Is Computing an Experimental Science?

Robin Milner

Text of the Inaugural Lecture of the Laboratory for Foundations of Computer Science[1]

We are beginning an ambitious programme of research into the Foundations of Computer Science. This isn't to say that we are beginning to study the theory of computation; this has been going on fruitfully for many years. Nevertheless the particular programme which we have put forward is a new kind of exercise. What makes it new is a central commitment to a double thesis: that the design of computing systems can only properly succeed if it is well grounded in theory, and that the important concepts in a theory can only emerge through protracted exposure to application.

If we take this commitment seriously, then we must make an organised attempt to unite the development of theory with its experimental application. That is why I chose a title for this lecture which, in its truly philosophical sense, I am not well equipped to answer. All the same, people involved with a developing subject ought to ask themselves difficult questions about what it is they are developing, even if the final answers are beyond them. For myself, I became committed to the inseparability of theory from experiment in the course of doing Computer Science, not through philosophical enquiry. But in this lecture I want to justify the commitment, by looking at some examples of how ideas and application have developed side-by-side. After all, however deep a philosophical study you wish to conduct, you must always support it by examples of scientific practice. By the time I have finished, I hope you will at least agree that the question "Is computing an experimental science?" is both fascinating and intricate. Whatever the answer, I hope even more that you will be convinced that our commitment to unite theory with application, and the research which we plan on the basis of this commitment, are both inevitable in the process of establishing firm foundations for computing as a science.

[1]This lecture, marking the inauguration of the Laboratory, was given on 17th January 1986 in the James Clerk Maxwell Building of the University of Edinburgh. The lecture was given by Professor Milner in his capacity as Director of the Laboratory before an audience of about 250 invited guests from U.K. Industrial, Academic and Governmental organisations

1 The nature of computation theory

There is a hard theoretical core to computation which few people would regard as experimental. This core is closely bound up with logic, and is the foundation stone of theoretical Computer Science; it is to do with what is computable, which is strongly connected to what is deducible. Around 1900, part of Hilbert's programme was to show that every mathematical truth would turn out to be deducible from a set of axioms, and we just had to find those axioms.

But of course this was exploded by the work of Gödel, Kleene, Church and Turing; it was found that any axiomatic system which is strong enough to entail enough of the things we want to be true will also, inescapably, be either inconsistent or too poor to entail other things which we regard as true. There will always be truths which are not deducible, and functions which are not computable. Out of this came several characterisations of what is computable, and they all agreed; Church's thesis states that they will always agree.

The hard core of computing theory ramified, with this foundation, into a classification of what is computable. This ramification began with recursion theory, showing that certain things are essentially harder to compute than others. Modern complexity theory carries on this programme; we now know that a lot of useful things can be computed in linear time (time which is a linear function of the size of the problem), others take non-linear but polynomial time, and unfortunately still others—undeniably *useful* things!—almost certainly take exponential time and a few of them definitely do.

For some Computer Scientists, this study—which gets harder and harder and certainly isn't finished—just *is* computation theory. To them, everything else is engineering of one kind or another; either *hardware* engineering (what configurations of matter do our computations reasonably fast) or *software* engineering (what configurations of text express our computations reasonably clearly).

I want to argue that this is a restrictive view. Of course, any engineering discipline employs mathematical theories in the service of design; theories which help both in *doing* good design and in *validating* artifacts, i.e. establishing that they do the job they were designed to do. Computing is no exception in this, though there isn't yet too much agreement about which mathematical theories are most appropriate to inform design. But it might be argued that these helpful mathematical theories are auxiliary, in the sense that they don't, like the hard core theory does, tell us anything about the nature of computation; all they do (however well they do it) is to help us in building complex computing systems

and in managing their complexity.

This is the position which I want to undermine. What I aim to show is that these apparently auxiliary theories, for managing the things which we build, are just as essential as the hard core; that is, they tell us just as many things, though different things, about the very nature of computation. They don't just help us to organise big systems better; they also provide for us structural concepts which we just cannot do without in thinking about our big systems. And finally, I want to show that we arrive at these concepts, and assess their claim to centrality (or: their inevitability) not just by thinking about big systems but by building them and working with them.

2 Engaging theory with experiment

Britain has always played a prominent part in fundamental computing research. Turing's part has already been mentioned. At Cambridge and Manchester the first stored-program computers—inspired probably by Von Neumann's ideas—were developed by Wilkes and Kilburn, around 1950. At Oxford in the late 60's Christopher Strachey and the American logician Dana Scott inspired one another in beginning the mathematical study of computer program semantics. This was a real fusion of engineering and logical insights, generating an energy which has fired the qualitative—rather than quantitative—study of computation ever since. For some reason USA in the last two decades has been largely preoccupied with the quantitative aspect of computation theory, called complexity theory, while in Britain the emphasis has been more upon semantics. In general, this semantic study has been most strongly represented here in Edinburgh, notably in the work of Plotkin; of course we acknowledge pioneering work such as that of Hoare in proposing special logics for programming, and of Kowalski in uniting logic with programming.

In parenthesis, it is interesting to remark that Artificial Intelligence—with its demand for a kind of programming vastly different from FORTRAN—has provided some of the impetus towards a study of semantics. Perhaps this was because a large part of AI's subject matter was always, and is now, automated inference. Perhaps it was also because the languages inspired by AI—LISP at MIT and POP2 at Edinburgh—were themselves challenging objects of semantic study, since they were informed by the lambda-calculus which Church had established at the foundation of computation theory.

However it began, the work here in Edinburgh on the foundations of computing theory has flourished over the past decade. One of its strongest contributions was to complexity theory, in the work of Valiant (who has left us for Harvard) and his followers. But it has otherwise concentrated on semantics: on program transformations which preserve meaning; on the model theory of programming, and in particular for non-determinism; on semantically-directed design of functional programming languages; on the methodology of computer-assisted reasoning, both in general and with particular application to program correctness; on the modular structure of program specifications; and on the conceptual basis of parallel computation.

In effect, then, we already have a Foundations Laboratory. Why are we gilding the lily, by bothering to establish one? Why should we give the impression that we want to begin to do something, when in fact we are doing it already?

I began to answer this question a few minutes ago. I claimed that one arrives at concepts (at any rate in this subject) not just by thinking about systems but by building them. In fact, the time is right to take this attitude really seriously, because suddenly almost everyone who builds computing systems is convinced that all system design—hardware or software—needs to be done within a rich conceptual frame, which is articulated by rigorous methodologies, which are in turn assisted by computer-based reasoning tools. One key point here is that the conceptual frame provided just by a *programming language* is too narrow; a wider frame is needed in which to understand the specifications of systems, and the methodology which articulates this understanding must be based on some form of logic.

It is excellent that this conviction is now widespread; but there is an attendant danger. Much use has been made recently of the term "formal methods" in relation to system design. The choice of this term (wherever it came from), rather than the term "theory", suggests that the methodology is paramount; I fear that it also reflects a mistaken assumption—that is, it suggests that the conceptual frame for design already exists and that we only need to animate it by the right formal methodology.

One need only recall that most of mathematics has arisen from some kind of application demand, often from the physical sciences, to see how unlikely is this assumption to be valid. So let us assume that it is invalid, and that the conceptual frame for system design is still incomplete. How do we develop it? Well, just as a physicist or chemist tests and refines his theories by carefully controlled experiment, so it should be with us. I believe this analogy is close, and that the word "experiment" is also correct for Computer Science. However, for

a physical scientist an experiment will reinforce (or undermine) his conceptual grasp of *what is true*; for a Computer Scientist, the experiment will reinforce (or undermine) his conceptual grasp of *how to design*. It takes a philosopher of science to decide whether a "theory of *what*" is really different from a "theory of *how*", but later I will boldly question the existence of such a distinction.

For now, let us return to the question of experiment. What is an experiment for us? I claim that it employs a prototype methodology constructed upon some conceptual basis, and that its outcome determines whether we adopt (or reject) the methodology, thus reinforcing (or undermining) the conceptual basis—the "theory of how". Notice that it won't usually be all or nothing. But at least one philosopher of science, Imre Lakatos, argues that no experiment ever completely confirms or denies a "theory of what", either.

In Edinburgh we have contributed the conceptual basis for some such experiments, and are beginning to assess the outcome. I would like to explain two which I have had something to do with.

1. In the mid-70's we designed LCF, a machine-assisted reasoning system based on Dana Scott's model of computation, together with certain notions of proof tactic or strategy with which a user could conduct his proof. At Edinburgh and then at Cambridge, Michael Gordon specialised this tool towards verifying the design of a computer; he called his tool LCF-LSM. Recently at RSRE, Malvern, Dr John Cullyer has succeeded, using Gordon's work, in proving real microcomputers correct (or, more often, faulty!), and he strongly advocates that no micro should be installed in any situation where humans are at risk unless such a proof has been done. This outcome reinforces at least part of our "theory of how"; other parts were not so much undermined as found unnecessary, but then this application was very specialised.

2. Around 1980 we put forward a Calculus of Communicating Systems, an algebraic system for reasoning about concurrent computation informed by a particular concept of *process*. In the last two years two Dutch scientists, Chris Vissers and Ed Brinksma, have designed a methodology based upon our work, which they call LOTOS (Language of Temporally Ordered Specifications), for describing "Communications Protocols"; a protocol is a discipline to be adopted by digital systems which need to exchange messages reliably, even in the presence of faulty media. The design has also been strongly influenced by the seminal work by Tony Hoare on communicating sequential processes. This language LOTOS is well-advanced as

> a candidate for an International Standard, and it has been found rather convenient for its purpose. At the same time, it identified the need for at least one construction which was not derivable from our theory. It has also served as the arena for a satisfyingly heated debate about the central concept of *process*! This has been criticised as too coarse in some respects, too fine in others, and the outcome is not yet clear. But one thing is clear: it is hard to imagine a better way of testing the process concept as a "theory of how".

These two examples should persuade you that theoretical innovation is necessary in Computer Science, that it can be tested, and that we certainly need more of it. Perhaps it has not persuaded you that we should change the way we work as a theoretical group, by founding a Laboratory; after all, in these two examples the experiments *were* done (or *are* being done), and the outcome is there for us to assess.

Well, a large part of the problem is the scale of it. It is the essence of Computer Science that if a methodology doesn't scale up, then it isn't a methodology at all. So a theory which doesn't support a scaling methodology isn't the right theory. But to tackle large experiments inevitably demands software tools. Building these tools alongside the theoretical development is the only way to make theoretical research really sensitive to potential use, and this joint pragma-theoretic approach demands both more research effort and more coordination than is normally supported by an academic department. Another part of the problem is that the people best able to conduct experiments with these tools are usually not academics, but industrial researchers. To induce them to do so requires an environment in which industrial researchers can both attend short courses and (in smaller numbers) reside for longer periods, to discover what theories there are, and to help us see what theories are lacking. And if we are to get feedback from experiment in a controlled and intimate manner, then the environment must both house joint experimental projects with Industry, and also be organised to respond to experiments conducted in the Companies themselves.

At the end of the lecture I shall outline how we propose to develop our Laboratory to make all this possible, but I hope now to have persuaded you that some special initiative is required in order that theoretical research should be closely engaged with its possible application. I now want to look in more detail at two new kinds of theory which promise to help in the design process, and which are being developed in Edinburgh and elsewhere.

3 New theories: (1) Types and Specifications

One of the most far-reaching experiments which Computer Scientists are now conducting is to provide the end user with a *specification-calculus*; that is, he is provided with a way of building large program systems not using programs as his conceptual unit, but instead using specifications—which describe *what* is to be done (not *how* to do it). We shall use the term "type" here as synonymous with "specification"; this reflects the insight that the concept of type familiar in logic can be aligned with what a programmer would like to call a specification. These types, and the properties which relate them, are essential in his "theory of how"; the experiment will succeed, and thereby reinforce this theory, just to the extent that the conceptual frame of types provides users with a vehicle for their design purposes.

A simple example will show how rich these types can be. Let us suppose that we are working with tables of various kinds. An employee table, giving the employment grade of each employee, would have type

$$(NAME, GRADE)\,TABLE$$

This is a simple data-type. But not only *data* may be specified by type; *operations* on data may also be specified by type. For example, a common operation in database systems is to *join* two tables. A salary scale may have type

$$(GRADE, SALARY)\,TABLE$$

Then the operation which will join tables of these two types into a single table can be specified by a type *JOIN*, which may be written

$$JOIN = \left\{ \begin{array}{l} \underline{given}\ input1\ \underline{of\ type}\ (NAME, GRADE)\,TABLE \\ \underline{and}\ input2\ \underline{of\ type}\ (GRADE, SALARY)\,TABLE \\ \underline{yields}\ output\ \underline{of\ type}\ (NAME, GRADE, SALARY)\,TABLE \end{array} \right.$$

Of course more succinct mathematical notation can be used here. Notice that the type *JOIN* doesn't say how to compute anything; it merely says how the output type is related to the input types. Also, the type *JOIN* is not very specific. There are many programs with this type which wouldn't do in the least what we want, which is to produce an output table which exactly combines the information of the two input tables. For example, it might produce an output table with just one entry

whatever the input tables were! A better specification can state what the output-input dependency should be, thus:

$$PROPERJOIN = \left\{ \begin{array}{l} \underline{given}\ input1\ \underline{of\ type}\ (NAME, GRADE)\,TABLE \\ \underline{and}\ input2\ \underline{of\ type}\ (GRADE, SALARY)\,TABLE \\ \underline{yields}\ output\ \underline{of\ type}\ (NAME, GRADE, SALARY)\,TABLE \\ \underline{such\ that}\ project(output)(NAME, GRADE) = input1 \\ \underline{and}\ project(output)(GRADE, SALARY) = input2 \end{array} \right.$$

Notice that this is tighter; anything of this type will also have the type *JOIN*, which we may express as follows

$$PROPERJOIN\ \underline{implies}\ JOIN$$

But still, the new type doesn't say how the joining operation should be done; it only gives conditions which it must satisfy.

It may appear from this example that a type is really a logical formula, and that the calculus of types is a logical system. This is indeed how it is often treated. But it has a strong algebraic character too. This is clearer if we take the example just a single step further. Notice that joining tables is an operation which makes sense whatever the elements in the tables are; it doesn't work just for the element types *NAME*, *GRADE* and *SALARY*. In other words, it is correct and much more useful to treat *JOIN* as a parametric type, where the parameters are types themselves. And this idea is much more general: it turns out that, with some help from category theory, we can build a hierarchy of specifications, where higher members in the hierarchy take lower members as parameters. This is just what Rod Burstall at Edinburgh, and others elsewhere, have done. What emerges, then, is a highly structured conceptual frame in which designers can think and reason about their designs before they express the fine detail in the form of programs. It is the structure among these type objects, not the notation (logical or otherwise) which informs the designer's understanding. It is exciting, but hardly surprising, that the Swedish logician Per Martin-Löf has shown us that his Constructive Type Theory, which he developed purely as a theory of reasoning, gives us an excellent framework in which to handle types and specifications.

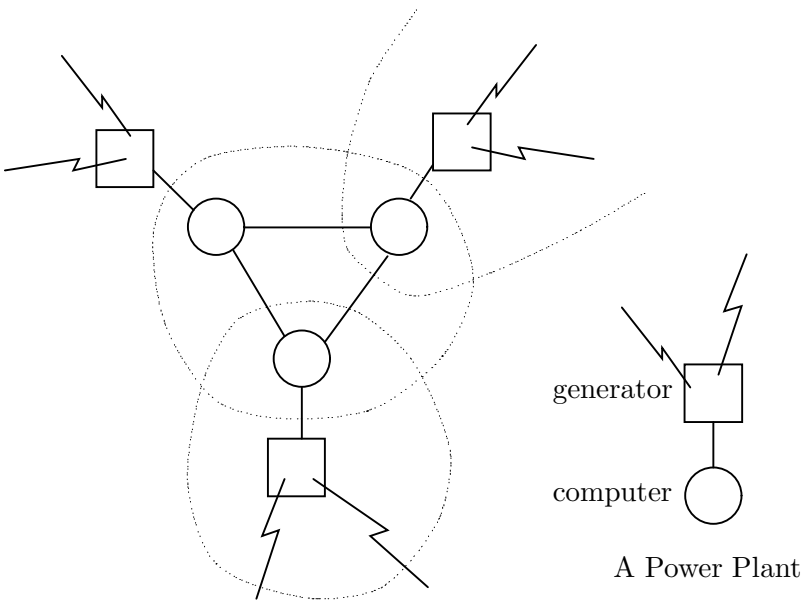
This experiment, dealing with specifications, is about the Man-Machine Interface. It isn't so much about the ergonomics of this interface, a subject which is of great importance and which was chosen as one of the foci of concentration of the Alvey programme. It is rather to do with the things that a human would

like to think about when he is controlling a machine. I would argue that however refined an ergonomic study is made in MMI, it cannot really succeed unless it is firmly based on a structured conceptual frame such as we have proposed for specifications.

4 New theories: (2) The Notion of Process

Let us now turn to the other kind of interface: the Machine-Machine Interface. When one of these machines is a computer and the other is an aircraft, then the communication discipline involves a host of problems to do with real time, estimation, feedback and fault tolerance which are both complex and highly specialised, and which I do not want to discuss here. What I do want to discuss is the symmetry of the interface, which is often obscured. After all, when an aircraft or chemical plant is controlled by a computer, then the interaction is definitely two-way; it helps to think of the computer too as a process, and each of the two processes evokes action from the other. So it leads to an integrated theory if we can model each agent in such a pair by the same conceptual notion of process.

There is more to it than this, because we are not restricted to two parties interacting. Consider three power plants, each consisting of a computer controlling a generator and with the computers linked to each other.



A Power Plant

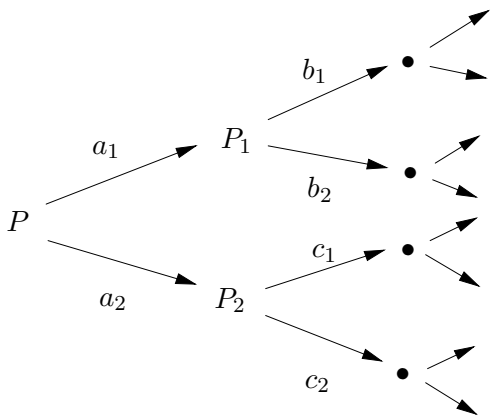
Evidently we can think of this in at least two ways, as the diagram suggests; either as three power plants linked together, or as three generators each linked to a process consisting of three interlinked computers. However we choose to subdivide the system, it is most important that we are able to treat any collection of processes, interacting with each other, as a single process at a higher level of abstraction. In other words, just as with specifications, we must be able to compose and decompose processes hierarchically if we are to have a useful theory.

What I now want to show is that there is a notion of *process* which is so general and so mathematically elementary that it rivals a notion which has become fundamental to a large body of mathematics: the notion of a *set*.

Looked at from the outside, all we can see of a process in a state P is that it can perform any one of a number of possible actions, and move to a new state P' say:

$$\begin{array}{ccc} P & \longrightarrow & P' \\ \text{state} & \text{action} & \text{state} \end{array}$$

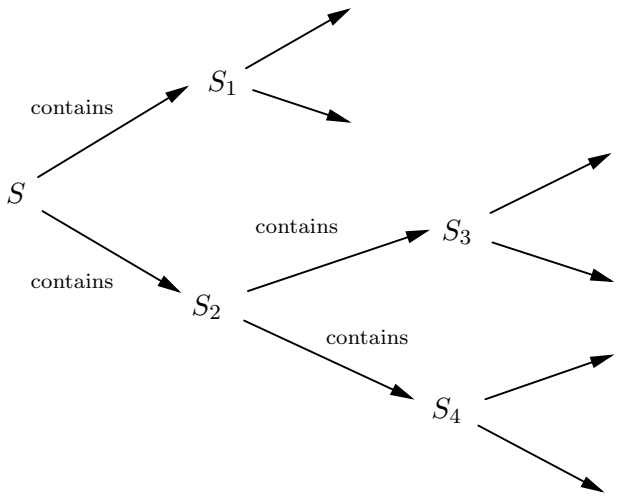
Since it can have several alternative actions in any state, its full capability can be arranged as a tree-like diagram:



Notice that some of these branches may just terminate (the process stops) while others may proceed infinitely. In fact with the help of David Park at Warwick we have worked out a theory in which a process is almost exactly a tree of this kind.

Now consider the mathematical notion of set, which may contain members which are atoms or individuals and other members which are again sets. Just as

a process can be described by a tree-like diagram which tells its *action* story, so a set may be described by a tree-like diagram which tells its *containment* story:



The analogy is that *process* corresponds to *set*, and *process action* corresponds to *set containment* (we can handle the slight difference that actions are of many kinds, while containment is just of one kind).

Take the analogy a step further. When are two sets S and T equal? Answer: when they contain equal members. So we naturally say that two processes P and Q are equal when, for each action, their successors upon that action are equal.

Now we come to a fascinating difficulty. We feel uncomfortable if our test for equality of sets gets into an infinite regress, which happens if there are containment chains which go on for ever

$$S \textit{ contains } S_1 \textit{ contains } S_2 \textit{ contains } \ldots \ ?$$

Such sets, where we can go on for ever taking a member without reaching an atom, are called *non-wellfounded*; because of the infinite regress, mathematicians have usually excluded these sets. But we don't want to exclude infinite processes! All sorts of simple objects, like parking-ticket machines or computers, are perfectly modelled by infinite processes (if we ignore physical decay). Recently a Manchester logician, Peter Aczel has argued that we don't need to exclude non-wellfounded sets either; he proposes what he calls the *antifoundation axiom*, which explains their nature in a very natural way. Within the last year or so, partly during his visit to Edinburgh, he discovered that the theories of infinite

processes on the one hand, and non-wellfounded sets on the other, are essentially *one and the same theory.*

For me this discovery is very exciting. True, it doesn't alter the theory in any way, nor does it provide very detailed suggestions on how to develop the theory of processes. In fact, the pragmatic ramifications of the theory of processes are likely to look very different from the ramifications of set theory which are important for the foundations of mathematics. But it does indicate that the concepts which are useful in a theory of how to build complex concurrent systems are not firmly established in traditional mathematics, and may even have an influence on the development of mathematics. I don't see why school text-books in the year 2000 shouldn't treat processes with as much familiarity as present-day text-books treat wellfounded sets. In other words, processes may make as much claim to existence as wellfounded sets; this is what I meant earlier by questioning the distinction between a "theory of what" and a "theory of how".

This philosophical digression may strike fear and doubt in the minds of people who hope that we shall do useful practical work and would like to use the results. But it should not; already this process work has shed light upon useful practical systems such as the communications protocols which I talked about earlier. We should not be surprised if the concepts which inform a new kind of practice also arouse a new kind of theoretical interest; perhaps we should rather expect it to be so.

5 Composing Calculi

Having given some examples of concepts, like type and process, around which we can build theories of how to design systems, I would like to consider how we can work with these concepts in practice, in a way which permits serious design experiments, which will in turn reinforce (or undermine) the theories.

Simple experiments can be done with pen and paper; anything more complex which will test whether a theory scales up, is going to need more sophisticated tools to help us manage all the details. Now we have seen that types or specifications can be expressed in a logical system, and the same is true of processes. It would indeed be nice if some single logical language, with a single axiomatic basis, were to satisfy all our needs, because then we could set about providing a tool kit of programs to assist designers in using this single calculus. Work along these lines is being done, and will certainly be a valuable experiment; a noteworthy example is the specification language ZED designed by Abrial, which is

being developed and put to experiment by Hoare and his group at Oxford.

But there is good evidence that this approach is limited. For all the success of logicians in founding a large part of mathematics upon a single logic, they have never succeeded—and seldom claimed that they would succeed—in getting mathematicians to do their work in this one medium. The different branches of mathematics are just too various, and our linguistic invention is just too fertile, for this kind of uniformity to be sustained. And the subject matter of computation is equally various, so we would expect the same need for plurality; indeed Kim Larsen and I recently did a small experiment in process validation and found that, in the space of three or four pages, we needed three or four distinct formal calculi to express the natural proof succinctly.

So how do we set about building computer systems which will help people to do different design experiments in whatever combination of calculi is most appropriate? The broad answer follows the general pattern which we have already followed, but we are now working one level higher. We wish to design systems which will help people to design systems; we therefore need a "theory of how" again: a theory of how to design these meta-systems. Here are three analogous questions:

1. What is a specification, and how can specifications be composed hierarchically?
2. What is a process, and how can processes be composed hierarchically?
3. What is a calculus, and how can calculi be composed hierarchically?

We looked at the first two questions earlier, and we are now confronted with the third question. We are using the term "calculus" to mean any systematic way of reasoning; it embraces both the theoretical and the pragmatic aspects of logic.

Consider a simple example. The calculus of matrices involves certain objects—scalars, vectors, matrices; it involves elementary operations like sum and product over these, and more complex operations like partitioning; it involves basic algebraic properties; it involves methods of calculation (e.g. of the inverse); at a more practical level it involves convenient visual means of display. But most important of all, it is a *parametric* calculus; it works over any field of elements. This shows that we must implement calculi in such a way that we can compose the calculus of matrices (with all its pragmatic aids) and the calculus of real numbers (with all its pragmatic aids) to obtain the calculus of real matrices. The same story can be told, almost word for word, for a calculus of specifications or of processes.

This relentless pursuit of hierarchy is now becoming familiar! All the same, its application to logical calculi is something rather new, and is strongly guided by the need for really practical reasoning tools. Some recent work at Edinburgh will help us towards a pragmatic definition of logic, which will allow logics to be built from component logics. Burstall and his group have used category theory to define a notion called *institution* which explains the composition of algebraic theories. At a more basic level, Plotkin (with some ideas from Paulson at Cambridge) is looking at how the apparently simple idea of *inference rule* should be presented to a computer.

This subject is too taxing to take any further in a general lecture; perhaps I have already gone too far for comfort. But I needed to expose this problem of implementing calculi in a hierarchical way, because it is so pervasive to the kind of system-analysis that we wish to promulgate into practical use. Indeed, the projects which have already been funded for our Laboratory are all seriously involved with building calculi which can be used in earnest; we intend to grapple with the theory of how to do this, side-by-side with solid and specific experiments in which industrial designers can participate.

One ingredient which is crucial to this work is the implementation medium. About ten years ago we designed ML, a programming metalanguage for implementing logic. There was no idea at that time of composing logics hierarchically. But in the intervening years the technology of structured functional programming has advanced impressively. Due to a continual flow of ideas about parameterisation from Burstall, MacQueen, Plotkin and others, we now have a vastly superior version of ML, whose power in handling parametric structures exceeds that of any other language in serious use. The language has probably been submitted to more theoretical analysis than any other, and is now reaching its last stages in implementation, in a development project funded initially by the SERC Software Technology Initiative and then adopted by the Alvey Directorate. The language is widely known and its implementation is highly portable; this means that our experiments can be conducted not only in Edinburgh but much more widely.

6 The Laboratory

I would like to finish by arguing that now is the right time, and Edinburgh is the right place, to mount a vigorous attack on computation theory founded on experiment, and by outlining where we have got to and what more is needed to mount this attack.

First, the Edinburgh environment is extraordinarily propitious for the exercise. I have spent some time explaining that we already have successful experience in building theories, with encouraging feedback from application. Our Computer Science Department fully supports the Laboratory, and we have never doubted that the Laboratory should grow within the Department rather than as an independent entity. First, this is because the Laboratory has a commitment to teaching—both undergraduate and postgraduate; we must do all we can to meet the huge demand for scientists and engineers who are well-grounded in the foundations of the subject. Second, we reject any artificial barrier between theory and engineering in Computer Science, and anticipate a healthy flow of ideas between the Laboratory and the Department as a whole. This is already exemplified by the work of George Milne in applying algebraic design methods to VLSI, and should extend to an interaction between theory and more general systems design, both hardware and software, as the Department intensifies its activity in this area.

Beyond our own department, we are lucky to have both the Artificial Intelligence Department and the Centre for Cognitive Science as neighbours. The AI Department has a strong research effort in mechanised reasoning with which we have always interacted, and in our relationship into Industry we hope to form useful links with the AI Applications Institute. Cognitive Science, on the other hand, represents the pervasive role played by logic in all branches of Information Technology. I suspect that we all find it baffling to try to demarcate between the sciences of Computing, Intelligence and Cognition; fortunately, there is so much to do that no-one finds any value in boundary disputes! The problem is how to benefit as fully as possible from each others' pursuits; we are grateful for the efforts of the Committee for Information Technology in giving some coordination to what is certainly the strongest IT research community in UK, probably in Europe.

If Edinburgh is the right place for our Laboratory, then also now is the right time. What makes it right is the strong synergy between Academia and Industry; they are pursuing the same goal of organised (rather than adhoc) understanding of system design. Industry's motive is to produce systems which sell; ours must be to understand the subject for its own sake. But I have tried to show not only that the same kind of understanding is needed for both purposes, but also that the method of reaching this understanding is largely the same—it lies in protracted and organised experiment with theories. For this reason we are confident that Industry will be interested in taking part in our experiments; indeed, our joint projects with ICL, Software Sciences and BP already give firm evidence of this.

As well as Industry, the Alvey Directorate have already given us considerable

financial support. Alvey have funded the post of Assistant Director of the Laboratory, in which we are lucky to have an enthusiastic and capable man, George Cleland; they are supporting our ML development project, and have recently bought us a Pyramid computer. Finally, a very strong commitment has come from SERC who have approved further funding for our research projects over the next 3 or 4 years, to the tune of about £620,000. This has placed us in a solid position; the *theoretical* backbone of the Laboratory is now assured. These projects are built around the implementation of logical calculi as a focus. What we shall be doing over the next few years is to surround this backbone with an increasing volume of interaction with *applications*; for this we look to Industry for increasingly active support, and to the University to help us find the space in which to exploit this support in the fullest possible manner. If we do our job properly we shall also have contributed something permanent to our subject, and made the case for establishing our Laboratory as a permanent entity.