Can Programming be liberated from the Von Neumann style ? A functional style and its Algebra of Programs.

-John Backus

Summary by: C.Phaneendra Reddy.

When Von Neumann and others conceived an von neumann computer, it was an elegant, practical, and unifying idea that simplified a number of programming and engineering problems that existed in the past. Von neumann computers are built arund a bottleneck. A program must make the overall change in the store by pumpng vast number of words back and forth through the von neumann bottleneck, the one word-at-a time concept though that bottleneck. Conventional programming languages are large, complex, and inflexible. Their limited expressive power is inadequate to justify their cost and size. There are three types of classes where programming languages are classified: they are simple operative models, appicative models. And von neumann models. Conventional lanuages are based on the programming style of the von neumann computer. Thus variables= storage cells; assignment statements=fetching, storing and arithematic; control statements=jump and test instructins; the symbol ' = ' is the linguistic von neumann bottleneck. Von Neumann languages also split programming into a world of expressions and a world of statements; the first of these is an orderly world, the second is a disorderly one, a world that structured programming has simplified somewhat, but without at- tacking the basic problems of the split itself and of the word-at-a-time style of conventional languages. Programming languages appear to be in trouble. Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more. A large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such

names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent from the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must be generated either by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must have been sent ... and so on. Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word- at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it. Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our thirty year old belief that there is only one kind of computer is the basis of our belief that there is only one kind of programming language, the conven- tional--von Neumann--language. The differences be- tween Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem. Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von

Neumann bottle- neck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does. Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces aone-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.
John Backus explained the above problems of conventional programming languaues . He tried to do fixation on Von neumann languages. He has continued the primacy of the von Neumann computer, and our dependency on it has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. He discussed about the Functional programming style. An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hier- archically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages. Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms. This algebra can be used to transform programs and to solve equationswhose "unknowns" are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws and are carried out in the same language in which programs are written. Combining

forms are chosen not only for their programming power but also for the power of their associated algebraic laws. General theorems of the algebra give the detailed behavior and termination conditions for large classes of programs. A new class of computing systems uses the functional programming style both in its programming language and in its state transition rules. Unlike von Neumann languages, these systems have semantics loosely coupled to states--only one state transition occurs per major computation. He then explained about word-at-a-type programming. In seeking an alternative to conventional languages we must first recognize that a system cannot be history sensitive (permit execution of one program to affect the behavior of a subsequent one) unless the system has some kind of state (which the first program can change and the second can access). Thus a history-sensitive model of a computing system must have a state-transition semantics, at least in this weak sense. But this does not mean that every computation must depend heavily on a complexstate, with many state changes required for each small part of the computation (as in von Neumann languages). To illustrate some alternatives to von Neumann lan- guages, I propose to sketch a class of history-sensitive computing systems, where each system: a) has a loosely coupled state-transition semantics in which a state tran- sition occurs only once in a major computation; b) has a simply structured state and simple transition rules; c) depends heavily on an underlying applicative system both to provide the basic programming language of the system and to describe its state transition. He then gave an informal description of a class of simple applicative programming systems called functional programming (FP) systems, iwhich "programs" are simply functions without variables.The description is followed by some examples and by a discus- sion of various properties of FP systems. An FP system is foundedon the use of a fixed set of combining forms called functional forms. These, plus simple definitions, are the onlymeans of building new functions from existing ones; they use no variables or substitution rules, and they

become the operations of an associated algebra of programs. All the functions of an FP system are of one type: they map objects into objects and always take a single argument. FP system has a set of functions that depends on its set of primitive functions, its set of functional forms, and its set of definitions. In particular, its set of functional forms is fixed once and for all, and this set determines the power of the system in a major way. For example, if its set of functional forms is empty, then its entire set of functions is just the set of primitive functions. In FFP systems one can create new functional forms. Functional forms are represented by object se- quences; the first element of a sequence determines which form it represents, while the remaining elements are the parameters of the form. The ability to define new functional forms in FFP systems is one consequence of the principal difference between them and FP systems: in FFP systems objects are used to "represent" functions in a systematic way. In an AST system, naming is accomplished by functions . Many usefulfunc- tions for altering and accessing a store can be defined (e,g., push, pop, purge, typed fetch, etc.). All these defi- nitions and their associated naming systems can be in- troduced without altering the AST framework. Different kinds of "stores" (e.g., with "typedcells") with individual naming systems can be used in one program. A cell in one store may contain another entire store. The important point about AST naming systems is that they utilize the functional nature of names Thus name functions can be composed and combined with other functions by functional forms. In contrast, functions and names in von Neumann languages are usually disjoint concepts and the function-like nature of names is almost totally concealed and useless, because a) names cannot be ap- plied as functions; b) there are no general means to combine nameswith othernames and functions; c) the objects to which name functions apply(stores) are not accessible as objects. The failure of von Neumann languages to treat names as functions may be one of their more important weaknesses. In any case, the ability to use names as functions and stores as objects may turn out to be

a useful and important programming concept, one which should be thorougly explored.