Ubiquity, an ACM publication
November 2010

Ubiquity Symposium

What is Computation?

Computation is Symbol Manipulation

by John S. Conery, University of Oregon

Editor's Introduction

In the second in the series of articles in the Ubiquity Symposium 'What is Computation?', Prof. John S. Conery of the University of Oregon explains why he believes computation can be seen as symbol manipulation.

For more articles in this series, see table of contents in the Editor's Introduction to the symposium.

Ubiquity, an ACM publication
November 2010

Ubiquity Symposium

What is Computation?

Computation is Symbol Manipulation

by John S. Conery, University of Oregon

For the last five years I have been teaching an introduction to computer science for non-majors. As might be expected, one of the first topics is "what is computation?". After reading Prof. Denning's article introducing this symposium on computation (Denning, 2010), and considering earlier articles by both symposium editors, I decided the definition in the textbook I wrote for the course best captures my view of the essence of computation:

> *A computation is a sequence of simple, well-defined steps that lead to the solution of a problem. The problem itself must be defined exactly and unambiguously, and each step in the computation that solves the problem must be described in very specific terms.* (Conery, 2010)

The keys to this definition, of course, are "problem" and "step." A more formal definition is that a problem, and its solution, must be encoded in the form of *symbols*; a step is a *symbol manipulation* that transforms one set of symbols into a new set of symbols. In the rest of this paper, I will use the term *state* to refer to a collection of symbols, and *transition* to refer to a step that maps an input state to an output state.

According to this definition, a computation is a discrete process, a sequence of distinct transitions. A classic example is the construction of a list of prime numbers using the Sieve of Eratosthenes. One starts by writing a list of integers, starting with two and ending with some upper limit. The list is the initial state of the computation. The first step removes all multiples of two—2, 4, 6, 8, etc. The next steps remove the multiples of three, then multiples of five, and so on, until only prime numbers remain. There are many ways to represent the integers and the list structure in symbolic form. If the "technology" being used is paper and pencil, numbers can be written as strings of digits, and lists can be written as sequences of numbers separated by commas. In a computer, the numbers are encoded in binary, and one can use any of several different schemes for representing lists as data structures.

This definition raises the question of whether a computation necessarily involves multiple steps. It may appear to be "cheating" to allow a computation to go from the initial state to the final state in one step, but in fact that's what happens in functions that use *memoization*, (a term that means the result of a computation is saved in a table so it can be reused at some point in the future), where results of a computation are stored in a table so they can be looked up the next time the function is called. After using the sieve algorithm to compute the list of primes less than some number *n*, the list is saved in row *n* of a table. The next time we want the a list of prime less than *n* the function looks in row *n* of the table to find the output. This single step is a computation, just as much as the original sifting operation was a computation; this second computation simply maps an integer *n* to a list of primes less than *n* in a single table look-up step.

A second question is about the size of a step. If we look at how the sieve is implemented in a typical program, we would see that what I described above as a single step—deleting multiples of a given number—is actually a series of smaller steps. A typical program will create a list of integers, and iterate over the list to remove composite numbers. Each step in the iteration removes one element from the list, and the intermediate states can certainly be considered states of the computation.

As computer scientists we recognize the grouping of several individual sub-steps into a larger step as an example of *abstraction*. Given a computation, it is almost always possible to "look inside" a single state transition, and to describe it as the result of a more detailed sub-computation that is itself a sequence of state transitions. If we continue looking more closely at the steps in the Sieve of Eratosthenes algorithm, the decision of whether a number is composite is a sequence of steps involving an application of the mod operator followed by a test for the result equaling zero. Moving down to even lower levels of detail when the algorithm is being run on a computer, we can look at the sequence of steps at the machine level, where the bit patterns in the CPU are passed to the functional units that carry out integer division and comparison with zero. We will eventually reach a point where it becomes too difficult, or even impossible, to describe the operations in terms of symbol manipulations, and simply conclude that continuous operations—in the case of a computer, electronic signals moving through a semiconductor VLSI chip—perform the single state transition that carries out a primitive computation. In practice, we only go to enough detail to convince people that a step can be mechanized and performed by a machine.

To recap, a computation is a sequence of state transitions, where a state is defined by a set of symbols. It does not matter if the transition is the result of several, more detailed, symbol manipulations, or a single transition performed by an analog device or some other system best described as continuous. The size of a step also does not matter. A step might change only one symbol, or it could replace all or part of the input state with a completely new set of symbols. It could be as small as a bit-level operation done inside an electronic computer, or a more abstract rewriting operation involving letters, digits and other symbols.

One aspect of computation that was not addressed above is the role of the "agent" carrying out the computation. Clearly there must be some structure to the computation, otherwise one could claim any collection of random symbols constituted a state, and any two unrelated states could form a computation. By calling a sequence of states a "computation" one is implying that there is some reason to correlate the states, and there is some process at work to map all or part of the symbols of one state into symbols in the next state.

In computer science the state transitions are defined by algorithms. In considering the relationship between algorithms and computations, the classic view is that an algorithm is a static description of what will eventually become a computation, and computations are dynamic sequences of observed states defined by an algorithm (*e.g.*, Dijkstra, 1976). The computations that result from the execution of an algorithm are what give us the sense that computations are sequences of states, rather than a single "leap of faith" from an input state to an output state. Algorithms allow us to break a complex problem into smaller parts, and the working of an algorithm gives us confidence that the computation is progressing inexorably toward a final solution. The correctness of the algorithm is an argument or proof that it always leads to the proper final solution.

To return to the example of a computation that produces a list of prime numbers, the steps in the Sieve of Eratosthenes algorithm are simple enough that we can prove the final result is correct. The algorithm leads to a systematic construction of a list of prime numbers, and we are convinced no prime number less than *n* has been left out of the list, and that every number in the final list is prime. The memoized version that looks up the answer in a table is convincing, but only because we have confidence in the computation that produced the table entry in the first place.

One of the questions raised by Denning (2010) is whether a continuous physical process could be considered a computation. According to the definition of computation presented above, there is no reason why a continuous device could not be the agent behind a computation. Any process that maps an input state to an output state, where each state is a collection of symbols, can be considered a computation. In fact, we can go one step further, and claim that *every* computation carried out by a physical system, whether it is an electronic computer, a human brain, or any other device, is ultimately the result of continuous operations. As we look lower in the levels of abstraction, and decompose the individual steps of a computation into simpler and simpler sub-steps, we eventually reach a point where the most primitive operations are best described as the result of continuous processes. We rarely go this far, however, because at some point the steps are simple enough that we trust they are correct steps specified by an algorithm. We think of a computer as a discrete device because we are only concerned that bit patterns change in a well-understood manner, and (unless we're computer engineers designing the circuits) we don't bother trying to describe the inner working of the continuous device that is ultimately responsible for the computation. The important point is that the computation is defined by the sequence of symbolic states, not the nature of the agent that carries out the computation.

If we want to use the metaphor of computation to describe natural processes, such as DNA translation, we can allow for other agents besides computers or humans who are following the steps of an algorithm to be the sources of the states in a computation. To put it another way, while it may be the case that the execution of an algorithm defines a computation, the converse is not necessarily true: not all computations are defined by an algorithm. As a computation, DNA translation is a mapping from strings of letters representing nucleotides into strings of letters representing amino acids. In this case, a biochemical process is the agent responsible for the state transitions.

It's interesting to note that this computation also has levels of abstraction. The high level mapping from gene sequence to protein sequence has intermediate steps that can themselves be described as symbol manipulations. A sequence of DNA is transcribed (copied) into a messenger RNA sequence, and then a series of operations transform the mRNA by adding a "cap" and a "tail" and splicing out unused portions of the gene (see for example Hunter, 1985). At the lowest level, a continuous physical process is responsible for the discrete states we use to describe the complete process of gene expression.

Interactive computations (Goldin, *et al.*, 2006), which involve human agents, are another example where state transitions are not always the result of an algorithm. In these systems, actions by human users are encoded symbolically (for example, the *x* and *y* coordinates of a pointer when it is clicked, the names of a character and modifiers when a key on a keyboard is pressed, digitized voice commands) and injected into the computation, where they can be acted upon by algorithms.

A question raised in the literature on interactive computing is whether computations can be non-terminating. In the classic view, where computations are defined by algorithms, computations are always finite, because algorithms must terminate in order to ensure their outputs are well-defined. Common examples used to explain the need for an interactive model that would include non-terminating computations are applications with graphical user interfaces, operating systems, and networks of web-based services.

I prefer to think of interactive computations as *indefinite* instead of infinite or non-terminating. Indefinite computations are also controlled by algorithms, and these algorithms do include conditions for terminating. Interactive applications have "quit" commands that close files and release other resources. Operating systems have commands that shut down the system, telling it to flush buffers, signal applications and daemons to exit, etc. A web server includes code that will cause it to exit gracefully, finishing up any pending requests before it can be restarted with a new configuration. What keeps these computations going indefinitely is the fact that they are set up to process inputs until a halt or exit command is seen. In a hybrid system, where a state includes symbols generated by algorithms, humans, remote systems, and other sources, the computation continues as long as there are requests. All that is required of the algorithms in such a system is that they map, in finite operations, a portion of one state of the computation to a portion of a successor state.

Another example of an indefinite computation is an implementation of the Sieve of Eratosthenes in a lazy functional language such as Haskell (see Hudak, *et al.*, 2007)[1]. Interestingly, the sieve function in such a language typically does not have a terminating condition. At first glance, the code looks like the definition of a nonterminating process, and if the code was to be evaluated by a strict (non-lazy) interpreter it would be caught in an infinite loop before it generated any output. But when the code is evaluated by the Haskell interpreter, it generates output on demand. The interpreter will set up process to carry out the computation of the list of primes, then suspend until a user or some other part of computation asks for an element from the list. When a request is made, the algorithm activates, computes the requested item, and then suspends again, just as an operating system or other interactive computation suspends until it receives another request.

To summarize, a computation is a discrete process, a sequence of states that are defined by symbols. The transition from one state to another is the result of some process or collection of processes, where a process could be an algorithm being executed on a single computer, a human interacting with an application running on a computer, another computer at a remote site on the Internet, or physical or biological systems that have states that can be represented symbolically.

References

Conery, J. S. *Explorations in Computing*. CRC Press, 2010.

Denning, P. J. *Ubiquity* Symposium 'What is computation?': Editor's Introduction, *Ubiquity*, 2010.

[1] Several different techniques for generating prime numbers, including a lazy implementation of the Sieve of Eratosthenes, are described at http://www.haskell.org/haskellwiki/Prime_numbers.

Dijkstra, E. W. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

Goldin, D. Q., Smolka, S. A., and Wegner, P. *Interactive Computation: The New Paradigm*. Springer, September 2006.

Hudak, P., Hughes, J., Jones, S. P., and Wadler, P. A history of Haskell: Being lazy with class. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), ACM.

Hunter, L. Molecular biology for computer scientists. In *Artificial Intelligence and Molecular Biology*. MIT Press, 1985, ch. 1, pp. 1–45 (available online at http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/ArtificialIntelligenceAndMolecularBiology).

About the Author

John S. Conery is a Professor of Computer and Information Science and a member of the Center for Ecology and Evolutionary Biology at the University of Oregon.

DOI: 10.1145/1880066.1889839