed. Line 3 retrieves function information from the environment, based on function name. Line 4 prepares the new environment for the analysis of the function body. The function

body is extracted at line 5. Next, line 6 gets a first formal parameter. In *MOQA* programming this parameter always binds to the *DLPO* object. From lines 7 – 12, depending on the number of user parameters, an initial random structure is created with the user specified size. An optional second parameter is also bound to its formal parameter, if it exists. Next, line 13 binds the initial *DLPO* object to its formal parameter in the analysis environment. Finally, lines 15 – 22 interpret the function body line by line and accumulate their time cost in the variable `comparison`. They also catches `MOQAreturn` in case function execution stops earlier.

Notice that each AST node has an associated function `moqaProcess`. It reflects the analysis semantics we defined in Section 3.5.2. Each call to `moqaProcess` produces a value and a number of comparisons needed to evaluate the node. The side-effects (e.g. update environment) are implemented inside the `moqaProcess` method.

Listing 4.13: Code Fragment: simplified `moqaProcess` method on `FuncUseNode`

```

1 @addToClass(moqa_ast.FuncUseNode)
2 def moqaProcess(self, env):
3     fname = self.funcName
4     args = self.children
5     fvalue = env_lookup(fname, env)
6     fparams = fvalue[1]
7     fbody = fvalue[2]
8     # omit codes...
9     funarg, comparison = args[0].moqaProcess(env)
10    retval = moqa_analyzer_sppo.RandomBag()
11    for i in range(len(funarg.randomstructures)):
12        rds = funarg.randomstructures[i]
13        prob = funarg.probs[i]
14        val, c = ACETAnalysis(env[0], fname, rds.size)
15        retval.add(val, prob)
16        comparison += prob * c
17    return retval, comparison

```

To illustrate how to map those semantics rules to real code, we show the code fragment in Listing 4.13 as an example.

In our implementation, both built-in function call and invoke user defined functions share the same AST node type. To illustrate the general idea, we provide a simplified code. The original code is complex (about 70 lines), because it has to deal with built-in function calls (`print`, `show` etc), and calls user defined functions with two parameters etc. We omit those details.

In `FuncUseNode`, the callee function name (the function that is being invoked) and the argument list are stored in an AST node. First we retrieve these values back at line 3–4. Next, the callee function definition is retrieved from the current environment at line 5. Then, the formal parameters and the function body are extracted. To prepare the function call, `moqaProcess` is recursively invoked on actual arguments. The returned value and number of comparisons are stored (see line 9). Finally, following the semantics we defined, each random structure is timed separately, and combining their associated probabilities, the time cost for a random structure is computed. The final resulting random bag together with the accumulated comparisons are returned to the caller.

Notice that to compute the time cost for a random structure (see line 14), we use `ACETAnalysis` with the size of a random structure as input to avail of a dynamic programming feature. This works in current *MOQA* programming, because all user defined functions are defined to only take a *DLPO* as their input, and giving the size of a *DLPO* is enough to reconstruct the structure.

Before leaving this section, we would like to briefly discuss the design and implementation of the *MOQA* analysis library (`moqa_sppo.py`). Different from the execution library, it implements random bags and *MOQA* basic operations over random bag structures. To derive average-case timing information, instead of executing the target function on all possible test cases, we rely on the *MOQA* random bag structure to abstractly interpret programs and accumulate operation costs based on *MOQA* timing functions. As a result, the average-case timing is computed automatically. The class diagram for this library is shown in Figure 4.5.

There are six classes in the module. Recall the *MOQA* timing functions are defined in terms of SP-orders (series parallel partial orders). The designed implementation is aimed to encapsulate SP-order structures. The *MOQA* operations

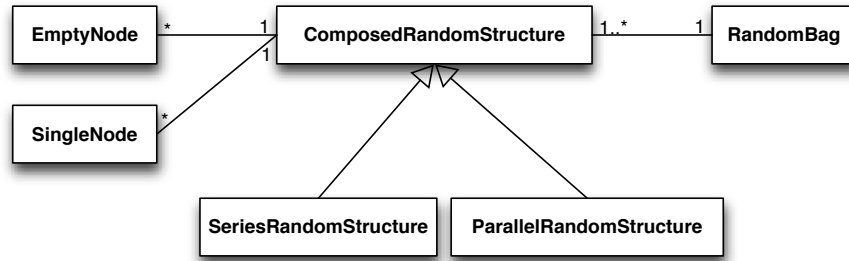


Figure 4.5: *MOQA* analysis library class diagram

are implemented to manipulate the shape of a structure.

The role for each class is listed below:

- **EmptyNode**: place holder for an empty node, it can be used for padding in order to ensure different SP-orders have the same shape, e.g. to ensure the result of a `split` operation has both components above and below.
- **SingleNode**: used as basic building block for a random structure.
- **ComposedRandomStructure**: an abstract class. Stores a list of components, where a component can be an `EmptyNode`, `SingleNode`, `SeriesRandomStructure` or a `ParallelRandomStructure`.
- **SeriesRandomStructure**: subclass of `ComposedRandomStructure`, presents a SP-order built up from a list of components with series composition.
- **ParallelRandomStructure**: subclass of `ComposedRandomStructure`, presents a SP-order built up from a list of components with parallel composition.
- **RandomBag**: presents a multiset of random structures. It is implemented as a list of random structures together with their multiplicities and probabilities.

Remark 4.4. Notice that in our implementation, for efficiency, the size n discrete random structure is not made with n `SingleNode` objects. Instead `ParallelRandomStructure` only has one `SingleNode` component but has a `repeat` property set to n .

In our implementation `CreateDiscreteRandomStructure` is used to make a n elements discrete random structure. The code snippet is shown in Listing 4.14.

Given input argument n , `CreateDiscreteRandomStructure` makes a discrete random structure with n nodes. Lines 2 – 4 deal with cases where $n == 0$ or 1. For $n \geq 2$, line 6 builds a n elements `ParallelRandomStructure`. The first argument `isAtomic=True` indicates this object is made with parallel composition of list of `SingleNode` objects. No complex structure is involved. The last argument `repeated` indicates the number of times a list of components is repeated. In our case a n elements discrete random structure is made by n -repeating `SingleNode`.

Listing 4.14: Create Discrete Random Structure in Analysis mode

```
1 def CreateDiscreteRandomStructure(n):
2     if n == 0: return EmptyNode()
3     if n == 1: return SingleNode()
4     components = [SingleNode()]
5     # Discrete Random Structure
6     DRDS = ParallelRandomStructure(True, components, n)
7     return DRDS
8
9 class ParallelRandomStructure(ComposedRandomStructure):
10    def __init__(self, isAtomic, components, repeated):
11        ComposedRandomStructure.__init__(self, components,
12                                         repeated)
13        self.isAtomic = isAtomic
14        if isAtomic:
15            self.size = repeated
16        else:
17            self.size = repeated *
18                sum(c.size for c in self.components)
19        ...
```

Example 4.2. In Figure 4.6, we give a series SP-order. It is series composed by three components: Y_1, Y_2, Y_3 . To build this object, we use a constructor for

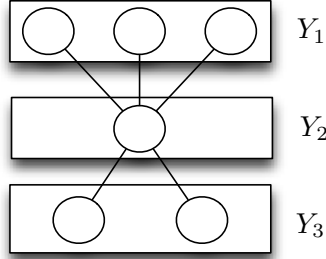


Figure 4.6: Series SPPO creation example

class `SeriesRandomStructure`. It has the same signature as the parallel version. `SeriesRandomStructure(False, [Y1,Y2,Y3],1)`, where $Y_2 = \text{SingleNode}()$, $Y_1 = \text{CreateDiscreteRandomStructure}(3)$, $Y_3 = \text{CreateDiscreteRandomStructure}(2)$. Notice this time `repeated=1` and `isAtomic=False`.

Remark 4.5. The components for a series SP-order are stored from highest to lowest, e.g. the first component in component list is the top component.

With this structure representation, *MOQA* basic operations can be expressed directly without label value comparison, complex push-down etc. The responsibility for most operations is to build SP-order structures with our representation.

For example, to implement the *MOQA Product* operation (see Listing 4.15), say we have two structures `rdsA` and `rdsB`, depending on types for `rdsA` and `rdsB`, we need to create the structure differently. There are several conditions that need to be designed carefully, especially when the operation involves a structure with repeated component. Currently we expand the structure with all nodes, then do the operation (as shown in lines 4 – 8, it calls a method `expand()` implemented by `ComposedRandomStructure`). Here we omit details that deal with special conditions in the *Product* operation, i.e. to product a sorted list with a single element. In such conditions, we can have an economic way to capture the repetition in the structure via an object attribute called `repeated`. We separate these special conditions to make our code run faster and data structures more compact.

Remark 4.6. A repeated structure currently only occurs in discrete partial orders or sorted orders. We would like to investigate this area in the future to explore a

more efficient way to present SP-orders and operations.

Generally, if two series structures product together, we make a new structure with components containing both parts. If we product one parallel and one series structure, we make a new structure by adding the parallel structure as a component to the series structure's component list. Finally, if two parallel structures product together, we make a new series structure with both parallel structures as components.

Listing 4.15: *MOQA* product operation in analysis mode

```
1 def product(rdsA, rdsB):
2     ....
3     components = []
4     # expand repeated components
5     if isinstance(rdsA, ComposedRandomStructure):
6         rdsA.expand()
7     if isinstance(rdsB, ComposedRandomStructure):
8         rdsB.expand()
9     # make new structure
10    if isinstance(rdsB, SeriesRandomStructure):
11        components += rdsB.components
12    else:
13        components.append(rdsB)
14    if isinstance(rdsA, SeriesRandomStructure):
15        components += rdsA.components
16    else:
17        components.append(rdsA)
18
19    randomstructure = SeriesRandomStructure(False, components, 1)
20    return randomstructure, t_product(rdsA, rdsB)
```

In our implementation, we first make a component list for the final structure (see line 3), In this list the order is the same as when you read components from top to bottom. Recall the *Product* definition in Section 2.3.3.2. The first operand

is placed below. Thus, after expanding structures, based on the type of the second operand, we add either itself or its components to the final component list (lines 10-13). The first operand is considered using the same logic.

With this order of consideration, we make sure the components of the second operand are added earlier than the first operand's components, which reflects in the final structure where components of the second operand are placed above all components of the first operand. Finally a new `SeriesRandomStructure` is made with a new component list. Notice that while we build a new structure, each operation timing cost is also derived by *MOQA*'s timing function. For the *Product* operation we implemented this in `t_product`.

The function shown in Listing 4.16 is the *MOQA Product* timing function, implemented by our module. It exactly follows the mathematical definition of the *Product* timing function (see Section 2.3.3.2). There are some helper functions, such as `t_tauDown` and `t_tauUp` etc. These are all implemented as suggested by the theory. We recursively calculate those values based on SP-order structures.

Listing 4.16: *MOQA* product timing function implementation

```
1 def t_product(rdsA, rdsB):
2     sizeA = rdsA.size
3     sizeB = rdsB.size
4     tauDown_A = t_tauDown(rdsA)
5     tauUp_B = t_tauUp(rdsB)
6     aMax = rdsA.getMax()
7     bMin = rdsB.getMin()
8     prob = float(sizeA*sizeB) / (sizeA + sizeB)
9     time = prob * (tauDown_A + tauUp_B) +
10           (prob+1) * (aMax + bMin - 1)
11     return time
```

4.9 Summary

In this chapter we presented the current implementation of the *MOQA* language interpreter. Because we have full control of the internal representation, the code analysis phase is easier compared with the original approach [48]. But, as one would expect, implementing a programming language is not an easy job. This chapter only provided the reader an overview of how the interpreter has been implemented, while it is not possible to demonstrate all technical measures that have been used. Of course, our implementation is never perfect. We provide some discussion and possible future enhancements in Chapter 8.

Chapter 5

MOQA Programming Examples and Evaluation

Contents

5.1	Insertionsort	129
5.2	Quicksort	131
5.3	Quickselect	133
5.4	Mergesort	136
5.5	TreapGen	137
5.6	Treapsort	141
5.7	Heapify	143
5.8	Summary	147

In this chapter, we present several *MOQA* algorithms and their implementation written in the *MOQA* language.

We will focus on sorting, searching and heap creation algorithms. The main focus for this chapter is to show the capability of the current language design and to evaluate the correctness of the automated average-case analysis performed by our interpreter analyzer.

For each algorithm, we first give a brief explanation of the algorithm and how we implement it in the *MOQA* language. Then, following a general mathematical analysis based on *MOQA* theory, we will compare the result from the interpreter analysis mode with the theoretical result to demonstrate correctness of the interpreter analysis mode.

5.1 Insertionsort

Insertionsort is probably the most basic sorting algorithm to implement in *MOQA*. The final sorted list is built by repeatedly inserting a single element into a sorted sublist [26, 57]. In *MOQA* programming the insertion is performed via the *Product* operation. The source code is shown in Listing 5.1, where variable *Z* builds the final sorted list.

Listing 5.1: *MOQA* Insertionsort

```
1 def isort(X)
2   let Z = X[0]
3   for i = 1 to |X| do
4     Z = Z <> X[i]
5   end
6   return Z
7 end
```

The timing for this function is only contributed by the `for` loop. At any step *i*, it computes the product of a sorted list *Z* (length *i*) and a single element *X*[*i*].

Theorem 5.1. *The average running time of MOQA Insertionsort on the size n random list is $\sum_{i=1}^{n-1} \frac{2i}{i+1} + \frac{i}{2}$.*

Proof. It is easy to see from the code that the running time of the algorithm is

$$\overline{T}_{isort}(R(\Delta_n)) = \sum_{i=1}^{n-1} \overline{T}[S_i \otimes \bullet],$$

where S_i presents a linear order of size i , \bullet denotes a single element, i ranges from 1 (single element) to $n - 1$ (last step before insertion).

Recall the timing function defined in Section 2.3.3.2:

$$\overline{T}[A \otimes B] = \frac{|A||B|}{|A| + |B|} (\tau_{down}(A) + \tau_{up}(B)) + \left(\frac{|A||B|}{|A| + |B|} + 1 \right) (|A_{max}| + |B_{min}| - 1).$$

In this context, A is a linear order of size i , and B is a singleton element.

Using the product timing function, we get:

$$\overline{T}[S_i \otimes \bullet] = \frac{i}{i+1} (\tau_{down}(S_i) + \tau_{up}(\bullet)) + \left(\frac{i}{i+1} + 1 \right) (1 + 1 - 1).$$

According to [32, 85], $\tau_{down}(S_i) = \frac{i+1}{2} - \frac{1}{i}$, thus

$$\overline{T}[S_i \otimes \bullet] = \frac{i}{i+1} \left(\frac{i+1}{2} - \frac{1}{i} \right) + \left(\frac{i}{i+1} + 1 \right) = \frac{2i}{i+1} + \frac{i}{2}$$

We conclude that the timing for this algorithm is:

$$\overline{T}_{isort}(R(\Delta_n)) = \sum_{i=1}^{n-1} \frac{2i}{i+1} + \frac{i}{2}$$

□

Next we compare this theoretical result with the result we get from the interpreter analyzer. Table 5.1 shows for different initial list lengths that the results of the interpreter and our formal analysis are exactly same, because the interpreter relies on the same timing function as in the theoretical analysis. It also confirms that our analyzer behaves correctly and produces precise results.

List size	Theoretical result	Interpreter result
32	303.883009609	303.883009609
64	1126.51221819	1126.51221819
128	4309.13370581	4309.13370581
256	16819.7513101	16819.7513101
512	66418.3669669	66418.3669669
1024	263920.981649	263920.981649

Table 5.1: Insertionsort: comparing the theoretical result with the interpreter result

5.2 Quicksort

Quicksort is a classical sorting algorithm and widely used in practice. Traditional Quicksort sorts a list by first selecting a pivot element, then partitioning the list into elements larger and smaller than the pivot. It recursively call the sort routine on the smaller and larger group of elements [26, 57]. In *MOQA* this algorithm can be implemented using the *Split* operation (for this operation’s details, see Section 2.3.3.1, page 28).

Listing 5.2: *MOQA* Quicksort

```

1 def qsort(X)
2   if |X| <= 1 do
3     return X
4   end
5   let pivot = X[0]
6   X >< pivot
7   qsort(X[0])
8   qsort(X[2])
9   return X
10 end

```

In Listing 5.2, we give a sample Quicksort implementation in the *MOQA* language. According to the size of input list *X*, the algorithm first determines

the base case, then invokes the *MOQA Split*, followed by two recursive calls on elements above: $X[0]$ and elements below: $X[2]$ (see Section 3.3.3.4, page 60 for the meaning of the index expressions). Finally the sorted list X is returned.

Theorem 5.2. *The average running time of MOQA Quicksort on the size n random list is:*

$$\bar{T}_{qsort}(R(\Delta_n)) = \begin{cases} 1, & \text{if } n \leq 1 \\ n + \frac{2}{n} \sum_{i=0}^{n-1} \bar{T}_{qsort}(R(\Delta_i)), & \text{if } n > 1 \end{cases}$$

Proof. The base case only has one comparison to carry out a conditional check. For input list sizes greater than 2, according to the result from *MOQA Split* operation (see Section 2.3.3.1, page 28), we derive the following recursion:

$$\bar{T}_{qsort}(R(\Delta_n)) = 1 + n - 1 + \sum_{i=0}^{n-1} \alpha_i (\bar{T}_{qsort}(R(\Delta_i)) + \bar{T}_{qsort}(R(\Delta_{n-i-1})))$$

where $1 + n - 1$ means one conditional check plus $n - 1$ comparisons required by the *Split* operation. Recursion also uses the random bag preservation result where $\alpha_i = \frac{k_i \times |R(P[i-1, n-1])|}{\sum_{i=1}^n k_i \times |R(P[i-1, n-1])|} = \frac{1}{n}$. This is the probability that the corresponding structure happens to occur. We refer the reader to [85] and [87] for further details. Thus we can simplify the equation to $n + \frac{2}{n} \sum_{i=0}^{n-1} \bar{T}_{qsort}(R(\Delta_i))$ as required. \square

Remark 5.1. The original *MOQA* result in [32, 85] did not count base case checking. These results are similar equations with minor technical modifications. Asymptotically the result is identical to our result.

As is clear from Table 5.2, with different initial list lengths, the theoretical results confirm our analyzer result and the theoretical result is identical with the interpreter result.

List size	Theoretical result	Interpreter result
32	182.860682899	182.860682899
64	446.372484148	446.372484148
128	1060.75194989	1060.75194989
256	2465.57997755	2465.57997755
512	5628.74596445	5628.74596445
1024	12663.4767948	12663.4767948

Table 5.2: Quicksort: comparing the theoretical result with the interpreter result

5.3 Quickselect

Quickselect finds a k^{th} smallest/largest element in a list with linear complexity. This algorithm is also based on the *Split* operation. By splitting the list according to a pivot, the algorithm either stops (already found the target element) or recursively finds the target k^{th} ranked element in a proper sublist.

In Listing 5.3 we provide an implementation for this algorithm in the *MOQA* language. Our algorithm finds the k^{th} smallest element in a list. If the input list is empty or a single element, we simply do nothing. Otherwise, we split the list to a three layer structure using the *Split* operation. Based on the number of elements below the pivot, three possible answers are returned.

- return pivot, there are $k - 1$ elements less than the pivot.
- return `qselect(bottom, k)`, there are more than k elements less than the pivot. Recursively find the rank k element in the bottom group.
- return `qselect(upper, k - |bottom| - 1)`, there are less than k elements that are less than the pivot. Recursively find the target element in the upper group with updated rank value.

Listing 5.3: *MOQA* Quickselect

```

1 def qselect(X, k)
2   if |X| <= 1 do
3     return X

```

```

4   end
5   let pivot = X[0]
6   X >< pivot
7   let upper = X[0]
8   let bottom = X[2]
9
10  if |bottom| == k - 1 do
11      return pivot
12  end
13  if |bottom| >= k do
14      return qselect(bottom, k)
15  else
16      return qselect(upper, k - |bottom| - 1)
17  end
18 end

```

Theorem 5.3. *The average running time of MOQA Quickselect to find the element of rank k in a discrete random structure of size n is given by*

$$\bar{T}_{qselect}(R(\Delta_n), k) = n+1 + \frac{1}{n} \left(\sum_{i=k}^{n-1} \bar{T}_{qselect}(R(\Delta_i), k) + \sum_{i=0}^{k-2} \bar{T}_{qselect}(R(\Delta_{n-1-i}), k-i-1) \right)$$

where for $n \leq 1$, $\bar{T}_{qselect}(R(\Delta_n), k) = 1$.

Proof. When $n \leq 1$ the first condition is satisfied, i.e. only one comparison is counted.

If $n > 1$ the rest of the code is executed. The timing involves several branches. After a pass through the first conditional expression, the next *Split* operation takes an extra $n - 1$ comparisons.

Now, let B_1 be the part of the codes starting from the condition `|bottom| == k - 1` to its `end` (lines 10 – 12). Let B_2 be the part of code starting from the next condition to its body (lines 13 – 14). Finally let B_3 be the code from lines 15 – 17.

Next, let p_1 be the probability that condition at line 10 is true, p_2 be the probability that the condition at line 13 is true. Then the timing recursion for this algorithm can be expressed as:

$$\overline{T}_{qselect}(R(\Delta_n), k) = 1 + n - 1 + p_1 \overline{T}_{B_1} + (1 - p_1)p_2 \overline{T}_{B_2} + (1 - p_1)(1 - p_2) \overline{T}_{B_3}$$

It is easy to see that $p_1 = Prob(|bottom| = k - 1) = \frac{1}{n}$, because each structure in the output random bag of a *Split* operation has the same probability (see Section 5.2 for value of α_i). By examining the result of the *Split* operation on n elements (see Section 2.3.3.1), $Prob(|bottom| \geq k) = \frac{n-1-k+1}{n}$. We can evaluate p_2 in a similar manner, as a conditional probability: $p_2 = Prob(|bottom| \geq k | \neg p_1) = \frac{Prob(|bottom| \geq k)}{Prob(\neg p_1)} = \frac{n-1-k+1}{n} \times \frac{n}{n-1} = \frac{n-k}{n-1}$. Thus our timing recursion can be simplified to

$$\overline{T}_{qselect}(R(\Delta_n), k) = n + \frac{1}{n} \overline{T}_{B_1} + \frac{n-k}{n} \overline{T}_{B_2} + \frac{k-1}{n} \overline{T}_{B_3}$$

The probabilities associated to each branch sum up to 1, and they can be computed by summing the probabilities attached to the structures that fall into a specific branch (each has probability $\frac{1}{n}$, count how many structures), this is indeed how our analyzer is implemented.

The time cost for branches, \overline{T}_{B_1} , \overline{T}_{B_2} , \overline{T}_{B_3} , are simply the average cost for the expressions inside each branch.

- $\overline{T}_{B_1} = 1$, only one conditional check
- $\overline{T}_{B_2} = \frac{\sum_{i=k}^{n-1} 1 + \overline{T}_{qselect}(R(\Delta_i), k)}{n-k}$, $n - k$ structures fall into this branch.
- $\overline{T}_{B_3} = \frac{\sum_{i=0}^{k-2} 1 + \overline{T}_{qselect}(R(\Delta_{n-1-i}), k-i-1)}{k-1}$, $k - 1$ structures fall into this branch.

Once we replace \overline{T}_{B_1} , \overline{T}_{B_2} , \overline{T}_{B_3} with their values and tidy up our recursions, we get the desired result:

$$\overline{T}_{qselect}(R(\Delta_n), k) = n + 1 + \frac{1}{n} \left(\sum_{i=k}^{n-1} \overline{T}_{qselect}(R(\Delta_i), k) + \sum_{i=0}^{k-2} \overline{T}_{qselect}(R(\Delta_{n-1-i}), k-i-1) \right)$$

□

Remark 5.2. Our equation is again slightly different from the original *MOQA* formula for Quickselect [32, 85] because we count the extra conditional check. But this analysis reflects how the interpreter works, and it is asymptotically equivalent with our past research result.

As you can see from Table 5.3, with different initial list sizes and a k value, the theoretical results are equal to our analyzer result.

List size	K	Theoretical result	Interpreter result
32	16	96.7586729806	96.7586729806
32	32	63.5	63.5
64	16	185.507164546	185.507164546
64	32	202.473521899	202.473521899
128	16	336.01687493	336.01687493
128	32	382.771738178	382.771738178

Table 5.3: Quickselect: comparing the theoretical result with the interpreter result

5.4 Mergesort

Similar to Quicksort, Mergesort is also a divide and conquer algorithm. By diving a list into two sublists it recursively calls the same method to sort sublists and finally merges pairs of sorted lists together.

Listing 5.4: *MOQA* Mergesort

```
1 def mergesort(X)
2   if |X| == 1 do
3     return X
4   end
5   let n = |X|
6   let first_half = X[:n/2]
7   let second_half = X[n/2:]
8   first_half = mergesort(first_half)
```

```

9   second_half = mergesort(second_half)
10  return Merge(first_half, second_half)
11  end

```

A Mergesort implementation in the \mathcal{MOQA} language is given in Listing 5.4. Notice that it uses the non-standard \mathcal{MOQA} operation `Merge`.

Theorem 5.4. *The average running time of \mathcal{MOQA} Mergesort on the size n random list is:*

$$\begin{aligned} \overline{T}_{mergesort}(R(\Delta_n)) = 1 + \overline{T}_{mergesort}(R(\Delta_{\lfloor \frac{n}{2} \rfloor})) + \overline{T}_{mergesort}(R(\Delta_{n - \lfloor \frac{n}{2} \rfloor + 1})) \\ + \overline{T}_{Merge}(R(\Delta_{\lfloor \frac{n}{2} \rfloor}), R(\Delta_{n - \lfloor \frac{n}{2} \rfloor + 1})) \end{aligned}$$

where $\overline{T}_{mergesort}(R(\Delta_1)) = 1$.

Proof. It is easy to see that the base case only needs one conditional check. For an input list size greater than 1, the timing recursion for this algorithm is the sum of each recursive call plus the time to merge two sorted lists and one conditional check. \square

Remark 5.3. As proven in [73], $\overline{T}_{Merge}(R(\Delta_n), R(\Delta_n)) = \frac{2n^2}{n+1}$. But there is no general solution to express the average time to merge two lists with different lengths (length n with $n + 1$). Instead, we use $\overline{T}_{Merge}(R(\Delta_n), R(\Delta_{n+1})) = \frac{2n(n+1)}{(n+(n+1))/2+1}$. This equation mimics the original merge timing function. The numerator is twice the product of two lists' length, the denominator is the average length of two lists plus one. We justify this equation in Appendix B1

Again, the theoretical results confirm our analyzer result, as shown in Table 5.4.

5.5 TreapGen

\mathcal{MOQA} Treaps are the basis for a sorting algorithm in \mathcal{MOQA} known as Treap-sort, which repeatedly calls `PercM` on the largest label in a treap. We shall look at Treap-sort in more detail in Section 5.6.

List size	Theoretical result	Interpreter result
32	184.495424837	184.495424837
64	432.051455734	432.051455734
128	991.133680698	991.133680698
256	2237.28286527	2237.28286527
512	4985.57351265	4985.57351265
1024	10994.1509239	10994.1509239

Table 5.4: Mergesort: comparing the theoretical result with the interpreter result

Given a list of elements (*DLPO* structure), we can convert this object to a treap with an algorithm called `treapgen`. The max-treap generation algorithm is shown in Listing 5.5. `treapgen` behaves similar to Quicksort. Instead of the *Split* operation, this algorithm depends on the *Top* operation (present by \wedge in the *MOQA* language).

Listing 5.5: *MOQA* Treapgen

```

1 def treapgen(X)
2   if |X| <= 1 do
3     return X
4   end
5   ^X
6   treapgen(X[0])
7   treapgen(X[2])
8   return X
9 end

```

As stated in Section 2.3.3.3, the *Top* operation creates a series partial order. To aid treap creation, the operation also keeps track of the node where the maximum label occurs. In *MOQA* programming, for the resulting object of the *Top* operation, we can select the components by their index (see Section 3.3.3.4). In our implementation, `X[0]` means a *DLPO* with labels occurring before the maximum label, `X[1]` refers to a single element containing the maximum label, `X[2]` is a *DLPO* with all labels occurring after the maximum label.

Here we illustrate this algorithm using an example shown in Figure 5.1. Initially, variable X binds to a *DLPO* with labels $\{3, 1, 5, 2, 4\}$. Once applied, the *Top* operation creates a series structure. We can refer to its components using the indices 0, 1 or 2. Recursively, we make a treap in the left component $X[0]$ and in the right component $X[2]$. The final structure is shown in Figure 5.1.

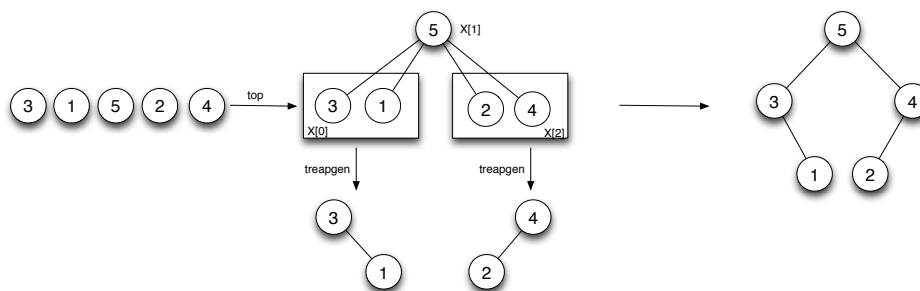


Figure 5.1: *MOQA* Treap creation example

Theorem 5.5. *The Treapgen algorithm produces a random bag $\mathcal{TREAP}(n)$, where each random structure has multiplicity one and forms a collection of treaps over a fixed tree. Treapgen determines a bijection from $R(\Delta_n)$ to $\mathcal{TREAP}(n)$.*

We refer the reader to [32, 85] for details of this proof. We present an example in Figure 5.2 to illustrate this result. In this example, the algorithm has input $R(\Delta_3)$ and as output, a random bag $\mathcal{TREAP}(3)$, where each random structure is separated by different color.

Theorem 5.6. *The average running time of *MOQA Treapgen* on the size n random list is:*

$$\bar{T}_{treapgen}(R(\Delta_n)) = \begin{cases} 1, & \text{if } n \leq 1 \\ n + \frac{2}{n} \sum_{i=0}^{n-1} \bar{T}_{treapgen}(R(\Delta_i)), & \text{if } n > 1 \end{cases}$$

Proof. The code shown in Listing 5.5 is nearly the same as for *MOQA Quicksort*. Actually these algorithms also share the same time complexity.

The base case again has one comparison to perform a conditional check.

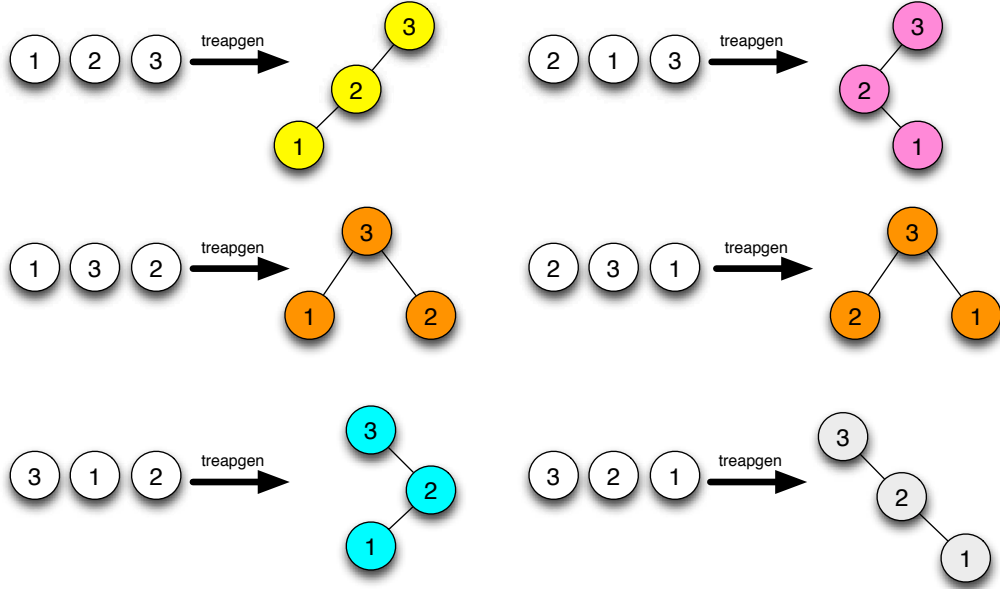


Figure 5.2: Treapgen algorithm maps $R(\Delta_3)$ to random bag $TREAP(3)$

Depending on where the maximum label is located in the original input list, the output of the *Top* operation might result in different structures. E.g, if the maximum label is located at the first place, *Top* will output a series structure with left component $R(\Delta_0)$ and right component $R(\Delta_{n-1})$. Generally we refer to this result by $T[i, j]$, which has i elements in the left component, and j elements in the right component. Notice that, rotated by 90° , this is exactly the result for the *Split* operation, for input list size greater than 2, according to the result for the *MOQA Top* operation, we can derive following recursion:

$$\bar{T}_{treapgen}(R(\Delta_n)) = 1 + n - 1 + \sum_{i=0}^{n-1} \alpha_i (\bar{T}_{treapgen}(R(\Delta_i)) + \bar{T}_{treapgen}(R(\Delta_{n-i-1})))$$

where $1 + n - 1$ means one conditional check plus $n - 1$ comparisons required by the *Top* operation. α_i is the probability that the corresponding structure happens to occur with. It reflects the probability that the maximum label occurs at place $i + 1$, thus it simplifies to $\frac{1}{n}$ for each structure. And we can simplify the equation to $n + \frac{2}{n} \sum_{i=0}^{n-1} \bar{T}_{treapgen}(R(\Delta_i))$ as required. \square

Remark 5.4. Again our equation counts base case checking, while the original *MOQA* result in [32, 85] did not count it. Asymptotically they are the same.

Since this algorithm and complexity is nearly the same as Quicksort, we omit rechecking it in our interpreter. This algorithm is used by Treapsort, the proof from Treapsort will also confirm the result from our interpreter analyzer.

5.6 Treapsort

Treapsort is a new sorting algorithm discovered within *MOQA* research [85]. Based on the *MOQA* treap structure, it repeatedly calls `PercM` on the largest label in a treap. This algorithm shares a similar complexity behaviour with Quicksort. Its worst-case time, like Quicksort, is $O(n^2)$ while its average-case time is $O(n \log n)$ [32, 85].

`PercM` is a standard *MOQA* operation and it acts on the maximum label in a *SLPO* (see Section 2.3.3.4). Basically this operation first percolates the maximum label to a bottom place, then brings it back to the top place with only one element directly below it.

Listing 5.6: *MOQA* Treapsort

```
1 def treapsort(X)
2   X = treapGen(X)
3   let treap = X
4   for i = 1 to |X| do
5     PercM(treap)
6     treap = treap[1]
7   end
8   return X
9 end
```

In Listing 5.6, we provide the Treapsort algorithm implementation in the *MOQA* language. Firstly, at line 2, a treap object is created. Next we make a

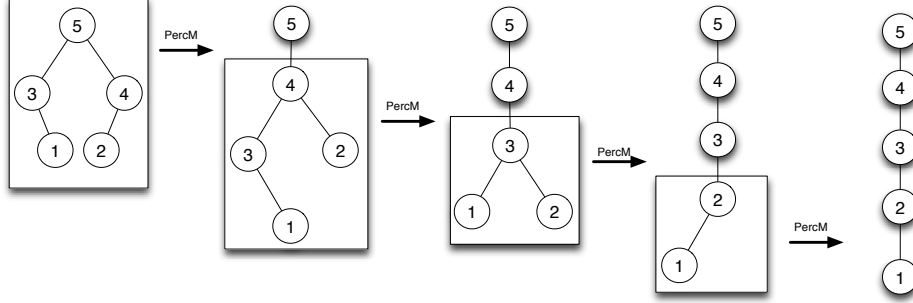


Figure 5.3: TreapSort example on a four element treap.

variable `treap` to track the treap structure where the operation `percM` is applied to. In the `for` loops, each time `percM` is performed on a treap structure, variable `treap` updates its binding to point to the new candidate treap structure (see Section 3.3.3.4 for treap structure indexing, page 60).

To illustrate how the algorithm works, in Figure 5.3, we visualize the execution and highlight the binding to variable `treap` with a block box.

Theorem 5.7. *The average running time of MOQA TreapSort on the size n random list is:*

$$\bar{T}_{treapSort}(R(\Delta_n)) = \bar{T}_{treapGen}(R(\Delta_n)) + \sum_{i=2}^n 2H_i - 2$$

Proof. This algorithm is mainly based on two operations, `PercM` and `treapGen`. The timing recursion for `treapGen` is defined in Section 5.5 and the timing function for `PercM` is $2H_i - 2$ for treap size i . Here we directly use the result from MOQA research on `PercM` operation (See Theorem 2.10). We refer the reader to [32, 85] for the formal proof. At each time, the `treap` variable points to a new treap with size reduced by one. Thus the timing for the `for` loop simply sums the time cost for `PercM` performed on these treaps. And the final timing for the algorithm needs to add the cost for creating the initial treap structure as well. \square

Again, theoretical results confirm our analyzer result, as shown in Table 5.5.

List size	Theoretical result	Interpreter result
32	322.721365798	322.721365798
64	807.07830163	807.07830163
128	1950.50389978	1950.50389978
256	4589.49328844	4589.49328844
512	10574.4919289	10574.4919289
1024	23961.286923	23961.286923

Table 5.5: Treapsort: comparing the theoretical result with the interpreter result

5.7 Heapify

Our last example is a binary heap creation algorithm: Heapify. This algorithm treats the input *DLPO* as a binary tree, where for $n > 1$ the n^{th} node is a child of the node at index $\lfloor \frac{n}{2} \rfloor$, and where each node might have a left child at index $2 * n$ and right child at index $2 * n + 1$ if they are within the index range.

In *MOQA* programming, our language is 0 indexed. To follow this convention in the Heapify algorithm, we design our algorithm by placing a dummy node in the first place.

The algorithm builds a heap by constructing smaller heaps bottom up and the final heap is stored at index 1. At each step, a smaller sub-heap is created by producing a single element with its two sub-children (if they both exist) or by producing a single element with its only child.

Listing 5.7: *MOQA* Code: Heapify

```

1 def heapify(X)
2   for i = (|X| - 1) / 2 downto 0 do
3     X[i] = X[2 * i : 2 * i+2] <> X[i]
4   end
5   return X[1]
6 end

```

We present our implementation for this algorithm in Listing 5.7. Notice that for a node, say at index i , we refer to its two children by $X[2 * i : 2 * i+2]$.

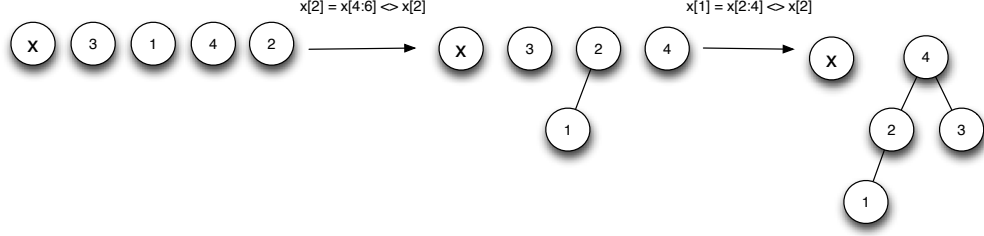


Figure 5.4: Heapify example to create a four node heap.

This is a slice operation. It gets elements from index $2 * i$ to $2 * i + 2$ but not including $2 * i + 2$. Thus it simply obtains two children at $X[2*i]$ and $X[2*i+1]$, but it also is clever enough to ignore $X[2*i+1]$ if it does not exist.

We show a running example in Figure 5.4. It creates a four nodes heap. The first node is a dummy node and is not invoked in the computation. On each iteration, a node value is overridden with a *MOQA Product* result, and gradually the final heap structure is created and stored at index 1.

Remark 5.5. In the following analysis of Heapify, we focus only on heaps of size $2^k - 1$, where k is a natural number such that $k \geq 1$. It presents the number of layers for a heap. This suffices and is standard practice in algorithmic analysis. For a formal justification of this approach we refer the reader to [63].

Lemma 5.8. *Given a complete binary tree \mathcal{B}_k , with k layers, the average number of comparisons made in pushing a label down is given by*

$$\tau_{down}(\mathcal{B}_k) = \frac{k2^{k+1} - 3 \times 2^k + 2}{2^k - 1}$$

Proof. Recall the SP-order definition for a complete binary tree with k layers. We have $\mathcal{B}_k = (\mathcal{B}_{k-1} || \mathcal{B}_{k-1}) \otimes \bullet$. It is easy to see that $|\mathcal{B}_k|_k = 2^k - 1$, $|(\mathcal{B}_k)_{min}| = 2^{k-1}$ and $|(\mathcal{B}_k)_{max}| = 1$. Applying the series and parallel composition laws for τ (see Theorem 2.8) to \mathcal{B}_k , in this case, $A = (\mathcal{B}_{k-1} || \mathcal{B}_{k-1})$ and $B = \bullet$. We get

$$\tau_{down}(A \otimes B) = \frac{|B| \tau_{down}(B) + \kappa_{down}(B) |A_{max}| + |A| (\tau_{down}(A) + A_{max} + \sigma_{down}(B))}{|A| + |B|}$$

$$\begin{aligned}\tau_{down}((\mathcal{B}_{k-1}||\mathcal{B}_{k-1}) \otimes \bullet) &= \frac{2 + (2^k - 2)(\tau_{down}(\mathcal{B}_{k-1}||\mathcal{B}_{k-1}) + 2)}{2^k - 1} \\ &= \frac{2 + (2^k - 2)(\tau_{down}(\mathcal{B}_{k-1}) + 2)}{2^k - 1}\end{aligned}$$

Notice that we simplify $\tau_{down}(\mathcal{B}_{k-1}||\mathcal{B}_{k-1})$ to $\tau_{down}(\mathcal{B}_{k-1})$ because of parallel composition laws for τ . Let $g(k) = (1 - 2^{-k})\tau_{down}(\mathcal{B}_k)$ with base case $g(1) = 0$. We get

$$\begin{aligned}g(k) &= \frac{2^k - 1}{2^k} \times \frac{2 + (2^k - 2)(\tau_{down}(\mathcal{B}_{k-1}) + 2)}{2^k - 1} \\ &= g(k-1) + 2 - 2^{1-k} \\ &= \cancel{g(1)} + 2(k-1) - \sum_{i=1}^{k-1} 2^{-i} = 2k - 2 - (1 - 2^{1-k})\end{aligned}$$

Finally, we divide $1 - 2^{-k}$ on both sides to get the desired result:

$$\tau_{down}(\mathcal{B}_k) = 2k - 2 - (1 - 2^{1-k}) \times \frac{2^k}{2^k - 1} = \frac{k2^{k+1} - 3 \times 2^k + 2}{2^k - 1}$$

□

Remark 5.6. A similar result for $\tau_{up}(\mathcal{B}_k)$ has been shown in [85].

Theorem 5.9. *The average number of comparisons made in building a k -layered complete binary tree \mathcal{B}_k by Product operation $(\mathcal{B}_{k-1}||\mathcal{B}_{k-1}) \otimes \bullet$ is:*

$$\overline{T}[(\mathcal{B}_{k-1}||\mathcal{B}_{k-1}) \otimes \bullet] = \frac{k2^{k+1} - 2^k - 2}{2^k - 1}$$

Proof. Recall that the timing function for the *Product* operation defined in Theorem 2.7. In this case, $A = (\mathcal{B}_{k-1}||\mathcal{B}_{k-1})$ and $B = \bullet$. Clearly, $|\mathcal{B}|_k = 2^k - 1$, $|(\mathcal{B}_k)_{min}| = 2^{k-1}$ and $|(\mathcal{B}_k)_{max}| = 1$. We get

$$\begin{aligned}
\overline{T}[(\mathcal{B}_{k-1}||\mathcal{B}_{k-1}) \otimes \bullet] &= \frac{2^k - 2}{2^k - 1}(\tau_{down}(\mathcal{B}_{k-1}||\mathcal{B}_{k-1}) + \tau_{up}(\bullet)) + \left(\frac{2^k - 2}{2^k - 1} + 1\right)(2 + 1 - 1) \\
&= \frac{2^k - 2}{2^k - 1}(\tau_{down}(\mathcal{B}_{k-1})) + \left(\frac{2^k - 2}{2^k - 1} + 1\right) \times 2
\end{aligned}$$

Applying Lemma 5.8, and replacing $\tau_{down}(\mathcal{B}_{k-1})$ with $\frac{(k-1)2^k - 3 \times 2^{k-1} + 2}{2^{k-1} - 1}$, after simplification, the desired result is achieved. \square

Theorem 5.10. *The average running time of MOQA Heapify on the size n random list is:*

$$\overline{T}_{heapify}(R(\Delta_n)) = 2 \times \overline{T}_{heapify}(R(\Delta_{\frac{n-1}{2}})) + \frac{k2^{k+1} - 2^k - 2}{2^k - 1}$$

where $n = 2^k - 1$ and $\overline{T}_{heapify}(R(\Delta_1)) = 0$.

Proof. As stated earlier in Remark 5.5, we only focus on heaps that have a complete binary tree structure. Thus we can construct our timing function for Heapify using a simple recursion. The times to build a heap with $2^k - 1$ nodes, consist of the total times to build two sub-heaps with size $\frac{n-1}{2}$, and the time to complete the product of two sub-heaps and the top element. With the help of the timing function we derived in Theorem 5.9, we obtain our desired result. \square

Finally, we compare our automated timing results from the interpreter analyzer with the theoretical values. Table 5.6 shows that, the theoretical results are equal to results computed by our analyzer.

List size	Theoretical result	Interpreter result
64	155.689708141	155.689708141
127	324.466030456	324.466030456
255	663.983041304	663.983041304
511	1344.99543682	1344.99543682
1023	2709.00749142	2709.00749142

Table 5.6: Heapify: comparing the theoretical result with the interpreter result

Remark 5.7. Notice that our interpreter analyzer can handle any heap size, because it abstractly simulates the process of heap creation and keeps track of the changes of the data structures and their time costs.

5.8 Summary

In this chapter we evaluated the capabilities of the *MOQA* language and the correctness of the interpreter analyzer has been demonstrated. Several common sorting, searching and data structure creation algorithms have been discussed. Their *MOQA* language implementations have been presented. Comparing the theoretical result with the result obtained from the automated analyzer, the correctness of the analyzer has been checked. There are limitations on the algorithms that can be implemented with the *MOQA* language. With the development of *MOQA* theory, it would be interesting to see other types of algorithms introduced to *MOQA* and implemented in the *MOQA* language. Also it would be a nice project to extend our language with extra features and remove some restrictions if possible. The detailed discussion of our future work is presented in Chapter 8.

Chapter 6

Parallel Extension of *MOQA*

Contents

6.1	Introduction	149
6.2	Related Work	150
6.3	Overview of Multithreaded Program Analysis	151
6.4	<i>MOQA</i> Theory Extension	153
6.5	Experimentation and Evaluating a Practical Example	155
6.5.1	Experiment Result	157
6.6	Summary	160

With this chapter, we start our second main topic of this thesis. We present a new way to analyse multithreaded fork-join programs under the *MOQA* theory, which is published in [39]. We demonstrate that *MOQA* theory can be competently applied to the core principles of parallel algorithms.

We start with a general introduction and a related work discussion in Section 6.1 and Section 6.2. Then, a short overview of multithreaded program analysis is presented in Section 6.3. We expand the *MOQA* modularity theory to a fork-join model (Section 6.4), and show that multithreaded algorithms which satisfy *MOQA* theory allow for easy analysis. Parallel Quicksort serves as an example and is presented in Section 6.5. Finally, a summary is provided in Section 6.6.

6.1 Introduction

With the advance of parallel architecture design, parallel programming has evolved in the past few years. There are various parallel hardware architectures, including multicore processors, large clusters of interconnected machines and recent parallel GPUs. Consequently, different programming languages exist for the architectures mentioned above. The requirement for parallel algorithm analysis also increased.

Our research focuses on performance analysis for fork-join programming on multicore processor machines. The fork-join parallelism is the simplest and most effective design technique for obtaining good parallel performance [62]. The performance analysis of the program in this model is closely related to the program code [21], as the code captures the notation of parallelism, which leaves the potential for it to be statically analysed. The model is suitable and efficient for divide and conquer algorithms, which underpins many kinds of problems, such as sorting (e.g. Quicksort, Mergesort) [26].

MOQA is a new method used to obtain average-case timing information of randomness preserving programs [85]. The data structure in the *MOQA* model is traceable, meaning that the probability of each structure occurring at any step during program execution is predictable. In the later sections, we will show a new approach to statically obtain the work and span for a fork-join program with *MOQA* theory. The work and span obtained will be represented by a recursive

equation in terms of input size. Because our method studies the properties of a parallel algorithm, by exploiting their internal dependence and critical path, our results can be applied to any fork-join programming platform.

There are several fork-join platforms available for programmers with different backgrounds. Cilk [38] is the first published framework that offered a fork-join solution, which provided simple linguistic extensions for multithreading to ANSI C. JCilk [28] is a Java version of Cilk. In our experiment we utilize the Java 7 Fork-Join framework(jsr166) [62], which is included in the new release of Java 7 [74]. By comparing with experimental results, we show that our approach is useful to bound the program performance and speedup in terms of problem size and that it's more accurate than asymptotic analysis.

6.2 Related Work

The performance of a serial application program can be measured by its execution time, while a multithreaded application program employs two more criteria, one named work, the other named span [21]. The tools to analyse serial programs are widely available, while few tools can perform an analysis of multithread programs in terms of their work and span.

Based on a monitored uni-processor execution and simulation, the VPPB (Visualization of Parallel Program Behaviour) [24] system can predict behaviour of a multithreaded Solaris program using any number of processors. It is a great tool to tune parallel programs and to find a critical path, but it is limited on C or C++ to Solaris systems and it does not provide information on work criteria.

Cilkview [47] uses dynamic instrumentation to collect metrics directly on optimized binary codes. It can provide a similar boundary as ours, but it is only available on the Cilk platform and needs a performance-collection run.

HPCToolkit [105] is a profiling tool for multithreaded programming but the metrics it provides are parallel idleness and overhead. These are good metrics for tuning but not for scalability prediction as work and span offer.

Currently none of the tools can provide an equation for work and span and most of them need a performance-collection run like Cilkview. This process will collect necessary information or make bookmarks in the source code. Because

our approach is based on *MOQA*'s traceable structures, the first recorded run is not necessary and our approach can provide a recursive equation as output.

6.3 Overview of Multithreaded Program Analysis

The dag (directed acyclic graph) model is a widely used model when analysing multithreaded programs. The theory behind the dag model was developed by many researchers [22, 23, 31, 42]. A precise and useful tutorial on the dag model can be found in [26].

The dag model of multithreading views the parallel program execution as a set of instructions with dependencies between them [26]. The vertices of the dag represent the sequential instructions with no parallel execution, and this set of serially executed instructions are also called strands in the model. The dependencies between strands (set of instructions) in the model is represented by dag edges. If strand x points to strand y it means that strand y cannot start until strand x has completed, when there is no edge between strand x and strand y we say that they are in parallel.

In Figure 6.1, for example, strand 4 has to wait until strand 1 and strand 2 are completed before it can start, strand 3 is in parallel with strand 4.

The fork-join program can be modelled using a computing dag. To help explain the pseudo-code in our later section, we introduce two new concurrency keywords, **Fork** and **Join**. The use of pseudo fork-join code makes our analysis independent of any particular language or platform. Fork is used before a procedure call, contrasting with ordinary procedure call. The caller thread may continue to execute without needing to wait for the callee procedure to complete. In the corresponding dag, Fork will create two dependency edges emanating from a strand. One goes to the strand containing the first instruction of the Forked procedure, the other goes to the strand containing the first instruction after the forked procedure. A Join statement creates dependency edges from the strand containing the last instruction of each Forked procedure to the strand containing the instruction immediately after Join.

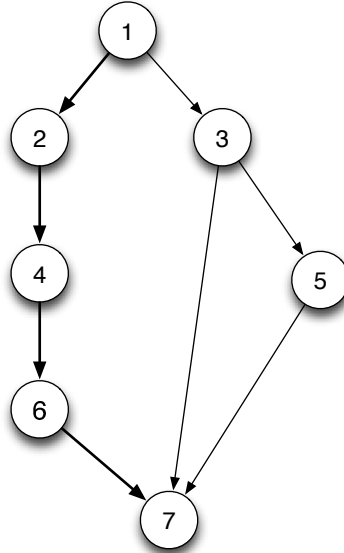


Figure 6.1: A dag representation of a multithreaded program execution. Each vertex is a strand, edges represent strand dependencies.

Two metrics **work** and **span** are used to gauge the theoretical efficiency of a multithreaded algorithm. Work is the total amount of time spent in all strands, and it corresponds to the execution time on 1 processor, it is represented by T_1 . The second measure is span, which is the theoretically fastest time the dag could be executed with an infinite number of processors available. T_∞ is used to represent span, it also means the critical path in the dag.

When considering the time taken with P processors, T_P is used. The **speedup** of P processor is $\frac{T_1}{T_P}$. It can be seen clearly from the definition of T_∞ that we have the following inequality:

$$T_\infty \leq T_P \leq T_1 \quad \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty}$$

These two inequalities will be used later to bound the performance of parallel programs, the work and span will be derived by our new method.

6.4 *MOQA* Theory Extension

In this section we will extend the *MOQA* theory stated in Section 2.3, to apply tracked data structures and distributions to the parallel field and make the link between multithreaded programs and *MOQA* theory.

Lemma 6.1. *If a random bag preserving operation P_1 is executed in parallel with another random bag preserving operation P_2 in the fork-join model, then the result of each operation still forms a random bag. Thus the parallel operation is still random bag preserving.*

Proof. Because P_1 is executed in parallel with P_2 in the fork-join model, the two tasks in the fork-join model won't need to communicate. Hence we can view them as two parallel strands in the computing dag. The parallel run won't affect the restructuring effect of the operation, because the operation itself runs in sequence. Thus random bag preservation holds. \square

When we consider an algorithm's complexity in sequential programming, a recursive equation in terms of problem input size is considered. We bring the same concept into the parallel field and introduce the following notations.

Theorem 6.2. *Average Case Execution Work Time (ACEWT) for an operation P with input random bag $R = \{(R_1, K_1), \dots, (R_n, K_n)\}$ is*

$$\bar{T}_1^P(R) = \sum_{i=1}^n \frac{K_i |R_i|}{|R|} \times \bar{T}_1^P(R_i)$$

Theorem 6.3. *Average Case Execution Span Time (ACEST) for an operation P with input random bag $R = \{(R_1, K_1), \dots, (R_n, K_n)\}$ is*

$$\bar{T}_\infty^P(R) = \sum_{i=1}^n \frac{K_i |R_i|}{|R|} \times \bar{T}_\infty^P(R_i)$$

Remark 6.1. The way to calculate ACEWT, ACEST and ACET is nearly the same, ACEWT and ACEST is extended ACET for parallel algorithm. ACEWT considers the complexity of the operation in terms of one processor, thus it is

identical with ACET. ACEST assumes that infinity number of processors are available, the average-case complexity is still the same pattern, the sum of complexities for each individual structure multiply its probability.

After defining the notations used to represent the average-case complexity of parallel algorithms, we extend Theorem 2.4 to deal with parallel execution where we incorporate our new average-case time notations.

Consider two random bag preserving programs (operations) P and Q such that executing P on random bag R results in random bag R' . The ACEWT and ACEST for the sequential execution P and Q are as follows:

Theorem 6.4.

$$\begin{aligned}\overline{T}_1^{P;Q}(R) &= \overline{T}_1^P(R) + \overline{T}_1^Q(R') \\ \overline{T}_\infty^{P;Q}(R) &= \overline{T}_\infty^P(R) + \overline{T}_\infty^Q(R')\end{aligned}$$

Proof. Since the two operations execute sequentially, one operation will have to wait before starting until the other operation is completed. Thus ACEWT and ACEST are the sum of two operations' complexities. \square

Now consider two operations P and Q in parallel. Again, P is executed on random bag R , and Q on random bag R' .

Theorem 6.5.

$$\begin{aligned}\overline{T}_1^{P\parallel Q}(R, R') &= \overline{T}_1^P(R) + \overline{T}_1^Q(R') \\ \overline{T}_\infty^{P\parallel Q}(R, R') &= \text{Max}\{\overline{T}_\infty^P(R), \overline{T}_\infty^Q(R')\}\end{aligned}$$

Proof. When two operations run in parallel, there is no dependency between them. If two or more processors are available, the average time will depend on the longest operation run. The work time is always the same with the sequential run. \square

The Fork-Join parallel computing model brings two new concurrent keywords: **Fork** and **Join** into algorithm pseudocode. During the analysis of an algorithm, we keep a set of forked procedures. When we encounter a join call we use Theorem 6.5 to calculate the algorithm's ACEWT and ACEST. For a sequential procedure call, we use the *MOQA* linear composition equation in Theorem 6.4.

6.5 Experimentation and Evaluating a Practical Example

In this section we deduce the ACEWT and the ACEST for parallel Quicksort based on our new method. To evaluate our result an experiment is designed and discussed in this section.

The pseudo code of fork-join Quicksort is defined below. The original Quicksort and its *MOQA* programming implementation and ACET analysis can be found in Section 5.2.

Next we will use the structure and distribution captured by *MOQA* combined with our new equation in Section 6.4 to analyse this algorithm.

Algorithm 6.1 Parallel QuickSort algorithm(PQS)

```

if  $L.size \leq 1$  then return n
else
    Split( $L, Up, Down$ );
    Fork ParallelQuickSort( $Up$ );
    Fork ParallelQuickSort( $Down$ );
    Join
end if

```

Remark 6.2. Parallel Quicksort algorithm based on *MOQA Split* operation, the details of this operation is in Section 2.3.3.1 (page 28).

When $n \leq 1$, $\bar{T}_\infty^{PQS}(R(\Delta_n)) = \bar{T}_1^{PQS}(R(\Delta_n)) = 0$

Next we will focus on the other part of the algorithm. There are three statements: *Split*; (*PQS*||*PQS*).

ACEWT of parallel Quicksort:

$$\begin{aligned}
 \bar{T}_1^{PQS}(R(\Delta_n)) &= \bar{T}_1^{split;(PQS||PQS)}(R(\Delta_n)) \\
 &= \bar{T}_1^{split}(R(\Delta_n)) + \bar{T}_1^{PQS||PQS}(Up, Down) \\
 &= \bar{T}_1^{split}(R(\Delta_n)) + \bar{T}_1^{PQS}(Up) + \bar{T}_1^{PQS}(Down)
 \end{aligned}$$

ACEST of parallel Quicksort:

$$\begin{aligned}
\overline{T}_\infty^{PQS}(R(\Delta_n)) &= \overline{T}_\infty^{split;(PQS||PQS)}(R(\Delta_n)) \\
&= \overline{T}_\infty^{split}(R(\Delta_n)) + \overline{T}_\infty^{PQS||PQS}(Up, Down) \\
&= \overline{T}_\infty^{split}(R(\Delta_n)) + \text{Max}[\overline{T}_\infty^{PQS}(Up), \overline{T}_\infty^{PQS}(Down)]
\end{aligned}$$

Notice *Up* and *Down* in the equation are the random bags resulting after the *Split* operation. Via Theorem 2.4 (page 27) and Theorem 2.5 (page 30) we obtain:

$$\begin{aligned}
\overline{T}_1^{PQS}(R(\Delta_n)) &= \overline{T}_1^{split}(R(\Delta_n)) + \\
&\quad \sum_{i=1}^n \alpha_i (\overline{T}_1^{PQS}(R(\Delta_{n-i})) + \overline{T}_1^{PQS}(R(\Delta_{i-1})))
\end{aligned}$$

$$\begin{aligned}
\overline{T}_\infty^{PQS}(R(\Delta_n)) &= \overline{T}_\infty^{split}(R(\Delta_n)) + \\
&\quad \sum_{i=1}^n \alpha_i \text{MAX}[\overline{T}_\infty^{PQS}(R(\Delta_{n-i})), \overline{T}_\infty^{PQS}(R(\Delta_{i-1}))]
\end{aligned}$$

$$\begin{aligned}
\alpha_i &= \frac{k_i \times |R(P[i-1, n-1])|}{\sum_{i=1}^n k_i \times |R(P[i-1, n-1])|} = \frac{1}{n} \\
\overline{T}_1^{split}(R(\Delta_n)) &= \overline{T}_\infty^{split}(R(\Delta_n)) = n - 1
\end{aligned}$$

Where α_i is the probability that the corresponding structure happens to occur. Because the *Split* operation is a sequential operation, its ACEWT and ACEST are the same, the comparisons amount to the list's size minus one.

The resulting data structures of the *Split* operation can be seen in Figure 2.11 (page 31). We replace the random structures $R(\Delta_i)$ by their size i :

$$\begin{aligned}
\overline{T}_\infty^{PQS}(n) &= \overline{T}_\infty^{split}(n) + \\
&\quad \sum_{i=1}^n \alpha_i \text{MAX}[\overline{T}_\infty^{PQS}(n-i), \overline{T}_\infty^{PQS}(i-1)]
\end{aligned}$$

Note that for the first half of the resulting structures, sorting in the upper part will always take a longer time because the size of the upper collection is greater

than the lower part. Symmetrically, in the second half the lower part will always take the longest time. Because of the symmetry of the resulting structures, the sum of the first half of the result will be the same as the sum of the second half. We separate the final recursive equation according to the lengths of the lists, because odd length lists will have a single structure in the middle of the resulting structure list and hence need to be treated separately.

We conclude:

$$\overline{T}_\infty^{PQS}(n) = \begin{cases} \text{For } n \text{ is even:} \\ n - 1 + \frac{2}{n} \sum_{i=1}^{\frac{n}{2}} \overline{T}_\infty^{PQS}(n - i) \\ \text{For } n \text{ is odd:} \\ n - 1 + \frac{2}{n} \sum_{i=1}^{\frac{n-1}{2}} \overline{T}_\infty^{PQS}(n - i) + \frac{1}{n} \overline{T}_\infty^{PQS}\left(\frac{n-1}{2}\right) \end{cases}$$

Base case: $\overline{T}_\infty(1) = 0$

Because ACEWT coincides with sequential algorithmic ACET analysis in *MOQA* we obtain:

$$\overline{T}_1^{PQS}(n) = (n - 1) + \frac{2}{n} \times \sum_{i=0}^{n-1} \overline{T}_1^{PQS}(i)$$

6.5.1 Experiment Result

To justify the correctness of our method, several experiments are designed. The experiments are undertaken using the OpenJDK 64-Bit Server VM (build 1.6.0-b09) on Fedora 9 (X86_64). The machine has two 2.6 GHz quad-core CPUs with 12GB of DRAM. We measure the average-case execution times of sequential and parallel Quicksort executed on a sample of 50,000 randomly generated lists. The experiments are undertaken with various lists sizes and different numbers of cores enabled. We compare the resulting average-case times and speedup with the number of comparisons and the speedup bound calculated from the recurrence

equations obtained by our method.

The generation of the inputs is a key issue. Indeed the quality of the experimental average-case time results is highly dependent on the distribution of the input data. For our study, we used Apache Commons Math [36] which is a library of lightweight, self-contained mathematics and statistics components written in Java.

In order to minimize timer granularity and JVM (Java virtual machine) warm-up artifacts, an initial sample list was running before starting the timers, and all data are medians of three runs.

The recurrences represent the average number of comparisons executed in a program. Of course this in general on its own will not give a sufficiently accurate running time boundary. We use a work-coefficient in conjunction with the number of comparisons to reflect the expected maximum(work) and minimum(span) amount of time to execute the program.

Here, for the purpose of the current experimental evaluation, we calculate the work-coefficient by taking the experimentally obtained ACEWT and divide this by the average number of comparisons over many lists of different sizes. The final value of the coefficient is the average of the values obtained for each sample.

Because of the inequality introduced in Section 6.3:

$$T_\infty \leq T_P \leq T_1$$

We claim that the performance of processor P will be bounded by the algorithm's ACEWT and ACEST. The experiment results in Figure 6.2 verify our prediction. The upper and lower bound in the diagram is generated by our recursive equation, which successfully set a performance boundary of the algorithm no matter how many cores were used.

With the knowledge of the ACEWT and ACEST recursive equation, we can bind the maximum speedup in terms of input size, using the inequality:

$$\frac{T_1}{T_P} \leq \frac{T_1}{T_\infty}$$

One may argue that asymptotic analysis will give a similar result. For asymp-

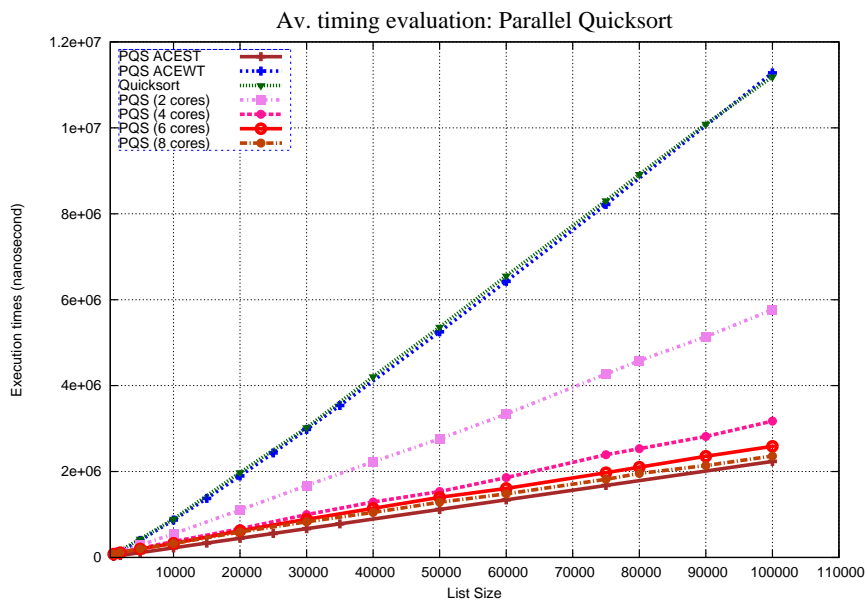


Figure 6.2: A comparison of average time execution between different core amounts and how ACEST and ACEWT bounds the times. Green represents the sequential execution time.

otic analysis, the ACEWT (Work) of parallel Quicksort is $O(n \log n)$ and the ACEST (Span) is $O(n)$, while the boundary speedup is $O(\log n)$ [26]. The result of the experiment is shown in Figure 6.3.

According to the experiment’s results, our method is much more accurate than the asymptotic boundary (with constant one). For the experiments with two and four cores, the speedup is bounded by the number of cores available to work. Meanwhile, for the case of six and eight cores, the bottleneck is the parallelism of the algorithm. The predicted speedup accurately captures this boundary.

The usefulness of the speedup boundary equation is not only limited to this. Because of the equation, we can get the boundary of the speedup in terms of problem size, thus paving the way to resource optimization. One example would be a web server in charge of sorting incoming data. With the help of the speedup prediction we might dynamically change the number of processors allocated, for instance, around speedup number, because more processors allocated won’t increase the speedup noticeably.

One more thing to notice is that on smaller sized lists, parallel sort may be

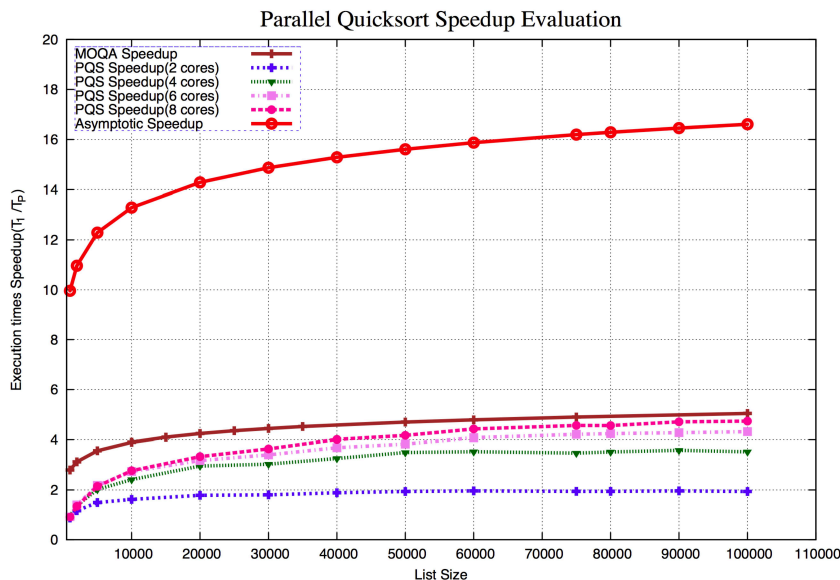


Figure 6.3: A comparison of the obtained speedup differences for core amounts compared to sequential execution and the theoretically obtained values

slower than sequential sort. This is because the overhead of creating one thread may be greater than sorting the list, thus a threshold is needed to switch between sequential sort and parallel sort based on the size of the list to be processed. The threshold is machine dependent, so currently we need to experiment to find a good threshold. In the future we hope to investigate this area and design a new algorithm to dynamically change the threshold on the fly and let the algorithm seek the value. The equation we obtained will be the guide for that algorithm. At the current stage, there are some brief discussions about this problem reported by K. Rea in [79].

6.6 Summary

In this chapter we extended the original *MOQA* theory and presented a new way to analyse a multithreaded fork-join program. We expanded the *MOQA* modularity theory to a fork-join model, and showed that multithreaded algorithms which satisfy *MOQA* theory can be easily analysed. Parallel Quicksort served as an example, where we deduced its recursive equation for work (ACEWT) and

span (ACEST). The experimental results confirmed that the predicted ACEWT and ACEST yield bound the performance of the algorithm and speedup, and proved to be much more accurate than asymptotic analysis.

Chapter 7

Applications of $MOQA$

Contents

7.1	Random Bag Preservation for Heap Algorithms . . .	163
7.1.1	Binary Heap	164
7.1.2	Leonardo Heap	165
7.1.3	Skew Heap	169
7.1.4	Min-Max Heap	172
7.2	Complexity and Entropy based on $MOQA$	176
7.2.1	Average Case Analysis of Treap Insertion	176
7.2.2	Entropy Analysis based on Random Bags	180
7.2.3	Tracking Entropy Changes in Sorting Algorithms . . .	183
7.3	Reversible Computing for $MOQA$	189
7.3.1	Background	189
7.3.2	Reversible Split Operation	190
7.3.3	Reversible Quicksort	196
7.4	Smoothed Analysis for $MOQA$	200
7.5	Summary	204

We investigate a wide range of applications related to *MOQA* research.

In Section 7.1, we look at several types of heaps and explore their creation algorithms. Random bag preservation properties for these algorithms are studied. We identify candidates that might be able to fit into *MOQA* context. Some types of heaps can be made with the current *MOQA* operations, while others require extension on operations and constructs within *MOQA*, but might be implemented in the future.

MOQA was originally designed to support average-case analysis. Because of the usage of random structures and random bag preserving operations, it also paves the way to other application areas.

Section 7.2, focuses on average-case complexity and entropy analysis. The treap insertion provides a new operation to the *MOQA* treap data structure, we investigate the running time of this operation. Then, we define entropy over a random bag and study several sorting algorithms by tracking their entropy changes with random bags.

In Section 7.3, we show how randomness preservation can be used to make *MOQA* partially reversible. A frugal encoding for the reversible *Split* operation is designed and a reversible Quicksort is studied as an example.

Finally, a general introduction to *MOQA* smoothed analysis is presented. We investigate how this metric fits into original *MOQA* and we integrate it with our interpreter analyzer.

7.1 Random Bag Preservation for Heap Algorithms

A heap is a tree based data structure which satisfies the heap property. Based on types of heap property, it is further classified to a max-heap or a min-heap. In our context we focus on max-heaps of which we give the definition below.

Definition 7.1. A heap is generally viewed as a labelled tree. The max-heap (min-heap) property requires the label stored in each node to be greater (less) than or equal to the labels stored in its children. A max-heap (min-heap) is a heap which satisfies the max-heap (min-heap) property.

In this section, we examine several types of heaps. They all satisfy the max-heap property, with one exception, in Section 7.1.4, we discuss a special type of heap called min-max heap, of which each node at an even level in the heap satisfies the min-heap property, while each node at an odd level in the heap satisfies the max-heap property.

There are several operations that are commonly performed on a heap. In the scope of this thesis, we only focus on the operations that build a heap object from a random input list. Different types of heaps have different heap creation algorithms. In the following sections, we focus on their random bag preservation property. We will also encounter examples that are not random bag preserving and hence do not fit in \mathcal{MOQA} theory in their current form. In the original \mathcal{MOQA} research, the binary heap creation algorithm *Heapify* was well studied. We present it below as our warm up example.

7.1.1 Binary Heap

A binary heap is a binary tree that satisfies two extra properties. One is called the shape property. The binary tree must be a left-complete binary tree, that is, each level of the tree is completely filled, except possibly the bottom level. At the bottom level, it is filled from left to right. The second property is the max-heap property (we focus on max-heaps).

As discussed in [85], the binary heap creation algorithm *Heapify* is random bag preserving.

We recall the *Heapify* procedure which uses *Push-Down* (see Section 2.3.3.2) to create a heap from a given list [26]. In Algorithm 7.1, we give the pseudocode for the *Heapify* procedure in \mathcal{MOQA} , using the *Product* operation \otimes . We refer the reader to Section 5.7 for a corresponding *Heapify* implementation in the \mathcal{MOQA} programming language.

Algorithm 7.1 \mathcal{MOQA} Binary Heap Heapify

```

input  $X : \Delta$ 
for  $j \leftarrow \lfloor \frac{|X|}{2} \rfloor$  downto 1 do
     $X[j] \leftarrow (X[2j], X[2j+1]) \otimes X[j]$ 
end for

```

Remark 7.1. For a labelled partial order stored in X , which presents a binary tree, the element with greatest index which has children is the element $X[\lfloor \frac{|X|}{2} \rfloor]$.

Theorem 7.1. *The \mathcal{MOQA} binary heap heapify procedure is random bag preserving.*

Proof. This is a direct result from repeated application of \mathcal{MOQA} 's basic operations. At each iteration, the \mathcal{MOQA} *Product* operation is applied on a random element with its two subtrees. Each subtree is an isolated subset of the input random structure (see Definition 2.10 for isolated subset, page 26). Because there is no relation between the two subtrees, and the top element is greater than rest of the elements, the \mathcal{MOQA} *Product* is guaranteed to operate on random structures. As proven in [85], the *Product* operation is random bag preserving. Hence the resulting structure will be a random structure and the algorithm is random bag preserving. \square

7.1.2 Leonardo Heap

As shown earlier, not all algorithms are random bag preserving (see Example 2.7, page 25). In the following, we introduce a new heap data structure, and show that its creation algorithm is not random bag preserving.

The Leonardo Heap is a special heap that is invented for an interesting sorting algorithm called Smoothsort. This sorting algorithm was originally created by the legendary Edsger Dijkstra [30]. Keith Schwarz provided a detailed explanation of this algorithm at the Stanford ACM Tech Talk [93].

Smoothsort is a comparison based sorting algorithm. Thus its performance is bounded by $O(n \log n)$ [26, 57]. But with clever design, this sorting algorithm provides great memory and runtime guarantees. Its worst-case and average-case performance are both asymptotically optimal, i.e. $O(n \log n)$. Even better, this sorting algorithm is an adaptive sort, it takes time closer to $O(n)$ if the input list is already sorted to some degree. This sorting algorithm can be viewed as a variant of heap sort with a special heap called the Leonardo heap. We focus on the Leonardo heap creation algorithm and examine if it is random bag preserving. We refer the reader to [30, 93] for details about Smoothsort.

The Leonardo heap is based on a number sequence called the Leonardo sequence, a close cousin of the well-known Fibonacci sequence ¹.

Definition 7.2. The Leonardo numbers are a sequence of numbers (denoted $L(0), L(1), L(2), \dots$) given by the following recursive equation:

$$L(n) = \begin{cases} 1 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ 1 + L(n - 1) + L(n - 2), & \text{if } n > 1. \end{cases}$$

Example 7.1. The first few Leonardo numbers are 1, 1, 3, 5, 9, 15, and 25.

Definition 7.3. A Leonardo tree is defined recursively. We use Lt_k to denote an order k Leonardo tree:

$$Lt_k = \begin{cases} \text{a singleton node} & \text{if } k = 0; \\ \text{a singleton node} & \text{if } k = 1; \\ \text{a node with two children, } Lt_{k-1} \text{ and } Lt_{k-2} \text{ (in that order),} & \text{if } k > 1. \end{cases}$$

Remark 7.2. It can be shown with a simple inductive proof that the number of nodes in the tree Lt_k is $L(k)$.

Example 7.2. In Figure 7.1, we show Leonardo trees Lt_0, Lt_1, Lt_2, Lt_3 .

Definition 7.4. A Leonardo heap is an ordered collection of Leonardo trees such that:

- The order of the trees is strictly decreasing, i.e. no two trees have the same order.
- A n elements Leonardo heap is made out of $O(\log n)$ Leonardo trees, and there are at most two trees with consecutive order.
- Each tree obeys the max-heap property (see Definition 7.1 for max-heap property).

¹There is another type of heap, called the Fibonacci heap based on Fibonacci sequence, but we won't cover it in this thesis

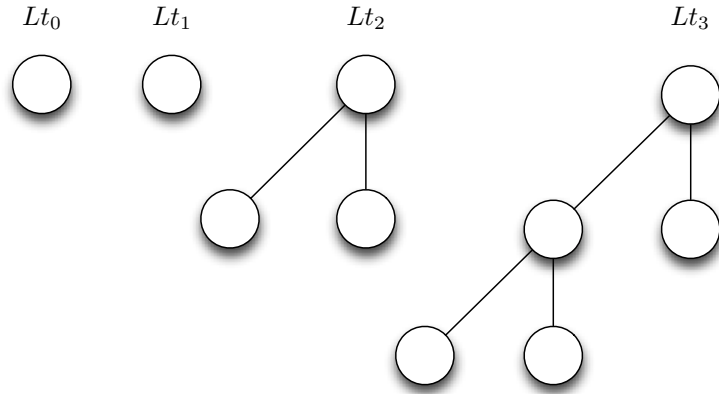


Figure 7.1: Leonardo tree examples: Lt_0, Lt_1, Lt_2, Lt_3

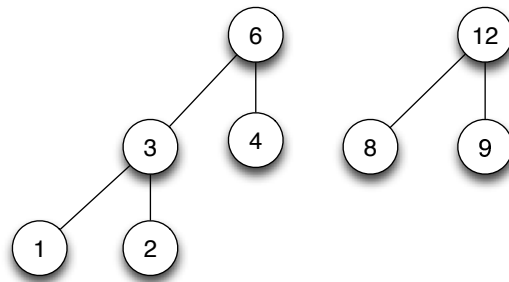


Figure 7.2: Eight nodes Leonardo Heap example

- The roots of the trees are in ascending order from left to right.

Example 7.3. In Figure 7.2, we show an example Leonardo heap with eight nodes.

In order to convert a list of elements to a Leonardo heap, we insert a new element into a partially built heap repeatedly. The whole process is started with a basic singleton node Leonardo heap that contains only the first element. When we insert a new element to a Leonardo heap, there are three properties we need to preserve:

- Ensure that the resulting Leonardo heap has the correct shape, e.g. trees stored in descending order.
- Ensure that the roots of Leonardo trees are sorted in ascending order from left to right.

-
- Ensure that each Leonardo tree obeys the max-heap property.

We describe the steps to insert a new element into a Leonardo heap in Algorithm 7.2.

Algorithm 7.2 Leonardo Heap Insertion

```

if last two trees have adjacent order then
    merge them into one new tree with inserted element.
    Push-Down on new tree.
else
    if last tree is not order 1 then
        Add a tree of order 1 with new element.
    else
        Add a tree of order 0 with new element.
    end if
    Shift new element to a proper heap, ensure that the tops of the heaps are
    sorted in ascending order from left to right.
    Push-Down on heap with new element.
end if

```

Example 7.4. To demonstrate how to convert a list of elements to a Leonardo heap, we present an example in Figure 7.3. It illustrates how to convert a Leonardo heap from input list $L = \{3, 1, 4, 2\}$. The creation steps repeatedly apply the Leonardo heap insertion algorithm. At the start, we have an empty heap, a single Lt_1 tree with element 3 is made. Next, insert element 1. A new tree Lt_0 is created with label 1, then rearranging top labels, 3 is swapped with 1. When 4 is inserted, the first condition in the insertion algorithm is satisfied, and a new tree Lt_2 combines both smaller trees. The labels are rearranged to ensure max-heap property. Repeat the same insertion step. We insert the last element 2 and obtain the Leonardo heap as shown in the figure.

With this necessary background, we can start to look at the randomness property for this heap creation algorithm. In this example, we can view the resulting heap as a parallel partial order, while each parallel component is a Leonardo tree. With a random list input, we hope that the resulting heaps form a random bag. Unfortunately, as we show in the following, this procedure is not random bag preserving.

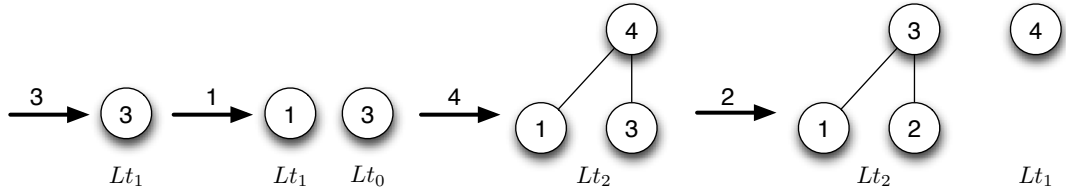


Figure 7.3: Leonardo Heap creation example with input List $L = [3, 1, 4, 2]$

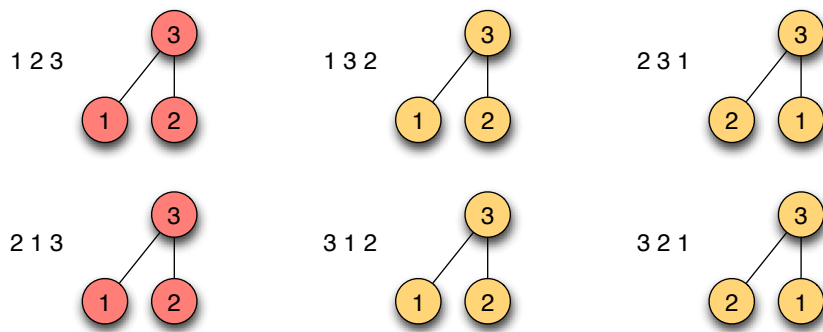


Figure 7.4: Leonardo Heap creation: random preserving contour example

Theorem 7.2. *The Leonardo heap creation algorithm is not random bag preserving.*

Proof. By a simple counter example, we show this result. For a three element random input list, say with labels $\{1, 2, 3\}$, the resulting Leonardo heaps always forms a \wedge shape partial order where of course 3 is always at the top. To make the resulting heap a valid random bag, we need bottom labels to have the same chance of being in order 1, 2 or 2, 1. Since before merging two trees, the insertion always makes the top elements of the trees in sorted order, we have more chance of resulting in order 1, 2. As shown in Figure 7.4, the last two columns form two \wedge shape random structures, but the left resulting heap in the first column is not a random structure. Thus this algorithm won't form a random bag output. \square

7.1.3 Skew Heap

The skew heap data structure is a self-adjusting heap. During each access or update operation, the data structure is adjusted in a simple, uniform way. Like a

binary heap, a skew heap is a heap-ordered binary tree. It was proposed by Sleator and Tarjan [99]. In comparison with binary heaps, skew heaps are advantageous because of their quicker merge operation. A functional implementation is also provided in [72].

Definition 7.5. A skew heap can be defined recursively as:

- A single element is a skew heap.
- The result of skew merging two skew heaps is also a skew heap.

In the following, we only describe how to insert one element e into a skew heap SH . The general merge operation of two skew heaps can be found in [99]. We present the pseudo-code in Algorithm 7.3. Notice that $root()$ gets a root element of a skew heap, while $left()$ and $right()$ get its left and right subtree respectively, and $makeHeap()$ constructs a new skew heap with its first argument as root element, and the second and third argument as left and right subtree.

Algorithm 7.3 Skew Heap Merge

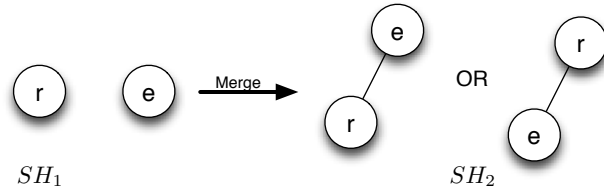
```

Merge( $SH, e$ )
if  $SH = \emptyset$  then return  $e$ 
else
  if  $e > root(SH)$  then
     $makeHeap(e, SH, \emptyset)$ 
  else
     $makeHeap(root(SH), Merge(right(SH), e), left(SH))$ 
  end if
end if

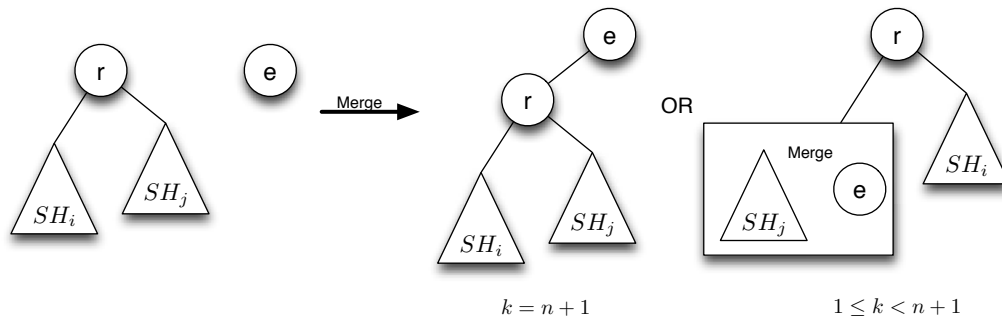
```

Theorem 7.3. *The Skew Heap Merge operation is random bag preserving.*

Proof. In the following we use SH_i to represent a skew heap of size i , and e as a new random number. We prove it by induction. When $i = 1$, $merge(SH_1, e)$ produces two possible skew heaps, as shown below. They have equal chance to be formed and each heap itself is a random structure. Thus, for the base case, this operation is random bag preserving.



Now, assume our inductive hypothesis: $merge(SH_k, e)$ produce a random bag when $k < n$. Consider $merge(SH_n, e)$. In the following figure, we show merge executed on a skew heap random structure.



A n element skew heap random structure must have a top element, with i elements in its left subtree (still a skew heap) and j elements in its right subtree (still a skew heap), where $i + j = n - 1$. When merging a random element say e , this new element will have a rank in the overall labels which we call k .

If $k = n + 1$, the new element is the biggest element. The resulting skew heap is always the first case in the figure above. That is, adding a top element to a random structure. It must result in a new random structure.

When $1 \leq k < n + 1$, the resulting skew heap is shown in the second case. The right subtree SH_i must be a random structure, because it is an isolated subset of the original random structure. In the left subtree, we merge SH_j with a new element e . Because of the inductive hypothesis ($j < n$), the left subtree also results in a random structure. Combining two random structures in parallel, then adding a maximum element, we again have a random structure. Thus in both cases, this operation transforms each random structure to a new random structure, hence it is random bag preserving. \square

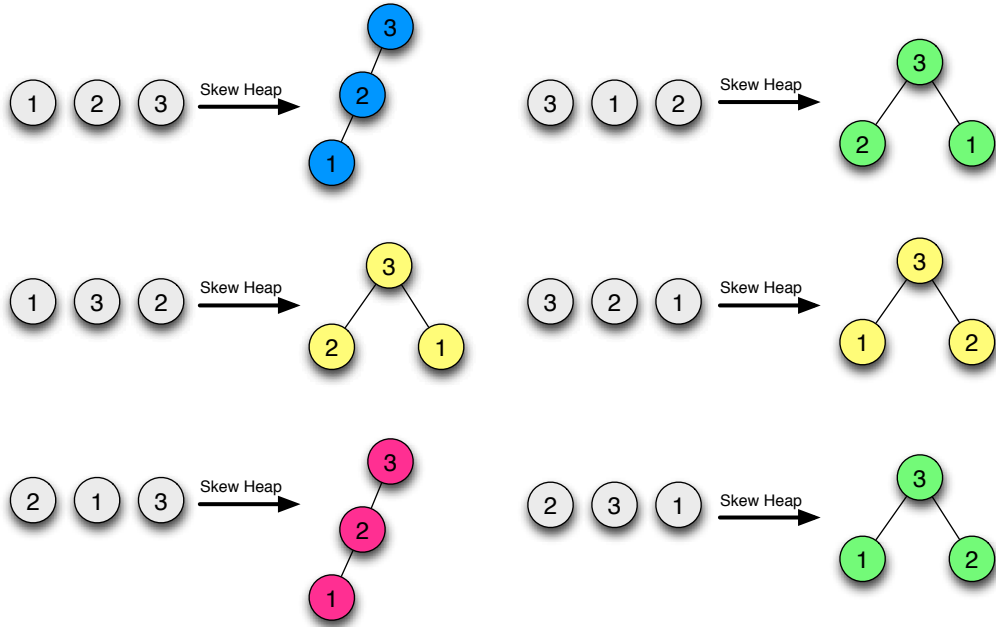


Figure 7.5: Skew Heap creation example: length three random list

Remark 7.3. In our proof, both SH_i and SH_j are isolated subsets of the original skew heap SH_n (see Definition 2.10, page 2.10). This is because subtrees are always isolated.

Corollary 7.4. *Converting a random list to a skew heap is random bag preserving.*

Proof. Note that constructing a skew heap from a random list is simply a repeated application of skew heap merge operations. As shown in Theorem 7.3, the skew heap merge operation is random bag preserving. The input random list is a random structure, thus this construction is random bag preserving. \square

Example 7.5. In Figure 7.5, we convert a length three random list to skew heaps, where each resulting random structure is coloured differently.

7.1.4 Min-Max Heap

A Min-Max heap is a modified version of a binary heap [16]. In this data structure the maximum and minimum label can be retrieved in constant time.

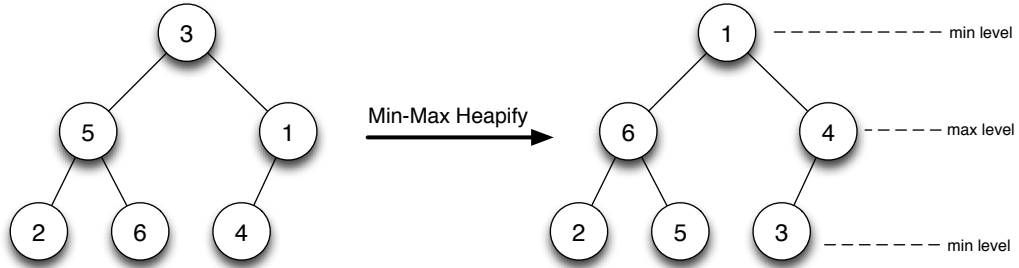


Figure 7.6: Min-Max heap built from input list $[3, 5, 1, 2, 6, 4]$

Like a binary heap, a Min-Max heap is represented as a complete binary tree. A tree is said to be Min-Max ordered if every element on even (odd) levels is less (greater) than all of its descendants, where the root is at level zero.

Example 7.6. Figure 7.6 illustrates a Min-Max heap constructed from an input list $[3, 5, 1, 2, 6, 4]$.

The detailed construction algorithm can be found in [16]. Before we continue, we note the abuse of notation in this section. Formally, Hasse diagrams are upwardly orientated partially order sets. However, the dual of a partially ordered set is a partially ordered set with the converse relation. In keeping with the Min-Max heap notation in [16] we will refer to a dual-Hasse diagram as a Hasse diagram in this section. Following from this, we refer to the *Dual-Product* operation as the *Product* operation in this section. The *Dual-Product* performs like a normal *Product* but keeps the min-heap property, and is denoted by notation \otimes^D .

We recall the *Heapify* procedure which uses Push-Down to create a binary heap from a given list. In Algorithm 7.1, we outline how to implement the *Heapify* procedure in *MOQA* using the *MOQA Dual-Product*, which is published in [40].

We adjust the *MOQA* binary *Heapify* algorithm to allow for the Min-Max property of the Min-Max heap. The *MOQA Min-Max heapify* procedure outlined in Algorithm 7.4 creates a partial order implicit within the normal Min-Max heap construction.

Algorithm 7.4 *MOQA Min-Max heapify*

```

input  $X : \Delta$ 
for  $j \leftarrow \lfloor \frac{|X|}{2} \rfloor$  downto 1 do
  if  $\lfloor \log_2(j) \rfloor \% 2 == 0$  then
     $X[j] \leftarrow (X[2j], X[2j+1]) \otimes^D X[j]$ 
  else
     $X[j] \leftarrow X[j] \otimes^D (X[2j], X[2j+1])$ 
  end if
end for

```

In *Min-Max heapify*, based on whether the product operation occurs at even level or odd level, the algorithm places the two subtrees either above or below the other singleton element. With this alternation, it tracks the partial order implicitly within the normal Min-Max heap.

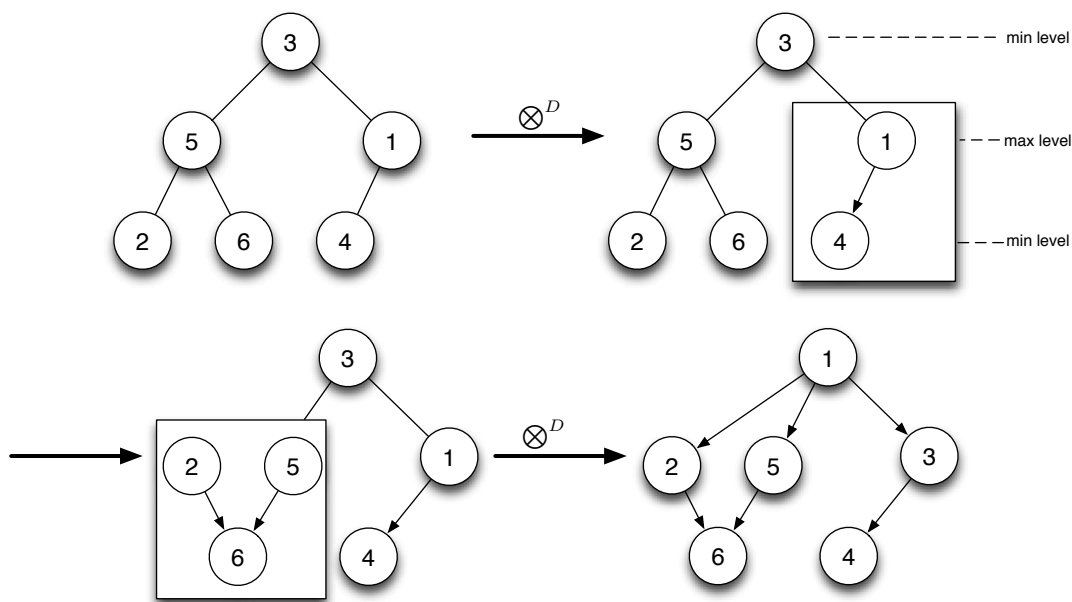


Figure 7.7: *MOQA Min-Max heapify* on input list $[3,5,1,2,6,4]$

Theorem 7.5. *The MOQA Min-Max heapify partial order construction is random bag preserving.*

Proof. The input for the algorithm is a trivial random structure. As the *MOQA* product is recursively applied to isolated subsets (see Definition 2.10, page 26), the algorithm is random bag preserving. \square

Example 7.7. Figure 7.7 illustrates *MOQA Min-Max heapify* on an input list [3, 5, 1, 2, 6, 4]. As can be seen, the partial order created is the partial order implicit within the normal Min-Max heap in Figure 7.6.

The *Min-Max heapify* could produce interesting partial orders, which we did not have before. As shown in Algorithm 7.4, the code for *Min-Max heapify* stays quite close to the original *Heapify* code (see Algorithm 7.1). The extension by a conditional expression may seem to allow one to derive its timing function easily, but in fact this is not an easy job. For the *Heapify* algorithm, the generated structure has a nice recursive definition, but for *Min-Max heapify*, currently we do not have a concise representation to capture the final structure. It makes the time analysis quite hard. We illustrate this by showing one structure produced by this algorithm in Figure 7.8. The full study of this problem is out of the scope of this thesis. It would be an interesting project for future investigation.

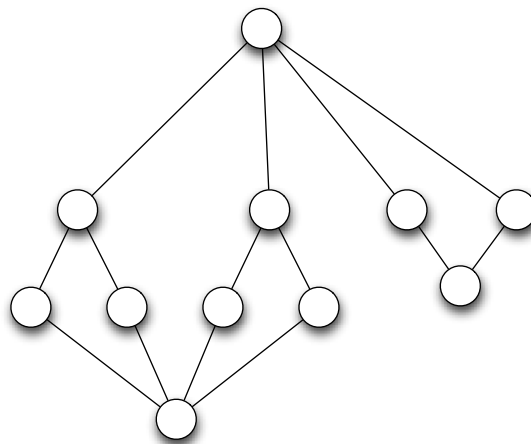


Figure 7.8: One possible partial order generated by *Min-Max heapify*

In conclusion, we note that the Skew heap has the ability to merge more quickly than the binary heap [99], and that the Min-Max heap allows us to find both the smallest and the largest element in constant time. With a suitable generalization, the Min-Max heap can also support similar order-statistics operations

efficiently (e.g., FindMedian or DeleteMedian) [16]. As such the operations associated with constructing these structures are useful candidates to incorporate in our *MOQA* language. We found these two data structures fit the *MOQA* context well as their associated operations are random bag preserving. As such, they are candidates for investigation in our future research, in particular in connection with determining their exact average-case time, e.g. via typical recursive formulas over series-parallel orders.

7.2 Complexity and Entropy based on *MOQA*

In this section, we present a new operation for the *MOQA* treap data structure, and its average running time is analysed. In the second part of this section, we investigate entropy and redefine it over a random bag. Because *MOQA* can track random bags during the execution of an algorithm, it is possible to keep track of data entropy as well.

7.2.1 Average Case Analysis of Treap Insertion

Consider inserting a random value into a n nodes random treap $TREAP(n)$. It will result in a $n + 1$ nodes random treap $TREAP(n + 1)$. The *LPOs* in $TREAP(n)$ and $TREAP(n + 1)$ are $n!$ and $(n + 1)!$ respectively (see Section 5.5, page 5.5 for an introduction on *MOQA* treaps).

We design a new algorithm for inserting a new random element into a *MOQA* treap. The pseudo-code is shown in Algorithm 7.5¹.

Algorithm 7.5 *MOQA* Treap Insertion

```

insert(b, F)
if root(F) < b then
    makeTreap(b, F, null)
else
    F = right(F)
    insert(b, F)
end if

```

¹This work is based on the discussion with Dr. P. Chebolu

Remark 7.4. **makeTreap**($b, F, null$) builds a \mathcal{MOQA} treap with root b and left child F , right child $null$. And $\text{root}(null) = -\infty$

The Algorithm 7.5 recursively inserts a new element b into a \mathcal{MOQA} treap F . We illustrate this algorithm in Figure 7.9. Notice in the figure, the value presents the rank of the LPO label. For example, 1 2 3 means a 3 elements input list with the biggest value last, the smallest value first. From second column to the last column, it shows the insertion of a new element with overall rank 1 to 4, e.g. the first example in the first row, second column. The input sequence is updated from 1, 2, 3 to 2, 3, 4, 1 since the input element has rank 1. It causes the other three elements all to increase their ranks by 1.

Theorem 7.6. *The average running time of \mathcal{MOQA} treap insertion on a n elements treap $\mathcal{TREAP}(n)$ is:*

$$\bar{T}_{Insert}(\mathcal{TREAP}(n)) = \begin{cases} 1 + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n+1} & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$$

Proof.

$$\bar{T}_{Insert}(\mathcal{TREAP}(n)) = \frac{C_n^{Insert}}{(n+1)!}$$

where C_n^{Insert} is the total number of comparisons and $(n+1)!$ is the number of new LPO s.

$$\bar{T}_{Insert}(\mathcal{TREAP}(n)) = \frac{(n+1)! \times 1 + \sum_{i=1}^{n-1} (k_i \times \bar{T}_{Insert}(\mathcal{TREAP}(i)))}{(n+1)!}$$

The first $(n+1)! \times 1$ in the denominator represents the comparisons carried out to compare the root of the treap with the new value. Since we have $(n+1)!$ new LPO s, the number of the comparisons in this part is $(n+1)! \times 1$. If the new value inserted is the biggest, the insertion process will stop. Thus the number of comparisons for that case is 1. For the other cases, when there is more than one node in the right part of the treap and the new value is not the biggest, more comparisons need to be counted. We use $k_i \times \bar{T}_{Insert}(\mathcal{TREAP}(i))$ to calculate

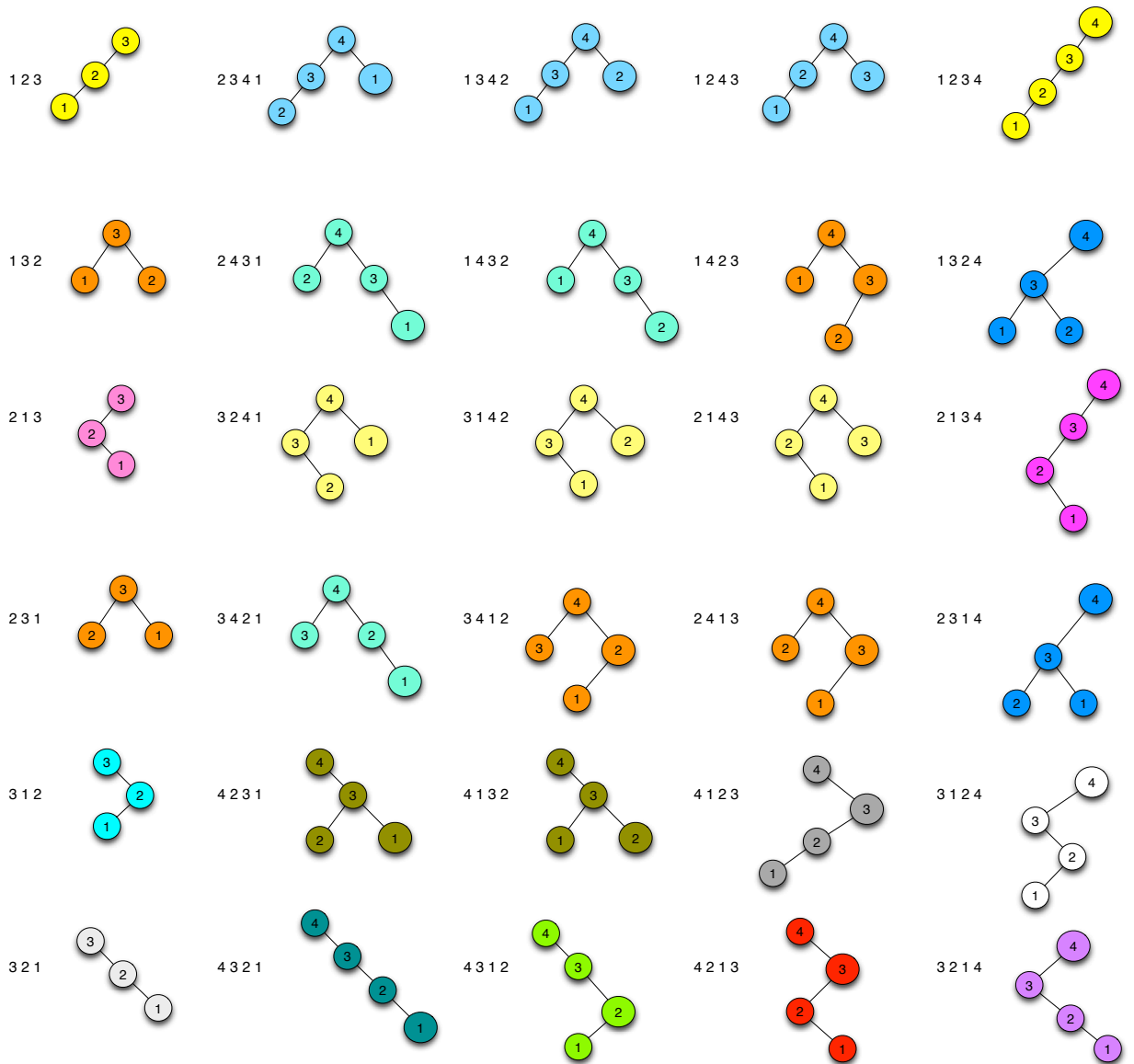


Figure 7.9: Result of random insertion on $MOQA\ TREAP(3)$

the number of comparisons needed to insert a random value into a right subtree with i nodes. We sum the comparisons for all possible right subtree size ranging from 1 to $n - 1$. In $k_i \times \bar{T}_{Insert}(\mathcal{TRRAP}(i))$, k_i is the number of $LPOs$ that are the result of inserting a new random label into a i nodes right subtree.

$$k_i = \binom{n-1}{i} \times i! \times (n-i-1)! \times n$$

Because the maximum label is always at the root, for a n node random treap, $\binom{n-1}{i}$ counts the number of ways we can choose the label set on the right subtree, and $i!$ and $(n-i-1)!$ are the permutations on the right and left subtrees. Thus we have $\binom{n-1}{i} \times i! \times (n-i-1)!$ $LPOs$ with i nodes at the right subtree. After inserting n possible new labels (with different overall ranks and discarding the largest label because the algorithm stops with one comparison), the new number of $LPOs$ with i nodes at the right subtree (before insertion) is $\binom{n-1}{i} \times i! \times (n-i-1)! \times n$, which is k_i .

Notice that:

$$\begin{aligned} \frac{k_i}{(n+1)!} &= \frac{\binom{n-1}{i} \times i! \times (n-i-1)! \times n}{(n+1)!} \\ &= \frac{(n-1)! \times \cancel{i!} \times \cancel{(n-i-1)!} \times n}{\cancel{i!} \times \cancel{(n-i-1)!} \times (n+1)!} \\ &= \frac{1}{n+1} \end{aligned}$$

Thus

$$\begin{aligned} \bar{T}_{Insert}(\mathcal{TRRAP}(n)) &= \frac{(n+1)! \times 1 + \sum_{i=1}^{n-1} (k_i \times \bar{T}_i^{Insert})}{(n+1)!} \\ &= 1 + \frac{1}{n+1} \times \sum_{i=1}^{n-1} \bar{T}_{Insert}(\mathcal{TRRAP}(i)) \end{aligned}$$

$$(n+1) \times \bar{T}_{Insert}(\mathcal{TRRAP}(n)) = (n+1) + \sum_{i=1}^{n-1} \bar{T}_{Insert}(\mathcal{TRRAP}(i)) \quad [a]$$

$$n \times \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-1)) = n + \sum_{i=1}^{n-2} \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(i)) \quad [b]$$

$$[a] - [b]$$

$$(n+1) \times \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n)) - n \times \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-1)) = 1 + \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-1))$$

$$\bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n)) = \frac{1}{n+1} + \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-1))$$

We can solve this recursion as follows:

$$\begin{aligned} \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n)) &= \frac{1}{n+1} + \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-1)) \\ \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-1)) &= \frac{1}{n} + \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-2)) \\ \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-2)) &= \frac{1}{n-1} + \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n-3)) \\ &\vdots \\ \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(2)) &= \frac{1}{3} + \bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(1)) \end{aligned}$$

Adding left and right parts and simplify them, we get:

$$\bar{T}_{Insert}(\mathcal{TR}\mathcal{E}\mathcal{A}\mathcal{P}(n)) = \begin{cases} 1 + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n+1} = H_{n+1} - \frac{1}{2} & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$$

where H_{n+1} is the $(n+1)^{th}$ Harmonic number. □

Remark 7.5. Since $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} < \int_1^n \frac{1}{x} dx$, $H_n < 1 + \ln n$ then $H_{n+1} - \frac{1}{2} < \frac{1}{2} + \ln(n+1)$. The treap insertion runs in $O(\log n)$.

7.2.2 Entropy Analysis based on Random Bags

The entropy function is used as a measurement of randomness. It is widely applied in information retrieval, machine learning etc [25, 52]. We recall the definition of entropy on a general random variable [68].

Definition 7.6. The entropy of a random variable X is given by

$$H(X) = - \sum_x Pr(X = x) \log_2 Pr(X = x).$$

This is calculated by the summation over all values x in the range of X .

For instance $X = \{a, a, b, c, c, c\}$, $Pr(X = a) = 2/6$, $Pr(X = b) = 1/6$, $Pr(X = c) = 3/6$, $H(X) = -Pr(X = a) * \log_2 Pr(X = a) - Pr(X = b) * \log_2 Pr(X = b) - Pr(X = c) * \log_2 Pr(X = c)$ thus $H(X) \approx 1.459$.

Using random bags, we show that it is possible to determine the entropy. Entropy determination could be a complement of average-case timing and provide extra information about an algorithm. Entropy information also might open up a new approach to support the average-case power estimation [92].

Theorem 7.7. The entropy of a random bag $R = \{(R_1, K_1), \dots, (R_n, K_n)\}$ is:

$$H(R) = - \sum_{i=1}^n |R_i| \frac{K_i}{|R|} \log_2 \frac{K_i}{|R|}$$

where $|R| = \sum_{i=1}^n K_i |R_i|$

Proof. Assume α_{ij} is the j^{th} LPO from R_i . Thus K_i copies of this LPO are in the random bag R . The probability of this LPO to occur in the random bag is: $P_{ij} = \frac{K_i}{|R|}$, where $|R| = \sum_{i=1}^n K_i |R_i|$ is the total number of LPOs in the random bag.

According to Definition 7.6,

$$\begin{aligned} H(R) &= - \sum_{i=1}^n \sum_{j=1}^{|R_i|} P_{ij} \log_2 P_{ij} \\ &= - \sum_{i=1}^n \sum_{j=1}^{|R_i|} \frac{K_i}{|R|} \log_2 \frac{K_i}{|R|} \end{aligned}$$

Because the LPOs belonging to the same structure R_i have equal probability $\frac{K_i}{|R|}$, we have: $H(R) = - \sum_{i=1}^n |R_i| \frac{K_i}{|R|} \log_2 \frac{K_i}{|R|}$ where $|R| = \sum_{i=1}^n K_i |R_i|$. \square

Recall that in the analysis of a sorting algorithm's average-case performance, unless stated otherwise, we will assume that the input list to the algorithm is a uniform random permutation, in which each of the $n!$ possible permutations are equally likely. In \mathcal{MOQA} we present a n element random input list as a random bag with a single random structure $R(\Delta_n)$. We refer to this random bag as $\mathcal{L}_n = \{(R(\Delta_n), 1)\}$.

We define a random variable $S = (s_1, s_2, s_3, \dots, s_{n!})$ to represent the random list to be sorted, where s_i is an instance of the random list. We have $|S| = n!$. Before sorting, the random variable $S = \mathcal{L}_n$.

Corollary 7.8. *The entropy of a random bag with a single random structure, $R = \{(R_1, K_1)\}$ is:*

$$H(R) = - \sum_{i=1}^1 |R_i| \frac{K_i}{|R|} \log_2 \frac{K_i}{|R|} = -\log_2 \frac{1}{|R_1|} = \log_2(|R_1|)$$

where $|R| = \sum_{i=1}^1 K_i |R_i| = K_1 |R_1|$

Remark 7.6. The entropy of a random bag with a single random structure is determined by the cardinality of the random structure. The multiplicity is not involved.

Example 7.8. The entropy for a random bag \mathcal{L}_n is given by

$$H(\mathcal{L}_n) = \log_2(n!)$$

Because $\mathcal{L}_n = \{(R(\Delta_n), 1)\}$, as shown in Corollary 7.8, its entropy is not related to the random structure's multiplicity. Because $|R_1| = |R(\Delta_n)| = n!$ we have

$$H(\mathcal{L}_n) = -\log_2 \frac{1}{n!} = \log_2(n!)$$

Due to the nature of sorting, at the end of the sorting process, the random variable S is equal to the random bag $\{(\mathcal{S}_n, n!)\}$, where \mathcal{S}_n represents sorted order. At the end of sorting $H(S) = 0$, because the only possible order left is the sorted order (This can be verified by Theorem 7.7). In the following we will show how $H(S)$ changes from computation step i to step $i + 1$ by analysing some common sorting algorithms such as Insertionsort, Treapsort and Mergesort.

7.2.3 Tracking Entropy Changes in Sorting Algorithms

We define the initial step entropy as the entropy before sorting, represented by $H_0(S)$. As shown in Corollary 7.8, $H_0(S) = \log_2(n!)$. The elements in a list are given by $(a_1, a_2, a_3 \dots a_n)$.

Insertionsort

Insertionsort keeps a sorted list and tries to insert a new element into it [26] (see Section 5.1). Initially the sorted list contains only the first element. The first step tries to insert a_2 into it, resulting in a sorted order $a_1 \geq a_2$. Notice that after this step, $H(S)$ had been changed because the first two elements have to be in the sorted order. After this first insertion, the first two elements are in order and the remaining elements still are in random order, thus the random variable S changed from \mathcal{L}_n to $\{(R(\mathcal{S}_2|\Delta_{n-2}), 2!)\}$. $\mathcal{S}_2|\Delta_{n-2}$ presents the partial order where the first two elements are in sorted order, and parallel with $n - 2$ random elements. The multiplicity for this random structure is $2!$ because the first two elements are in order, hence the original lists with the same elements in the first two positions result in the same *LPO*.

According to Corollary 7.8, after the first insertion,

$$H_1(S) = \log_2(|R(\mathcal{S}_2|\Delta_{n-2})|) = \log_2\left(\binom{n}{2}(n-2)!\right)$$

Generally, after step i , the first $i + 1$ elements will be in sorted order. The random bag will be $\{(R(\mathcal{S}_{i+1}|\Delta_{n-i-1}), (i+1)!)\}$.

Theorem 7.9. *In Insertionsort, after the i^{th} insertion,*

$$H_i(S) = \log_2\left(\binom{n}{i+1}(n-i-1)!\right) = \log_2(n!) - \log_2(i+1)!$$

where $i \leq n - 1$.

Proof. This is a direct result from Corollary 7.8. □

TreapSort

As stated earlier, TreapSort is a new sorting algorithm, based on the *MOQA* Treap data structure. We refer the reader to Section 5.6 (page 141) and [85] for details of this algorithm.

Firstly, each input list is transformed to a *MOQA* treap by the TreapGen algorithm, and each permutation order uniquely determines the shape of the treap, and maps to a unique *LPO* in the random bag.

At each sorting step, the maximum element will be pushed down in the treap and then placed back to the top. The rest of the elements still keep the random property.

Theorem 7.10. *After the i^{th} Perc^M operation in TreapSort,*

$$H_i(S) = \log_2((n - i)!)$$

where $i \leq n - 1$.

Proof. We start with random bag $\{(\text{TREAP}(n), 1)\}$, and $|\text{TREAP}(n)| = n!$, $H_0(S) = \log_2(n!)$. After the first Perc^M , according to Theorem 2.11, the random bag changed to $\{(\text{TREAP}(n - 1) \otimes \bullet, n)\}$. Next, after the second Perc^M , the random bag changed to $\{(\text{TREAP}(n - 2) \otimes \bullet \otimes \bullet, n(n - 1))\}$. Thus, after the i^{th} Perc^M , the random bag is $\{(\text{TREAP}(n - i) \otimes \mathcal{S}_i, \frac{n!}{(n - i)!})\}$. Applying Corollary 7.8 to the random bag, and because $|\text{TREAP}(n - i) \otimes \mathcal{S}_i| = (n - i)!$, we get the above result. \square

Mergesort

Mergesort is another widely used algorithm [26], which has both worst-case and average-case complexity $O(n \log n)$. There are two methods for implementing a Mergesort algorithm: a top-down approach or a bottom-up approach. We introduce a *MOQA* implementation with a top-down approach in Section 5.4. In this section, we use a bottom-up approach, because its random bag after the i^{th} merge is easier to represent, but the technique we used, with the necessary modification, should also work on the top-down version.

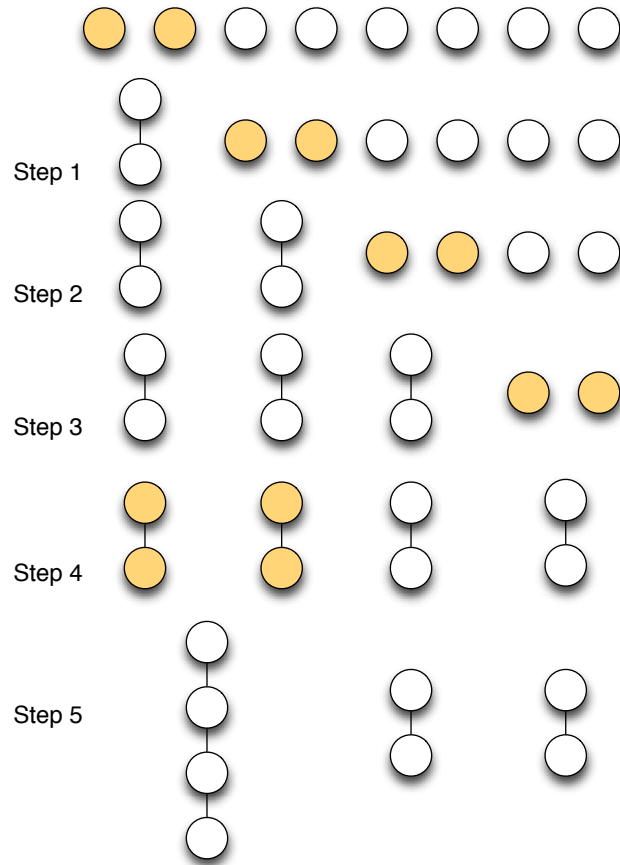


Figure 7.10: Bottom-up Mergesort example: first five merge operations

The bottom-up approach recursively breaks the array in half and then merges the results together. Bottom-up Mergesort loops over the list using intervals of varying sizes from 1 to $\frac{n}{2}$. At each step, adjacent intervals are merged together. Subsequent loops, execute over the list with larger intervals and merge our previously merged (smaller) intervals together.

In Figure 7.10, we show a bottom-up Mergesort example on the input with list size 8. We only present the first five merge operations. To sort the whole list we need two more merge operations. In the figure above, at each step the nodes involved in the merge operation are highlighted in color.

To know the entropy after the i^{th} merge, we need to know the output random bag. In the following we will assume that the number of elements in the list to

be sorted is a power of 2. The result will be that all of the sublists at each level of the recursion tree will have the same size. The recursion tree will be a full and balanced binary tree.

An example of such a recursion tree is shown in the left part of Figure 7.11. Notice that each node has a number associated with it. It represents the size of the list being sorted during that recursive call. The label M_i in each internal node represents the sequential (sequence) numbers of the merge operation when sorting the whole list. For example, M_1 means the first merge and it occurred when the algorithm merged the first two elements.

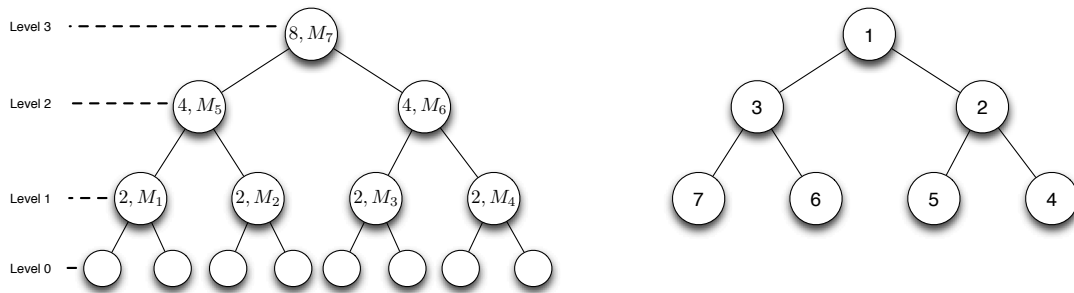


Figure 7.11: Mergesort recursion tree on list size eight

In the following, we will analyse the entropy in terms of each merging step. An equation will be derived to calculate the entropy value after the i^{th} merging.

Definition 7.7. For an input list of size 2^k , we define levels of a recursion tree from bottom to top. The leaf node is at level 0. The maximum level is k .

Example 7.9. For the recursion tree in Figure 7.11. $k = 3$, and total number of elements is $n = 2^k = 8$.

Lemma 7.11. After completing a merge at level i , 2^i more elements will be in order.

Proof. Because merging at level i will merge two lists from level $i-1$, each merged list has length 2^{i-1} . Thus after this merge 2^i , more elements will be in order. \square

Before we continue, we introduce the notation $\prod_i \mathcal{S}$. It constructs a random structure by parallel composing the component \mathcal{S} , i times. The parallel operation is denoted by \parallel .

Example 7.10.

$$\coprod_3 \mathcal{S}_2 = \mathcal{S}_2 \parallel \mathcal{S}_2 \parallel \mathcal{S}_2$$

Theorem 7.12. *To sort $n = 2^k$ elements, if we know the i^{th} merge is at level t and is the r^{th} merge at its level, then the resulting random structure after this merge is:*

$$\coprod_r \mathcal{S}_{2^t} \parallel \coprod_{\frac{n-r2^t}{2^{t-1}}} \mathcal{S}_{2^{t-1}}$$

where $\mathcal{S}_1 = \bullet$.

This means that the random structure is composed by r parallel \mathcal{S}_{2^t} random structures first, then paralleled with $\frac{n-r2^t}{2^{t-1}}$ $\mathcal{S}_{2^{t-1}}$ random structures. Then the entropy after the i^{th} merge is simply the logarithm of the cardinality of the structure.

$$H_i(S) = \log_2 \left(\prod_{i=0}^{r-1} \binom{n - i2^t}{2^t} \prod_{j=0}^{\frac{n-r2^t}{2^{t-1}} - 1} \binom{n - r2^t - j2^{t-1}}{2^{t-1}} \right)$$

because of Corollary 7.8.

Now, for the i^{th} merge operation, if we know its level number t and the value r (r^{th} merge in its level) in the recursion tree, according to Theorem 7.12, we can compute the resulting random structure and entropy. We illustrate this in the next example. The way to compute the values t and r is covered in Theorem 7.13.

Example 7.11. In Figure 7.11, the third merge M_3 is at level $t = 1$, with order $r = 3$. Thus according to Theorem 7.12, after this merge operation, the random structure is $\mathcal{S}_2 \parallel \mathcal{S}_2 \parallel \mathcal{S}_2 \parallel \bullet \parallel \bullet$, and its entropy

$$\begin{aligned} H_3(S) &= \log_2 \left(\prod_{i=0}^{3-1} \binom{8 - i2^1}{2^1} \prod_{j=0}^{\frac{8-3 \times 2^1}{2^{1-1}} - 1} \binom{8 - 3 \times 2^1 - j2^{1-1}}{2^{1-1}} \right) \\ &= \log_2 \left(\binom{8}{2} \binom{6}{2} \binom{4}{2} \binom{2}{1} \binom{1}{1} \right) = 12.299 \end{aligned}$$

Theorem 7.13. *When sorting $n = 2^k$ elements in Mergesort, the i^{th} merge operation is at level t and is the r^{th} merge at its level in the recursion tree, where*

$$t = k - \lfloor \log_2(n - i) \rfloor$$

$$r = i - n + \frac{n}{2^{t-1}}$$

Proof. We prove this with the help of a full binary tree. In this full binary tree, each node is marked with value 1 to n , and from top to bottom, right to left. We show one example of a full binary tree in the right part of Figure 7.11. It can be seen, for each i^{th} merge, that its corresponding node in the full binary tree is node $n - i$. E.g. In Figure 7.11, M_2 corresponds to node $8 - 2 = 6$, M_5 corresponds to node $8 - 5 = 3$. We define the level of the full binary tree from top to bottom, and starting with 0.

When sorting $n = 2^k$ elements in Mergesort, for the i^{th} merge operation M_i in the recursion tree, we denote its level in the recursion tree by t and the corresponding node in the full binary tree at level h , thus $t + h = k$ and we have:

$$t + \lfloor \log_2(n - i) \rfloor = k$$

For the i^{th} merge operation M_i , if at level t in the recursion tree, the first merge that occurred at this level is, say the j^{th} merge. Then $j = 1 + \frac{n}{2^1} + \frac{n}{2^2} + \dots + \frac{n}{2^{t-1}} = 1 + n - \frac{n}{2^{t-1}}$. And we have the order for the i^{th} merge at level t : $i - (1 + n - \frac{n}{2^{t-1}}) + 1 = i - n + \frac{n}{2^{t-1}}$ \square

Example 7.12. Sorting eight elements with Mergesort, M_3 is at level $t = 3 - \lfloor \log_2(8 - 3) \rfloor = 1$ with order $r = 3 - 8 + \frac{8}{2^{1-1}} = 3$. M_7 is at level $t = 3 - \lfloor \log_2(8 - 7) \rfloor = 3$ with order $r = 7 - 8 + \frac{8}{2^{3-1}} = 1$.

In this section, we have shown that \mathcal{MOQA} random bags support an easy approach to entropy analysis. We derived the formulas for computing the entropy after each step (or k steps) for several sorting algorithms. With this entropy information we could predict the randomness of output sequences. Some applications might avail of this benefit, such as partially sorting an input list. Another application is that we can now, after processing data, compute the entropy and hence the compressibility which could help with resource budgeting etc. The study of

these applications are beyond the scope of this thesis, which we leave it for future investigation.

7.3 Reversible Computing for $MOQA$

The results on reversible $MOQA$ is joint work with D. Early and M. Schellekens. Our paper has been published in [33]. The potential of reversible $MOQA$ was originally discussed by M. Schellekens in [87]. The mathematical proof of the frugal encoding has been obtained by D. Early. We focus on a simplification and an explanation of the original mathematical concepts. We approach the problem from a computer science point of view by providing pseudo-codes and illustrate the execution of algorithm's both forward and reverse, using real examples.

7.3.1 Background

Reversible $MOQA$ discussed in [87] and [32] complements traditional applications of reversibility with a new application domain, that of average-case cost analysis (where cost can be running time or power usage) of reversible $MOQA$ programs. Here, we provide the frugal encoding for the reversible $MOQA$ *Split* operation and illustrate the approach via a reversible version of the well-known Quicksort algorithm.

Reversibility traditionally plays a role in hardware design, with implications for low power design [19, 60, 106]. A few exceptions focus on high-level reversible languages, including the language JANUS and the work discussed in [116]. Most reversible approaches remain at hardware level. As observed, the use of $MOQA$ as a high level reversible language brings a new type of application to the area of reversible computing. As pointed out in [85] a sufficient condition for algorithms to be analyzable in a modular way is that they are random bag preserving. Not all algorithms are random bag preserving, a case in point being the traditional heapsort algorithm [85]. As shown in [87], random bag preservation can typically be guaranteed by ensuring a “locally” one-to-one mapping, e.g. a mapping guaranteed to be one-to-one on each of the parts of a partition of the inputs¹.

¹For example, the $MOQA$ product operation as discussed in [85], Theorem 5.1

MOQA's random bag preserving programs are ensured to allow for a greatly simplified average-case analysis. The key to understanding *MOQA* as a new application domain for reversible computing is that its programs, with little additional book-keeping become fully reversible [32, 87, 90]. Hence we establish a link between reversibility and the capacity for modular (i.e. semi-automated) average-case analysis. Of course, general algorithms typically can be subjected to average-case analysis techniques. The key point is that some algorithms, like heapsort, are not random bag preserving and hence either require complicated (non-automatable) techniques or escape average-case analysis by current techniques. As a degree of reversibility lies at the heart of random bag preservation [87], reversibility has the potential to play a fundamental role in the design of modularly predictable algorithms. Since with a little more bookkeeping, *MOQA* becomes a fully reversible language, the exploration of its reversible properties is worthwhile, in particular since the reversible programs in turn allow for an exact prediction of average-case computation time. Hence we can predict in a static way the cost of computing forward and backward in the language.

The reversible aspects of *MOQA* open up possibilities to apply *MOQA* to determine the average-case power usage but possibly also to use *MOQA* to achieve power optimization based on traditional reversible approaches [115]. We refer the reader to Chapter 2 for the necessary introduction to *MOQA* data structures.

7.3.2 Reversible Split Operation

We refer the reader to Section 2.3.3.1 (page 28) for an introduction on the *MOQA Split* operation. In the following context, for the resulting structure of a *Split* operation, we will call the upper part Y_1 , the middle pivot Y_2 and the bottom part Y_3 .

Efficient Encoding

First, we show how to efficiently encode the information needed to reverse the split of a list into two sublists (upper part and bottom part). We assume that all lists are ordered.

Clearly, if we know the position of each item in the upper sub-list in the original list, this is enough. For example, if the two sub-lists are (f, g, q, p, z) and (m, s, b, t) , and if we know that the elements of the upper list were initially in position 1, 3, 4, 8 and 9, then the original list except pivot element must be:

$$\begin{array}{cccccccc} (f, & m, & g, & q, & s, & b, & t, & p, & z) \\ & 1 & & 3 & 4 & & & 8 & 9 \end{array}$$

So, given positive integers $k < n$, suppose we have two lists of length k and $n - k$. To combine them in the original order, we need k distinct integers between 1 and n . We write these as $\{x_i\}_{i=1}^k$, where the x_i are in ascending order. Clearly there are $\binom{n}{k}$ different sets with these properties.

One way to encode a subset of size k from a set of size n is with a binary string of length n , whose i^{th} bit is 1 if and only if the i^{th} element of the set is included in the subset. However, if k is very small or very large (relative to n), this encoding is very inefficient. For example, we can encode a one-element subset with a number between 1 and n , or $\log_2(n)$ bits, whereas this method would require n bits.

Using this method to reverse the split operation would give a worst-case reversal overhead for Quicksort of $O(n!2^{\frac{(n^2-n)}{2}})$. In our case, the overhead here is the number of different possible combinations (and hence the amount of space needed to store all possible configurations needed to do the reversal). So if there are two numbers to be recorded, when the first can assume p different values and the second can assume q different values, then in total we need to be able to store $p \times q$ different values to be able to record all distinct cases. In terms of this method, for reversing the whole *Split* operation, which would record what position the pivot was in and record for each element whether it was above or below, the overhead for split on a list of length n would be $n2^{n-1}$. In the worst case where the list is already sorted, the total overhead would then be: $\prod_{i=n}^1 i2^{i-1} = n!2^{n-1+n-2+\dots+2+1}$, which is $O(n!2^{\frac{(n^2-n)}{2}})$. We will show that a more frugal encoding can achieve the same result with a maximum overhead of $n!$.

The following lemma shows how to encode position indices $\{x_i\}_{i=1}^k$.

Lemma 7.14 (D. Early [33]). *Given a positive integer n and an integer $k \in [0, n]$, the function $f(\{x_i\}_{i=1}^k) = \sum_{i=1}^k \binom{x_i-1}{i}$ is a one-to-one mapping from the k -element subsets of the first n integers (in ascending order) to the set of integers from 0 to $\binom{n}{k} - 1$.*

Proof. First, we prove that no two subsets map to the same value (i.e., f is an injection). Suppose that $f(\{x_i\}_{i=1}^k) = f(\{y_i\}_{i=1}^k)$, and that $x_i \neq y_i$ for some $i \in [1, k]$. Let j be the largest value of i for which $x_i \neq y_i$. We can assume that $x_j > y_j$. Now for all $i \leq j$, $y_i \leq x_j - 1 - j + i$ (note: $x_j > y_j$, $y_j > y_i$, both x_i and y_j are integers in range $[1, n]$) and so:

$$\begin{aligned} \sum_{i=1}^j \binom{y_i-1}{i} &\leq \sum_{i=1}^j \binom{x_j-2-j+i}{i} = -1 + \sum_{i=0}^j \binom{x_j-2-j+i}{i} \text{ (change of index)} \\ &= -1 + \binom{x_j-1}{j} < \binom{x_j-1}{j} \end{aligned}$$

Where the last equation follows from the hockeystick lemma [117].

So:

$$\begin{aligned} f(\{y_i\}_{i=1}^k) &= \sum_{i=1}^k \binom{y_i-1}{i} = \sum_{i=1}^j \binom{y_i-1}{i} + \sum_{i=j+1}^k \binom{x_i-1}{i} \\ &< \sum_{i=j}^k \binom{x_i-1}{i} \leq f(\{x_i\}_{i=1}^k) \end{aligned}$$

Which contradicts the assumption. To avoid contradiction, f must be an injection.

Now we prove upper and lower bounds on f . Clearly $f(\{x_i\}_{i=1}^k) \geq 0$. To get the upper bound, note that $x_i \leq n - k + i$, and thus:

$$\begin{aligned}
f(\{x_i\}_{i=1}^k) &\leq \sum_{i=1}^k \binom{n-k-1+i}{i} = -1 + \sum_{i=0}^k \binom{n-k-1+i}{i} \text{ (change of index)} \\
&= -1 + \binom{n}{k}
\end{aligned}$$

And the last equation again follows from the hockeystick lemma [117]. So the range of f is $[0, \binom{n}{k} - 1]$. But now there are $\binom{n}{k}$ distinct subsets, $\binom{n}{k}$ possible outputs and each input maps to a distinct output, so f must be one-to-one. \square

We provide a brief intuition for the indexing of the subsets:

- We define an order on the subsets whereby subset A is greater than subset B if the largest element in one set but not the other is in subset A. Consider the number of subsets less than a given subset, which contains the elements of ranks x_1, x_2, \dots, x_k in the full set.
- If the element of rank x_p in the overall set is the largest that is not common to both subsets, then all the larger elements are in common, and the smaller subset can have any p elements from among the smallest $x_p - 1$ in the set, a total of $\binom{x_p-1}{p}$ possibilities.
- But now, for any pair of distinct subsets, there is only one largest element in one but not the other, and so any subset smaller than the given one must match this pattern for some $p \in [1, k]$. So the total number of smaller subsets is $\sum_{i=1}^k \binom{x_i-1}{i}$.
- Now, assigning each subset an index which is the number of smaller subsets gives each subset a unique index between 0 and $\binom{n}{k} - 1$.

We also briefly outline an algorithm for extracting the sequence $\{x_i\}_{i=1}^k$ given $f(\{x_i\}_{i=1}^k)$ (and also the values of n and k) in Algorithm 7.6.

Algorithm 7.6 Extracting the sequence $\{x_i\}_{i=1}^k$ given $f(\{x_i\}_{i=1}^k)$

Input: N -- $f(\{x_i\}_{i=1}^k)$, n -- size of original list, k -- sublist size.

Output: S (a set $\{x_i\}_{i=1}^k$)

Extract(N, n, k) :

$j \leftarrow k$

$S \leftarrow \emptyset$

for $i \leftarrow n$ to 1 **do**

if $N \geq \binom{i-1}{j}$ **then**

$S \leftarrow \{i\} \cup S$

$N \leftarrow N - \binom{i}{j}$

$j \leftarrow j - 1$

end if

end for

return S

Example 7.13. Suppose we split $(f, m, g, q, s, b, t, p, z)$ into (f, g, q, p, z) and (m, s, b, t) . Then $(x_1, x_2, x_3, x_4, x_5) = (1, 3, 4, 8, 9)$.

$$\begin{aligned} f(\{x_i\}_{i=1}^5) &= \binom{9-1}{5} + \binom{8-1}{4} + \binom{4-1}{3} + \binom{3-1}{2} + \binom{1-1}{1} \\ &= 56 + 35 + 1 + 1 + 0 = 93 \end{aligned}$$

Now, given $N = 93$, $n = 9$, $k = 5$, we set $j = 5$ and run the algorithm:

$i = 9, j = 5, 93 > \binom{9-1}{5}$ so $S = \{9\}, j = 4, N = 93 - \binom{9-1}{5} = 37$

$i = 8, j = 4, 37 > \binom{8-1}{4}$ so $S = \{8, 9\}, j = 3, N = 37 - \binom{8-1}{4} = 2$

$i = 7, j = 3, 2 < \binom{7-1}{3}$ so skip

$i = 6, j = 3, 2 < \binom{6-1}{3}$ so skip

$i = 5, j = 3, 2 < \binom{5-1}{3}$ so skip

$i = 4, j = 3, 2 > \binom{4-1}{3}$ so $S = \{4, 8, 9\}, j = 2, N = 2 - \binom{4-1}{3} = 1$

$i = 3, j = 2, 1 = \binom{3-1}{2}$ so $S = \{3, 4, 8, 9\}, j = 1, N = 1 - \binom{3-1}{2} = 0$

$i = 2, j = 1, 0 < \binom{2-1}{1}$ so skip

$i = 1, j = 1, 0 = \binom{1-1}{1}$ so $S = \{1, 3, 4, 8, 9\}, j = 0, N = 0 - \binom{1-1}{1} = 0$

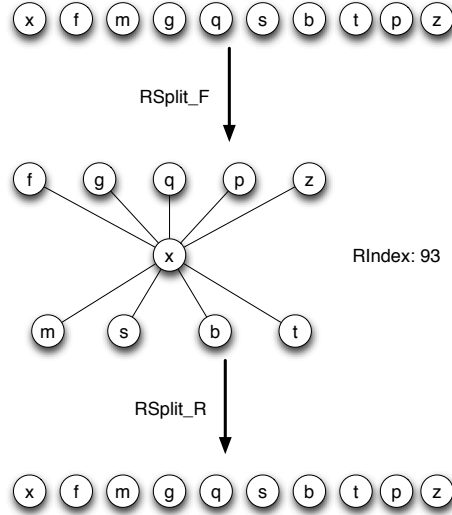


Figure 7.12: Forward and reverse split on random list $[x, f, m, g, q, s, b, t, p, z]$

Algorithm 7.7 Reversible Split algorithm: Forward computing

Input: a discrete *LPO* L .

Output: a three-layered series *LPO* (Y_1, Y_2, Y_3) and reversal index : *RIndex*.

$RSplit_F(L)$: ▷ Using the book notation, the top part is Y_1 ,
 $(Y_1, Y_2, Y_3) \leftarrow Split(L)$ ▷ pos maps element y_i to its position x_i in L
 $RIndex \leftarrow f(\{pos(y_i) - 1\}_{y_i \in Y_1})$ ▷ f defined in Lemma 7.14
return $(Y_1, Y_2, Y_3), RIndex$

Thus, for a *MOQA Split* operation we can keep track of the encoding for the upper elements in the resulting *LPO* and restore their original position with the help of the *Extract* algorithm. In reverse computing, each operation will have two versions, one for forward computing and the other for reverse computing. We design the reversible *Split* operation in Algorithm 7.7 and Algorithm 7.8. The forward version using the encoding we define in Lemma 7.14 calculates a reversal index. The reverse computing version, uses the reversal index to restore the structure.

Algorithm 7.8 Reversible Split algorithm: Reverse computing

Input: a three-layered series $LPO (Y_1, Y_2, Y_3)$ and reversal index : $RIndex$.
Output: a discrete $LPO L$.

```
 $RSplit\_R((Y_1, Y_2, Y_3), RIndex) :$   
 $n \leftarrow |Y_1| + |Y_2| + |Y_3|$   
 $X = Extract(RIndex, n, |Y_1|)$   
 $L \leftarrow [Y_2]$   
for  $i \leftarrow 2$  to  $n$  do  
  if  $i - 1 \in X$  then  
     $L \leftarrow L + Y_1[1]$   
     $Del(Y_1[1])$  ▷ Remove first element from  $Y_1$   
  else  
     $L \leftarrow L + Y_3[1]$   
     $Del(Y_3[1])$  ▷ Remove first element from  $Y_3$   
  end if  
end for  
return  $L$ 
```

Example 7.14. We provide an example for the forward and reverse split operation in Figure 7.12.

7.3.3 Reversible Quicksort

We define a reversible Quicksort algorithm, Q' in Algorithm 7.9, which takes a discrete $LPO L$ as an argument and returns a linear $LPO L^*$ and an integer between 1 and $|L|!$. Given L^* and the integer, we can recover L .

We note that $C_2 \in [1, |Y_3|!]$ by assumption, $C_1 \in [1, |Y_1|!]$ by assumption. $C_0 \in [0, \binom{|L|-1}{|Y_1|} - 1]$ from Lemma 7.14, and $|Y_3| \in [0, n - 1]$ from the definition of $Split$, and so the min and max values of the code returned are 1 and

$$\begin{aligned} & (|L| - 1)(|L| - 1)! + \left(\binom{|L| - 1}{|Y_1|} - 1 \right) |Y_1|! |Y_3|! + (|Y_1|! - 1) |Y_3|! + |Y_3|! \\ &= (|L| - 1)(|L| - 1)! + \left(\frac{(|L| - 1)!}{|Y_1|! |Y_3|!} - 1 \right) |Y_1|! |Y_3|! + (|Y_1|! - 1) |Y_3|! + |Y_3|! \\ &= |L|! - (|L| - 1)! + (|L| - 1)! - |Y_1|! |Y_3|! + |Y_1|! |Y_3|! - |Y_3|! + |Y_3|! = |L|! \end{aligned}$$

Algorithm 7.9 Reversible Quicksort algorithm: Forward computing

Input: a discrete *LPO* L Output: a linear *LPO* L^* and reversal index

```
 $Q'(L) :$ 
if  $|L| \leq 1$  then
  return  $(L, 1)$ 
else
   $(Y_1, Y_2, Y_3), C_0 \leftarrow RSplit\_F(L)$   $\triangleright$  The top part is  $Y_1$ ,
 $\triangleright$  the pivot is  $Y_2$  and the bottom part is  $Y_3$ .
   $(Y_1, C_1) \leftarrow Q'(Y_1)$   $\triangleright$  Let the code returned be  $C_1$ 
   $(Y_3, C_2) \leftarrow Q'(Y_3)$   $\triangleright$  Let the code returned be  $C_2$ 
  return  $([Y_1 : Y_2 : Y_3], |Y_3|(|L| - 1)! + C_0|Y_1|!|Y_3|! + (C_1 - 1)|Y_3|! + C_2)$ 
end if
```

as expected.

Note that this encoding is the most efficient possible, since all $n!$ different unsorted lists are mapped to the same sorted output.

We briefly outline the intuition for the reversal index returned.

In order to reverse Quicksort, using the recursive structure of the algorithm, we need three pieces of information: (i) the location of the pivot, which is encoded by $|Y_3|$, the number of elements placed below the pivot, (ii) the order in which the nodes above and below the pivot originally appeared, which is encoded by C_0 as outlined in the previous section, and (iii) the information needed to reverse each of the two recursive calls on the upper and lower parts, which are encoded in C_1 and C_2 respectively. We would like to store these four numbers in a way that allows us to recover each of them.

We could store them as a quadruple (a, b, c, d) , but then the recursion would mean that c and d were tuples themselves, and the final n -tuple could be very large — so we need to combine them. The way we do this is similar to the different digits in a number. To store 4 numbers a, b, c , and d between 0 and 9, we can compute $N = a * 10^3 + b * 10^2 + c * 10 + d$, and easily extract each one. In the same way, if a, b, c , and d are non-negative and less than some different upper bounds A, B, C , and D (where in the previous case $A = B = C = D = 10$), then we can encode the combination as $N = a * BCD + b * CD + c * D + d$. This is

the essence of how we have encoded the reversal index (with some adjustments for codes that range between 1 and n instead of 0 and $n - 1$).

We can then extract the digits using floors and ceilings. For example, to extract b from N above, we can use $b = \lfloor N/CD \rfloor - B * \lfloor N/BCD \rfloor$. The first floor expression gives $a * B + b$, because $c/BC + d/BCD$ is the fractional part of N/CD , and similarly the second one gives a . Again, the technique needs to be adapted slightly for codes that range between 1 and n instead of 0 and $n - 1$, but the essential idea is the same. We now show how we can use this information to construct a reverse Quicksort algorithm.

Given a sorted list L^* and a reversal index N , we can reverse Q' to get the input LPO as follows:

Algorithm 7.10 Reversible Quicksort algorithm: Reverse computing

Input: a sorted list L^* and a reversal index N

Output: a discrete LPO L

$\overline{Q}'(L^*, N) :$

if $|L^*| \leq 1$ **then**

return L^*

else

$k \leftarrow \lceil \frac{N}{(|L^*+1|)!} \rceil$ $\triangleright k^{th}$ smallest label in L^* is the first pivot.

$Y_2 \leftarrow L^*[k]$ $Y_1 \leftarrow L^*[1 : k - 1]$ $Y_3 \leftarrow L^*[k + 1 : end]$
 $\triangleright : L^*[a : b]$ means elements from a to b

$C_2 \leftarrow N - (k - 1)! \lfloor \frac{N-1}{(k-1)!} \rfloor$ $C_1 \leftarrow 1 + \frac{N - C_2 - (k-1)! (|L^*+k)! \lfloor \frac{N-1}{(k-1)! (|L^*+k)!} \rfloor}{k-1!}$
 \triangleright compute reversal index for upper and bottom parts.

$Y_1 \leftarrow \overline{Q}'(Y_1, C_1)$ \triangleright Reverse upper part

$Y_3 \leftarrow \overline{Q}'(Y_3, C_2)$ \triangleright Reverse bottom part

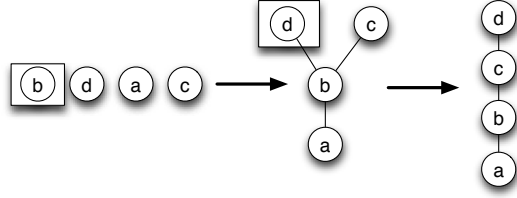
$C_0 \leftarrow \frac{N - (|L^*+1)! (k-1) - (C_1-1) (k-1)! - C_2}{(k-1)! (|L^*+k)!}$

$L \leftarrow RSplit_R((Y_1, Y_2, Y_3), C_0)$

return L

end if

Example 7.15. We use Quicksort to sort the list $[b, d, a, c]$ as follows:



Forward Computing

Let's start with Q' first. A split operation applied on the list $[b, d, a, c]$ will partition it into two sublists $[d, c]$ and $[a]$, and the according positions in the original list for sublist $[d, c]$ are $[1, 3]$, thus $C_0 = f(\{x_i\}_{i=1}^2) = \binom{3-1}{2} + \binom{1-1}{1} = 1$. Recursively we look at the upper part, two sublists $[c]$ and an empty list. So $C_1 = 1 \times (2-1)! + \binom{1-1}{1} \times 0! \times 1! + (1-1)! + 1 = 2$. For the down part, because it only has one element, so $C_2 = 1$. Thus the final code returned for this sorting is: $1 \times (4-1)! + 1 \times 2! \times 1! + (2-1)1! + 1 = 6 + 2 + 1 + 1 = 10$. So Q' will give us the sorted list $[d, c, b, a]$ and the reversal index 10. Next let us reverse the sorting.

Reverse Computing

First find the first pivot, $k = \lceil \frac{N}{(|L^*+1|)} \rceil = \lceil \frac{10}{(4-1)!} \rceil = 2$, thus the second smallest label is the pivot, which is b . So Y_1 contains $[d, c]$, Y_3 contains $[a]$. $C_2 = N - (k-1)! \lfloor \frac{N-1}{(k-1)!} \rfloor = 10 - (2-1)! \lfloor \frac{10-1}{(2-1)!} \rfloor = 1$ and $C_1 = \frac{N - C_2 - (k-1)! \lfloor \frac{N-1}{(k-1)! \lfloor \frac{N-1}{(L^*+k)!} \rfloor} \rfloor}{k-1!} + 1 = \frac{10 - 1 - (2-1)! \lfloor \frac{10-1}{(2-1)! \lfloor \frac{9}{2} \rfloor} \rfloor}{1} + 1 = \frac{10 - 1 - 2 \lfloor \frac{9}{2} \rfloor}{1} + 1 = 2$.

Recursively we reverse $Y_1 = [d, c]$ with reversal index 2: pivot $k = \lceil \frac{N}{(|L^*+1|)} \rceil = \lceil \frac{2}{(2-1)!} \rceil = 2$, thus the second smallest element d is the pivot in the sorted sublist $[d, c]$. $C_2 = 2 - (2-1)! \lfloor \frac{2-1}{(2-1)!} \rfloor = 1$, $C_1 = 1 + \frac{2-1-(2-1)!(2-2)! \lfloor \frac{2-1}{(2-1)!(2-2)!} \rfloor}{(2-1)!} = 1$. For its recursive cases, only containing a single element and the empty element, we restore the sorted sublist $[d, c]$ with $C_0 = \frac{2-(2-1)!(2-1)-(1-1)(2-1)!-1}{(2-1)!(2-2)!} = 0$. The final reversed sublist places the pivot d in first position, places upper elements (empty) according to the extracted sequence from C_0 (still empty), combined with reversed bottom elements (the single element c). We obtain the reversed unsorted sublist $[d, c]$ for this recursive call.

Similarly, we restore Y_3 with reversal index 1 and obtain the reversed sublist $[a]$.

Finally we combine the two reversed sublists $[d, c]$ and $[a]$ with $C_0 = \frac{10-(4-1)!(2-1)-(2-1)(2-1)!-1}{2!} = 1$. According to Algorithm 7.6, we can obtain the original position indices for $[d, c]$, which is $[1, 3]$ as can be verified in Section 7.15. Thus finally we reversed Quicksort and restored the input sequence $[b, d, a, c]$.

7.4 Smoothed Analysis for $MOQA$

This work is based on the recent research carried out in our research group [89]. We focus on how it integrates with our interpreter analyzer. Prior work from M. Schellekens and D. Early [88] has shown the fruitful links between $MOQA$ and smoothed complexity analysis.

Smoothed analysis is a recent approach to measure how unusual the worst-case running time is [100]. This new measurement is valuable for many algorithms where there is a wide divergence between average-case and worst-case running times. For example, Quicksort has an optimal average-case running time $O(n \log n)$ for length n random lists, while it has a worst-case running time of $O(n^2)$. Because, in a practical context, most of data is not uniformly distributed, it is crucial to know how likely the algorithm will perform with its worst-case running time. Smoothed analysis tries to answer this question by considering inputs that are subject to some random perturbation. Perturbations can be defined in various ways. We will give one such definition below.

Definition 7.8. The smoothed Measure of an algorithm acting on an input is the average running time of the algorithm over the perturbations of that input instance [100].

Definition 7.9. The smoothed complexity of an algorithm is the the worst smoothed measure of the algorithm on any input instance [100].

The early work on smoothed complexity [100], dealt with continuous problems such as the Simplex method. [17] proposes a similar approach for discrete sorting

problems based on a method called *partial permutations*. We follow this model in \mathcal{MOQA} smoothed complexity but with a slight simplification introduced in [89].

Definition 7.10. Let $S = (s_1, s_2, \dots, s_n)$ be a sequence of n elements, where a probability $\sigma \in [0, 1]$. A σ -*partial permutation* of S is a random sequence $S' = (s'_1, s'_2, \dots, s'_n)$ obtained from S in two steps [17]

- Each element from S is selected with probability σ (independent selections)
- if m elements are selected, we choose one of the $m!$ permutations of the selected elements (uniformly at random) and rearrange them in that order, leaving the positions of all the other elements unchanged.

The parameter σ is used to measure the degree of perturbation. When σ becomes large, the perturbations on the input become significant, and the smoothed complexity tends towards the average-case running time. On the other hand, as σ becomes small, the perturbations become insignificant on the original instance, and the smoothed complexity tends towards the worst-case running time.

As a first step to merging the \mathcal{MOQA} approach with smoothed complexity analysis, we follow the definition of σ -*partial permutations* according to [89]. Recall, we generally assume that any \mathcal{MOQA} program starts from a discrete partial order (random lists as input).

Definition 7.11. Let $S = (s_1, s_2, \dots, s_n)$ be a sequence of n elements, if $\sigma = \frac{k}{n}$ ($0 \leq k \leq n$), we define a σ -*partial permutations* function $Pert_k^n$. It maps each sequence $s \in S$ to a collection of n elements sequences as follows:

- For all possible subsets of k elements out of S
- we choose all of the $k!$ permutations of the elements and rearrange the original elements according to these permutations.

For a n elements list, after $Pert_k^n$, a group of $\binom{n}{k} k!$ lists are obtained.

Example 7.16. Let $S = (a, b, c)$, $Pert_2^3(S) = \{(a, b, c), (b, a, c), (a, b, c), (c, b, a), (a, b, c), (a, c, b)\}$.

In \mathcal{MOQA} smoothed complexity, instead of σ , we generally refer to degree of perturbation by the k value.

Definition 7.12 ([89]). For a \mathcal{MOQA} program P , with inputs n elements random lists, the \mathcal{MOQA} smoothed complexity with perturbation value k is defined by:

$$T_P^S(n, k) = \text{Max}_{s \in R(\Delta_n)}(\overline{T}_P(\text{Pert}_k^n(s)))$$

Remark 7.7. The formal justification is provided in [89]. With this “simplified” definition, we can have a very clear interpretation of the degree of perturbation (see Example 7.17). Notice that to obtain smoothed complexity, the average-case analysis of the program with certain inputs is needed, and this is where the original \mathcal{MOQA} theory comes into play.

Example 7.17. Given a \mathcal{MOQA} program P with $R(\Delta_3)$ as input.

- For the case of $k = 1$, we consider partial permutations Pert_1^3 on inputs $\mathcal{I} = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$. For each input, there are three possible selections on 1 element. The only permutation on 1 element is the identity. Hence e.g. $\text{Pert}_1^3((2, 1, 3)) = \{(2, 1, 3), (2, 1, 3), (2, 1, 3)\}$

$$\overline{T}_P(\text{Pert}_1^3((2, 1, 3))) = \frac{\sum_{i=1}^3 T_P((2, 1, 3))}{3} = T_P((2, 1, 3))$$

So in general:

$$T_P^S(n, 1) = \text{Max}_{s \in R(\Delta_n)}(T_P(s)) = T_P^W(n)$$

This verifies that the smallest perturbations yield the worst-case time.

- For the case of $k = n$, for each input $s \in R(\Delta_n)$, $\text{Pert}_n^n(s) = R(\Delta_n)$. So:

$$T_P^S(n, n) = \text{Max}_{s \in R(\Delta_n)}(\overline{T}_P(R(\Delta_n))) = \overline{T}_P(n)$$

This verifies that the largest perturbations yield the average-case time.

Thus, the \mathcal{MOQA} smoothed complexity is a function of k (or viewed as σ) which interpolates between the worst-case and average-case running times. The dependence on the perturbation value k gives a sense of how unusual an occurrence of the worst-case input actually is.

Smoothed Analysis of Quicksort

In the following, we present how smoothed analysis is integrated with our interpreter analyzer, in more recent research, following up from [89]. Smoothed analysis of Quicksort is used as an example. We use results obtained in [89] and omit details of how these results are derived.

Recall that Quicksort is based on the *MOQA Split* operation (see Section 5.2, page 131). To facilitate *MOQA* smoothed analysis, a smoothed split operation *Split^S* is obtained in [89]. In comparison with the original split operation, the smoothed version produces different multiplicities based on the perturbation value k . Based on this modified resulting random bag, the smoothed complexity is obtained. We outline the smoothed split operation below:

Theorem 7.15 ([89]).

$$Split^S : R(\Delta_n) \mapsto \{(R(P[0, n-1]), K_{n-1}, P_{n-1}), \dots, (R(P[n-1, 0]), K_0, P_0)\}$$

and where

$$K_i = \begin{cases} \frac{(n-k-1)(n-1)!}{(n-k)!}, & \text{if } i = 0 \\ \binom{n}{k} \frac{k!(k-1)}{n(n-1)} \binom{n-1}{j-1}, & \text{if } i \neq 0 \end{cases}$$

$$P_i = \begin{cases} \frac{n-k+1}{n}, & \text{if } i = 0 \\ \frac{k-1}{n(n-1)}, & \text{if } i \neq 0 \end{cases}$$

and k is perturbation value, K_i and P_i are the multiplicity and probability of the i^{th} random structure respectively.

$$\bar{T}_{split^S}(\Delta_n) = n - 1$$

To integrate smoothed analysis in our interpreter analyzer, we simply modify the original *Split* operation to *Split^S*. Most of our codes do not need to be modified, because the shapes of resulting random structures do not change. We only modify the codes to calculate the multiplicity and the probability for each random structure and the smoothed complexity is obtained by the same process as in analysis mode. The analyzer walks over the abstract syntax tree and ab-

strictly interprets *MOQA* operations based on random bags and accumulates the number of comparisons made. Similar to the Quicksort average-case analysis (See Section 5.2). Based on the modified multiplicity, the smoothed complexity we are computing is: $T_{qsort}^S(n, k) = n + \sum_{i=0}^{n-1} P_i(T_{qsort}^S(i, k) + T_{qsort}^S(n - i - 1, k))$ where P_i is probability of the i^{th} random structure, which is defined in Theorem 7.15. (The details of Quicksort smoothed complexity are shown in [89]).

In the shell command, a user can use the flag `-S` to determine whether to use smoothed analysis mode. By this method, the interpreter can switch between the normal *Split* or and the smoothed *Split^S* operation.

Consider the following command:

```
1 $ python moqa.py -S50,5 Quicksort.moqa
```

It invokes the *MOQA* interpreter smoothed analysis mode on `Quicksort.moqa`. `-S50,5` tells the interpreter to do the smoothed analysis (`-S`) with an initial partial order of size 50 and a perturbation value $k = 5$.

Figure 7.13 illustrates the results produced by our interpreter analyzer with different degrees of perturbation ¹. It can be seen, with increasing perturbation value, that the running time gradually becomes smaller and reaches its average-case timing when the perturbation value equals the input list size. With this automated result, in future work, we could study how the algorithm is affected by perturbations in order to identify where the performance of the algorithm changes from worst-case to average-case. This information could support predicting the unlikelihood of worst-case occurrences which might support soft real-time applications.

7.5 Summary

In this chapter, we presented a number of useful applications of the *MOQA* method. By examining random bag preservation properties of several heap creation algorithms, we identified possible candidates that might be further inves-

¹The formulas in Theorem 7.15 only focus on the smoothed complexity with perturbation value $k \geq 2$, because our interpreter relies on these results only. The worst-case complexity ($k = 1$) is calculated via the standard analysis technique.

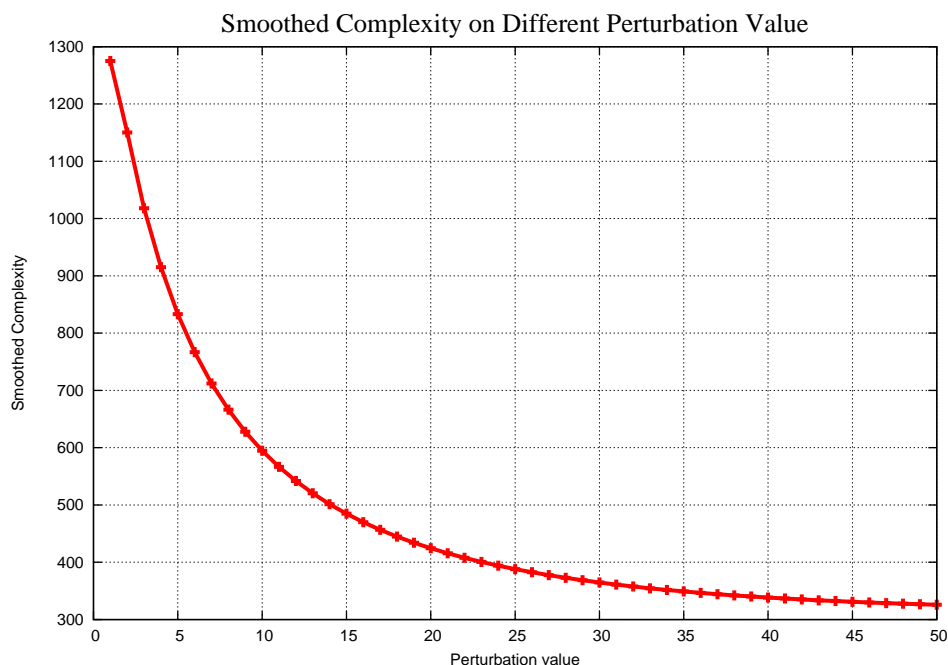


Figure 7.13: *MOQA* Smoothed Analysis of Quicksort with input list size 50 on different degrees of perturbation

tigated in future *MOQA* research. Skew heap and Min-Max heap are found to be reasonable candidates, for their partial order creations are random bag preserving, while Leonardo Heap is not. With necessary extensions in *MOQA*, i.e., by extending the conditional expression to allow for odd-even level detection etc, these new structures might be introduced in *MOQA* and support the automated time analysis in the future.

Next, the insertion operation was introduced for the *MOQA* treap data structure and its complexity was analysed.

Then, based on *MOQA* random bags and the ability track data distributions throughout computation, entropy changes have been derived for several algorithms. Also, in this chapter, we showed the frugal encoding underpinning the reversible *MOQA Split* operation, where the encoding can be achieved through the bookkeeping of a single number. The applicability of the encoding has been demonstrated via reversible Quicksort. Finally, we briefly discussed *MOQA* smoothed analysis. With a small extension to our interpreter analyzer, we can

provide a smoothed analysis for \mathcal{MOQA} programs and a smoothed analysis for Quicksort has been given as an example.

Chapter 8

Conclusions and Future Work

Contents

8.1	Conclusions	208
8.2	Future Work	210
8.2.1	From An Application Perspective	210
8.2.2	From A Theoretical Perspective	212

8.1 Conclusions

This thesis has considered the problem of *MOQA* research extensions from both practical and theoretical aspects.

Trying to overcome the problems pointed out in Section 3.2.1 (page 47), we developed a new domain specific language (Chapter 3 to Chapter 5). The new *MOQA* language is a standalone scripting language and provides automated average-case analysis. No other programming language of this scope (in particular, capable of handling general data structures) has been able to achieve this. The language has a clean and concise syntax, similar to a normal procedural programming language, powered using the original *MOQA* theory. The programmer defines manipulations over a *LPO* using *MOQA* basic operations. The analyzer analyses the running time using the notation of random bags and timing functions associated to each operation.

Because of the standalone language design, we can prevent most invalid *MOQA* programs both by syntax checking and by semantics checking. This not only improves the usability of our tools but also tidies up programs and makes code analysis easier.

The well studied interpreter architecture also avails us of a nice extension ability. We illustrated this feature by extending the analyzer to smoothed analysis in Section 7.4. Adding extra language constructs or operations also does not complicate matters due to clear pipeline in the interpreter, e.g. Lexer, Parser etc.

Our approach to automated average-case analysis differs from *Distrtrack* [48]. It benefits from directly accessing the abstract syntax tree (AST). By walking over the AST and abstractly executing statements in terms of random bags, the analyzer provides automated timing analysis. Because of this essential difference, our approach not only provides automated time analysis on recursive data structures, but also on general *MOQA* structures (see Section 5.7 for a timing analysis of the Heapify algorithm, which is not possible in *Distrtrack*).

Everything has two sides. Currently our analyzer provides the user instant feedback on a specific problem size, but there is no general equation output. Due to the abstract execution in the analyzer, currently it cannot handle large

problem sizes, e.g. 1 billion. It simply takes too much memory and large recursive call stacks. If we have a general equation, we might be able to approximate the answer. Because exact timing is not so important in very large problems, automated asymptotic analysis might suffice in this case, i.e., both approaches find a use in their respective contexts.

In this thesis we also developed a formal definition of the *MOQA* language. Both syntax and semantics are discussed. This specification could serve as a guideline to aid future *MOQA* language development and it would be interesting to see the language implemented with other programming languages. We discussed a Python implementation as a prototype for this language and verified the correctness of the theory (Chapter 4 and Chapter 5). It illustrates that *MOQA* research not only provides valuable theoretical results, but also exhibits potential to explore practical implications.

Besides these benefits, our interpreter also supports a graphical display of the final results and the *LPO* at various points in the execution. This was not supported in the previous approach.

The second part of the thesis focused on *MOQA* extensions, and several useful applications related to *MOQA* research were explored.

We started with *MOQA* parallel extensions. Modularity theory was extended to a fork-join model. A new way to analyse a multithreaded fork-join program under *MOQA* theory was presented. We showed that multithreaded algorithms which satisfy *MOQA* theory can be easily analysed. Parallel Quicksort served as a case study and justified our theory (Chapter 6).

Next, we examined several new heap creation algorithms. By studying their random bag preserving properties, Skew heap and Min-Max heap were found to fit the *MOQA* context well and could be candidates for investigation in future research.

We introduced an insertion operation to the *MOQA* treap data structure and its complexity was derived. We also showed that *MOQA* random bags provide an easy approach not only to capture average-case cost, but also to entropy analysis. We demonstrated this property by tracking entropy changes in several sorting algorithms. It has been proven in prior work [32, 87, 90], that there is a strong link between *MOQA* and reversible computing. With some additional

bookkeeping, *MOQA* operations can be designed reversible. We demonstrated this feature through a reversible *Split* operation and reversible Quicksort [33]. In the last part, we briefly discussed recent research result on smoothed analysis and presented how it can be integrated into our interpreter analyzer (Chapter 7).

8.2 Future Work

There are two possible directions that could be our future research, from an application perspective and a theoretical perspective. We discuss them in the following sections.

8.2.1 From An Application Perspective

The first possible improvement to the *MOQA* language is to add extra features that we have discussed theoretically in this thesis. There is potential to modify the *MOQA* language to incorporate parallel execution. The Go language is a great example that provides parallel execution by using keyword `Goroutine` in the language [8]. Like the Go language, it might be possible to add `fork` and `join` keywords in the *MOQA* language to enable fork-join execution in our interpreter at runtime. Using the theory we developed in this thesis, it would be interesting to obtain an automated analysis of fork-join *MOQA* programs' complexity and to derive the work and span boundary.

As shown in this thesis, data entropy is tracked naturally with the help of random bags (see Section 7.2.3). Inside the interpreter, these random bags are tracked to enable automated average-case analysis. With necessary modification, the interpreter should be able to handle entropy tracking automatically and output a data sheet or graph to illustrate the entropy changes for a *MOQA* program.

Another improvement to the *MOQA* language interpreter would be an animation of the random bags at various points in the code analysed. Currently, the interpreter only supports a graphical display of a *LPO*. It would be possible to display a random bag, for the analyzer keeps tracking these structures throughout

computation.

The future research could also investigate a more economical representation of random bags and structures in the interpreter analyzer to handle bigger problem sizes, or we might be able to develop an automated asymptotic analysis. For asymptotic analysis, the data we need to track might be reduced (e.g. only structure size is involved) and enable us a better performance in large problems. Of course, by setting a threshold, switching between extract analysis to asymptotic analysis would also be a smart approach.

MOQA builds upon data restructuring operations and algorithms. And these operations are generally focused on subsets of original structures (formally called isolated subsets, see Section 2.3). In functional programming, pattern matching is a widely used feature. It helps you decompose and navigate data structures in a very convenient, and compact syntax [20, 72]. The future work could take advantage of pattern matching and redesign *MOQA* into a functional language. The functional programming paradigm might in nature be more close to *MOQA* theory.

So far, *MOQA* theory takes comparisons as basic instructions. However, the theory could be further developed to lower level instructions, e.g. in terms of MIPS assembly. It would be interesting to see the redesign of our language and the analyzer to handle these low level performance predictions. Also, the language might need to change from interpreting to a compiled language. And as shown by other projects in the group, there is a connection between *MOQA* and compositionality in measuring power consumption [92, 115]. With a low level redesign, we might be able to bridge the gap between *MOQA* language and automated prediction of power consumption.

Finally, in Chapter 6, we demonstrated the extension of *MOQA* theory to a parallel field. Because the information obtained by *MOQA* theory has a great potential in other fields, future research could focus on using this information to fine tune parallel programs, e.g. by dynamically changing threshold, and by optimizing processor allocation in the system. There are some applications already discussed in [79], e.g. parallel Mergesort, under my guidance. Also we could experiment and spread *MOQA* theory to other platforms. E.g. the recent GPU computing frameworks: CUDA or OpenCL [54].

8.2.2 From A Theoretical Perspective

We have provided some theoretical contributions on *MOQA* in the thesis. e.g. *MOQA* language formal syntax and operational semantics, extensions to the parallel file, entropy tracking, new algorithms analysis etc. However, it would be interesting to continue these works to explore other possibilities and obtain more results. For example, we raised an open question in Section 7.1.4 (page 172), future research could dig further to the output structures of the *Min-Max heapify* and derive its complexity.

In particular, at the moment the initial input to all *MOQA* programs is considered to be a random list, though there is no inherent restriction in *MOQA*'s theory to do so. Future research could relax this restriction or allow the user to specify the input random bag. With this relaxation more interesting algorithms might be analysed with the *MOQA* approach.

Currently, *MOQA* only deals with input data which are uniformly distributed. It is worthwhile developing an extension of *MOQA* theory to handle other data distributions, e.g. the normal distribution. These data distributions are more widely used to model complex system in practice. Of course, smoothed complexity has been motivated by this consideration and we have been made explorations in this direction.

In terms of a parallel analysis based on *MOQA*, further investigating our method for the case of other fork-join programs and showing its applicability would be a worthwhile endeavour. The formalized extension of our method to other frameworks such as CUDA or OpenCL would be an interesting project.

Finally, regarding reversibility on *MOQA* theory, the first step in future work might be to explore a generalization to ensure frugal encodings for all of *MOQA* operations. The encoding could lift *MOQA* from a language underpinning static average-case analysis to a reversible language, capable of exact average-cost predictions. Follow on work could focus on extracting the benefits of both aspects, and, on exploring the interesting novel connection between guaranteeing a modular derivation of the average computation cost (a key requirement to develop static timing tools) and reversibility of language operations.

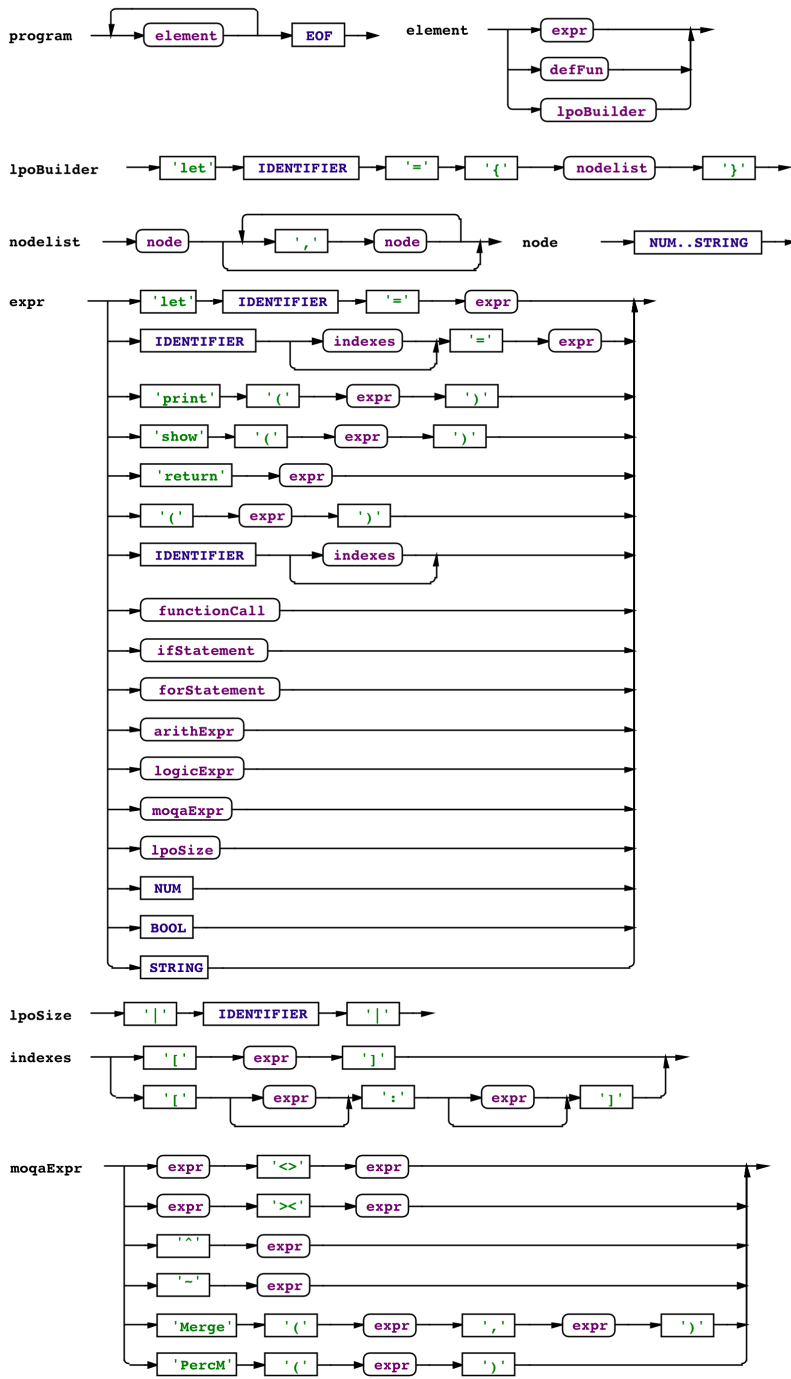
Appendix A

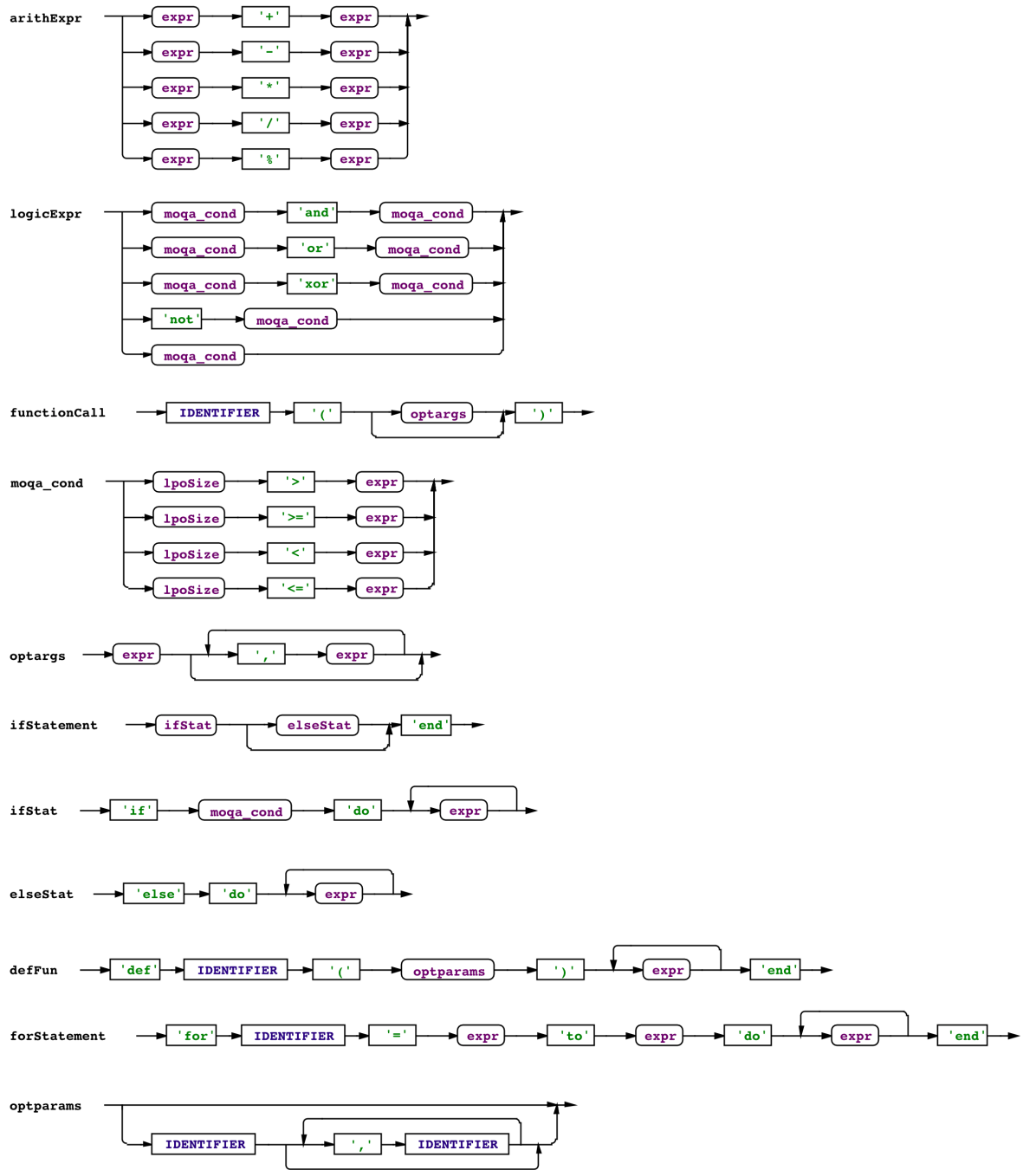
MOQA Language

Contents

A1	<i>MOQA</i> Language Syntax Specification	214
A2	<i>MOQA</i> Language Lexer	216
A3	<i>MOQA</i> Language Parser	219

A1 *MOQA* Language Syntax Specification





A2 *MOQA* Language Lexer

Listing A.1: *MOQA* Language Lexer (moqa_tokens.py)

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 # Copyright (C) 2012 by CEOL, Ang Gao <a.gao@cs.ucc.ie>
5 # All rights reserved.
6
7 """
8 This is a set of regular expressions defining a lexer
9 for MOQA fragments.
10 """
11
12 import ply.lex as lex
13
14 tokens = (
15     'AND',          # and
16     'COMMA',       # ,
17     'DIVIDE',      # /
18     'ELSE',        # else
19     'ASSIGN',      # =
20     'EQUALEQUAL', # ==
21     'FALSE',      # false
22     'DEF',        # def
23     'GE',         # >=
24     'GT',         # >
25     'IDENTIFIER', # eg: factorial
26     'IF',         # if
27     'FOR',        # for
28     'DO',         # do
29     'LE',         # <=
30     'LPAREN',    # (
31     'LT',        # <
32     'MINUS',     # -
33     'MOD',       # %
34     'NOT',       # !
35     'NUMBER',    # eg: 1234 5.678
36     'OR',        # or
37     'PLUS',     # +
38     'TO',       # to
39     'DOWNTO',   # downto
40     'END',      # end
41     'RETURN',   # return
42     'RPAREN',   # )
```

```

43 'STRING',      # "this is a \"tricky\" string"
44 'TIMES',      # *
45 'TRUE',       # true
46 'LET',        # let
47 'PRINT',      # print
48 'SHOW',       # show
49 'BAR',        # |
50 'LBRACE',     # {
51 'RBRACE',     # }
52 'SPLIT',      # <>
53 'PRODUCT',   # <>
54 'TOP',        # ^
55 'BOT',        # ~
56 'LBRACK',    # [
57 'RBRACK',    # ]
58 'COLON',     # :
59 'XOR',        # xor
60 'MERGE',     # merge
61 'PERCM',     # percM
62 )
63
64 states = (
65 ('comment', 'exclusive'), # /* ... */
66 )
67
68 def t_comment(t):
69     r'\/*'
70     t.lexer.begin('comment')
71
72 def t_comment_end(t):
73     r'\*/'
74     t.lexer.lineno += t.value.count('\n')
75     t.lexer.begin('INITIAL')
76     pass
77
78 def t_comment_error(t):
79     t.lexer.skip(1)
80
81 def t_eolcomment(t):
82     r'//.*'
83     pass
84
85 reserved = [ 'def', 'if', 'let', 'return', 'for', 'else', 'true', \
86             'false', 'print', 'show', 'end', 'do', 'and', 'or', 'to', \
87             'xor', 'Merge', 'downto', 'PercM']
88
89 def t_IDENTIFIER(t):
90     r'[A-Za-z][0-9A-Za-z_]*'
91     if t.value in reserved:
92         t.type = t.value.upper()

```

```

93     return t
94
95 def t_NUMBER(t):
96     r'--[0-9]+(\.[0-9]*)?'
97     t.value = float(t.value)
98     return t
99
100 def t_STRING(t):
101     r'"([\\"\\]|(\\.))*"
102     t.value = t.value[1:-1] # strip off "
103     return t
104
105 t_COMMA = r','
106 t_DIVIDE = r'/'
107 t_EQUALEQUAL = r'=='
108 t_ASSIGN = r'='
109 t_LPAREN = r'\('
110 t_MINUS = r'-'
111 t_MOD = r'%'
112 t_NOT = r'!'
113 t_PLUS = r'\+'
114 t_RPAREN = r'\)'
115 t_TIMES = r'\*'
116 t_LE = r'<='
117 t_LT = r'<'
118 t_GT = r'>'
119 t_GE = r'>='
120 t_LBRACE = r'{'
121 t_RBRACE = r'}'
122 t_LBRACK = r'\['
123 t_RBRACK = r'\]'
124 t_BAR = r'\|'
125 t_SPLIT = r'><'
126 t_PRODUCT = r'<>'
127 t_TOP = r'^'
128 t_BOT = r'^'
129 t_COLON = r'\:'
130 t_ignore = ' \t\v\r'
131 t_comment_ignore = ' \t\v\r'
132
133 def t_newline(t):
134     r'\n'
135     t.lexer.lineno += 1
136
137 def t_error(t):
138     print "MOQA Lexer: Illegal character " + t.value[0]
139     t.lexer.skip(1)

```

A3 *MOQA* Language Parser

Listing A.2: *MOQA* Language Parser (moqa_grammar.py)

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 # Copyright (C) 2012 by CEOL, Ang Gao <a.gao@cs.ucc.ie>
5 # All rights reserved.
6
7 """
8 This is a grammar definition file for MOQA language.
9 """
10
11 import ply.lex as lex
12 import ply.yacc as yacc
13 from moqa_tokens import tokens
14 import moqa_ast
15
16 start = 'program'
17
18 precedence = (
19     ('left', 'OR', 'XOR'),
20     ('left', 'AND'),
21     ('left', 'EQUALEQUAL'),
22     ('left', 'LT', 'LE', 'GT', 'GE'),
23     ('left', 'PLUS', 'MINUS'),
24     ('left', 'TIMES', 'DIVIDE', 'MOD'),
25     ('right', 'NOT'),
26 )
27
28 # program -> element program
29 def p_module_list(p):
30     'program : element program'
31     p[0] = moqa_ast.ModuleNode([p[1]] + p[2].children)
32
33 # program -> element
34 def p_module_empty(p):
35     'program : element'
36     p[0] = moqa_ast.ModuleNode([p[1]])
37
38 # element -> expr | defFun
39 def p_element(p):
40     '''
41     element : expr
42             | defFun
```

```

43         | lpoBuilder
44     '''
45     p[0] = moqa_ast.ElementNode(p[1])
46
47 # lpoBuilder -> let Identifier = { nodelist }
48 def p_element_lpoBuilder(p):
49     'lpoBuilder : LET IDENTIFIER ASSIGN LBRACE nodelist RBRACE'
50     idf = moqa_ast.IdentifierNode(p[2])
51     lpo = moqa_ast.LPONode(p[5])
52     p[0] = moqa_ast.LetExprNode([idf, lpo])
53
54 # expr -> let Identifier = expr ;
55 def p_expr_let(p):
56     'expr : LET IDENTIFIER ASSIGN expr'
57     idf = moqa_ast.IdentifierNode(p[2])
58     p[0] = moqa_ast.LetExprNode([idf, p[4]])
59
60 # expr -> Identifier = expr ;
61 def p_expr_idf(p):
62     'expr : IDENTIFIER ASSIGN expr'
63     idf = moqa_ast.IdentifierNode(p[1])
64     p[0] = moqa_ast.AssignExprNode([idf, p[3]])
65
66 # expr -> Identifier indexes = expr ;
67 def p_expr_idf_idx(p):
68     'expr : IDENTIFIER indexes ASSIGN expr'
69     idf = moqa_ast.IdentifierNode(p[1])
70     slice = moqa_ast.SliceNode([idf, p[2]])
71     p[0] = moqa_ast.AssignExprNode([slice, p[4]])
72
73 # expr -> return expr ;
74 def p_expr_return(p):
75     'expr : RETURN expr'
76     p[0] = moqa_ast.ReturnExprNode(p[2])
77
78 # expr -> Identifier ( optargs )
79 def p_expr_funcall(p):
80     'expr : IDENTIFIER LPAREN optargs RPAREN'
81     p[0] = moqa_ast.FuncUseNode(p[1], p[3])
82
83 # expr -> Print ( optargs )
84 def p_expr_print(p):
85     'expr : PRINT LPAREN optargs RPAREN'
86     p[0] = moqa_ast.FuncUseNode('print', p[3])
87
88 # expr -> Show ( optargs )
89 def p_expr_show(p):
90     'expr : SHOW LPAREN optargs RPAREN'
91     p[0] = moqa_ast.FuncUseNode('show', p[3])
92

```

```

93 # expr -> lpoSize
94 def p_expr_lposize(p):
95     'expr : lpoSize'
96     p[0] = p[1]
97
98 # expr -> | Identifier |
99 def p_lpoSize(p):
100     'lpoSize : BAR IDENTIFIER BAR'
101     idf = moqa_ast.IdentifierNode(p[2])
102     p[0] = moqa_ast.FuncUseNode('size', idf)
103
104 # expr -> ifExpr
105 def p_expr_ifStatement(p):
106     'expr : ifExpr'
107     p[0] = p[1]
108
109 def p_expr_forStatement(p):
110     'expr : forStatement'
111     p[0] = p[1]
112
113 # expr -> Bool
114 def p_expr_bool(p):
115     'expr : TRUE'
116     p[0] = moqa_ast.BoolNode(True)
117
118 def p_expr_bool_f(p):
119     'expr : FALSE'
120     p[0] = moqa_ast.BoolNode(False)
121
122 # expr -> logicExpr
123 def p_expr_logicExpr(p):
124     'expr : logicExpr'
125     p[0] = p[1]
126
127 # logicExpr -> moqa_cond op moqa_cond
128 def p_logicExpr(p):
129     '''
130     logicExpr : moqa_cond AND moqa_cond
131                | moqa_cond OR moqa_cond
132                | moqa_cond XOR moqa_cond
133     '''
134     p[0] = moqa_ast.OpNode(p[2], [p[1],p[3]])
135
136 # logicExpr -> not moqa_cond
137 def p_logicExpr_not(p):
138     'logicExpr : NOT moqa_cond'
139     p[0] = moqa_ast.OpNode('NOT', [p[2]])
140
141 # expr -> moqa_expr
142 def p_expr_moqa(p):

```

```

143     'expr : moqaExpr'
144     p[0] = p[1]
145
146 # moqaExpr -> expr op expr
147 def p_moqa_expr_binop(p):
148     '''
149     moqaExpr : expr SPLIT expr
150               | expr PRODUCT expr
151     '''
152     p[0] = moqa_ast.MoqaOpNode(p[2], [p[1], p[3]])
153
154 # moqaExpr -> Merge(expr, expr)
155 def p_moqa_expr_binop_merge(p):
156     '''
157     moqaExpr : MERGE LPAREN expr COMMA expr RPAREN
158     '''
159     p[0] = moqa_ast.MoqaOpNode(p[1], [p[3], p[5]])
160
161
162 # moqaExpr -> PercM(expr)
163 def p_moqa_expr_unaryop_percm(p):
164     '''
165     moqaExpr : PERCM LPAREN expr RPAREN
166     '''
167     p[0] = moqa_ast.MoqaOpNode(p[1], [p[3]])
168
169 # moqaExpr -> op expr
170 def p_moqa_expr_unaryop(p):
171     '''
172     moqaExpr : TOP expr
173               | BOT expr
174     '''
175     p[0] = moqa_ast.MoqaOpNode(p[1], [p[2]])
176
177 # expr -> Identifier
178 def p_expr_idenf(p):
179     'expr : IDENTIFIER'
180     p[0] = moqa_ast.IdentifierNode(p[1])
181
182 # expr -> IDENTIFIER indexes
183 def p_expr_idenf_idx(p):
184     'expr : IDENTIFIER indexes'
185     idf = moqa_ast.IdentifierNode(p[1])
186     p[0] = moqa_ast.SliceNode([idf, p[2]])
187
188 # indexes -> ( expr )
189 def p_indexes_simple(p):
190     'indexes : LBRACK expr RBRACK'
191     p[0] = moqa_ast.IndexNode([p[2]])
192

```

```

193 # indexes -> (expr : expr)
194 def p_indexes_slice(p):
195     'indexes : LBRACK expr COLON expr RBRACK'
196     p[0] = moqa_ast.IndexNode([p[2], p[4]])
197
198 # indexes -> (expr :)
199 def p_indexes_slice_lower(p):
200     'indexes : LBRACK expr COLON RBRACK'
201     p[0] = moqa_ast.IndexNode([p[2], moqa_ast.NumNode(-1)])
202
203 # indexes -> (: expr)
204 def p_indexes_slice_upper(p):
205     'indexes : LBRACK COLON expr RBRACK'
206     p[0] = moqa_ast.IndexNode([moqa_ast.NumNode(0), p[3]])
207
208 # nodelist -> node (, nodelist)*
209 def p_nodelist_one(p):
210     'nodelist : node'
211     p[0] = [p[1]]
212
213 def p_nodelist(p):
214     'nodelist : node COMMA nodelist'
215     p[0] = [p[1]] + p[3]
216
217 def p_node(p):
218     '''node : NUMBER
219           | STRING
220     '''
221     p[0] = p[1]
222
223 # optargs -> args
224 def p_optargs(p):
225     'optargs : args'
226     p[0] = p[1]
227
228 # optargs ->
229 def p_optargs_empty(p):
230     'optargs : '
231     p[0] = []
232
233 # args -> expr, args
234 def p_args(p):
235     'args : expr COMMA args'
236     p[0] = [p[1]] + p[3]
237
238 # args -> expr
239 def p_args_one(p):
240     'args : expr'
241     p[0] = [p[1]]
242

```

```

243 # ifExpr -> ifStat optElseStat End
244 def p_expr_ifExpr(p):
245     'ifExpr : ifStat optElseStat END'
246     p[0] = moqa_ast.IfNode(p[1].children + [p[2]])
247
248 # optElseStat ->
249 def p_optElseStat_empty(p):
250     'optElseStat : '
251     p[0] = []
252
253 # optElseStat -> exprList
254 def p_optElseStat_exprs(p):
255     'optElseStat : ELSE exprlist'
256     p[0] = p[2]
257
258 # ifStat -> If moqa_cond Do exprlist
259 def p_ifExpr_ifStat(p):
260     'ifStat : IF moqa_cond DO exprlist'
261     p[0] = moqa_ast.IfNode([p[2]]+[p[4]])
262
263 # exprlist -> expr
264 def p_exprlist_single(p):
265     'exprlist : expr'
266     p[0] = [p[1]]
267
268 # exprlist -> expr exprlist
269 def p_exprlist(p):
270     'exprlist : expr exprlist'
271     p[0] = [p[1]] + p[2]
272
273 # moqa_cond -> |Identifier| op arithExpr
274 def p_moqa_cond(p):
275     '''moqa_cond : lpoSize LT expr
276                 | lpoSize GT expr
277                 | lpoSize GE expr
278                 | lpoSize LE expr
279                 | lpoSize EQUALEQUAL expr
280     '''
281     p[0] = moqa_ast.OpNode(p[2], [p[1],p[3]])
282
283 # expr -> ( expr )
284 def p_expr_paren(p):
285     'expr : LPAREN expr RPAREN'
286     p[0] = p[2]
287
288 # arithExpr -> arithExpr op arithExpr
289 def p_arithExpr(p):
290     '''
291     arithExpr : expr PLUS expr
292               | expr MINUS expr

```

```

293         | expr TIMES  expr
294         | expr DIVIDE expr
295         | expr MOD    expr
296     '''
297     p[0] = moqa_ast.OpNode(p[2], [p[1],p[3]])
298
299 # expr -> NUMBER
300 def p_expr_number(p):
301     'expr : NUMBER'
302     p[0] = moqa_ast.NumNode(p[1])
303
304 # expr -> arithExpr
305 def p_expr_arithExpr(p):
306     'expr : arithExpr'
307     p[0] = p[1]
308
309 # forStatement -> For IDENTIFIER = expr to expr Do expr+ End
310 def p_forStatement_to(p):
311     'forStatement : FOR IDENTIFIER ASSIGN expr TO expr DO exprlist END'
312     idf = moqa_ast.IdentifierNode(p[2])
313     p[0] = moqa_ast.ForNode([idf, p[4], p[6]]+ p[8])
314
315 # forStatement -> For IDENTIFIER = expr downto expr Do expr+ End
316 def p_forStatement_downto(p):
317     'forStatement : FOR IDENTIFIER ASSIGN expr DOWNTO expr DO exprlist END'
318     idf = moqa_ast.IdentifierNode(p[2])
319     p[0] = moqa_ast.DownForNode([idf, p[4], p[6]]+ p[8])
320
321 # defFun -> Def IDENTIFIER ( optparams ) expr+ end
322 def p_defFun(p):
323     'defFun : DEF IDENTIFIER LPAREN optparams RPAREN exprlist END'
324     p[0] = moqa_ast.FuncDefNode(p[2], p[4], p[6])
325
326 # optparams -> IDENTIFIER (, IDENTIFIER)*
327 def p_optparams(p):
328     'optparams : params'
329     p[0] = p[1]
330
331 def p_optparams_empty(p):
332     'optparams : '
333     p[0] = [ ]
334
335 def p_params(p):
336     'params : IDENTIFIER COMMA params'
337     p[0] = [p[1]] + p[3]
338
339 def p_params_one(p):
340     'params : IDENTIFIER'
341     p[0] = [p[1]]
342
343 def p_error(p):
344     raise SyntaxError

```

Listing A.3: *MOQA* interpreter environment implementation

```
1 # env = (parent_env, {})
2 def env_lookup(name, env):
3     if name in env[1]:
4         return (env[1])[name]
5     elif env[0] == None:
6         return None
7     else:
8         return env_lookup(name, env[0])
9
10 def env_update(name, value, env):
11     if name in env[1]:
12         (env[1])[name] = value
13     elif not (env[0] == None):
14         env_update(name, value, env[0])
15     raise RuntimeError(str(name) + ' is not defined')
```

Appendix B

Contents

B1	Approximation Timing for the Merge Operation . . .	228
----	--	-----

B1 Approximation Timing for the Merge Operation

n	n+1	Experimentation	Approximation	Difference
1	2	1.6667	1.6000	0.0667
2	3	3.5	3.4286	0.0714
3	4	5.4	5.3333	0.0667
4	5	7.3333	7.2727	0.0606
5	6	9.2857	9.2308	0.0549
6	7	11.25	11.2000	0.0500
7	8	13.2222	13.1765	0.0457
8	9	15.2	15.1579	0.0421
9	10	17.1818	17.1429	0.0389
10	11	19.1667	19.1304	0.0363
11	12	21.1538	21.1200	0.0338
12	13	23.1429	23.1111	0.0318
13	14	25.1333	25.1034	0.0299
14	15	27.125	27.0968	0.0282
15	16	29.0106	29.0909	0.0803
16	17	31.0718	31.0857	0.0139

Table B.1: Times for merging two lists of length n and $n + 1$

As proven in [73], merging two lists of same length has the following equation:

$$\overline{T}_{Merge}(R(\Delta_n), R(\Delta_n)) = \frac{2n^2}{n + 1}$$

But there is no general solution to express the average time of merging two lists with different lengths as discussed in Section 5.4 (page 136). We approximate the merging time for two lists of length n and $n + 1$ with the following equation:

$$\overline{T}_{Merge}(R(\Delta_n), R(\Delta_{n+1})) = \frac{2n(n + 1)}{(n + (n + 1))/2 + 1}$$

This equation mimics the original merge timing function. The numerator is twice the product of two lists' length, the denominator is the average length of two lists plus one. We justify the usefulness of the equation by experiments.

In our experiment, we count the exact average-case times for merging lists with lengths ranging from 1 to 11, while for the bigger problem size, we sample 1,000,000 cases and obtain their average times. The final result is shown in Table B.1. The difference between the experimentation and our approximation is small.

References

- [1] aiT WCET Analyzers. <http://www.absint.com/ait/>, 2012.
- [2] GCC New Parser. http://gcc.gnu.org/wiki/New_C_Parser, 2012.
- [3] INRIA Algotib: The Algorithms Project's Library and Other Packages of the Algorithms Project. <http://algo.inria.fr/libraries/>, 2012.
- [4] PLY (Python Lex-Yacc). <http://www.dabeaz.com/ply/>, 2012.
- [5] The python profilers. <http://docs.python.org/2/library/profile.html>, 2012.
- [6] Ruby Interpreter. <https://github.com/ruby/ruby/blob/trunk/parse.y>, 2012.
- [7] Yourkit profiler. <http://www.yourkit.com/>, 2012.
- [8] The Go Programming Language. <http://golang.org/>, 2012.
- [9] Graphviz - Graph Visualization Software. <http://www.graphviz.org/>, 2012.
- [10] pydot (Python interface to Graphviz's Dot language). <http://code.google.com/p/pydot/>, 2012.
- [11] Global interpreter lock. <http://wiki.python.org/moin/GlobalInterpreterLock>, 2012.
- [12] Descriptions of Python uses. <http://www.python.org/about/quotes/>, 2012.

REFERENCES

- [13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006.
- [14] Matthieu Amiguet. Teaching compilers with python, PyCon US 2010. <http://pyvideo.org/video/241/>.
- [15] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press; 2nd edition, 2002.
- [16] M. D. Atkinson, Jörg-Rüdiger Sack, Nicola Santoro, and Thomas Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29(10):996–1000, 1986.
- [17] Cyril Banderier, René Beier, and Kurt Mehlhorn. Smoothed Analysis of Three Combinatorial Problems. In *Mathematical Foundations of Computer Science*, pages 198–207, 2003.
- [18] Iain Bate, Guillem Bernat, and Peter P. Puschner. Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis. In *Object-Oriented Real-Time Distributed Computing*, pages 83–90, 2002.
- [19] C. H. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17:525–532, 1973.
- [20] Richard S. Bird and Philip Wadler. *Introduction to functional programming*. Prentice Hall, 1988.
- [21] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39: 85–97, March 1996. ISSN 0001-0782.
- [22] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.
- [23] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, April 1974. ISSN 0004-5411.
- [24] Magnus Broberg, Lars Lundberg, and Håkan Grahn. Vppb - a visualization and performance prediction tool for multithreaded solaris programs. In *in*

- Proc. 12th International Parallel Processing Symposium*, pages 770–776, 1998.
- [25] G. Christakos. Bayesian maximum entropy bme. In *Advanced Mapping of Environmental Data*, pages 247–306. Advanced Mapping of Environmental Data, ISTE, 2010.
- [26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.
- [27] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.
- [28] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.
- [29] B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge Univ Pr, 2002.
- [30] Edsger W. Dijkstra. Smoothsort, an Alternative for Sorting In Situ. *Science of Computer Programming*, 1:223–233, 1982.
- [31] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38:408–423, March 1989. ISSN 0018-9340.
- [32] Diarmuid Early. *A Mathematical Analysis of the MOQA Language*. PhD thesis, University College Cork, 2010.
- [33] Diarmuid Early, Ang Gao, and Michel Schellekens. Frugal encoding in reversible MOQA a case study for quicksort. In *4th Workshop on Reversible Computation*, 2012.

- [34] Philippe Flajolet and Robert Sedgewick. The average case analysis of algorithms. *Theoretical Computer Science*, 79:2376–1994, 1993.
- [35] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 79(1):37 – 109, 1991.
- [36] The Apache Software Foundation. Commons-math: The apache commons mathematics library. <http://commons.apache.org/math/>.
- [37] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.
- [38] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI '98*, pages 212–223, New York, NY, USA, 1998. ACM.
- [39] Ang Gao, Kenneth Rea, and Michel Schellekens. Static average case analysis fork-join framework programs based on *MOQA* method. In *Sixth International Symposium on Parallel Computing in Electrical Engineering (PARELEC 2011), 4-5 April 2011, Luton, United Kingdom*, pages 1–6. IEEE Computer Society, 2011.
- [40] Ang Gao, Aoife Hennessy, and Michel Schellekens. *MOQA* min-max heapify: A randomness preserving algorithm. In *10th International Conference Of Numerical Analysis And Applied Mathematics*, 2012.
- [41] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1995.
- [42] R. L. Graham and R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.
- [43] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. 1994.
- [44] Ralph E. Griswold, J. F. Poage, and I. P. Polonsky. *The Snobol 4 Programming Language*. Prentice Hall, 1971.

REFERENCES

- [45] Bernd Grobauer. Cost Recurrences for DML Programs. *Sigplan Notices*, 36:253–264, 2001.
- [46] Carl A Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.
- [47] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.
- [48] Dave Hickey. *Tracking data structures for automated average time analysis*. PhD thesis, University College Cork, 2008.
- [49] Timothy J. Hickey and Jacques Cohen. Automating program analysis. *Journal of The ACM*, 35:185–220, 1988.
- [50] Glenn H. Holloway and Judy A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, 5:402–417, 1979.
- [51] Glenn H. Holloway and Judy A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, 5:402–417, 1979.
- [52] Paul B. Kantor and Jung Jin Lee. The maximum entropy principle in information retrieval. In *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '86, pages 269–274, New York, NY, USA, 1986. ACM.
- [53] Rainer Kemp. *Fundamentals of the Average Case Analysis of Particular Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 99th edition, 1985.
- [54] David Kirk and Wen mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann (Elsevier), 2010.

-
- [55] Raimund Kirner and Peter Puschner. Supporting Control-Flow-Dependent Execution Times on WCET Calculation. In *Deutschsprachige WCET-Tagung, Germany*, Oct. 2000.
- [56] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Euromicro Conference on Real-Time Systems*, pages 31–40, 2002.
- [57] Donald E. Knuth. *The Art of Computer Programming, Volumes 3*. Addison-Wesley Professional, 2011.
- [58] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schutz, A. Vrchoticky, and R. Zainlinger. Real-Time System Development: The Programming Model of MARS. In *International Symposium on Autonomous Decentralized Systems*, pages 290–299, 1993.
- [59] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.
- [60] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 44:261–269, 2000.
- [61] John Fiskio Lasseter. Toolkits for the Automatic Construction of Data Flow Analyzers. *Technical report, Computer & Information Science Dept, University of Oregon*, 2005.
- [62] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande, JAVA '00*, pages 36–43, New York, NY, USA, 2000. ACM.
- [63] Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison Wesley, 2nd edition, 2006.
- [64] Hosam M. Mahmoud. *Sorting: A Distribution Theory*. Wiley-Interscience, 2000.

REFERENCES

- [65] Daniel Le Métayer. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10:248–266, 1988.
- [66] Marni Mishna. Attribute grammars and automatic complexity analysis. *Advances in Applied Mathematics*, 30:189–207, 2003.
- [67] D. Mittermair and P. Puschner. Which sorting algorithms to choose for hard real-time applications. In *Euromicro Conference on Real-Time Systems*, pages 250–257, 1997.
- [68] M. Mitzenmacher and E. Upfal. *Probability and computing: randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
- [69] Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
- [70] R.H. Möhring. Computationally tractable classes of ordered sets. *Algorithms and order*, 255:105–194, 1989.
- [71] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
- [72] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
- [73] M. O’keeffe and Michel Schellekens. Average merge time: an intuitive interpretation. *Electronic Notes in Theoretical Computer Science*, 74, 2002.
- [74] Oracle. Jdk 7 features. <http://openjdk.java.net/projects/jdk7/features/>.
- [75] Terence Parr. *The Definitive Antlr Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.
- [76] Peter P. Puschner and Alan Burns. Writing Temporally Predictable Code. In *Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–94, 2002.

-
- [77] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 28(2-3):179–215, 1996.
- [78] Lyle Harold Ramshaw. *Formalizing the analysis of algorithms*. PhD thesis, Stanford, CA, USA, 1979. AAI8001994.
- [79] Kenneth J Rea. Explorations on parallel programming and extensions of the \mathcal{MOQA} theory. Master’s thesis, University College Cork, 2012.
- [80] John H. Reif and Harry R. Lewis. Efficient Symbolic Analysis of Programs. *Journal of Computer and System Sciences*, 32:280–314, 1986.
- [81] Brian Reistad and David K. Gifford. Static Dependent Costs for Estimating Execution Time. *ACM Sigplan Lisp Pointers*, VII:65–78, 1994.
- [82] Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. *Static Program Analysis via 3Valued Logic*. 2002.
- [83] Mads Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and Computer Architecture*, pages 144–156, 1989.
- [84] Barbara G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 5:216–226, 1979.
- [85] Michel Schellekens. *A Modular Calculus for the Average Cost of Data Structuring: Efficiency-Oriented Programming in \mathcal{MOQA}* . Springer, 2008.
- [86] Michel Schellekens. A random bag preserving product operation. *Electron. Notes Theor. Comput. Sci.*, 225:341–360, January 2009. ISSN 1571-0661.
- [87] Michel Schellekens. \mathcal{MOQA} unlocking the potential of compositional static average-case analysis. *Journal of Logic and Algebraic Programming*, 79(1): 61 – 83, 2010. ISSN 1567-8326. Special Issue: Logic, Computability and Topology in Computer Science: A New Perspective for Old Disciplines.
- [88] Michel Schellekens and Diarmuid Early. Running time of the treapsort algorithm. *Journal of Theoretical Computer Science*, 2012.

- [89] Michel Schellekens and Aoife Hennessy. *MOQA* smoothed complexity. 2012. Preprint.
- [90] Michel Schellekens, Diarmuid Early, Emanuel Popovici, and Dilip Vasudevan. A high level reversible language for modular average-case analysis. In *preliminary proceedings of the Reversible Computing Workshop-RC2009, a satellite workshop of ETAPS 2009*.
- [91] Michel Schellekens, David Hickey, and Greg Bollella. ACETT , a Linearly-Compositional Programming Language for (semi-)automated Average-Case analysis. In *WIP Proceedings, 25th IEEE International Real-Time Systems Symposium*, 2004.
- [92] Michel Schellekens, Diarmuid Early, Dilip Vasudevan, Jiaoyan Chen, Ang Gao, and Emanuel Popovici. *MOQA* logic units for modular power analysis. 2012. Preprint.
- [93] Keith Schwarz. Demystifying smoothsort. ACM Tech Talk, Stanford University, 2011.
- [94] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Professional, 1995.
- [95] Robert Sedgewick and Kevin Wayne. *Algorithms (4th Edition)*. Addison-Wesley Professional, 2011.
- [96] Erik Yu shing Hu, Guillem Bernat, and Andy J. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. In *Workshop on Object-Oriented Real-Time Dependable Systems*, pages 77–84, 2002.
- [97] Nishant Sinha. Symbolic Program Analysis Using Term Rewriting and Generalization. In *Formal Methods in Computer-Aided Design*, pages 1–9, 2008.
- [98] Michael Sipser. *Introduction to the Theory of Computation*. Brooks/Cole; 2nd Revised edition edition, 2005.

REFERENCES

- [99] Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Heaps. *Siam Journal on Computing*, 15:52–69, 1986.
- [100] Daniel A. Spielman and Shang hua Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. In *Journal of the ACM*, pages 296–305, 2001.
- [101] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, May 2004. ISSN 0004-5411.
- [102] Richard P. Stanley. *Enumerative Combinatorics, Volume 2*. Cambridge University Press, 1999.
- [103] Andrew Stone, Michelle Strout, and Shweta Behere. May/must analysis and the DFAgen data-flow analysis generator. *Information & Software Technology*, 51:1440–1453, 2009.
- [104] D. Sundararajan. *The Discrete Fourier Transform: Theory, Algorithms and Applications*. World Scientific Publishing Co Pte Ltd, 2011.
- [105] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '09, pages 229–240, New York, NY, USA, 2009. ACM.
- [106] Tommaso Toffoli. Reversible computing. In *International Colloquium on Automata, Languages and Programming*, pages 632–644, 1980.
- [107] Jacinta Townley, Joseph Manning, and Michel Schellekens. Sorting Algorithms in *MOQA*. *Electronic Notes in Theoretical Computer Science*, 225: 391–404, 2009.
- [108] Alan Mathison Turing. Rounding-off Errors in Matrix Processes. *Quarterly Journal of Mechanics and Applied Mathematics*, 1:287–308, 1948.

- [109] Jeffrey Scott Vitter and Philippe Flajolet. Average-Case Analysis of Algorithms and Data Structures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier, 1990.
- [110] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229 – 239, 1980.
- [111] Reinhard Wilhelm. Timing Analysis and Timing Predictability. In *Formal Methods for Components and Objects*, pages 317–323, 2004.
- [112] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7:1–53, 2008.
- [113] Glynn Winskel. *Formal Semantics of Programming Languages*. The MIT Press, 1993.
- [114] Wolfgang Wogerer. A Survey of Static Program Analysis Techniques. *Technische Universität Wien*, 2005.
- [115] Tingcong Ye, Dilip Vasudevan, Jiaoyan Chen, Emanuel Popovici, and Michel Schellekens. Static Average Case Power Estimation Technique for Block Ciphers. In *Euromicro Symposium on Digital Systems Design*, pages 689–696, 2010.
- [116] Tetsuo Yokoyama and Robert Glck. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 144–153, 2007.
- [117] P Zeitz. *The Art and Craft of Problem Solving*. 1999.