Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

$Anurag\ Mishra^1$, $SridharChimalakonda^2$
Indian Institute of Information Technology, Chittoor, Sri City, A.P., India
$anurag.m13@iiits.in^1$,$sridhar_ch@research.iiit.ac.in^2$

***Abstract*—According to J. Backus, conventional style of programming has been growing enormously but it hasn't been growing stronger. By stronger, he means that it has got loads of defects at it's very basic level and that causes it to be both "fat and weak". This defect comes from the Von Neumann style of programming. The basic problem of conventional programming is their division of programming into statements, inability to effectively use powerful combining forms. He then introduces the concept of functional programming. Functional programming is much more useful than conventional programming as it lets us combine various forms to create programs. The data used in functional progamming is structured and is nonrepetitive and nonrecursive. It enables the use of combining forms which can use high level programs which can further can be used to develop more high level programs which was out of scope in conventional programming. A new class of computing systems have been developed that uses the functional programming as its programming language and also in its state transition rules.**

***Index Terms*—functional programming, algebra of programs, combining forms, functional forms, programming languages, von Neumann computers, yon Neumann languages, models of computing systems**

I. Introduction

A. Conventional Programming Language

Each new language invented is simply inheriting most of the features directly from its predecessors and add some new features. Instead of looking at the programs, the inventors are just looking at the ways of inventing new language. The programming language is just becoming fatter day by day becuase of the same. There is an urgent need for the development of a language that looks at the program rather than just meeting the needs.

B. Models of Computing Systems

There is always a model of a computing system that its programs control. Some models are pure abstractions, some are represented by hardware, and others by compiling or interpretive programs. There are various criterias of choosing models such as:
{i} is there an elegant and concise mathematical description of the model,
{ii} does the model has a notion of storage or not i.e, whether it stores the information to be later used by another program,
{iii} does a program successively transform states (which are not programs) until a terminal state is reached, and

Table 1: Models of Computing Systems

	Simple Operational	Applicative	Von Neumann
Examples	Turing Machines	Pure LISP, FP	Conventional
Semantics	Simple State	No state	Complex state
Program Clarity	Unclear	Clear	Clear

{iv} are expressions of a process or computation?

He characterized models into 3 categories:
{i} Simple Operational models - They are concise, have storage, program clarity is very limited and state transition is there. {ii} Applicative models - Functional Programming, no storage, programs are clear and conceptually useful. {iii} Von Neumann models - Complex, Bulky, have storage and not conceptually clear.

He also points towards the major defects in Von Neumann languages such as their gross size and inflexibility, lack of useful mathematical properties and the obstacles they present to reasoning about programs. Although a great amount of excellent work has been published on proving facts about programs, von Neumann languages have almost no properties that are helpful in this direction and have many properties that are obstacles (e.g., side effects, aliasing). proofs about programs use the language of logic, not the language of programming. Proofs talk about programs but cannot involve them directly since the axioms of von Neumann languages are so unusable.

II. Bottleneck in Conventional Programming - John Backus View

Von Neumann computers is composed of 3 parts: Central Processing unit, Storage and a connecting tube that can transmit a single word between the CPU and the storage for the program to be executed. John Backus called this tube as Von Neumann bottleneck as everytime the word had to be sent between the storage and CPU, it had to flow through that pipe. According to him, most of the data that flows through this pipe is not useful and is merely the names of the data. We can think of more primitive ways of developing systems that can perform better than just pushing a single word at a time. This leads to creation of the word at either CPU level or storage level and is then transmitted. Until this is completed, the program does not move forward.

Conventional programming are abnormal state forms of Von Neumann PCs. All the Conventional programming, for example, Fortran, Algol depend on programming style of Von Neumann. In these Conventional programming, we assign, instruct, get and do comparable operations. Task in the guidelines characterize the bottleneck in the Von Neumann dialects. Consider a common program; at its middle are various task articulations containing some subscripted factors. Every task proclamation creates an one word outcome. The program must bring about these announcements to be executed ordinarily, while modifying subscript qualities, so as to execute the store, since it must be done single word at once. The software engineer is subsequently worried with the stream of words through the task bottleneck as he outlines the home of control explanations to bring about the important redundancies.

The assignment operator splits the instruction or statement into 2 parts: one on the right side that contains the algebraic operators defined in a sequential and logical manner and the other on the left hand i.e, assignment statement itself. All the other statements of the language exist in order to make it possible to perform a computation that must be based on this :the assignment statement.

The second issue of von Neumann dialects is that their variable parts have so minimal expressive control. Their colossal size is articulate evidence of this; all things considered, if the developer realized that each one of those features, which he now incorporates with the system, could be included later as variable parts, he would not be so anxious to incorporate them with the system.

In the standard models, an address is sent through the tube. Now, for this to be a valid operation, it is required that the address is either generated in some CPU process. On the off chance that the address is sent from the store, then its address should either have been sent from the store or produced in the CPU, etc. In any case, if the address is produced in the CPU, it must be created either by a settled run or by a direction that was sent through the tube, in which case its address more likely than not been sent etc. One hindrance to the utilization of joining structures is the part between the expression world and the announcement world in von Neumann languages. Practical structures normally have a place with the universe of expressions; however no matter how effective they will be they can just form expressions that create a single word result.

III. Functional programming - Advantages and Limits

Functional Programming Systems are characterized by:

"An FP system has a single operation, application. If f is a function and x is an object, then f:x is an application and denotes the object which is the result of applying f to x. f is the operator of the application and x is the operand."

Objects are "bottom" (or "undefined"), atoms, or sequences of objects, and a whole set of primitive functions is suggested that inspect, shorten, extend, merge, distribute, and massage objects. Most of these operations are defined in terms of rearranging and/or deleting and/or creating multiple copies of sequence elements.

The structure and syntax of functional programming systems is like:

- A set of objects O.
- A set of functions F. It maps objects into objects.
- An operation and application.
- A set of functional forms. F and O combines to form new F
- A set of definitions that define some F and assigns a name to each definition.

The operational and Behavioural properties of functional programming systems are:

- Computation of some f:x where f is a function and x is an object is the evaluation of the expression.

A. Von Neumann Program for Inner Product

```
c = 0
for i=1 step 1 until n do:
  c = c + a[i] x b[i]
```

B. Functional Program for Inner Product

```
Def Innerproduct
  = (Insert +)o(ApplyToAll x)oTranspose
```

Problem with Von Neumann program (as defined by J. Backus):

- It is dynamic and repitive.
- There is always an invisible state involved during the execution of programs according to which the program runs.
- It is structural and requires procedural declaration of all its variables.
- Only one word at a time is communicated between CPU and storage by repetition of it.

Advantages of Functional Programming are:

- It does not keep any states and instead it is executed only based on the arguments that passed through it.
- It is non repetitive i.e we can understand about it without having to mentally execute it.
- There is no transmission of single words between CPU and storage involved and instead it looks at the whole conceptual unit as one while executing the program.
- It is not specific to certain kind of data. It can work for any conceptual data.

Limitations of Functional Programming : Functional programming has several limitations:

- There is no state concept.
- It is a fixed system once defined.
- We cannot change the I/O
- Defining of new functional forms is prohibited.

IV. Objects and Functions in Functional Programming

An object x is either an atom, a sequence <x .... , X_n> whose elements x_i are objects, or "bottom" or "undefined". The atom ϕ is used to denote the empty sequence and is the only object which is both an atom and a sequence.

All functions fin F map objects into objects and are bottom-preserving: f:Object = Object, for all f in F. Every function in F is either primitive, that is, supplied with the system, or it is defined (see below), or it is a functional form. Following is the solution proposed through Functional Programming:

- Selector : <x_1, x_2, x_3,.....,x_n> -> x_1
- Tail : Ti:x = <x_1, x_2, x_3,.....,x_n> -> <x_2, x_3,....., x_n>
- Atom predicate, equality predicate, null predicate
- Reverse a sequence
- Distribute (pair an element with all the elements of a sequence)
- Transpose (matrix transposition)

Next he depicts some arrangement of Functional Structures, i.e. methods for altering capacities - Composition, Construction, Insert, Apply to All (which applies a function to all elements of its operand sequence separately). The illustrations are extremely conventional (factorial, inner-product, matrix multiplication). If f is a primitive function, then one has its description and knows how to apply it. If f is a functional form, then the description of the form tells how to compute f: x in terms of the parameters of the form, which can be done by further use of these rules. If f is defined, Def = r, then to find f:x one computes r:x, which can be done by further use of these rules. If none of these, then f:x = 1. Of course, the use of these rules may not terminate for some f and some x, in which case we assign the value f:x = 1.

Some functional forms as suggested by John Backus:

- Composition: (f(g)):x $\equiv$ f:(g:x)
- Condition: (p -> f;g):x $\equiv$ p:x = T
- Constant: $\bar{x}$:y $\equiv$ y
- Insert: f:x $\equiv$ x = <x_1>
- Apply to all
- While

V. Conclusion

He concludes that formal systems dedicated for functioning programming can be developed along with new forms for functional programming. He also proposes a new hybrid language that has an applicative subsystem, set of definitions of applicative subsystem and a set of transition rules to define I/O.

John Backus intends the software developer to focus on programs rather than the languages. He focusses on improving reasoning about the software developed. We should follow equational reasoning to solve the programs and this structure is less complex than the Von Nuemann model. He also emphasizes on developing new architecture to execute applicative systems.

Even today, functional programming has not become mainstream. He likewise examines various applicative computer designs and the need to create and test more functional models of practical frameworks as the future reason for such outlines.

VI. Current Situation

Even today, FP has not become too mainstream. However, many specific systems have come up that focuses on functional programming. Some FP constructs are being implemented such as pattern matching etc. Also, people are focussing on using FP while concurrency as they feel that the tasks would become much more simplified and easier if implemented using FP systems.

References

[1] Backus, J., Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs., Comm. ACM, 21, 8 (Aug. 1978) 613 - 641

[2] Rem, M., Associons and the Closure Statement, Ph.D Thesis, Eindhoven University of Technology, 12 October 1976

[3] Scott, D., and Strachey, C. Towards a mathematical semantics for computer languages. Proc. Symp. on Comptrs. and Automata, Polytechnic Inst. of Brooklyn, 1971.

[4] Scott, D. Lattice-theoretic models for various type-free calculi. Proc. Fourth Int. Congress for Logic, Methodology, and the Philosophy of Science, Bucharest, 1972