Theoretical Computer Science 562 (2015) 129–145

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Memoryless computation: New results, constructions, and extensions

Maximilien Gadouleau [a,*], Søren Riis [b]

[a] School of Engineering and Computing Sciences, Durham University, South Road, Durham DH1 3LE, UK
[b] School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London E1 4NS, UK

ARTICLE INFO

Article history:
Received 8 March 2013
Received in revised form 22 August 2014
Accepted 18 September 2014
Available online 28 September 2014
Communicated by G. Ausiello

Keywords:
Memoryless computation
Models of computation
Computational difficulty
Symmetric group
Theory of data
Combinatorics

ABSTRACT

In this paper, we are interested in memoryless computation, a modern paradigm to compute functions which generalises the famous XOR swap algorithm to exchange the contents of two variables without using a buffer. In memoryless computation, programs are only allowed to update one variable at a time. We first consider programs which do not use any memory. We study the maximum and average number of updates required to compute functions without memory. We then derive the exact number of instructions required to compute any manipulation of variables. This shows that combining variables, instead of simply moving them around, not only allows for memoryless programs, but also yields shorter programs. Second, we show that allowing programs to use memory is also incorporated in the memoryless computation framework. We then quantify the gains obtained by using memory: this leads to shorter programs and allows us to use only binary instructions, which is not sufficient in general when no memory is used.

1. Introduction

How do you swap the contents of two Boolean variables x and y by updating one variable at a time? The common approach is to use a buffer t, and to do as follows (using pseudo-code).

$$t \leftarrow x$$
$$x \leftarrow y$$
$$y \leftarrow t.$$

However, a famous programmer's trick consists in using XOR, which can be viewed as addition over a binary vector space:

$$x \leftarrow x + y$$
$$y \leftarrow x + y$$
$$x \leftarrow x + y.$$

The swap can thus be performed without any buffer. The aim is to generalise this idea to compute any possible function without additional memory.

* Corresponding author.
E-mail addresses: m.r.gadouleau@durham.ac.uk (M. Gadouleau), smriis@eecs.qmul.ac.uk (S. Riis).

http://dx.doi.org/10.1016/j.tcs.2014.09.040
0304-3975/

Memoryless computation (MC)—referred to as closed iterative calculus in [4] and *in situ* programs or computation with no memory in [7]—is a modern paradigm for computing functions, which offers two main innovations. The first introduces a completely different view on how to compute functions. The basic example is the XOR swap described above. Unlike traditional computing, which views the registers as "black boxes," MC takes advantage of the nature of the information contained in those registers and combines the values of the different registers. Thus, it can be seen as the computing analogue of Network Coding, a revolutionary technique to transmit data through a network which lets the intermediate nodes combine the messages they receive [24]. In particular, the XOR swap is the analogue of the canonical example of Network Coding, the so-called butterfly network [1].

The second main innovation lies in the computational model used for MC, which can be briefly described as follows. A processing unit has n registers $x_1, \ldots, x_n$ containing data over a finite alphabet A and has to compute a function $f : A^n \to A^n$ which possibly modifies the values of all registers. It is allowed any updates which only modify one register at a time (i.e., $x_i \leftarrow g(x_1, \ldots, x_n)$ for some $g : A^n \to A$), which are called *instructions*. A sequence of instructions computing a given function is a *program* for that function. Because an instruction is viewed as a quantum of complexity (akin to a clock cycle), the (memoryless) *complexity* of a function is defined as the minimum length of a program computing that function. For instance, the complexity of the swap of two bits is equal to three instructions.

Although MC is still mainly treated from a theoretical point of view, it has a wide range of possible long term applications, especially for computationally expensive problems. It can already be shown to offer several advantages over traditional computing. First, MC offers a computational speed-up at the core level. Indeed, MC yields arbitrarily shorter programs than traditional computing when manipulating variables (see Corollary 2 for more details). Secondly, MC does not rely on additional buffers and hence performs computations in line. Memory management is a tedious task which can significantly slow down computations [20] by bringing a significant overhead. This problem is particularly important for parallel architectures with shared memory [20]. Instead, MC uses no data memory and thus eases concurrent execution of different tasks by preventing memory conflicts.

While the XOR swap described above is folklore, MC was developed by Burckel et al. in [4,5,8–10,6] and more extensively in [7]. It is notably proved that any function can be computed without memory. Moreover, only $2n - 1$ instructions are needed to compute any bijective function $f : A^n \to A^n$; any function $f : A^n \to A^n$ can be computed in only $4n - 3$ instructions. A survey of these results and a striking relation between MC and switching networks [21] and is provided in [7].

We would like to emphasize the novelty of the results of this paper and how they differ from those in the literature. First, many aspects considered in this paper are completely novel. These include the study of the average memoryless complexity in Sections 3.1 and 3.3, the study of manipulations of variables in Section 4, the use of binary instructions in Theorem 6 and the use of additional registers in Section 5. Moreover, some of the results presented in this paper extend or generalise some of those given in the literature. For instance, by extending a key lemma in [7], we manage to extend the flexible approach to constructing programs of length $4n - 3$ introduced in [6] and [7, Section 5] for the Boolean case to any alphabet, thus answering part of Open problem 2 in [7]. Other results provide some matching upper and lower bounds which are absent in the literature, e.g. in Theorem 2. Finally, we also provide an alternative and much shorter proof to the seminal Theorem 1, which states that any function can be computed without memory.

The rest of the paper is organised as follows. Section 2 reviews the memoryless computation model and proves that it is universal: any function can be computed without memory. Section 3 then investigates the number of updates required to compute any function. Section 4 determines the complexity of manipulating variables without memory and shows that memoryless computation yields shorter programs than traditional methods. Section 5 finally proves that additional registers (or memory) can be added into the memoryless computation model without loss of generality.

2. Model for memoryless computation

2.1. Instructions and programs

We first review the model for memoryless computation introduced in [4] and subsequently developed in [5,8–10,6,7] and surveyed in [7].

Let A be a finite set, referred to as the *alphabet*, of cardinality q and let n be a positive integer (without loss, we shall usually regard A as $\mathbb{Z}_q$ or $\mathrm{GF}(q)$ when q is a prime power). We refer to any element of A^n as a *state*. We view any transformation f of A^n (i.e., any function $f : A^n \to A^n$) as a tuple of functions $f = (f_1, \ldots, f_n)$, where $f_i : A^n \to A$ is referred to as the i-th coordinate function of f. In particular, a coordinate function is *trivial* if it is equal to the identity, i.e. $f_i(x) = x_i$; it is nontrivial otherwise. When considering a sequence of transformations, we shall use superscripts, e.g. $f^k : A^n \to A^n$ for all k—and hence f^k shall never mean taking f to the power k.

Definition 1 *(Instruction).* An *instruction* is a transformation g of A^n with at most one nontrivial coordinate function g_i. If g is not the identity, we say that the instruction *updates* y_i for $y = (y_1, \ldots, y_n) \in A^n$ and we denote it as

$$y_i \leftarrow g_i(y).$$

A permutation instruction is an instruction which maps A^n bijectively onto A^n (i.e. is a permutation of A^n).