Elements of

I am greatly honored to receive this award, bearing the name of Alan Turing. Perhaps Turing would be pleased that it should go to someone educated at his old college, King's College at Cambridge. While there in 1956 I wrote my first computer program; it was on the EDSAC. Of course EDSAC made history. But I am ashamed to say it did not lure me into computing, and I ignored computers for four years. In 1960 I thought that computers might be more peaceful to handle than schoolchildren—I was then a teacher—so I applied for a job at Ferranti in London, at the time of Pegasus. I was asked at the interview whether I would like to devote my life to computers. This daunting notion had never crossed my mind. Well, here I am still, and I have had the lucky chance to grow alongside computer science.

This award gives an unusual opportunity, and I hope a license, to reflect on a line of research from a personal point of view. I thought I should seize the opportunity, because among my interests there is one thread which has preoccupied me for 20 years. Describing this kind of experience can surely yield insight, provided one remembers that it is a personal thread; science is woven from many such threads and is all the stronger when each thread is hard to trace in the finished fabric.

The thread which I want to pick up is the semantic basis of concurrent computation. I shall begin by explaining how I came to see that concurrency requires a fresh approach, not merely an extension of the repertoire of entities and constructions which explain sequential computing. Then I shall talk about my efforts to find basic constructions for concurrency, guided by experience with sequential semantics. This is the work which led to a Calculus for Communicating Systems (CCS). At that point I shall briefly discuss the extent to which these constructions may be understood mathematically, in the way that sequential computing may be understood in terms of functions. Finally, I shall outline a new basic calculus for concurrency; it gives prominence to the old idea of *naming* or *reference*, which has hitherto been treated as a second-class citizen by theories of computing.

I make a disclaimer. I reject the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent computation, which is in a sense the *whole* of our subject—containing sequential computing as a well-behaved special area. We need many *levels of explanation:* many different languages, calculi, and theories for the different specialisms. The applications are various: the flow of information in an insurance company, network communications, the real-time communication among in-flight control computers, concurrency control in a database, the behavior of parallel object-oriented programs, the semantic analysis of variables in concurrent logic programming. We surely do not expect the terms of discussion and analysis to be the same for all of these.

But there is a complementary *claim* to make, and it is this: Computer scientists, as all scientists, seek a common framework in which to link and to organize many levels of explanation; moreover, this common framework must be semantic, since our explanations (including programs) are typically in formal language—and often in a mixture of formalisms, to deal with the large heterogeneous systems which are our business. For the much smaller world of sequential computation, a common semantic framework is founded on the central notion of a *mathematical function* and is formally expressed in a functional calculus—of which Alonzo Church's λ-calculus is the famous prototype. Functions are an essential ingredient of the air we breathe, so to speak, when we discuss the semantics of sequential programming. But for concurrent programming and interactive systems in general, we have nothing comparable.

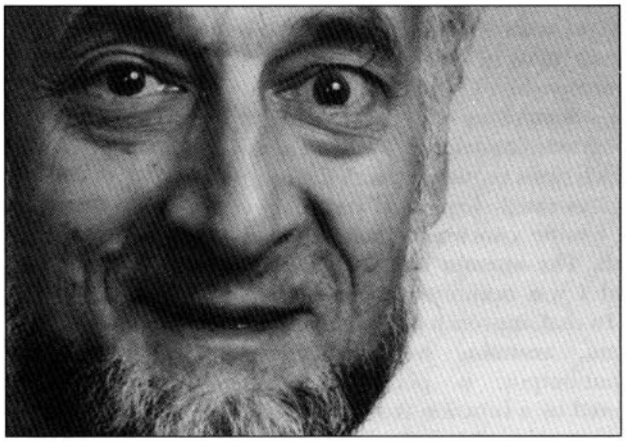
So where do we find the semantic ingredients for concurrency, or how can we distill them? It is an ambitious goal because, as I said earlier, concurrency is ubiquitous. I believe that the right ideas to explain concurrent computing will only come from a dialectic between models from logic and mathematics and a proper distillation of a practical experience.

I conduct a piece of the dialectic. I try to reconcile the antithesis—for it does seem to be one—between two things: on the one hand, the purity and simplicity exemplified by the calculus of functions and, on the other hand, some very concrete ideas about concurrency and interaction suggested by programming and the realities of communication.

Interaction

Turing Award Lecture

Robin Milner



Elements of Interaction

Entities

Through the seventies, I became convinced that a theory of concurrency and interaction requires a new conceptual framework, not just a refinement of what we find natural for sequential computing.

Often, the experiences which give one conviction are not planned and not profound. But I want to recall one of mine, because it serves the theme here in more than one way. It arose when I was trying to extend the Scott–Strachey approach to programming-language semantics, which deals beautifully with the most sophisticated sequential languages, to handle concurrent languages as well. The attempt had to be made, and I was optimistic about success.

In that approach a sequential program, assuming no intermediate input/output, is perfectly represented by a function from memories to memories. (I use the term "memory" to mean a memory state, containing values for all the program variables.) But Dana Scott developed a theory of *domains*—partially ordered sets of a special nature—which provides meaning for the λ-calculus, the prime functional calculus. So in the Scott–Strachey approach, the meaning of an imperative program lies in the domain given by the equation

$$\text{Program Meanings} = \text{Memories} \rightarrow \text{Memories}.$$

Everything works well with this domain, and the reason is: that to every syntactic construction in any sequential language, there corresponds an abstract operation which builds the meaning of a composite program from the meanings of its component programs. That is, the semantics is *compositional*—an essential property.

Now, one of the things that concurrency introduces is nondeterminism. (Of course you can also have nondeterminism without concurrency, but in my opinion it is concurrency which *inflicts* nondeterminism on you.) Plotkin dealt with nondeterminism by means of his powerdomain construction, a *tour de force* of domain theory. It provides, for any suitable domain D, the powerdomain $\mathcal{P}(\mathcal{D})$ whose elements are subsets of D. So with nondeterminism in mind we can redefine the meanings of programs as

$$\text{Program Meanings} = \text{Memories} \rightarrow \mathcal{P}(\text{Memories})$$

—essentially relations over memories. This semantics is perfectly compositional for the kind of nondeterministic language which you get by adding "don't care" branching to a sequential language.

But concurrency has a shock in store; the compositionality *is lost* if you can combine subprograms to run in parallel, because they can interfere with one another. To be precise, there are programs P_1 and P_2 which have the same relational meaning, but which behave differently when each runs in parallel with a third program Q. A simple example is this:

Program P_1 : x := 1 ; x := x + 1
Program P_2 : x := 2

In the absence of interference, P_1 and P_2 both transform the initial memory by replacing the value of x by 2, so they have the same meaning. But if you take the program

Program Q : x := 3

and run it in parallel with P_1 and P_2 in turn:

Program R_1 : P_1 par Q
Program R_2 : P_2 par Q

then the programs R_1 and R_2 have different meaning. (Even if an assignment statement is executed indivisibly, R_1 can end up with x equal to 2, 3, or 4, while R_2 can only end up with x equal to 2 or 3.) So a compositional semantics must be more refined; it has to take account of the way that a program *interacts* with the memory.

This phenomenon is hardly a surprise, with hindsight. But if we cannot use functions or relations over memories to interpret concurrent programs, then what *can* we use? Well, one can quite naturally give the relational meaning a finer granularity, so that it records every step which a program makes from one memory access to the next—and this can be done without leaving domain theory. But the phenomenon taught me a more radical lesson: Once the memory is no longer at the behest of a single master, then the master-to-slave (or: function-to-value) view of the program-to-memory relationship becomes a bit of a fiction. An old proverb states: He who serves two masters serves none. It is better to develop a general model of interactive systems in which the program-to-memory interaction is just a special case of interaction among peers.

It helps to visualize. Figure 1 shows the shared-memory model, very informally. It just represents the active/passive distinction between components, by using differently shaped nodes. (I shall consistently use squares for active processes in my pictures and circles for passive things.) Of course, in general the programs use several variables, all stored in M.

To remove the active/passive distinction, we shall elevate M to the status of a process; then we regard program variables x, y, . . . as the names of *channels of interaction* between program and memory, as shown in Figure 2.

Now, thinking more generally, let us use memories to illustrate the idea that processes—of *any* kind—can be composed to make larger ones.

In the sequential world one can maintain the convenient fiction that a memory is monolithic; but this is quite unrealistic in concurrent programming, because different parts of memory may be accessed simultaneously. So we go one step further, as shown in Figure 3, and regard each *cell* of memory as a process, X say, linked to one or more programs (themselves processes) by an appropriately named channel.

Software engineers may well resist this homogeneous treatment and firmly adhere to the shared-memory model; it is important for them, because it admits a methodology which can help in writing correct programs. Theoreticians may reply that to tolerate two kinds of entity in a *basic* model, where one kind will do, is scientific anathema; they may also point out that the active/passive distinction of the shared-memory model does not easily accommodate hybrids, such as a database which reorganizes itself while you are not using it. And both these attitudes are right.

So let us recall the need for many levels of explanation. William of Occam opposed the proliferation of entities, but only when carried beyond what is needed —*præter necessitatem*! Computer systems *engineers* have a pressing need for a rich ontology; they welcome the ability to use different concepts and models for different purposes. For example, the shared-memory model is a natural part of their repertoire. But computer *scientists* must also look for something basic which underlies the various models; they are interested not only in individual designs and systems, but also in a unified theory of their ingredients. To attain unity in a basic model of concurrency, all interactions—and therefore all interactors—must be treated alike; that is why I have called this work "Elements of Interaction."

To avoid the impression that the only interactors I am thinking of are programs, or memories, or computer systems, I show in Figure 4 a mobile telephone network in which the channels are radio channels. The communication protocol allows a car to switch channels to whichever base station is nearest, the whole system being monitored and controlled centrally. Now, we want our constructions to describe such systems perfectly well, at a discrete level; the elements of interaction must not be specific to computer systems.

Much of what I have been saying was already well understood in the sixties by Carl-Adam Petri, who pioneered the scientific modeling of discrete concurrent systems. Petri's work has a secure place at the root of concurrency theory. He declared the aim that his theory of nets should—at its lowest levels—serve impartially as a model of the physical world and as a model of computation. Already, for him, a memory register and a program are modeled by the *same kind of object*—namely a net—and this breaks down the active/passive dichotomy. The conceptual framework of net theory is as spare as one can imagine. This has indeed paid off in clarity and depth, both for the analysis of individual systems and for the classification of systems.

Static Constructions

Besides calling the question the active/passive dichotomy for the *entities* of which a system is composed, concurrency demands a fresh approach

Figure 1. The shared memory model

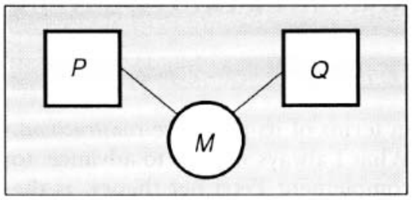


Figure 2. Memory as an interactive process

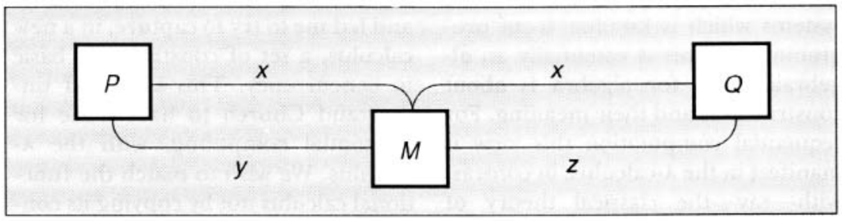


Figure 3. Memory as a distributed process

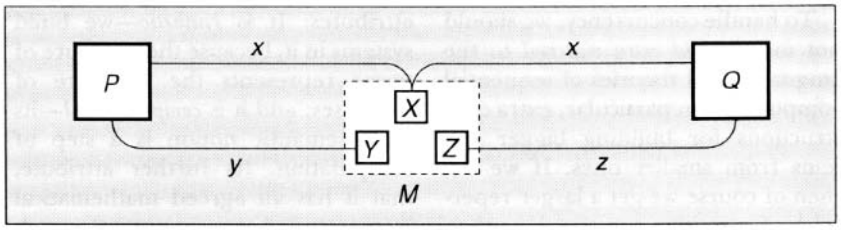
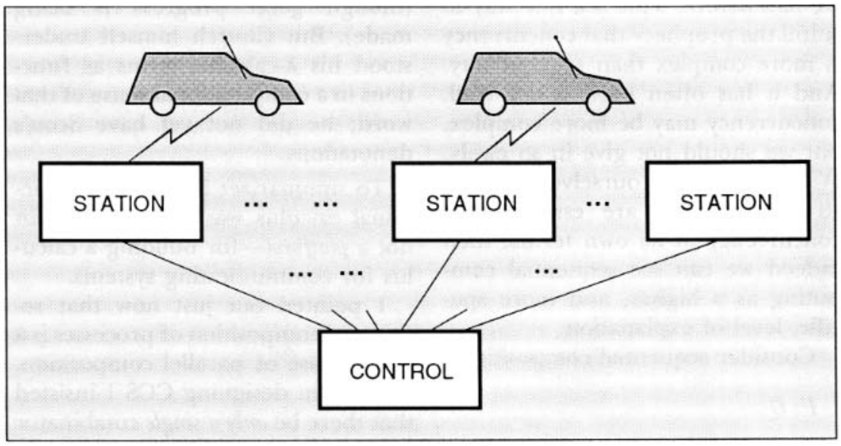


Figure 4. A mobile telephone network



in terms of its primitive *constructions.* What I always wanted to advance, to complement Petri net theory, is the synthetic or compositional view of systems which is familiar from programming. This is essentially an algebraic view, for algebra is about constructions and their meaning. For sequential computation this view is manifest in the λ-calculus, in contrast with—say—the classical theory of automata.

To handle concurrency, we should not merely *add extra material* to the languages and theories of sequential computing—in particular, extra constructions for building bigger systems from smaller ones. If we do, then of course we get a larger repertoire of primitive constructions than we had before. This is a fine way to fulfill the prophecy that concurrency is more complex than sequentiality. And it has often been done. Well, concurrency may be more complex, but we should not give in so easily. We should limit ourselves to constructions which are essential for concurrency in its own terms; then indeed we can see sequential computing as a higher, and more specific, level of explanation.

Consider sequential composition

$$P; Q$$

—the familiar semicolon, the essential glue of sequential imperative programming. To get concurrency, should we keep sequential composition and just *add* parallel composition? Well, we might want to do that for a programming language, because we must give programmers their familiar tools as well as newer things. But should we do it in a basic calculus? I believe not; for sequential composition is indeed a special case of parallel composition

$$P|Q$$

when this construction is properly understood. I understand it to mean that P and Q are acting side by side, interacting in whatever way we have designed them to interact. So sequential composition is the special case in which the only interaction occurs when P finishes and Q begins. To allow a *special* kind of interaction here would violate our principle that, in the basic model, all interactions are of the same kind.

It was this sort of mundane observation which prevented me from trying to *extrapolate* from sequentiality and led me to try to capture, in a new calculus, a set of constructions basic to concurrency. This is what I understand Church to have done for sequential computing, with the λ-calculus. We wish to match the functional calculus not by copying its constructions, but by emulating two of its attributes: It is *synthetic*—we build systems in it, because the structure of terms represents the structure of processes; and it is *computational*—its basic semantic notion is a step of computation. Its further attribute, that it has an agreed mathematical interpretation, we cannot yet match (though good progress is being made). But Church himself understood his λ-calculus terms as functions in a computational sense of that word; he did not yet have Scott's denotations.

To summarize: For me, the functional calculus was a *paradigm*—but not a *platform*—for building a calculus for communicating systems.

I pointed out just now that sequential composition of processes is a special case of parallel composition. Indeed, in designing CCS I insisted that there be *only a single* combinator for combining processes which interact or which coexist. This may seem a tall order, for I also insisted that memory registers be modeled as processes, so this same combinator must be able to assemble them into a memory, to compose the processes which use them, and to combine processes with memory. But one combinator does indeed suffice, and this is because all interactions can indeed be treated in the same way. For example, we can write the system of Figure 3 as

$$P|M|Q \quad \text{where} \quad M = X|Y|Z$$

or as

$$P|X|Y|Z|Q$$

The very same expression will be used even when the programs P and Q interact in some other way, over and above their interaction *via* memory, or when X, Y, and Z are not simply storage registers, but perhaps processes that are intermediary between the programs and a remote memory. The form of the expression is independent of the nature of these five processes.

The algebraic nature of the calculus is beginning to emerge, with this single combinator at its heart. The intuition behind parallel composition is that we are simply assembling the components of a system together—so we expect the combinator to be associative and commutative. This is why we have no brackets in our expressions. Each different ordering and bracketing of the members would represent a different partition of a system into subsystems.

How can our algebra reflect more explicitly the structure induced by the *linkage* among system components? We note first that the components P, Q, X, . . . in Figure 3 will themselves be process expressions; moreover, the channel y links only the members P and Y, since those will be the only expressions in which the channel name y appears. We do not give here the process expression for a register like Y; suffice it to say that each such expression will determine its location as a channel name. Thus we can say that from P's viewpoint, the name y *locates* the cell Y. Now Figure 3 exhibits this idea of *location* very clearly; we also want our algebra to capture the idea. For this purpose, we introduce a further combinator to ensure that the register Y is accessible only to P—i.e., that the channel y is local to them. We call this new combinator *restriction;* for example, in the expression

$$\nu y(Y|P)$$

the channel y is restricted for use between Y and P. The greek letter ν is used partly for the pun on "new"; in fact, restriction is just a distillation of the notion of *local variable declaration* in programming. Thus for the whole system of Figure 3 we may more accurately write

$$\nu x\Big(X|\nu y(Y|P)|\nu z(Z|Q)\Big) \qquad (1)$$

This kind of expression well represents the *spatial* or *static* structure of interactive systems in general, though we have only illustrated it for programs and memory. Diagrams like Figure 3, which we may call *flow graphs,* are quite formal objects. Indeed, restriction and composition obey certain equations which define the algebraic theory of flow graphs.

Dynamic Constructions

Now let us turn to the *dynamic* aspect of such systems: their behavior. It is in the dynamics, in fact, that we can contrast the sequential, hierarchical control of the λ-calculus with the concurrent, heterarchical control of CCS. In the λ-calculus, all computation comes down to just one thing, called *reduction*—the act of passing an argument to a function; we may call this the *atom of behavior* of the λ-calculus. In just the same way, an *interaction*—the passage of a single datum between processes—is the atom of behavior in CCS. We shall now see how this allows symmetry between partners and how it reflects the idea that each process in a community has persistent identity.

We shall look at this contrast in terms of the basic computational rule of each calculus. In each calculus, systems are built using a binary combinator. In the λ-calculus the combinator is called *application* and is neither commutative nor associative. When one term M is applied to another, N,

$$M(N)$$

then the first term M—the *operator*—holds control. The operator is committed to receive N, the *operand;* if and when the operator takes the form of an *abstraction* $\lambda x.M[x]$ (where the square brackets indicate that M may contain the bound variable x), then the symbol λ represents the operator's locus of control, and the following *reduction* will occur:

$$(\lambda x.M[x])(N) \rightarrow M[N]$$

Sources and Related Work

Some of the important sources and relevant papers are cited here, section by section.

Entities. Tennent [36] is a classic exposition of the Scott–Strachey approach to semantics. Gunter and Scott [13] is a recent exposition of Scott's domain theory. It contains a section on powerdomains, which were originally published in [32]. Petri's Ph.D. dissertation [31] is the first publication on Petri net theory, and Reisig [33] is an introductory textbook.

Static constructions. Church [9] is the first publication on the λ-calculus; Barendregt [5] gives a recent and complete exposition of the λ-calculus. Milner [21] gives the first exposition of CCS and brings it up to date in a later textbook [22]. Milner in [20] gives the algebra of flow graphs, a static algebra of processes.

Dynamic constructions. Hoare [16] introduces the programming language CSP and in a textbook [17] gives its algebraic theory. Baeten and Weijland [4] expound the process algebra ACP due originally to Bergstra and Klop. The specification language Lotos was designed by Brinksma [7].

Meaning. Some important domain-theoretic models for concurrency are the failures model of Brookes et al. [8], Hennessy's acceptance trees [14], and Abramsky's domain for bisimulation [1]. The observation equivalence of processes was introduced by Milner [21]; Park [30] placed it on a firmer mathematical footing with the notion of bisimulation. The event structures of Nielsen et al. [28] combine domain theory with the causality structure of Petri nets.

Values. Böhm and Berarducci [6] show how to encode data structures into the λ-calculus. The second-order (polymorphic) λ-calculus was discovered in the early seventies independently by Girard [11] and Reynolds [34]. An early presentation of Hewitt's Actors model is by Hewitt et al. [15]; a recent book on Actors is by Agha [2]. Smalltalk [12] is probably the first programming language to treat values—even the simplest, like numbers—as objects which receive messages.

Names. Astesiano and Zucca [3] studied a version of CCS in which channels (i.e., names, in the terms of this article) could be parametrized on values, thus allowing some mobility. Kennaway and Sleep [19] built the idea of transmitting names as messages into a language for distributed operating systems. In 1980, at Århus, Mogens Nielsen and I had tried to treat mobility algebraically in CCS, but failed. Engberg and Nielsen [10] later succeeded, giving the first algebraic treatment of a process calculus with dynamic reconfiguration (their report is unpublished). These ideas were refined and strengthened by Milner et al. leading to the π-calculus [25]. Milner [23] gives a recent tutorial exposition of a more general form, also treating type structure.

As an accessible illustration of how to apply the π-calculus, Orava and Parrow [29] made a rigorous study of a simplified mobile telephone network, which I have used here. A translation of the λ-calculus into the π-calculus is given by Milner [24]. Walker [38] explored first the use of the π-calculus to give semantics to concurrent object-oriented programming. Honda and Tokoro [18] give an interesting variant of the π-calculus suitable for asynchronous communication, which brings it closer to Actors and to object-oriented programming.

Another way to achieve mobility is to allow *processes themselves* to be transmitted in interaction. Nielson [27] and Thomsen [37] have studied these so-called *second-order* processes. Thomsen also gave a translation of second-order processes into the π-calculus and showed that it preserves operational behavior. Sangiorgi generalizes this translation to ω-order processes and has proved that a suitable behavioral equivalence is preserved in both directions by the translation. His results are summarized by Milner [23] and will appear in Sangiorgi's Ph.D. dissertation [35]. This work on higher order reinforces the claim of expressive completeness for the π-calculus.

Conclusion. The Turing award lecture of Newell and Simon [26] examines the nature of empirical enquiry in computer science.

Elements of Interaction

(where the brackets on the right indicate that N has replaced x in M). This is the basic computational rule of the λ-calculus, the dynamics of function application; for many people it is more familiar as the copy rule of Algol60, which was derived from it. Figure 5 represents this operation pictorially; note that the operand N is represented by a circle, being a passive datum in the reduction—it will only later assume control, as an operator, when M allows it to do so.

In CCS on the other hand, when two terms P and Q are composed in parallel,

$$P|Q$$

then *both* hold control. P may receive a message from Q, just as Q may receive one from P. There are two so-called *actions* which a term may take, allowing it to interact with another. When one partner takes the *input* form $\lambda x.P[x]$ while the other takes the *output* form $\bar{\lambda}V.Q$, then we have the composition

$$\lambda x.P[x]|\bar{\lambda}V.Q$$

in which the partners have what may be called *complementary* loci of control λ and $\bar{\lambda}$—that is, the positive and negative ends of a channel named λ. Then the following *interaction* may occur:

$$\lambda x.P[x]|\bar{\lambda}V.Q \rightarrow P[V]|Q$$

It is a synchronization of the partners' actions.

But crucially, in concurrency there are *many* channels, not just one; so the symbol λ, which represented the single locus of control in the λ-calculus, now becomes just one of many channels, perhaps concurrently active. CCS uses the simple names a, b, c, . . . for such channels. Thus we arrive at the basic computation rule of CCS, shown pictorially in Figure 6. Computation just consists in the iteration of this single rule, repeatedly transforming an expression; at any stage the structure of the expression represents the spatial configuration of the system. The diagram in Figure 6 shows the dynamic behavior—the passage of a passive datum—superposed on the flow graph which represents the spatial configuration.

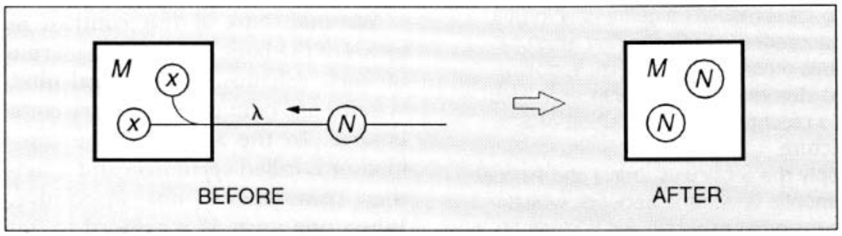
Let us look more closely at the differences here. *First,* in the λ-calculus case the second partner was merely a passive datum, but in the CCS case it yields up its datum V and continues an independent existence. It is exactly this which admits *continual* interaction among concurrent processes, each retaining its identity.

Second, parallel composition is *commutative;* $P|Q$ means the same as $Q|P$. In CCS, in contrast to the λ-calculus, either partner may act as the receiver—and indeed that partner may later become the transmitter.

Third, parallel composition is also *associative;* together with commutativity, this frees the interaction discipline from the term structure, which represents hierarchical control in the λ-calculus. As we saw in the example of program-memory interaction, the control discipline is now determined by which channels a, b, . . . link which processes.

These details are somewhat technical. But the focal point is this: The difference between concurrent and sequential computation can be concentrated in the single *basic computation rule* of the respective calculi. By modifying the notion of *reduction* from the λ-calculus, we attain a rule of *interaction* for concurrent processes, and this is the first stage in our synthesis between the purity of the functional model and the dynamics of real concurrent systems.

The idea of synchronized interaction as a programming primitive was known to Tony Hoare before I expressed it in algebraic form. The fact that these steps were taken independently, with different motives, is some evidence that the idea is a natural one. Hoare incorporated the idea in his programming language CSP and later with colleagues developed a theory around it. It provided the basis for the rendezvous mechanism of Ada, as well as becoming the atom of interaction in the programming language Occam. My contribution was to make interaction the cornerstone of an algebraic calculus. The process algebras developed in the eighties, such as CCS, CSP, and the ACP of Bergstra and Klop, have been the subject of much semantic



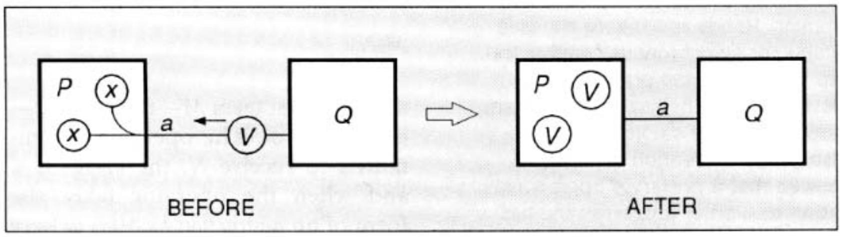


Figure 5. The λ-calculus reduction $(\lambda x.M[x])N \rightarrow M[N]$

Figure 6. The CCS interaction $ax.P[x]|\bar{a}V.Q \rightarrow P[V]|Q$

research. They have also become, or have grown into, actively used design tools; in particular, the specification language Lotos has been widely applied to communication protocols.

Meaning

I would like to allay the fear that concurrency, at least in the way I am treating it, is desperately concrete and mechanistic. Are there not abstract mathematical things underlying it all, for which we should reserve the term "process"—rather than dignifying our algebraic expressions with that name? After all, we have come to use the term "function" in an abstract sense, so that when we point to an expression in the λ-calculus and say *"that function,"* people usually take us to mean *"the mathematical function denoted by that expression."* If an expression has no agreed denotation we may feel uncomfortable.

The semantics of processes has a large and growing literature. It is not simple; but some exciting progress has been made. I spoke earlier about the possible use of Scott's *domains* for concurrent programs, and indeed a variety of interesting domains are studied for process algebra. There has also been much research on the meaning of a process as determined by its *observable behavior.* The idea is this: To observe a process is exactly to interact with it, and two systems—say two process terms in CCS—should have the same denotation if and only if we cannot distinguish them by observation. This research is largely devoted to classifying the stronger and weaker kinds of observational distinction which can be made and to a mathematical characterization in each case. Much has been achieved; much remains to be done.

These remarks about observation, so far, apply equally to sequential processes; but—as might be expected—concurrency raises problems all of its own. One important topic is the concept of *causal independence,* which is central in Petri net theory. Now, two processes—each having concurrent components—may be indistinguishable to an external observer, and yet differ in the causal relationship among their observed actions. I may observe you falling out of the tree and then observe the ambulance arriving, but I still do not know if one caused the other. Hence observational semantics ignores causality. But there are also good reasons for semantic models which respect causal connection—such as the *event structures* of Nielsen et al. This topic is too complex to tackle here; I mention it only to show that concurrency does indeed raise new semantic questions.

Whatever mathematical models are studied, I believe that process calculi provide an essential perspective for the study. Many people will only be satisfied with the semantic theory of concurrent systems when—eventually—it becomes an abstract theory as well as a formal one. But to attain this goal we must first distill the essence of the dynamics of interaction; this is what a formal process calculus like CCS tries to do.

Values

We now embark on the second stage in a synthesis between the functional paradigm and the realities of interaction. This time I am concerned with making a concurrent calculus fully expressive, within its own conceptual frame. First, I need to explain how the λ-calculus succeeds in this respect, while CCS and similar concurrency calculi fall short. Later we shall find a remedy.

Often people use the λ-calculus informally, simply to derive new operators over familiar data. When discussing arithmetic, they will write

$f = \lambda x\ (x^2 + 1)$

or perhaps

$F = \lambda f\ \lambda x\ (f(x) + 1)$

—i.e., they mix the constructions of the λ-calculus with those of arithmetic, not wishing to code one into the other. Treated thus, the λ-calculus is like some computational scaffolding, up and down which the *real workers* climb: data such as numbers, and operators such as squaring, adding, or differentiating.

This auxiliary use of the λ-calculus is very natural, but hardly makes for a self-contained model of computing within the functional framework. Each time we add new types of data structure—like arrays, lists, trees—we are in effect extending the calculus. But it is important for the understanding of programming languages, for example, that the basic model which we use to explain them should be as homogeneous and complete as possible, requiring no *ad hoc* extension.

Quite remarkably, the λ-calculus does achieve this homogeneity and completeness. In its pure form, it does indeed have full power to represent data structures and compute with them; we can do everything with the scaffolding alone. Church showed this for arithmetic; Böhm and others have extended it to other forms of data structure. Moreover, there are *type disciplines*—notably the Girard-Reynolds second-order λ-calculus—which allow this to be done in a controlled way, building a uniform framework for analyzing sequential programming languages—and *their* respective type disciplines.

Now, what about concurrency? Most concurrency formalisms also provide a scaffolding, of a different kind, on which values and operators can move around; these values and operators are quite distinct from the scaffolding itself. Look again at CCS (Figure 6); in that interaction, the datum V is actually an expression standing for something like a *number*—quite a different type of animal from the *process* expression P.

To underline the point, Figure 7 shows a CCS definition for a process *Double,* which repeatedly inputs a number along the a channel and outputs twice that number on the b channel. Regarding the short definition, it contains no less than *five* different kinds of thing: *processes* (e.g., *Double*), *channels* (a, b), *variables* (x), *operators* (×), and *value expressions* ($2 \times x$). And four of these (all except operators) already occur in the basic computation rule of CCS—see Figure 6. Is not this promiscuity excessive?

Well, it is perfectly admissible in a high-level programming language or specification language such as Lotos, where a user expects a rich and redundant tool kit. But for a truly basic calculus the situation is less comfortable. A basic calculus should impose as little taxonomy as it can, because different higher levels of explanation will impose different taxonomies.

Now, the pure λ-calculus is built with just two kinds of thing: *terms* and *variables*. Can we achieve the same economy for a process calculus? Carl Hewitt, with his Actors model, responded to this challenge long ago; he declared that a value, an operator on values, and a process should all be the same kind of thing: an *actor*. This goal impressed me, because it implies the homogeneity and completeness of expression which we have discussed. But it was long before I could see how to attain the goal in terms of an algebraic calculus. I knew that CCS fell short; but it would, I hoped, yield insight to find the rest of the way.

Names

Recently, building on important insights by Engberg and Nielsen my colleagues Joachim Parrow and David Walker and I have found a calculus which has the same kind of internal completeness as the pure λ-calculus. It is also terse; it is hard to see how it can be further simplified, within the tradition of algebraic process calculi. We call it the π-calculus, and the key idea is *naming* or *reference*. (Perhaps we should not be surprised that we attain greater completeness of expression by giving greater weight to naming; for concurrent activity in a system entails the independent *identity* of its components, and to exploit this identity requires a means of *identification*.)

Naming, though so well hidden in some models (such as the functional model), is pervasive in practical computing. Think of all the variants it has. In essence, it is just that which gives access to . . . well, anything; but we tend to think of *variables, addresses, locations, pointers, channels* differently in different contexts—really because they give access to different things. In programming languages, ancient and modern, the variety is striking: the variables of Algol60, the pointers of Pascal, the channels of Ada or Occam, the logic variables of logic programming, the object references in object-oriented programming, and so on. So, in the spirit of Hewitt, our first step is to demand that all things denoted by terms or accessed by names—values, registers, operators, processes, objects—are the same kind of thing: they should *all* be processes. Thereafter we regard access-by-name as the raw material of computation; what we then have to do is to define means by which processes—themselves *accessible*—manipulate *access*.

To get a flavor of this radical change, look again at the CCS computation rule, Figure 6. In that picture a, x, and V are of different kinds: a channel, a variable, a value. In the π-calculus, the channel and variable will both be just *names*, while a value like V will be a *process located by a name*. The crucial intuition is this: A value is just a special kind of process, one which can be repeatedly the subject of the same observation. So the expression V will now indeed be in the same class as any other process expression—and to observe it, there must be a channel for interaction. Thus the value V will be located by a name, say v, just as a memory cell X (see Figure 3) was located by the name x. So in the π-calculus the computation step takes the form

$$ax.P[x]|V|\bar{a}v.Q \rightarrow P[v]|V|Q$$

The computation step is shown pictorially in Figure 8. In that picture every capital letter stands for a process, while every small letter is a name. Note also that a name can be either *actively used* as a channel (as in the case of a) or *passively mentioned* as a datum (as in the case of x and v).

But in this interaction the value V is no longer a participant, not even passively. Thus the simplest way to present the interaction is as follows:

$$ax.P[x]|\bar{a}v.Q \rightarrow P[v]|Q$$

This, finally, is the basic rule of computation in the π-calculus; it shows clearly the crucial difference from CCS—that the transmitted datum is now "just" a name, a means of access. Since names have no structure, the computation step is truly atomic—in contrast with the λ-calculus where the operand of a reduction may be a complex datum.

You may well have two reactions to this. *First,* you may see nothing new; you were always aware that pointers are passed around instead of actual values, in real down-to-earth computing. I reply that if the elements of interaction are already familiar from programming, then it is all for the better—and also, that we shall use them outside the field of programming as well.

Second, you may object that all this mechanical business—pointers and whatnot—was swept under the carpet and was supposed to remain well hidden, when we began to think of computing in a properly abstract way. I agree, to this extent: Although we think about sequential computing in a nicely abstract way, we have not gone very far in that direction for concurrency. But here is the crux: On the one hand the notion of reference *refuses* to remain hidden, when we properly confront reality; on the other hand it seems to resist theoretical treatment—at least, it has received little. I believe the π-calculus begins to resolve this impasse; it begins to provide a tractable theory for reference, and thereby also for concurrency and interaction.

I shall not discuss the π-calculus in any detail. But to show that it approaches the λ-calculus in economy, let me exhibit the grammar of both calculi, side by side (see Figures 9 and 10). The *alternative action* construction of the π-calculus means—as in CSP, CCS, or Occam—that exactly one alternative will occur. The *replication* construction $!P$ roughly means $P|P|\cdots$— as many copies of P as you like in parallel. We have already illustrated the other constructions.

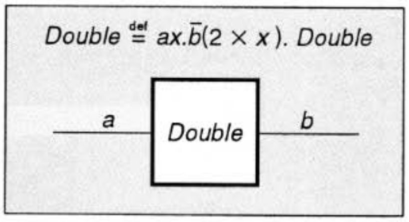
To see how the calculus works, recall the mobile telephone network of Figure 4. We shall model the channel between a car and a station as two π-calculus names—one for talking and one for switching; see Figure 11. The figure shows the definition of the CAR process, parametric on these two channels. There are two alternative actions for the CAR; it can *talk,* or it can *switch* its channels if requested by the STATION. (The CAR is defined recursively here, but recursion can be derived from replication in the π-calculus.) The figure shows also an expression defining the whole network. The other components are just as easy to define as the CAR.

The "movement" of the car from station to station, in this example, is no different from the "movement" of the value V modeled in Figure 10. Thus movement is not confined to values; movement of all kinds of processes among each other can be modeled in this way. The essence of the π-calculus lies in its technical management of the interplay between *restriction,* which models spatial configuration (e.g., the expression (1) for Figure 3), with the dynamic *variation* of configuration.

To summarize: There are several reasons to claim generality for the π-calculus, even in this simple form:

- It gives a direct description of systems which change their configuration, such as a mobile telephone network, or a distributed operating system with its flow of jobs and allocation of resources.
- It allows a uniform way to define data structures. Thus, it supports a process algebra like CCS as a higher level of explanation.
- There is a simple translation of the λ-calculus itself into π-calculus, which is faithful to computational behavior. Thus, it supports functional programming as a higher level of explanation.
- It provides a convenient semantic substrate for object-oriented programming and indeed other programming paradigms.
- It is amenable, like the λ-calculus, to type disciplines; this will allow us to add the taxonomies which are so important in a tractable semantic framework.

It was actually the *first,* the most concrete, of these five properties which we first strove to attain: the power to describe *mobility,* or changing configuration. This was the goal that shaped the π-calculus. It was an unsolicited delight to find that mobil-



$$Double \overset{\text{def}}{=} ax.\bar{b}(2 \times x).\ Double$$

Figure 7. A doubling process in CCS

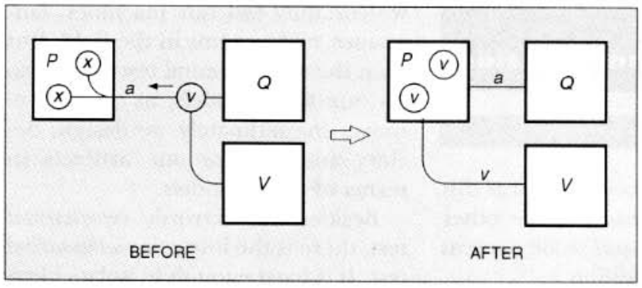


Figure 8. The π-calculus interaction $ax.P[x] \mid V \mid \bar{a}v.Q \to P[v] \mid V \mid Q$

variables $x, y, z, \ldots$

terms $M ::= x$ variable

$\lambda y.M$ abstraction

$M_1(M_2)$ application

(*the occurrence of y is binding*)

basic rule of computation $(\lambda y.M_1[y])(M_2) \to M_1[M_2]$

Figure 9. The λ-calculus

names $x, y, z, \ldots$

action terms $A ::= \bar{x}z.P$ send z along x

$xy.P$ receive any y along x

terms $P ::= A_1 + \cdots + A_n$ alternative action $(n \geq 0)$

$P_1 \mid P_2$ composition

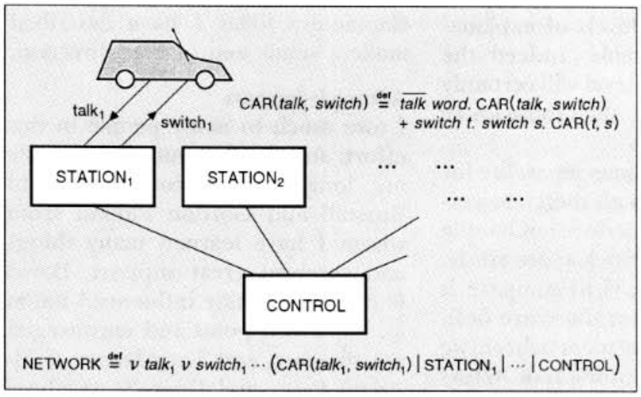
νyP restriction

$!P$ replication

(*the occurrences of y are binding*)

basic rule of computation $xy.P_1[y] \mid \bar{x}z.P_2 \to P_1[z] \mid P_2$

Figure 10. A simple form of the π-calculus



$\text{CAR}(talk, switch) \overset{\text{def}}{=} \overline{talk}\ word.\ \text{CAR}(talk, switch)$
$+\ switch\ t.\ switch\ s.\ \text{CAR}(t, s)$

$\text{NETWORK} \overset{\text{def}}{=} \nu\ talk_1\ \nu\ switch_1 \cdots (\text{CAR}(talk_1, switch_1) \mid \text{STATION}_1 \mid \cdots \mid \text{CONTROL})$

Figure 11. The mobile telephone network in the π-calculus

ity was, in fact, the key; as soon as this was technically mastered, the other properties were found to be present with no further addition.

Conclusion

I have traced a thread in the search for a basic model of concurrency, guided by a sequential paradigm. I have dared to talk about semantics because I have always hoped that the right semantic primitives would be familiar to all of us—not because we think *about* them, but because we naturally cast our thoughts *in terms of* them. Whether or not the primitives I have discussed are the right ones, they are certainly of that familiar nature.

You may have been offended by my persistent *reductionism;* again and again I have explained one thing in terms of another (a value is "just" a process, and so on). I can see no other way to reach something which is both general and tractable. But let me repeat: Many levels of explanation are indispensable. Indeed the entities at a higher level will certainly be of greater variety than those lower down.

I have claimed some generality for the π-calculus, though there are certainly things which it does not handle directly. It needs much more study. An important task is to compare it with Hewitt's Actors; there are definite points of agreement where we have followed the intuition of Actors and also some subtle differences, such as the treatment of names. More generally, the π-calculus is a *formal* calculus, while the Actors model, in spirit closer to the approach of physics, sets out to identify the *laws* which govern the primitive concepts of interaction.

Finally, how we can assess, or test, a model of computing such as a process calculus? In an important sense computer science is indeed an experimental science, even though—as Alan Newell and Herbert Simon point out in their 1975 Turing lecture—it may not fit a narrow stereotype of the experimental method, for we certainly test our machines, languages, and systems in the field. But then the experimental test bears also on our basic models, at second remove; for ultimately we design, define, and analyze our artifacts in terms of these models.

Besides the extrinsic *experimental* test, there is the intrinsic *mathematical* test. It is hard enough to isolate ideas which are really basic to computing practice; it is even harder to find those which also admit a tractable theory. It is precisely the conflict between these two tests which underlies the dialectic method which I have illustrated. I have tried to show how a distillation from practice, guided by an established logical paradigm, can yield a sharply defined candidate theory which must now be submitted to deeper mathematical and deeper experimental tests.

I believe that computer science differs little from physics, in this general scientific method, even if not in its experimental criteria. Like many computer scientists, I hope for a broad *informatical* science of phenomena—both manmade and natural—to match the rich existing *physical* science. I shall be happy if the elementary ideas I have described make a small step in that direction.

Acknowledgments

I owe much to many people in this effort, and thank them all; especially my long-standing colleagues Rod Burstall and Gordon Plotkin from whom I have learned many things and received great support, David Park who strongly influenced me at an important point and encouraged me all along, and Tony Hoare, Carl-Adam Petri, and Dana Scott whose work has been an inspiration to me.

References

1. Abramsky, S. A domain equation for bisimulation. *J. Inf. Comput. 92* (1991), 161–218.
2. Agha, G.A. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.
3. Astesiano, E. and Zucca, E. Parametric channels via label expressions in CCS. *J. Theor. Comput. Sci. 33* (1984), 45–64.
4. Baeten, J.C.M. and Weijland, W.P. *Process Algebra.* Cambridge University Press, Cambridge, Mass., 1990.
5. Barendregt, H.P. *The Lambda Calculus.* North Holland, Amsterdam, 1981.
6. Böhm, C. and Berarducci, A. Automatic synthesis of typed Λ-programs on term algebras. *J. Theor. Comput. Sci. 39* (1985), 135–154.
7. Brinksma, E. On the design of Extended LOTOS, A specification language for open distributed systems. Ph.D. dissertation, Univ. of Twente, 1988.
8. Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. A theory of communicating sequential processes. *J. ACM 31* (1984), 560–599.
9. Church, A. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, N.J., 1946.
10. Engberg, U. and Nielsen, M. A calculus of communicating systems with label passing. Rep. DAIMI PB-208, Computer Science Dept., Univ. of Århus, Århus, Denmark, 1986.
11. Girard, J.-Y. The system F of variable types, fifteen years later. *J. Theor. Comput. Sci. 45* (1986), 159–192.
12. Goldber, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.
13. Gunter, C.A. and Scott, D.S. Semantic domains. In *Handbook of Theoretical Computer Science,* vol A. Elsevier, New York, 1990, pp. 633–674.
14. Hennessy, M. *Algebraic Theory of Processes.* MIT Press, Cambridge, Mass., 1988.
15. Hewitt, C.E., Bishop, P., and Steiger, R. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence.* 1973, pp. 235–245.
16. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21* (1978), 666–677.
17. Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1985.
18. Honda, K. and Tokoro, M. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming.* Lecture Notes in Computer Science, vol. 512. Springer-Verlag, New York, 1991, pp. 133–147.
19. Kennaway, J.R. and Sleep, M.R. Syntax and informal semantics of DyNe, a parallel language. In *Lecture Notes in Computer Science,* vol. 207. Springer-Verlag, New York, 1985, pp. 222–

230.

20. Milner, R. Flow graphs and flow algebras. *ACM 26*, 4 (Oct. 1979), 794–818.

21. Milner, R. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York, 1980.

22. Milner, R. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

23. Milner, R. The polyadic π-calculus: A tutorial. Res. Rep. LFCS-91-180, Lab. for Foundations of Computer Science, Computer Science Dept., Edinburgh Univ., 1991.

24. Milner, R. Functions as processes. Res. Rep. No. 1154, INRIA, Sophia Antipolis, 1990.

25. Milner, R., Parrow, J. and Walker, D. A calculus of mobile processes. Rep. ECS-LFCS-89-85 and -86, Lab. for Foundations of Computer Science, Computer Science Dept., Edinburgh Univ., 1989.

26. Newell, A. and Simon, H.A. Computer science as empirical enquiry: Symbols and search. *Commun. ACM 19* (1976), 113–126.

27. Nielson, F. The typed λ-calculus with first-class processes. In *Proceedings of PARLE 89.* Lecture Notes in Computer Science, vol. 366. Springer-Verlag, New York, 1989.

28. Nielsen, M., Plotkin, G.D. and Winskel, G. Petri nets, event structures and domains. *J. Theor. Comput. Sci. 13* (1981).

29. Orava, F. and Parrow, J. An algebraic verification of a mobile network. Internal Rep., SICS, Sweden.

30. Park, D.M.R. Concurrency and automata on infinite sequences. In *Lecture Notes in Computer Science,* vol. 104. Springer-Verlag, New York, 1980.

31. Petri, C.A. *Communikation mit Automaten.* Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962. In German.

32. Plotkin, G.D. A powerdomain construction. *SIAM J. Comput. 5* (1976), 452–487.

33. Reisig, W. *Petri Nets.* EATCS Monographs on Theoretical Computer Science, W. Brauer, G. Rozenberg, A. Salomaa, Eds. Springer-Verlag, New York, 1983.

34. Reynolds, J.C. Towards a theory of type structure. In *Lecture Notes in Computer Science,* vol. 19. Springer-Verlag, New York, 1974, pp. 408–425.

35. Sangiorgi, D. Forthcoming Ph.D. thesis. Univ. of Edinburgh, 1992.

36. Tennent, R.D. *Principles of Programming Languages.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

37. Thomsen, B. Calculi for higher-order communicating systems. Ph.D. dissertation, Imperial College, London Univ., 1990.

38. Walker, D.J. π-calculus semantics of object-oriented programming languages. In *Proceedings of the Conference on Theoretical Aspects of Computer Software* (Japan). Lecture Notes in Computer Science, vol. 526. Springer-Verlag, New York, 1991, pp. 532–547.

CR Categories and Subject Descriptors: D.3.1 [**Programming languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming languages**]: Language Classifications—*concurrent, distributed and parallel languages*; *object-oriented languages*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; F.3.2 [**Logic and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*; *operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*; *functional constructs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms: Languages, Theory

Additional Keywords and Phrases: CCS, interaction, naming and reference, pi calculus, process algebra, process calculus, reduction rule

About the Author:

ROBIN MILNER is professor of computation theory in the Computer Science department at the University of Edinburgh, where he has worked for 20 years. Before that he researched for two years in the Artificial Intelligence Laboratory at Stanford University. In 1986 he became founding director of the Laboratory for Foundations of Computer Science at Edinburgh. Currently he holds a five-year senior research fellowship from the UK Science and Engineering Research Council.