It's good nice to be here and

~~It's a pleasure to be able~~ to participate in ~~this series of~~ these talks ~~about future computing systems.~~ It is particularly fun to come to a nice conservative lab like this and promote some revolution. But that is what I want to do. I want to urge you to throw off the shackles of the past and become revolutionaries in the field of programming.

Seriously though, I want to talk about the impact of programming on the future of computing and about why those shackles of the past are keeping programming in the dark ages.

Apparently it is clear to almost everyone that programming is an immense obstacle ~~in the way of~~ blocking progress in computing. Thus we are always hearing about "the software crisis", and about the cost, inflexibility and unreliability of programs.

So here's my short summary of the programming problem:

SLIDE 1

The Programming Problem

- Computers - fast + cheap
  - becoming more so
- Programming - slow + expensive
  - becoming more so

And here's a bit more detail:

SLIDE 2

Programming so complex that:

- Average user can't write program
- Must rely on "application" programs
  - very limiting, or
  - have own complex language

Different scale & kind of thinking for
- problem
- program

Ken Wilson

Programs very expensive

Not reusable for building new programs

⇒ Millions of users don't have full access to computers

[EXPLAIN]

END SLIDE 2

So the proggramming problem is this: why is programming still the expensive, slow, confusing mess that it is?

Surely, if we were working with the right basic concepts, we could have made better progress. So I think it is inescapable that the cause of this frustrating situation, this software crisis, is a very pervasive and fundamental one; it clearly requires a profound revolution in our concepts if we are to make real progress.

Programming is in a situation today similar to that of astronomy before Copernicus. In the fifteenth century astronomers were struggling with complicated but largely successful theories to explain the motions of the planets. Of course, now we can see that if you take the Earth as a fixed point, as they did, then solving problems in astronomy is going to be very difficult. And that difficulty is going to persist despite the cleverness of the mathematics you use. As long as you insist on the fundamental misconception of an Earth-centered universe, astronomy is going to be a terrible mess.

Today we have a similar misconception about the nature of programs; it is a very fundamental misconception. And programming will continue to be the difficult mess that it has been for the last thirty years, unless we recognize it.

In the 15th and 16th centuries mathematicians succeeded in producing theories that helped astronomers to compute the motion of the planets -- although with great difficulty. Despite these difficulties, many astronomers, including one of the greatest astronomers of the time, who was born 3 years after the death of Copernicus, all refused to believe that the earth-centered concept was the source of this great complexity and steadfastly rejected Copernicus' ideas.

We can now see that early astronomers might have suspected some fundamental misconception, simply because of the complexity of their theories and the work it took to calculate a planet's position. Today, computer scientists have been working with the same conventional concept of programs for thirty years. They have developed some elegant theories about such programs, but programming continues to be essentially the same complicated mess it has always been.

This situation is illustrated by two earlier talks in this series. Tony Hoare presented a really beautiful theory about conventional programs. He said that

SLIDE 3

Programming is a mathematical activity

— Tony Hoare

On the other hand, here is my capsule summary of Ken Wilson's talk:

SLIDE 4

Programming is a complicated mess

– Ken Wilson (?)

END SLIDE 4

Ken Wilson devoted much of his talk to illustrating how messy and complicated programming is for scientists. If Hoare is right, you would expect that a brilliant scientist like Ken Wilson would be using the mathematics of programs to simplify his work.

So why don't pretty theories translate into simpler concepts and less work for the programmer? In short, why is programming still such a terrible mess?

I believe that the answer is that our concepts about programs are flawed, not just in a few details but fundamentally, in ways that are as devastating to the advance of programming as the Earth-centered concept was to astronomy.

But I don't pretend to be a latter-day Copernicus of computer science who has the final, just-right concept for programs. In fact, several changes in our basic concepts will probably be required to make programming a much simpler discipline.

In astronomy, all problems did not go away with the change from an earth-centered concept to a sun-centered one. There were other issues that were independent of these concepts, such as issues involving relativity.

And it will probably be that way with programming; several concepts, including at least one fundamentally new one, will have to come together before programming can become an order of magnitude simpler and clearer. That is the kind of improvement that we desperately need. Bringing it about will require a revolution in the way we think about programs; it will require a rebirth of creative effort by computer scientists.

I do believe that we can identify one fundamental flaw in our concept of programs: it is the store-centered concept, the idea

that a program is a mapping of stores into stores, the basic concept of conventional programs. I want to try to convince you that the store-centered, von Neumann concept is as great a hindrance to the advance of programming as the earth-centered concept was to that of astronomy.

My talk will not be entirely a negative argument; it will propose an alternative concept, the notion of function level programs that I've been working on with my friends in San Jose for some time now. This new concept of programs eliminates a great many of the complexities of the conventional ~~concept~~ one.

However, the function level concept introduces some new problems because it does not easily cover some areas that are handled quite simply by the von Neumann concept, such as maintaining a permanent file system and controlling input-output devices.

So other new concepts are needed to simplify these areas, in the same way that the function level concept simplifies the central area of programming: namely, the transformation of input data into output.

Many people, including my friends and I, are working on such concepts, but it is still too early to say how well these are going to work out.

Now let's look at the basic problem with the von Neumann concept of programs:

SLIDE 5

Conventional = von Neumann concept of program:

Program : Stores $\longrightarrow$ Stores

P_1 $a := b + c$ $\begin{cases} \langle x, y \rangle \longrightarrow x+y & \text{Purpose} \\ b, c \longrightarrow a & \text{st. plan} \end{cases}$

P_2 $a := \sqrt{b}$ $\begin{cases} x \longrightarrow \sqrt{x} \\ b \longrightarrow a \end{cases}$

P_3 $d := \sqrt{a}$ $\begin{cases} x \longrightarrow \sqrt{x} \\ a \longrightarrow d \end{cases}$

Program $\Longleftrightarrow$ purpose + storage plan

$P ; Q$ meaningful $\Longleftrightarrow$

purposes and storage plans compatible

$P_1 ; P_2$

purposes compatible

storage plans not compatible

$P_1 ; P_3$

$\langle x, y \rangle \longrightarrow \sqrt{x+y}$ combined purpose

$b, c \longrightarrow d$ combined st. plan

[EXPLAIN]

The need to have programs which must have both compatible purposes and compatible storage plans in order to combine them means that it is not generally possible to combine two programs that have not been planned together.

At the very least it means that you have to design a master storage plan and then change the storage plans of all the programs you are going to use so that they fit this one master plan. This is what we do when we use subroutines in writing a program. Storage planning is an archaic activity that should not be required of the programmer.

The worst aspect of the von Neumann concept of programs is that every program has a complex extra dimension -- its storage plan -- that is totally useless but absolutely essential to know if you are going to use the program. And that extra dimension prevents you building new programs from existing ones without first changing them.

Now let's contrast the von Neumann concept with a different one, one that views programs as functions or transformations.

SLIDE 6

Function level concept of programs

Program : objects ⟶ objects
(atoms + seqs)

P_1 + purpose: $\langle x, y \rangle \longrightarrow x+y$

P_2 sqrt purpose: $x \longrightarrow \sqrt{x}$

Program $\Longleftrightarrow$ purpose

$P_2 \circ P_1$ meaningful $\Longleftrightarrow$
Purposes compatible

$\langle 1,3 \rangle \xrightarrow[+]{P_1} 4 \xrightarrow[\text{sqrt}]{P_2} 2$

$P_2 \circ P_1$

In this concept of programs, a program is the same as its purpose, a program is just the mapping that transforms its input into its output.

Instead of stores, programs map the objects you're concerned with into other objects. These objects are either atoms, such as numbers and symbols, or sequences built from either atoms or other sequences.

Thus in this view the program P1 of the previous example becomes just the operation "+" that transforms a pair of numbers into their sum, and P2 becomes just the operation square root. Such programs have no need to give names to their inputs or outputs.

Thus the analog of our previous example is the composition of P2 and P1 [but now we read from right to left to have P1 applied first]. Only now, with this new concept of programs as functions, it always makes sense to compose programs if it makes sense to compose their purposes, because program and purpose are identical. Programs no longer have that unnecessary extra dimension, a storage plan.

Thus P2 · P1 is a new program that transforms a pair of numbers into the square root of their sum.

END SLIDE 6

Since the von Neumann concept makes it so difficult to put programs together and to reason about and transform programs, it seems strange that we don't discard it and adopt the more elegant one. But I think there is a very fundamental reason why we haven't. That reason concerns the most basic tradition of natural language and of mathematics, the tradition of how we interpret sentences.

SLIDE 7

Traditional interpretation rule

Natural language

The cat runs.

Mathematics

$a + b$

Conventional programs

$c := a + b$

Store = association between names and referents

[EXPLAIN]

END SLIDE 7

So the von Neumann concept is closely tied to our traditions of language and breaking with that concept means breaking with those traditions.

It will be difficult for people to accustom themselves to non-von Neumann languages, despite the advantages that come from doing so.

Galileo had to contend with ancient religious beliefs in order to establish the theory of Copernicus; and so will proponents of function level programs have to contend with ancient traditions of language and mathematics before computer scientists will recognize the simplicity of this new concept.

Anyway, it is difficult for people to get used to dealing with programs that do not name the things they operate on, and that is what function level programs do. To understand how they do this, let's take a closer look at how this new kind of program is put together and how they differ from von Neumann programs.

The real work in a von Neumann program is done by expressions. An important element in expressions are user-defined functions. Let's consider how these functions are defined.

SLIDE 8

Function definitions

Object level

for all objects x

$$f : x = g : (h : x)$$

Function level

$$f = g \circ h$$

where, for all functions r, s & all objects x,

$$(r \circ s) : x = r : (s : x)$$

Functions can be defined either at the object level or at the function level. We are used to defining them at the object level. But when you look at it closely, you find that this is really a strange process.

You are given some already defined functions, g and h in this case, and you want to define f. So the first thing you do is descend from the level of functions, that is, the level of g and h, and introduce an object you call x.

You then start applying the functions you have to objects, here you apply h to x and then g to that result, until you have built the result-object g:(h:x). Now you have an object, but you want to define a function; so you say that the object f:x equals the result object g:(h:x) for all objects x.

Thus you never really define the function, you define the object f:x, and you do that for every object x.

The real difficulty of object level definitions is that they focus on the mathematics of objects. And that mathematics may be interesting, but it is quite different from the mathematics of programs, and this is the mathematics that is important for programmers, if you want general theorems about programs.

The second way of defining a function is at the function level.

In the object level definition, we are concerned with constructing objects; in this function level definition, we are concerned with constructing functions. Our definition simply says that f is built by applying the operation of composition to the functions g and h.

Notice that we do not need a variable in this definition of f, whereas the variable x is essential in the object level definition. All we need to know is what composition means, and it has the usual meaning in mathematics, as shown.

END SLIDE 8

Now we've seen how functions can be defined at the object level or at the function level. So next, let's look at how von Neumann programs are put together. We've already noted that most of their work is done by expressions; and that these are object level expressions.

SLIDE 9

Von Neumann programs

Assignment = elementary programs

vbl := expression

Program forming operations (PFOs)

composition	P ; Q
if-then-else	if e then P else Q
while	while e do P

The most elementary von Neumann program is an assignment statement, consisting of a variable on the left and an expression on the right. Oddly enough, even though expressions are object level constructs, programs are built at the function level. That is, given some programs that you want to combine to form a new program, you combine them by program-forming operations, just as we build a function at the function level.

For von Neumann programs, these are the principal operations for building programs: composition [the semicolon], if-then-else, and while. I will assume that everyone is familiar with the way these operations work.

The important point to note is that they are operations, not punctuation, as the semicolon suggests. Thus P semicolon Q is a new program formed from P and Q by applying the operation composition.

But it is worth remembering also that most of the work of writing a program is writing expressions. Thus programming in von Neumann languages means working with two different technologies, one for building expressions and defining functions, the other for building programs. And one of these technologies is object level, the other is function level.

In the function level approach, programs are just functions, so we have a single technology of expressions for building functions and programs [which are the same].

SLIDE 10

Function level programs

Elementary programs (primitives)

$1 : \langle x_1, \ldots, x_n \rangle = x_1 \qquad tl : \langle x_1, \ldots, x_n \rangle = \langle x_2 \ldots x_n \rangle$

$+ : \langle x, y \rangle = x + y$

$sub1 : x = x - 1$ etc

$\overline{3} : x = 3$ constant-valued functions

Program forming operations

composition $\quad f \circ g : x = f : (g : x)$

condition $\quad (p \rightarrow f; g) : x = \begin{cases} f : x & p : x = T \\ g : x & p : x = F \end{cases}$

construction $\quad [f, g] : x = \langle f : x, g : x \rangle$

insert $\quad /f : \langle x_1, \ldots, x_n \rangle = f : \langle x_1, /f : \langle x_2 \ldots x_n \rangle \rangle$

$/+ = \Sigma$

apply-to-all $\quad \alpha f : \langle x_1, \ldots, x_n \rangle = \langle f : x_1, \ldots, f : x_n \rangle$

[EXPLAIN]

Now to make clearer how definitions of function level programs work, let's look at another example of a function level definition, this time using a function variable so that we can improve the readability of the definition.

SLIDE 11

Function definitions, cont'd

Object level for all objects x:

$$length:x = null:x \Rightarrow 0 ; +:\langle 1, length:(tl:x)\rangle$$

Function level for all functions x:

$$length \circ x = null \circ x \rightarrow \bar{0}; + \circ [\bar{1}, length \circ tl \circ x]$$

Setting $x = id$, use: id is unit for $\circ$

$$length = null \rightarrow \bar{0}; + \circ [\bar{1}, length \circ tl]$$

Closed form definition:

$$length = /+ \circ \alpha \bar{1}$$

$$/+ \circ \alpha \bar{T} : \langle A, B, C \rangle$$

$$= /+ : \langle \bar{T}:A, \bar{T}:B, \bar{T}:C \rangle$$

$$= /+ : \langle 1, 1, 1 \rangle$$

$$= 3$$

[EXPLAIN]

This example shows that every object level definition can be exactly mimicked by a function level one, and that there are other function level definitions that cannot be mimicked by an object level one.

The existence of these non-isomorphic definitions shows that the space of function level programs is richer than the space of object level ones, and that function level programming is therefore more powerful. This greater power comes from the fact that the programmer has not only the set of existing functions from which to build a new one, as in object level definitions, but also a fairly rich set of program forming operations that he can use to put them together. And these operations are lacking at the object level.

END SLIDE 11

Next I would like to discuss the mathematics of programs. You have heard from Tony Hoare about the mathematics of conventional programs and I want to sketch the mathematics of function level programs. In order to compare them, it is first necessary to talk a bit about two different kinds of mathematical disciplines. Some mathematics is procedural in nature; axioms and theorems tell you how to make a step in a calculation or deduction. This kind of mathematics is the basis for a methodology. But it does not provide direct answers to problems.

We tend to use procedural mathematics when we're working in a system that is too weak to provide general solutions for the problems we're interested in. Here's a small example.

SLIDE 12

A weak mathematical system

Domain: integers

Operations: $+$, $\sqrt{\ }$

Problem Given

$$\sqrt{x} = 3, \text{ find } x$$

Procedure: try $x_1 = 1$, $x_{i+1} = x_i + 1$ until $x_9 = 9$

A stronger system:

Domain: integers

Operations: $+$, $\sqrt{\ }$, $*$

Law: $\sqrt{x} = y \Rightarrow x = y * y$

[EXPLAIN]

In this system there are only two operations, plus and square root. There are no strong laws that relate one to the other. So to solve a problem like this one, we are reduced to some procedure like trying successive integers.

We can make the system a lot stronger by adding the operation of multiplication. When we do, we then have the law shown at the bottom, which gives the general solution to our problem and to many similar ones.

In contrast with such weak mathematical systems, there are a number of strong ones, such as the classical system of real numbers under addition and multiplication. If we consider systems that may have more than two operations, and ask, what are the important properties of strong systems, that is, systems that have equations for most of their laws and theorems, and that have general solutions to many classes of problems, then we might find that these are the properties we want.

SLIDE 12a

Strong mathematical systems

1. Single domain D (operations Θ)

$$\theta \in \Theta \Rightarrow \theta \in D^n \to D$$

2. Simply understood operations

$$\theta \in \Theta,\ d_1, \ldots, d_n \in D$$

understand $d_1, \ldots, d_n \Rightarrow$

understand $\theta : \langle d_1, \ldots, d_n \rangle$

understand $P\ Q \Rightarrow$ understand $(P \circ Q)$

3. Symmetric laws

For many pairs $\theta, \theta' \in \Theta$, there is a law

$$\theta(E_1(\theta') \ldots E_n(\theta')) = \theta'(F_1(\theta) \ldots F_m(\theta))$$

$$(a + b) * c = (a * c) + (b * c)$$

A typical system has a single domain and all its operations map one or more elements of that domain back into the domain. Thus for example, addition maps pairs of numbers into numbers.

If a system has more than one domain, then its equations become more complex. And it's difficult to relate what happens in one domain to what happens in another.

A second important property of a good system to work with is that its operations are easily understood. This means that if you understand what certain elements of the domain are, then you understand the element that you get by applying an operation to those elements.

For example, if the operation is composition, then this means that if you understand what two programs P and Q do, then you understand, very simply and directly, what the program P·Q does.

If operations lack this property of simple understandability, then it is hard to have a good intuitive understanding of the elements that these operations build and of the system as a whole.

The third important property of these systems is that they have laws that relate the behavior of one operation to that of another. The strongest laws of this kind are symmetric ones; these laws equate an expression whose main connective is the operation o, but have subexpressions involving o', with a second expression whose main connective is o' and has subexpressions involving o. For example, the distributive law of algebra is a symmetric one, since it expresses a product involving addition, as addition involving products.

These symmetric laws are the main way to express general transformations of a computation. Without them you can't change the way an element is computed; once its main operation is given, there is no general way to compute it by a different operation, unless its main

operation is part of some symmetric law.

Now we can see what the mathematics of programs looks like.

SLIDE 12b

Mathematics of programs

Domain = Universe of programs

Operations = Program forming operation

$op : \langle P_1, \ldots, P_n \rangle = P$

eg

$composition : \langle P\ Q \rangle = P \circ Q$

$condition : \langle P, Q, R \rangle = P \rightarrow Q ; R$

It is just a system whose domain is the set of programs, and whose operations are the program forming operations.

This sometimes confuses people, because programs themselves are operations. Nevertheless, if you want to have a mathematics of programs, then it is important to view them as just things, things that get transformed by program forming operations.

Thus for example, composition maps two programs into a new one, and condition maps three programs into a new one.

The important thing to notice is that the mathematics of such a system depends entirely on the properties of its operations.

As an example of such a system, let's look at the mathematics of function level programs.

SLIDE 13

Mathematics of function level programs

Basic axioms

For all programs p, f, g, h :

$$(p \to f; g) \circ h = p \circ h \to f \circ h; g \circ h$$

$$h \circ (p \to f; g) = p \to h \circ f; h \circ g$$

$$[f, g] \circ h = [f \circ h, g \circ h]$$

$$[(p \to f; g), h] = p \to [f, h]; [g, h]$$

= Symmetric laws

plus, for example

$\circ$ associative, unit = id

many laws for $\to$, eg

$$(p \to q; r) \to s; t =$$
$$p \to (q \to s; t); (r \to s; t)$$

many laws relating above 3 with $/$, α

+ many laws about primitive programs

[EXPLAIN]

And now let's look at the mathematics of von Neumann programs.

SLIDE 14

Mathematics of von Neumann programs

Basic axiom for all exprs b, all programs P, Q:

$$(\underline{\text{if}}\ b\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q);R = \underline{\text{if}}\ b\ \underline{\text{then}}\ P;R\ \underline{\text{else}}\ Q;R$$

only symmetric law relating composition, if-then-else, while

plus, for example,

; associative, unit = id

laws for if-then-else (≥ 2 occurence

Other operations (Hoare)

$P \cup Q$

$P \setminus Q = wp(P, Q)$

Here we find there is just one symmetric law for the 3 principal program forming operations; it relates composition and if-then-else. There is no symmetric law that relates while with either composition or if-then-else.

There are, again, laws about composition -- it's associative and has a unit; and there are a number of laws about if-then-else by itself, just as before.

Now we come to the question of other operations on von Neumann programs, such as those proposed by Tony Hoare. I am not clear whether he proposes to use these operations, such as P cup Q, and P under Q, as actual program forming operations, that is, as operations that would appear in a completed program, or whether, as I suspect, these operations are intended for reasoning about a program during its development, but would not appear in the final program.

In any case we can now compare the mathematics of von Neumann programs and that of function level programs with respect to our three criteria for strong mathematical systems. Of course, this means comparing the properties of their program forming operations, particularly their three principal ones.

SLIDE ~~15~~ 14b

Here we see that the von Neumann program forming operations are not mappings of just programs into programs; instead they have two domains, boolean expressions and programs. So, for example, While maps an expression b and a program P into the program "While b do P".

The fact that program forming operations have two domains instead of one leads to a great many complications and makes the mathematics of von Neumann programs weaker than it might be, because it is hard to reflect what goes on in one domain, in the other one, at least by means of equations.

With regard to the question of understanding program forming operations and the programs they build, if you understand an expression b and a program P, that does not mean that you understand "while b do P". In fact, there is a very simple expression b and a very simple program P, which are very easy to understand, and yet no one understands what "while b do P" does.

Similarly if you understand two programs P and Q, you do not understand very simply what "P;Q" does. As Tony Hoare remarked in his lecture, you can compute P;Q if you are good at manipulating quantifiers in the predicate calculus. And that's not a very simple thing, in fact, these expressions are notoriously hard to understand in some cases.

So the point here is that it is hard to have an intuitive understanding of what programs do, even when you understand their parts; and a lot of difficult methodology is needed to find programs that will do a specified job.

Slide 15

And finally there is the key question of symmetric laws, the question that determines the strength of the mathematics your system will have. As we saw, there is only one symmetric law for the three principal program forming operations, and none involving While, perhaps the most vital one of the three.

Mathematics of programs, equational

von Neumann PFOs

stores → {T, F}

1. Domains. = {boolean expressions
   PFO: Exp × Progrs → Progrs {programs stores → stores
2. Understanding PFOs. and their results
   Understand b, P ⇒
   understand <u>While</u> b <u>do</u> P
   Understand P, Q ⇒
   understand P \ Q
3. Symmetric laws.
   Only one for 3 principal PFOs

Function level PFOs

1. Domain. = programs
2. Understanding PFOs.
   for any PFO, eg understand f, g
   ⇒ understand [f, g]
3. Symmetric laws.
   4 for 3 principal PFOs
   plus many others

The consequence of this last fact is that the mathematics of von Neumann programs is weak, that you can do very little to express general transformations of programs.

slide 15

In contrast to that, if we look at those same three questions for the operations of function level programs, we find, first, that they operate on a single domain; that is, they combine just programs to form programs.

This means that the system of function level programs is fundamentally simple.

Second, they have the nice property that if you understand ~~some~~ several programs and build a new one from them by applying a program forming operation, then you always understand, very simply and directly, what the new program does. For example, if you understand two programs f and g, then you understand what [f,g] does, namely it produces the pair consisting of the result of f and the result of g.

This property means that it is easy to understand any constructed program, one step at a time, with as many steps to that understanding as there are program forming operations.

And finally, with respect to the question of symmetric laws, as we saw, there are 4 symmetric laws for the 3 principal program forming operations. Thus there is a symmetric law for every pair of the 3 operations, with 2 laws for one of these pairs.

This means that there is a strong mathematics of function level programs, with a lot of general theorems in the form of equations. This makes for a mathematics that is easy to apply, one that makes possible the general solution of ~~a large~~ many classes of problems once and

for all. The kind of mathematics that can be the foundation of a real engineering discipline. I'll show you some examples in a bit.

First, it is interesting to note that you can't strengthen the mathematics of von Neumann programs by adding construction as a program forming operation. If you could, there would be symmetric laws relating it to both composition and if-then-else. And you would then have the use of an extremely simple and powerful program forming operation.

SLIDE 15a

$[P, Q] : store$

$= \langle P : store \ , \ Q : store \rangle$

$= \langle store_P \ , \ store_Q \rangle$

$\neq store'$

So,

$[P \ Q] \notin stores \longrightarrow stores$

But you can't do that, because the construction of two programs always produces a pair of results, so the construction of two programs would not map stores into stores but rather stores into pairs of stores and such a mapping is not a von Neumann program because a pair of stores is not a store.

END SLIDE 15a

In any case, the evidence is very strong that the mathematics of von Neumann programs is weak and is doomed to remain weak unless someone can find some magical new program forming operation that obeys strong symmetric laws with the existing ones. It is hard to imagine such an operation.

Because the real difficulty lies in the basic property of von Neumann programs, the fact that they have both a purpose and a storage plan, and are therefore fundamentally difficult to combine meaningfully, no matter what operations you use. Recall that if these programs don't have a common storage plan, they can't be combined by any operation, unless it alters those plans.

The result of the very weak mathematics of von Neumann programs is that we are confined to a tedious, complicated methodology for developing them. We have no body of theorems that programmers can apply directly to program fragments simply by substitution.

Therefore programmers are constantly solving the same kinds of problems over and over again because we have no mathematics capable of expressing their general solution.

The structured programming methodology that Hoare and his followers have developed is a really admirable one, especially, given the enormous difficulties that von Neumann programs present and the lack of strong laws for program forming operations.

I believe this methodology resembles, in some ways, the one that enabled pre-Copernican astronomers to compute the positions of the

planets. Certainly, both are very complicated.

And so, even for brilliant programmers like Ken Wilson, this highly developed methodology still leaves programming a difficult, slow and confusing process.

I submit that the reasons for this are fundamental and require a new concept of programs in order to make real progress.

> Such progress would require that we have more powerful ways of building programs from existing ones.
>
> It would require a strong mathematics of programs so that we could accumulate a lot of knowledge in the form of general solutions for many different problems that arise in programming. And to be easily applicable these general solutions should be in the form of equations.

Although function level programs do not yet cover as large an area of applications as conventional programs, they do have many of the properties needed to make such progress.

Let me show you some examples of how some problems can be solved using general results in the mathematics of function level programs.

SLIDE 16

Directly applicable results, example

Problem Given

$$sum = null \to \bar{0}; + \circ [1, sum \circ tl]$$

Find a closed form solution for sum

Theorem Given

$$f = p \to q; h \circ [i, f \circ j]$$

Then, for all $p\ q\ h\ i\ j$:

$$f = p \to q; \ldots; p \circ j^n \to Q^n(q); \ldots$$

where

$$Q^n(q) = /h \circ [i, i \circ j, \ldots, i \circ j^{n-1}, q \circ j^n]$$

$$sum = null \to \bar{0}; \ldots; null \circ tl^n \to Q^n(\bar{0}); \ldots$$

$$Q^n(\bar{0}) = /+ \circ [1, 1 \circ tl, \ldots 1 \circ tl^{n-1}, \bar{0} \circ tl^n]$$

$$= /+ \circ [1, 2, \ldots, n, \bar{0}]$$

$$= /+ \circ [1, 2, \ldots, n]$$

$$= /+ \circ id = /+$$

$$sum = /+$$

[EXPLAIN]

And now let's look at one final example.

SLIDE 17

Directly applicable results, example 2

Problem: Given

$$fact = eq0 \to \bar{1}\,;\ * \circ [id, fact \circ sub1]$$

$$fa \circ [x, y] = eq0 \circ x \to y\,;\ fa \circ [sub1 \circ x,\ * \circ [y, x]]$$

prove that

$$fa : \langle x, 1 \rangle = fact : x$$

Theorem Given h assoc., unit u and

$$f = p \to q\,;\ h \circ [i,\ f \circ j]$$

then

$$f = f' \circ [id, \bar{u}]$$

where

$$f' \circ [x, y] = p \circ x \to h \circ [y,\ q \circ x]\,;\ f' \circ [j \circ x,\ h \circ [y,\ i \circ x]]$$

[EXPLAIN]

The point about these two examples is that each used a theorem that provides the solution to many other problems in addition to the ones we considered. These are the kind of theorems that we need to develop if we want programming to be an engineering discipline.

Let me close by saying that the evidence shows that we have been working with a fundamentally flawed notion of program for the last 30 years. So let me urge you not to put up with it any longer. Revolt.

Investigate other concepts of programs. Start thinking and programming on a higher level, even if it is hard at first.

Just be careful that any new concept you propose is not as complicated as the von Neumann concept. And be careful that it has a decent mathematics associated with it.

If we don't change our ways in programming, we'll remain like those 16th century astronomers who wouldn't listen to Copernicus [or anybody else], we'll just stay [redacted] mired in our favorite misconception like they did.

Thank you.