Michiel Schellekens

A Modular Calculus for the Average Cost of Data Structuring

Forewords by
Greg Bollella and Dana Scott

Springer

A Modular Calculus for the Average Cost of Data Structuring

A Modular Calculus for the Average Cost of Data Structuring

by

Michel Schellekens
University College Cork-National University of Ireland
Ireland

Springer

Michel Schellekens
University College Cork (UCC)
National University of Ireland, Cork
Department of Computer Science
Centre for Efficiency-Oriented Languages
Western Road
Cork, Ireland
Email: m.schellekens@cs.ucc.ie

Cover art:
Title of art work: This little light of mine

Tapestry artist: Pascale De Coninck
www.pascaledeconinck.com

Photographer: Dori O'Connell
www.dorioconnell.com

Tutorial CD:
CALVIN AND HOBBES ©1986 Watterson.
Dist. By UNIVERSAL PRESS SYNDICATE.
Reprinted with permission. All rights reserved.

ISBN-13: 978-0-387-73383-8 e-ISBN-13: 978-0-387-73384-5

Library of Congress Control Number: 2008925540

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

To my wife, Pascale De Coninck, and parents, Yvan and Yvonne Schellekens.

Foreword

As of June 2008 it will have been 50 years since the award of my Princeton Ph.D. During that academic year, aside from my research in symbolic logic, I worked on the von Neumann computer at the Institute for Advanced Study to program a small combinatorial puzzle. It took some collaboration, a little special coding, many hours of trial and error, and the use of punched cards to get the correct sequence of computations done on that machine. Today the problem is an exercise for undergraduate classes.

The IAS computer was the prototype of machines built by IBM that were expensive and very power hungry. They were as big as dinosaurs — and nearly as slow! The progress in computer architecture over that half century since 57/58 (and even over the last 25 years) has been truly astonishing. Now, for a fairly modest price, I have sitting here on my desk an Intel Core Duo laptop running at a processor speed of 2.16 GHz, with an L2 cache of 2 MB, an on-chip memory of 1 GB, and a bus speed of 667 MHz. The hard disk has a capacity of over 93 GB, and the external back-up disk can store nearly 300 GB. The laptop itself has wireless I can use at the coffee house down the street, a CD and DVD player, video output for lectures (or movies I can show at home), and an internet connection via a high-speed cable modem here on my desk. At the moment as I write I am listening to classical baroque music over internet radio, but I can get literally hundreds of sound connections for all kinds of music and talk radio. People would have killed for such personal computing power only 20 years ago. And as the clerk at the computer store reminded me last week, my laptop is even now somewhat outdated! By comparison the old IAS machine — now standing as a sad, dead fossil at the Smithsonian Institution Museum in Washington, D.C. — seems a rather small, baby dinosaur.

Let us also note that in less than a decade another dinosaur, the supercomputer, has had a remarkable reincarnation. Genetic research, cryptography, and astronomy — to name only a few areas — today would be all but unthinkable without the use of supercomputers. A widely circulated news report this week told us:

> A recent IBM research project that aims to replace electricity with pulses of light to make data transfer between processor cores up to 100 times faster could lead to laptop-sized supercomputers and drastically improved power consumption. The technology, called silicon

> nanophotonics, replaces electronic wires with pulses of light in optical fibers for faster and more efficient data transfers between cores on a chip.

Not only are the reports of the death of Moore's Law from 1965 much exaggerated, as the alternatives to the older silicon technology are coming forward, but the multi-core design will require a complete rethinking of algorithm development to take advantage of on-chip parallelism.

And here is another news report that just came to me this hour via an e-mail list:

> Intel has announced one of the smallest flash-memory drives that could give handheld devices the power of desktop computers. The chip will compete with similar chips from Samsung, which are used in gadgets such as Apple's iPod and iPhone, but Intel's chip comes with a built-in standard electronics controller, which makes it easy and inexpensive to combine multiple chips into a single, high-capacity hard drive. Since being introduced in the late 1990s, flash memory has revolutionized consumer electronics due to flash-memory chips being smaller, more durable, and more energy efficient than magnetic hard disks, making them the ideal replacement for hard drives in handheld devices such as MP3 players, mobile phones, and even some high-end laptops.

Clearly we live in very exciting (computer) times!

I do not know whether I will still be alive to see the practical realization of quantum computing, but I do know that I myself cannot even begin to take advantage of the computing power sitting right here before me today. Of course, experts are indeed putting the new machines through their paces and doing great new science; but I also believe we have very, very much to learn still about crafting efficient algorithms. No matter what new machine is produced, it is always possible to invent a nasty problem beyond its capacity in memory and/or speed. But that does not mean that the new facilities cannot produce excellent and improved results on older problems – if we know how to take advantage of the new architectures and improved capabilities.

This brings us to the question of algorithm analysis, the topic of the present monograph. Let us note first that worst-case analysis need not hold back good algorithm development. An early success was Linear Programming: though there can be very tough problems, current algorithms work very, very well in practical applications. A somewhat related example is the problem of solving Boolean equations. The general question is NP-complete, but recent implementations are having remarkable success. Here is a quotation from the Wikipedia entry for the "Boolean satisfiability problem":

> Modern SAT solvers (developed in the last ten years) come in two flavours: "conflict-driven" and "look-ahead". Conflict-driven solvers augment the basic DPLL search algorithm with efficient conflict analysis, clause learning, non-chronological backtracking (aka backjumping), as well as "two-watched-literals" unit propagation, adaptive branching, and random restarts. These "extras" to the basic systematic search have been empirically shown to be essential for handling the large SAT instances that arise in Electronic Design Automation (EDA). Look-ahead solvers have especially strengthened reductions (going beyond unit-clause propagation) and the heuristics, and they are generally stronger than conflict-driven solvers on hard instances (while conflict-driven solvers can be much better on large instances which have inside actually an easy instance).
>
> Modern SAT solvers are also having significant impact on the fields of software verification, constraint solving in artificial intelligence, and operations research, among others. Powerful solvers are readily available in the public domain, and are remarkably easy to use. In particular, MiniSAT, which was relatively successful at the 2005 SAT competition, only

has about 600 lines of code. Minisat is an example of a conflict driven solver, and an example for look-ahead solvers is march_dl, which won a prize at the 2007 SAT competition.

The point here, it seems to me, is that clever heuristics can have huge pay-offs. The rub is that the finding of good heuristics is an art: past success may yield clues and inspiration for attacking future problems, but continuing success is by no means assured.

In the book before us, Michel Schellekens reports on work by him and with his collaborators on a systematic method for doing the average-case analyses of a wide class of algorithms.

He explains how the new approach builds on traditional methods, while solving new problems in a new way. He makes a strong case for the effectiveness of the scientific and mathematical foundations introduced for giving greater analysis accuracy and re-use adaptability to the algorithm designer.

As the author outlines in the concluding Chapter 11, exploiting features assuring modularity always seems to be good design advice. The language $\mathcal{MOQA}$ offers both serious examples of how to achieve modularity as well as some interesting theoretical problems to be explored. The parallel facilities of the language also open up other possible areas for study which could impact both future software and hardware design.

The basics set out here so clearly should lead to many new investigations and results.

Berkeley, California, December 2007 *Dana S. Scott*

Foreword

Every human on the planet routinely, daily, grapples with estimating the relationship between best-case, average-case, and worst-case times for a myriad of issues. How long will the drive to the airport take? How long will I be in labor? How long will the monsoons last? When will I get through the checkout line? Will this professor ever stop talking? Typically, we humans deal with these estimates with aplomb, easily exploring the space of probabilities and consequences to come up with a daily schedule that is mostly right. However, some of us, maybe as a group, scientists and engineers, but certainly myself, often look too deep. It is of no end of annoyance to my spouse that I normally arrive at an airport very early. Typically, early enough to get a meal and get some work done. Traffic on Hwy 101 along the San Francisco Bay Peninsula, which I take to get to the San Francisco airport from my home in Palo Alto, is notorious for having a very wide distribution of transit times. And, not wanting to be rushed, I estimate near-worst-case and arrive, really, too early. What is the actual worst-cast travel time from Palo Alto to San Francisco? Well, if an earthquake struck nearby it could be days or weeks. So, my estimate of travel time doesn't really consider the actual worst-case but incorporates some level of probabilistic analysis to come up with a typical worst-case. My spouse, on the other hand, likely estimates toward best-case and really isn't too often too late. She just ignores more outliers than I do and things generally work out. Most humans, I assume, proceed similarly.

When we move into the realm of real-time scheduling, ignoring outliers is no longer possible. Definitive mechanisms have to be in place to accommodate all outcomes (except those of complete system failure). For systems in which computation must occur before a specified deadline the analysis must include a value for what is typically called worst-case execution time (WCET). The literature contains much work on WCET, how to measure it and how to constrain it, for every conceivable programming language, runtime, and environment. It is a very difficult problem. And, a miscalculation can cause system-wide failures.

In my role as specification lead of the expert group for Java Specification Request 001, The Real-Time Specification for Java (RTSJ), one of the most important areas, to me, was to provide the RTSJ platform with fundamental semantics and interfaces which would allow developers to manage WCET miscalculations with more ease

that in typical real-time development environments. The RTSJ includes a subsystem called, 'cost enforcement'. 'Cost' is the term used in real-time scheduling analysis to mean the amount of time a task needs to use the processor. WCET is often used as the value for cost. Cost enforcement in the RTSJ looks at the cost parameter not only as information for the real-time scheduling analysis but also as a contract with the system. The system will not allow a task to use more processor time than the value given in the cost parameter. If a task does attempt to exceed this value it is stopped. In systems without such a mechanism what may happen is that an errant task, or a task for which WCET was miscalculated, can cause a cascade of deadline misses often resulting in a system-wide failure. With cost enforcement the errant task is stopped and the system has the opportunity to dynamically correct the situation. It is my belief, and also of the JSR-001 expert group, that any modern, serious real-time system requires a mechanism like cost enforcement. The point I wish to make here is that because WCET is so difficult to predict yet crucial to correct system function designers must often go to great lengths to accommodate the inherent inaccuracies.

So we come to the work of Michel Schellekens, the subject of this book. I first interacted with Michel on the advice of the Director of University Relations at Sun Microsystems, Inc., where I am a Distinguished Engineer and lead the Real-Time Java effort. Michel's work immediately interested me because he was attempting to shed new light onto an area previously thought to be, by the best minds in computer science and computational theory, essentially opaque. Michel and I have kept in touch over the years. I strongly supported him with the founding of the Center for Efficiency Oriented Languages (CEOL), including a donation of the RTSJ platform on Sun servers, and attended the opening in Cork on November 10, 2003.

Michel's work attacks, head on, the problem: Is there a way to obtain a theoretical upper bound on the average time complexity of an algorithm given average inputs. This is not trivial. Algorithmic time complexity, although difficult, often yields concrete results for best and worst case situations. One inspects the algorithm and imagines an input set which will cause the algorithm to do the least or most work, respectively. However, even thinking about what is an average case set of inputs, how to identify such a set in a general way and to track such sets throughout the computation to derive an upper bound on average time complexity is truly amazing in scope.

The work in this book clearly shows that the full story on best-, average-, and worst-case execution time is not yet fully written. I fully expect Michel's work to move from the algorithmic domain into the execution domain and thence into useful commercial products. It's only a matter of time. When this happens it will be interesting to note that the computer scientists, computational theorists and computer system practitioners have taken over five decades to produce systems which deal with the relationship between best-, average-, and worst-case times as easily as my spouse does on every trip to the airport.

Palo Alto, California, November 2007 *Greg Bollella*

Preface

The Analysis of Algorithms is a core Computer Science area which provides information on the expected, i.e. the average-case, performance of algorithms. Such information is useful in a variety of applications, including power estimation and resource budgetting in a real-time context. The Analysis of Algorithms also provides fundamental insights in the design of efficient software. Hence, both from an applied and a theoretical perspective, the investigation of improved methods and tools for static average-case analysis is a worthwhile goal.

Average-Case Analysis involves a variety of techniques which, typically, do not allow for automation. Currently algorithms must be analyzed on a case-by-case basis and it is not feasible in general to statically derive the average number of basic steps carried out by an algorithm during its execution. Various bottle-neck problems have been high-lighted in the literature and some well-known algorithms escape analysis.

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 [Knu73, FS95, Ram96, Vui80]. As pointed out in [Vui80]:

> A progress in our understanding of these questions should drastically affect the way in which we discover and explain the fundamental algorithms, as catalogued by Knuth [Knu73] and Aho et al [AHU87].

The aim of this work is to present a new approach to the Average-Case Analysis of Algorithms, based on the novel notion of random bags and their preservation. The view presented here is that the notion of a random bag may serve as a unifying model for abstract data structures and their data distribution, while random bag preservation enables the constructive tracking of the distribution during computations. The approach inspired novel algorithms and considerably simplified their average-case analysis.

The work presents a modular calculus for static average-case analysis which drastically simplifies the analysis and opens up the way for novel explorations on static timing tools. Random bags also contribute a visual way to represent data and their distributions, which, in addition to facilitating average-case analysis, provides a useful teaching aid.

A parallel between the role of Static Analysis in Software Engineering and the role of Calculus in "real" Engineering may be helpful to illustrate the motivation behind the research. Engineering offers the capacity to analyse the strength of a construction, such as a bridge, by analyzing its blue prints, rather than subjecting it to heavy loads to test its limits. This approach should ideally find a natural parallel in Software Engineering via Static Program Analysis. Rather than executing a program on a large selection of inputs to experimentally derive information on its average-case behaviour, the goal is to derive this information statically via an analysis of the program's source code. Calculus supports the analysis of blue-prints in Engineering. Similarly, the aim of this work is to provide a foundation for a Calculus supporting Static Average-Case Analysis of a program's source code. This is a major challenge and our aim is not to provide an all-encompassing answer. Instead, we focus on the introduction of new advances in this area as a basis for a simplified and unified theory of average-case analysis and as a potential platform on which to build future improved modular static analysis tools.

A central aspect of the novel approach, which distinguishes it from prior approaches to Average-Case Analysis, is the use of randomness preservation to ensure the compositionality property, well-known from the Semantics and the Real-Time Language areas, in the context of the Analysis of Algorithms. Compositionality can rightfully be referred to as the "golden key" to static analysis, witnessed by its central role in static worst-case time analysis. A main theme of this work is that compositionality, combined with the capacity for tracking data distributions, unlocks a novel technique for modular average-case analysis. This approach provides the inspiration for the $\mathcal{MOQA}$[1] "language". The language essentially consists of a suite of random bag preserving data-structuring operations together with conditionals, for-loops and recursion and hence can be incorporated in *any* traditional programming language, importing all of its benefits in a familiar context[2].

A key feature of $\mathcal{MOQA}$ is that its operations have been purpose designed to ensure the capacity for a compositional static average-case analysis of $\mathcal{MOQA}$ code. The guaranteed compositionality property of $\mathcal{MOQA}$ programs brings a strong advantage for the programmer. The capacity to combine parts of code, where the average-time is simply the sum of the times of the parts, is a helpful advantage in static analysis. Moreover, re-use is a key factor in our approach: once the average time is determined for a piece of code, then this time will hold in any context. Hence it can be re-used and the timing impact is always the same. Compositionality also improves precision of static average-case analysis, supporting the determination of accurate estimates on the average number of basic operations of programs.

It is a main theme of the current work to introduce the new foundation for average-case analysis and to illustrate its applicability, as well as to motivate and specify the $\mathcal{MOQA}$ language and discuss its associated static average-case timing tool *Distri-Track*.

[1] MOdular Quantitative Analysis.

[2] $\mathcal{MOQA}$ is implemented at CEOL in Java 5.0 as $\mathcal{MOQA}$-Java.

The work is carried out at the intersection of several areas: Analysis of Algorithms and Random Structures, Semantics, Real-Time Languages, Static Program Analysis, Modular Design and the mathematical theories of Finite Partial Orders, Linear Extensions, Multi-Sets and Probability Theory. Hence the material may be useful for a variety of researchers and students, with interests in Computer Science, Electrical Engineering or Mathematics.

We provide an overview of the chapters in this work.

Chapter 1 provides an introduction to the new techniques for average-case analysis and focuses on a motivation of the central notions involved. This includes a motivation of compositionality as the "golden key" to static timing and the need for novel language design to reach compositionality, including the related concept of an Efficiency-Oriented Language.

The chapter provides a brief introduction to the $\mathcal{MOQA}$ language, for which static average-case timing can be achieved in a modular way through the tracking of distributions. Random bags are introduced as concise ways to capture data and their distribution and distribution tracking is incorporated via the concept of random bag preservation. The split operation, well-known from algorithms such as Quicksort and Quickselect, is provided as an example of a random bag preserving operation. This example also serves to illustrate the tracking of distributions in $\mathcal{MOQA}$ and the use of the notion of a separative function to establish random bag preservation.

The chapter also discusses the central Linear-Compositionality Theorem, which forms the basis for the static derivation of the average-case time of $\mathcal{MOQA}$ programs. Advantages of the $\mathcal{MOQA}$ approach are outlined and the chapter concludes with a discussion of the related area of bridging Semantics and Complexity and the area of Real-Time Languages.

Chapter 2 presents introductory notions, including partial orders, series-parallel orders, trees, heaps and bags. A brief overview of some basic sorting algorithms is provided as well as an introduction to standard timing measures, including exact time, total time, worst-case, best-case and average-case time.

Chapter 3 introduces the central notion of compositionality, including IO-compositionality. Worst-case time is shown to be semi-IO-compositional while average-case time is shown to be IO-compositional. The Average-Case Time Paradox is discussed in this context. This paradox regards the fact that even though average-case time is shown to have better compositionality properties than worst-case time, in practice the derivation of average-case time is known to be much more difficult than worst-case time. The paradox is shown to be linked to the potential lack of randomness preservation of standard algorithms, including well-known examples such as Bubblesort and Heapsort. Moreover, the chapter motivates how IO-compositionality of the average-case time measure can be used, in combination with randomness preservation, to obtain linear-compositionality. This greatly facilitates average-case time analysis and overcomes the Average-Case Time Paradox.

Chapter 4 revisits in a slightly more general context, the fundamental notions of random structures, random bags and their preservation, which have been introduced in Chapter 1. The State Theorem is presented which enables an interpretation of

states in random structures as "generalized permutations". Chapter 4 also introduces the central notion of an isolated subset. An isolated subset forms a subset of a partial order such that the restriction of the random structure over this partial order to the isolated subset is guaranteed to yield a new random structure. A simplified definition of an isolated subset is obtained for the case of series-parallel orders. The chapter concludes with the Extension Theorem, which demonstrates that it is sufficient to define random bag preserving operations locally on an isolated subset, where the extension of the operation to the entire random structure is obtained in a natural way.

Chapter 5 introduces the basic $\mathcal{MOQA}$ operations, including the Random Product, the Random Deletion and Percolation, the Random Projection, the Random Split and the Top and Bot operations. Each of these $\mathcal{MOQA}$ operations is shown to be random bag preserving. Deletion operations typically are not included in the context of automated average-case analysis, since the analysis of deletions with respect to average-case time is well-known to be problematic, even in the context of traditional average-case analysis. Hence the Random Deletion opens up the way for the inclusion of novel algorithms, such as Percolating Heapsort and Treapsort, which are analyzed in Chapter 9. The Extension Theorem of Chapter 4 is applied to extend these operations from local applications on isolated subsets to applications over the entire random structure. Uniformly random bag preserving operations are singled out as of particular interest, since this type of operations enables simplifications of probability computations in later chapters. The $\mathcal{MOQA}$ operations are shown to preserve series-parallel data structures which yields a characterization of the so-called $\mathcal{MOQA}$ atomic-constructible data structures as series-parallel orders. Finally, some simplifications for the series-parallel case are obtained in the context of the computation of cardinalities of random structures. Such simplifications for series-parallel orders will also be useful in the context of Chapter 6, which regards the average-case analysis of the basic $\mathcal{MOQA}$ operations.

Chapter 6, joint with D. Early, presents the detailed average-case analysis of the basic $\mathcal{MOQA}$ operations, resulting in the formulas obtained by D. Early. As shown in Chapter 7, $\mathcal{MOQA}$ programs are Linearly-Compositional with respect to the average-case time, i.e. their average-case time can be expressed as linear combinations of the average-case times of more basic components. Hence, ultimately, a successful average-case time derivation yields the average-case time of $\mathcal{MOQA}$ programs, expressed in terms of the average-case times of the basic $\mathcal{MOQA}$ operations. Formulas for the average-case times of basic $\mathcal{MOQA}$ operations are obtained in Chapter 6 and simplified formulas are derived for the case of series-parallel orders. These formulas are systematically applied in Chapter 9, which presents examples of compositional average-case time derivations of $\mathcal{MOQA}$ programs. Finally, the formulas of this chapter are illustrated via basic applications involving inductively defined data structures, such as linear orders and complete binary trees. Chapter 6 concludes with a demonstration of combinatorial identities used in the derivation of the average-case time formulas.

Chapter 7 provides the specifications for the $\mathcal{MOQA}$ language, with special attention given to conditionals and recursion, which typically form a challenge for static timing analysis. The random bag preservation of $\mathcal{MOQA}$ programs is demon-

strated and the method for the linear-compositional derivation of the average-time of $\mathcal{MOQA}$ programs is outlined.

Chapter 8 provides examples of well-known sorting and search algorithms implemented in $\mathcal{MOQA}$. It also includes examples of two novel algorithms, Percolating Heapsort, the first randomness preserving version of the Heapsort algorithm, and Treapsort, a sorting algorithm over treaps; both of which are essentially based on the Random Deletion operation of Chapter 5.

Chapter 9 provides the compositional average-case time derivation of the programs discussed in Chapter 8, with a main focus on illustrating the use of random bags in this context. The chapter in particular presents the first exact average-case time analysis of a heapsort variant via an analysis of Percolating Heapsort. Compositional average-case time derivations, whenever appropriate, rely on the formulas obtained in Chapter 6. The derivations obtained in this chapter illustrate the basic techniques involved in the static timing tool *Distri-Track*.

Chapter 10, joint with D. Hickey and M. Boubekeur, discusses in more detail the static timing tool *Distri-Track*, developed by D. Hickey. *Distri-Track* analyses $\mathcal{MOQA}$ algorithms programmed in Java, using an implementation of $\mathcal{MOQA}$ by J. Townley. *Distri-Track* enables the automated static derivation of average-case time of most of the $\mathcal{MOQA}$ programs presented in Chapter 8. Experiments, including comparisons with time derivations relying on a Java profiler, are discussed, as well as potential implications for Real-Time Languages. Finally, Chapter 11 presents the conclusion and some potential future work.

The book is accompanied by a software tutorial "Static average-case analysis of programs: a beginner's guide to successful tracking". The tutorial requires Adobe Flash Player, which is freely available online at http://www.adobe.com/. The tutorial provides an introduction to the main concepts used in this work as well as videos illustrating the basic $\mathcal{MOQA}$ operations and a selection of $\mathcal{MOQA}$ programs. The reader is advised to read Chapters 1 and 3, followed by a viewing of the tutorial, before proceeding with later chapters in this work.

Cork, Ireland, December 2007 *Michel Schellekens*

Acknowledgements

Sincere thanks to the CEOL-team for helpful comments on the presentation of the work and countless hours of interesting discussions. It has been a pleasure to work with excellent, critical and inquisitive students. Warm thanks to colleagues Joseph Manning and Emanuel Popovici for their friendship and support. Special thanks to Chantal Berline who reviewed some of the earlier drafts and provided several PPS students "on loan" to CEOL. Steve Brookes suggested the relation with pomsets. Schloß Dagstuhl and DAAD supported a research stay during which some early ideas were explored. Science Foundation Ireland's strong support[3] enabled the full exploration of the approach, the implementation of the $\mathcal{MOQA}$ language and the associated static timing tool *Distri-Track*, as well as the Flash implementation of the $\mathcal{MOQA}$ tutorial. My thanks to the following researchers, both from academia and industry, who were supportive from the start and whose kind words made a difference in challenging times: Chantal Berline, Greg Bollella, Steve Brookes, Roberto di Cosmo, Gordon Plotkin, Dana Scott and Dave Vernon. Also thanks to Rob Esser, Philippe Flajolet, Don Knuth and Peter Puschner for encouraging comments on the nature of the work. The following researchers co-authored chapters in this work: Diarmuid Early co-authored Chapter 6, "Average-Case Time of Basic $\mathcal{MOQA}$ Operations" and Dave Hickey and Menouer Boubekeur co-authored Chapter 10, "Distri-Track". David Devlin and Yin Jie Chen developed the Flash implementation of the $\mathcal{MOQA}$ tutorial, which accompanies this work.

[3] SFI Investigator Award 02/IN.1/181.

Contents