MOQA; unlocking the potential of compositional static average-case analysis

M.P. Schellekens
University College Cork, Department of Computer Science, Centre for Efficiency-Oriented Languages (CEOL), Ireland

ARTICLE INFO
Available online 11 February 2009
Dedicated to Dieter Spreen on the occasion of his 60th birthday, with thanks for his friendship and interesting discussions over the years!

Keywords:
Randomness preservation
Algorithms
Average-case analysis
Modularity
Compositionality
Static timing analysis
Static power analysis

ABSTRACT
Compositionality is the “golden key” to static analysis and plays a central role in static worst-case time analysis. We show that compositionality, combined with the capacity for tracking data distributions, unlocks a useful novel technique for average-case analysis.

The applicability of the technique has been demonstrated via the static average-case analysis tool Distri-Track. The tool automatically extracts average-case time from source code of programs implemented in the novel programming language MOQA (MOdular Quantitative Analysis). MOQA enables the prediction of the average number of basic steps performed in a computation, paving the way for static analysis of complexity measures such as average time or average power use. MOQA has as a unique feature a guaranteed average-case timing compositionality.

The compositionality property 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 very helpful advantage in static analysis, something which is not available in current languages. Moreover, re-use is a key factor in the MOQA 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 MOQA programs.

The MOQA “language” essentially consists of a suite of data-structuring operations together with conditionals, for-loops and recursion. As such MOQA can be incorporated in any traditional programming language, importing all of its benefits in a familiar context (MOQA is implemented at CEOL in Java 5.0 as MOQA-Java). Compositionality for average-case is subtle and one may easily be tempted to conclude that compositionality “comes for free”. For genuine compositional reasoning however, one needs to be able to track data and their distribution throughout computations; a non-trivial problem. The lack of an efficient method to track distributions has plagued all prior static average-case analysis approaches. We show how MOQA enables the finitary representation and tracking of the distribution of data states throughout computations. This enables one to unlock the true potential of compositional reasoning. Links with reversible computing are discussed. The highly visual aspect of this novel and unified approach to the Analysis of Algorithms also has a pedagogical advantage, providing students with useful insights in the nature of algorithms and their analysis.

© 2009 Elsevier Inc. All rights reserved.

E-mail address: m.schellekens@cs.ucc.ie

1567-8326/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2009.02.006
1. Introductory notions

We recall [8] that if $S$ is a finite sample space then for any event $A \subseteq S$, we have: $Pr(A) = \sum_{s \in A} Pr(s)$. In the case $Pr(s) = 1/|S|$, we have a uniform distribution over $S$. In such a case the experiment is often described as “picking an element of $S$ at random”. A typical example of a uniform distribution is the process of flipping a fair coin [8], one for which the probability of obtaining a head is the same as the probability of obtaining a tail, i.e. $1/2$. If one flips the coin $n$ times, we have the uniform probability distribution defined on the sample space $S = \{H, T\}^n$, where $S$ has size $2^n$. Each elementary event of the sample space can be represented as a string of length $n$ over $\{H, T\}$ and each occurs with probability $1/2^n$. An example of interest to average-case analysis is the set of all lists of a given size $n$ that have pairwise distinct elements. If one identifies lists up to label-isomorphism then this set has exactly $n!$ elements. These $n!$ lists represent the $n$ “states” a random list can be in. When all input lists for the program are equally likely to occur during the execution of the program, each of the lists has probability $1/n!$, i.e. their distribution is uniform and the inputs are referred to as “random” lists.

The terminology of randomness has spread in the computer science literature and “random structures” as well as “randomness preservation” are notions that refer to a structure with a uniform distribution and to preserving the uniform distribution respectively, as opposed to the much more general use of a “random variable” in probability theory.

The following function notation is used: $f|A$ indicates the restriction of the function $f$ to the subset $A$ of the domain of $f$.

The notion of a bag will be useful to capture distributions in our context. We assume the reader is familiar with the concept of a bag, also called multiset [1]. A bag is a finite set-like object in which order is ignored but multiplicity is explicitly significant. We refer to the number of times an element occurs in a bag as the multiplicity of the element. The cardinality of a bag is the sum of the multiplicities of its elements. Each bag $A$ of $n$ elements has an associated set $B = \{b_1, \ldots, b_k\}$ such that all elements of $A$ belong to $B$ and such that each element $b_i$ of $B$ is repeated $K_i$ times where $1 \leq K_i \leq n$ and $\sum_{i=1}^{k} K_i = n$. It is clear that a bag $A$ can be represented in this way as a set of pairs $\{(b_1, K_1), \ldots, (b_k, K_k)\}$.

Two bags $A$ and $B$ with set representations $\{(b_1, K_1), \ldots, (b_k, K_k)\}$ and $\{(c_1, L_1), \ldots, (c_l, L_l)\}$ have the same cardinality in case $\sum_{i=1}^{k} K_i = \sum_{j=1}^{l} L_j$.

A bag $A$ with representation $\{(b_1, K_1), \ldots, (b_k, K_k)\}$ is uniformly distributed iff for each $i \in \{1, \ldots, n\}$. $K_i = K_j$. It is clear that if $A$ is a bag $\{(b_1, K_1), \ldots, (b_k, K_k)\}$ and $A$ is uniformly distributed then we can simply use the strict bag representation $(B, K)$ for $A$, which indicates that the bag $A$ consists of $K$ copies of the associated set $B$. In particular: $|A| = |B|$. Each element of a uniformly distributed bag $A$ with associated set $B$ arises with equal probability of $\frac{|B|}{|A|} = \frac{K_i}{n}$. For a general strict representation of a bag $\{(b_1, K_1), \ldots, (b_k, K_k)\}$, the probability of each element $b_i$ to occur in the bag is: $\frac{K_i}{\sum_{i=1}^{k} K_i}$.

To keep track of the number of times that a particular output is produced, we will indicate the range of the input-output function of a program as a bag. For any program $P$ we indicate the bag of its inputs by $I_P$. The bag of inputs of size $n$ is denoted by $I_P(n)$. A bag of inputs $I$ for a program $P$ is a sub bag of the input bag $I_P$. Typically we will require that $I \subseteq I_P(n)$ for some $n$. $O_P(I)$ denotes the bag of outputs, referred to as “the output bag”, of the computations of a program $P$ on a bag of inputs $I$. If we abbreviate $I_P(n)$ by $I_n$ then we denote the output bag $O_P(I_n)$ by $O_P(n)$.

Under the assumption that each input produces a corresponding output, it is clear that in case $I$ is an input bag for a program $P$, the bags $I$ and $O_P(I)$ have the same cardinality. We remark that in our context all programs are assumed to terminate and inputs are required to produce outputs. The termination condition is consistent with standard assumptions in the Real-Time language context, where for timing purposes programs are required to terminate.

Example 1. Consider any sorting algorithm $P$, where the inputs of the algorithm consist of the $n!$ lists of size $n$. The bag of outputs $O_P(I_n)$ has as strict representation $\{(S_n, n!)\}$ and consists of $n!$ copies of the sorted list $S_n$.

We assume that the reader is familiar with partial orders, order-ismorphism, the discrete order, the total (or linear) order and Hasse diagrams [10].

We call the partial order $\langle X_2, \subseteq_2 \rangle$ a refinement of the partial order $\langle X_1, \subseteq_1 \rangle$ in case $X_2 \subseteq X_1$ and $\forall x, y \in X_2. x \subseteq_1 y \Rightarrow x \subseteq_2 y$. In other words a new partial order refines a first one, in case its underlying set is included in the underlying set of the first one and all order relations of the first partial order are still satisfied in the new partial order.

MOQA computations will involve operations which systematically refine orders under consideration. For instance, a sorting algorithm will gradually introduce more order and hence will refine the ordering under consideration. Other operations may simply leave the original order intact, which is interpreted as a trivial refinement.

2. Introduction

We focus on a motivation and introduction of the basic principles underlying the new MOQA language, rather than provide a more rigorous introduction to the language for which we refer the reader to [49]. This includes a motivation of the central notion of random bags, which are used to represent data distributions, and random bag preservation, which enables the tracking of such distributions. We present a high level overview of the MOQA language as well as an illustration of its nature, scope and applicability. To illustrate the applicability of the new technique we obtain the compositional derivation of the exact average-case time of a MOQA implementation of the well-known algorithm Quicksort. This derivation is very close in spirit to traditional semantic style compositional reasoning; opening up new avenues for investigating the open problem.
of bridging Semantics and Complexity, which will be explored elsewhere. The example serves to illustrate the graphical nature of the tracking of the distributions. This is a key feature, supporting the MOQA programmer in producing code and supporting the functioning of the static timing tool Distri-Track [20,21]. It also has the potential to provide an educational benefit in the teaching of algorithms courses.

2.1. Static average-case analysis

2.1.1. The need for static average-case analysis tools

Static average-case analysis merits independent exploration in view of its core Computer Science nature [26,8]. From a practical point of view, static average-case analysis tools have the potential to contribute to a variety of areas. Average-case execution time (ACET) is a key measure in estimating heat-dissipation/power consumption, since it provides information on "typical" input behaviour [56,32]. Static average-case analysis also provides crucial information complementing worst-case execution time (WCET) information. Such complementary information can potentially aid better budgeting of resources in a Real-Time context [30]. However, at this stage there are no widely applicable static average-case analysis tools available. Industry needs to rely on simulation, i.e. the execution of code on a (sufficiently large) selection of data to experimentally derive information on the average-case behaviour. This entails imprecision as sample spaces are not necessarily representative and implies an extra cost factor as simulation is time consuming. The simulation problems affect both software and hardware analysis. Moreover, re-use is excluded since current languages and hardware are not certified to be modular (compositional). Our aim is to present a new approach which can pave the way for novel static analysis tools to address this need and which may, in the longer term, pave the way for modular hardware analysis.

2.1.2. Compositionality: the golden key to static analysis

To understand why the compositionality principle is a crucial static timing principle, a comparison with static worst-case timing is useful. It is well-known that static worst-case timing techniques have been successfully developed and a variety of tools have been developed as a result. There are a multitude of these of which we only report a limited selection, ranging from academic approaches, e.g. [23,18], to commercial ones such as Absint’s WCET analyzers. The principle which enables the development of static worst-case timing tools is a partial compositionality principle which lies at the heart of all current static worst-case timing tools. This principle essentially states that the worst-case time of the sequential execution of two programs is bounded by the sum of the worst-case times of these programs, which enables WCET estimation. This allows real-time engineers for instance to estimate the worst-case time of a for-loop in terms of a summation over the worst-case times of the executions of the for-loop body. Note that the worst-case measure is only partially compositional, in that we cannot get the exact determination of the worst-case time, only an upper bound of this time. For a framework developed to address full compositionality for WCET, we refer the interested reader to [6].

Similar to the usefulness of a (partial) compositionality principle in a WCET context, the availability of a compositionality principle for average-case time can pave the way for static average-case timing tools. However, it is clear that none of the academic texts currently supplies a method which is formally guaranteed to provide a compositional average-case timing. To address compositionality in a formal way, the following crucial question needs to be answered: given the average-case time of two programs P_1 and P_2, how can this information be used to determine the average-case time of the sequential execution of P_1; P_2? Unless this compositionality question is resolved, static average-case tools remain beyond reach.

2.1.3. The main bottleneck for static average-case analysis

It is easy to see why the above compositionality question proves problematic to answer: the average-case time of P_2, in the context of the sequential execution P_1; P_2, depends on the distribution of the input data for P_2. This distribution however in practice is typically not known since it will depend on the computation determined by P_1. Assume for instance that we know the distribution of the input data for P_1 and say we denote these input data by the collection I_1. The actual computation of P_1 over these input data I_1 will produce the new input data for P_2, say I_2. However, one typically cannot track the distribution throughout the computation, i.e. one cannot, in general, compute the distribution over I_2 from the distribution over I_1. Results have appeared on distribution transformations [27], but these methods remain purely mathematical and do not lend themselves to a concrete method for effectively computing new distributions from prior ones. Probabilistic attribute grammars have been proposed as one remedy to represent standard distributions in an effective (computable) way [19], but again, no systematic method for efficiently tracking these grammars throughout the computations is offered. Attribute grammars give rise to useful ways to determine additional quantitative information such as standard deviation [29].

The compositionality problem for average-case analysis has been overcome via the MOQA approach, as outlined below. In a nutshell, the static average-case analysis tool Distri-Track statically tracks the distribution of the data-states during the computation of MOQA programs. This tracking is achieved through a finitary representation of the distribution via a "random bag" and through a careful design of the basic operations to ensure that the capacity for such distribution representation is preserved throughout the computation.

The method addresses deep open problems in the average-case analysis of algorithms, but fortunately the basic principles underlying MOQA can be readily explained. The notion of random bags and their preservation are the principal and intuitive concepts underlying our approach, leading to a new foundation for the Analysis of Algorithms. These concepts capture the data distribution and its preservation. Since the MOQA language and the notion of distribution tracking based on random
2.2. Removing the bottleneck for static average-case analysis

2.2.1. The need for novel language design

The core obstacle to unlocking the potential of compositionality for static average-case timing tools is that certain well-known classical data structuring operations, such as the delete operation, fail to exhibit compositional behaviour w.r.t. the average-case time measure. This problem prevents the analysis of algorithms such as Heapsort which involves updates involving repeated insertions and deletions and which are a well-known bottleneck in the average-case analysis area [14]. Hence our target shifted from model design to novel language design. This aim is captured by the notion of an “Efficiency-Oriented language”.

2.2.2. Efficiency-Oriented languages

Efficiency-Oriented languages (EO-languages) are languages which have purpose designed operations to enable the modular (i.e. compositional) static extraction of quantitative information, such as time or power use. Novel language constructs, intended to achieve compositionality, are designed to replace traditional language constructs, whenever needed, including conditionals, loops, or standard data structuring operations. This ensures that Efficiency-Oriented languages can be naturally embedded/used in traditional programming languages, by replacing standard language constructs, when needed, by suitable implementations of their intended replacements.

A first example of an Efficiency-Oriented language is provided by the Burns-Puschner language which achieves compositionality for the worst-case measure through a purpose designed conditional statement [6].

The programming language MOQA is the first Efficiency-Oriented language to enable the compositional determination of the average-case number of basic instructions of its programs. MOQA has been specified and implemented in Java 5.0 at CEOL. We remark that the language constructs have been designed, when needed, to replace the standard data structuring operations to achieve compositionality. MOQA essentially consists of a suite of data structuring operations which can be implemented in any standard programming language.

It is important to distinguish Efficiency-Oriented languages from standard Real-Time languages. Standard Real-Time languages are typically obtained from existing programming languages by restrictions which guarantee termination, such as exclusions of while-loops and by restrictions on nested conditionals, to make worst-case analysis practical. Real-Time languages typically do not significantly alter existing language constructs. This has the advantage that Real-Time languages remain close in spirit to traditional programming languages.

The disadvantage of this approach is that compositionality cannot be certified for Real-Time languages. This has two significant implications: a loss of precision, clear from the fact that worst-case estimates typically overshoot the real worst-case time through upper bounds, and lack of re-use of prior estimates in case of a change in code. I.e. traceability is typically lost in this context. Efficiency-Oriented languages address this problem through novel language construct design to guarantee compositionality.

In designing Efficiency-Oriented languages, it is important that novel constructs do not drastically increase the complexity of standard constructs which they replace. The language introduced in [6], does tend to increase time. The Real-Time language area accepts moderate slow-downs if increased predictability results. For the case of [6] great care needs to be taken since the slow-down can be drastic. Burns and Puschner [6] discusses how to avoid this problem. MOQA novel constructs typically have equal or improved speed when ignoring book keeping overheads, or lead to an increase by a constant factor.

2.2.3. MOQA an EO-language for static average-case analysis

We describe some earlier work on the MOQA approach and related work. Limited applicability affects all of the current static average-case analysis methods, including LUO [13,14] which is the most widely researched average-case timing approach, and also the Cohen-Hickey approach based on Ramshaw’s approach to average-case timing [19,33].

A main bottleneck to static average-case analysis techniques, was posed by algorithms over so-called dynamic data structures [12–15]. Dynamic data structures are data structures subject to systematic updates through repeated insertions and deletions. Examples are heaps and binary search trees.

We identified compositionality as a crucial property in average-case analysis. The lack of compositionality prevents for instance the analysis of dynamic data structure algorithms such as Heapsort [50]. As pointed out in [11], the exact average-case time of all Heapsort variants is unknown to date. This is directly linked to the fact that standard Heapsort [1] does not “preserve randomness”. We provide an example in Section 3.4 which illustrates this fact. [50] reports on a new version of Heapsort, Percolating Heapsort, which is faster (both in average number of comparisons and in “real-time” as measured by a Java profiler) than all standard Heapsort variants. The algorithm has been directly designed based on the MOQA delete

---

1 A terminology coined by CEOL.
2 With some exceptions to this rule in the context of RT-Java which also needs to account for time involved in garbage collection.
3 Not to be confused with the standard Computer Science interpretation of dynamic programming.
The open problem which stood over 30 years has been resolved based on the MOQA delete operation and by exploiting MOQA compositional approach as part of the analysis. Automation of the technique still provides a further challenge as discussed in [49].

The MOQA language also provides a means to implement programs close to standard practice. Prior approaches, such as LÚO, required potentially drastic reimplementation of programs in order to fit algebraic molds to enable algebraic manipulation methods to be applied in that context. Next, we describe some prior work on MOQA.

Boubekeur et al. [3–5] report on applications of the MOQA approach in a Real-Time Java context. Townley and Schellekens [55] illustrates sorting algorithms implemented in MOQA. Schellekens [48] presents the product MOQA operation; one of the basic operations of the language. Finally, Hickey [20,21] reports on the Distri-Track static average-case timing tool which is applicable to MOQA code.

Thus far however, no systematic overview of the MOQA language and its principle ideas is available. We provide such an overview here and motivate and introduce the fundamental ideas underlying the general approach. The full specification and formal verification of the MOQA language forms the subject of the Springer book “A modular calculus for the average cost of data structuring” [49]. We will also discuss some connections between MOQA and high-level reversible languages. Most reversible approaches remained at hardware level. It is interesting to note that the use of MOQA as a high level reversible language brings a new type of application to the area of reversible computing, namely that of modular static average-case analysis.

2.3. Background information

This section is included to take into account that the material may be of interest to readers with a variety of backgrounds. The first section addresses readers with a background in Semantics, providing an overview of relevant research in the area. The reader not specialized in Semantics, can omit this section which is independent from the remainder of the paper.

The second subsection addresses the notion of static timing in our context.

2.3.1. Overview of prior semantic approaches to complexity analysis

Compositionality is a key notion in the Semantics area. Taking into account its potential use in the static analysis area, the Semantics community has for quite some time speculated that semantic methods, involving formal models to specify the computational “meaning” of a program, could provide a means to extract complexity information on the program. The challenge is to use semantic methods to obtain information such as the amount of time it takes to execute the program.

The first attempt to combine semantics and complexity was made by Gurr [17] under guidance of G. Plotkin. Gurr developed a semantic framework for exact time analysis, i.e. the determination of the time which a program takes per input. Industry has traditionally focused on alternative measures such as the worst-case and average-case measure. The semantic treatment of these fundamental measures however proved to be problematic though some progress was made on the worst-case measure [17].

Schellekens [42] focused on combining semantic approaches with average-case complexity analysis. This led to the novel theory of complexity spaces [43], geared towards extracting average-case time information of programs and which has been applied in the context of Divide and Conquer algorithms. The work on complexity spaces received extensive follow-up by the Valencia-based research group led by Romaguera [34–39,51,40].

Several conferences advocated the importance of combining Semantics and Complexity, such as the IFIP2000 International Conference on Exploring New Frontiers of Theoretical Informatics, Sendai, Japan, which stressed bridging Semantics and Complexity as a main research challenge. Later research focused on building new models, which potentially could be used to extract quantitative information of programs (such as timing and power use). Such models are referred to as Quantitative Domains [45]. It has been shown that all domains (i.e. semantic models) are quantifiable via a partial metric. Hence such models could potentially be used to extract quantitative information of programs [46,47].

The promise of bridging Semantics and Complexity lies in the potential to extract additional quantitative information from software code in a compositional way.

However, purely semantic based approaches cannot resolve the problem of extracting complexity information at source code level.

It became clear, as outlined in Section 2.2, that in order to develop successful static average-case timing tools one needed to proceed beyond the development of refined models for existing languages and novel language design is required. Quantitative domains in general do not support, at this stage, the derivation of complexity information for standard programming languages and there is a clear need for novel language design to bridge Semantics and Complexity.

2.3.2. The meaning of static timing in our context

Before continuing with a discussion of the MOQA language we clarify the meaning of static timing. Static timing as usual refers to the capacity to estimate the time which a program takes upon execution, directly from an analysis of the program’s code, as opposed to deriving information on the time from an actual running of the program (simulation). The situation is similar to analyzing the blue prints for a bridge to determine its load capacity, as opposed to driving heavy trucks across to test this in practice.
To clarify the meaning of “time” in our context, we remark that our investigations have focused on data restructuring algorithms which are comparison driven, i.e. for which each action (data-reorganization) is based on a prior comparison between data. The average-case time $T_A(n)$ of an algorithm $A$ is then defined as the average number of comparisons carried out over inputs of size $n$. This is in line with the standard approach in the average-case analysis area [26,14]. To fine-tune the static analysis further, other basic operations (such as swaps and assignments) can be accounted for as discussed in [5]. Preliminary tests have indicated that these results are quite close to results obtained on the average-case execution time using a Java profiler [5].

3. The $\text{MOQA}$ language

3.1. General description

3.1.1. $\text{MOQA}$ nature and scope

$\text{MOQA}$ is a special-purpose high-level language. The $\text{MOQA}$ data structuring operations were originally designed to incorporate all the standard operations over abstract data types [31]. $\text{MOQA}$ has extensive programming capacity in the sense that it incorporates for-loops, (terminating) recursion, and conditionals. This approach enables the programming of a wide variety of data restructuring algorithms, such as most sorting and searching algorithms, including those operating over dynamic data structures.

$\text{MOQA}$ can be viewed as a suite of data structuring operations, which can be implemented in any existing general purpose programming language. Note that we do not propose $\text{MOQA}$ as a stand alone novel language. Instead we advocate its use in a variety of contexts through appropriate implementation of the basic operations in accordance with $\text{MOQA}$ principles.

As a first example of such an implementation, $\text{MOQA}$-Java has been implemented in Java 5.0 at CEOL. Currently, $\text{MOQA}$’s applicability is restricted to data restructuring contexts. Investigations are ongoing at CEOL to extend the scope to include numeric operations and general graph based applications, such as shortest path methods. The $\text{MOQA}$ version reported here regards the language as described above, where extensions are regarded as a separate issue which will be reported elsewhere.

3.1.2. $\text{MOQA}$ data structures and data-labelings

$\text{MOQA}$’s data structures are (a finite number of) finite partial orders, represented by Hasse diagrams. A Hasse diagram represents the immediate directed links between elements, omitting transitive and reflexive links [10]. Hence a wide variety of standard data structures can be incorporated in this context. The nodes in these graphs are interpreted as variables storing data.

The $\text{MOQA}$ data are labelings of the data structures. A data-labeling is simply an assignment of a finite number of values, one value per node of the data structure. These labels can be any value, e.g. natural numbers, real numbers or words, or even paired data or other data structures containing data, etc. Two conditions need to be satisfied in this context:

1. Label-comparability: Any two labels need to be comparable with respect to a given order on labels. For instance, natural numbers are always comparable and so are real numbers, words,...

2. Order-consistency: In assigning labels as values to the nodes of a data structure, the directed links of the data structure need to be respected. In other words, if there is a directed link from a node $x$ to a node $y$, then the label assigned to $x$ must be less than or equal to the label assigned to $y$.

We refer to any assignment of labels to all nodes of the data structure, respecting the above order-consistency condition, as a data-labeling.

Formally, this can be concisely captured as follows:

A data structure is a finite partial order $X = (X, \sqsubseteq)$ and labels are taken from a total order $L = (\mathcal{L}^*, \preceq)$. A data-labeling then is a morphism of orders $F: X \rightarrow L$.

$\text{MOQA}$ programs compute over data-labelings and transform data-labelings to new data-labelings at each stage of the computation.

We briefly discuss the treatment of repeated labels in our context. At this stage we remark that one approach to deal with repeated labels is to assume that each label, repeated or not, comes equipped with a special tie-breaker value. During computations this index value is used as a tie-breaker to decide an outcome of a comparison between identical labels. This of course amounts to considering all labels distinct and hence our analysis, which is carried out on states and under the assumption of distinct labels, will yield the correct result.

\footnote{Labels are totally ordered.}
Hence from here on we will modify the definition of a data-labeling to be an increasing bijection or in other words, an increasing function where label values are pairwise distinct.

**Definition 2.** A data-labeling \( F \) is a function from a finite partial order \( (X, \sqsubseteq) \) to a countable totally ordered set of labels \( L^* \) such that \( F \) is an increasing bijection.

**Definition 3.** Consider a data structure determined by a finite partial order \( (X, \sqsubseteq) \) and a linearly ordered collection of labels \( L \). \( \mathcal{D}_{L^*}(X, \sqsubseteq) \) denotes the collection of all data-labelings over the partial order \( (X, \sqsubseteq) \).

Alternatively we can state that the set of data-labelings \( \mathcal{D}_{L^*}(X, \sqsubseteq) \) is the set \( \text{Hom}[X, L] \).

**Notation 4.** We let \( [k] \) denote the finite set \( \{1, \ldots, k\} \) for \( k \in \mathbb{N} \). In case \( [k] \) is equipped with the discrete order, we denote this by \( \Delta_k \). In case \( [k] \) is equipped with the linear order, we denote this by \( k \).

**Example 5.** We consider the example of the discrete partial order of size 3, denoted by \( \Delta_3 \) and the collection of natural number labels \( L = \mathbb{N} \). \( \mathcal{D}_{\mathbb{N}}(\Delta_3) \) consists of all lists of size 3 which take natural number values.

**Example 6.** We consider an example of a data structure a tree \( T_4 \) of size 4. The tree \( T_4 \) is determined by the Hasse diagram displayed on the left below:

Three examples of data-labelings of the tree \( T_4 \) are displayed: the labeling \( F_1 \) with the labels \( \{1, 4, 6, 10\} \), a second data-labeling \( F_2 \) with the labels \( \{2, 4, 6, 9\} \) and a third data-labeling \( F_3 \) with the labels \( \{5, 9, 22, 35\} \). Note that the data-labelings of the collection \( \mathcal{D}_{L^*}(T_4) \) are “heaps” in the traditional meaning of this data structure [8].

3.1.3. \textit{MOQA} states

To enable timing of computations it is important to identify the states that data-labelings can be in. Essentially, states reflect the relative order that the labels can be in on any given data structure. The values of the labels are irrelevant in this context, only their relative order is captured.

Consider for instance data-labelings \( F_1 \) and \( F_2 \) of Example 6. These data-labelings are in different states since the order between the labels of \( y \) and \( z \) differs, i.e. for data-labeling \( F_1 \) the label for \( y \) is smaller than the label for \( z \), while for data-labeling \( F_2 \) the opposite holds. Note that data-labelings \( F_2 \) and \( F_3 \) are considered to be in the same state since the relative order between labels is the same for both labelings.

In the context of so-called comparison-based algorithms, i.e. algorithms for which each action is based on a prior comparison, the computation time can be reliably estimated by the number of comparisons performed during a computation. This number can then be multiplied by the expected time it takes to carry out actions following a comparison, to produce a reasonable estimate of the execution time. In this context data-labelings which are in the same state will lead to the same number of comparisons during a computation. Hence, for the purpose of the analysis and under the assumption that data are equally likely to occur in any of the states, it suffices to carry out the analysis on states as opposed to on general data-labelings. This reduces the data to be considered from a potentially infinite space to a finite space.

This type of reduction is a natural approach consistent with standard algorithms analysis [1], formalized and generalized in our context via the notion of a random structure, which essentially captures the finite state space. We will provide further examples and a formalization of this notion below.

As usual in the analysis of algorithms, to simplify the analysis, we will assume that there are no repeated labels involved in the states. Various treatments of the general analysis, accounting for repeated labels, are considered a separate subject which will be discussed elsewhere. Since, as observed earlier on, data-labelings can be assumed not to have repeated labels, the approach is a reasonable one in our context.

To further illustrate the notion of a state, consider a simple example of unordered lists of size 3. These can be represented in \textit{MOQA} as data-labelings of a finite partial order of size 3, say with nodes \( x, y, z \), where no two distinct nodes are linked.\(^5\)

\(^5\) The discrete partial order of size 3.
We recall from Example 5 that this order is denoted by $\Delta_3$. Note that there can be infinitely many data-labelings. Consider for instance the infinite sub-collection of lists of size 3, consisting of a particular selection of sorted lists, say:

$$(1,2,3), (2,3,4), (3,4,5), \ldots, (n,n+1,n+2), \ldots$$

Despite the possibility of infinitely many data-labelings, in practice each data-labeling will occur in a unique state. Moreover, there will only be finitely many such states. For instance, each of the above data-labelings is sorted, and hence occurs in the unique sorted state. Similarly there are infinitely many reverse sorted lists, all of which are in the reverse sorted state. To represent all states a list of size 3 can be in, we fix three labels. Any three labels will do for this purpose, say the labels 1,2,3.

As pointed out earlier, we do not account for repeated labels in this context, i.e. labels of representatives are distinct. The six possible states for a list of size 3 can hence be represented by the $3!$ possible data-labelings using only the 3 fixed labels 1,2 and 3:

$$(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1).$$

We illustrate this further with the data-labelings for the tree $T_4$ in Example 6. We recall that these data-labelings are heaps of size 4. If we use four distinct values, say 1,2,3,4 to represent the states then we have only three possible states as displayed.

![Diagram of heaps]

Returning to Example 6, note that the first heap (data labeling $F_1$ is in state $H_4[1]$. The second and third heap (data labelings $F_2$ and $F_3$) are in state $H_4[2]$, since the relative order of labels is the same as the relative order of state $H_4[2]$.

### 3.1.4. Random structures to capture states

States are obtained by identifying data-labelings when their relative order is the same. This can be formally captured by the notion of a labeling-isomorphism.

**Definition 7.** Consider two partial orders $(X_1, \sqsubseteq_1)$ and $(X_2, \sqsubseteq_2)$.

A function $\Psi: X_1 \to X_2$ is increasing if $\forall x, y \in X_1, x \sqsubseteq_1 y \implies \Psi(x) \sqsubseteq_2 \Psi(y)$. Consider data-labelings $F_1 \in \mathcal{D}_{\sqsubseteq_1}(X_1, \sqsubseteq_1)$ and $F_2 \in \mathcal{D}_{\sqsubseteq_2}(X_2, \sqsubseteq_2)$. Let $\leq_1$ and $\leq_2$ represent the linear orders on the label sets $L_1$ and $L_2$ respectively.

Data-labelings $F_1: X_1 \to L_1$ and $F_2: X_2 \to L_2$ are labeling-isomorphic iff

1. the underlying orders are isomorphic; there exists an increasing bijection $\Psi$ from $X_1$ to $X_2$ which has an increasing inverse.
2. the bijection $\Psi$ respects the ordering on labels, i.e.

$$(\ast) \ \forall x, y \in X_1, F_1(x) \leq_1 F_1(y) \implies F_2(\Psi(x)) \leq_2 F_2(\Psi(y)).$$

In case $F_1$ and $F_2$ are labeling-isomorphic, we denote this by $F_1 \approx F_2$.

Note that in condition (2) it suffices to require that the implication $\implies$ holds as opposed to the equivalence $\Leftrightarrow$ since data-labelings are injective by Definition 2.

To simplify the presentation, we assume that we only consider data-labelings over a single data structure, i.e. over a single partial order. We revisit the case of multiple data structures in Section 3.2.2.

In case the finite partial orders $(X_1, \sqsubseteq_1)$ and $(X_2, \sqsubseteq_2)$ coincide, it suffices to consider the bijection in the definition of labeling-isomorphic to be the identity function, while ($\ast$) reduces to:

$$(\ast\ast) \ \forall x, y \in X_1, F_1(x) \leq_1 F_1(y) \iff F_2(x) \leq_2 F_2(y).$$

To distinguish this case from the general one, we define the following equivalence relation on data-labelings $F_1 \in \mathcal{D}_{\sqsubseteq_1}(X_1, \sqsubseteq_1)$ and $F_2 \in \mathcal{D}_{\sqsubseteq_2}(X_2, \sqsubseteq_2)$:

$$F_1 \approx^* F_2 \iff (X_1, \sqsubseteq_1) \text{ and } (X_2, \sqsubseteq_2) \text{ coincide and } (\ast\ast) \text{ holds.}$$

The collection of states over a single data structure can be captured via the notion of a random structure.

**Definition 8.** A random structure $\mathcal{R}(X, \sqsubseteq)$ for a given set of data-labelings, say $\mathcal{D}_{\sqsubseteq}(X, \sqsubseteq)$, is defined to be the quotient of this set by the labeling-equivalence relation $\approx$, i.e.

$$\mathcal{R}(X, \sqsubseteq) = \mathcal{D}_{\sqsubseteq}(X, \sqsubseteq)/\approx.$$
Remark 9. Representatives of the equivalence classes (i.e. “states”) can be picked by fixing a fixed set of labels \( \mathcal{L} \subseteq \mathcal{L}^* \) with same cardinality as the finite partial order \((X, \sqsubseteq)\). Any such choice of labels \( \mathcal{L} \) will do. Up to isomorphism the label set may always be regarded as \( k \) with \( k = |X| \). Representatives, i.e. states, are then obtained by considering the collection of data-labelings which only have pairwise-distinct labels from the selected subset of labels.\(^6\) We will continue to work in the following with the collection of representatives (states) as opposed to the collection of equivalence classes, where we denote this collection of representatives chosen w.r.t to a set of labels \( \mathcal{L} \) by \( \mathcal{R}_C(X, \sqsubseteq) \). We will refer to this collection, with some abuse of terminology, as a random structure.

Random structures capture the states over a given data structure and represent the fact that each data-labeling is assumed to occur with equal probability in any of the states of the random structure.

In summary:

In \( \text{MQQA} \), the data-labelings of a given data structure occur in finitely many states. Each data-labeling will occur in one of these finitely many unique states at any given time. Moreover, data-labelings have equal chance to occur in one of these states. The finite collection of data states is referred to as a random structure.

For data structures, such as lists and heaps, we use the following notation, where we work modulo identification up to labeling-isomorphic copies: \( \mathcal{A}_n \) denotes the set of \( n \) non-isomorphic lists of size \( n \) with pairwise distinct elements, \( \mathcal{H}_n \) denotes the set of non-isomorphic heaps of size \( n \) with pairwise distinct elements. Also, we let \( \mathcal{S}_n \) denote the set consisting of the single sorted list of size \( n \).

We discuss probability distributions in more detail below.

3.2. Tracking distributions

3.2.1. The uniform distribution

Static analysis techniques for average-case analysis have focused on random data to provide information on typical (average) running time. This is for instance the case for sorting algorithms such as Quicksort, for which the overall performance on average, with respect to the uniform distribution on input lists, is determined to be optimal. The performance will of course depend on the actual collection of inputs provided for a particular application. The performance under the assumption of uniform data distributions is used as an indicator of the typical time the algorithm will take on arbitrary data. The analysis under the assumption of uniform input data distribution is of course also reasonable in the context of randomized input data [8].

The assumption of random data amounts to considering inputs equally likely to occur in any of a given number of finite states. Though our method transcends this assumption through the use of random bags, we note at this stage that random data can be concisely captured via the notion of a random structure. The intuition behind a random structure is that the structure determines a uniform distribution.

Consider for instance the random structure over \( \Delta_3 \) corresponding to Example 5:

\[
\mathcal{R}(\Delta_3) = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}.
\]

The random structure \( \mathcal{R}(\Delta_3) \) represents the fact that all data-labelings over \( \Delta_3 \) are equally likely to occur, i.e. each list of size 3 has equal probability of \( \frac{1}{6} \) to occur in one of these six states. In other words, the distribution is uniform.

Another example of a random structure is the collection of the three states for the tree partial order discussed in Example 6, i.e.

\[
\mathcal{R}(T_4) = \{H_4[1], H_4[2], H_4[3]\}.
\]

All data-labelings (heaps) of size 4 are equally likely to occur, with probability \( \frac{1}{3} \), in one of these three states.

Finally, we discuss the extreme case of the “random” structure over the linear partial order of size 3. This linear partial order is denoted by \( S_3 \), where “S” is used for “Sorted”. This random structure only has one possible state. It is “random” in the sense that it contains all states allowed by the partial order. Its unique state is the sorted list of size 3, which for the label collection \( \{1, 2, 3\} \) yields:

\[
\mathcal{R}(S_3) = \{(1, 2, 3)\}.
\]

All data-labelings for the linear partial order are equally likely, with probability 1, to occur in the single sorted state.

3.2.2. S-distributions

In practice of course, there may be several data structures in any given collection of inputs. To represent this, the notion of a random bag is introduced. A random bag represents the data structures involved as well as the distribution of the data-labelings via the relative distribution of the random structures in the random bag.

A random bag consists of finitely many random structures, \( R_1, \ldots, R_n \), each of which has a multiplicity. A multiplicity is a natural number indicating the frequency with which data states occur for a particular data structure, relative to the other data structures. This enables a representation of distributions which is more general than the uniform distribution.

\(^6\) Labelings of a partial order using labels from finite linearly ordered set of same cardinality as the partial order correspond to the well-known mathematical notion of linear extensions of a partial order. In Computer Science, this notion is also referred to as a topological sort.
A random bag is a concise notion capturing both S-probability and S-distribution. The diagram illustrates the S-distribution corresponding to the random bag \(((R_1,2), (R_2,1), (R_3,5), (R_4,3))\).

### 3.3. Random bag preservation

We introduce the central notion of random bag preservation, which implies the capacity for the tracking of S-distributions during the computations.

The reference to “randomness” preservation may lead one to believe that the aim is to preserve “chaos”. Quite to the contrary, the aim is to preserve a specific distribution of the original inputs and hence to impose a very particular structure on the outcomes of each computational step. As opposed to being chaotic, the randomness preservation which we will formalize is much more controlled, and aimed at preserving and tracking S-distributions.

We make the following assumption on data-labelings:

Two data-labelings \(F_1\) and \(F_2\) are distinct in case they differ when interpreted as a pair \((F_1, (X_1, \sqsubseteq_1))\) and a pair \((F_2, (X_2, \sqsubseteq_2))\), for which their underlying order is explicitly taken into account. I.e. two data-labelings differ in case

1. their underlying partial orders are distinct (i.e. set or partial order differ), or
2. in case the underlying partial orders are identical to a partial order \((X, \sqsubseteq)\) and \(\exists x \in X. F(x) \neq G(x)\).

Next, we introduce the notion of labeling-invariant function.

**Definition 10.** Consider a collection of data-labelings \(\mathcal{D}_C(X, \sqsubseteq)\) and a finite collection of pairwise distinct partial orders \((X_1, \sqsubseteq_1), \ldots, (X_n, \sqsubseteq_n)\). The function \(\Psi: \mathcal{D}_C(X, \sqsubseteq) \rightarrow \mathcal{D}_C(X_1, \sqsubseteq_1) \cup \cdots \cup \mathcal{D}_C(X_n, \sqsubseteq_n)\) is a refining function in case each of \((X_1, \sqsubseteq_1), \ldots, (X_n, \sqsubseteq_n)\) is a refinement of the partial order \((X, \sqsubseteq)\). A function \(\Psi\) is labeling-invariant iff

\[ \forall F_1, F_2 \in \mathcal{D}_C(X, \sqsubseteq). F_1 \approx F_2 \Rightarrow \exists i. \Psi(F_1), \Psi(F_2) \in \mathcal{D}_C(X_i, \sqsubseteq_i) \text{ and } \Psi(F_1) \approx \Psi(F_2). \]

**Definition 11.** A function \(\Psi: \mathcal{D}_C(X, \sqsubseteq) \rightarrow \mathcal{D}_C(X_1, \sqsubseteq_1) \cup \cdots \cup \mathcal{D}_C(X_n, \sqsubseteq_n)\) is random bag preserving if the following holds:

---

7 In gratitude to my parents Y. and Y. Schellekens.
(1) $\Psi$ is refining.
(2) $\Psi$ is labeling-invariant.
(3) If the input data-labelings, after identification up to labeling isomorphism form a random bag $R$, then the bag of data-labelings produced from $R$ by application of $\Psi$ yields, after identification up to labeling-isomorphism, a random bag $R'$.

If $\Psi$ is random bag preserving as above, then we denote this by:

$$\Psi: R \mapsto \rightarrow \circ R'.$$

Remark 12. Note that, using the above notation, the strict random bag corresponding to the random bag $R'$ must have the form $\{(R(X_1, \sqsubseteq_1), K_1), \ldots, (R(X_n, \sqsubseteq_n), K_n)\}$.

Lemma 13. If $\Psi: D_L(X, \sqsubseteq) \rightarrow D_L(X_1, \sqsubseteq_1) \cup \ldots \cup D_L(X_n, \sqsubseteq_n)$ and $\Psi: R \mapsto \rightarrow \circ R'$ then the bag consisting of the images of $\Psi$ over the elements of $D_L(X, \sqsubseteq)$ yields, after identification up to labeling-isomorphism, the random bag $R'$.

Proof. This follows from the fact that $\Psi$ is labeling-invariant. □

Remark 14. Random bag preserving functions can be defined over several data structures as opposed to over a single data structure, simply by requiring the function to be random bag preserving over each data structure. Hence in practice random bag preserving functions $\Psi$ will transform a random bag $R$ into a random bag $R'$, denoted by $\Psi R \mapsto \rightarrow \circ R'$.

For our purposes it will suffice to work with a sufficient condition which ensures random bag preservation as outlined in Section 3.6.

It is important to point out that random bag preservation does not necessarily hold in practice. The reader may safely omit the following counter example on first reading and revisit at a later stage. It is included to illustrate the necessity of guaranteeing randomness preservation.

3.4. The necessity of guaranteeing random bag preservation

To illustrate the need of guaranteeing random bag preservation, we consider the example of traditional Heapsort, which involves a non-randomness preserving “selection” part. The reader may, prior to reading this section, benefit from a brief look at the traditional Heapsort algorithm discussed in [26]. Heapsort consists of a Heapify phase and a Selection phase. The Heapify phase forms a heap out of any given list as specified in [26]. The Selection phase essentially amounts to a “delete”-style operation, even though elements are not actually removed, only ignored during the computation. The Selection phase proceeds as follows: it swaps a label at a specific leaf of the heap with the root label. It subsequently ignores the new leaf label, which is the maximum label, in the remainder of the program execution. The rest of the tree may no longer form a heap due to the swap operation. Hence Heapify is called again to create a heap out of the remaining tree and Selection is recursively called over the newly created heap.

Counter-Example 15 (Heapsort). Consider lists of size 4 where we assume that list elements are pairwise distinct. After the Heapify part, viewed over all twenty-four input states of size 4, a total of 3 non-label-isomorphic heap states are created which arise with equal probability of $\frac{1}{3}$. We display these heaps of size 4, which we denote, as in the states obtained for Example 6, by $H_4[1], H_4[2]$ and $H_4[3]$, for labels 1, 2, 3, 4:

However, this uniform distribution is violated by the Selection phase, which swaps the root label, i.e. the maximum label, with the left-most leaf label. After this phase, the algorithm focuses on the heaps of size 3, obtained by ignoring the left-most leaf. Since the resulting heaps are of size 3, consisting of a root and two leaves, there are two possible states. We display these states for labels 1, 2, 3 in the figure below, where the state displayed first is referred to as $H_3[1]$ and the state displayed below is referred to as $H_3[2]$. It is clearly impossible that these two states are created from the three states of size 4 with equal probability. Hence the random structure consisting of the two states, which represents uniform distribution, is not an
adequate representation of the true distribution. With the notion of random structure at our disposal, we remark that the states produced during this phase of the computation do not correspond to the two states of the underlying partial order. The two necessary states are produced, but an extra copy of one of these states is also produced. Hence the resulting bag of states does not form a random structure, nor of course a random bag. In fact, one can verify that $H_4[1]$ and $H_4[3]$ both are transformed to $H_3[2]$, while $H_4[2]$ is transformed to $H_3[1]$ during the execution of Heapsort, as displayed in the picture below.

![Diagram](image)

The argument to show that Heapsort does not preserve the uniform distribution of its data is based on an example discussed in [11], which makes an attempt to solve the open problem of designing a “randomness preserving” version of Heapsort. Edelkamp observes in this context that: “Diese Betrachtung hat eine exakte average-case Analyse von allen HEAPSORT-Varianten bis dato unmöglich gemacht”.8 To clarify the problem further, and minimizing formality, we remark that in order to obtain a compositional derivation of the average-time of Heapsort for arbitrary size $n$, one needs in particular to express this time for the case of size 4 as:

1. The average-time over all $4!$ list states of size 4
   (used by the Heapify procedure to create the heaps of size 4) +
2. The average-time over the 3 heap states of size 4
   (used by the first call to the Selection phase) +
3. The average time over the 2 hat-shaped heap states of size 3
   (used by an iterated call to the Heapify and Selection phase).

It is part (3) which cannot be used in practice to compute the average-time of the iterated call to the Heapify and Selection phase. In the standard average-case time approach of [26], the Selection phase operates on heaps and its average-case time needs to be computed over the possible states of the heaps of a given size, where heaps are assumed to occur equally likely in any of the two given states.

We recall that the two states produced for the hat-shaped partial order are not equally distributed. The first state $H_3[1]$ occurs once, while the second state $H_3[2]$ occurs twice, indicating that heaps are twice more likely to occur in the second state than in the first. Hence we can no longer express the average-time of the Selection phase in terms of the states of the heaps under consideration and one effectively loses the capacity to track the distribution of the data in question.

This prevents the generation of a recurrence equation (in terms of size) which expresses the average-case time. It is clear that the average-case analysis of Heapsort is notoriously hard due to the fact that Heapsort’s Selection process does not preserve randomness (cf. [26,52,57,11]) and similar problems arise for deletions and insertions in binary search trees. As pointed out by Knuth in [26], regarding the analysis of Heapsort: “But the selection phase is another story, which yet remains to be written!” The optimality of Heapsort’s average comparison time was demonstrated relatively recently by Schaffer and Sedgewick [52] via an argument by contradiction (cf. also [57]). This does not lend itself for static average-case analysis. The exact average comparison time for the (selection phase of this) algorithm and for any of its current variants, remained unknown [11]. The problematic nature of determining Heapsort’s Average-Case Time has also been pointed out in [14]. In [50] a new variant of Heapsort is presented, Percolating Heapsort, which does preserve randomness. This new algorithm allows for an elegant and straightforward analysis of its exact average-case time; which constitutes a radical simplification, in comparison with prior average-case analysis methods for Heapsort and its variants, as discussed in [49]. In [49], the analysis is achieved via a backward analysis and complications regarding full automation are discussed.

The lack of “randomness preservation” is also cited in [14] as preventing an automated average-case analysis of Heapsort.

Other random bag preserving operations such as the product operation have been introduced in [48].

---

8 “This fact (i.e. the non-preservation of the uniform distribution for size 4) has made an exact average-case analysis of all HEAPSORT-variants impossible to date.” [11] did not resolve the problem of producing a randomness preserving version of Heapsort. Such a version, called Percolating Heapsort, has been obtained in [50,49].
3.5. Links with reversibility

Reversibility traditionally plays a role in hardware design, with implications for low power design [28,2,54,53]. A few exceptions focus on high-level reversible languages, including the language JANUS9 and the work discussed in [58]. Reversibility refers to the fact that for any output it is possible to re-compute the input which gave rise to the output in question. This enables one to back-track a computation and “recycle” energy stored during the computation after completing the computation in question.

Reversibility typically relies on a one-to-one correspondence between inputs and outputs as is clear from reversible gates. Even though MOQA was not intended to be a reversible language, MOQA is close in spirit to reversible languages. Indeed, random bag preservation typically is established via a one-to-one correspondence between inputs and outputs of basic MOQA operations. This is evident from the product operation [48], which is “locally” one-to-one, under a natural condition outlined in [48] and can be verified to be one-to-one in case indices are tracked during computation, where swaps between labels result in a corresponding swap between indices of elements.

We remark at this stage that the MOQA operations which do not reduce the size of the data structure10 are intrinsically “reversible”. Reversibility is strongly linked to the one-to-one condition and we will focus on this aspect in formulating the sufficient condition for random bag preservation below. The general reversibility of MOQA will be addressed elsewhere.

It is interesting to note that with certain modifications MOQA can be viewed as a high level reversible language which transcends traditional applications of reversibility in the context of low power analysis, via its applicability in the general area of static average-case analysis.

3.6. A sufficient condition for random bag preservation

We now return to a sufficient condition for random bag preservation and an illustration of such an operation via the traditional SPLIT operation.

We formulate the notion of weakly-reversible functions. Such functions are called weakly-reversible since they only require to be bijective. In practice, proofs that MOQA operations typically will correspond to functions which are reversible, i.e. one can compute each input from a given output, which implies bijectivity. For our purposes it suffices to focus on weak-reversibility, which only requires bijectivity.

Definition 16. Consider a collection of data-labelings $D_L(X, \sqsubseteq)$ and a finite collection of pairwise distinct partial orders $(X_1, \sqsubseteq_1), \ldots, (X_n, \sqsubseteq_n)$. Consider a function $\Psi: D_L(X, \sqsubseteq) \rightarrow D_L(X_1, \sqsubseteq_1) \cup \ldots \cup D_L(X_n, \sqsubseteq_n)$. $\Psi$ is weakly-reversible iff

1. $\Psi$ is refining.
2. $\Psi$ is labeling-invariant.
3. $\Psi|\mathcal{R}(X, \sqsubseteq): \mathcal{R}(X, \sqsubseteq) \rightarrow \mathcal{R}(X_1, \sqsubseteq_1) \cup \ldots \cup \mathcal{R}(X_n, \sqsubseteq_n)$ is a bijection.

Remark 17. Weakly-reversible functions can be defined over several data structures as opposed to over a single data structure, simply by requiring the function to be weakly-reversible over each data structure.

Finally, we remark that if a function $\Psi$ is weakly-reversible over a data structure $(X, \sqsubseteq)$, it is guaranteed to give rise to a bag of output labelings which, after identification up to labeling-isomorphism, give exactly rise to a finite collection of random structures $\mathcal{R}(X_1, \sqsubseteq_1), \ldots, \mathcal{R}(X_n, \sqsubseteq_n)$.

This determines a random bag $((\mathcal{R}(X_1, \sqsubseteq_1), 1) \ldots ((\mathcal{R}(X_n, \sqsubseteq_n), 1))$, in which each random structure has multiplicity one.

Hence we have the following result.

Proposition 18. Weakly-Reversible functions are random bag preserving.

Note that not all random bag preserving functions are weakly-reversible.

We give sufficient conditions for random bag preserving functions to be weakly-reversible.

Proposition 19. A random bag preserving function $\Psi$ with domain $D_L(X, \sqsubseteq)$ and corresponding random structure $\mathcal{R}$ is weakly-reversible in case the bag of images of $\Psi$ over $\mathcal{R}$ is a set. Equivalently, a random bag preserving function is weakly-reversible in case $\Psi|\mathcal{R}$ is a bijection.

In practice we will perform an extra identification for each random bag preserving function as follows. The output random bag is subjected, as part of the algorithmic analysis, to identification up to order-isomorphism.

9 http://www.cise.ufl.edu/~mpf/rc/.

10 I.e. the operations excluding the delete and the projection operation.
For weakly-reversible functions, the random bag which has multiplicities constant one, gives rise to a random bag 
\{(R(X_{i_1}, X_{\leq i_1}), K_1), \ldots, (R(X_{i_k}, X_{\leq i_k}), K_k)\}, where of course \(\sum_{j=1}^{k} K_j = n\) and where multiplicities are not necessarily one.

We illustrate this via the \textit{SPLIT} example below. Note that all operations are stored in a \texttt{MOQA} library, joint with information on the random bag they produce (for arbitrary size \(n\)) as well as with the multiplicities in question, which for each operation have been determined in terms of \(n\) in advance. This information is then used by Distri-Track to extract the average-case information statically.

### 3.6.1. \textit{SPLIT}: an illustration of random bag preservation

\(S\)-distributions arise naturally, even if one starts with uniformly distributed data at the outset. One well-known operation which illustrates this effect is the \textit{SPLIT} operation used in algorithms such as Quicksort and Quickselect.

Rather than developing this immediately in full formality, i.e. for lists of arbitrary size \(n\), we first show that \textit{SPLIT} is random bag preserving for lists of size 3 and 4. The general case is treated in Section 4. We consider a simple version of the \textit{SPLIT} operation. Other versions of \textit{SPLIT}, such as those using two pointers starting at beginning and end of a list [1], result in a similar random bag. We use a simpler version to reduce the technicalities. The pivot for \textit{SPLIT} is chosen to be the first element of the list. This choice is again irrelevant. Other choices will result in similar random bags with minor technical modifications.

\textit{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, \(i \in \{2, \ldots, n\}\), with the pivot label. In case the label of the \(i\)-th element is greater than the pivot label, this element and its label is placed above the pivot. Otherwise it is placed below the pivot.

**Example 20.** We illustrate the effect of executing \textit{SPLIT} on lists of size 3.

\[ x_1 : 1, x_2 : 2, x_3 : 3 \quad x_1 : 1, x_2 : 3, x_3 : 2 \quad x_1 : 2, x_2 : 1, x_3 : 3 \]

\[ \begin{array}{ccc}
    \text{s} & \text{p} & \text{l} \\
    \downarrow & \downarrow & \downarrow \\
    t & l & t \\
\end{array} \]

\[ x_1 : 1 \quad x_2 : 2 \quad x_3 : 3 \]

\[ \begin{array}{ccc}
    \text{s} & \text{p} & \text{l} \\
    \downarrow & \downarrow & \downarrow \\
    t & l & t \\
\end{array} \]

\[ x_1 : 1 \quad x_1 : 1 \quad x_1 : 2 \]

\[ x_2 : 1 \quad x_2 : 1 \]

\[ x_3 : 3 \]

\[ x_3 : 2 \]

\[ x_3 : 2 \]

\[ x_3 : 3 \]

Example 20. We illustrate the effect of executing \textit{SPLIT} on lists of size 3.
It is clear from the above example that when \textit{SPLIT} is executed on the random structure over the discrete partial order of size 3, i.e. \(R(\Delta_3)\), where \textit{SPLIT} is executed over the \(3! = 6\) random lists, the result is a random bag consisting of three new random structures. The first random structure is the random structure over the 3 element V-shaped partial order, denoted in the following by \(\lor_3\). The second random structure and the third random structure are both the random structure over the linear order of size 3, denoted by \(S_3\). Though the elements of the two linear orders displayed above differ, we will identify these orders in our analysis up to order isomorphism. This means that we have 2 copies of the random structure over \(S_3\) in the random bag. Finally, we obtain the random structure over the 3 element wedge-shaped partial order, denoted by \(\land_3\).

In conclusion, \textit{SPLIT} transforms labelings over \(\Delta_3\) into a labeling over \(\lor_3, S_3\) or \(\land_3\). Moreover, it is clear from our example above that the input states correspond to \(R(\Delta_3)\), while the output labelings correspond exactly to the states in the random bag \(\{(R(\lor_3), 1), (R(S_3), 2), (R(\land_3), 1)\}\). We remark that \textit{SPLIT} is a labeling isomorphism as required and is a bijection between the random structure \(\Delta_3\) and the random bag \(\{(R(\lor_3, 1), (R(S_3), 2), (R(\land_3), 1)\}\). Hence \textit{SPLIT} is a random bag preserving operation over the random structure \(R(\Delta_3)\).

We remark at this stage that there is a clear visual nature to the partial orders associated with the random bag. Indeed, “star”-like objects are being created, with a center element, the pivot, and with in each case a collection of elements above the pivot and below the pivot.

For the case of \(\lor_3\) there are two elements above the pivot and zero below the pivot. For the case of \(S_3\) there is one element above the pivot and one element below the pivot. For the case of \(\land_3\) there are zero elements above the pivot and 2 elements below.

This can be generalized to \(n\) elements as follows.

The partial order \(P[i,j]\) over \(i+j+1\) elements is defined to be the order which has one central pivot element, \(i\) elements above the pivot and \(j\) elements below the pivot, as illustrated below.

The partial order \(P[i,j]\)

\[
\begin{array}{c}
\text{i elements} \\
\begin{array}{c}
\cdots \\
\uparrow \\
\cdots \\
\end{array}
\end{array}
\begin{array}{c}
\text{j elements} \\
\begin{array}{c}
\cdots \\
\downarrow \\
\cdots \\
\end{array}
\end{array}
\]

In general, the partial orders created by the \textit{SPLIT} operation, after identification up to order-isomorphism, are given by:

\(P[n-1,0], P[n-2,1], P[n-3,2], \ldots, P[2,n-3], P[1,n-2], P[0,n-1]\) as dismayed below.

\[
\begin{array}{ccccccc}
\begin{array}{c}
n-1 \\
\cdots \\
\end{array} & \begin{array}{c}
n-2 \\
\cdots \\
\end{array} & \begin{array}{c}
n-3 \\
\cdots \\
\end{array} & \begin{array}{c}
2 \\
\cdots \\
\end{array} & \begin{array}{c}
1 \\
\cdots \\
\end{array} & \begin{array}{c}
0 \\
\cdots \\
\end{array}
\end{array}
\]

Hence for lists of size 3 it is clear that \textit{SPLIT} transforms the random structure \(R(\Delta_3)\) into the random bag \(\{(R(P[0,2], 1), (R(P[1,1], 2), (R(P[2,0], 1))\}\).

\textbf{Example 21.} We illustrate that a similar result arises when \textit{SPLIT} is executed on lists of size 4. We record the effect of executing \textit{SPLIT} on the \(4! = 24\) states of size 4. First we remark that for the 6 states which have the pivot labeled with 1, \textit{SPLIT} will transform these 6 states exactly in to the 6 states of the partial order \(P[3,0]\) displayed below.

\[
\begin{array}{c}
\uparrow \\
\begin{array}{c}
\cdots \\
\downarrow \\
\cdots \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
0 \\
\cdots \\
1 \\
\cdots \\
2 \\
\cdots \\
n-3 \\
\cdots \\
n-2 \\
\cdots \\
n-1 \\
\cdots \\
\end{array}
\end{array}
\]
Secondly we remark that the 6 states which have the pivot labeled with 4, $SPLIT$ will transform these 6 states exactly in the 6 states of the partial order $P[0, 3]$ displayed below.

Hence it remains to look in the situation where the pivot is labeled with 2 or 3. The results for the pivot labeled with 2 are displayed below.

\[
\begin{align*}
    x_1 &: 2, x_2 &: 1, x_3 &: 3, x_4 &: 4 \\
    x_1 &: 2, x_2 &: 1, x_3 &: 4, x_4 &: 3 \\
    x_1 &: 2, x_2 &: 3, x_3 &: 1, x_4 &: 4
\end{align*}
\]

We remark that the first two states in the top row form a random structure. The same holds for the third state in the top row and the second state in the bottom row. Finally, the first and third state in the bottom row form again a random structure.

Clearly the partial orders of these three random structures are order-isomorphic. Hence we obtain three copies of the random structure over the partial order $P[2, 1]$ displayed below.

\[
\begin{align*}
    x_1 &: 2, x_2 &: 3, x_3 &: 4, x_4 &: 1 \\
    x_1 &: 2, x_2 &: 4, x_3 &: 1, x_4 &: 3 \\
    x_1 &: 2, x_2 &: 4, x_3 &: 3, x_4 &: 1
\end{align*}
\]

We remark that the first two states in the top row form a random structure. The same holds for the third state in the top row and the second state in the bottom row. Finally, the first and third state in the bottom row form again a random structure.

Clearly the partial orders of these three random structures are order-isomorphic. Hence we obtain three copies of the random structure over the partial order $P[2, 1]$ displayed below.

\[
\begin{align*}
    x_1 &: 2, x_2 &: 1, x_3 &: 3, x_4 &: 4 \\
    x_1 &: 2, x_2 &: 4, x_3 &: 1, x_4 &: 3 \\
    x_1 &: 2, x_2 &: 4, x_3 &: 3, x_4 &: 1
\end{align*}
\]

The result for the pivot labeled with 3 are displayed below.
We remark that the first and the third state in the top row together with the underlying partial order form a random structure. The same holds for the second state in the top row and the first state in the bottom row. Finally, the second and third state in the bottom row form again a random structure.

Clearly the partial orders of these three random structures are order-isomorphic. Hence we obtain three copies of the random structure over the partial order $P[1,2]$ displayed below.

Hence $\texttt{SPLIT}$ transforms $R(L_4)$ into the random bag

$$((R(P[3,0]),1), (R(P[2,1]),3), (R(P[1,2]),3), (R(P[0,3]),1)).$$

Again, this is the result of the fact that $\texttt{SPLIT}$ is a labeling isomorphism and hence a bijection from $R(\Delta_4)$ to the random structures over the 8 different partial orders as displayed above.

Regarding reversibility, we remark that it is perfectly possible to “un-compute” an output to create the original input, using combined information on labels and element indices. Consider for instance any of the labelings given above. It is clear that these simply record the positions of the various elements. Hence all one needs to do to regenerate the original input is to order the elements according to index order.

Our target is not un-computing in this context. The indices are carried around to illustrate that the operations are actually bijective in the sense clarified above.

It is interesting however to note that $\texttt{MOQA}$ can be interpreted as a high-level reversible language, supporting the modular derivation of the average-computation cost. Moreover, its reversibility is not trivial. Uncomputing the original input
is direct from the indices. However in general, un-computing in the context of a repeated execution of operations is non-trivial. E.g. a repeated execution of $\textit{MOQA}$ product operations still allows one to systematically reverse each step in the computation, determining step by step the previous input until one reaches the original input.

Finally, we observe that the multiplicities for $\textit{SPLIT}$ can be directly computed in terms of input size $n$ as outlined below.

### 3.6.2. $\textit{SPLIT}$: the general case

#### The random split of a discrete partial order

To determine the multiplicities one needs to simply remark that the general split operation, for input lists of size $n$, will produce, after identification up to order isomorphism, the partial orders $P[n-1,0], P[n-2,1], P[n-3,2], \ldots, P[2,n-3], P[1,n-2], P[0,n-1]$. 

Prior to identification up to order isomorphism, one can easily determine that the number of partial orders which are (non-identically) order isomorphic to $P[i,j]$, where $i,j \in \{0,1,\ldots,n-1\}$ and $i+j = n-1$, is $K_i = \binom{n-1}{i} = \binom{n-1}{j}$. For instance, consider Example 5. There are $\binom{3}{1} = 2$ copies of the (linear) partial order $P[1,1]$.

Hence $\textit{SPLIT}$ is a random bag preserving operation which maps the random structure $R(\Delta n)$ to the random bag $\{(R(P[n-1,0]), K_{n-1}), \ldots, (R(P[0,n-1]), K_0)\}$, i.e. we have the following result.

**Lemma 22.** $\textit{SPLIT}$ is random bag preserving, where

$$
\text{SPLIT}: R(\Delta n) \mapsto \{(R(P[n-1,0]), K_{n-1}), \ldots, (R(P[0,n-1]), K_0)\},
$$

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

#### 3.6.3. Tracking $S$-distributions in $\textit{MOQA}$

In our programming language $\textit{MOQA}$, the tracking of distributions is achieved by having a typed context in which types are the finite partial orders underlying the random structures (random bags) and for which each operation transforms a collection of types (paired with their multiplicities) into a new collection of types (paired with their multiplicities). Each operation is formally guaranteed to preserve random bags. As a result the types and the multiplicities of the data can be tracked during the entire computation.

This approach means that we do not need to determine the resulting random bags by computing all output states in a computation from all possible input states. It suffices to identify the operation in question and our operation rules supply, from the given partial orders (types) and multiplicities for input data, directly the new partial orders and multiplicities of the output data. This is feasible since $\textit{MOQA}$ operations are verified to be random bag preserving, where, for each such operation, a constructive definition is given of the transformation of the partial orders as well as formulas for computing the multiplicities.

Multiplicities are crucial since they enable the book-keeping of output copies during the computation in a modular way; which in turn is directly linked with the capacity to generate recurrence equations expressing the average-case number of basic operations in a compositional way. This last aspect is clarified via the Compositionality Theorem below.

The static analysis tool Distri-Track, developed at CEOL, statically extracts the average number of basic operations from $\textit{MOQA}$ code.

We outline the basic $\textit{MOQA}$ operations below.

### 3.7. $\textit{MOQA}$: operations

#### 3.7.1. An overview of the $\textit{MOQA}$ basic operations

We discuss four of the main random bag preserving $\textit{MOQA}$ operations below to give an idea of the nature of the language. The first two operations, the random product and the random delete, are core operations for creating and destroying data.

**The random product operation:** $\otimes$

This operation can play the role of an insertion of a single element into a data structure, in case one of the data structures provided consists of a single element. This operation also plays a crucial role whenever data structures are merged into a larger whole. See [48] for further details.

This binary operation places a first data structure below a second, where all elements of the first order are strictly below all elements of the second. This operation operates as follows:

- 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.

This operation results in a single new random structure.

**The random delete operation:** $\text{DEL}(k)$

This operation plays a crucial role in removing labels from a data structure. See [50,49] for further details on this operation.
This unary operation removes a label from a partial order by:

- redefining it to be the smallest label present,
- pushing down the label to restore order,
- removing the label together with the minimal element to which it has been pushed-down.

This operation results in a sequence of random structures, of a length which equals the number of minimal elements of the original order, and for which the underlying partial orders are obtained from the original partial order by the systematic removal of a single minimal element.

\( \text{MOQA} \) supports other random bag preserving operations such as the random projection operation \( \text{PROJ}(I) \), which enables the restriction of data-labelings to local parts, called “isolated parts” \( I \), of a given data structure, where an isolation property is verified on the suborder in question to ensure random bag preservation. Another operation is the random split operation \( \text{SPLIT} \). This operation plays a crucial role in reorganizing data quickly, e.g. in the context of Quicksort, Quickselect, etc. As outlined above, the operation functions on discrete data structures, selecting a pivot and reorganizing labels above or below the pivot depending on their respective order w.r.t. the pivot label. \( \text{MOQA} \) incorporates a number of other operations, such as \( \text{Top} \) and \( \text{Bot} \) to determine minimum and maximum labels, all of which are random bag preserving [49]. All \( \text{MOQA} \) operations can be applied locally to a data structure as determined by the notion of an “isolated suborder.” This will be discussed in more detail in [49]. The timing tool \( \text{Distri-Track} \) statically expresses the average-case number of comparisons of a \( \text{MOQA} \) program in terms of the average-case number of comparisons of the basic operations, exploiting compositionality.

### 3.7.2. Conditionals, loops, recursion

\( \text{MOQA} \) is equipped with conditionals, for-loops and a restricted type of recursion (guaranteed to terminate), all of which have been purposely designed to ensure the corresponding operations are random bag preserving. These higher level operations are discussed separately in [49].

### 3.8. Compositionality

As outlined in the abstract, one may easily be tempted to conclude that average-case time is automatically compositional. This stems from the fact that average-case time satisfies a simple type of compositionality, namely that of IO-compositionality as outlined below. We discuss this first type of compositionality. As discussed in Section 2.1.3, it is crucial to have the capacity to track the data distributions in order to fully exploit compositionality, which is the topic of Section 3.8.2.

#### 3.8.1. Average-case time is IO-compositional

**Theorem 23.** The average-time measure is universal IO-Compositional, i.e. the following equality holds for any language \( P \mathcal{L} \) and for any two programs \( P_1, P_2 \) of \( P \mathcal{L} \), where \( P_1 \) operates on a finite input bag \( I \) and produces the output bag \( O_{P_1}(I) \):

\[
T_{P_1,P_2}(I) = T_{P_1}(I) + T_{P_2}(O_{P_1}(I)).
\]

**Proof**

\[
T_{P_1,P_2}(I) = \frac{\sum_{I \in \mathcal{L}} T_{P_1,P_2}(I)}{|\mathcal{L}|} = \frac{\sum_{I \in \mathcal{L}} T_{P_1}(I) + \sum_{J \in O_{P_1}(I)} T_{P_2}(J)}{|\mathcal{L}|} = T_{P_1}(I) + T_{P_2}(O_{P_1}(I)),
\]

where the last equality follows from the fact that \( |\mathcal{L}| = |O_{P_1}(I)| \) (cf. Section 1).

#### 3.8.2. Linear-Compositionality Theorem

The Linear-Compositionality Theorem states two facts. First, the average time of the sequential composition of two random bag preserving programs can be expressed as the sum of the average times of the programs. Secondly, the average time of a random bag preserving program on a random bag is a linear combination of the average times over the random structures of the random bag. The linear coefficients correspond to the probabilities involved. For completeness, we include the technical definition and the theorem.

**Definition 24.** A random bag preserving program \( P \) is a program for which the collection of input data-labelings, after identification up to labeling-isomorphism (i.e. up to identical relative order), forms a random bag. Moreover, the input-output function of the program, denoted by \( \|P\| \), is random bag preserving over the input random bag.

**Theorem 25** (Compositionality).

1. Consider a random bag preserving program \( P \) such that \( \|P\| : R \rightarrow R' \). Then:

\[
T_{P,Q}(R) = T_P(R) + T_Q(R').
\]
2. Consider a random bag \( R = ((R_1, K_1), \ldots, (R_p, K_p)) \); then

\[
T_p(R) = \sum_{i=1}^{i=p} \text{Prob}_i \times T_p(R_i),
\]

where \( \text{Prob}_i \) is the S-probability, i.e. \( \text{Prob}[F \in R_i] \).

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

\[
T_p(R) = T_p(R_1).
\]

Proof. (1) follows directly from Theorem 23.

(2) (a) follows since

\[
\sum_{F \in R} T_p(F) = T_p(R_i) \times |R_i| \text{ and } |R| = \sum_{i=1}^{i=p} K_i \times |R_i|.
\]

Thus:

\[
T_p(R) = \sum_{F \in R} T_p(F) = \sum_{i=1}^{i=p} K_i \sum_{F \in R_i} T_p(F) = \sum_{i=1}^{i=p} K_i \times |R_i| \times T_p(R_i) = \sum_{i=1}^{i=p} \text{Prob}_i \times T_p(R_i).
\]

(2)(b) follows from 2(a). \( \square \)

The systematic application of this result on the sequential parts of \( \mathcal{MQQA} \) code yields a recurrence equation expressing the exact average-case number of comparisons in terms of input size.\(^{11}\) Note that this enables a determination of the exact average-case time as opposed to only asymptotic information.

\( \mathcal{MQQA} \) allows one to statically extract recurrence equations for the average-case time from \( \mathcal{MQQA} \) source code in a modular fashion, via the timing tool Distri-Track [20]. After this generation, standard approaches can be followed to either completely solve the recurrence equation through generating functions and a mathematical software package such as Maple or Mathematica, or to obtain information on the ACET for inputs within a given size bound through computing the recurrence via dynamic programming in a fast and effective way. Of course one can also derive asymptotic time information from the recurrence equations.

4. Application: compositional analysis of \( \mathcal{MQQA} \) implementation of Quicksort

The following result is immediate since \( \text{SPLIT} \) carries out \( n - 1 \) comparisons on any data-labeling over \( \Delta_n \), the discrete partial order of size \( n \).

Lemma 26. \( T_{\text{SPLIT}}(n) = n - 1 \).

We introduce some notation regarding \( \text{SPLIT} \). As pointed out earlier, the \( \text{SPLIT} \) operation when executed on labelings over the discrete order of size \( n \), produces a random bag for which each partial order is order-isomorphic to one of the orders \( P[n - 1, 0], P[n - 2, 1], P[n - 3, 2], \ldots, P[2, n - 3], P[1, n - 2], P[0, n - 1] \). We recall that each of these orders has a "star-shape", displayed below.

\[
\begin{array}{ccccccc}
n - 1 & n - 2 & n - 3 & 2 & 1 & 0 \\
\end{array}
\]

\( ^{11} \) Under the assumption of distinct labels.
Star-shaped orders are defined to be partial orders which are order-isomorphic to some \( P(i, j) \). Star-shaped partial orders have the following form: a central element (the pivot element), an upper part, \( I_{UPPER} \), consisting of the \( i \) elements above the pivot and a lower part, \( I_{LOWER} \), consisting of the \( j \) elements below the pivot. Note that for the star-shaped orders which are order-isomorphic to \( P[n - 1, 0] \) or \( P[0, n - 1] \), the respective \( I_{LOWER} \) and \( I_{UPPER} \) parts are defined to be empty.

Quicksort will, following a call to \( \text{SPLIT} \) recursively call itself on the restriction of the data-labelings to the \( I_{UPPER} \) and the \( I_{LOWER} \) part of the star-shaped partial orders.

This is an example of an application of an operation to labelings which are the restrictions of labelings to so-called “isolated suborders” of the partial order. Rather than defining this notion in full generality, we point out here that isolated suborders are suborders for which the restriction of the labelings to the suborder forms (multiple copies of) a random structure.

We show that the restriction of data-labelings produced by \( \text{SPLIT} \) to \( I_{UPPER} \) and \( I_{LOWER} \) yield random bags and we determine the multiplicities involved.

**Lemma 27.** Consider labelings over a star-shaped partial order \((X, \sqsubseteq)\), which is order-isomorphic to \( P[i, j] \). The restrictions of data-labelings \( F \) of \( \Delta(X, \sqsubseteq) \) to \( I_{UPPER} \), \( F|_{I_{UPPER}} \), form a random bag \((\{R(\Delta_i), j!\})\). Similarly the restrictions of these data-labelings \( F|_{I_{LOWER}} \) form the random bag \((\{R(\Delta_j), i!\})\).

**Proof.** We consider representatives of the equivalence classes in \( R(X, \sqsubseteq) \), the data-labelings which take labels from the set \([n] = \{1, \ldots, n\} \). Hence, for each such \( \Delta \), \( F \) of \( R(X, \sqsubseteq) \) is isomorphic to \( P[i, j] \). \( F \) must have the label \( j + 1 \) as pivot label, the set \([1, \ldots, j]\) as labels of the \( I_{LOWER} \) collection and the set \([j + 2, \ldots, n]\) as labels of the \( I_{UPPER} \) collection.

Clearly, for each fixed labeling of the \( I_{LOWER} \) collection, there are exactly \( i! \) labelings of the upper collection, forming exactly \( j! \) copies of the random structure \( R(\Delta_i) \).

4.1. **MDQA-code for Quicksort**

In the code for Quicksort, the variable \( X \) stores the input data-labelings. The variable \( Y_1 \) stores the parts of the data-labeling produced by \( \text{Split}(X) \) restricted to \( I_{UPPER} \) and the variable \( Y_2 \) stores the part of the data-labeling restricted to \( I_{LOWER} \).

**Quicksort**

\[ n = 0 \text{ or } n = 1: \text{Quicksort}(X) = X, \text{ and for } n \geq 2: \]
\[ \text{Quicksort}(X) = [\text{SPLIT}(X); \text{Quicksort}(Y_1); \text{Quicksort}(Y_2)] \]

4.2. **Compositional average-case analysis of Quicksort**

Since \( \text{SPLIT} \) is random bag preserving we obtain, by Lemma 22, that

\[ \text{SPLIT}: R(\Delta_n) \mapsto (R(P[n - 1, 0]), K_{n-1}), \ldots, (R(P[0, n - 1]), K_0), \]

where \( K_i = \binom{n-i}{J} \), \( j = n - 1 - i \).

Moreover, by Lemma 27 the labelings stored in the variable \( Y_1 \) correspond to the random bag \((\{R(\Delta_i), L_j\})\) and the labelings stored in \( Y_2 \) correspond to the random bag \((\{R(\Delta_j), L_i\})\), where \( L_i = i! \) and \( L_j = j! \).

So

\[ \text{SPLIT}: R(\Delta_n) \mapsto ((R(\Delta_{n-1}, M_{n-1}), \ldots, (R(\Delta_0, M_0)) \]
\[ \text{SPLIT}: R(\Delta_n) \mapsto ((R(\Delta_0, N_0), \ldots, (R(\Delta_{n-1}), N_{n-1})), \]

where \( \forall i \in [n - 1, \ldots, 0] \) and \( j \in [0, \ldots, n - 1] \), we obtain: \( M_i = K_i L_j = \binom{n-1}{j}! = \frac{(n-1)!}{j!} \) and \( N_j = K_i L_i = \binom{n-1}{i}! = \frac{(n-1)!}{i!} \). Hence \( \forall i \in [n - 1, \ldots, 0] \), \( M_i = N_i \) and an application of the Compositionality Theorem yields:

\[ T_{\text{Quicksort}}(R(\Delta_n)) = T_{\text{SPLIT}}(n) + T_{\text{Quicksort}}((R(\Delta_0), M_0), \ldots, (R(\Delta_{n-1}), M_{n-1}))) \]
\[ + T_{\text{Quicksort}}((R(\Delta_{n-1}), M_{n-1}), \ldots, (R(\Delta_0, M_0)))). \]

Via Lemma 26 and Theorem 25 we obtain:

\[ T_{\text{Quicksort}}(R(\Delta_n)) = (n - 1) + 2 \sum_{i=0}^{n-1} a_i \times T_{\text{Quicksort}}(R(\Delta_i)), \]

\[^{12} \text{Cf.} [49] \text{ for a complete discussion.} \]
where \( \forall i \in \{0, \ldots, n-1\} \), \( \alpha_i = \frac{M_i \times |R_\Delta|}{|\Delta|} \). Note that \( M_i = \binom{n-1}{i} \times (n - i - 1)! = \frac{|R_\Delta i|}{n^i \times |\Delta|} \). Hence \( \forall i \in \{0, \ldots, n-1\} \), \( \alpha_i = \frac{1}{i} \) and thus, replacing the random structures \( R_\Delta i \) by their size \( i \):

\[
T_{\text{Quicksort}}(n) = (n - 1) + \frac{2}{n} \times \sum_{i=0}^{n-1} T_{\text{Quicksort}}(i).
\]

This recurrence equation is identical to the recurrence equation expressing the comparison time for Quicksort as obtained in standard algorithms textbooks and can be solved via traditional means. We sketch the approach: we let \( g(n) = T_{\text{Quicksort}}(n) \). Then:

\[
g(n) = (n - 1) + \frac{2}{n} \sum_{i=0}^{n-1} g(i).
\]

To solve this recurrence, one can show that \( ng(n) - (n-1)g(n-1) = 2(n-1) \) and, dividing by \( n(n + 1) \) yields:

\[
\frac{g(n)}{n+1} - \frac{g(n-1)}{n} = \frac{2(n-1)}{n(n+1)}.
\]

Finally, let \( \Psi(n) = \frac{gm}{n+1} \). Then

\[
\Psi(n) = \Psi(n + 1) + \frac{2n - 2}{n(n+1)}
\]

and thus\(^{13} \)

\[
\Psi(n) = \sum_{i=1}^{n} \frac{2i - 2}{n+1} \approx \sum_{i=1}^{n} \frac{2}{i} \approx 2\ln(n).
\]

Hence \( g(n) \approx 2\ln(n) + 1 \), i.e.

\[
(*) T_{\text{Quicksort}}(n) \approx 2(n+1)\ln(n).
\]

5. Conclusion

The paper provides a new foundation for the Analysis of Algorithms based on the notion of random bag preservation. The method involves the establishment of the connection between random bag preservation and compositional average-case analysis. The constructive tracking of data distributions for random bag preserving programs enables the static compositional derivation of the average-case time of MOQA programs, as illustrated by the application of the method to Quicksort. Relations with reversible computing have been indicated, including the potential of MOQA to serve as a high-level reversible programming language. The paper speculated that the use of MOQA as a high level reversible language could bring a new type of application to the area of reversible computing, namely that of modular static average-case analysis. This will be further explored elsewhere. The visual nature of the timing method based on the tracking of data structures has been illustrated as well on the particular example of Quicksort.

References


\(^{13}\) Using the harmonic numbers approximation: \( \sum_{i=1}^{n} \frac{1}{i} = \ln(n) \).


M.P. Schellekens, Quantitative Domain Theory, ERCIM News No. 50, July 2002.


