RJ 3994 (44852) 8/23/83
Computer Science

Research Report

THE COMING REVOLUTION IN COMPUTING

John Backus

IBM Research Laboratory
San Jose, California 95193

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

RJ 3994 (44852) 8/23/83
Computer Science

THE COMING REVOLUTION IN COMPUTING

John Backus

IBM Research Laboratory
San Jose, California

ABSTRACT

This paper is the text of a lecture given at MIT on May 5, 1983. It discusses the problems with programming and computer design. The following are some of the points it makes:

1) Programming is far too expensive and cuts off millions of potential users from direct access to computers. Computer design makes poor use of VLSI and makes computers hard to program. A revolution is required to correct these problems.

2) Our concept of what a program is determines how we program and how we design machines. We have stuck with the wrong concept for thirty years. Instead of having old-style programs that map stores into stores we can have new-style programs that map objects into objects -- the objects of interest that are stored in stores.

3) Old-style programs cannot be combined to build new programs unless they have a common storage plan; new-style ones can easily be combined to achieve the combination of their purposes.

4) Old-style programs are "object level" whereas new-style ones are "function level". Object level programs have a mathematics of objects, whereas function level ones have a mathematics of programs. If we are to make programming an order of magnitude simpler, we shall need to optimize to a degree now unheard of, and we shall need a powerful mathematics of programs to make that possible.

The last sections of the paper address some problems students will face if they pursue the proposed revolution: the difficulties of creative work in a new field, the conservatism of professors. The problems and probable reception of this revolution are compared with those of the "Fortran revolution" of 25 years ago.

Text of a lecture given at M.I.T. on May 5, 1983.

THE COMING REVOLUTION IN COMPUTING

John Backus
IBM Research Laboratory
San Jose, California

Why am I claiming, you might ask, that there is going to be a revolution in computing? By a revolution, I mean a radical change in some of our basic concepts about programs and about computers.

You might say that's preposterous; after all, computer science has developed a lot of sophisticated ideas over the last 25 years, and you might believe that all we need to do is to improve a few things here and there.

You might say, in short, that by now we really understand what computing is all about, that all we need is evolution not revolution. Right?

The revolution I speak of has already begun in a very small way, and since MIT is already an important center of its activity, I'm sure that at least some of you do not share this conservative viewpoint.

Well, in any case, I think the conservative view is wrong. Let me try to explain why.

Why is there going to be a revolution?

Because both programming and machine design are in a bad state. Neither is able to meet our pressing current needs; and neither has made much progress in the last 25 years.

Let's consider programming. It has two big problems: first, it is far too slow and expensive a process; and second, its difficulty cuts off the average user from control of the computer. And it is important to recognize that, using our current concepts, these two problems have barely improved over the last 25 years.

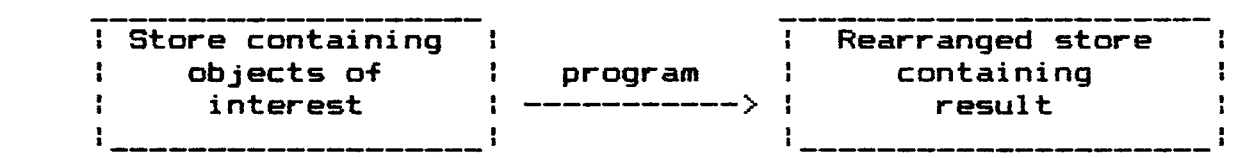
If anything, programming is becoming even less accessible to the non-expert as programming languages get more complex.

Next, let's consider the problem of computer design. Current computers are designed to operate only on individual words; they require complex operating systems. Serial computers are hard to program; and parallel computers are even harder. But their most serious problem is that they make poor use of the capabilities of VLSI circuits.

There are millions of people who will want to use the flood of ever cheaper computers being produced. If they are to do so really effectively, they must be able to write programs themselves.

The Old and New Concepts of 'Program'

Old concept: programs in [stores --> stores]



example,

if add is the program c := a+b,

and sqrt is the program e:= squareroot(d),

then add;sqrt is not a program for squareroot(a+b)

New concept: programs in [objects --> objects]

example,

(sqrt ∘ +):<4,5> = 3

program object result

SLIDE 1

It indicates that the composition of conventional programs for addition and square root will not give a program for the square root of the sum, unless they have been planned together, that is, unless the square root program happens to take its input from the result cell of the addition program.

As you can see on the slide, the addition program leaves its result in cell c, but the square root program takes its input from cell d. So the program formed by composing them is meaningless.

On the other hand, the composition of the corresponding functions will, of course, give the desired function. Here we see that the addition function maps the pair 4,5 into 9, which the square root function then maps into 3.

So composing the two old-style programs gives nonsense, but composing the new-style ones gives the composition of their purposes, the square root of the sum.

In this example we also see that, for programs to be functions, the space of objects must contain compound objects, such as the pair 4,5.

[slide off]

So I submit that one insight we shall need in the coming revolution is that we have been working with the wrong concept of a program for over 30 years. We must learn that there is a much more powerful concept, which is to regard programs as functions on the space of the objects we're interested in, not on some artificial space of 'stores' that merely contain those objects.

Adopting this new concept of a program will cause us a lot of anguish, in spite of its many advantages, because it means dropping the store as the most fundamental structure in computer science. And without stores as the basis of programs, we lose the means to use names in place of the objects they name. In short, we lose a facility that is the basis of our entire use of language, a facility that is an integral part of all mathematics.

Our entire training teaches us to use names in place of the things they denote. But the functional view of programs will severely restrict our ability to do this.

It will take a lot of courage and practice to get used to the new world that the new concept of programs will create. But I believe it will be well worth the effort. After all, we have to do something to get out of the mess we're in. And we can't expect that it will take place without any pain or effort.

In the old concept of programs, there was always a necessary division between the world of expressions, which is a well-behaved world, and the world of statements and programs, which is not well-behaved.

In the new concept of programs as functions there is only the world of programs, because programs are just expressions. Without the division of program elements into two worlds there is greater freedom for interaction between them.

But these program-expressions differ from the expressions in conventional programs in one very important respect: they are function level expressions, whereas ordinary expressions are object level ones.

This means that the operations in function level expressions map functions into functions, whereas the operations in object level expressions map objects into objects.

Here is a slide that illustrates the difference between object level and function level programs. [slide 2](see page 5a).

We are given two programs for defining the program average, that is, we are given 'plus' and 'half'. In the object level program, we immediately leave the level of these given programs and descend to the object level to construct objects. To do this, we introduce the objects x and y, then, by applying the programs we are given, we build first the sum, and from that we build the average. Both are objects.

Now, to get the program 'average' itself, we must move from the object level back up to the function level. That is, we move from the result-object, half(x+y), to the program 'average' by abstracting the object variables x and y.

In the function level program, we start with the same given programs and construct the program 'average' completely at the function level. That is, by applying the program-forming operation composition to the two programs 'half' and 'plus', we get the program 'average'. In doing so, objects are not mentioned.

We can see that the function level program contains no object names or variables, whereas the object level program completely depends on them.

We can also see that if the function level program is applied to the pair 3,7 that it will form first their sum and then their average, 5. We could easily prove that applying average to the pair x,y gives the quantity half(x+y), for all x and y.

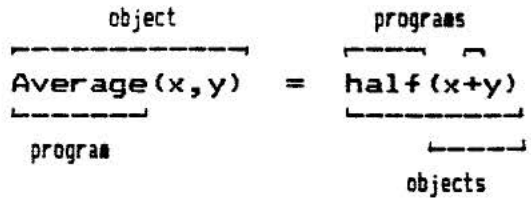
Notice that in the object level program we are concerned with objects and operations on objects, whereas in the function level program we are concerned with functions and operations on functions. In the function level approach, it is important to regard composition as an operation on programs, not as punctuation that separates them, as the usual semicolon notation suggests.

We can use many different program-forming operations to build function level programs, and we can find sets of program-forming operations that obey powerful algebraic laws. These laws then become the axioms of a mathematics of programs.

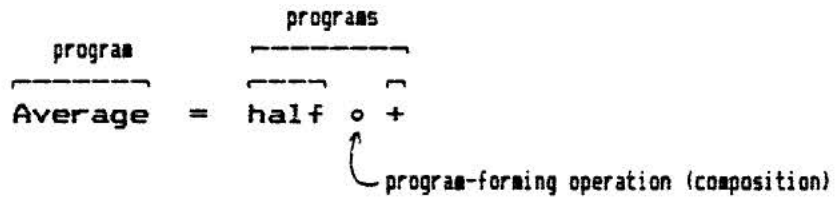
Object level & function level programs

Object level

For all objects x and y:



Function level



Program applied:

Average:<3,7> = (half ∘ +):<3,7>

= half:(+:<3,7>)

= half:10

= 5

SLIDE 2

The difference between the two kinds of programs means that the mathematics of object level programs will primarily concern objects and object-forming operations whereas the mathematics of function-level programs will concern programs and program-forming operations.

Just as you can prove general theorems about numbers at the object level, in the same way you can prove general theorems about programs at the function level. But just as the theorems about numbers require that plus and times obey certain algebraic laws, so will theorems about programs require that the program-forming operations obey algebraic laws.

So, if we want to prove things about programs, if we want to transform them and solve equations for them, then we need a mathematics of programs, not of objects. Therefore we need function level programs.

[slide off]

In view of all this, I believe that a second basic insight that we shall need for the coming revolution is that we have been thinking and programming at the wrong level. We have constantly worked at the object level, without being aware of the great advantages of thinking and working at the function level.

The principal advantage of doing so is that function level programming carries with it powerful mathematical properties. And these properties will enable us to transform and optimize programs, and to prove program correctness to a degree that goes far beyond what can be done with the old object level programs, since their mathematics is concerned with objects, not programs.

In the coming revolution we shall need to optimize programs to a degree that is not possible at present. Because if the casual user writes programs, he must not be concerned with efficiency. And we must be able to turn his worst efforts into efficient programs. And to do that we shall need very powerful mathematical tools.

One way of seeing the advantages of moving from the object level to the function level is to look at the advances in our work on data types. Originally we thought that data types concerned the structure of data objects. In recent years we have come to understand that data types are best characterized, not by the structure of objects, but by the structure of the operations on those objects, and that structure is determined by the algebraic axioms the operations satisfy.

In the area of programs, we are now at the earlier stage of understanding, and we must change that understanding in a similar

way. We are at present concerned with programs and their structure, as determined by the arrangement of semicolons, if-then- else's, and while-do's. The function level viewpoint will move our attention from the structure of programs to the structure of the program-forming operations, as that structure is given by various algebraic laws.

It is interesting to observe that, of the two main insights I'm suggesting are essential to the revolution, that the first one, the revision of our concept of programs, is already embodied in pure Lisp, since its programs map objects into objects. But the second insight, that is, moving to the function level, is found in only a rudimentary form in any present language.

Lisp in particular is an object level language, in practice. This is borne out by the fact that Lisp compilers do not improve the performance of function level Lisp programs. They do not, because Lisp programmers do not write such programs, and therefore these compilers are not designed to deal with function level programs. (It is worth noting that one can write function level programs in Lisp.)

So far, we have talked about why a revolution in computing must take place and about two particular insights that may be important for its success. We might now ask: what consequences can we expect from such radical changes in our basic concepts?

In the area of machine design we are already seeing a considerable interest in new designs. And MIT is certainly a primary center for such interests. But as the revolution progresses and the insights I have described, as well as others, become better understood, I believe we may see even more profound changes in the thinking of computer architects, changes that go beyond those we are already seeing.

In addition to machines that can do many things in parallel, we may see a new generation of very simple serial machines that are designed to execute function level programs with better efficiency than von Neumann machines can execute them.

We may see machines that can operate, at the machine level, on large structured objects. The inability of current machines to do this is one of their biggest shortcomings. We may see a new concept of a "store" arise, so that rearrangement operations on such large objects can take place directly in the store. We may see other new kinds of stores develop that go far beyond the primitive idea of a box that delivers a single word upon receiving an address.

In the area of programming we should see a new concept of program, one that focuses on building programs, not objects, one that leads to thinking at the function level instead of at the object level. We should see an upsurge of interest in the operations for building programs and in the mathematical properties of these program-forming operations.

We should see the development of a mathematics of programs, with a large catalog of general theorems that are useful for proving the correctness of many programs and that are also useful for optimizing programs to a degree that is not now possible.

We should also see much more involvement of mathematicians in computer science, since one consequence of the revolution should be the development of important theorems that would be of immediate practical value. This would contrast with today's typical results, which are often beautiful and intellectually important, but are useful only in setting bounds on what is possible. Instead, we need theorems like those in engineering that are constantly used in practice.

If things work out well, we may even see some deep general theorems about programs that are now the province of pure mathematics, theorems that assure us that two completely different algorithms actually accomplish the same thing. Such theorems might allow us, for example, to have optimizers that could turn a fourier transform program into a <u>fast</u> fourier transform.

Whatever new kinds of computers the revolution brings, we will see the heavy use of mathematical techniques to optimize programs for them, whether they are parallel or serial, exotic or von Neumann.

And we should see theorem provers that will use theorems about existing programs to prove the correctness of a new program built from them. In this way, we can hope to avoid proofs of large programs, where each one must start completely from the beginning.

It is pleasant to speculate about all the wonderful things that might happen if the revolution I speak of comes about, but there are many difficulties to be overcome before that can happen. Most of all, a great rebirth of creative effort will be required, similar to the one that brought us, long ago, the beginnings of computer science.

Much of the revolution will be created by today's students. So I would like to address to the students who are here, some thoughts about some of the fun and some of the difficulties they will face if they want to be part of this.

First of all let me say I envy you; instead of having to absorb a lot of knowledge and then trying to find some new little niche in which to make a contribution, instead of that you have a vast, mostly unexplored territory stretching out in front of you. There are dozens of good problems that can be seen right now and hundreds yet to be identified. Rather than textbooks full of theory and practice, you will have only a few simple guideposts in this new territory.

Some of the difficulty and the excitement will be in overcoming a number of obstacles. First of all, unless you are lucky, you may get precious little encouragement. Even though MIT is a leader in this revolution, some of you may have professors whose vested interest in their knowledge of computer science will discourage them from being part of a revolution that may discard a large part of that knowledge.

For example, today people are making careers out of describing languages like Ada. But after the revolution, what will these people do?

So, unless you are lucky and find a professor who is already working in this new area, you will have to work hard to show him -- or her -- that you can do something worthwhile in such a new, unproved area, because he will find it difficult to help you, unless he takes up this work himself. But perhaps you can even convince him to do that.

And even if you have the most favorable conditions, you will face the problems that go with any really creative work. We are all more comfortable in familiar areas, using familiar tools to get results (that may be new and difficult) but that we feel confident of achieving. But real creative work is not often like that, despite the romantic glamour that the media associates with it.

They like to portray creative work as something you do walking along a beautiful beach, and suddenly lightening strikes and you have a wonderful, important idea. To say the least, this doesn't happen very often.

In reality, most people hate creative work because it consists mostly of hard work that leads to failure. It consists of struggling to have an idea, then working very hard to learn how to use it, and finally, after a long time, learning that it really just doesn't work. It consists of repeating this discouraging cycle over and over until finally one idea does work. And after that other successful ideas must be found to make a whole, complete scheme that comes together in a practical way.

So if you want to do creative work, you must be ready to fail, and fail a lot. But perhaps you have already discovered this for yourself.

But, if you like creative work in spite of all this, the coming revolution in computing will give you a lot of new opportunities. If you are not afraid of going into a new area with few tools, and few results, an area in which you will have to invent all these things, then you are in for an exciting time. Then you will have the opportunity to invent the concepts, the tools and the methods that future professors will study and write textbooks about.

You will have the opportunity to change the world of computing so that programming is very different but very much easier than it is today, making the computer far more accessible to the user. You will have the opportunity to change even the basic meanings of the word 'program' and the word 'computer'.

You might be saying by now, enough!, that all this is too far fetched, that it is an opium dream or some kind of delusion that this fellow Backus is suffering from. And perhaps you're right. But you can't be sure, either, can you?.

To help you decide, it might be interesting at this point to compare the beginning of this revolution in computing with an earlier, smaller and less dramatic revolution, one that began almost thirty years ago. I'm referring to the change from assembly language to Fortran, to the beginning of what we like to call high-level languages.

The goal of this earlier revolution was the same as the main goal of today's. It was to make computers much more accessible to the user and to make programming much less expensive.

Since I took part in the first one, I thought it might be helpful to recall some of the difficulties we faced then. Recalling the past might help students and people just starting out in computing to understand the problems and attitudes they will face if they choose to take part in <u>today's</u> revolution in computing -- let me call it the functional revolution.

Although there are many similarities between the two revolutions, it is important to realize that the functional revolution is different in two very important ways from the Fortran revolution. First of all, it is going to be far more technically difficult. And second, it will face much more resistance.

Let me first compare the two revolutions with respect to technical and theoretical difficulties. These can be divided into two main areas. The first area concerns problems of the major concepts of programming and of language design. The second area concerns problems of implementation and optimization.

So first, let's compare Fortran and functional languages with respect to their concepts and language design.

In the case of Fortran, there were few really new basic concepts, although many new 'features' and conveniences were introduced in the language.

The development of the Fortran language itself was a very offhand affair. We just used basic ideas from mathematical notation and from assembly language and then designed the language by trial and error. The Fortran language, like most languages, has no mathematical properties, so we did not have to be concerned with mathematical issues. It took us about six months to design the language.

On the other hand, developing the underlying concepts of the functional revolution is orders of magnitude harder than it was for Fortran. The work on some of the basic concepts I discussed earlier has taken many years. And much more remains to be done.

Unlike the Fortran case, the problems of designing functional languages presents a number of technical and theoretical problems. This is because these languages have important mathematical properties, as I have mentioned, and therefore their design requires a lot of thought about how to obtain these properties in the first place, and then to make sure that new features don't destroy them.

For example, suppose you want to include infinite sequences as objects, and you want to introduce some partially ordered domain for defining them, then you must be very careful about the definition of your primitive functions on this domain, to make sure that they remain continuous functions, continuous that is, in the sense of Dana Scott.

So much for concepts and language issues.

Now let's compare the implementation and optimization problems for the Fortran and functional cases.

In the case of Fortran the main problem was optimization. It may be hard to realize now that implementing Fortran was not technically easy. We had to prove that we could optimize programs so that they would run as fast as programs programmed in assembly language.

Doing that involved a lot of creative work by my friends on the project. And they were really amazingly successful. They created a lot of the basic concepts and methods of today's compiler technology.

They were so successful that their 1957 compiler turned out to optimize better than any later compilers for the next 20 years! It wasn't until 1978 that a compiler was developed that could optimize as well, overall, as the Fortran I compiler. (I know this is hard to believe, but I have this on the authority of Fran

Allen who is one of the leading experts on optimization.)

It is interesting that one of the main technical problems of the functional revolution is also optimization, just as it was for Fortran. In our case, it is sort of amusing that we will have to prove that functional programs can run as fast as Fortran programs! And we can't wait until parallel machines come along to make that a lot easier.

But Fortran programs were much closer to machine programs than functional ones are. Therefore the amount of optimizing needed for functional programs will be much greater than it was then, and it will be very different in character.

Thus, perhaps the hardest thing the functional revolution has to prove is that the mathematical properties of its programs are powerful enough to perform the enormous feats of optimization that will be needed.

So, that ends my comparison of implementation problems.

Now let's turn to the question of how the computing world reacted to the Fortran revolution, and then let's speculate a bit about how we can expect it to react to the functional revolution.

In the case of Fortran, the world of computing, for the most part, liked the language, and had no real disputes with us over concepts. They were just very skeptical. People in the establishment, those with the most training, were the most skeptical. And they were mostly skeptical about our claims that Fortran programs would run as fast as assembly language programs.

In a preliminary report on the language that we released in November, 1954, we even suggested that Fortran would use such sophiticated techniques, that it would often produce programs that were even faster than manually coded ones. I have a copy of a wonderful letter from Dr W W Leutert in response to our report. He was then chief of the Army Ballistics Computing Lab, and very much a part of the computing establishment.

His letter, dated one week after our report in 1954, in response to our statement that we would produce faster programs than manual ones, says, quote, I believe the opposite of this statement is true, unquote. Elsewhere Leutert very much doubts that we would turn out programs that ran even half as fast.

It is amusing to read this letter today, since we know now that our early claims about Fortran were basically correct. But it would be unfair to make fun of Dr Leutert's judgment at the time, because almost everyone felt the same way, both in and out of the establishment. They felt that way because many other groups had

made big claims for their systems that had not turned out to be true.

After Fortran was released and people found our claims to be basically true, it rapidly gained wide acceptance. It did so because it was a proposition you just couldn't refuse. It provided too much clarity of expression, and too much economy over any existing way of programming, for you not to use it.

Essentially that is what the functional revolution must do, too. It must provide a proposition that is even harder to refuse, because it will be much harder for users to give up their old ways and take up the new ones, than it was for Fortran users.

And we, too, will have to overcome the skepticism of all those who have been told too often that language X or language Y was going to really simplify programming.

All Fortran had to do was to prove the skeptics wrong about the speed of its programs. Users were not required to change their way of thinking. They did not have to give up practices they had used all their working life and learn a completely new way of doing things.

For the functional revolution to prove that <u>it</u> is a proposition you can't refuse, it has to demonstrate that relearning how to think about and write programs pays off in a very big way.

It must show that you can write programs about an order of magnitude faster and cheaper. That programs are reusable in building new ones. That programs can be proved correct with very little help from the user. And, finally, that programs will run as fast as they ever did on old machines and even faster on new, parallel ones.

These are some of the goals of the revolution. There are lots of people who will laugh if you say you're working on them. Lots of people will say it's impossible. Perhaps it is. But we won't find out unless we try.

And if we do show that it's possible, or even just part of it is possible, those of you who had the courage to work on it, and ignore the skeptics, will be glad that you were part of the challenge and excitement of the functional revolution.

Thank you for your attention and good luck.