METAPHILOSOPHY

Published by Blackwell Publishing Ltd, 9600 Garsington Road, Oxford OX4 2DQ, UK, and 350 Main Street, Malden, MA 02148, USA
METAPHILOSOPHY
Vol. 41, No. 3, April 2010
0026-1068

ABSTRACTION, LAW, AND FREEDOM IN COMPUTER SCIENCE

TIMOTHY COLBURN AND GARY SHUTE

Abstract: Laws of computer science are prescriptive in nature but can have descriptive analogs in the physical sciences. Here, we describe a law of conservation of information in network programming, and various laws of computational motion (invariants) for programming in general, along with their pedagogical utility. Invariants specify constraints on objects in abstract computational worlds, so we describe language and data abstraction employed by software developers and compare them to Floridi's concept of levels of abstraction. We also consider Floridi's structural account of reality and its fit for describing abstract computational worlds. Being abstract, such worlds are products of programmers' creative imaginations, so any "laws" in these worlds are easily broken. The worlds of computational objects need laws in the form of self-prescribed invariants, but the suspension of these laws might be creative acts. Bending the rules of abstract reality facilitates algorithm design, as we demonstrate through the example of search trees.

Keywords: abstraction, computer science, freedom, invariants, law.

Introduction

Despite its title, this essay has nothing to do with legal, moral, or political issues. Had the title been "Abstraction, Law, and Freedom in *Cyberspace*," it no doubt would have had something to do with such issues, since it would have been about the behavior of people. But our concern is with computer science as a science, and since the subject matter of computer science is interaction patterns among various computational abstractions, we address the concept of "law" in the scientific, rather than the legal, sense.

Our topic bears on Luciano Floridi's work in philosophy of information in two ways. First, to understand the role of law in computer science, one must understand the world in which computer scientists work, a world consisting primarily of abstractions. Floridi, acknowledging a certain debt to computer science, has recently advocated for "the method of levels of abstraction" (Floridi 2008a) as an approach to philosophical investigation in general. Second, our investigation into the concept of law

in computer science reveals a paradigmatic case of Floridi's "informational nature of reality" (Floridi 2008b). In characterizing reality informationally, Floridi appeals again, in part, to a computing discipline—namely, software engineering and its current focus on object-oriented programming. It is fruitful to consider, while we explicate our account of abstraction and law in computer science, whether it impacts the related but more general philosophical claims of Floridi.

As a science that deals essentially with abstractions, computer science *creates its own subject matter*. The programs, algorithms, data structures, and other objects of computer science are abstractions subject to logical—but not physical—constraints. We therefore expect its laws to be different in substantial ways from laws in the natural sciences. The laws of computer science go beyond the traditional concept of laws as merely *descriptive*, allowing *prescriptive* laws that constrain programmers while also describing in a traditional scientific lawlike way the behavior of physically realized computational processes.

Laws of nature describe phenomena in ways that are general and spatially and temporally unrestricted, allowing us to explain and predict specific events in time. Computer scientists also need to explain and predict specific events, but these events occur in computational worlds of their own making. What they need is not laws that describe the phenomena they bring about; instead they need laws that prescribe constraints for their subject matter, keeping it within limits of their abstract understanding, so that the concrete processes that ultimately run on physical machines can be controlled.

In the section "Computer Science as the Master of Its Domain," we describe the language and data abstraction employed by software developers and compare them to Floridi's concept of levels of abstraction. Floridi also borrows the technical concept of "objects" from the object-oriented programming of software engineering, and we consider how objects fit with a structural account of reality. In "The Concept of Law in Computer Science," we show how laws of computer science are prescriptive in nature but can have descriptive analogs in the physical sciences. We describe a law of conservation of information in network programming, and various laws of computational motion for programming in general. These latter laws often take the form of *invariants* in software design. In "Computer Science Laws as Invariants," we show how invariants can have both pedagogical and prescriptive effects by describing how invariants are enforced in a famous sorting algorithm conceived by C. A. R. Hoare.

A prescriptive law, as an invariant applied to a program, algorithm, or data structure, specifies constraints on objects in abstract computational worlds. Being abstract, computational worlds are products of programmers' creative imaginations, so any "laws" in these worlds are easily broken. For programmers, the breaking of a prescriptive law is

tantamount to the relaxing of a constraint. Sometimes, bending the rules of their abstract reality facilitates algorithm design, as we demonstrate finally through the example of search trees in the section "The Interplay of Freedom and Constraint."

Computer Science as the Master of Its Domain

Computer science is distinct from both natural and social science in that *it creates its own subject matter*. Natural science has nature and social science has human behavior as subject matter, but in neither case is nature or human behavior actually *created* by science; the two are *studied*, and observations made through such study are *explained*. Computer science, however, at once creates and studies abstractions in the form of procedures, data types, active objects, and the virtual machines that manipulate them.

Computer science shares with mathematics the distinction of studying primarily abstractions. However, as we argue in Colburn and Shute 2007, the primary objective of mathematics is the creation and manipulation of *inference structures*, while the primary objective of computer science is the creation and manipulation of *interaction patterns*. Certainly, the objective of computer science is at times similar to that of mathematics—for example, when proving theorems about formal languages and the automata that process them. However, the central activity of computer science is the production of software, and this activity is primarily characterized not by the creation and exploitation of inference structures but by the modeling of interaction patterns. The kind of interaction involved depends upon the level of abstraction used to describe programs. At a basic level, software prescribes the interacting of a certain part of computer memory, namely, the program itself, and another part of memory, called the program data, through explicit instructions carried out by a processor. At a different level, software embodies algorithms that prescribe interactions among subroutines, which are cooperating pieces of programs. At a still different level, every software system is an interaction of computational processes. Today's extremely complex software is possible only through abstraction levels that obscure machine-oriented concepts. Still, these levels are used to describe interaction patterns, whether they be between software objects or between a user and a system.

What is a "level of abstraction" in computer science? The history of software development tells a story of an increasing distance between programmers and the machine-oriented entities that provide the foundation of their work, such as machine instructions, machine-oriented processes, and machine-oriented data types. Language abstraction accounts for this distance by allowing programmers to describe computational processes through linguistic constructs that hide details about the machine entities by allowing underlying software to handle those details.

At the most basic physical level, a computer process is a series of changes in the state of a machine, where each state is described by the presence or absence of electrical charges in memory and processor elements. But programmers need not be directly concerned with machine states so described, because they can make use of languages that allow them to think in other terms. An assembly language programmer can ignore electrical charges and logic gates in favor of language involving *registers*, *memory locations*, and *subroutines*. A C language programmer can in turn ignore assembly language constructs in favor of language involving *variables*, *pointers*, *arrays*, *structures*, and *functions*. A Java language programmer can ignore some C language constructs by employing language involving *objects* and *methods*. The concepts introduced by each of these languages are not just old concepts with new names. They significantly enlarge the vocabulary of the programmer with new functionality while simultaneously freeing the programmer from having to attend to tedious details. For example, in the move from C to Java, programmers have new access to *active* objects, that is, data structures that are encapsulated with behavior so that they amount to a simulated network of software computers, while at the same time being released from the administrative work of managing the memory required to create these objects.

As levels of programming language abstraction increase, the languages become more expressive in the sense that programmers can manipulate direct analogs of objects that populate the world they are modeling, like shopping carts, chat rooms, and basketball teams. This expressiveness is only possible by hiding the complexity of the interaction patterns occurring at lower levels. It is no accident that these levels mirror the history of computer programming language development, for hiding low-level complexity can only be accomplished through a combination of more sophisticated language translators and runtime systems, along with faster processing hardware and more abundant storage.

We have shown that other forms of computer science abstraction besides language abstraction, for example procedural abstraction (Colburn 2003) and data abstraction (Colburn and Shute 2007), are also characterized by the hiding of details between levels of description, which is called *information hiding* in the parlance of computer science.

Floridi proposes to use levels of abstraction to moderate long-standing philosophical debates through a method that "clarifies assumptions, facilitates comparisons, enhances rigour and hence promotes the resolution of possible conceptual confusions" (Floridi 2008a, 326). These features are certainly advantages in a philosophical context, and there is no doubt that abstraction through information hiding in computer science goes a long way toward mitigating "conceptual confusions," but how similar are Floridi's and computer science's levels of abstraction provided by programming languages?

In Floridi's view, a level of abstraction is, strictly speaking, a collection (or "vector") of observables, or interpreted typed variables. What makes the collection an abstraction is what the variables' interpretations choose to ignore. For example, a level of abstraction (LoA) of interest to those purchasing wine may consist of the observables *maker*, *region*, *vintage*, *supplier*, *quantity*, and *price*, while a LoA for those tasting wine may consist of *nose*, *robe*, *color*, *acidity*, *fruit*, and *length* (2008a, 309). So a LoA by itself is, loosely, a point of view, one chosen simply to suit a particular purpose.

One can see how the observables available to descriptions of computational processes in different programming languages provide different views of the entities participating in computational models of the world. Some programmers see registers and subroutines, others see variables and functions, and still others see objects and methods. But while the choice of wine LoA between the two examples given above would be made solely on the basis of some underlying purpose, the choice of abstraction level for a software application involves considering both an underlying purpose and a need to make use of a higher LoA. While there are some exceptions in the world of specialized hardware programming, most software applications for general purpose machines today require enhanced expressiveness through LoAs that do not require the programmer to be concerned with the architecture of the machine. The level of abstraction used by one programming language is "higher" than another to the extent that it hides more machine architecture details and allows the natural representation of concepts in the world being modeled by a program.

Computer scientists often engage in comparisons of programming languages on the basis of their expressiveness or fitness for various purposes. If a programming language can be said to embody a single LoA, computer scientists would therefore be interested in the relationships of multiple languages through the arrangement of multiple LoAs, something Floridi considers in his concept of a "gradient" of abstractions (GoA). To develop the idea of a GoA, Floridi first introduces the notion of a "moderated" LoA. In general, not all possible combinations of values for variables in a LoA are possible—for example, wine cannot be both white and highly tannic. A predicate that constrains the acceptable values of a LoA's variables is called its "behavior." When you combine a LoA with a behavior, you have a moderated LoA (2008a, 310).

Before considering GoAs for programming languages, let us consider the concept of a programming language as a moderated LoA, that is, one with a "behavior." While there is an obvious sense in which some observables, such as program variables, have value constraints (a program variable, for example, cannot be both integral and boolean), other observables, such as functions, have no value constraints other than the

syntax rules that govern their construction. Yet functions nevertheless possess behavior.

There is a ready explanation for this disconnect. The model for Floridi's behavior concept is inspired by the information modeling activity of system specifiers, whose objective is the complete functional description of particular envisaged applications in terms of system state changes. Programming language developers, however, are in the quite different business of providing the tools that facilitate the implementation of the products of the specifiers, and programmers themselves use those tools to create computational objects of varying degrees of expressivity. It seems possible in principle to extend Floridi's concept of behavior to cover function behavior as well as the behavior of typed variables.

Returning now to Floridi's idea of a gradient of abstractions, we find that a GoA is constructed from a set of moderated LoAs. The idea, according to Floridi, is that "[w]hilst a LoA formalises the scope or granularity of a single model, a GoA provides a way of varying the LoA in order to make observations at differing levels of abstraction" (2008a, 311). Why would one want to do this? By way of explanation, Floridi again considers the wine domain. Since tasting wine and purchasing wine use different LoAs, someone who is interested in both tasting and purchasing could combine these LoAs into one GoA that relates the observables in each. So while a LoA is characterized by the predicates defined on its observables, a GoA requires explicit relations between each pair of LoAs. Floridi describes these relations formally using standard Cartesian product notation and conditions that ensure that the related LoAs have behaviors that are consistent. As Floridi points out, these consistency conditions are rather weak and do not define any particularly interesting relations between the LoAs. However, by adding certain conditions he defines a "disjoint" GoA, or one whose pairwise LoAs have no observables in common, and a "nested" GoA, or one whose LoAs can be linearly arranged such that the only relations are between adjacent pairs (2008a, 312).

Nested GoAs are useful because they can "describe a complex system exactly at each level of abstraction and incrementally more accurately" (2008a, 313), as in neuroscientific studies that begin by focusing on brain area functions generally and then move to consideration of individual neurons. It is interesting to note that while the gradient in a nested GoA goes from more abstract to more concrete, the gradient at work in computer science programming language abstraction proceeds along an orthogonal dimension, namely, from the more machine-oriented to the more world-oriented.

Put another way, nested GoAs (though not GoAs in general) are often constructed for the sake of more fine-tuned *analysis*, while the new abstraction levels offered by computer science programming languages present greater opportunities for *synthesis* in the construction of

programming objects of ever-larger content than those available before. The content of a programmer's computational objects *enlarge* as more of the underlying machine details of representing them are hidden from the programmer by the chosen implementation language. This is particularly evident when considering the data abstraction capabilities of higher-level languages.

Lower-level languages such as assembly languages offer basic data types like numbers, characters, and strings. These are types that are themselves abstractions for the hard-wired circuitry of a general-purpose computer. If a programmer wants to write, say, a Web browser using assembly language, he must laboriously implement a high-level notion such as a communication socket in the language of the machine—numbers, characters, strings, and so on. As all assembly programmers know, this is an impoverished vocabulary requiring painstaking and time-consuming coding that is unrelated to the central problems of Web browsing. Hundreds or perhaps thousands of lines of code might be required just to perform what amounts to machine-level data book-keeping. By using a higher-level language such as C, the programmer can take advantage of richer types such as structures (that can group the basic machine types automatically) and pointers (that can take on the burden of memory address calculations), relieving a good deal of the drudgery of manipulating these basic machine types. But by using higher-level languages still, such as C++ or Java, programmers can expand their coding language to include types whose values are communication sockets themselves. No "shoehorning" of the higher-level notion of a communication socket using lower-level programming types is necessary; a communication socket actually *is* a data type in these higher-level languages.

It may very well be possible to define a gradient of abstractions for programming languages and data types that would fit Floridi's model, so long as the relations between the LoAs making up the gradient accurately capture the nature of information hiding on which software development so crucially depends. In fact, the concept of a GoA may be useful for characterizing the relations among the various "paradigms" of programming languages. For example, the fundamentally imperative and machine-oriented nature of assembly language programming and the purely functional nature of Lisp or Scheme may place them in a disjoint GoA, while the historical evolution of the C language into C++ suggests a nested GoA for them. Floridi points out (2008a, 314) that hierarchical GoAs that are less restricted than nested GoAs can also be defined, arranging their LoAs in tree structures. The many programming paradigms that have evolved in computer science would likely fit on such a GoA.

Whatever the programming paradigm, by hiding the complexity of dealing with lower-level data types, the information hiding provided by

high-level languages gives programmers and scientists the expressiveness and power to create higher-level objects that populate ever more complex worlds of their own making. Such worlds exhibit Floridi's informational structural realism. Structural realism (SR) gives primacy to the *relations* that hold among objects being studied, for it is they, rather than the *relata* themselves, that facilitate explanation, instrumentation, and prediction within the system under investigation (Floridi 2008b, 220). What makes Floridi's SR informational is that questions about the ontological status of the *relata* can be put aside in favor of a minimal, informational conception of the objects bearing the relations. To explicate this idea, Floridi borrows the technical concept of an "object" from computer science's discipline of *object-oriented programming* (OOP). Because such objects encapsulate both state (Floridi's observables in a LoA) and behavior (how they relate or interact with other objects), they constitute paradigm candidates for the structural objects embraced by SR. But because they are abstractions, they avoid any ontological calls for substance or material in the characterization of structural objects.

The ontologically noncommittal nature of Floridi's informational structural realism (ISR) comes through when he uses Van Fraassen's categorization of types of structuralism and puts ISR in the category of "in-between" structuralism: it is neither radical (structure is all there is) nor moderate (there are nonstructural features that science does not describe). Instead, "the structure described by science does have a bearer, but that bearer has no other features at all" (Floridi 2008b, 221f.). However, as any object-oriented programmer knows, information objects are richly featured, even apart from the important relations they bear to other objects. A file folder, for example, has a size and a location besides the relations to the parent folder that contains it and the subfolders it contains. True, the objects of OOP are structured, but they are not mere relations; they are rich with *attributes*—OOP parlance for nonrelational properties. A programmer could, of course, replace an object *O*'s attributes with other objects, so that *O* consists only of relations with other objects, but eventually those other objects must have attributes that are nonrelational. So not all objects can be featureless, which Floridi seems to desire.

At the same time, one of the powerful features of OOP that distinguishes it sharply from the programming paradigms preceding it is that program objects can *be* relations themselves, not just participate in them. So a programmer modeling a ticket agency can describe *spectator* and *show* objects, but it is possible (and in most cases preferable) to model the relationship that is a spectator's attending a show as itself an object as well.

Such is the generic nature of the "object" in OOP, whether an object is a relation or a relatum is entirely dependent on context. The object-oriented programmer has not only expressive but also ontological

freedom in crafting his objects, and with that freedom comes the need for constraints on object behavior in the form of laws.

The Concept of Law in Computer Science

Modern epistemology is concerned in part with how we use our sense experience to acquire immediate knowledge of individual objects, processes, and events in the physical world through the interaction of our own bodies with it. But as John Hospers remarks, "If our knowledge ended there, we would have no means of dealing effectively with the world. The kind of knowledge we acquire through the sciences begins only when we notice *regularities* in the course of events" (1967, 229). When a statement of such a regularity admits of no exceptions, as in *Water at the pressure found at sea level boils at 212 degrees Fahrenheit*, it is called a "law of nature" (230).

There is a rich twentieth-century analytic philosophy tradition of trying to characterize exactly what scientific "lawlikeness" is, but there is general agreement that, at the very least, a statement is lawlike if it is general (i.e., universally quantified), admits of no spatial or temporal restrictions, and is nonaccidental (i.e., is in some sense necessary) (see, e.g., Danto and Morgenbesser 1960, 177). When knowledge has these properties, it can be used to both explain and predict particular observed phenomena and occurrences. Being able to predict what will happen in a given situation allows us to control future events, with the attendant power that accompanies such control.

Computer science, as we mentioned, does not study nature, at least the nature studied by physics, chemistry, or biology. It studies, of course, *information* objects (in a general sense, not necessarily in the sense of OOP), and most software developers would view their reality in Floridi's way as "mind-independent and constituted by structural objects that are neither substantial nor material ... but informational" (Floridi 2008b, 241). But if computer science is a science, albeit a science concerned with information objects, and science attempts to discover empirical laws, with what laws, if any, are computer scientists concerned?

Some pronouncements, for example, the celebrated *Moore's Law*, are empirically based predictions about the future of technology. Moore's Law makes the observation that the number of integrated circuit components (such as transistors) that can fit on a silicon wafer will double every two years. While this observation, a version of which was first made in 1965, has proved uncannily correct, it does not fit the criteria for a scientific law, since even those who uphold it do not believe that it is temporally unrestricted; the technology of integrated circuit creation has physical limits, and when these limits are reached Moore's "Law" will become false.

Other purported "laws" having to do with the future of technology have already proven false. *Grosch's Law*, also coined in 1965, stated that the cost of computing varies only with the square root of the increase in speed, and so it supported the development of large supercomputers. But the opposite has emerged: economies of scale in most cases are achieved by clustering large numbers of ordinary processors and disk drives.

If Moore's Law and Grosch's Law are technology reports, other statements seem to embody, if not laws of nature, then *laws of computation*. For example, Alan Turing demonstrated that no procedure can be written that can determine whether any given procedure will halt or execute indefinitely. Such a statement satisfies the lawlikeness criteria given above, namely, generality, spatiotemporal independence, and being nonaccidental, but it is a statement of a mathematical theorem, not an empirical law supported by observation.

Other pronouncements that come to be known as "laws" are also really mathematical relationships applied to problems in computer science. *Amdahl's Law*, for example, gives an upper bound on the speedup one can expect when attempting to parallelize a serial algorithm by running it concurrently on a fixed number of processors. Similarly, *Gustafson's Law* modifies Amdahl's Law by removing the constraint that the number of processors be fixed. Each of these laws is deduced a priori from abstract concepts, not a posteriori from observations.

There is another important set of mathematical laws with which computer scientists are fundamentally concerned, dealing with the bounds on space and time that are imposed on computational processes by their inherent complexity. Just as Amdahl's and Gustafson's laws use formal mathematical arguments to give limits on the speedup obtained through the use of multiple processors, computer scientists use mathematical methods to classify individual algorithms, for example algorithms to search various kinds of data structures, in terms of the memory required to store the data structures and the number of operations required to search them. It is important to note that these analyses are based not on actual physical runnings of the algorithms through programs running on physical machines but on the analysis of algorithms as abstract mathematical objects. No empirical study or investigation is involved; in fact, a typical objective of such analysis is to determine whether a given computation can be accomplished within reasonable space and time bounds regardless of technology prognostications regarding the speed of future machines. Such statements about abstract computational objects may seem to satisfy the above criteria of lawlikeness, but these statements are supported by formal proofs and not empirical investigation.

So is there a sense in which computer science can be said to "discover" scientific laws in the empirical sense? We think computer scientists don't

discover laws; they must make them. When a programmer specifies an abstract procedure for sorting an abstract list of objects using the operations of an abstract machine, it is because that procedure will be implemented, through a complex and remarkable chain of electronic events, on a given machine in the physical world. Those events must be able to be accurately predicted in order for a program to serve the purpose for which it was intended. For this to happen, these electronic events must obey laws, but to *describe* these laws at the physical level of electrons and microcircuits would serve no purpose, because the physical laws of electromagnetism at the molecular level are irrelevant to a programmer's objective, which is to describe, in the formal language of a machine, the interaction patterns of abstract objects like variables, procedures, and data structures. Instead, the electronic events unfold as they do, and (one hopes) as they should, because the programmer *prescribes* laws in the realm of the abstract. Programmers must make their abstract worlds behave as though there were laws, so that the physical processes they produce benefit from the explanatory and predictive power that accompanies laws. To this end, computer science relies heavily, if not essentially, on *metaphor*.

In Colburn and Shute 2008 we describe how metaphors can help computer scientists treat the objects in their abstract worlds scientifically. For example, although there is no natural "law of conservation of information," network programmers make things work as though there were one, designing error detection and correction algorithms to ensure that bits are not lost during transmission. Their conservation "law" relies upon a *flow* metaphor; although bits of information do not "flow" in the way that continuous fluids do, it helps immeasurably to "pretend" as though they do, because it allows network scientists to formulate precise mathematical conditions on information throughput and to design programs and devices that exploit them.

The flow metaphor is pervasive and finds its way into systems programming, as programmers find and plug "memory leaks" and fastidiously "flush" data buffers. But the flow metaphor is itself a special case of a more general metaphor of *motion* that is even more pervasive in computer science. Talk of motion in computer science is largely metaphorical, since when you look inside a running computer the only things moving are the cooling fan and disk drives (which are probably on the verge of becoming quaint anachronisms). Yet descriptions of the abstract worlds of computer scientists are replete with references to motion, from program jumps and exits, to exception throws and catches, to memory stores and retrievals, to control loops and branches. This is to be expected, of course, since the subject matter of computer science is *interaction* patterns.

But the "motion" occurring in the computer scientist's abstract world would be chaotic if not somehow constrained, and so we place limits that

are familiar to us from the physical world. In the case of network programming, we borrow from physics to come up with a law of conservation of information. For programming in general we borrow from physical laws of motion to come up with laws of computation in the form of *invariants*.

Computer Science Laws as Invariants

Just as natural laws admit of no exceptions, when programmers prescribe laws for their abstract worlds they must make sure they admit of no variation. The mandate of a descriptive law of nature is to discover and describe what *is*. The mandate of a prescriptive law of computation is to legislate what *must hold*, invariably, while a computation takes place. Here are some examples of loosely stated prescriptive laws:

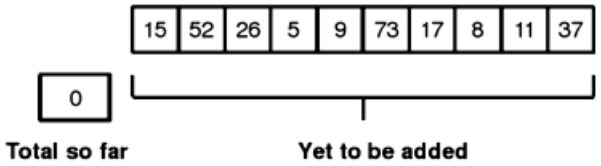
- *The parity bit must equal the exclusive-or of the remaining bits* (used in error checking code)
- *The array index must not be outside the range [0..n–1]* where *n* is the size of the array (to avoid out-of-bounds references)
- *The segments of an array must at all times stand in some given relationships to one another* (to sum or sort the array, as in the examples below)

These prescriptions state *constraints* that must hold in their respective computational worlds. When a constraint is maintained throughout a computational interaction, for example, during the calling of a procedure or an iteration of a loop, it becomes an *invariant*. Invariants do double duty. First, they function as pedagogical devices for explaining algorithms (see, e.g., Gries 1981), even for people with little prior programming experience. Second, they have the generality of laws in that they are specified for algorithms, which are abstractions, and as such embody computational laws governing *any* concrete processes that implement them.

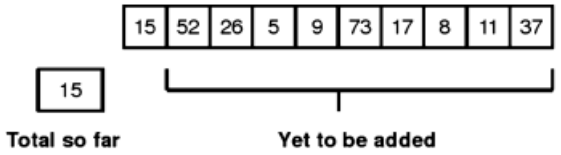
It is important to distinguish between the role of an invariant in an abstract computational process and that of an *axiom* in a formal model. Invariants are things we want to *make* true, while axioms are *assumed* to be true unless the model shows they are inconsistent. But while we endeavor to make invariants true, in keeping with computer science as the master of its domain they are often broken. There are two basic reasons for this. First, invariants often operate at levels of abstraction that cannot be enforced at the programming level; that is, programming requires steps that temporarily violate invariants, with later steps restoring them. Second, an invariant might be violated to adapt a computational

structure or algorithm for a different purpose. In what follows we describe both of these scenarios.

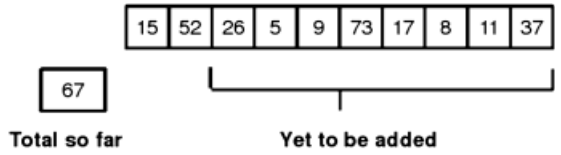
Complex computational processes can be brought about by enforcing complex invariants, which themselves can be created by accommodating simpler ones. For example, an invariant for summing the entries of an array (a contiguous list of data) can be formulated as a picture showing the sum of the entries considered so far and the part of the array whose entries have not yet been added to the sum:



Here the partial sum has been initialized to zero and none of the entries has been added to it. The partial sum plus the entries not yet considered add up to the total of all entries in the array, which is 253. This is an invariant, because it remains true as the algorithm proceeds. After one entry is considered the picture looks like this:



After two entries:

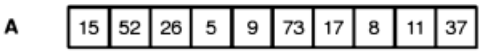


At each step the invariant holds. Eventually the partial sum is 253, and there are no more entries to consider. The invariant embodies a law that a computational process must follow in order to correctly total the entries of the array: it must manage the partial sum while reducing the unexamined array segment until it is empty.

When programming students are directed to implement this algorithm in a programming language, they learn that in order to keep track of the

"Yet to be added" part of the array, they need to manage an index, call it *i*, that points to the beginning of that part. After having updated the partial sum with the first element of that part, the invariant is violated because the unexamined part has not been shrunk. It is up to the programmer to restore the invariant by incrementing *i*, which has the effect of shrinking the unexamined array segment. Describing invariants and how to maintain them through individual program steps is an essential part of the computer science education process. We shall see later that violating invariants (however temporarily) occurs not only during the programming process but also in the development of algorithms themselves.

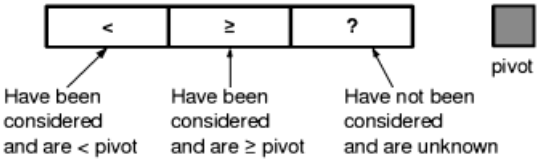
The power of invariants can also be seen in a famous sorting algorithm developed by Hoare. To understand *Quicksort*, a sorting procedure celebrated for its efficiency, consider the same array as before:



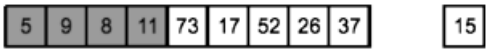
Hoare realized that if **A** could be partitioned into three segments, with all numbers in the left-most segment less than the single number (called the *pivot*) in the middle segment, and all numbers in the right-most segment greater than or equal to the pivot, as in this arrangement (15 is the pivot):

then the pivot element would be in its correct place in the final sorted array. Hoare further realized that if the left-most and right-most segments of **AP** were themselves partitioned in the same way, then *their* pivot elements would also be in their correct places in the final sorted array. If this process of partitioning segments is carried out recursively until each remaining segment has only zero or one element, the array will be completely sorted, and it will have been done efficiently provided that the partitioning was done efficiently.

The key to efficient partitioning of an array segment is to maintain three subsegments, which we will call $<$, $\geq$, and **?**, that maintain the following invariant:



When the partitioning algorithm begins on **A**, the first element (15) is chosen as the pivot, all the non-pivot elements are unknown and part of the **?** segment, and the $<$ and $\geq$ segments are empty. As the algorithm proceeds, the size of the unknown segment steadily decreases while $<$ and $\geq$ grow, until finally the picture looks like:



Here the $<$ segment is shaded, the $\geq$ segment is unshaded, and the unknown segment **?** is empty. To finish the partitioning, the first element of the $\geq$ segment, 73, is replaced by the pivot and moved to the end of the array to produce the partitioned array **AP** shown above. The partitioning algorithm can now be recursively applied to the left and right subsegments to eventually sort the array.

It would have been difficult for anyone, even Hoare, to conceive of this algorithm without the help of a guiding invariant. Programmers start with invariants as prescriptive laws and then try to create abstract worlds that obey them. When an abstract process maintains a carefully prescribed invariant, its concrete realization will behave as though governed by a descriptive computational law. That is, its behavior will be predictable, controllable, and correct for its purpose. Thus invariants are to program objects what laws of nature are to natural objects. Just as a planet circling a sun cannot help but follow Kepler's laws of planetary motion and be predictable in its course, a program written to obey an invariant cannot help but behave in a predictable way.

The Interplay of Freedom and Constraint

Francis Bacon wrote, "Nature, to be commanded, must be obeyed" (Bacon 1889). This applies in computer science, but with a twist—program developers prescribe laws for programs, and then must ensure the programs obey these laws. The laws then become constraints on the programmer. But these are not constraints in a political sense. The danger of political lawlessness is external—your unconstrained freedom is a threat to my freedom and vice versa, but the danger of lawlessness in computer science is internal—our minds need constraints in order to reason through the consequences of programming decisions. For example, it is the programmer's imperative to reestablish the invariant for each iteration of a loop.

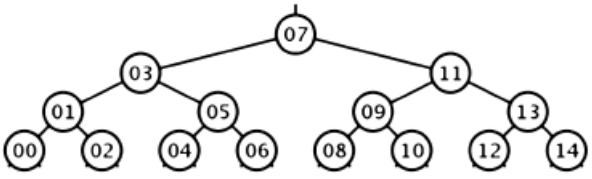
While constraints in general limit freedom, in the case of programming they make it much easier to achieve an objective, through the breaking down of a problem solution into a sequence of small steps, each governed by an invariant.

What seems to limit freedom actually opens up new pathways. Imagine a person exploring a jungle without a compass. She has freedom to move anywhere she wishes, but she has no guidance. As she makes her way, she has to frequently avoid entanglements, causing her to lose her direction. After a while, her path through the jungle is essentially a free but random walk. She has freedom of movement but lacks knowledge, so she cannot make meaningful progress.

Now give her a compass that constrains her movement but provides knowledge in the form of directional feedback. Even with entanglements she can proceed in a relatively straight line by correcting her direction after each deviation. By limiting her freedom in the short term (adding a compass correction after each entanglement), she increases her long-term freedom—her ability either to explore deeper into the jungle or to emerge from it.

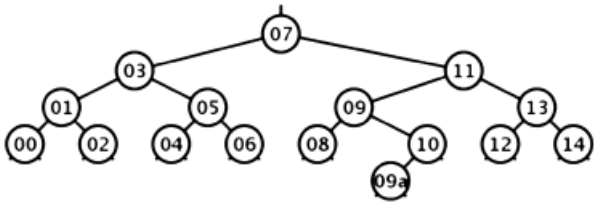
The compass constrains the action of the jungle explorer just as program invariants constrain the action of computational processes. But in actual practice, programmers may or may not "think up" invariants before writing their programs. Beginning programmers often have a loose and vague idea of what their programs should do and start programming without constructing invariants. Just as a jungle explorer may get lucky and emerge without a compass, a programmer who disregards invariants may get lucky and produce a program whose behavior seems correct, but he cannot be sure that it is correct in all instances. Through invariants, however, programmers can be confident in their problem solutions, because the interactions they produce are governed by law, albeit prescriptive law.

While we have shown that programmers learn to manage arrays by temporarily violating and then restoring invariants, this approach can also be fruitful in algorithm development, a nonprogramming endeavor. For example, consider a data structure known as a *binary search tree* (BST). BSTs facilitate looking up data using a key. Their functionality is similar to telephone directories, where the keys are people's names and the data are addresses and telephone numbers. Here is a BST whose keys are simple strings (for simplicity, the data are not shown):



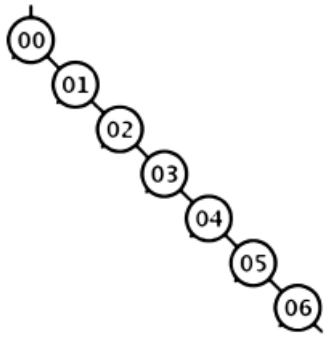
In order to preserve order among a BST's data items, a programmer must maintain the following invariant: for any node in the BST, all keys in its left subtree must be smaller than its key, and all keys in its right

subtree must be greater than its key. When a new node is added to a BST, its key is compared with the key of the tree's root (the topmost node). If it is smaller, the new node will be placed in the left subtree, otherwise in the right. The appropriate subtree is recursively searched until an available space is found on one of the tree's leaves (bottommost nodes). Here is the example tree after the node with key **09a** is added:



This arrangement facilitates data retrieval by key, since a key can be located in time proportional to the height of the tree. If a tree is balanced, as in the one above, a key can be located efficiently even if the number of nodes is large. For example, a balanced tree of one million nodes has a height of about 20.

Unfortunately, the structure of a BST is determined by the order in which nodes are added to it, so nothing guarantees that a BST will be balanced. Here is a BST in which nodes with keys **00**, **01**, **02**, **03**, **04**, **05**, and **06** have been added in that order:



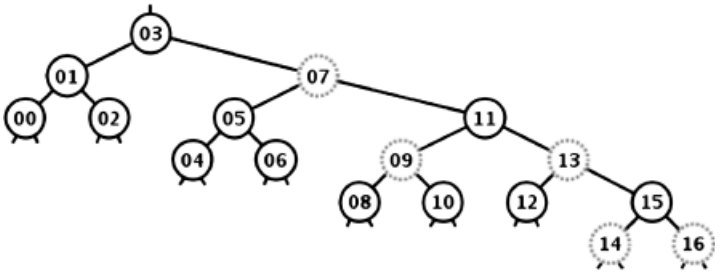
Although this structure satisfies the order invariant for BSTs, it cannot be efficiently searched, since it is not balanced. If one million nodes are added to a BST in key order, finding nodes with higher numbered keys will take time proportional to its height, which is one million (compared to 20 in a balanced BST of a million nodes).

To solve this problem, computer scientists have devised a kind of self-balancing BST known as a *red-black tree* (RBT). In addition to the ordering invariant imposed on BSTs, RBTs introduce the concept of a

node's *color*, requiring every node to be either red or black, with the following additional constraints:

1. All downward paths from the top (root) of the tree to the bottom (leaves) must contain the same number of black nodes.
2. The parent of a red node, if it exists, is black.

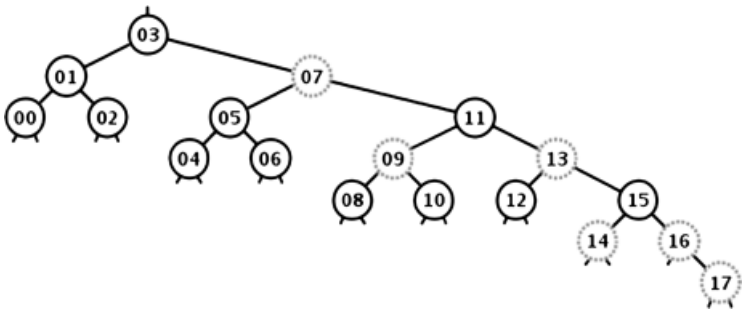
Here is a BST that is also a RBT (red nodes are shown here as dotted circles):



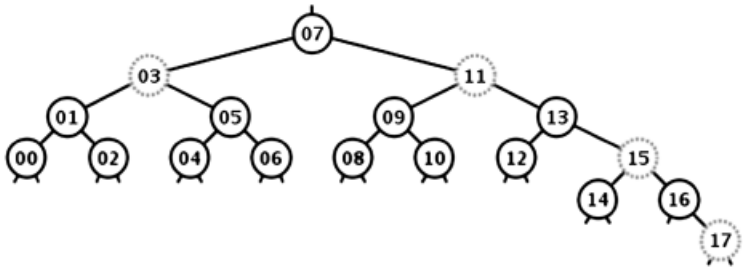
The RBT constraints do not make a search tree perfectly balanced, but they do ensure that no search path is longer than twice the length of search paths in a perfectly balanced tree. Thus in a tree with one million nodes, search paths will be no longer than 40.

This arrangement works without a hitch for some cases. Consider adding a node with key **09a**. Using the standard BST adding algorithm, it can be added as the left subtree of node **10**. Then it can satisfy the RBT constraints by being colored red. Now consider adding a node with key **17**. The BST adding algorithm would put it as the right subtree of node **16**. However, coloring it black would violate RBT constraint 1, while coloring it red would violate RBT constraint 2.

But because computer science is the master of its domain, we can choose to violate our own laws through the temporary relaxing of constraints. While the suspension of a natural law would be called a miracle, the temporary violation of our RBT constraints, with a little ingenuity, can result in an efficient data structure. So we go ahead and violate constraint 2 by adding **17** and coloring it red (shown here as a dotted circle):



Without going into detail, suffice it to say that by tweaking this law-breaking RBT in various ways (through structural changes known as rotations and certain recolorings) it can be nudged back into being a law-abiding citizen of the computational world with both RBT constraints satisfied:



Conclusion

The ultimate nature of computational reality is, of course, informational. That is what allows computer science to create its own empirical laws, and to be free to break them when it sees fit. We have shown that the worlds of computational objects need laws in the form of self-prescribed invariants, but also that the suspension of these laws might be creative acts, resulting in neither anarchy nor miracles.

Whether the ultimate nature of *all* reality is informational is not as obvious. But if computer science has sage advice for philosophy, Floridi has seized upon the right concepts in levels of abstraction and the methodology of object-oriented programming, for they are the catalysts of progress in computer science and software engineering. While OOP may one day be replaced by another latest and greatest programming paradigm, the march toward ever-higher levels of programming abstrac-

tions will continue. Whether this march coincides with a move toward informational ontology is yet to be seen.

Department of Computer Science
University of Minnesota
320 Heller Hall, UMD
Duluth, MN 55812
USA
tcolburn@d.umn.edu
gshute@d.umn.edu

References

Bacon, Francis. 1889. *Novum Organum*. Edited by Thomas Fowler. Oxford: Clarendon Press.

Colburn, Timothy. 2003. "Methodology of Computer Science." In *The Blackwell Guide to the Philosophy of Computing and Information*, edited by Luciano Floridi, 318–26. Oxford: Blackwell.

Colburn, Timothy, and Gary Shute. 2007. "Abstraction in Computer Science." *Minds and Machines* 17:169–84.

———. 2008. "Metaphor in Computer Science." *Journal of Applied Logic* 6:526–33.

Danto, Arthur, and Sidney Morgenbesser. 1960. "Introduction to Part 2, Law and Theories." In *Philosophy of Science*, edited by Arthur Danto and Sidney Morgenbesser, 177–81. Cleveland, Ohio: Meridian Books.

Floridi, Luciano. 2008a. "The Method of Levels of Abstraction." *Minds and Machines* 18:303–29.

———. 2008b. "A Defence of Informational Structural Realism." *Synthese* 161:219–53.

Gries, David. 1981. *The Science of Programming*. New York: Springer.

Hoare, C. A. R. 1962. "Quicksort." *Computer Journal* 5:10–15.

Hospers, John. 1967. *An Introduction to Philosophical Analysis*. 2nd ed. Englewood Cliffs, N.J.: Prentice Hall.