Fachhochschule Nordwestschweiz
Hochschule für Informatik

«Can Programming Be Liberated from the von Neumann Style?»

John Backus 1977

Aaron Ebnöther
3. Oktober 2025

Schedule

1. Context
2. Summary
3. My thoughts
4. Discussion

Feel free to interrupt at any time, PL circle is about exchange and conversation!

Who was John Backus?

- Studied Math
- Was at IBM most of his professional life
- Mainly worked on programming languages, compilers and language theory
- Main figure in developing FORTRAN
- Backus-Naur Form (BNF)

FORTRAN

- Heavily imperative
- Assignment statement is very important
- Archaic syntax

```
      PROGRAM SUM_ARRAY
      INTEGER N, I
      PARAMETER (N = 5)
      INTEGER A(N)
      INTEGER TOTAL

      DATA A /1, 2, 3, 4, 5/

      TOTAL = 0
      DO 100 I = 1, N
         TOTAL = TOTAL + A(I)
  100 CONTINUE

      PRINT *, 'SUM IS ', TOTAL
      END
```

Premise

- There’s no real progress in PL development
- No new ideas
- Feature creep leads to languages becoming more bloated

- Most languages are heavily shaped by how the hardware works (von Neumann model)
- Assignments are THE core concept in most PLs
- The world of assignments is unorderly, little to no structure
- There’s no useful theory to analyze or transform or simplify programs

- Acknowledges existence of lambda calculus
- Notes that a system without “history-sensitivity” (mutable state, IO) is useless in practice

Backus's Vision

– It should be easy to see what a program calculates just by looking at it
– State transitions should be used sparingly
– It should be possible to analyze programs algebraically similar to how we can deduce meaning of mathematical terms by applying a combination of theorems

His approach to achieving this vision involves four core elements:

- Functional style of programming
- Algebra of functional programs
- Formal Functional Programming System
- Applicative State Transition System

FP systems

- Like lambda calculus but more restricted
- Lambda calculus is too flexible which makes it difficult to develop an algebra for it
- There’s a fixed number of functions such as add, tail, transpose, length, equals, etc…
- Functional Forms are like higher-order functions or combinators
  - Function composition
  - Insert (like fold or reduce)
  - Apply to All (like map)
  - Binary to Unary (similar to uncurrying)
  - While
  - If-then-else
  - Etc…

FP Systems Factorial Program

$$\textbf{Def } ! \equiv eq0 \rightarrow \bar{1};\ \times\circ[id, !\circ sub1]$$

where

$$\textbf{Def } eq0 \equiv eq\circ[id, \bar{0}]$$
$$\textbf{Def } sub1 \equiv -\circ[id, \bar{1}]$$

FP Systems Summary

Limitations:

- Not history-sensitive
- Is not necessarily Turing-complete depending on what primitive functions and functional forms are available
- Result of a computation can never be a function because function expressions are not objects by definition
- Performance will be poor when naively translated to von Neumann hardware

Main advantage:

- System is simple, easier to reason about, easier to create an algebra for this kind of system

Algebra of FP Systems

- There are ways to proof conventional programs -> cumbersome and requires vast mathematical knowledge
- Goal: Normal programmers can proof their programs correct
- Idea: To prove a program, you use the same language the program is written in
- Some laws define rules for how to transform one program into an equivalent one

I.1 $[f_1, \dots, f_n] \circ g \equiv [f_1 \circ g, \dots, f_n \circ g]$

I.2 $\alpha f \circ [g_1, \dots, g_n] \equiv [f \circ g_1, \dots, f \circ g_n]$

1. $a^m a^n = a^{m+n}$
2. $\dfrac{a^m}{a^n} = a^{m-n}$
3. $(a^m)^n = a^{mn}$
4. $(ab)^n = a^n b^n$

Algebra of FP Systems

- Backus provides two more advanced proofs
  - Recursion Theorem
  - Iteration Theorem
- They provide a framework for working with recursive and iterative programs
- Recursion Theorem
  - Given a program that follows a certain recursive structure it is valid to expand the function so you can obtain a non-recursive expression
  - You can use this expression to inductively prove correctness of a program (like Backus does for the factorial program)
- Idea: Use general laws and theorems to prove properties of programs instead of starting at square one for each program

FFP Systems (Formal FP)

- Builds on top of FP Systems
- Objects can represent functions
- Can compute new programs as data
- Allows user defined functional forms
- Still pure and therefore no state
- System is more complex therefore harder to build a useful algebra for it

AST (Applicative State Transition)

- How to implement these ideas in the real world where mutable state and IO is needed?
- Basic idea:
  - Wait for user input or some external trigger
  - Evaluate the program which is mostly an FP program
  - The program can access the current state but not modify it during computation
  - The result of the program is a new state which is used next iteration
  - Repeat

Which of Backus's idea were implemented?

- Functional Programming has grown significantly more popular since the paper's publication
- Languages such as Haskell embrace the idea to have a simple core language and make extensive use of combinators to build complex systems from simple primitives
- Today, even in imperative languages simple combinators such as map or fold are commonplace
  - Properties of these combinators exploited for example to improve performance
- AST model has influenced how we think about state manipulation
  - Nowadays, it's considered good practice to reduce mutable state
  - Haskell has similarities, you usually try to compute a value purely and then at the very end mutate some state using IO, similar to the "update every major computation" idea
  - React is very similar to AST

Which of Backus's idea were NOT implemented?

- Hardware to natively run FP programs
- His FP and FFP systems never really caught on
- Concept of working within a system that has a fixed set of combinators

Discussion