Journal of Artificial General Intelligence 3(3) 11-42, 2012
DOI: 10.2478/v10229-011-0020-6

Submitted 2012-07-25
Accepted 2012-12-03

Reasoning with Computer Code: a new Mathematical Logic

Sergio Pissanetzky SERGIO@SCICONTROLS.COM
Department of Physics, Graduate School, Texas A&M University, College Station, Texas, USA. Retired

Editors: Kristinn R. Thórisson, Eric Nivel, Ricardo Sanz

Abstract

A logic is a mathematical model of knowledge used to study how we reason, how we describe the world, and how we infer the conclusions that determine our behavior. The logic presented here is natural. It has been experimentally observed, not designed. It represents knowledge as a *causal set*, includes a new type of inference based on the minimization of an *action functional*, and generates its own semantics, making it unnecessary to prescribe one. This logic is suitable for high-level reasoning with computer code, including tasks such as self-programming, object-oriented analysis, refactoring, systems integration, code reuse, and automated programming from sensor-acquired data.

A strong theoretical foundation exists for the new logic. The inference derives laws of conservation from the *permutation symmetry* of the causal set, and calculates the corresponding *conserved quantities*. The association between symmetries and conservation laws is a fundamental and well-known law of nature and a general principle in modern theoretical Physics. The conserved quantities take the form of a nested hierarchy of invariant partitions of the given set. The logic associates elements of the set and binds them together to form the levels of the hierarchy. It is conjectured that the hierarchy corresponds to the *invariant representations* that the brain is known to generate. The hierarchies also represent fully object-oriented, self-generated code, that can be directly compiled and executed (when a compiler becomes available), or translated to a suitable programming language.

The approach is constructivist because all entities are constructed bottom-up, with the fundamental principles of nature being at the bottom, and their existence is proved by construction.

The new logic is mathematically introduced and later discussed in the context of transformations of algorithms and computer programs. We discuss what a full self-programming capability would really mean. We argue that self-programming and the fundamental question about the origin of algorithms are inextricably linked. We discuss previously published, fully automated applications to self-programming, and present a virtual machine that supports the logic, an algorithm that allows for the virtual machine to be simulated on a digital computer, and a fully explained neural network implementation of the algorithm.

Keywords: AGI, emergent inference, mathematical logic, self-programming, virtual machine

1. Introduction

The original motivation for the logic discussed in this paper was the discovery in the course of computational experiments of extraordinary self-organization properties in canonical matrices when a certain *functional* was minimized. This was later confirmed by additional computational experiments. The fact that this logic was directly discovered, rather than engineered or derived from another theory, is important, because it implies that the new logic is natural.

Causal sets are a particular case of partially ordered sets. Canonical matrices have certain advantages for the computational experiments, but they are equivalent to causal sets and causal sets are more fundamental, so the decision was made to continue developing the theory in terms of causal sets. In addition, causal sets are used to represent causality in physical systems, thus establishing a strong link between causality and the new logic. I therefore propose that the new logic be known as *causal logic* (CL), and the corresponding inference as *causal inference* (CI). There have been previous attempts at establishing a causal logic with a proper semantics and inference, see for example Shafer (1998). But these attempts were limited, in large part due to the lack of an appropriate inference. CI has been discussed in Pissanetzky (1984) and Pissanetzky (2011c). Mathematical evidence for the existence of CI was discussed in Pissanetzky (2011a, Section 4).

CL finds its theoretical foundation in the fundamental principles of Physics: causality, symmetry, least-action, and thermodynamics. The Principle of Causality states that effects follow their causes. The Principle of Symmetry states that every symmetry of the action in a physical system has a corresponding *self-organized*, invariant structure, also known as an *attractor*. The Principle of Least Action specifies the trajectory of a dynamical system in its state space as one of least action. And the laws of Thermodynamics establish that the energy of a closed system can not change, and its entropy can not decrease. These four principles summarize the supreme law of nature, the law of laws. CL is *grounded* on the four principles because it was derived directly from them. By interpreting the functional mentioned above as *physical action*, as discussed by Pissanetzky (2012e), CL becomes a theory of Physics on its own right. The theory is explained in more detail in the supplementary material (Pissanetzky, 2012b).

The conservation laws and resulting invariant structures of behavior are of particular interest in artificial intelligence because of the well-known ability of our brain to create structures of knowledge that are conserved (also known as *invariant representations* of knowledge), in the sense that they remain invariant under certain transformations. For example, a person recognizes a chair even if the chair moves to a different position or is turned upside down, or if the person moves the eyes or the head, because the representation of the chair in the brain remains invariant under these transformations. These matters are further discussed in Section 2.1 below.

One way to assess the power of representation of a mathematical model of the world is to enumerate the different physical systems it can represent. By this measure, computer models are likely to be the most powerful. But any computer program is a causal set. A *finite algorithm* is defined as one with a finite number of variables and finite domains for the states of the variables. All computer programs used in real-world computers satisfy these conditions and are finite algorithms, even if they are meant to approximate real-valued functions. Finite algorithms satisfy the definition of causet, and are causets. The set of variables of a finite algorithm, when appropriately renamed to take causality into account, satisfies the definition of causal set. A finite algorithm will either halt or be periodic. If it halts, it is itself a causal set. If it is periodic, the period is a causal set. Detailed algebraic transformations between notation used for causal sets and programming languages, both ways, have been obtained and published (see Pissanetzky, 2009, Sec. 3). Although the transformations are somewhat cumbersome, they can be automated for each programming language once and for all. Note that a causal set can be disconnected and represent multiple causal sequences, or admit random events that initiate new causal chains, just like computer programs. These issues, and the sources of causal sets, are further discussed in Section 3 below, in connection with the question about the origin of algorithms. Algorithmic aspects of partially ordered sets have also been considered by Schröder (2002).

Hence, causal sets are at least as powerful as computer models to represent the world. In Section 2.4, we will see that, because of the inference, they are in fact far more powerful than that. Causal sets are also equivalent to acyclic digraphs and canonical matrices.

When working with causal sets and computer models, it is important to keep in mind that causal logic works very differently than traditional software development. In traditional development, a human developer acquires knowledge about something and then writes a program about it. In causal logic, knowledge is not represented as a traditional computer program. Instead:

> *In causal logic, knowledge is entered as input. There is no program representing knowledge. If causal logic is implemented on a computer, then a program is necessary to handle input/output and to minimize the functional, but not for processing knowledge. The inference, not the programmer, generates the program as output from the knowledge provided as input.*

The fact that knowledge is input and program is output, is what makes causal logic essential for self-programming. It is also essential for all applications where knowledge of the world comes from sensors, not necessarily from humans. This is the case for all autonomous agents.

Causal logic, with the functional that powers it, is a pure mathematical object. It is natural and immutable. It exists in and of itself, independently of anything else. This fact has been verified by applying CL to hundreds of causal sets. CL has, however, strong and direct connections with numerous aspects of Physics, Computer Science, Artificial Intelligence, Complex Systems Science, and other sciences. CL starts from the symmetry that all causal sets have, and finds the attractor. The symmetry is represented by the set of legal permutations of the causal set. To this symmetry, there corresponds a partition of the set known as a *block system* that is invariant under the permutations. More detail is given in Section 2.1.

A very strong connection between CL and intelligence was found at the precise moment where CL was discovered. CL was discovered in the course of a very simple computational experiment performed by the author, using himself both as the observer an the subject. The experiment was one of refactoring, and the problem was a simple law of Analytical Mechanics, the law of independence of the components of motion of a point mass. The experiment is not objective, but it is reproducible by anyone. Its value is not in its objectivity, but in the fact that it facilitated a discovery, the discovery of CL, that had eluded so many other researchers. This was achieved because the experiment was simple enough and the selected problem was appropriate for this type of research. Several similar experiments were later performed using documented and published data from other human subjects. These experiments are objective and observer-independent. They confirmed that CL does describe high brain function in the examined subjects and established the connection between CL and intelligence. Of course, nothing experimental is definitive, and many more experiments are needed.

There really isn't any preliminary work by other authors conducive to CL. CL solves the famous *binding problem*. A quick way to introduce the binding problem is the following statement by Douglas Hofstadter, written in his characteristic style (Hofstadter, 1985, page 633):

> *The major question of AI is this: What in the world is going on to enable you to convert 100,000,000 retinal dots into one single word 'mother' in one tenth of a second?*

In other words, how do the retinal dots bind together and produce 'mother'? It can be said that the binding problem started in mid-19th century with Hermann von Helmholtz and continued in the

20th century with Bertrand Rusell and many of the greatest names of that century. An account of this research can be found in the overview of related work (Pissanetzky, 2012c) in the supplementary material. But the problem remains unsolved to this day.

Section 2 introduces the theory. Included are the fundamentals of causets and causal inference, the nature and properties of the resulting structures, an example with an application to parallel programming, and an introduction to second-order causal logic.

Section 3 covers applications and the practical use of causal logic to solve problems. Material in this Section is strongly focused on the issue of setting up a problem in terms of causal sets. Experience has shown that many developers, not accustomed to using causal sets, tend to believe that using them would be more difficult than writing a program. That is not so, and the Section attempts to dispel the feeling.

Section 4 deals with implementations. Included in Section 4 are the virtual machine, a new much better version of a previously published algorithm known as SCA, and a neural network implementation of causal logic that can be installed as a massively parallel computer, with a fully explained example. The connection between causal logic and the brain is further explored, and several computational experiments based on causal logic are referenced and briefly discussed.

Additional material that is either too detailed or less directly related to self-programming is provided as supplementary material. Section 4.6 contains a description of the material and links to the supplementary files. The material includes an overview of the theory, the reading of which requires a background in Physics, and a special section on verification of the theory, where predictions obtained from the theory are compared against experimental results and heuristic theories from several other disciplines, including Neuroscience, Biophysics, Evolution, Artificial Intelligence, Computer Science, even Google patents. Of critical importance is the prediction that dendritic trees in the brain must be optimally short, made in (Pissanetzky, 2011a, Sec. 3.8), which was recently independently confirmed by Cuntz, Mathy, and Häusser (June 2012), who proposed a 2/3 power law for all regions of all brains across species, with extensive experimental support and replacing a previous 4/3 power law. This is a dramatic confirmation of the theory proposed in the present paper.

Also included as supplementary material is an extensive and detailed case study (Pissanetzky, 2012a), based on a publicly available Java program, and a study on some aspects of object recognition (see Pissanetzky, 2012d).

2. Causal Set Fundamentals

A *causal set*, or *causet*, is a tuple $\Sigma = (S, \omega)$, where S is a finite set and ω is a *binary relation* among the elements of S that is irreflexive, acyclic, and transitive. Some authors add a condition of *local finiteness*, asserting that any order interval in Σ is finite even if Σ is allowed to grow indefinitely. A causet can be considered as a particular case of a *partially ordered set*, or *poset*, but posets can be reflexive and not necessarily finite or acyclic. According to the definition, any finite algorithm is a causet, because the causal relations between the variables of the algorithm satisfy the same conditions specified for causets. The same applies to computer programs. Because of this, the elements of S are sometimes referred to as *variables*. In first order causal logic, the nature or properties of the elements are ignored. The elements can be anything, for example Σ can represent an algorithm consisting of many instructions, and each instruction can be represented by an element of S. But in some cases the properties of the elements can profoundly affect causality itself. This is

the subject of second order logic. Second order logic is only briefly discussed in Section 2.6 because it is beyond the scope of the paper.

The most important application of causets is *causal analysis*, the mathematical analysis of causal physical systems. In this case the binary relation ω is one of *precedence*, and the symbol $\prec$ is used to represent it. Here is an example of a causet with 10 elements and 9 precedence relations:

$$\begin{aligned} S &= \{a, b, c, d, e, f, g, h, i\} \\ \omega &= \{a \prec b, a \prec c, b \prec d, c \prec e, c \prec g, d \prec f, e \prec h, g \prec i\} \\ \Sigma &= (S, \omega). \end{aligned} \tag{1}$$

The nature or meaning of the elements of S is irrelevant. In the example, they represent tasks to be assigned to the processors of a parallel computer.

To every causet $\Sigma = (S, \omega)$ there corresponds an acyclic *directed graph* $G = (V, E)$ where to every $x \in S$ there corresponds a vertex in V, and (x, y) is a directed edge in E iff $(x \prec y) \in \omega$. If G is connected, Σ is also said to be connected. If G is not connected, then G has *connected components*, and to every connected component in G there corresponds a connected component in Σ, and the connected component is itself a causet. For definitions and algorithms on digraphs see for example Pissanetzky (1984, Ch. 5).

Causal sets are finite partially ordered sets, and share their properties. There is a vast literature on partially ordered sets, much of it dedicated to continuous partially ordered sets. For finite partially ordered sets, see for example Caspard, Leclerc, and Monjardet (2012).

2.1 Causal Inference

Let $\Sigma = (S, \omega)$ be a causal set, and let $n = |S|$ be the size of S. Set S has $n!$ permutations, but only some of them are legal. A permutation is *legal* when it does not violate the partial order. Let $\Pi(S, \omega)$ be the set of legal permutations of S under ω, and let $\Psi(S, \omega)$ be the *power set* of $\Pi(S, \omega)$. The power set of Π is the collection of all subsets of Π. I will frequently write simply Π or Ψ to simplify notation.

Set $\Pi(S, \omega)$ represents the *symmetry* of causet Σ. Because of the assumption that ω is a partial order, not total, set Π contains more than one legal permutation. Each permutation in Π represents Σ differently. This situation, where an object can be represented in more than one way, is known as a *symmetry*. Any causet that is not a total order has a symmetry.

But symmetries in the representation of systems are always associated with *laws of conservation* of certain natural quantities known as *conserved quantities* or *invariant structures*. This is a supreme law of nature. Knowing that Σ has a symmetry, we must be immediately interested in the natural structures that correspond to that symmetry and their properties of invariance.

Using causal inference (CI), the invariant structures can be calculated directly from the symmetry of the causet. The structures appear as a *block system* on S. A block system is a partition of S that remains invariant under transformations induced by certain permutations in $\Pi(S, \omega)$. To find the block system, it is first necessary to find the appropriate permutations, and then determine the (unique) block system that is conserved under them. The first task is performed by CI. The second task is standard in group theory, and can be performed with existing tools. The second task will not be discussed in this paper, but examples are presented.

To perform its task, which is to find the subset of selected permutations from set $\Pi(S,\omega)$, CI uses a *functional* F, defined as an expression in terms of the partial order ω. The minimization of F restricts the space of legal permutations to a subspace that has the properties being described.

Let now $\pi \in \Pi(S,\omega)$ be a legal permutation of S. Functional F, called the *cost* of permutation π, is now defined over set Π as follows:
1. number the elements of S in the order they appear in π, starting, say, from 1, and let $\nu(\alpha)$ be the number assigned to some element $\alpha \in S$;
2. if $r : \alpha \prec \beta$ is a relation in ω, then the cost of relation r is defined to be $C(r) = \nu(\beta) - \nu(\alpha)$, and $C(r) > 0$ because π is legal; and
3. $L(\pi)$, the cost of permutation π, is twice the sum of the costs of all relations in ω. The expression for the functional is:

$$L(\pi) = 2\sum_{r\in\omega} C(r). \tag{2}$$

A functional is a map that assigns a number, not necessarily different, to each permutation. In this case, the number is a positive integer. For example, permutation (b,e,g,d,f,h,i,a,c,j) in Eq. (1) is legal. The numbers assigned to elements of S are $\nu(b) = 1$, $\nu(e) = 2$, $\nu(g) = 3$, etc. The cost of this permutation is 28.

The functional plays the role of a *hash function* on the space $\Pi(S,\omega)$ of legal permutations. Like a hash function, it partitions Π into subsets, each containing permutations with the same cost. It then becomes possible to *minimize* the functional, which means finding the subset of permutations with the minimum cost, say $\Pi_{\min}(S,\omega) \subseteq \Pi(S,\omega)$. Subset $\Pi_{\min}$ is non-empty and may contain more than one permutation. However, unlike a hash function, the minimization of $L(\pi)$ creates associations among the elements of S, and the associations result in structure, which is detectable. $\Pi_{\min}$ has the following remarkable mathematical property:

> *Subset* $\Pi_{\min}(S,\omega)$ *is either a permutation group of set* S *or a generator for a permutation group of* S. *In either case, subset* $\Pi_{\min}$ *induces a block system in set* S.

A *block system* is a partition of S into *blocks*, or disjoint subsets, that have the property of being *stable*, meaning that they are *invariant* under the action of $\Pi_{\min}$. In other words, the same blocks are found in all permutations in $\Pi_{\min}$. For our purposes, we use *minimal* blocks, that is, blocks that can not be reduced in size without loosing the property of being blocks.

When the block system is generated, the relations in the partial order ω of the original causet are separated into two groups. Some of them are *encapsulated* into the blocks. For all practical purposes, they become a property of the blocks. The remaining relations are *induced* to the block system itself, and become relations among the blocks. Induction makes the block system itself a causal set, the elements of which are the blocks, and the partial order of which consists of the induced relations. In summary, the process just described can be viewed as a function that takes a causet and obtains another causet that represents a block system over the first. To formalize this statement, define:

$$\begin{aligned} \mathcal{H} &= \{\Sigma | \Sigma \text{ is a causet}\} \\ \mathcal{B} &= \{B | B \text{ is an inheritance map}\} \\ \mathcal{E} &: \mathcal{H} \to \mathcal{H} \times \mathcal{B} \end{aligned} \tag{3}$$

where $\mathcal{H}$ is the collection of all causets, $\mathcal{B}$ is the collection of all inheritance maps, both of size infinite but countable, and $\mathcal{E}$ is the function that takes a causet $\Sigma \in \mathcal{H}$ and corresponds it to a pair (Σ', B), where $\Sigma' \in \mathcal{H}$ is the resulting causet, possibly smaller, and $B \in \mathcal{B}$ is the resulting inheritance map. An *inheritance map* is a function that maps from each element of Σ' to its corresponding block in Σ, including the order of the elements of Σ contained in the block and the list of their associations and their types as found by CI.

Function $\mathcal{E}$ is believed to be uncomputable, and also *unprovable*, because it can only be proved by construction, and constructing all cases is not possible. To be more precise about the uncomputability of $\mathcal{E}$, the part that is uncomputable is the *form* of the expression of the functional L that powers CI, Eq. (2). But once the expression is given, as it is in this paper, its *value* is easily computable on any machine. It is very easy to implement CI on a PC. The map of $\mathcal{E}$ is *bijective*, and the inverse function $\mathcal{E}^{-1}$ exists and is computable. $\mathcal{E}^{-1}$ reconstructs set Σ from Σ' and the information contained in the inheritance map. Note that everything being said here about causets can also be said about algorithms or computer programs. Transformations reported in Pissanetzky (2009, Sec. III) can be used to compute $\mathcal{E}^{-1}$ in the case of computer code, but in the case of causets in causet notation it is easier to reconstruct Σ directly.

When function $\mathcal{E}$ is applied to a causet, the result is another causet and the corresponding inheritance map. Function $\mathcal{E}$ can be re-applied to the new causet, and the process can be repeated until an *exhaustion* condition is reached. The result of the multiple application of $\mathcal{E}$ to the original causet is a *nested hierarchy* of causets, organized by *inheritance* into levels of the hierarchy. The levels of the hierarchy possess the properties of a *mathematical fractal*. Under certain conditions, such as the requirement for using minimal blocks in order to maximize the depth of the hierarchy, the hierarchy is unique. This hierarchy is the conserved structure or invariant representation that corresponds to the symmetry of the original causet.

$\mathcal{E}$ and $\mathcal{E}^{-1}$ can operate on entire algorithms, or computer programs, hence the title of this paper "Reasoning with computer code." The elements of the causet can be lines of code, subroutines, or even entire programs. Of course, the critical question still remains: what if anything do these conserved structures have to do with the brain's invariant representations. This question can only be addressed by experiment. Experiments in Section 4.5 suggest that they are the same.

2.2 An example of Causal Inference in Parallel Programming

This is an example of an application of CI to solve the assignment problem of parallel programming in a simplified case, where a specified set of tasks is to be assigned to the processors of a parallel computer. There are 9 tasks, called $a, b, c, d, e, f, g, h, i$. Also given are the 8 task inter-dependencies $a \prec b$, $a \prec c$, $b \prec d$, $c \prec e$, $c \prec g$, $d \prec f$, $e \prec h$, and $g \prec i$, meaning for example that task a must complete before execution of task b can begin. The solution to the problem obtained by a human analyst is the directed graph shown in Fig. 1(a). To cast this problem in the form of a causet, the 9 tasks are taken as the elements of set S, and the 8 task inter-dependencies as the precedence relations. The resulting causet is given in Eq. (1).

For the purpose of causal analysis, the nature of the elements of S is irrelevant. This example is intended to illustrate, at a presentation level, a basic technique that can be applied to automate parallel programming. Extensions or improvements of this basic technique can, then, be developed, and used for commercial purposes. Current problems in parallel programming have grown so much that they are said to be exceeding the mental capacity of programmers. Full automation is urgent.

A solution to this problem will now be developed using CI, and compared with the solution obtained by the human analyst of Fig. 1(a). The procedure involves 5 iterations, illustrated in Fig. 1(b). System Σ corresponds to level $L1$ in the figure. Each iteration associates tasks, and creates an invariant partition of set S, taking the system from its current level to the next higher level.

Causet Σ has 336 legal permutations, is reduced (no redundant relations), and connected (one connected component). The cost of the 336 permutations, according to the functional of Eq. (2), ranges from 26 to 40. The least cost is 26, and there are 2 permutations with that cost. They are:

$$\begin{array}{l}(a, b, d, f, c, e, h, g, i)\\(a, b, d, f, c, g, i, e, h)\end{array} \tag{4}$$

The block system they induce in S is:

$$B = (a)(b)(d)(f)(c)(e-h)(g-i) \tag{5}$$

By simple inspection, block system B can be verified to be invariant under the action of the two permutations of Eq. (4). The difference between set S and system B is the presence of the two non-trivial blocks $(e-h)$ and $(g-i)$, where the $-$ sign indicates a relation of *immediate precedence*. In other words, e immediately precedes h and g immediately precedes i. Block $(e-h)$ not only *encapsulates* relation $e < h$ from partial order ω, but also confirms that there exist no other relations in ω that could affect that relation. The same applies to block $(g-i)$. The two blocks represent *associations* that the inference has automatically created, associations that exist in B but did not exist in S. The process of emergence that leads from S to B is one of inference, because the associations in B are a new fact derived from known facts. Block system B corresponds to level $L2$ in Fig. 1(b). Many more examples with small systems like this one are given in Pissanetzky (2011a, Sec. 4).

2.3 Constructing a hierarchy of block systems

As explained above, when a block system is created, the original precedence relations become either *encapsulated* into the blocks, or *induced* as relations among the blocks. In some cases, duplicated induced relations may appear, and these need to be eliminated. Let us rename the 7 blocks with $a_2, b_2, c_2, d_2, e_2, f_2, g_2$, in the order they appear in Eq. (5):

$$\begin{array}{cccccccc}\text{rename} & (a) & (b) & (d) & (f) & (c) & (e-h) & (g-i)\\ \text{with} & a_2 & b_2 & c_2 & d_2 & e_2 & f_2 & g_2\end{array} \tag{6}$$

The only purpose of renaming is to simplify notation. Now, consider the partial order ω of Eq. (1), which is repeated here for convenience:

$$\omega = \{a \prec b, a \prec c, b \prec d, c \prec e, c \prec g, d \prec f, e \prec h, g \prec i\} \tag{7}$$

Induction of the relations in ω into block system B of Eq. (5) results in the following partial order:

$$\begin{array}{cccccccc}\omega & (a \prec b) & (a \prec c) & (b \prec d) & (c \prec e) & (c \prec g) & (d \prec f) & (e \prec h) & (g \prec i)\\ \omega_2 & (a_2 \prec b_2) & (a_2 \prec e_2) & (b_2 \prec c_2) & (e_2 \prec f_2) & (e_2 \prec g_2) & (c_2 \prec d_2) & f_2 & g_2\end{array} \tag{8}$$

The bottom line contains the partial order ω_2 for the blocks. The first six relations have been induced into ω_2. The last two have been encapsulated into blocks f_2 and g_2, respectively, and do

not participate in ω_2. Block system B of Eq. (5) can now be written as a causal set, say Σ_2, given in the alphabetical order:

$$\begin{aligned} S_2 &= \{a_2, b_2, c_2, d_2, e_2, f_2, g_2\} \\ \omega_2 &= \{a_2 \prec b_2, a_2 \prec e_2, b_2 \prec c_2, c_2 \prec d_2, e_2 \prec f_2, e_2 \prec g_2\} \\ \Sigma_2 &= (S_2, \omega_2). \end{aligned} \tag{9}$$

In Fig. 1(b), causal set Σ of Eq. (1) has been represented as level $L1$, and causal set Σ_2 of Eq. (9) as level L_2. In summary, the process just described is: given a causal set, find the associations, permute accordingly, calculate the block partition, and use it to obtain a new, usually smaller causal set.

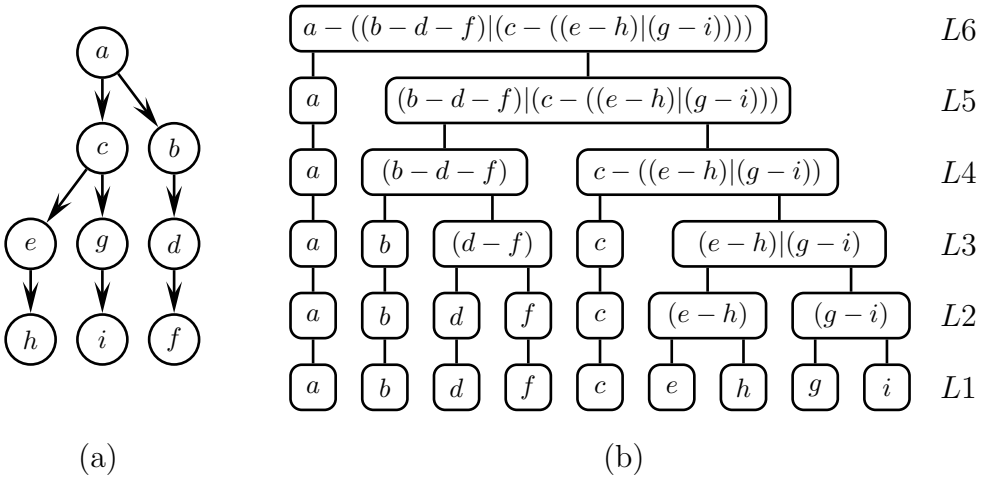

Figure 1: (a) The solution to the problem of parallel programming. (b) The 6-level hierarchy of structures obtained in Section 2.3. "–" indicates immediate precedence, in that order, and "|" indicates no precedence.

We will now apply the same procedure to Σ_2. Causal set Σ_2 is of order $n_2 = 7$, has 6 relations, is reduced and connected, and has 40 legal permutations with a cost range from 20 to 28. There are 4 least-cost permutations, and they are the following:

$$\begin{aligned} &(a_2, b_2, c_2, d_2, e_2, f_2, g_2) \\ &(a_2, b_2, c_2, d_2, e_2, g_2, f_2) \\ &(a_2, e_2, f_2, g_2, b_2, c_2, d_2) \\ &(a_2, e_2, g_2, f_2, b_2, c_2, d_2) \end{aligned} \tag{10}$$

The block system they induce in S_2 is:

$$B_2 = (a_2)(b_2)(c_2 - d_2)(e_2)(f_2|g_2), \tag{11}$$

where the "|" sign indicates association but not precedence. Block system B_2 is a new causal set, say Σ_3, with 5 elements and 4 relations. It corresponds to level $L3$ in Fig. 1(b). Renaming the 5

elements as a_3, b_3, c_3, d_3, e_3, respectively, the following causet is obtained:

$$\begin{aligned} S_3 &= \{a_3, b_3, c_3, d_3, e_3\} \\ \omega_3 &= \{a_3 \prec b_3, a_3 \prec d_3, b_3 \prec c_3, d_3 \prec e_3\} \\ \Sigma_3 &= (S_3, \omega_3), \end{aligned} \tag{12}$$

corresponding to level L_3 of the hierarchy. System Σ_3 has only 6 legal permutations, the least cost is 12, and there are 2 permutations with that cost. They are:

$$\begin{aligned} &(a_3, b_3, c_3, d_3, e_3) \\ &(a_3, d_3, e_3, b_3, c_3) \end{aligned} \tag{13}$$

They induce in S_3 the following block system:

$$B_3 = (a_3)(b_3 - c_3)(d_3 - e_3), \tag{14}$$

which corresponds to level $L4$ in the hierarchy. Two more systems and two more iterations are needed to complete the remaining of the hierarchy, for a total of 6 levels. They are simple enough to be solved manually, and are left as an exercise for the reader. The highest level in the hierarchy corresponds exactly to the human analyst's solution of Fig. 1(a), which was discussed at the beginning of Section 2.2. This is the nested hierarchy of associations and invariant partitions determined by causal set Σ of Eq. (1).

This example reveals in a dramatic way the progressive build up of associations and structure obtained by the inference in each one of the 5 iterations. The first iteration associates e with h and g with i, each in immediate sequence but independently of each other, indicating that the two pairs of tasks can execute in parallel. The second iteration associates d with f in sequence, and pairs $e - h$ and $g - i$, not in sequence, creating a parallel computer with 2 CPU's, shown in level $L3$. Iteration 3 associates b to $d - f$ in sequence, creating the sequence $b - d - f$, suitable for a single CPU. Iteration 3 also associates task c as a predecessor of both $e - h$ and $g - i$. In a similar way, one step at a time, iterations 4 and 5 find the remaining associations and create the parallel computer in Fig. 1 (a).

At this point, it is very important to review the process that led from the definition of system Σ in Eq. (1) to the resulting structures shown in Fig. 1(b). This process is mechanical, its only input is system Σ, and its output is the hierarchy of block systems. Nothing in the process depends on Σ or contains any intelligence related to Σ. The only parameter in the process is the functional of Eq. (2), but the form of the functional is universal and independent of Σ, and its value is determined only by the partial order ω. It follows that all the structures in Fig. 1(b) *depend on, and are determined only by* the partial order ω in system Σ. This statement gives rise to the following principle, which is actually a consequence of the Principle of Emergence:

> **Principle of Structure of the Information.** *The causal information describing a system determines its structure.*

The process by which the structures are found is called *emergence* or *self-organization*. This process removes entropy from the physical system and causes it to converge to *attractors*, which are the structures being sought. At that point, the system becomes *conservative*. Its state space shrinks to the attractors, and it is forced to remain there unless it again gains energy, and therefore entropy. In

AGI, the process of gaining entropy is called *learning*. As the system learns, it gains entropy and becomes disordered. At the same time, if it is open, it again looses energy to the environment and converges to a different order. The two processes compete, causing the system to constantly evolve between states of disorder and constantly changing attractors. An outside observer can only detect the attractors. The observer will see the system as if it were in a state of constant change, of constant motion.

2.4 The significance of Causal Inference

There is no reason to believe that the existence of invariant structures will be less important in AGI or self-programming than, for example, in modern theoretical Physics, where they are fundamental. This is particularly so because our brains are well-known to generate *invariant representations* of our experience all the time, see for example Hawkins (2004), and to use them to derive *meaning* from knowledge. The ability of causal sets to support the creation of invariant structures is the cornerstone of the work being presented here.

It is also very important to note that the formation of the invariant structures is unintended. It is a *side effect* of the minimization of the functional. The functional is natural. It has been observed and it is used exactly as it was observed. It has not been engineered or designed for any particular purpose, or for obtaining any particular type of structures. As a consequence, the resulting structures depend only on the given causet, and can not be altered or changed in any way. They can be used for engineering AGI or self-programming, but can not be engineered themselves. Engineering these structures would be comparable to, for example, engineering the logarithm function. Hence, causal inference is a natural principle for the engineering of AGI and self-programming:

> *Causal inference is hereby proposed as a natural principle for the engineering of AGI and self-programming.*

In addition, it is also critically important to note that CI is a behavior-preserving transformation. In Software Engineering, behavior-preserving transformations are known as *refactorings* (see Opdyke, 1992), but they exist for all kinds of knowledge. The behavior *already exists* in the knowledge represented by the original causal set, it is not created by CI. The acquired knowledge itself is the behavior. If the result of CI is an algorithm or computer program, the same algorithm or computer program exists in the given causet. The difference is in the notation and the presence of structures, or *entropy*. Causet notation is a "high entropy" (low order) notation that emphasizes knowledge acquisition and processing, not understandability by a human subject. Algorithm or programming language notation is a "low entropy" (high order) notation that represents high level features in terms of low level features, emphasizes understandability, and is intended for use by a human subject. CI *is not* an algorithm factory. CI does not create algorithms, it merely *reasons* with an algorithm by reducing entropy, increasing order, and improving understandability. Hence, the origin of algorithms is not to be found in CI, but in the *input* to CI:

> *The origin of algorithms is observation, defined as the process of acquisition of knowledge about the world.*

This statement can be rephrased as a mathematical equation:

> *acquired knowledge = algorithm = behavior.*

There are many ways for acquiring knowledge and representing it as a causet. Some of them are discussed in Section 3.

2.5 Self-programming and the origin of algorithms

Computer programming, as practiced today by human developers, can be viewed as a process where the developer receives certain information from various sources, and is asked to create an algorithm. It is tempting to append "with certain properties," but it is better to consider "certain properties" as part of the given information. The sources of the information may include a problem statement, additional clarifications from subject-matter engineers, and the developer's knowledge about some programming language. They may also include existing algorithms, already coded by other developers, and even data obtained directly from sensors. For example, a developer may receive patterns or pictures that need to be recognized. It is better to refer to this type of information as *knowledge about the world*, or as *experience*.

The developer's task is to *infer* an algorithm *from* the given knowledge. The two terms, "infer" and "from", contain the clue for the answer to the fundamental question about the origin of algorithms. The answer is that algorithms originate from experience, and from experience only. A developer will stop developing if she finds herself short of experience, and will seek more detail. Developers never guess. Turing referred to algorithms as "instruction tables" to be developed by "mathematicians with experience in solving puzzles." Computer programming can be defined as the art of inferring an algorithm from a given body of experience.

When programming is viewed as an art practiced by humans where an algorithm is inferred from given knowledge, then self-programming should be viewed as a mechanical process that does the same. A machine with such a capability should do two things: represent the knowledge, and infer new facts from known facts, with the additional requirement that the new facts should be algorithms. By definition of logic, such a machine is a mathematical logic with an inference that infers algorithms. The new causal logic (CL) discussed in this paper has precisely those capabilities. As discussed in Section 3, CL has an exceptional ability to represent knowledge. As discussed in Section 2.1, causal inference (CI) can infer new facts from the existing knowledge, and as discussed in Section 2.4, the facts that CI infers are algorithms. The issue of *meaning* is addressed as part of the example in Section 4.3.

A more detailed answer to the question about the origin of algorithms would be that, working from a body of knowledge in a generally disorganized state, CI infers a unique algorithm, corresponding to that body, which is in a fully organized state and contains structures from which *meaning* can be drawn.

In this light, man-made programming can now be understood as a very indirect, exceedingly labor-intensive and wasteful process, that consists of three parts. The first part is called *analysis*, where a human analyst acquires knowledge from external sources and creates in her mind an organized, structured algorithm. She then extracts meaning from the structures, and uses that meaning to create a *design* for the future program, consisting (hopefully) of the same structures, but now described in natural language. Finally, human developers working from the design capture the meaning again and write the final code in some programming language that a compiler can understand.

2.6 Second-order causal logic

As explained above, in first-order causal logic, the nature of the elements of the original causal set is irrelevant. But the elements can have properties that, in some cases, affect the causality represented by the causal relations. For example, it is possible for a precedence relation such as

$a \prec b$ to be canceled away as a result of properties of element a, making b independent of a. But independence is symmetry, and symmetries lead to the existence of invariant structures. One very important example of this situation is Noether's theorem (Noether, 1918), where a simple cancelation of causality in a certain type of differential equations results in observable structures, known as *conserved quantities*, which the theorem allows to calculate, and which proved to be of a fundamental importance in modern theoretical Physics. To deal with such situations, a second-order causal logic is proposed, where the properties of the elements of the causet are taken into account.

Below is an example of a simple subcase of Noether's equation. Consider the following finite sets:

$$\begin{aligned} t &= \{a, b\} \\ s &= \{c, d\} \\ x &= \{e, f, g, h\} \\ v &= \{i, j, k, \ell\} \end{aligned} \tag{15}$$

Consider also set K, to be determined, and the following three functions:

$$\begin{aligned} f_x &: t \times s \to x \\ f_v &: t \times s \to v \\ f_K &: x \times v \to K \end{aligned} \tag{16}$$

Functions f_x, f_v, and f_K are defined in Fig. 2. The symbol $\star$ used in the table defining f_K indicates values that are not used.

t \ s	c	d
a	e	f
b	g	h

$f_x : t \times s \to x$

t \ s	c	d
a	i	j
b	k	ℓ

$f_v : t \times s \to v$

x \ v	ℓ	i	k	j
g	$*$	$*$	m	$*$
f	$*$	$*$	$*$	n
e	$*$	m	$*$	$*$
h	n	$*$	$*$	$*$

$f_K : x \times v \to K$

t \ s	c	d
a	m	n
b	m	n

$f_K^\star : t \times s \to K$

Figure 2: Definitions of functions f_x, f_v, f_K, f_K^* used in Section 2.6.

The independent variables are t and s, where t corresponds to time and s will be shown to be a symmetry. Since each of x and v depend on t and s, and K depends on x, v, one would expect K to also depend on t and s. However, the fact that each of x and v depends on both t and s establishes a bijection between x and v and reduces set $x \times v$ from 16 elements to only 4. In fact, if $t = a$ and $s = c$, then $x = e$ and $v = i$, and other combinations of the value $x = e$ with other values of v are eliminated. The result is that only 4 values of K are possible. For example, if $x = e$, then the value $v = j$ can not happen. This property is the equivalent of a Lagrangian formulation in Physics.

If, in addition, the 4 allowed values of K are pairwise the same, indicating the existence of a symmetry, then set K has only 2 elements, m and n in this case, or $K = \{m, n\}$, and K becomes independent of t as indicated by function $f_K^\star$ in Fig. /refFIG124. Set K is the conserved quantity. This effect is a *cancelation of causality*, one where two causal relations cancel each other as a result

of properties of the elements. The cancelation of causality is symmetry, and the symmetry results in a conserved quantity, in this case set K.

3. Using Causal Logic to Solve Problems

Prospective users of causal logic, not accustomed to using causal sets, tend to feel that using them would require an additional effort on their part. However, it is easy to prove that setting up a problem in terms of causal sets never requires more effort than setting it up in the traditional way, that is, by writing a program. Because the program *is* a causal set. In a worst-case scenario, it is always possible to write the program and use it as-is for input as a causal set. This situation may be useful for the case of existing software. However, the situation would be very different if the developer has to write code.

Writing code involves several tasks: learning and coding the different parts of the problem, putting the code together, refactoring it for organization, and creating structures such as subroutines, classes and objects for understandability. Most of these tasks can be eliminated if a causal logic-based development platform were available. It would suffice for the developer to learn the different parts of the problem and code them, and then use the result directly as input. The logic would do the rest. I did not have a chance to put such a platform into effect, but I do have experience in expressing knowledge about problems in causal set form.

Much of the present Section is focused on *sources of causal sets*, the aim being to demonstrate how our many and very different ways of thinking about knowledge are unified when we think in terms of causal sets. The Section explains how functions, programs, differential equations, even real numbers and the infinite, can in fact be thought of directly as causal sets and expressed accordingly.

Causets must be considered as eminently dynamical entities, because they are used to model *open* systems, systems that can *interact* with their environment. They originate from sources, they can *grow* by adding elements to S and/or relations to ω, they can suffer cancelations or simplifications and loose some elements or relations, they can be partitioned and give rise to other causets. Growth can initiate a new component or add to an existing component. In practice, very large multi-component causets are the norm rather than the exception.

No logic makes sense unless it has a substantial power of representation of our knowledge about the world. The following sections describe a variety of *sources of causets*, that illustrate how very different real world problems can be described in terms of causets, as a preliminary step before the application of CI to effect the self-organization and create the structures.

For self-programming purposes, once a causet has been obtained, or even while it is being obtained by any of the various procedures described below, causal inference can be used to self-organize the causet and create the hierarchical structures that represent the classes and objects of the corresponding object-oriented program. This is further discussed in Section 3.3.

3.1 Functions

According to the definition of causet, any function $f : x \to y$ is a causet if sets x and y are finite. If x and y are disjoint, then $S = x \cup y$. If they are not, the causet can still be obtained by renaming the elements of y. For example, the function $y = x^2$ on the natural numbers $x < 4$. The causal set describing this function is:

$$S \;=\; \{1, 2, 3, , 1', 4', 9'\} \tag{17}$$

$$\omega \quad = \quad \{1 \prec 1', 2 \prec 4', 3 \prec 9'\}$$

More in general, any finite-domain algorithm or computer program that halts can be converted to causet form. If it doesn't halt, then it is periodic, and each period is a causet. There follows that any computer model of a physical system can be converted to a causet model. Conversely, any causet can be written as an algorithm or computer program. The fact that computer models have been used to simulate virtually everything, gives a good idea of the power of representation of causets. These issues have been examined un full detail in Pissanetzky (2009, Sec. III), and are briefly revisited in Section 3.3 of this paper.

A function over an infinite domain can be described in the same way we humans do in our own mental representations: by reference to the notion of mathematical limit. This is covered in Section 3.2.

An interesting case, is that of a system of *simultaneous* algebraic equations, where "simultaneous" means that all the equations must be satisfied at the same time and are therefore not causal. However, every method used to actually solve such a system, for example substitution, or the Gauss method, of the inverse matrix method, is sequential and causal, and can therefore be described by a causal set. The case of differential equations, including simultaneous differential equations, is discussed in Section 3.4.

3.2 The mathematical limit

The notion of mathematical limit is a statement about the *absence* of a causal relation from the control variable of some recursive process to some property described by the recursion. For example, the notion that there exists and infinite number of natural numbers can be formalized by saying that, given any natural number n, $n + 1$ always exists. Which means that the property defined as "$n + 1$ always exists" is independent of n. In computational terms, one would say that the property can be taken outside the loop controlled by variable n.

The reader can easily verify that every statement that is made in science regarding infinite or continuous mathematical objects, is always a reference to the notion of mathematical limit. For example, if we work with real numbers, we are referred to a recursive process, that given two real numbers $a < b$ then a third real number c always exists such that $a < c$ and $c < b$, and to the absence of a causal relation between the size of the interval and that property.

Consequently, the causal definition of the mathematical limit consists of the recursion and the absence of the causal relation as *predecessors* of the limit. Whenever these two predecessors exist, the limit is taken. In causal set theory, which uses finite causal sets to represent knowledge, infinite and continuous mathematical objects are dealt with in the same way as we humans deal with them: by using finite means.

3.3 Computer programs as causal sets

A computer executing a program is an example of a causal system. The computer is a physical system, and the program describes its evolution in time through a series of states. The time dependence is what makes the computer a causal system, independently of the program it is executing. For example, consider a loop that contains a statement in C that does not appear to be causal, such as:

$$A = A + 1; \tag{18}$$

The value found in memory location A at a certain time t is increased by 1 and the result is stored in the same location at a later time, say $t + \tau$. Formally, this same operation can be written as follows:

$$A_{t+\tau} = A_t + 1; \tag{19}$$

which is now clearly causal and can be represented by causet $S = \{A_t, A_{t+\tau}, 1\}, \omega = \{A_t \prec A_{t+\tau}, 1 \prec A_{t+\tau}\}$. Converting a computer program to causet notation entails unrolling all loops and making explicit the time dependence of variables. The result can be a very large causet. However, if certain allowances are made, it should be possible to develop a C-like or Java-like language that allows loops and memory and code reuse and can still be mathematically treated as a causet for uses such as applying this logic and reasoning with the code, and automatic object-oriented analysis and refactoring.

Causets can also be obtained directly from certain models of computation (Bolognesi, 2010).

3.4 Differential equations

The notation used for differential equations (DEs) is a combination of causal relations and equally causal algebraic relations. When we write a DE that contains a differential, say dx, we are considering two points, say x_1, x_2, such that $x_2 - x_1$ is "small," where "small" is to be defined later. If we write a partial differential, say ∂x, we assume that x may depend on more than one variable, but, for the purpose of calculating the boundaries of the interval x_1, x_2, we consider only the dependence of x on one of those variables, say y, and an interval y_1, y_2 that is small. The partial derivative is the quotient between the two, which has (or may not have) the property of being independent – absence of a causal relation – from the size of the interval $y_2 - y_1$. All the statements above are finite and causal, and it is possible to write a causal set that describes the causal relations between the various variables and functions involved in the DE.

The causal relations between the variables may be useful by themselves. An example of an application of second order causal logic to a subclass of the Noether class of PDEs is presented in Section 2.6. There is reason to believe that the entire class of Noether PDEs can be treated in a similar way.

When it comes to solving a DE, it is necessary to recall the fact that DEs are "on the other side of the mathematical limit." Some DEs may have analytical solutions, which are as well on the other side of the limit. In all other cases, it becomes necessary to come this side of the limit by discretizing the DE. There are many kinds of numerical methods for doing this, such as Gauss-Seidel, finite differences, finite elements, but the feature that all these methods have in common is that they are finite, and they are also causal, and can therefore be described by a causal set.

3.5 Learning as a source of causets

Besides functions, algorithms, computer programs, and differential equations, a causet can originate and grow by acquiring new knowledge. From the perspective of the causet, it makes little difference where the new knowledge comes from. But for the purpose of analysis, it is better to follow tradition and identify the source as either a *teacher* or a *sensor*.

Learning from a teacher is one possible cause of growth for causets. At the most basic level, a teacher that teaches in the style of one-fact-at-a-time is actually specifying new elements and new relations for the causet. For example, a teacher can state:

a credit card has a name, an address, and a number.

This statement specifies four elements, *credit card*, *name*, *address*, and *number*, and declares that *name*, *address*, and *number* precede *credit card* (one can't have a credit card without a name, an address, or a number). An implementation of the causal set may realize that "name" has no predecessors, meaning it has not been defined, and ask the question:

what is a name?,

to which the teacher may reply:

a name consists of a first name, a middle initial, and a last name,

where *first name*, *middle initial*, and *last name* are added as elements of the set and declared as predecessors of *name*. That way, a large causal set is built step by step. If, in addition, a process of causal inference exists, that process can go on uninterrupted while the process of learning is taking place. Since CI is behavior-preserving, the two processes, CI and learning, are independent, and can work concurrently. This is a characteristic of host-guest systems. A more detailed discussion of learning with a teacher is available in (Pissanetzky, 2009, Sec. V.F).

Note that the nature of the elements introduced by the teacher is irrelevant. An element can have a structure of its own, or be as complicated or as abstract as desired. When the teacher states that name has first name and last name, he is specifying additional structure for the existing variable name. At a higher level of learning, most of the elements are of a much higher level, but they are still elements that can be used in a causal set and/or refined to any desired degree. Think of the many computer-based courses that teach just about everything. If the course is a program, then it is already a causet and can be converted to causet notation, even without an understanding of natural language processing. Computer-based courses are certainly a very important source of causets.

3.6 Sensors as a source of causets

Sensors - and sensory organs as well - are natural sources of causets. Sensors collect causal information about an event. Consider a TV camera with many pixels. If light from the environment impinges on one of the pixels, that's an event. It happens at a known point in space - the location of the pixel in the camera and the location of the camera - and at a known time. The event is causal. The pixel generates an electric signal. That's two elements, the light and the signal, and one relation, light precedes the signal. If the camera has one million pixels, that's a causet with one million elements and one million precedence relations. And all that for each instant of time. That is a great deal of causal information being collected by a sensor.

The notion of learning from sensors is very general, and can be extended to include cases of great interest in self-programming, such as the integration of systems and the reuse of existing computer code. In these cases, the "sensor" would be a computer file, and the process of reading the file as input would be one of unsupervised learning. The basic technology for both integration and reuse would be that of merging two different causal sets. This is easy to do if the two causets are element-disjoint.

An integrated software development platform where programs are represented as causal sets has been considered in previous publications. This is a fully integrated and fully functional platform, which naturally supports systems integration, the reuse of existing code, object-oriented analysis, and refactoring. The platform can communicate both with humans and machines. It communicates with machines because causets are executable algorithms that can run directly in an interpreter or be

compiled to regular machine code, once suitable interpreters and compilers are developed. It also communicates directly with humans because transformations between causets and programming languages are possible, both ways, and will be fully automated once suitable translators become available. See (Pissanetzky, 2009, Sec. III) for details, and Section 4.3 in this paper for a small example.

3.7 Random number generators

A random number generator works as a source for causets. The system prompts the generator and obtains a random number in response. That's a causal relation. The response of the system to the *value* of the random number is a collection of causal sequences. To each value there corresponds a different causal sequence, and they all can be represented as a single causal set or used to grow an existing one. Note that random numbers used in computers are always finite integers, even if the generator is a chip based on truly random quantum noise, so the condition that causets are finite is not violated.

The same considerations apply to any random interactions or unexpected events that may happen inside the system being modeled. For example, a radioactive nucleus may suddenly decay, where the term "suddenly" gives rise to two causal sequences, one representing the reaction of the system to the decay, and the other representing the normal evolution of the system for the case where the decay has not happened.

3.8 Partitions, inference

A *partition* of a set S is a collection of subsets of S such that every element of S belongs to exactly one of the subsets. If set S has a partial order ω, then ω will *induce* a partial order, say ω', among the subsets. If the collection of subsets is designated as set S', so that the elements of S' are the subsets of S, then S' together with partial order $\omega\prime$ is a causet. Partition of a causet followed with order induction generates another smaller causet.

Causal inference (CI), discussed in Section 2.1, systematically generates partitions of a given causet with certain properties of invariance. CI is an important mechanism for causet generation.

3.9 Section closing

Section 3 is about using CL to solve problems. It may be useful to close it with the following general remark. Assume you are in doubt. You are working with CL and you are not sure how to interact with it and get it to do something useful. Then, I recommend the following state of mind: imagine you are working with a human. An infant, a student, a professional. Just imagine. I am not saying CL will do what they do, but just imagine. If you work with a human, you would either teach her, or make sure she already knows whatever you need her to know.

With CL, it is just the same. You train it. I have done it, many times, see my computational experiments. CL does not care about meaning (in first-order CL). If you have a problem, you train it with the knowledge *pertinent* to that problem. Of course, it must be *all* the pertinent knowledge, but there is no need to dig any deeper, for example there is no need for the CL to learn everything that an infant is supposed to learn. If you are solving a problem in self-programming, there is no need for the CL to know how to grab or understand a natural language. If the CL is for a washing

machine, it does not need to know high Mathematics. Just like humans. But unlike humans, once a CL prototype for a washing machine is built, copies can be made and mass-produced.

If you think of CL as if it were a human, then my computational experiments, reported or referenced in this paper, will serve as general directions. You will be building prototypes. Mass-production will come later.

4. Implementing Causal Logic on a Computer

In a physical system, emergence and self-organization happen naturally. According to the Second Law of Thermodynamics, the system looses excess free energy to its surroundings. There follows a decrease of the system's entropy and an increase of order, and the system converges to its attractors. Instead, in a computer implementation of causal logic, it becomes necessary to define a computer model and to simulate the loss of entropy by minimizing the action functional of Eq. (2). Self-organization will follow, as the model system converges to its attractors.

The definition of AGI as a machine with a human level of intelligence calls for a model consisting at least of "the world", "the machine", and some form of input and output that allows the machine to learn about the world and act upon it. Input and output are required to be represented as causal sets. The problem of representing physical systems as causal sets has been extensively considered in Section 3 and the corresponding subsections. The output stream is also a causet. Causets are finite-domain algorithms, or finite-domain computer programs that can directly drive actuators. A virtual machine that can be installed on any computer, receive a causet as input, minimize the functional to generate the self-organized structures, and output a causet that can drive actuators, is discussed in the next Section.

4.1 The virtual machine

The proposed AGI virtual machine is a host-guest system (Pissanetzky, 2010). It consists of the following:

(1) A causal set (causet), used as a data structure, or memory. Saying that a causet is part of the virtual machine is also saying that the causet's mathematical properties are incorporated in the machine, without the need for any heuristics. These properties are extensively discussed in several sections of this paper.

(2) An input stream carrying causal information from the outside world. This input stream can originate from a variety of sources, including sensors and senses. The possible sources of causal input for the virtual machine are discussed in Section 3.

(3) The functional for causets, which is an intrinsic mathematical property of causets, as discussed in Section 2.1.

(4) A means for finding stationary points of the functional and their corresponding block systems. These block systems are the invariant representations of interest, which will be sent to the output stream or fed back for further processing. A regular computer is a suitable means for performing this task, but the possibility that certain natural systems can perform the same task can not be excluded.

(5) An output stream, where causets representing the invariant representations generated by the machine are communicated to the outside world or fed back to the machine as input for further processing.

A prototype virtual machine installed on a personal computer has been used for all the computational experiments reported in this paper.

In the machine, the input stream brings causal information from the outside world. The causet implemented by the virtual machine grows as it *learns* that information. The process that minimizes the functional self-organizes the information by associating elements and organizing them into blocks, and generates the invariant representations as a hierarchy of block systems. The block systems are algorithms, or self-programs, automatically generated by the virtual machine from the causal input and ready to be compiled and executed on a regular computer or sent to a robotic actuator.

An analysis of some of causet's mathematical properties and their expected impact on Artificial Intelligence and Software Engineering is available (Pissanetzky, 2011a). A chronology of discovery is also included.

4.2 A new version of the Scope Constriction Algorithm

The scope constriction algorithm (SCA) is an algorithm used to find permutations of a given causet that minimize the functional of Eq. (2). Previous versions of SCA use a canonical matrix to represent the causet, and monitor the search by repeatedly calculating the entire value of the functional each time a change is made. But the matrix adds complexity to the representation and inefficiency to the process, and the need to calculate a global property each time it is affected by a local change adds further inefficiency. Furthermore, the fact that the functional is an essentially local property of each neighborhood in the causet, is ignored.

The present approach uses the causet itself, represented only by an integer number equal to the number of elements in the set, and a list of the precedence relations. This eliminates both sources of inefficiency and results in a local algorithm. By being local, the algorithm becomes amenable to massive parallelization, which means essentially constant execution time for any size of causet that does not exceed the number of processors, and reduction of execution time for larger causets by a factor equal to the number of processors.

The new version presented here is easier to explain by considering the causet as a linear *neural network*, where neurons correspond to elements of the causet, and causal relations are represented as directed "connections" between the neurons. The neurons act like AND gates. They activate only when *all* incoming connections are active. They are physically positioned along a straight line. Two adjacent neurons can switch their positions along the line, while preserving their connections. The connections can be thought of as elastic strings that "pull" from the neurons and cause them to switch positions. The value of the functional is the total length of all the connections, so, by switching positions within the constraints of the precedence relations, the neurons yield to the pull and effectively minimize the functional.

This description is intended for presentation purposes only, and does not imply a unique way of implementing the algorithm. In particular, it does not imply that the neurons must physically switch their positions. There may be many other ways. For example, two adjacent neurons could switch their connections instead of switching their positions. Or a second row of neurons could be used to map the positions of the neurons in the first row, without switching anything physically. Or, simply, each neuron in the row may keep a record of its position in the permutation and update this record each time a permutation takes place. Or, there could be an initially very large number of redundant connections and a mechanism that keeps only the shortest. In either case, the result would be the

same. The last mechanism may be the more realistic, as biological neurons in the brain are known to have up to 10,000 synapses per neuron. For our purpose here, which is to explain the algorithm, it is easier to think that neurons do switch their positions physically.

This version of the neural net is one-dimensional. Two-dimensional and three-dimensional versions are possible, and are possibly much more interesting than the 1D version, but have not been explored. Also, for comparison with the brain, a 3D version would possibly be more appropriate.

The initial set up of the algorithm requires the definition of a causet, such as the one given in Eq. (1), and let $n = |S|$.

Step 1. Initial setup. To each element e, associate three numbers: the number of predecessors $p(e)$, the number of successors $s(e)$, and their difference $p(e) - s(e)$. I will refer to this difference as the "net pull" on element e. Predecessors are said to 'pull up" from the element, successors "pull down" (pull left and pull right may also be used). These 3 numbers will remain constant during the entire algorithm, which guarantees the preservation of behavior and makes the algorithm more efficient because the 3 numbers are never recalculated. Finally, arrange all elements, or "neurons," along a line in the order of some arbitrary but legal permutation.

Step 2. Pair selection. Consider any arbitrary pair of adjacent neurons. If they are commutative, go to Step 3. Otherwise discard and go to Step 2 again. Two neurons are said to *commute* if they are not related by any precedence relation. If no commutative pairs are found, then stop.

Step 3. Commutation. Calculate the "differential pull", which is the net pull of the neuron on the left less the net pull of the neuron on the right of the pair. If the differential pull is negative, commute and go to step 2. A negative differential pull is said to be *profitable*, because the commutation reduces the cost of the functional. If the differential pull is positive, do not commute, and go to step 2. If it is 0, then commutation is optional. Usually, commutation is effected if some profit for doing so can be found. Otherwise it is not effected. Examples are given below. Go to step 2.

This completes the basic algorithm. A more advanced version should include the notion of *shift*, or *migration* of an element. Consider one element, say e, and its adjacent element to the right, say e'. Let also e'' be the element next to e' on its right. If the commutation (e, e') is performed, then element e becomes adjacent to element e''. If e and e'' are commutative, and the commutation is effected, then the net effect would be that e has shifted right for 2 steps. Element e can continue shifting right until it meets one of its successors, where it must stop.

The net shift may be very short or very long. Elements can also shift left. The shift of an element is characterized by its *direction* and its *extent*, the number of steps involved in the shift. When performing a shift, the algorithm has two undetermined parameters: the selection of seed element, and the direction of shift, left or right, for that step. Results may be different for different combinations of selections. This arbitrariness is actually an advantage, because it can be used to find many different least-cost permutations and populate the set $\Pi_{\min}(S, \omega)$ by simply randomizing both parameters. As explained above, it is necessary to have as many least-cost permutations as possible in set $\Pi_{\min}$, short of having them all, in order to obtain a good block system.

This algorithm is local. The actual value of the global cost is irrelevant. It is fully parallelizable, by assigning one adjacent pair per processor, and it is much simpler than the matrix-based version, because there is no need to access the tables of rows and columns in the matrix. The algorithm operates directly from the given causet, and preserves behavior.

When a causet contains more than one connected component, this algorithm will find them all and will create an initial block system with one block per component. The components are subsets of the causet, and can be given in any order, which is precisely the condition for a subset to be a

block. Finding a partition in components is very convenient. When it happens, it becomes possible to divide the problem into smaller subproblems, one per each component, and solve each one of them separately.

Experience gained with the study of small systems (Pissanetzky, 2011a, Sec. 4) indicates that a higher-order version of this algorithm that includes *block commutation* is necessary. Block commutation considers the commutation of two adjacent blocks of elements, at least one of which contains more than one element. It is possible for the commutation of two adjacent blocks to be profitable as a whole even though no profitable single-element commutations are available inside the blocks or at their boundary. Shifts, considered above, are an example of block commutation. Block commutation would follow the same rules outlined above, but is not further expanded here.

4.3 A Neural Network Implementation of Causal Inference

Because of the conjecture made in this paper that meaning comes from associations, and that associations require inference, so that the causal order of things is

$$\textit{inference} \rightarrow \textit{associations} \rightarrow \textit{structure} \rightarrow \textit{meaning},$$

and because CI *creates* associations and structure and therefore meaning, but ignores any previous meaning of the elements of the set, the presentation in this Section intentionally starts with a causet that appears meaningless for the reader. Of course, the interpretation of the results, including the meaning that the elements have for us humans, are discussed at the end of the Section.

The model presented here is *minimalist*. It extracts only a few features deemed basic and essential for the operation of a physical system. Minimalist models do not explain the system in full, but they serve as proof of principle and provide a foundation for further, more detailed exploits.

A researcher, working long before Newton, had observed the motion of a billiard ball in three dimensions, and had measured a number of parameters that appeared to describe the motion. He named the parameters arbitrarily, because, for him, they didn't have any meaning.

The researcher followed an approach known as a *thought experiment*. After interpolating his experimental points, he came up with 18 simple equations that represent the motion with good accuracy. To help the reader into a state of mind similar to that of the researcher, I will do the same thing: I will write the 18 equations, using arbitrary symbols, and not telling what their modern

interpretation is until the end of the Section. The 18 equations are:

$$\begin{array}{rcl} a & = & \kappa \times \tau \\ b & = & \theta \times \mu \\ c & = & \eta \times \nu \\ d & = & \eta \times \lambda \\ e & = & \eta \times \mu \\ f & = & \kappa \times \phi \\ g & = & \kappa \times \sigma \\ h & = & \theta \times \lambda \\ i & = & \theta \times \nu \\ j & = & c + f \\ \alpha & = & \tau + b \\ \beta & = & \sigma + h \\ k & = & e + a \\ \gamma & = & \pi + k \\ \ell & = & d + g \\ \delta & = & \xi + \ell \\ \epsilon & = & \phi + i \\ \zeta & = & \rho + j \end{array} \tag{20}$$

These 18 equations represent the information that the virtual machine is initially trained with. They can also be viewed as a computer program, written in this case in single-assignment C, where each equation represents one statement of the program. Anyone with a minimum of programming experience will soon realize that this program was written by a very inexperienced developer. But this is precisely the point I am trying to convey: that the virtual machine of Section 4.1, with the information contained in Eq. (20) as input, and nothing else, can do the same work that an experienced programmer would do: organize the code, and design classes and objects. CL can do the entire programming job by itself, without the need for any additional human expertise. This is how self-programming should be defined.

The causal set for the program above is Σ, defined as follows:

$$\begin{array}{rcl} S & = & \{a, b, c, d, e, f, g, h, i, j, k, \ell, \alpha, \beta, \gamma, \delta, \epsilon, \zeta\} \\ \omega & = & \{a \prec k, b \prec \alpha, c \prec j, d \prec \ell, e \prec k, f \prec j, g \prec \ell, h \prec \beta, i \prec \epsilon, j \prec \zeta, k \prec \gamma, \ell \prec \delta\} \\ \Sigma & = & (S, \omega). \end{array} \tag{21}$$

The input variables, defined as variables that participate in the equations but do not appear in set S, and are assumed to be given, are: $\kappa, \theta, \eta, \tau, \sigma, \pi, \xi, \phi, \rho, \mu, \nu, \lambda$.

The application of the SCA algorithm to the causet of Eq. (21) is displayed in full detail in Fig. 3. The original set S is listed in any arbitrary, but legal order, in line 1. In this case, the order is that of the given equations. The pull for each element is listed below the name of the element. This pull is carried along with the element as the element shifts. Pairs eligible for commutation are those with a negative pull. In this case, the only pair is (i, j), which has a differential pull of -2. Once i and j are commuted, i becomes adjacent with α. Pair (i, α) again has a differential pull of -2, so it is also commuted, causing i to become adjacent with β, and so on. In all, element i shifts all the way to ϵ, but not further because pair (i, ϵ) is not commutative. Line 2 indicates the result.

In line 2, h is shifted right to the left of β. In line 3, g is shifted right just left of ℓ. Note that, at some point in the course of this last shift, g reaches h, where the differential pull of (g, h) is 0.

When the differential pull is 0, commutation is optional, and a decision must be made on whether to commute or not to commute. In this case, it is convenient to effect the commutation and allow g to reach β, where the pull of pair (g, β) is favorable. The same strategy is applied many times in the course of the process.

In line 4, since pair (f, j) is not commutative, e is shifted right to k. In line 5, d is shifted right to ℓ. In line 6, since commuting pair (c, f) would be worthless, b is shifted right to just left of α. In line 7, a is shifted right to e. The boundary for a is k, but it is not worth commuting a with e. Finally, in line 8, since f can not be shifted right, ζ is shifted left all the way to the right of e, where it stays. Line 9 contains the final least-cost permutation. In this case, no more profitable shifts are possible.

In this process, a particular choice was made for the direction and extent of each shift. But the choice is arbitrary, and the resulting permutation depends on the choice. By randomizing the selection and extent of each shift, or, preferably, of each commutation, different permutations will be obtained. This idea constitutes a systematic procedure for obtaining the set of all least-cost permutations of set S, which is required for the calculation of the block system. See an example in Eq. (10). In this particular case, the given causet has $n = 18$ elements and $r = 12$ relations.

1. Original	a	b	c	d	e	f	g	h	i	j	α	β	k	γ	ℓ	δ	ϵ	ζ
	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1	1	1
2. Shift i right to ϵ	a	b	c	d	e	f	g	h	j	α	β	k	γ	ℓ	δ	i	ϵ	ζ
	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1	-1	1	1
3. Shift h right to β	a	b	c	d	e	f	g	j	α	h	β	k	γ	ℓ	δ	i	ϵ	ζ
	-1	-1	-1	-1	-1	-1	-1	1	1	-1	1	1	1	1	1	-1	1	1
4. Shift g right to ℓ	a	b	c	d	e	f	j	α	h	β	k	γ	g	ℓ	δ	i	ϵ	ζ
	-1	-1	-1	-1	-1	-1	1	1	-1	1	1	1	-1	1	1	-1	1	1
5. Shift e right to k	a	b	c	d	f	j	α	h	β	e	k	γ	g	ℓ	δ	i	ϵ	ζ
	-1	-1	-1	-1	-1	1	1	-1	1	-1	1	1	-1	1	1	-1	1	1
6. Shift d right to ℓ	a	b	c	f	j	α	h	β	e	k	γ	g	d	ℓ	δ	i	ϵ	ζ
	-1	-1	-1	-1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	1	1
7. Shift b right to α	a	c	f	j	b	α	h	β	e	k	γ	g	d	ℓ	δ	i	ϵ	ζ
	-1	-1	-1	1	-1	1	-1	1	-1	1	1	-1	-1	1	1	-1	1	1
8. Shift a right to e	c	f	j	b	α	h	β	a	e	k	γ	g	d	ℓ	δ	i	ϵ	ζ
	-1	-1	1	-1	1	-1	1	-1	-1	1	1	-1	-1	1	1	-1	1	1
9. Shift ζ left to j	c	f	j	ζ	b	α	h	β	a	e	k	γ	g	d	ℓ	δ	i	ϵ
	-1	-1	1	1	-1	1	-1	1	-1	-1	1	1	-1	-1	1	1	-1	1

Figure 3: The 9 steps performed by the CI algorithm SCA to find one least-cost permutation.

The number m of connected components must satisfy $m \geq n - r$, which is 6 in this case, and 6 components have been found. However, in a general case, the actual number is not known beforehand. It must be determined, and the components identified. The procedure to do so is very simple, once some permutation with the least cost has been identified by SCA. Figure 4 illustrates the procedure. The central row of the table lists the permutation from line 9 of Fig. 3. The top row lists the number of connections that "pull up" from each element, and the bottom row lists those that "pull-down." Connected components are disconnected from each other. A connected

pull up	0	0	2	1	0	1	0	1	0	0	2	1	0	0	2	1	0	1		
elements	c	f	j	ζ	b	α	h	β	a	e	k	γ	g	d	ℓ	δ	i	ϵ		
pull down	1	1	1	0	1	0	1	0	1	1	1	0	1	1	1	0	1	0		

Figure 4: The connected components for the causal set of Eq. (21). The presence of a 0 at the top of some column, and a 0 at the bottom of the following column identifies a boundary between two connected components, located between the two columns. This figure represents the structures existing but hidden in the original information, Eqs. (20).

component must have edges with 0 connections pulling up, and 0 pulling down. The table very clearly shows 6 components, 3 with 2 elements each, and 3 with 4 elements each, as indicated by the vertical divisions. In addition, all three 2-element components have $(0, 1)$ of them pulling up, and $(1, 0)$ pulling down, indicating that they are *objects* of the same *class*. The same happens with all three 4-element components, where the pull-up and pull-down counts are $(0, 0, 2, 1)$ and $(1, 1, 1, 0)$, respectively. Furthermore, there exists a one-one association between each 2-element object and each 4-element object. In conclusion, the SCA algorithm has identified 6 connected components, consisting of two classes with 3 objects each. Meaning can now be derived from the various structures and associations just described. For example, they can be named, and those names will be "meaningful." The 4-element class could be named "coordinate," and its 3 objects as r_x-component, r_y-component, and r_z-component. The 2-element class would be "velocity," with elements v_x, v_y, and v_z. The associations relate r_x with v_x, r_y with v_y, and r_z with v_z, effectively separating the 6 objects into 3 groups of 2 objects each. This property is known in Physics as the law of independence of the components of motion. It is usually taught in Physics-101.

All these conclusions have not been *created* by CL. They obviously exist in the original observations, Eqs. (20), but they are not recognizable. CL is the process that *organizes* the information by creating meaningful structure. This process is also known as *binding*, or sometimes as *binding of the qualia*. The binding process is not limited to software. There are many other applications, but this is a paper on software, and the other applications are beyond scope.

Just as Eqs. (20) are a computer program, so also are the structures defined in Fig. (4). But with one difference: an object-oriented (OO) design exists in the last case, which did not exist in the first. CL has created a full OO design for an originally very poorly organized and non-OO piece of software. CL, which is not a program and has not even been told what OO is, has, nevertheless and of its own initiative, created the OO design. It is possible to write a program for converting Fig. (4) directly into Java code.

As explained above, Eqs. (20) were intentionally written in a confusing notation. This was done to insist on the point that CL does not depend on meaning. Quite on the contrary, CL is believed to be the source of meaning. To complete this argument, it is now necessary to explain the physical meaning of the various variables, as understood by a human physicist, and compare with the meaning of the results obtained by CL. The problem at hand is that of a billiard ball of mass m whose center of mass is initially at position $\mathbf{r}_0$ and has velocity $\mathbf{v}_0$. The ball is subject to the action of a constant force $\mathbf{F}$ (bold-face characters indicate vectors). The problem is to find the position $\mathbf{r}$ and velocity $\mathbf{v}$ of the center of mass after a time interval Δt and under the action of a constant force.

This problem is solved using Newton's equations:

$$\begin{aligned} r_x &= r_{x0} + v_{x0}\Delta t + \frac{F_x}{2m}\Delta t^2 \\ r_y &= r_{y0} + v_{y0}\Delta t + \frac{F_y}{2m}\Delta t^2 \\ r_z &= r_{z0} + v_{z0}\Delta t + \frac{F_z}{2m}\Delta t^2 \end{aligned} \tag{22}$$

where subscripts x, y, z designate the components of the respective vectors. The input data for Eqs. (20) are the initial values, as follows: $\xi = r_{x0}, \rho = r_{y0}, \pi = r_{z0}, \sigma = v_{x0}, \phi = v_{y0}, \tau = v_{z0}, \lambda = F_x, \nu = F_y, \mu = F_z, \eta = \Delta t^2/2m, \kappa = \Delta t, \theta = \Delta t/m$. The output values are the results: $r_x = \delta, r_y = \zeta, r_z = \gamma, v_x = \beta, v_y = \epsilon, v_z = \alpha$. Expressions for the remaining variables easily follow. A preliminary version of this experiment was published (Pissanetzky, 2010, Sec. VIII-A).

Eqs. (22) reveal that the motion of the billiard ball is described by three equations, one for each component of motion, and that two independent variables per component are necessary to properly describe the motion. That's 3 groups of 2 objects each. The partition in connected components in Fig. 4 also contains the same two conclusions.

4.4 The brain as a point of reference for AGI and Self-Programming

Currently, the brain is the only known physical system that is intelligent. In AGI conferences, the definition of AGI is "a machine with a human level of intelligence." This definition makes the brain a necessary point of reference for AGI studies. In the case of self-programming, the situation is a little different. Self-programming is viewed as a component of AGI capable of learning whole new sets of skills from experience, possibly the result of a constructivist approach focused on self-organized code and architecture. This definition contains no direct reference to humans, but indirectly involves the human brain, which is still the only known physical system capable of self-organizing and self-programming from experience.

Self-programming is one of the GUAPs, or Great Unsolved Automation Problems of Software Engineering (Pissanetzky, 2011b). Other examples are OO design, image recognition, systems integration, the semantic web, and many others. The GUAPs share two characteristics: they are (relatively) easy for humans but very difficult for computers to solve, and they all require some form of inference with the ability to represent general experience of the world and self-organize it into meaningful structures, which currently only the brain appears to possess.

In this paper, I have proposed causal logic (CL) as a candidate for automating the GUAPs, including self-programming. Although the implementation of CL and that of the brain are different, a number of common features can be found if a comparison between CL function and brain function is made. Some of them follow:

- CL is a form of inference with the ability to learn and represent general experience of the world and self-organize it into structures. The brain has the ability to learn and represent general experience of the world, and to infer new facts from that experience.
- CL's structures and the brain's representations are observable and invariant under transformations.
- Experience is acquired by CL as input, not as program. Experience is acquired by the brain as input, not as program.
- CL is a host-guest system (Pissanetzky, 2010). The brain can also be viewed as a host-guest system, where the wetware is the host and knowledge is the guest.

• CL is natural, a property of nature, not man-made. It has been experimentally observed, not derived from another theory, not engineered or designed for any particular purpose. The brain is a natural organ, a not man-made. It evolved by natural selection, because individuals with bigger brains survived better, not because evolution intended to "engineer" it for some particular purpose.

• The theory of CL is consistent with the foundations of theoretical Physics. The brain is a physical system, that obeys the laws of Physics.

• CL can be viewed as a neural network (Section 4.3), where elements of the causet correspond to neurons and causal relations to synaptic connections (of the on-off type), and the connections serve as memory to store knowledge.

• CL admits of a massively parallel implementation on specialized hardware, where execution time is approximately independent of the size of the problem. This is possible because the functional is naturally local, not because it had been designed to be that way. By being local, a big problem can be subdivided in many small problems, each of which can be solved by a local processor, and all processors work at the same time. The "massive parallelism" of the brain is legendary and doesn't need an introduction.

In view of all of which, I have conjectured (Pissanetzky, 2011a, conjecture K2), but did not claim, that CL is the logic used by high brain function to perform those tasks. This conjecture can only be addressed experimentally. CL has not been actually observed in the brain, although some indirect evidence is available such as the existence of neural cliques (Lin, Osan, and Tsien, 2006), that remind one of the clusters of artificial neurons that form in the neural network of Section 4.3, and the existence of a regular 3D grid structure underlying the complexity of the primate brain (Wedeen et al., March 2012).

To support this research, I have also proposed an experimental program based on computational experiments that avoid the complexities of implementation of the brain by considering it as a host-guest system with an input and a corresponding output. This approach is discussed in the next Section.

4.5 Computational experiments

I believe that CL does actually exist in the brain, and I trust that, some day, real brain experiments will be performed to verify it that is true. And if so, how does it work and exactly where it resides. I also trust that, some day, enough observational knowledge about the brain will exist to create a computer model sufficiently detailed to support such studies. None of this has happened yet, so for now, one option is to consider a computational approach with experiments that treat the brain as a black box with an input and an output, and avoid the need for a detailed model. Such experiments should be viewed as a test for CL, intended to verify whether CL works like high brain function. But they can also shed light in the opposite direction. For, if it turns out that CL does indeed work like high brain function, then it would also be true that high brain function works like CL. And if it does, one will want to know exactly what makes the implementation of high brain function in the brain capable of working like CL.

The conceptual setup for the experiments consists of a causal set describing a certain body of knowledge. The causal set is supplied to a human analyst, and, as input, to an installation of the CL virtual machine. It is assumed that the virtual machine initially contains no stored knowledge, meaning that *all* necessary knowledge is contained in the input causet. After processing, the two outputs are compared. In order to ensure that the measurements are objective and reproducible,

published material is used as the source of the necessary information. Three suitable types of sources of such material have been identified, and, fortunately, they are abundant and easily accessible. They are: object-oriented software designs, theories of Physics, and visual images. However, depending on their particular characteristics, they may be used in different ways in their corresponding experiments.

I have performed only one experiment using a visual image. The input is a digitized picture, in this case a set of black points on a white background, and the output is the resulting structural hierarchy. Actual recognition would have required to keep a database of previously generated hierarchies in memory, which is not allowed by the initial condition, so machine recognition was not included in the experiment. To deal with this limitation, human observers (in one case the entire audience in a workshop), were asked to interpret a display of the digitized image, which they had never seen before, and describe their reaction. The experiment is described in Section 4.5 of Pissanetzky (2011a), and it showed satisfactory results.

I have performed several experiments in the area of object-oriented software design. In a typical case, a human analyst receives a problem statement, for example describing a company, and possibly some additional explanations from the company's subject-matter engineers. The analyst is not expected to know anything about the company prior to receiving the document or the explanations, so the initial condition is satisfied. Actual development is not included in the experiments in order to avoid the need for a programming language, which developers usually learn beforehand and keep in memory at the time they engage in the analysis. Since the actual information received by the analyst is usually considered confidential and is published, I had to recreate it by applying function $\mathcal{E}^{-1}$ to the published structures. This involves a behavior-preserving process where all structure is removed from the software (for example by translating from Java to single-assignment C), and the resulting causal set is randomized to destroy any remaining order before using it as input to CL.

I have performed several experiments with software. One them is discussed in Section 4.3. I call this experiment the *foundational experiment* for CL, because it is the experiment where CL was discovered. A detailed account of the process of discovery is available in the Introduction of Pissanetzky (2011a). Based on the fact that natural systems self-organize, and computers and programs do not, I knew I had to look for self-organization in a natural system. I chose myself as the system, the high function in my own brain. I intended only for the experiment to be simple enough to search for CL, but not necessarily objective or observer-independent. It is, however, easily reproducible by anyone. For the setup, I selected one of the GUAPs, and I wanted to find out why it was so easy for people but so difficult for computers. I started with the simplest problem I could find, the one discussed in Section 4.3.

Since then, a number of observer-independent computational experiments have been conducted in the area of software self-organization, all of them based on independently created published material, described in full detail by other authors and publicly available. The final product, obtained by a human analyst, was converted first to an unstructured causet by application of function $\mathcal{E}^{-1}$, and a process of *learning* was used to train the CL virtual machine with that material. CL was, then, applied to generate structures, and the final structures obtained by CL were compared with the original man-made structures.

One of such experiments is a rather extensive and detailed case study on object-oriented analysis and design, supported by a complete set of files, available as supplementary material. This case study is based on a publicly available Java program that describes a real-world token-ring, and is

used to teach refactoring in European universities. The reader, particularly if interested in self-programming, is encouraged to peruse this material.

Another experiment on software is the example on parallel programming discussed in Section 2.2 of this paper.

In the area of theories of Physics, I have performed only two experiments. One of them is the foundational experiment, which represents a law of Physics based on Newton's equations of motion. The other is a set of equations known as the Euler equations for the motion of a rigid body. This last one was not published. There was also a study of hundreds of small causal systems, as reported in Section 4 of Pissanetzky (2011a), intended not so much as experiments but as an analysis of the properties of CL.

Results were satisfactory in all cases. Not one single disagreement was found. Many of these experiments are toy problems. But the size of an experiment is not a good measure of its value. The size depends on factors such as the power of the computer, the amount of money available to buy a bigger one, and the time available for developing and perfecting the programs. Better measures are the significance of the principle underlying the work, and the scalability of the process. By these measures, the computational experiments reported here are very strong.

4.6 Supplementary Material

The present paper is supplemented by a number of files containing material that is either too detailed or less directly related to self-programming. This material begins with an overview of the theory (Pissanetzky, 2012b), the reading of which requires a background in Physics, continues with an article on verification (Pissanetzky, 2012f), where predictions obtained from the theory are compared against experimental results and heuristic theories from several other disciplines, including Neuroscience, Biophysics, Evolution, Artificial Intelligence, Computer Science, even Google patents and Philosophy, and a historical overview of previous work (Pissanetzky, 2012c). Other articles may be added in the future. Of critical importance is the prediction that dendritic trees in the brain must be optimally short, made in Section 3.8 of Pissanetzky (2011a). This prediction was independently confirmed by Cuntz, Mathy, and Häusser (June 2012), who proposed a 2/3 power law for all parts of all brains across species, with extensive experimental support and replacing a previous 4/3 power law. This is a dramatic confirmation of the theory proposed in the present paper.

The supplementary material also includes an extensive case study (Pissanetzky, 2012a), based on a publicly available Java program, where the complete process is explained in full detail, and all the intermediate files are included. And a study on image recognition (Pissanetzky, 2012d), where edges separating groups of points on a plane are identified without writing a single line of code specific to that problem.

5. Conclusions and Outlook

In this paper, I have proposed a constructivist approach to self-programming in the context of a new mathematical logic, to be known as *causal logic*. The logic consists of a *causal set* that represents knowledge about the world, a *functional* that depends only on the causal set, and a process of inference, to be known as *causal inference*, that minimizes the value of the functional. The minimization creates *associations* among elements of the causal set and uses them to *self-organize* the knowledge and create *order* and invariant *structures* over the causal set.

Having established all that, I proceeded to argue in support of my claim that causal logic can be used for reasoning with computer code. I noted that, under certain conditions, *any algorithm is a causal set*, because it satisfies the definition. And I also noted that *any program is a causal set*, because the program is a finite algorithm. If put in reverse, these statements read that causal sets themselves are executable programs, and that anything said about causal sets or causal logic also applies to computer programs. Which implies that *causal logic applies to computer programs* and qualifies as a *mathematical approach to computer programming* that allows reasoning with code.

I have further noted that the hierarchies of structures created by causal logic represent *classes of objects and inheritance relationships* among them, that the blocks in the levels of the hierarchy are *objects* of those classes, and that an entire hierarchy can be considered as a UML class diagram representing the originally given knowledge in an organized and structured way. To qualify the reference to "originally given knowledge," I examined a number of sources of knowledge about the world, and the particular ways in which these various sources always produce causal information about the world that can be expressed in the form of a causal set, or a computer program. By enumerating several of those sources, I meant to imply that their number includes nearly every process used in computation to represent knowledge and work with it. Based on which, I have noted that causal sets represent a means for universal connectivity, and proposed a *software development platform* based on causal sets that can communicate with man by way of understandable programming languages, and with machine by way of executable code.

Still pursuing the same approach, I have considered traditional computer programming as a process where a human analyst learns from the world, codes the acquired knowledge, and reasons with the code in order to organize it and find structures for an algorithm. Finally, the analyst translates the algorithm to a programming language that both man and machine can understand. In view of all of which, I have proposed that a self-programming machine should be able to do the same.

But machines cannot reason with code. People can. The inability of machines to reason with code has left the deepest scar in the whole of modern computation: the separation of input and program. Input is information about some system in the outside world. Program is information about the same system in the outside world, only this time the information has to be reasoned upon by a human analyst. Information about the system of interest has been artificially separated into two parts, one for the machine, the other for the programmer. A self-programming machine should be able to do the reasoning on its own and not depend on a human programmer.

The machine proposed in this paper has gone a long way in that direction. It has no program specific to the system under analysis. It still has, at least in this version, a program to deal with input/output and to minimize the functional, but the reasons for that are only circumstantial, dictated by limitations in time and resources. With time, that program too will become part of the input. The unification of input and program is now possible in view of the discovery of causal inference and the new ability of machines to reason with code.

With those goals in mind, I have presented a virtual machine that implements the inference and needs no problem-specific program. I have discussed an implementation of the virtual machine as a neural network, and presented and fully explained a direct application of the neural network to a real problem. I have also presented or referenced a number of examples or experiments that illustrate actual experience in self-programming with real-world problems, without any need for human intervention. They include a case study in Java code for a token ring, an example in parallel programming where tasks are assigned to processors, and a small example in image recognition

that serves as a proof of principle. The experiments cover major tasks in Software Engineering such as object-oriented analysis, refactoring, and system integration. All these tasks are in the area of computer programming, if done by a human developer, or of self-programming, if done by the machine.

All the technology needed to proceed with further development of these projects is currently available. An easy (but slow) implementation of causal logic on a personal computer is accessible to anyone with computer experience. To make it faster, a massively parallel neural network-style implementation is necessary, using specialized hardware, and following the guidelines given in the corresponding section. It may be possible to develop a hardware unit for causal logic that plugs in a USB port, and make it commercial. The parallelization of causal logic is, quite possibly, one of the very first tasks that needs to be undertaken next.

The development of Causal Logic (CL) is just beginning. This long paper is only an introduction. There is a very large number of connections with a variety of branches of science, each one of which needs to be explored and appropriately accounted for. A large volume of future research is expected after publication of this paper. My vision is that, with time, mass-produced CL machines trained for different purposes will become available, in the form of LSI chips. Users or manufacturers will be able to specialize the machines by providing additional training specific to their interests, or even directly from sensors. For now, our task is to make all that come into existence.

References

Bolognesi, T. 2010. Causal Sets from Simple Models of Computation. *arXiv* 1004(3128):1–33. arXiv:1004.3128 Available electronically from http://arxiv.org/abs/1004.3128.

Caspard, N.; Leclerc, B.; and Monjardet, B. 2012. *Finite Ordered Sets*. New York: Cambridge University Press.

Cuntz, H.; Mathy, A.; and Häusser, M. June 2012. A scaling law derived from optimal dendritic wiring. *PNAS, 2012, DOI: 10.1073/pnas.1200430109* 1–5. Available electronically from http://www.pnas.org/content/109/27/11014.

Hawkins, J. 2004. *On Intelligence*. New york: Times Books.

Hofstadter, D. R. 1985. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. New York: Basic Books, Inc.

Lin, L.; Osan, R.; and Tsien, J. Z. 2006. Organizing principles of real-time memory encoding: neural clique assemblies and universal neural codes. *Trends in Neuroscience* 29(1):48–57. Available electronically from http://www.ncbi.nlm.nih.gov/pubmed/16325278.

Noether, E. 1918. Invariant Variation Problems. *Nachr. d. König. Gesellsch. d. Wiss. zu Göttingen* Math-phys 1918:235–257. English translation: arXiv:physics/0503066v1 Available electronically from http://arxiv.org/pdf/physics/0503066s

Opdyke, W. F. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation, Dep. of Computer Science, Univ. of Illinois, Urbana-Champaign, Illinois, USA. Available electronically from http://www-public.it-sudparis.eu/ gibson/Teaching/CSC7302/ReadingMaterial/Opdyke92.pdf.

Pissanetzky, S. 1984. *Sparse Matrix Technology*. London: Academic Press.

Pissanetzky, S. 2009. A new Universal Model of Computation and its Contribution to Learning, Intelligence, Parallelism, Ontologies, Refactoring, and the Sharing of Resources. *Int. J. of Information and Mathematical Sciences* 5:143–173. Available electronically from https://www.waset.org/journals/ijims/v5/v5-2-17.pdf.

Pissanetzky, S. 2010. Coupled Dynamics in Host-Guest Complex Systems Duplicates Emergent Behavior in the Brain. *World Academy of Science, Engineering, and Technology* 68:1–9. Available electronically from https://www.waset.org/journals/waset/v44/v44-1.pdf.

Pissanetzky, S. 2011a. Emergence and Self-organization in Partially Ordered Sets. *Complexity* 17(2):19–38.

Pissanetzky, S. 2011b. Emergent inference and the future of NASA. Workshop, NASA, NASA Gilruth Center, Johnson Space Center, Clear Lake, TX. Available electronically at http://www.scicontrols.com/Publications/AbstractNASA2011.pdf.

Pissanetzky, S. 2011c. Structural Emergence in Partially Ordered Sets is the Key to Intelligence. In *Artificial General Intelligence*, 92–101. Available electronically from http://dl.acm.org/citation.cfm?id=2032884.

Pissanetzky, S. 2012a. A case study: the European Example. Available electronically at http://www.scicontrols.com/Articles/EuropeanExample.htm.

Pissanetzky, S. 2012b. The Detailed Dynamics of Dynamical Systems. Available electronically at http://www.scicontrols.com/Articles/TheoryOfDetailedDynamics.htm.

Pissanetzky, S. 2012c. Overview of Previous Work on Causal Logic. Available electronically at http://www.scicontrols.com/Articles/OverviewOfPreviousWork.htm.

Pissanetzky, S. 2012d. Separating points. Available electronically at http://www.scicontrols.com/Articles/PointSeparation.htm.

Pissanetzky, S. 2012e. Symmetry, structure, and causets in discrete quantum gravity. *Bulletin of the American Physical Society* 57(2):H1.0005. Available electronically from http://meeting.aps.org/Meeting/TSS12/Event/173348.

Pissanetzky, S. 2012f. Verification of the Theory of Detailed Dynamics. Available electronically at http://www.scicontrols.com/Articles/VerificationForTheoryOfDetailedDynamics.htm.

Schröder, B. S. W. 2002. *Ordered sets*. Boston, USA: Birkhäuser.

Shafer, G. 1998. Causal Logic. Available electronically from http://www.glennshafer.com/assets/downloads/articles/article62.pdf.

Wedeen, V. J.; Rosene, D. L.; Wang, R.; Dai, G.; Mortazavi, F.; Hagmann, P.; Kaas, J. H.; and Tseng, W. I. March 2012. The geometric structure of the brain fiber pathways. *Science* DOI: 10.1126/science.1215280:1628–1634. Available electronically from http://www.sciencemag.org/content/335/6076/1628.abstract.