BACKUS78

Backus, John, "Can Programming Be Liberated from the von Neumanr style? Functional Style and Its Algebra of Programs", IBM Research Laboratory, San Jose, CACM August 1978, Volume 21, Number 8

The central claim of this paper is that most modern programming languages aren't different from one another in many key respects, and that the traits they share are not good ones. The paper criticises these languages, which Backus calls "von Neumann Style," and presents some ideas about the alternatives. Before he can criticise conventional languages, Backus first has to outline what they have in common: Von Neumann's conception of what a general purpose electronic computer consists of. For von Neumann, who was describing the various computer-building projects going on around him, a computer has two important parts: a store that can remember mass tables of data, and a processor to read, change, and write these values. These two parts are connected in such a way that the processor can ask the store a value to keep in a particular location. Backus pictures the connection between the processor and the store as the essential quality of a von Neumann machine. Backus pictures this connection as a tube between the processor and the store. This third part can transmit values and addresses between the other two parts, but only one at a time. Most uses for computers involve making big changes in the contents of the store. Bit changes have to be accomplished one change at a time, passing addresses and values back and forth through this tube. The tube thus becomes a bottle-neck. Most all computer hardware still fits this description. And most conventional languages are heavily influenced by this conception of what computing is. What have conventional languages inherited? Conventional languages have variables, which mimic locations in the store. Assignment statements describe which locations to move from the store into the processor, the value to send back to the store, and where to put it. Not only are large changes accomplished one step at a time by the low-level hardware, but the "high-level" programmer conceives of them in the same way. Conventional languages also split programs into expressions and statements. Expressions follow simple rules. One expression can be proved equivalent to another using high school algebra. Statements, on the other hand, are full of side effects, and the each step depends on previous side effects. In these languages, most programming has to be done with the statements instead of the expressions. Backus thinks things would be better if programming could mostly be done on the expressions side of the split. Backus talks about some of the alternatives to the standard model of computing. Simple operational models like Turing machines and automata are based completely on the idea of state. That is to say that they are "all statement and no expression". Applicative models like Church's Lambda calculus and pure Lisp are "all expression and no statement." Their problem is that they are not history sensitive, which limits their usefulness. Dialects of Lisp that are commonly used are history sensitive. History sensitivity requires that the model include some kind of state. But Backus believes that each step of the computation shouldn't have to rely on the state. Because standard Lisps allow side effects anywhere within expressions, each step can be affected by the state. What Backus wants to find are systems with state (so they can be history sensitive) in which the bulk of the computation done is independent of the state. Half of the problem is figuring out what the question is. Backus presents the question well. But he admits that he has "not yet found a satisfying solution to the many conflicting requirements that a good language must resolve." In this paper he sketches a four part approach towards a class of solutions. The first element is a simple functional programming language. The second is an algebra of functional programs. The third step is to extend the capabilities of the simple language. The fourth is to introduce a state with simple transitions that occur only once per major computation. The simple functional programming language presented has no variables. Backus has at least two reasons for this. First, he associates variables with a von Neumann approach to programming, as mentioned above. What he ignores is that naming objects is a natural part of problem solving. Further, it increases program clarity. The second reason is that a program with variables needs to have a procedure heading added to make it general, and this introduces complex substitution rules. It seems that he exaggerates a little here: Lamboa expressions are as easy to write as their bodies and substitution rules are only complex when side effects are present. Backus demonstrates how programs written in this style can be manipulated like algebraic expressions. Using this technique, he spends a page proving that two one-line programs do the same thing. In fact, except for one 8-line program, all example programs are one-liners. This paper is worthwhile reading, however, if only for Backus's statement of the problem.