Turing, Computing and Communication

Robin Milner

Cambridge University, Cambridge, United Kingdom

Summary. This essay is a slightly edited transcription of a lecture given in 1997 in King's College, Cambridge, where Alan Turing had been a Fellow. The lecture was part of a meeting to celebrate the 60th anniversary of the publication of Turing's paper *On computable numbers, with an application to the Entscheidungsproblem*, published in the *Proceedings of the London Mathematical Society* in 1937.

1 Introduction

How has computer science developed since Turing's founding ideas? His thinking bore strongly both upon the possibility of mechanical intelligence and upon logical foundations. One cannot do justice to both in a short lecture, and I shall continue the discussion of logical foundations begun in the previous lecture.

Physical stored-program computers came to exist some ten years after Turing's paper on the *entscheidungsproblem*, notably with the EDSAC in the Cambridge Mathematical Laboratory in 1949, under the leadership of Maurice Wilkes; a great engineering achievement. Thus logic and engineering are the two foundation stones of computer science; our constructions rest firmly on both foundations, and thereby strengthen both. I shall discuss how the logical foundation has developed through practical experience.

My thesis is that this logical foundation has changed a lot since Turing, but harks back to him. To be more precise:

THESIS:

1 Computing has grown into *informatics*, the science of interactive systems.

2 Turing's logical computing machines are matched by a *logic of interaction.*

My message is that we must develop this logical theory, partly because otherwise the interactive systems which we build, or which just happen, will escape our understanding and the consequences may be serious, and partly because it is a new scientific challenge. Besides, it has all the charm of inventing the science of navigation while already onboard ship.

2 Concepts in Computer Science

In natural science, concepts arise from the urge to understand observed phenomena. But in computer science, concepts arise as distillations of our design of systems. This is immediately evident in Turing's work, most strikingly with the concept of a *universal logical computing machine*.

By 1937 there was already a rich repertoire of computational procedures. Typically they involved a hand calculating machine and a schematic use of paper in solving, say, a type of differential equation following a specific algorithm. Turing's class of logical computing machines—which he also called "paper machines"—was surely distilled from this repertoire of procedures. But he distilled more, namely the idea of a *universal* paper machine which can analyse and manipulate descriptions of members of the class, even of itself. This demonstrated the logical possibility of the general-purpose stored-program computer.

Turing also, among others, distilled the idea of the *subroutine* in computing. The distillation of this idea was a continuing affair, and didn't happen all at once. Turing's term for subroutine was "subsidiary operation"; anyone familiar with numerical methods must have known exactly what that meant when referring to humanly performed operations.

A concept rarely stands clear unless it has been reached from different angles. The gene is a prime example; it was seen first logically, then physically. So each computer design, whether logical or—like the EDSAC—physical, was a step in the distillation of the notion of subroutine. The distillation continued with the notion of *parametric procedure* in high-level programming languages such as ALGOL, where the humble subroutine was endowed with a rich taxonomy which might have surprised Turing himself. Each high-level language is, at least, a universal paper machine; but each one also expresses higher-level concepts distilled from practice.

In modern computing we build and analyse huge systems, equal in complexity to many systems found in nature—e.g., an ecology. So in computing, as in natural science, there must be many levels of description. Computer science has its organisms, its molecules and its elementary particles—its biology, chemistry and physics:

Levels of Description

Natural Science		Computer Science
Biology	ORGANISMS	Databases, networks, ...
Chemistry	MOLECULES	Metaphors of programming
Physics	PARTICLES (ELEMENTS)	Primitives of programming

At the level of organism we find, for example, species of database and network, each with a conceptual armoury. At the level of molecule we find the metaphors, like parametric procedure, provided by programming languages. At the particle level we find—as it were—the most basic parts of speech. (I make no apology for talking so much in terms of language. Computers like screwdrivers are prosthetic devices, but the means to control them is linguistic, not muscular.) The best of these parts of speech and the best of the metaphors become accepted modes of thought; that is, they become concepts.

3 From Metaphor to Concept

I shall now discuss a couple of molecular concepts or metaphors, distilled over the last thirty years, in which the notion of interaction is prominent.

There is a Babel of programming languages. This is not surprising; much of the world we live in can be modelled, analysed or controlled by program, and each application domain has its own structure. But sometimes a central idea finds its first clear expression in a language designed for a particular problem domain. Such was the case with the problem domain of *simulation*.

In the 1960s there was a great vogue in simulation languages. New ones kept emerging. They all gave you ways of making queues of things (in the process which you wished to simulate), giving objects attributes which would determine how long it took to process them, giving agents attributes to determine what things they could process, tossing coins to make it random, and recording what happened in a histogram. These languages usually did not last; one can simulate so many real-world processes that no single genre of language can cover them all. So simulation languages merged into the general stream.

But not without effect. One of them highlighted a new metaphor: the notion of a community of agents all *doing* things to each other, each persisting in time but changing state. This is the notion known to programmers as an *object*, possessing its own state and its repertoire of activities, or so-called *methods*; it is now so famous that even non-programmers have heard of it. It originated in the simulation language known as Simula, invented by Ole-Johann Dahl and Kristen Nygaard. *Object-oriented programming* is now a widely accepted metaphor used in applications which have nothing to do with simulation. So the abstract notion of agent or active object, from being a

convenient metaphor, is graduating to the status of a concept in computer science.

Even more fundamental to computing, at the molecular level, is the time-honoured concept of *algorithm*. Until quite recently it could be defined no better than "the kind of process enacted by a computer program", which is no help at all if we are trying to understand what computational processes are! But recently algorithms have come to be characterized precisely as *game-theoretic interactions*. We could hardly wish for better evidence that the notion of interaction is basic to computer science.

4 Concurrent Processes

The notion of *agent* or *active object* brings programming ontology—if you like, the metaphors programmers use in design—much closer to the real world. So why, you may ask, did we not *always* write programs in terms of interactive agents? The answer lies partly in von Neumann's so-called bottleneck, and I want to describe this before I talk about new parts of speech, or elements.

The early computers all followed the model of John von Neumann, in which—as far as the programmer was concerned—only one thing could happen at once; at any given time only one agent could be active. So the possibility of *concurrent activity* or even *co-existence* of such agents could not be expressed in a program—even though underneath, as it were in the machine's subconscious, many wheels would whirr and circuits cycle simultaneously. One can speculate why this sequential discipline was adopted. The familiar calculational procedures, which computers were designed to relieve us of, were all inherently sequential; not at all like cooking recipes which ask you to conduct several processes at once—for example, to slice the beans *while* the water is coming to the boil. This in turn may be because our conscious thought process is sequential; we have so little short term memory that we can't easily think of more than one thing at once.

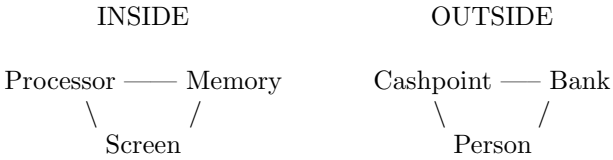
The bursting of von Neumann's bottleneck is due in part to the premature birth and later triumph of the metaphor of object-oriented programming. But a river never breaks its banks in one place. In the 1960s and 1970s the designers of computer operating systems, people like Edsgar Dijkstra and Tony Hoare, were ill-content with sequential programming metaphors. Programming in the von Neumann model was too much like a child's construction kit; you can build the lorry but you can't build the engine. Consider several programs running simultaneously inside a computer. They may only *appear* to run simultaneously, by virtue of time-slicing, but in any case you need to write the master program—the so-called operating system—which controls them all by interacting with them. This is not *sequential* but *concurrent* activity; you need new language to express concurrent activity, and new theory for it. You cannot decently express it as a metaphor in a sequential language.

Indeed, in the same period, Carl-Adam Petri developed a new model of concurrent processes not only to describe computational behaviour, but also to model office information systems. He was among the first to point out that concurrency is the norm, not the exception.

What this amounts to is that computer scientists began to invent new parts of speech, new elements, to express the metaphors suitable for interactive concurrent systems.

5 The Old and the New Computer Science

The first part of my thesis was that the river of computer science has indeed burst its von Neumann banks, and has become a structural theory of interaction. I call it *informatics* here; I don't know a better word which is as free of misleading connotation. It goes far beyond describing what programs do; it claims that the kind of interactions which go on under the bonnet of a sequential program are no different from those which occur —even involving human components— in the world outside. For example, we have no need to describe these two systems in different terms, if we are thinking of information-flow:



Thus software, from being a *prescription* for how to do something —in Turing's terms a "list of instructions"— becomes much more akin to a *description* of behaviour, not only programmed *on* a computer, but occurring by hap or design *inside* or *outside* it. Here is a set of contrasts, distinguishing the old computer science as a limiting case of the new:

Old Computing		New Computing
Prescription	···	Description
Hierarchical design	···	Heterarchical phenomena
Determinism	···	Nondeterminism
End-result	···	Continuing interaction
(Extension)		(Intension)

Take the first line: Software no longer just *prescribes* behaviour to take place inside a computer; instead, it *describes* information flow in wider systems.

Take the second line: We can no longer confine ourselves to systems which are neatly organised, like an army with colonels and platoons. Consider the Internet; it is a linkage of autonomous agents, more of an informatic rabble than an army. Of course we built many of its parts; but the whole is a heterarchical assembly—something of a *natural* phenomenon.

Take the third line: We can never know enough about an assembly of autonomous agents to predict each twist in its behaviour. We have to take nondeterminism as elementary, not just temporary laziness which we can amend later by supplying values for all the hidden variables.

Take the fourth line: The meaning of a conventional computer program, as far as a user is concerned, is just the mathematical function it evaluates. But we users are *inside* our interactive systems; we care about what continually goes on. The meaning surely lies in the whole conversation, not just its end-result. (Indeed there may be no end-result, since there may have been no goal.)

Now, here are some sharper contrasts which hint at what might be the elements of a mathematical theory of interactive systems:

	Computation	Interaction
ACTIVE ENTITY P:	program	active object, agent
ITS MEANING:	function	process
STATICS (COMBINATION):	sequential composition $P_1; P_2$	parallel composition $P_1 \parallel P_2$
DYNAMICS (ACTION):	operate on datum	send/receive message

In the first line, note especially that all *programs* are prescriptive—they are designed with a purpose; *agents* need be neither designed nor purposeful. As for *meanings*, there is a big knowledge gap; we have an impressive mathematical theory of functions, but we still have no consensus on a corresponding theory of discrete processes. (Of course we are working on it.) The *composition* of programs emphasizes the sequentiality imposed by the designer; but in interactive systems everything can happen as soon as the interactions which trigger it have occurred. Finally, concerning *action*, note the asymmetry in computation between an active operator and a passive operand; in an interactive system, messages pass between active peers.

6 Elements of Interaction

Now, what are the new particles —parts of speech, or elements— which allow one to express interaction? They lie at the same elementary level as the operation of a Turing machine on its tape, but they differ. For much longer than the reign of modern computers, the basic idiom of algorithm has been the asymmetric, hierarchical notion of operator acting on operand. But this does not suffice to express interaction between agents as peers; worse, it locks the mind away from the proper mode of thought.

So we must find an elementary model which does for *interaction* what Turing's logical machines do for *computation*. The second part of my thesis was that there is a logic of informatic action, and in my view it is based upon two fundamental elements:

Logical Elements of Interaction
Synchronized action
Channel, or vocative name

These two fit together perfectly; indeed, like quarks, they hardly exist apart. Synchronization is between an action—the vocative use of a name—by one agent, and a reaction by another. At this level, *names* and *channels* are the same thing; in fact, they are the essence of several superficially different things which computer scientists have called *links*, *pointers*, *references*, *identifiers*, *addresses*, ..., and so on. These elements seem slight in themselves, but they serve to unify our theory; they can form the basis of a logical calculus not only for traditional computation but for the wider range of interactive systems.

There are many systems of increasing importance in our lives which show the pervasive role played by naming and synchronized action. We don't have to look far for an example; consider simply a document—not a paper copy, but the virtual kind that exists on the Internet:

- A piece of hypertext representing a document exists nowhere in linear form. It's a mass of pointers, or names, which link its parts in a tree-like way.
- But it does not stop at tree-like structures. Parts of the document will be links into other structures; many links to one structure, for economy.
- When you "click" on such a link, you synchronize your action with an action by the document.
- It does not stop at static structures; some links may command a translation or even a summarization of the text-agent which they call.
- Not all parts reside at one site; some parts may lie across the Atlantic.
- It does not stop at textual structures. Some links will call up animated pictures, others will provide exercises for the reader, games to play, and so on.

All this, just starting from the notion of a document! The web will be much more tangled for other applications. But the point is that you don't just *read* a document like this—you *interact* with it.

I ask you to think of the term "information" actively, as the *activity of informing*. An atomic message, then, is not a passive datum but an action which synchronizes two agents. Our example of active documents has shown that the active/passive polarization between operator and operand, between process and data, is no longer realistic—and we have removed this limitation.

7 Reflection: Back to Turing

We have briefly explored what computer science has become, having been launched logically by Turing, and physically by the earliest computers. The technological story is of course a marvel, and has been a prerequisite for the

informatic story, which is what concerns us here. To summarize: Turing's *paper machines* have evolved into the kind of informatic web in which we now live. They are truly virtual, not physical; they are webs of naming, calling, migrating in a sense which has little to do with where they reside, or with how they are physically represented.

Can we ask about these webs the kind of question Turing asked about his paper machines? Both Turing machines and informatic webs are what Herbert Simon and Allen Newell have called *symbol systems*. In each class of symbol systems, one can ask whether a member of the class can represent and manipulate some property of the class itself. Such a phenomenon is called *reflection*. In particular, consider the following:

- A computing entity can *compute a means of computing* (consider the universal Turing machine).
- Can a cognitive entity *know about knowing*?
- Can a learning entity *learn how to learn*?

... and so on. If the answer is "yes", we are inclined to think that the class of entities is properly adult, has come of age. Consider then:

- Can a communicating entity *communicate a means of communicating*?

This question differs intriguingly from the one about computing entities, because it concerns systems of agents in a heterarchy. In a heterarchy you cannot manipulate another agent, in the sense that a universal Turing machine interprets another. The concept of a universal Turing machine relies on a sharp distinction between passive data (e.g., the description of a machine) and active agent (e.g., the machine itself), and I have made a case for eroding this distinction. But in an interactive system you can, by communicating with your neighbour, acquire new links and relinquish old ones. So distributed computing is also adult, in the above sense. In our informatic webs, agents can acquire new contacts by link-manipulation, and so realize new forms of behaviour. That is, a web can spin itself.

To conclude: I believe that computing has evolved in a direction which would excite Alan Turing. His search for primitives continues to inspire our search. He would surely agree that these primitives must relate to computing practice, since he himself spent much effort on plans to build a physical computer, the ACE, not just logical ones. In the same way, but in a wider sense, our primitives relate to informatic practice. So I shall be sorry if computer science ever flies apart into two disciplines, one theoretical and one technological. We are back to our two foundation stones, logic and engineering; among all his other legacies, Turing embodies the wisdom of arching between them.