50 Years of Turing Award

S P Suresh
October 20, 2017
Chennai Mathematical Institute
http://www.cmi.ac.in/~spsuresh

Robin Milner

Biography

- **Arthur John Robin Gorell Milner**
  (13 Jan 1934 – 20 Mar 2010)
- Born into a military family in Yealmpton, England
- Scholarship to Eton College (1947)
- Served in the Royal Engineers
- Graduated in 1957 from King's College, Cambridge

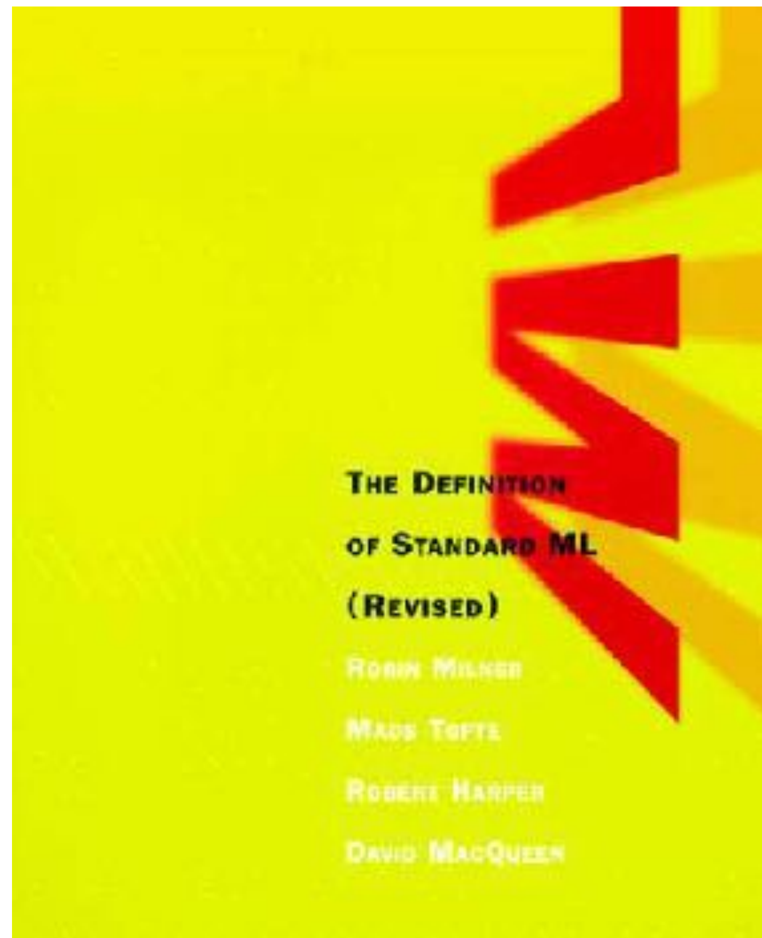
Academic career

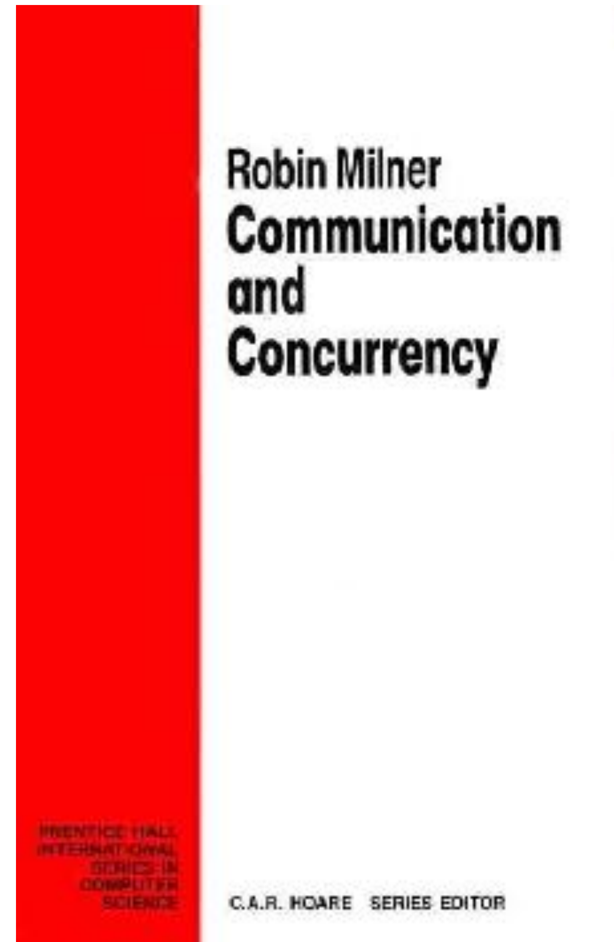
- Worked as a schoolteacher
- Programmer at Ferranti
- Various positions
  - City University, London
  - Swansea University
  - Stanford University
- University of Edinburgh (from 1973)
  - Co-founded Laboratory for Foundations of Computer Science

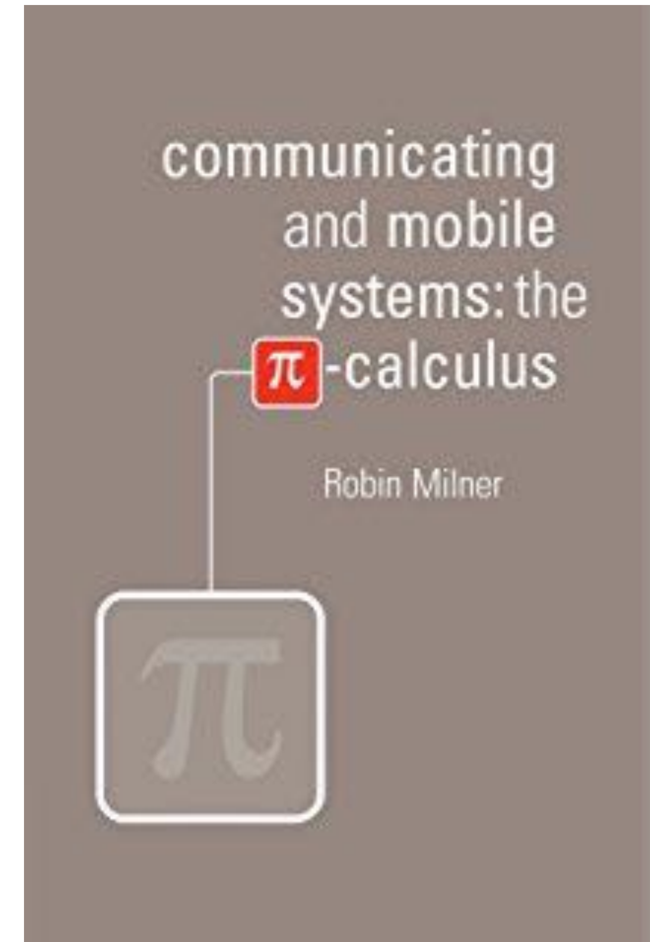
Contributions to CS

- Developer of LCF, an interactive automated theorem prover
- Developed ML, a language for LCF
  - polymorphic type inference
  - type-safe exception handling
  - Hindley-Milner type system
- Analysis of concurrent systems
  - Calculus of communicating systems (CCS)
  - π-calculus
- Formulated full abstraction

Influential classics







Honors and Awards

- Fellow of the Royal Society
- Distinguished Fellow of the British Computer Society
- ACM Turing Award winner (1991)

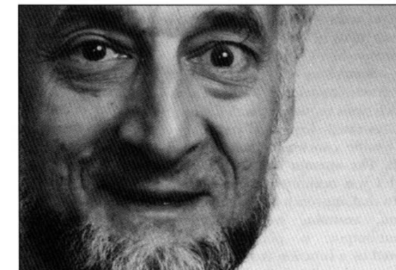
Turing award citation

- Mentions all four contributions (LCF, ML, CCS, full abstraction)
- Influence of lectures by Christopher Strachey and Dana Scott, on denotational semantics and LCF
- Work with John McCarthy's AI project
- Details of implementing LCF, and birth of ML (meta language)
- Never studied for a PhD, but got an honorary doctorate in 1997

Elements of Interaction

Turing Award Lecture

Robin Milner



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.

Type inference

- Untyped languages are extremely convenient
  - `procedure sort:`
    - `sort([1,4,5,3,2])`
    - `sort("cedbaf")`
- But they are also extremely unsafe
  - ```
    x = True;
    x + 5;
    ```
  - Compiles fine but is a run-time error
    ```

Type inference

- Static typing guarantees safety
  - Type errors caught at compile time itself
- But is irritating
  - Cannot have the same `sort` routine work for integer arrays as well as strings
  - Have to annotate all types explicitly

Type inference

- Type inference offers the best of both worlds – safety and convenience

- ```
  sort [] = []
  sort [x] = [x]
  sort (x:xs) = sort [y <- xs | y <=x]
                ++ [x]
                ++ sort [y <-xs | y > x]
  ```

- ```
  sort [1,3,5,4,2]
  sort "becdaf"
  ```

- But `sort True` will not even compile

- How is this possible, even though the programs do not mention any types at all?

Type inference

- Programming languages like ML and Haskell infer types
- Key feature in many other modern programming languages (Go, Rust, Swift, …)
- All typing errors are caught at compile time
  - Programmers not required to specify types

Examples

- What is the type of `plus` defined by

```
plus x y = x + y
```

- `+` is only applicable on numbers
  - So infer that `x` and `y` are of type `Int`, and `plus` is of type

```
Int -> Int -> Int
```

Examples

- What is the type of `twice` defined by

```
twice f x = f (f x)
```

- `x` could be anything
- f is being applied to x, so it is a function
- `f` is also applied to `f x`, so the value returned has the same type as argument
- So `f :: a -> a` and `twice :: (a -> a) -> a -> a`
- Function is higher order and polymorphic

Examples

- Polymorphism:

```
f = let id x = x in
        (id not) (id True)
```

- Here `id` has type `a -> a`
    - In `id True`, we take `a = Bool`
    - In `id not`, we take `a = Bool -> Bool`
- Functions can also be recursive

```
f = let y g = g (y g) in
        y (...)
```

- `y :: (a -> a) -> a`

Damas-Milner algorithm

- Described in a 1982 paper by Damas & Milner
- Very efficient algorithm that handles a rich variety of programs
- Fundamental in the study of typed lambda calculus – principal type schemes (Roger Hindley)
- Has a deep and lasting influence on PL design

Strand 2: CCS

- A calculus of concurrency and interaction
- Arose from trying to extend denotational semantics to concurrent programs / systems

Concurrency and composition

- Consider the programs **P1** and **P2**

```
P1: x := 1; x := x + 1
P2: x := 2
```

- They compute the same function – hence same denotation
- But consider `P1 || Q` and `P2 || Q`, where `Q` is

```
Q: x := 3
```

- They have different behaviors, because of interference
- Hence their denotation can be composed from that of their components
- Need a more refined compositional semantics

Concurrency and composition

- View each of `P1` and `P2` in more detail
- Do not view memory as passive, and program as active
- Promote everything to an entity capable of interaction
- Memory interacts via the channel `x`
- Study interactions rather than states and state transformations

Synchronized interaction

- Central rule:

$$a(x).P(x) \mid \underline{a}V.Q \;\text{-}\!> \; P(V) \mid Q$$

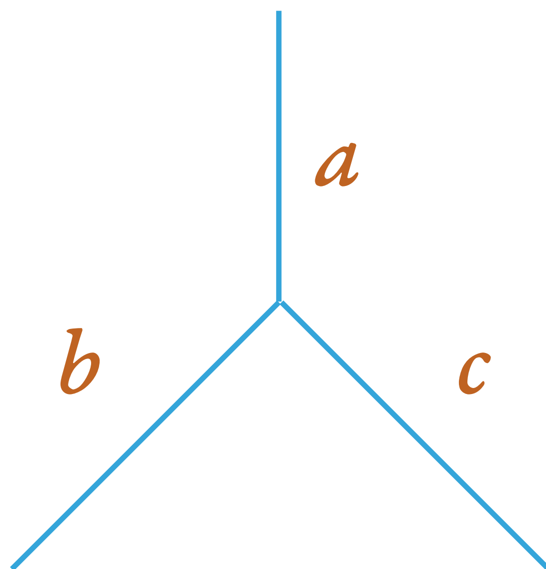
- Also proposed by C.A.R. Hoare (CSP)

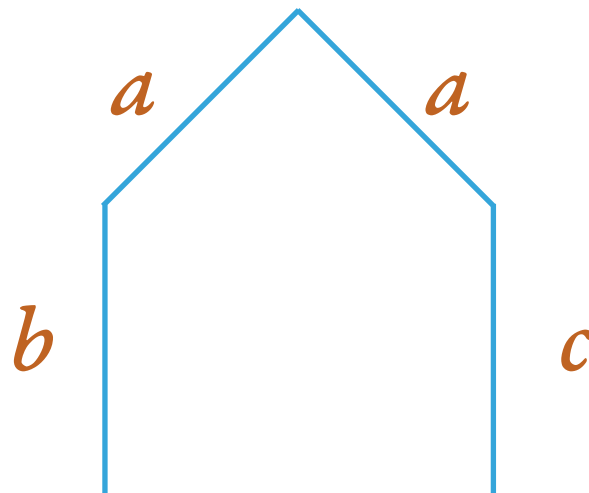
More CCS

- CCS provides a compositional syntax for describing processes (of the same status) that interact with each other via channels
- Allows parallel and nondeterministic composition
- Devices for managing names to be public / private etc.
- π-calculus builds on this with more operations on channels – useful for modeling mobile processes

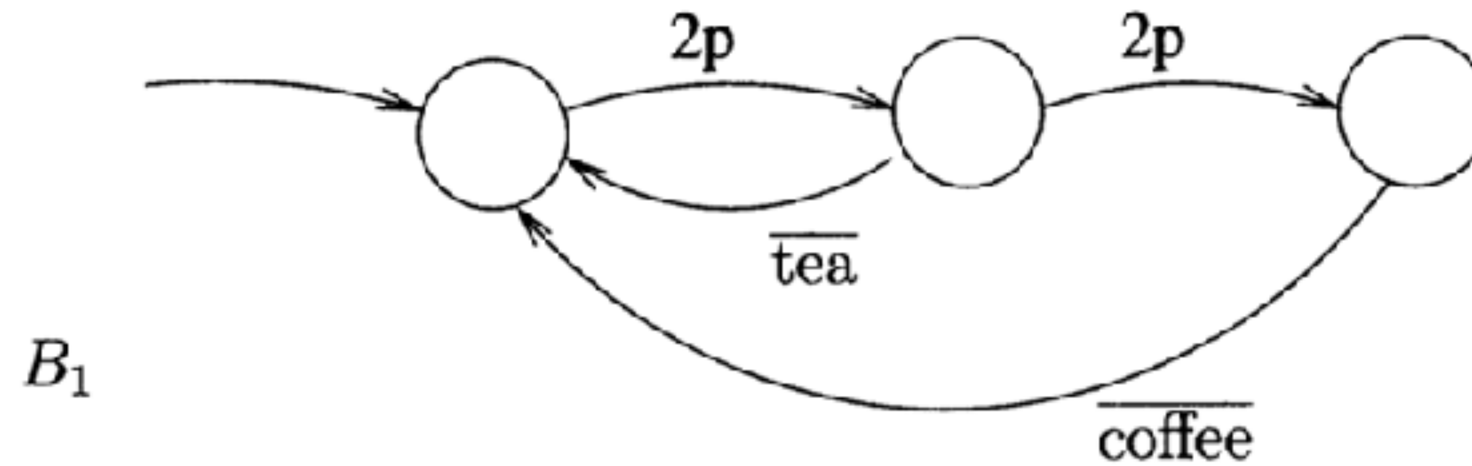
Observational equivalence

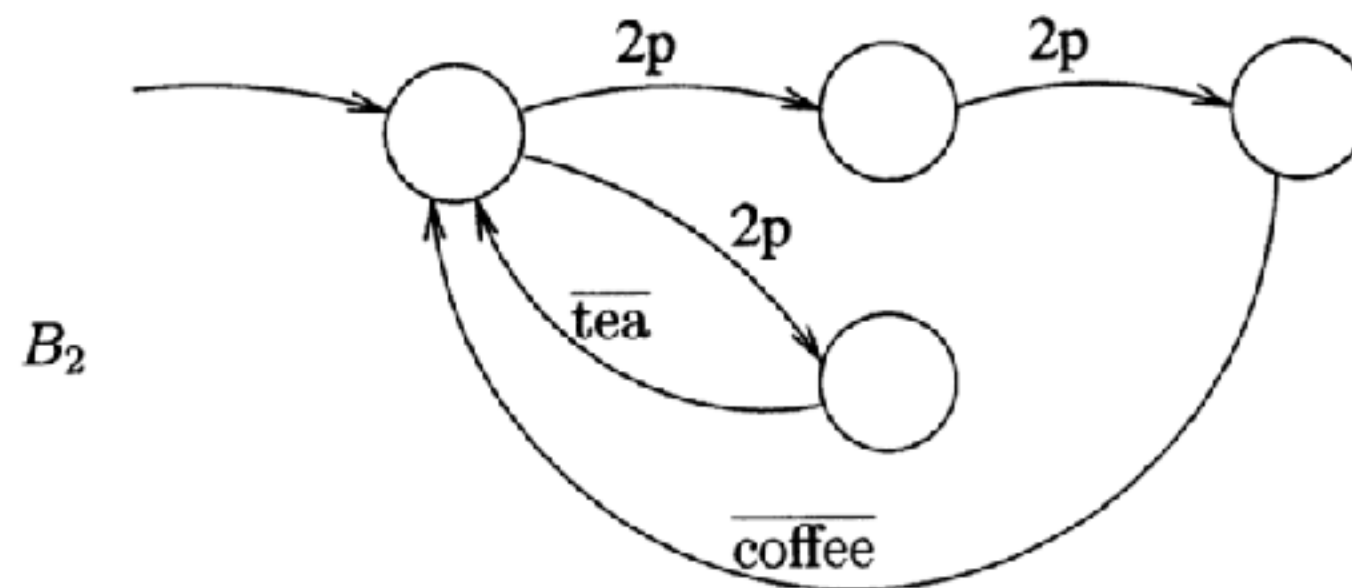
- When do we deem two processes to be equal? When they display the same patterns of interactions





Vending machines





Vending machines

- $L(B_1) = L(B_2)$
- But B_2 displays an irritating amount of autonomy!
- The point is:

$$2\mathrm{p} \cdot (\overline{\mathrm{tea}} + 2\mathrm{p} \cdot \overline{\mathrm{coffee}}) = 2\mathrm{p} \cdot \overline{\mathrm{tea}} + 2\mathrm{p} \cdot 2\mathrm{p} \cdot \overline{\mathrm{coffee}}$$

is a valid equation on languages, but the two sides do not represent the same pattern of interaction

Observational equivalence

- Central semantic notion in CCS – now a central notion in CS
- Shortly after publishing his paper on CCS, Milner considerably reworked this based on the notion of bisimulation (David Parks)
- Extremely rich concept that has given rise to a variety of refinements

Legacy

- Groundbreaking work in multiple areas
- Fundamental concepts that touch the lives of many programmers and theoreticians for many generations
- Not just new concepts, but formulated complete theories
- From an interview a few weeks before his death:
  I would dearly love to follow this kind of work through to the front line of design and experience, but I don't think I'll be around for long enough. I'm making up for it by listening and talking to those who will be!