Frugal Encoding in Reversible \textit{MOQA}: A Case Study for Quicksort

Diarmuid Early, Ang Gao, and Michel Schellekens

Centre for Efficiency Oriented Languages
University College Cork
Ireland*
{a.gao,m.schellekens}@cs.ucc.ie

Abstract. \textit{MOQA} is a high-level data structuring language, designed to allow for modular static timing analysis. In essence, \textit{MOQA} allows the programmer to determine the average running time of a broad class of programmes directly from the code in a (semi-)automated way. The \textit{MOQA} language has the property of randomness preservation which means that applying any operation to a random structure, results in an output isomorphic to one or more random structures, which is the key to systematic timing. The language, its implementation and the associated timing tool have been reported on in the literature. Randomness preservation is key in ensuring modular timing derivation. A degree of reversibility in turn is a key aspect of ensuring randomness preservation. All operations of the \textit{MOQA} language can be made reversible with minimal additional bookkeeping. A challenge in achieving this encoding in a frugal way is to ensure subsets of data can be stored without excessive overheads. The paper focuses on illustrating such an encoding for the case of the Quicksort algorithm. Similar encodings are explored to ensure efficient reversibility of all \textit{MOQA} operations. The paper is self contained, i.e. no prior knowledge of the \textit{MOQA} language is needed to follow the encoding argument. We show how to efficiently encode the information needed to reverse the split of a list into two sublists. The code for reversible Quicksort is provided and an example illustrates the algorithm’s reverse execution.

Keywords: Reversible computing, Encoding, Algorithms, Quicksort, Sorting, Data structures, Partial orders, Random Structures, Time analysis, \textit{MOQA} language.

1 Introduction

\cite{Sch08} introduces \textit{MOQA}, a high-level data structuring language, designed to allow for modular static timing analysis. In essence, \textit{MOQA} allows the programmer to determine the average running time of a broad class of programs.

* This work was supported by the Science Foundation Ireland (SFI) Grant under Grant number: 07/IN.1/I977.
directly from the code in a (semi-)automated way. For further discussion of the mechanics of the process, see [Sch08 Hic08 SHB04]. As pointed out in [Sch08], the modularity 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. Modularity also improves precision of static average-case analysis, supporting the determination of accurate estimates on the average number of basic operations of MOQA programs. Reversible MOQA discussed in [Sch10] and [Ear10] complements traditional applications of reversibility with a new application domain, that of average-case cost analysis (where cost can be running time or power usage) of reversible MOQA programs. We provide here the frugal encoding for the reversible MOQA split operation and illustrate the approach via a reversible version of the well-known Quicksort algorithm.

Reversibility traditionally plays a role in hardware design, with implications for low power design [Lan61, Ben73, Tof80]. A few exceptions focus on high-level reversible languages, including the language JANUS and the work discussed in [YG07]. Most reversible approaches remained at hardware level. As observed, the use of MOQA as a high level reversible language brings a new type of application to the area of reversible computing. As pointed out in [Sch08] a sufficient condition for algorithms to be analyzable in a modular way is that they are random bag preserving. Not all algorithms are random bag preserving though, a case in point being the traditional heapsort algorithm [Sch08]. As shown in [Sch10], random bag preservation can typically be guaranteed by ensuring a “locally” one-to-one mapping, e.g. a mapping guaranteed to be one-to-one on each of the parts of a partition of the input. MOQA’s random bag preserving programs are ensured to allow for a greatly simplified average-case analysis. The key to understanding MOQA as a new application domain for reversible computing is that its programs, with little additional book-keeping become fully reversible [SEPV09 Sch10 Ear10]. Hence we establish a link between reversibility and the capacity for modular (i.e. semi-automated) average-case analysis. Of course, general algorithms typically can be subjected to average-case analysis techniques. The key point is that some algorithms, like heapsort, are not random bag preserving and hence either require complicated (non-automatable) techniques or escape average-case analysis by current techniques. As a degree of reversibility lies at the heart of random bag preservation [Sch10], reversibility has the potential to play a fundamental role in the design of modularly predictable algorithms. Since with a little more bookkeeping, MOQA becomes a fully reversible language, the exploration of its reversible properties is worthwhile, in particular since the reversible programs in turn allow for an exact prediction of average-case computation time. Hence we can predict in a static way, the cost of computing forward and backward in the language.

1 For example, the MOQA product operation as discussed in [Sch08], Theorem 5.1.