DOI:10.1145/2209249.2209251

An Integral Number and Its Consequences

Pondering Moshe Y. Vardi's Editor's Letter "What Is an Algorithm?" (Mar. 2012), one should consider the fact that in its most abstract and tangible form an algorithm is simply an integral number and that an interpretation of an algorithm as an abstract state machine reflects this truth. A researcher or engineer can then hypothesize other representations of the algorithm's number that might enhance computational efficiency.

One truly elegant notion supporting computational theory is that each of us can interpret a number's symbolic representation differently. Depending on which lens one uses, the view might reveal an integer, a floating-point value, a character in a collating sequence, or a computational behavior (such as function, procedure, or subroutine).

Likewise, one could take the view that an algorithm in its most abstract state is a complete collection of all the intangible thoughts and ideas that generate its design, whether based on lists of imperatives, nesting functions, states and transitions between them, transfer of tokens between containers in a Petri net, or any equivalent representation of a Turing machine. As such, an algorithm represents these ephemeral thoughts and ideas combined, ultimately resisting definitive and universal quantification and measurement.

To gain deeper insight into the abstractions surrounding a given algorithm, one would do well to also revisit earlier works on the topic. Pondering the original intent of the early visionaries of computing by republishing their work with critiques from today's leaders would be enlightening and enjoyable. As with many advances in science, their efforts were often met with stark criticism. However, reassessing their intent today could spark further refinement of modern concepts derived from computing's inherently iterative and optimizing research life cycle.

Jody Sharpe, Omaha, NE

Author's Response:

A mathematical model tries to capture (abstractly) the essence of the phenomenon being modeled. From this perspective, a person saying, "What is an algorithm?" would not be satisfied with the answer "An algorithm is an integral number."

Moshe Y. Vardi, Editor-in-Chief

Give the Past Its Due

David Anderson's Historical Reflections column "The Future of the Past" (May 2012) made a persuasive case for preserving computer hardware and software artifacts. I would add that computer science has not preserved its paper well, either; for example, when The American Federation of Information Processing Societies, the parent of both ACM and IEEE, folded in 1990, its records, periodicals collection, and books went to the dump, as did the records of the Microelectronics and Computer Technology Corporation, or MCC, a major consortium for which I worked in the 1980s. I welcome efforts such as those led by Bruce Damer, whose Digibarn Museum (http://www.digibarn.com/) houses paper materials, as well as working versions of vintage PC hardware and GUI software.

Computer science can do even more to preserve its legacy, encouraging and assisting ethnographers to look into technology development and use. Otherwise, future historians, as well as computer professionals, will be left to infer or guess how preserved artifacts were designed, used, and ultimately abandoned, much as we speculate today about the use of unearthed artifacts of ancient civilizations.

Jonathan Grudin, Redmond, WA

Assignment Is Asymmetric

When discussing a mistake in the C language, Paul W. Abrahams in his letter to the editor "Still Paying for a C Mistake" (Apr. 2012) concerning Poul-Henning Kamp's practice article "The Most Expensive One-Byte Mistake" (Sept. 2011) apparently fell victim to a C fallacy when citing the "celebrated C one-liner"

```
while (*s++ = *t++);
```

explaining its effect as "copies the string at `s` to the string at `t`." But in C it is the other way round, because `*s++ = *t++` is an assignment expression that works from right to left.[1] This means the assignment is a noncommutative operation that in C would be represented by the symmetric symbol =, not by an asymmetric symbol (such as := or ←). With such a symbol this fallacy would disappear like this

```
while (*s++ := *t++); or while (*s++ ← *t++);
```

These one-liners also observe the design rule that "form follows function." Using this notation, the statement (with the effect mentioned by Abrahams) should instead be

```
while (*s++ =: *t++); or while (*s++ → *t++);
```

Jürgen F.H. Winkler,
Feldkirchen-Westerham, Germany

Reference

1. Ritchie, D.M. *C Reference Manual*. Bell Telephone Laboratories, Murray Hill, NJ, 1975; http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf

Concerning the letter to the editor by Paul W. Abrahams (Apr. 2012), saying "C was designed with the PDP-11 instruction set very much in mind" conflicts with the fact that the C language was based on the B language, which was written for the PDP-7; the PDP-11 did not yet exist.

Concerning ++ and – modes, C's creator Dennis M. Ritchie wrote, "People often guess that they were created to use the auto-increment and auto-decrement address modes provided by the DEC PDP-11 on which C and Unix first became popular. This is historically impossible, since there was no PDP-11 when B was developed... Even

the alleged similarity between PDP-11 assembly and C is apparently 'a folk myth,' pure and simple, which continues propagating because it's a handy sound bite, not because there is any truth to it whatsoever"; see *The Development of the C Language* at http://cm.bell-labs.com/cm/cs/who/dmr/chist.html.

Moreover, while the C statement

```
while (*s++ = *t++) ;
```

can be compiled into two lines of PDP-11 code, the PDP-11 code given by Abrahams was not correct. The `mov` instruction moves a whole 16b word, or 2B at a time, and is likewise not correct. The auto-increment operator was +, not ++. A programmer would use `(R0)` to point to the character string, not `(@R0)`. The programmer can sometimes replace `(R0)` with `@R0`, but this code should not be concatenated in the manner cited by Abrahams. It is also possible to write `@(R0)+`, but its meaning is different from what is needed here. The code should be

```
A: movb (R0)+, (R1)+
bne A ;Branch if Not Equal (to zero)
```

However, this does not mean the PDP-11 could not have handled a length variable just as efficiently. Using `R0` and `R1` to hold the source and destination addresses for the string cited by Abrahams, it would suffice to move the length of the string into another register `R2`. The programmer would then have

```
A: movb (R0)+, (R1)+
sob R2,A
```

The `sob` instruction, which was misunderstood by some programmers, means "Subtract One and Branch (if equal to zero)" and was clearly quite powerful.

Roger Caffin, Sydney, Australia

Author's Response:

Winkler is correct; I inadvertently reversed `s` and `t`, with `s` the target and `t` the source. The code might have been more transparent if I had written it as `while (*t++ = *s++);` since `t` would indeed then stand for "target" and `s` for "source."

As for Caffin, I suppose both Poul-Henning Kamp (Sept. 2011) and I were taken in by the folk myth, which seems to have had wide currency. Neither of us was aware of Dennis M. Ritchie's enlightening paper from the 1993 History of Programming Languages Conference, as Caffin usefully cited. In it, Ritchie definitively clarified what he and Ken Thompson were thinking: "This change was made partially to avoid the limitation on the length of a string caused by holding the count in an 8- or 9-bit slot, and partly because maintaining the count seemed, in our experience, less convenient than using a terminator."

None of this negates Kamp's main point, on which I totally agree, that the decision to use a terminating null rather than a length count in representing strings was a mistake that ultimately proved very costly.

Paul W. Abrahams, Deerfield, MA

Still Wondering Why LINQ Matters

The sole purpose of the article "Why LINQ Matters: Cloud Composability Guaranteed" (Apr. 2012) by Brian Beckman seemed to be to try to show that LINQ is buzzword-compatible, as in "cloud" and "RESTful Web services." While I agree with Beckman's main point—that composability and designing for it are important and useful—I object to the article's exposition for the following reasons:

First, it claimed LINQ is applicable to graphs (among other things). If a graph is to be treated as a collection, it must first be linearized (such as through a topological sort), but what about cycles? Moreover, not at all obvious was that the result of a query is independent of the chosen linearization. Without concrete examples, the article was rather unconvincing when stating that, say, convolutions, continuations, and exceptions can be viewed as sequences usable for LINQ operators.

Second, the article was largely devoted to convincing the reader that a LINQ query can be represented as an abstract syntax tree (AST) traversable in postorder and encodable in a uniform resource identifier (URI)—which is kind of a boring, well-known fact. As an engineer-practitioner, I object to the idea of encoding queries in URIs though accept it for the sake of the intellectual exercise.

Third, its description of the `EXPAND` category of operators, claimed as crucial for composability, was lacking. For example, for the `SelectMany` operator, I had to infer from nearby text what `SelectMany` is supposed to do, but the article's other "examples" (one sentence and Figure 7) contributed nothing to my understanding of what the category is supposed to do. Why would "all orders from all customers" require `flattening/SelectMany` when only a single collection—orders—is involved?

Finally, I disagree with the claim that orders of arguments can ruin the property of operator composability. Unless Beckman expects developers to (manually?) encode queries in URIs by pasting in text fragments, a machine might reconstruct an AST from a representation with the "wrong" order of arguments and use it as a starting point for constructing the AST of another query. Not clear is how this might affect embedding in URIs, since the AST must still be reconstructed, modified, and reencoded.

Zeljko Vrba, Oslo, Norway

Author's Response:

LINQ's foundation is the monad type class borrowed from the Haskell programming language. A monad is an interface for managing monoidal container structures, including lists, sets, bags, and permutations. LINQ does not presuppose or guarantee order. No matter how one might linearize a graph into, say, an ordered list or unordered bag, LINQ will not change the given order.

As for convolutions, continuations, and exceptions, I need only show that a collection is monadic to show it is also LINQable. For convolutions, under the Fourier representation, LINQ operates on vector spaces, which are monads. For the continuation and exception monads (called the Maybe monad), see Haskell documentation http://www.haskell.org/haskellwiki/Haskell.

Vrba and I will respectfully disagree on the order of arguments. I grant that machine reordering is a functional fix, but it also introduces an extra step at the call site I would rather avoid as a point of style.

Brian Beckman, Redmond, WA

Communications welcomes your opinion. To submit a Letter to the Editor, please limit yourself to 500 words or less, and send to letters@cacm.acm.org.