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-

Milner

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 × 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-