Embracing the Laws of Physics:

Three Reversible Models of Computation

Jacques Carette Roshan P. James Amr Sabry

McMaster University Google Indiana University

July 7, 2018

Abstract

Our main models of computation (the Turing Machine and the RAM) and most modern computer architectures make fundamental assumptions about which primitive operations are realizable on a physical computing device. The consensus is that these primitive operations include logical operations like conjunction, disjunction and negation, as well as reading and writing to a large collection of memory locations. This perspective conforms to a macro-level view of physics and indeed these operations are realizable using macro-level devices involving thousands of electrons. This point of view is however incompatible with computation realized using quantum devices or analyzed using elementary thermodynamics as both these fundamental physical theories imply that information is a conserved quantity of physical processes and hence of primitive computational operations.

Our aim is to redevelop foundational computational models in a way that embraces the principle of conservation of information. We first define what information is and what its conservation means in a computational setting. We emphasize the idea that computations must be reversible transformations on data. One can think of data as modeled using topological spaces and programs as modeled by reversible deformations of these spaces. We then illustrate this idea using three notions of data and their associated reversible computational models. The first instance only assumes unstructured finite data, i.e., discrete topological spaces. The corresponding notion of reversible computation is that of permutations. We show how this simple model subsumes conventional computations on finite sets. We then consider a modern structured notion of data based on the Curry-Howard correspondence between logic and type theory. We develop the corresponding notion of reversible deformations using a sound and complete programming language for witnessing type isomorphisms and proof terms for commutative semirings. We then “move up a level” to examine spaces that treat programs as data, which is a crucial notion for any universal model of computation. To derive the corresponding notion of reversible programs between programs, i.e., reversible program equivalences, we look at the “higher dimensional” analog to commutative semirings: symmetric rig groupoids. The coherence laws for these groupoids turn out to be exactly the sound and complete reversible program equivalences we seek.

We conclude with some possible generalizations inspired by homotopy type theory and survey several open directions for further research.

* Reversibility, the Missing Principle

What kind of operations can computers perform? This question has been answered several times in the last hundred years, where each answer proposes an abstract model of computation that specifies allowable operations and (usually) their cost. The emerging consensus, reflected in both early models of computations such as the Turing Machine and the RAM as well as in the early Von Neumann models and in modern computer architectures, is that basic computer operations include logical operations like conjunction, disjunction, and negation, as well as reading from and writing to a large (infinite) collection of memory locations. From this small set of primitive operations emerges all higher-level programming languages and abstractions.

1

No doubt, this consensus on the available primitive physical operations has been successful. And these operations can indeed be performed on a computer. Yet, today, with a possible quantum computing revolution in sight and with unprecedented explosion in embedded computers and cyber-physical systems, there are reasons to re-think this foundational question again. In fact, the calls to re-think this foundational question have been proclaimed by physicists almost forty years ago as the following two quotes testify:

Toﬀoli 1980 [1]: Mathematical models of computation are abstract constructions, by their nature unfettered by physical laws. However, if these models are to give indications that are relevant to concrete computing, they must somehow capture, albeit in a selective and stylized way, certain general physical restrictions to which all concrete computing processes are subjected.

Feynman 1982 [2]: Another thing that has been suggested early was that natural laws are reversible, but that computer rules are not. But this turned out to be false; the computer rules can be reversible, and it has been a very, very useful thing to notice and to discover that. This is a place where the relationship of physics and computation has turned itself the other way and told us something about the possibilities of computation. So this is an interesting subject because it tells us something about computer rules.

These quotes by Toﬀoli and Feynman both highlight the consequences of two obvious observations: (i) all the operations that a computer performs reduce to basic physical operations; and (ii) there is a mismatch between the logical operations of a typical model of computation (which are logically irreversible) and the fundamental laws of physics (which are reversible). One could certainly dismiss the mismatch as irrelevant to the practice of computing but our thesis is that the next computing revolution is likely to be founded on revised models of computation that are designed to be in closer harmony with the laws of physics.

After a detailed introduction on the origins of “logically reversibile computer operations” and an excursion into the origins of “irreversible computer operations,” we will develop three reversible models of computation and discuss their potential applications.

Maxwell’s Daemon. To fully appreciate the missing principle of “reversibility” in conventional computing, we go back to an old thought experiment by J. C. Maxwell. The details are codified in a letter that Maxwell wrote to P. G Tait in 1867 – the letter, whose ideas are now known as Maxwell’s Daemon, tells of a thought experiment that seems to indicate that intelligent beings can somehow violate the second law of thermodynamics, thereby violating physics itself. Many resolutions were oﬀered for this conundrum (for a compilation, see the book by Leﬀ and Rex [3]), but none withstood careful scrutiny until the establishment of Landauer’s Principle in 1961 [4] – a principle whose experimental validation happened in 2012 [5].

Maxwell’s Daemon appears to violate the second law of thermodynamics by having a tiny “intelligence” observing the movement of individual particles of a gas and separating fast moving particles from slow moving ones, thereby reducing the total entropy of the system. Landauer’s resolution of the daemon relied on two ideas that took root only a few decades earlier: the formal notion of computation (through the work of Turing [6], Church [7], and others) and the formal notion of information (through the work of Shannon [8]). Landauer reasoned that the computation done by the finite brain of the daemon involves getting information about the movement of molecules, storing that information, analyzing that information to act on it, and then

— and this is the critical step — overwriting it to make room for the next computation. In other words, the computation that is manipulating information in the daemon’s brain must be thermodynamic work, thereby bringing the daemon back into the fold of physics.

This is a strange and wonderful idea: information, physics, and computation are inextricably linked. In contrast, when the early models of computation were developed, there was no compelling reason to take the information content of computations into consideration – in fact, at that time there was no quantifiable notion of information. These models followed in the footsteps of logic where, following hundreds of years of tradition, the truth of a statement was seen as “absolute” and independent of any reasoning, understanding, or action. Statements were either true or false with no regard to any “observer” and the idea that statements had information content that should be preserved was outside the classical understanding of logic. Hence

2

the fact that conventional logic operations such as conjunction and disjunction were logically irreversible and hence lose information was not a concern. Landauer’s observation implied however that ideas in each field have consequences for the other [9, 10, 11, 12, 13, 14, 15]. To really appreciate this fact, we delve deeper into the origin of our computational models and argue that they are essentially reflections of contemporary laws of physics.

Origins of Computational Models. Current high-level programming languages as well as current hard-ware are both based on the mathematical formalization of logic developed by De Morgan, Venn, Boole, and Peirce in the mid to late 1800s. Going back to Boole’s 1853 book *entitled An Investigation of the Laws of Thought*, on which are Founded the Mathematical Theories of Logic and Probabilities, we find that the opening sentence of Ch. 1 is:

The design of the following treatise is to investigate the fundamental laws of those operations of the mind by which reasoning is performed;

which clearly identifies the “source” of the logical laws as mirroring Boole’s understanding of human reasoning.

A few chapters later, we find:

Proposition IV. That axiom of metaphysicians which is termed the principle of contradiction, and which aﬃrms that it is impossible for any being to possess a quality, and at the same time not to possess it, is a consequence of the fundamental law of thought, whose expression is x2 = x.

This “law” is reasonable in a classical world but is violated by the postulates of quantum mechanics. Although a detailed historical analysis of Boole’s ideas in the light of modern physics is beyond our scope, the above quotes should convey the idea that our elementary computing notions date back to ideas that were thought reasonable in the late 1800s.

Machines that “compute” are quite old. Müller (1786) first conceived of the idea of a “diﬀerence machine,” which Babbage (1819–1822) was able to construct. There are other computer precursors as well – the first stored programs were actually for looms, most notably those of Bouchon (1725) which operated on a paper tape, and Jacquard (1804) which operated by chains of punched cards. But it was only in the 20th century that computer science emerged as a formal discipline. One of the pioneering works was Alan Turing’s seminal paper [6] 1936 which established the idea that computation has a formal interpretation and that all computability can be captured within a formal system. Implicit in this achievement however is the idea that abstract models of computation are just that – abstractions of computation realized in the physical world. Indeed, going back to Turing’s 1936 article On Computable Numbers, with an Application to the Entscheidungsproblem,, the opening sentence of Sec. 1 is:

We have said that the computable numbers are those whose decimals are calculable by finite means [. . . ] the justification lies in the fact that the human memory is necessarily limited.

In Sec. 9, we find:

I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount.

It is worth noting that these assumptions are both physical (on distances) and metaphysical (on restrictions of the mind). If we take the human mind to be a physical “machine” which performs computation, then when both of the above assumptions are translated into the language of physics, they embody what is known as the “Bekenstein bound” [16], which is an upper limit on the amount of information that can be contained within a given finite region of space. A detailed historical account of these ideas in the context of modern physics is again beyond our scope. However, the quotes above, like the ones before, should convey the ideas that our theories of computation and complexity are based on some physical assumptions that Turing and others found reasonable in the 1930s.

To summarize, a major achievement of computer science has been the development of abstract models of computation that shield the discipline from rapid changes in the underlying technology. Yet, as eﬀective

3

as these models have been, one must note that they embody several implicit physical assumptions and these assumptions are based on a certain understanding of the laws of physics. Our understanding of physics has evolved tremendously since 1900! Thus it is time to revisit these abstractions, especially with respect to quantum mechanics. Indeed one should take the physical principles underlying quantum mechanics, the most successful physical theory known to us, and adapt computation to “learn” from these principles. In the words of Girard [17]:

In other terms, what is so good in logic that quantum physics should obey? Can’t we imagine that our conceptions about logic are wrong, so wrong that they are unable to cope with the quantum miracle? [. . . ] Instead of teaching logic to nature, it is more reasonable to learn from her. Instead of interpreting quantum into logic, we shall interpret logic into quantum.

There are, in fact, many diﬀerent quantum mechanical principles which are at odds with our current models of computation. In this paper, we will focus on the previously identified principle of reversibility. In more detail, we will view data as an explicit representation of information and programs as processes that transform information in a reversible way, i.e., processes that are subject to the physical principle of conservation of information. We will formalize this idea and follow its consequences, which will turn out to be far reaching.

Programs as Reversible Deformations. To better understand the essence of “conservation of informa-tion” in the context of computing, we first look for analogous ideas in physics, but this time at the macro scale. Viewing information as a physical object, what does it mean to transform an object in such a way that we do not lose its fundamental character?

For rigid objects (like a chair), the only such transformations are translations and rotations. But what about something more flexible, with multiple representations, such as a water balloon? Such objects can be deformed in various ways, but still retain their fundamental character – as long as we do not puncture them or over-stretch them. Ignoring material characteristics (i.e. over-stretching), what is special about these deformations, as well as for translations and rotations, is that they correspond to continuous maps, with a continuous inverse. In fact, even more is true: they are analytic maps, with analytic inverses. For our purpose, the most important part is that such maps are infinitely diﬀerentiable. In other words, not only is there an inverse to the deformation, but its derivative is also invertible, and so on.

When we look around, we find many diﬀerent words for related concepts: isomorphism, equivalence, sameness, equality, interchangeability, comparability, and correspondence, to name a few. Some of these are informal concepts, while others have formal mathematical meaning. More importantly, even amongst the formal concepts, there are diﬀerences – which is why there are so many of them! Because there are many such notions, we also need to walk our way through them to find the one which is “just right.” Thus we seek a concept which is neither too strong nor too weak, that will express when some structured information should be treated as “the same.” We can draw an analogy with topology: in topology, all point sets can always be equipped with either the discrete or the indiscrete topology, but both of these extremes are rarely useful. We will develop our working notion of “sameness” as we go through the various components that make up a programming language.

Starting from the physical perspective, whatever our notion of data is, we will be interested in programs as representing transformations of that data which are reversible. In other words, we want our programs-as-transformations to “play well” with the inherent notion of “sameness” that our data will carry. Thus we need to start by looking at what structure our data has, which will help us define an appropriate notion of a reversible program. Of course, when programs themselves are data, things do get more complicated. In the following sections, we will look at diﬀerent natural classes of data, and explore the corresponding notion of reversible programs.

To summarize, we will take “the same” as a fundamental principle and derive what it means for data, programs, program transformations, as well as proofs / deductions, to be “the same” – in a manner consistent with preservation of information. This stands in stark contrast with most current approaches to reversible computation, which start from current models of computation involving irreversible operations and try to find various ways to patch things up so as to be reversible.

4

Reversible Programming Languages. The practice of programming languages is replete with ad hoc instances of reversible computations: database transactions, mechanisms for data provenance, checkpoints, stack and exception traces, logs, backups, rollback recoveries, version control systems, reverse engineering, software transactional memories, continuations, backtracking search, and multiple-level “undo” features in commercial applications. In the early nineties, Baker [18, 13] argued for a systematic, first-class, treatment of reversibility. But intensive research in full-fledged reversible models of computations and reversible programming languages was only sparked by the discovery of deep connections between physics and computation [4, 19, 1, 10, 20], and by the potential for eﬃcient quantum computation [2].

The early developments of reversible programming languages started with a conventional programming language, e.g., an extended -calculus, and either

1. extended the language with a history mechanism [21, 22, 23, 24], or
2. imposed constraints on the control flow constructs to make them reversible [25].

More modern approaches recognize that reversible programming languages require a fresh approach and should be designed from first principles without the detour via conventional irreversible languages [26, 27, 28, 29].

In previous work, Carette, Bowman, James, and Sabry [30, 31, 32] introduced the family of typed reversible languages. As motivated above, the starting point for this development is the physical principle of “conservation of information” [33, 34] and the family of languages is designed to embrace this principle by requiring all computations to preserve information.

The fragment without recursive types is universal for reversible boolean circuits [31] and the extension with recursive types and trace operators [35] is a Turing-complete reversible language [31, 30]. While at first sight, too might appear ad hoc, it really arises naturally from an “extended” view of the Curry-Howard correspondence [32]: rather than looking at mere inhabitation as the main source of analogy between logic and computation, type equivalences becomes the source of analogy. Taking inspiration from the fact that many terms of the -calculus arise from Cartesian Closed Categories including, most importantly, a variety of propositional equalities and computation rules, allows us to pursue that analogy further. Some of the details of this development will be motivated and explained in the present paper.