An interview with

Karen Frenkel: [redacted]

Please describe, in the simplest terms possible, the essence of Logical for Computable Functions (LCF).

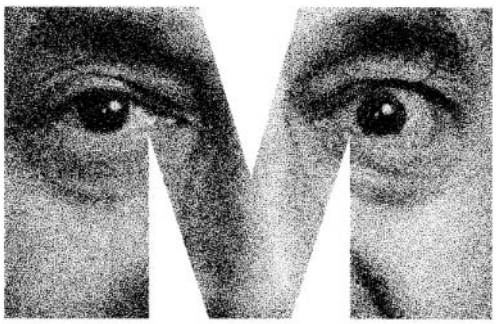
Robin Milner: [redacted]

I came quite late to research. I taught for five years in City University in London, and before that I'd worked for three years for Ferranti Computers, where I did some programming. While I was teaching at City University, I became interested in artificial intelligence, but also in what programming means. I also got interested in mathematical logic, but all of these were separate threads.

KF: *That would have been in the late 1960s?*

RM: Yes, from 1960 to 1963, I was at Ferranti, and from 1963 to 1968, I was at City University. Then I got a chance to do a full-time research post with David Cooper who was head of the computer science department at Swansea College in the University of Wales. Under the stimulus of David, I worked on understanding programming and program verification. I wrote myself a small theorem-proving program. At that time, there was a tremendous research effort in mathematical theorem proving, trying to do automatically what mathematicians have done with deep thought over the centuries.

Another thread was stimulated by the work of Michael Paterson, who had written a wonderful thesis on program schematology—part of his joint work with David Luckham and David Park. That thread was the study of the shapes of different programs as opposed to their content, or put another way, studying the general shape of their evaluation structure as opposed to the actual numbers that they were processing. The work of Paterson and Cooper made me want to understand how you could verify computer programs. Pioneering work had recently been done by Bob Floyd at Stanford and by Tony Hoare in Britain.

Robin  ilner

We use the term "metalanguage" for a language that talks about other languages. That's why ML came into existence.

My first attempt was to take a simple program—many people were doing similar work at that time—and extract from it what you would need to prove that it was working properly. Those were called verification conditions. They were fed into an automatic theorem-proving program that would then prove that the original program was working properly. This ran into the sand because it is far too difficult for a computer to do. Human intelligence must accompany the business of verifying that programs, which are human constructions, actually work. So, I wasn't sure how human assistance could be brought to bear on this in a rigorous way.

Then I learned of Dana Scott, who, with Christopher Strachey, went into the foundations of programming languages. I not only had some discussion with Strachey about programming languages, I also listened to Scott's lectures while he was visiting Oxford University and read his writings. It was a very exciting time, because together they began to work out the mathematical meaning of programming languages.

I hoped to start from the very rock bottom with the mathematical meaning of languages, and use that as a firm foundation for reasoning about computer programs that anyone might have written. I wondered how one might do this. And Scott actually gave the hint by building, so to speak, some of his mathematical models into a formal logical system that could be used as a basis for computer assistance. When you want to get a computer to assist with something, it has to be formal; it has to be exact. The way it's going to help must be absolutely formal. Scott didn't actually call it the Logic for Computable Functions. I called it that. But he embedded some of his mathematical understanding in this formal, logical system.

I was very excited about this. I went to Stanford University in 1971, and got the chance to work with John McCarthy in his artificial intelligence lab. When I arrived, it was quite amusing because they wanted to do this kind of work, but they were also very keen to do some practical work. There was a lot of work on foundations of computer science, or meanings of programming languages, along with the artificial intelligence. But we all need, in order to satisfy our funding agencies, something that actually runs and buzzes and whistles, something that actually convinces them that not only are we trying to understand computers but [that] we're also using them; you have to make your work concrete in order to get credibility. I remember a meeting in McCarthy's AI lab where we were all wondering what implementation should be done next to get machines to help in understanding our own software activities. I was the new boy, and I actually wanted to do the next implementation because I realized that Scott's work was just waiting to go into a computer program that would then help us to reason about other computer programs. So I said, "I'll implement Scott's logic." That was exciting because Scott had created something more general than most people realized.

We worked on proving that a compiler was correct. If you write a program in a high-level programming language, it's then converted by the compiler into a low-level program. That shouldn't change the meaning. If it does, that's a disaster. So one of the first things you must do is show that the translation is correct. McCarthy and his students had done some initial case studies on this. And we proved with LCF the correctness of a more complex compiler, which worked with a richer language. I hadn't realized that was going to be possible. To me, it was part of the general activity of eventually reaching something that ordinary programmers could use to check out their own programs.

Designing ML

KF: *Did you try your proof techniques on certain languages already in existence when you were at Stanford?*

RM: Yes, and I'd also tried that at Swansea, but I rapidly began trying to prove properties of whole languages, so I wasn't focused on a particular language. McCarthy, (Allan) Newell, and (Herb) Simon's language, LISP, was a wonderful tool with which to write all the software that would eventually do this verifi-

cation. So McCarthy's laboratory and McCarthy himself were a wonderful liaison for me because LISP processing was a wonderful vehicle for what you might call nonnumerical programming—programming which realizes different forms of thinking than numerical mathematics. That was a vehicle for doing all my work. In fact, LISP led to the design of Metalanguage (ML).

I left Stanford after two years, having had reasonable success with this reasoning tool. But it was very rigid. That is, the way I could interact with this machine in helping me to reason, was that I could ask it to do certain formal transformations, and it would do them correctly. By the way, the real pioneer of machine-assisted reasoning was deBruijn in Holland who invented his Automath system before this; I didn't know about it at the time.

But the way that I wanted LCF to help in the reasoning business was this: If you've got a machine helping you, you want to not only get it to check what you're doing, but to be able to communicate to it certain general strategies for reasoning. I needed a medium by which I could communicate to the machine certain general procedures for reasoning that it would later invoke, at my behest, on particular problems. I would not have to lead it through the elementary steps every time. I wanted to be able to give it larger and larger chunks of reasoning power, built up from the smaller chunks. So I would have to have a language by which I could communicate to the machine these tactics or strategies.

Then, you come back to the problem of building houses on sand because the more languages you bring into your process, the more possibilities you have for appearing to talk sense but are actually talking nonsense. So the language we needed to express the reasoning capabilities had to be very robust. We use the term "metalanguage," for a language that talks about other languages. That's why ML came into existence. It was the metalanguage with which we would interact with the machine in doing verification. It had to have what we call a rigorous type structure because that's the way programming languages avoid talking certain kinds of nonsense. But it also had to be very flexible because it was actually going to be used, and I didn't want to design a language that would slow me down. It had to have certain features that were at the frontier of programming language design, such as higher-order functions of the greatest possible power, and also side effects and exceptions. An exception is just a way of getting out of something that you shouldn't be doing because it's not working. Since strategies don't work every so often, you use an exception to say, "Wrap this up. I'm going to try something else." In a programming language, an exception is absolutely vital. And it was vital for this particular application.

All of this directed the design of ML, which occurred in Edinburgh with other colleagues from 1974 onward.

KF: *That was a 12-year project?*

RM: Yes, ML began just as a vehicle for communicating proof strategies within the LCF work. Malcolm Newey and Lockwood Morris, both of whom I had met at Stanford, and later Christopher Wadsworth and Mike Gordon came to work with me, and we created this language and some mathematical understanding for it.

The LCF system at Edinburgh then became the language ML with some particular reasoning power expressed within that language. Gradually, the language became more and more important.

A Longtime Collaboration

KF: *What was it like to collaborate with people on developing a language over the course of more than a decade? How did you work together?*

RM: That was a wonderful experience. It came together in ways that could not be predicted or planned. When I got to Edinburgh, I had a research project funded by the Science and Engineering Research Council (the British equivalent of the NSF). The first to join me were Newey and Morris. We weren't quite clear what the language should be, and we tossed ideas around among ourselves. I remember Morris wrote the first compiler for ML and left it behind in Edinburgh six weeks after he'd finished it. Nobody ever found any mistakes in it. It was the first implementation of ML. And Newey and I worked on other parts of the implementation as well.

When Wadsworth and Gordon came, we developed the language more carefully so that it could serve as a basis for really big reasoning projects. At that point, the project divided; the reasoning work went on along one line, and language development of ML itself went along another. ML went from being a special language for this particular task to a general language. And that happened in a beautiful, but unplanned way. One now-famous contributor was an Italian graduate student, Luca Cardelli, who wanted a language for his Ph.D. work, so he implemented an extension of ML. Then somebody else discovered this was a good language to teach to students. It then began a life as a general-purpose language because we started teaching it to second-year undergraduate students. It turned out to be a way of learning to program.

KF: *Was that one of the surprises?*

RM: Yes, one of the reasons it turned out to be general-purpose was because the demands of the application—the LCF work—were so strong that if a language could do all of that, then it would also do a lot of other things as well. That happened in an uncontrolled way for a while. People used different dialects either because they liked experimenting in language design, or because they wanted it for teaching more clearly, I suppose.

About 1983, on suggestion of Bernard Sufrin at Oxford, I felt we ought to pull the threads together to see if there was a bigger language that comprised all the ideas people had been tossing about. I produced a proposal for standardizing this language. We began to have very intense discussions because language design isn't easy, and people disagree about it. But we acquired a group of about 15 people who worked via email. We had a distributed effort involving David MacQueen at Bell Labs. So we began to play with the design and tried to firm it up.

Then another splendid thing hap-

The great challenge and greatest excitement was that we were always interacting with three things: the design of the language, its implementation, and the formal definition itself.

pened. MacQueen invented a new upper level to the language that made it more appropriate for large programming exercises. This enabled you to write large modular programs and assist "programming in the large." MacQueen had been in Edinburgh working with Rod Burstall, who had a mathematical project which actually led to MacQueen's idea of modules of ML. So some of the mathematical, or the theoretical research, fed into the design of the language in that way.

For the next four to five years we were standardizing this language. It went through design after design. In 1989 we still didn't all completely agree, but those charged with writing the formal definition of the language published it with MIT Press.

KF: *What was the greatest challenge in those 12 years?*

RM: In terms of the language design, for me it was creating the formal definition, because the design had to be enshrined in an absolutely rigorous definition.

KF: *Enshrined?*

RM: Yes, it had to be expressed completely rigorously. Not many languages have had that completely rigorous definition. Others have had it in part. Our aim was to have not only a definition that was completely rigorous, but was also quite small. The language was supposed to be powerful but so harmonious and so well structured that it didn't take many pages to write down the definition of everything you could do in it. Eventually it took 100 pages, which is perhaps an order of magnitude smaller than for some powerful languages like ADA, for which the formal definition is not easy.

For me, the greatest challenge and the greatest excitement was that we were always interacting with three things: the design of the language, its implementation (because it always was being implemented experimentally), and the formal definition itself. You would design something, and then you would find out that you could implement it well, perhaps, but that you couldn't write down the formal definition very clearly because the formal definition showed there was something missing in the design. So you'd go back to the design. Or you might go back to the design because something was not implemented very well.

Concurrency and Parallelism

KF: *Let's move on to Calculus for Communicating Systems (CCS).*

RM: The development of CCS also went on for a long time. As Stanford, I got interested in trying to understand concurrent computing and parallel computing programs. I tried to express the meaning of concurrent computing in terms that had been used for other programming languages that were sequential. I found it wasn't easy, and I felt that concurrency needed a conceptual framework, which we did not have. And I didn't know what it should be.

Of course, I didn't know about his work, but Carl-Adam Petri had already pursued such a goal. But my motivation was actually to show how you could build concurrent systems—how you could create a conceptual framework in which you could compose and synthesize larger and larger concurrent systems from smaller ones and still retain a handle on what it all means. That's why I eventually approached the algebraic method. Algebra is about combining things to make other things and the laws that govern the ways you stick things together. In multiplication, $A \times B$ is a more complicated thing than A or B. Multiplication has certain laws. That was exactly the same as parallel composition. Two programs, P in parallel with Q, give you a more complicated program, and that parallel composition obeys some algebraic laws. So CCS was an attempt to algebraicize the primitives of concurrency.

Concurrency Theory and Hardware

KF: *To what extent does the hardware affect the theory of concurrency? Doesn't it matter the way the processors are linked and whether the machine is fine- or coarse-grained?*

RM: Yes, that's a big question because there are two things you might be trying to do when you're studying parallelism. You might be studying the meaning of a parallel programming language that is going to run on some hardware. Or you might be trying to describe a concurrent sys-

tem that is a piece of hardware. What I was really aiming at was not just a theory of programming, but a way to describe concurrent activity of any kind. And the machine is just one example of something that has highly concurrent activity. So there must be a theory of concurrency that can describe the machines on which you might run a parallel programming language. You must be able to apply your theory at these different levels. Parallel programming is just one kind of concurrent activity.

It does matter from the point of view of efficiency, and even feasibility, on what hardware you will run a parallel language. Some languages will be much easier to run on some hardware than others. But that very match, that very question, "What is it about a machine that makes it possible or impossible to run a certain programming language?" should be part of the theory of computer science. Computer science should be able to address that machine-language link, just as it addresses the language itself.

The theory should also address interaction between humans and interaction in any organization. The programming language is a small part of concurrent activity. So one is aiming for a theory that is actually broader than computer science.

KF: *Do you think the interrelationship between a parallel language and a parallel machine is more problematic than that of sequential languages and sequential machines?*

RM: Yes, I do. I don't think it's more problematic just because it's one step up. It's more problematic because it has a larger number of degrees of freedom. The only limitation in trying to understand concurrency in the way that computer scientists do, is that we're trying to understand it discretely rather than continuously. On the whole, we're interested in discrete events in computing, rather than continuous ones. At least that's been the fashion for the last three decades.

There are also so many different ways by which you might tackle the problem. Sequential computing was a wonderful beginning for us. The normal way you would write a program from the very beginning would always be sequential, and that gave us an inroad into understanding computing.

But back to concurrency, I needed a new conceptual framework. When I went to Denmark in 1979 and 1980, I gave some lectures in which it came together rather well. I wrote the book on CCS (*A Calculus of Communicating Systems*), which was essentially a ramification of 10 lectures I gave there. For the next five or so years, I just studied CCS. Between 1980 and 1985 I began to develop the mathematics to see whether you could find some abstract mathematical understanding underneath it.

CCS was obviously not perfect, but it existed as a concrete language and method of description. It had an algebraic theory, and you could go underneath it as I began to see whether it could be explained more smoothly, more abstractly mathematically. But then it could also be used in design studies. That is, it could be used as a way of understanding communications protocols, how traffic lights work for example, or simple concurrent systems in practice.

CCS was used to explain the specification language LOTOS, which is a way of describing communications protocols and which is very widely used now. CCS was at a level that enabled you to dig below for semantic understanding, but you could also build on top of it to get more useful things in the field.

During the 1980s, Tony Hoare's work was going along in parallel. And he had similar ideas to mine. He invented the language CSP before I did the algebraic work on CCS. Later, he pursued the theory of CSP in a way that was complementary to my algebraic theory. So the algebraic field became very rich. His ideas were very different and very important. Then there began a search for a general framework that encompassed his ideas and those in CCS and other process algebras, notably those of Jan Bergstra and Jan Willem Klop in Holland on ACP. And that, in turn, was used on a lot of applications.

We've been looking at these different approaches for many years, trying to subsume them into a whole. We sometimes succeed, and we sometimes fail. Actually, researchers in these three algebraic endeavors have been collaborating lately to iron out some differences. But it keeps exploding because no sooner do you try to bring them together than you discover that there are extra things that you want to add. In the beginning, we didn't talk about real time or probability. We wanted to attach probabilities to the possibilities that are expressed in these process algebras.

One line of development that links the algebraic concurrency endeavor and the already existing and very fruitful work of Petri on Petri nets has become rather promising and yet difficult because the conceptual basis in the two cases doesn't line up perfectly. So the theory of concurrency is enriched by having these different approaches and then looking for what it takes to commonize them, to actually find out that they're not contradictory and that they're about different aspects of essentially the same problem. This field has become enormously rich in the last decade.

Object-Oriented Programming

KF: *There are three characterizations of programming methods: imperative or structured, functional, which is what you started out doing, and object oriented. Can you comment on the strong trend toward object-oriented programming in the U.S.?*

RM: Object-oriented programming is a wonderful example of how fruitful things don't happen very precisely. That is, the programming community has come up with this tantalizing, powerful, and productive way of thinking that is obviously with us to stay because it gives people just what they want. But at the same time it's very difficult to understand mathematically and semantically. People at the forefront of object-oriented programming are aware that it's important and powerful. And they want to deepen semantic understanding of it.

It's a challenge to somebody who wants to develop a theory to make sure it will actually contribute to that understanding. Different computer science theories are needed to explain object-oriented programming because it is partly about concur-

Computer science is not only a study of a basic theory, and it is not just the business of making things happen. It's actually a study of how things happen.

rency. It naturally talks about objects that coexist and must therefore be allowed to behave simultaneously. That's parallelism or concurrency.

It's also about the classification of objects: the type structure, how objects inherit methods from the class to which they belong. If you define new subclasses, then you can add new methods. An object's methods determine how it may be used. You want to insert some fire walls into a programming language design so people will stop trying to subject an object to operations that are meaningless for that object. That's what type structuring is all about.

One theory that is contributing a lot to object-oriented programming is the development of a type structure. That part does not primarily have to do with concurrency. On the other hand, the dynamic part, or the behavioral aspect of object-oriented programming, particularly concurrent object-oriented programming, is precisely what we should try to understand from the point of view of concurrency because concurrency is about independent objects. These may work together, cooperate, or confuse one another. So you have to be able to explain object-oriented programming of the concurrent kind in terms of a basic concurrency theory. Otherwise that theory isn't any good.

There have been some very exciting initiatives to create that understanding on the basis of theories we have currently. Carl Hewitt comes in here because his notion of Actors, which he invented 20 years ago, was a great stimulus in providing the conceptual framework for object-oriented programming. We need to underpin both object-oriented programming and the Actors model with something more formal or more basic. That's where may process algebra work (with Joachim Parrow and David Walker) is going now. In my lecture I discuss the dynamic behavior of objects in object-oriented programming. Very interesting work by Japan's Kohei Honda and Mario Tokoro bridges between our process algebra work and concurrent object-oriented programming. It shows how one can be adapted or trained to explain the other. That's a very active field.

My present concurrency research is in developing a mathematical theory of this model, constantly testing it (either myself or asking others to) in various, more practical frameworks, like object-oriented programming, logic programming, even functional or imperative programming, to make sure it can underlie all of these things. If it can't, then it's the wrong approach.

I'm not saying everybody wants to think at this low level. But if you're ever going to run a functional program on a big concurrent machine while running an object-oriented program on the same machine, and even communicate between them, then you'd have to have some common theoretical framework in which they can both be explained. Otherwise, it's awful. We have total incomprehension. We do have these heterogeneous systems with one part understood in terms of functional programming or some different model, and other parts in object oriented. But you must be able to underpin them all. Reliability really is terribly important.

The theory is still not as clean as it should be. There is a basic theory waiting to be revealed more clearly. It has elements of Petri net theory; it has elements of the important notions of reference and naming.

This theory is guided—it's not wandering along in a trackless fashion. It's being submitted to the mathematical test all the time: Can we really work with it? Is it tractable mathematically? Does it really underpin all of the different things that people are doing in practice? If it tries to escape from these criteria then we shall notice. And then, it should cease to be interesting.

KF: *The title of a paper you wrote in 1987, "Is Computing an Experimental Science?" raises a very interesting question. What prompted you to write it at the time?*

RM: It resulted from ideas that had been around for a long time coming together with a particular practical need. We had good funding from our Science and Engineering Council. In the middle of the 1980s, the Alvey Program was supposed to stimulate industry and join it onto

academic research. We feared the effort would be drained from the theoretical part of the research and that funding would be devoted to shorter-term activities. These are highly necessary, but we didn't want that to happen to the detriment of theoretical work. The trouble with computer science is that it's so applicable you can be blamed if you're not doing immediately applicable research. There is such a demand for application research that if you are doing something with longer-term implications, you can be criticized for not doing the most urgent thing. That's a great danger, because computer science is so broad that if it doesn't have a basic theory, we will be lost. There's so much going on, so much computing, so much communications. How could it be that this doesn't have a theory? It has to go alongside the practice.

Therefore, the more you respond to shorter term, however necessary and important the demands, the more you run the risk that you're starving the basic science, which should be growing up at the same time. We asked the Alvey Directorate to remember that the basic research program is pretty strong in Britain and argued it should not be a casualty of this new and exciting effort. We asked them to recognize us as a laboratory that would link between the theoretical research and industrial practice. They responded generously.

My colleagues and I decided to make the case that the relationship between us and the Alvey initiative should be bidirectional. You don't just take things out of academia and apply them. You use the industrial experience as a guide. That's the experiment. That's what computer science *is* at the moment—one large experiment. And it's a very uncontrolled experiment.

So the questions are, "Is that absolute nonsense? Is it an experiment? Or is that just a crazy thing to say?" Some will say it's crazy—all that computer science is is a bag of tricks. You use computers, and you use them in different ways, just as you use water for different purposes.

In the paper which I gave at the inauguration of our laboratory, I made the case that if you look deeply, you find less difference between computer science and other established sciences than you might expect. The experiments we do in computer science are not that much dirtier or fuzzier than some experiments in other sciences. Of course, physics is the queen of the sciences, and the experimental discipline is extremely sophisticated. We can't match physics in the refinement of our experimental discipline, but that isn't to say we're not an experimental discipline. When you test a piece of software to see whether it works, that's some kind of experiment. And in testing a piece of software, what are you testing at the same time? You're also testing the language in which it's written, and you're testing the formal theory that underlies that language. Why can't that be called experimental science? I suppose the way I say it seems defensive. Many people would say, "Yes, of course, it's an experiment." But enough people will have assumed there isn't a science. So it is actually worth underlining that I think it is experimental.

I claimed further in that paper that there are concepts from computer science that will influence mathematics. Most people learned about sets in mathematics when they were children. When we understand about processes as mathematical objects, school children in the second and third decade in the twenty-first century will actually think about processes in the same abstract way they now think about sets. That will be because of computer science. It's experimental in that you're drawing concepts out of practice. Obviously, you don't draw concepts out which don't inform the activity. Experimental science is a way of developing a conceptual framework by drawing out the concepts that actually do appear to underlie the practice, but testing them all the time.

Advice to Students

KF: *Do you have any advice that you would like to give to young people who are starting out in computer science about the best way to prepare themselves in computing?*

RM: One doesn't want to sound pompous. The best thing to do, whether you're of a theoretical or a practical bent, is to treat the subject as neither purely theoretical or purely practical. The worst thing you can do is to follow your bent, which would probably be on one of those sides, and ignore the other side. The whole richness of the subject comes from the interplay between practice and theory.

Many will pretty soon find themselves ignoring one of those components because they will naturally become very applications oriented or very basic-research oriented. But the longer we can keep the link between the theoretical frontier and the practical frontier, the better the whole thing will be. We should encourage the next generation to respect that link. If you don't respect that, you lose a whole degree of freedom in the interest of the subject.

What makes it interesting is that the link is there. Computer science is actually a study of things that happen. It's not only a study of a basic theory, and it is not just the business of making things happen. It's actually a study of how things happen. So the advice is: Don't lose the link! ◻

About the Author:
KAREN A. FRENKEL is senior editor, ACM Publications and Special Projects. **Author's Current Address:** ACM, 1515 Broadway, New York, NY 10036.