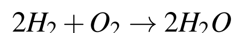
Models of Computation that Conserve Data

Ian Mackie

The idea of *conservation of data* is useful to build programming languages and hardware that support fixed-space programs: data cannot be created or destroyed. In this paper we give some motivations for this work and discuss some related ideas such as linearity and reversibility. We then give some examples of models of computation that can be adapted to work in this way. We conclude by discussing some ongoing work in this area.

1 Introduction

Conservation of energy, conservation of momentum, conservation of mass, and numerous other examples from physics, chemistry and biology provide us with a number of fundamental laws of nature. For example, the law of conservation of mass can be seen in a chemical reaction such as:



$$2H_2 + O_2 \to 2H_2O$$

where the number of atoms in the input must balance with the output: nothing is gained or lost. We will come back to this example later.

The question that we want to address here is “what can be conserved in computation?”. We investigate conserving *data*, but what exactly should we take as “data”? In 1961, Rolf Landauer [9] discovered that energy is needed to erase data. This led to much work on reversible computation [1] (and quantum computation [5]) to avoid erasing data, and similar arguments have also been made for avoiding copying data. These observations and works have motivated our investigation into an approach to computation where data is conserved: programs do not erase or copy data (i.e., the program inputs are not copied or erased).

We review some of the work on reversible and quantum computation to avoid erasing data, and put forward other approaches, such as linearity and directly conserving data, by defining new models. We suggest an approach where syntax and semantics are developed hand-in-hand so that data conservation is also reflected in the syntax. Depending on the level of abstraction, we can talk about data at a number of different levels.

The work is parametrised on what we take as the data that we conserve: bits, memory locations, data structures, programs, etc., and also which kind of computational model we wish to use. We briefly review the appropriateness of term rewriting systems, lambda calculus, interaction nets, chemical abstract machine, and most importantly, geometry of interaction, as well as suggesting some directions for new models.

To justify that the general ideas can work, we give some examples of models of computation that conserve data, and show several programming examples (sorting algorithms). In these examples the syntax and semantics both reflect the use of data in the program, and thus we obtain a visually appealing syntax for in-place algorithms.

We conclude the paper with some suggestions of adapting existing models of computation and developing new models and syntax. Developing a programming language syntax at the same time as understanding the problem semantically seems fruitful, and programming directly with the computational

Preliminary Report. Final version to appear in:
DCM 2018

model avoids the potential distinction as to what is conserved. Much of the effort is currently being applied in these areas, where we envisage applications to topics such as in-place algorithm design, and compiler technology for optimising these algorithms.

Overview. The rest of this paper is organised as follows. In the next section we give some background and related work. In Section 3 we give some ideas for conservation of data. In Section 4 we give an approach using interaction nets. In Section 5 we give an approach using the geometry of interaction. Finally we conclude in Section 6.

2 Background

Taking as a starting point the work of R. Landauer [9] that energy is only needed to erase data (logically irreversible operations), much work has been done on reversible computation (and quantum computation) to avoid erasing data. Linear logic and the study of linearity in computation (for example, functions that use their arguments exactly once) are related notions that avoid erasing (and copying data). We very briefly describe each of these to compare them.

Reversible computation By reversible, we mean a bi-deterministic computation that has the ability to move forwards or backwards in a deterministic way. The ideas were introduced in physics, and are relevant to quantum computation, but they also play a role in computer science through programming language design and implementation.

We note that it is easy to make any computation reversible if we keep the history. For example, if t and u are successive states of a computation, then we could map $t \to u$ into $t^{\sigma} \to u^{t:\sigma}$, where σ is a list of the previous states, and in this way we could easily reverse the step. Clearly this is not what is intended by reversibility (it's a form of cheating) as it has nothing to do with avoiding erasing bits (i.e. it is not thermodynamically reversible). But there are ways to make computation reversible, from programming language level to hardware level: processors, assembly language and programming language design and compilation of other languages.

Linearity The idea of linearity is to study programs where computation does not create or destroy data. A program is said to be linear if it uses each input exactly once in producing its output, so computations just reorganise data. The Linear λ-calculus has been used extensively for this work, and we remark that one can linearise any function by copying arguments. For example, $\lambda x.xx$ becomes $\lambda xy.xy$, and we have to remember to give 2 copies of the argument. However, similar to the remarks about reversible computing above, we can see this as a form of cheating, and extra copies of input are needed that cannot be computed at run-time. Nevertheless, the topic received some interest. In addition, we can iterate linear functions to recover computational power: primitive recursive functions and Gödel's System T can be defined linearly. But these approaches cannot be seen to conserve data. An alternative is to focus on the data and try to conserve it directly.

Conservation of data. We propose an alternative to the above which is a direct approach to neither erasing nor copying data. We remark that it is easy to define such systems if we cheat again: start any computation with some spare data to be used when needed, and accumulate garbage (unused data) at the end. So computation can be given as long as we know how much spare data we need. However, we would like to do better than that with the models we give below.

Comparing Linear, reversible and data conserved A natural question to ask is if any of the above paradigms above are equivalent. The simple answer is no: for example there are reversible computations that do not conserve data, and there are computations that conserve data that are not reversible. Depending on the level of abstraction though, there are times when these are equivalent. We will discuss more on the relationship between these in the full paper.

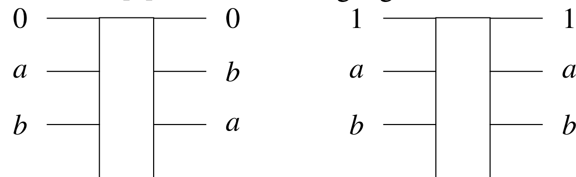
What can be conserved in computation? We investigate conserving data... but what is "data"? It depends on level of abstraction. Some examples include:

- bits: 0 and 1;
- primitive data (numbers, Booleans, etc.);
- data structures;
- memory locations;
- programs (programs as data, first class functions).

We note that in different levels of abstraction we focus on different aspects of the computation, and of course cast away other details. For example, syntax might artificially introduce "data": we have already seen this in the chemical reaction given at the start of the paper: is the "+" operation preserved here? And of course when abstracting details, we avoid talking about other aspects: again, in the chemical reaction energy is needed and produced in this violent reaction (that is not included in the question at all).

Our aim: define models and a syntax for computations that preserve data, and develop applications to memory management (efficient compilation, in-place algorithms, embedded systems, etc.)

Hardware: Fredkin gate Fredkin [4] introduced a logic gate, a controlled swap operation:



The two cases for this gate are shown above: if the top input is 0, then the other two inputs are swapped. Otherwise, the values pass through the gate unchanged. It's important to observe also that the control line is preserved (it is one of the outputs). With this gate we remark:

- The number of 0s and 1s is preserved, so all circuits are just permutations of the input.
- Fredkin showed that this gate is universal. In particular, it is possible to build "and", "or" and "not" gates. For example: "and" is simulated (middle output) by fixing $b = 0$; "not" (bottom output) by fixing $a = 1$, $b = 0$.
- It is also possible to encode copying of bits (top 2 outputs) in a restricted way by setting $a = 1$, $b = 0$, but note that there is an extra output that we don't need, and to facilitate the copy an additional 0 and 1 were needed.

Composing these gates therefore allows us to build a variety of computations with a general pattern that we provide a number of inputs, together with a number of other inputs needed to encode the required computation, and the result is obtained together with some additional bits that are not part of the result.

Knowing that we have this hardware, let's look at higher-level languages to see how we can map back to this.

3 Models that Preserve Data

There is a common pattern in the development of languages for specific hardware:

- Parallel programming:
  - Hardware
  - New programming paradigm/languages
  - Compilation of extant languages (e.g. automatic parallelization)
- Quantum programming
  - Hardware
  - New programming paradigm/languages
  - Compilation of extant languages (e.g. simulation)
- Reversible, Linear, etc.

In all these cases, the common pattern when a new hardware is created involves the development of new languages and/or new paradigms. In addition, compilations of extant languages to the new paradigm are investigated. Hardware can be developed at a different rate to language development (there are many quantum programming languages for example, but few computers available to run the programs on).

What we want to do is to add *Conservation of data* to the above list. Specifically, develop hardware that can run these programs (Fredkin gate is universal and provides a key to the ideas), build assembly language to program such hardware, define new programming languages that can express conservation of data in a natural way, and finally study compilation of existing languages into this framework.

As a starting point, and what we shall do in the rest of this paper is to ask if we can devise such models by imposing constraints on existing models of computation. Some candidates include:

- Linear λ-calculus. Unfortunately, the syntax does not fit well:

  $$(\lambda x.x)(\lambda x.x) \to \lambda x.x$$

  as data (the program) is consumed. This is a consequence of the program and data being in the same framework.

- Term Rewriting Systems. Function-constructor systems can be developed to capture conserving data.

- Interaction nets: some developments are reported below for a restricted form of network.

- Geometry of interaction: we have made some interesting developments that allow for a direct compilation into hardware.

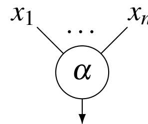
However, we need to decide/agree on a reasonable abstraction: the question is again “what is data?”. Below we look at two examples where some progress has been made. In the full version of this paper we will discuss the λ-calculus and term rewriting systems, in addition to some other models such as Kahn process networks and cellular automata (including some ideas inspired by a universal system [2]).

4 First Approach: Interaction Nets

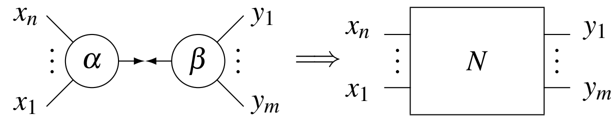
Interaction nets are a particular kind of rewriting system (cf. term rewriting systems [7, 3]). They were put forward as a generalisation of proof nets which are a graphical representation of linear logic proofs [6]. In this paper we present a variant of this paradigm as a candidate for writing programs that preserve data. Some of the key reasons why interaction nets are useful to us are that they are:

- a diagrammatic model of computation where both programs and data are given the same status;
- computation is rule based, and *all* the computation is expressed by the rules:
  - no hidden garbage collection
  - no additional copying machinery

We show that interaction nets can be used as a model for conservation of data. We briefly recall the main definitions for interaction nets. Analogous to term rewriting systems, we have a user-defined signature (α, β, etc.) and a set of user-defined nodes labelled by symbols in the signature:



which are drawn as circles or squares. Nets are built from these nodes, and computation is given by a set of user-defined rewrite rules that have the form:



At most one rule for each pair of agents is allowed, and we remark that the interface is preserved under reduction. A useful example are the Interaction Combinators [8] that we give in Figure 1.

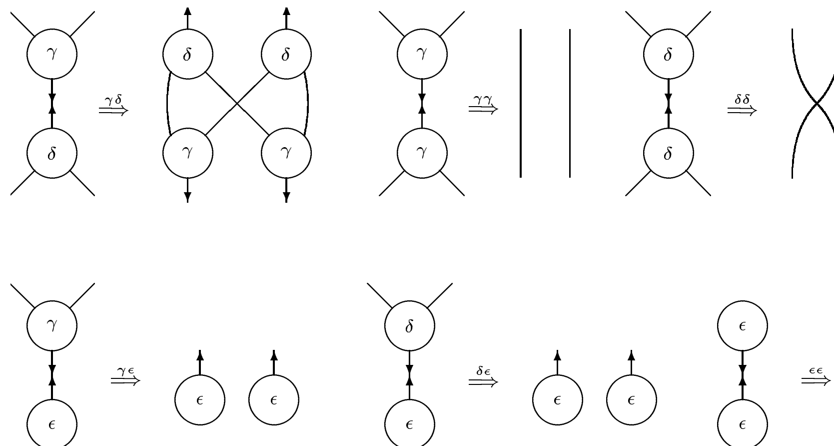
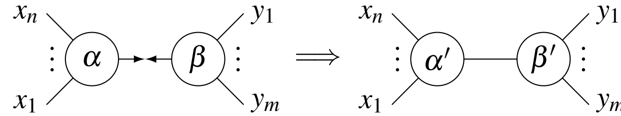


Figure 1: Interaction combinators

It is worth pointing out that these rules exemplify, visually, that computation consists of: copy, reorganise, and erase steps.

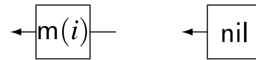
Hard interaction nets This is a class of interaction nets where interaction is restricted to the form where the topology of the network is fixed:



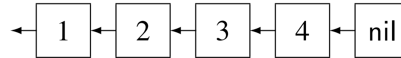
with appropriate principal ports in RHS. This version of interaction nets is guaranteed to conserve the structure of the network by construction. It is our first candidate for conservation of data.

4.1 Examples

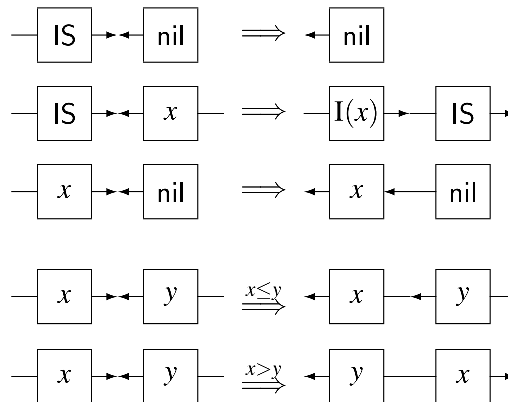
Insertion sort is one of the first sorting algorithms that we learn, and has been well studied in all programming paradigms. We first explain how to represent data using interaction nets. We can represent a memory location, containing an integer i, simply as a node holding the value. This can then be used to give the representation of a list of nodes, with the addition of a nil node.



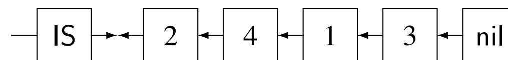
In the diagram above, the node m has one principal port that will be used to interact with it, and one auxiliary port to connect to the remaining elements of the list. The nil node just has one principal port, and no auxiliary ports. To simplify the diagrams, we often just write the contents of the node and omit the name when no confusion will arise. For example, here is a list of 4 elements:



Note that this representation of a list will only allow rewrites through the first element, but just like in other languages, other variants can be implemented. We can now implement insertion sort over this data structure.



The above five rules fully describe the algorithm: there is no other computational machinery required. The example net below can be used to illustrate the dynamics, which we leave for the reader.



We note from this example that:

- Insertion sort can be implemented in-place. It is not clear (visually) if any other programming language has this property.
- There is a choice of application of the rules in the example given, but the order of application does not matter, as this system of rewriting is deterministic (all reductions lead to the same answer).
- Thus, the algorithm is in-place no matter which order of rewrites we perform, which is very strong property.

All we need to do is restrict the programmer to using this fragment. The main point of this example is that the graphical representation of the problem is directly cast into the graphical language. The algorithm given can be understood as programming directly with the internal data structures, rather than some syntax describing it. In addition, all rewrite steps correspond to steps in the computation: there is no need to introduce additional data structures and operations that are not part of the problem.

There are other ways to use interaction nets to conserve data, and we will include those in the full version of this paper.

5 Second approach - Geometry of Interaction

One way to understand the geometry of interaction approach to data flow is that it is history-free, and it keeps the "least information needed". This motivates the investigation of using these ideas to conserve data at run-time.

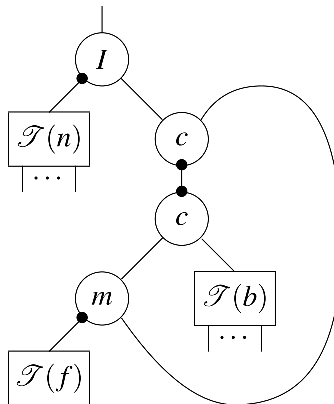
For Multiplicative Linear Logic, it is well understood how to execute a program with:

- fixed network (program)
- single token (run-time system)

where the token can be implemented as a single number (register). But we can take this right down to the machine level:

- spare 0,1 (known in advance by the type of edges)
- implement the operators (known as p, q) as permutations on the token. This can be done using a permutation ordered binary number system.

This idea can be extended to richer languages, including primitive recursive functions and Gödel's System T, using similar ideas. For example, iteration $\mathsf{iter}\ n\ f\ b = f^n(b)$ can be encoded as a fixed network, where $\mathscr{T}(\cdot)$ is the compilation function.



Details of how we can compile this in such as way as to conserve data will be given in the full paper.

6 Conclusion

We have only given a snapshot of work in this area; many strands are currently being developed. Other models are being investigated, for instance a chemical abstract machine (gamma model of programming), amongst others. Development of machine architecture/assembly language is under development, and then compiling extant languages into conserving machine instructions can be investigated further.

Programming directly with a model of computation such as interaction nets seems to have many advantages. But there are many more approaches possible to study conservation of data, and it depends on what we want to count (i.e. the parameter).

References

[1] S. Abramsky (2005): *A Structural Approach To Reversible Computation*. *Theoretical Computer Science* 347, pp. 441–464.

[2] Matthew Cook (2004): *Universality in Elementary Cellular Automata*. *Complex Systems* 15(1). Available at `http://www.complex-systems.com/abstracts/v15_i01_a01.html`.

[3] N. Dershowitz & J.-P. Jouannaud (1989): *Rewrite Systems*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science: Formal Methods and Semantics*, B, North-Holland.

[4] E. Fredkin & T. Toffoli (2002): *Conservative Logic*. In A. Adamatzky, editor: *Collision-Based Computing*, Springer, pp. 47–81.

[5] Simon Gay & Ian Mackie, editors (2010): *Semantic Techniques in Quantum Computation*. Cambridge University Press. ISBN 978-0521513746.

[6] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–102.

[7] Jan-Willem Klop (1992): *Term Rewriting Systems*. In Samson Abramsky, Dov. M. Gabbay & Thomas S.E. Maibaum, editors: *Handbook of Logic in Computer Science*, 2, Oxford University Press, pp. 1–116.

[8] Yves Lafont (1997): *Interaction Combinators*. *Information and Computation* 137(1), pp. 69–101.

[9] R. Landauer (1961): *Irreversibility and Heat Generation in the Computing Process*. *IBM Journal of Research and Development* 5(3).