# MICHAEL JOHN CALDWELL GORDON

28 February 1948—22 August 2017

*Biogr. Mems Fell. R. Soc.* **65**, 89–113 (2018)

Michael GC Gordon

# MICHAEL JOHN CALDWELL GORDON

28 February 1948—22 August 2017

Elected FRS 1994

By Lawrence C. Paulson FRS*

*Computer Laboratory, University of Cambridge, Cambridge, UK*

Michael Gordon was a pioneer in the field of interactive theorem proving and hardware verification. In the 1970s, he had the vision of formally verifying system designs, proving their correctness using mathematics and logic. He demonstrated his ideas on real-world computer designs. His students extended the work to such diverse areas as the verification of floating-point algorithms, the verification of probabilistic algorithms and the verified translation of source code to correct machine code. He was elected to the Royal Society in 1994, and he continued to produce outstanding research until retirement.

His achievements include his work at Edinburgh University helping to create Edinburgh LCF, the first interactive theorem prover of its kind, and the ML family of functional programming languages. He adopted higher-order logic as a general formalism for verification, showing that it could specify hardware designs from the gate level right up to the processor level. It turned out to be an ideal formalism for many problems in computer science and mathematics. His tools and techniques have exerted a huge influence across the field of formal verification.

## Early life

Mike Gordon was born in Ripon, Yorkshire, to John Gordon and Daphne Mavis Gordon (née More). He had perhaps a lonely childhood: he was an only child, and his father committed suicide when Mike was eight years old. His mother sent him as a boarding pupil first to 'the notorious Dartington Hall' (where he forgot how to read) and then to Bedales school, which he regarded 'as being my home between the ages of 8 and 18' (33). Bedales was then a mixed, progressive school specializing in the arts.

*lp15@cam.ac.uk

http://dx.doi.org/10.1098/rsbm.2018.0019 Published by the Royal Society

Mike was a quiet pupil but showed early signs of a lively, scientific mind. He built model aeroplanes, some petrol powered and radio controlled. Once he slipped into the chemistry lab and synthesized methyl mercaptan, to impress his friends with its terrible smell. On another occasion, he made nitrogen triiodide crystals—which explode when stepped on—and sprinkled them in the library.[1] Pupils called him Gecko because of his bright, prominent eyes and surprised expression: a look he never lost.[2]

In 1966, Mike was accepted to the University of Cambridge to study engineering. As preparation, he took a gap year as a management trainee at the North Thames Gas Board (34). This was his first exposure to the real world after a childhood spent at boarding school, and it came as a shock. The staff were divided on class lines, white coats for the management and brown coats for the workers, with separate toilets and canteens. He observed time and motion studies and the compilation of tables listing, for example, 'how long it would take to put a single screw into a wall for different screw sizes'; these data would then be used to set deadlines for workers. He liked to joke about this system, but he clearly saw it as wasteful and oppressive. He spent much of his time at the Beckton Gas Works: a vast, bleak and partly derelict site that would later become the shattered city of Hué in Stanley Kubrick's Vietnam war movie, *Full Metal Jacket*.

Mike's gap year experience destroyed his enthusiasm for engineering. However, during this time he stumbled upon symbolic logic, buying logic books to read while commuting between home and the Beckton Gas Works, and so he decided to study mathematics as 'the furthest subject from engineering that didn't involve writing essays'. Initially he struggled with mathematics (his subject change would be forbidden today), but he improved year after year and eventually graduated with a First (35):

> Although I found the course very tough, it gave me the tools and confidence to feel that with sufficient effort … I could master any mathematical material I needed. This laid a solid foundation for my subsequent academic career.

Mike's first exposure to computers came in 1969, after his second year at Cambridge, when he took a summer job at the National Physical Laboratory (NPL) (36). He learnt how to boot up a Honeywell DDP-516 by manually keying in a loader using switches, and to load machine code via paper tape. This machine was likely the inspiration for the 16-bit minicomputer that Mike designed later as the canonical example for his verification techniques. He worked on pattern recognition, writing code to identify printed characters by testing for specific features. Today, machine learning is invariably used for such tasks, and in fact Mike wrote a final year essay on perceptrons, a primitive type of neural network. This experience lured Mike to the University of Edinburgh School of Artificial Intelligence, where he ultimately specialized in programming language theory.

## Research milieu: verification and semantics

Computer programming has been plagued by errors from the earliest days. Ideas for verifying programs mathematically proliferated during the 1960s. Robert Floyd proposed a

[1] Simon Laughlin, personal communication, 8 February 2018.

[2] Stephen Levinson, personal communication, 17 January 2018.

methodology for attaching and verifying logical assertions within flowcharts (Floyd 1967). In a landmark paper (Hoare 1989), C. A. R. Hoare (FRS 1982) proposed a similar technique, but taking the form of a novel logical calculus combining program fragments and mathematical assertions. It worked beautifully, at least on small examples.

This technique was a form of *programming language semantics*: a precise specification of the meaning of every construct of a given programming language. For example, consider the program fragment `A+B`, for computing the sum of the values of `A` and `B`, two computable expressions. What happens if the sum is too large to be represented on the computer? What if `B`, although nonzero, is much smaller than `A`, so precision is lost and `A+B` turns out to equal `A`? Further complications arise if evaluating `A` and `B` causes side effects, such as writing to memory; then there is no reason why `A+B` should equal `B+A` or why `A+A` should equal `2*A`. For another example, suppose we have a vector `V` whose components are `V[1]`, ..., `V[`$n$`]`, and consider a command to copy data into `V`. If more than $n$ elements are supplied, then they may get copied into an arbitrary part of memory. This is the classic buffer overflow error, which has caused innumerable security vulnerabilities. One remedy for such issues is to precisely specify the semantics of every programming language construct so that ambiguities and vulnerabilities can be identified and eliminated.

During the 1960s, Dana Scott and Christopher Strachey were developing the *denotational* approach to semantics (Scott 1970). This involves defining functions mapping programming constructs such as expressions, statements and types into suitable mathematical domains. A key idea is the use of *partial orderings* to deal with non-termination. For example, if $f$ and $g$ are computable partial functions on the natural numbers, then $f \sqsubseteq g$ means that for all $x$, if $f(x)$ is defined then $g(x) = f(x)$, and we say '$f$ approximates $g$'. That idea came from recursive function theory. But once we accept that not everything is a number and grasp the need for functions themselves to be values, this simplifies to $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x$. Basic domains like the natural numbers are made into partial orderings by affixing a 'bottom element' $\perp$, with $\perp \sqsubseteq n$ for every natural number $n$. Domain theory requires functions to be *monotonic*—if $x \sqsubseteq y$ then $f(x) \sqsubseteq f(y)$. The intuition is that a computable function cannot know that its argument is failing to terminate, and can never do more with less. Functions must also be *continuous* (limit-preserving). The intuition is that an infinite computation delivers nothing more than the results of successive finite computations. Sometimes called *fixed-point theory*, these techniques could specify the semantics of any recursive function definition.

Scott's Oxford technical report (Scott 1970)—still rewarding to read—outlined this mathematically sophisticated and elegant approach. It set off a frenzy of activity. Researchers strove to extend and simplify Scott and Strachey's highly abstruse techniques, while relating them to Hoare logic on the one hand and to more intuitive semantic notions on the other.

Denotational semantics makes heavy use of the λ-*calculus* (Barendregt 1984): a tiny, primitive language of functions. Terms of the λ-calculus include

- *variables* $x, y, z, \ldots$
- *abstractions* $(\lambda x.M)$, where $M$ is a term, and
- *applications* $(MN)$, where $M$ and $N$ are terms.

The abstraction $(\lambda x.M)$ is intended to represent a function, and $((\lambda x.M)N)$ can be 'reduced' to $M[N/x]$: the result of substituting $N$ for $x$ in $M$. Versions of the λ-calculus are used in denotational semantics and higher-order logic. The original, *untyped* λ-calculus can express arbitrary computations, but its terms are meaningless symbol strings. The *typed* λ-calculus assigns types to all variables, yielding a straightforward set-theoretic semantics: types denote sets and abstractions denote functions. The typed system is therefore more intuitive, but also more restrictive. It assigns $(\lambda x.M)$ the type $\sigma \rightarrow \tau$ if $x$ has type $\sigma$ and $M$ has type $\tau$; it allows $(MN)$ only if $M$ has type $\sigma \rightarrow \tau$ and $N$ has type $\sigma$. It rejects terms like $(\lambda xy.y(xxy))(\lambda xy.y(xxy))$, Turing's fixed-point combinator, which can express recursion.

A danger with these beautiful but sophisticated mathematical techniques is that they might be used incorrectly, not capturing the intended behaviour of the programming constructs being defined. To eliminate this risk, one could specify the behaviour in a more natural form (so called *operational semantics*) and prove the two specifications to be equivalent. This was the topic of the dissertation (1) for which Mike received his PhD from the University of Edinburgh in 1973, supervised by Rod Burstall.

Mike proved the equivalence of the denotational and operational semantics of pure LISP. He presented an early example of what is now called a *structural* operational semantics: reduction relations defined as logical inference systems.

> Mike Gordon's thesis . . . contains a pretty rule-based operational semantics, with the environment needed to model dynamic binding incorporated in the configuration; this was the first treatment of part of a real programming language. (Plotkin 2004, p. 5)

LISP presented a particular challenge due to its unusual treatment of variables. And so Mike obtained an invitation from LISP's inventor, John McCarthy, to work for a year at his Artificial Intelligence Laboratory at Stanford University.

## Edinburgh, Stanford and Edinburgh LCF

The period from 1970 to 1981 set the stage for Mike's career. In 1970, when Mike began his PhD research at Edinburgh, computer science there was fragmented among rival departments. He worked in the Department of Machine Intelligence, which was part of the School of Artificial Intelligence. While he undertook research on the semantics of LISP, others in the school were working on formal logic and automated reasoning.

Formal logic is concerned with precisely specified languages along with symbols for logical connectives such as 'and' ($\wedge$), 'or' ($\vee$), 'not' ($\neg$), 'implies' ($\rightarrow$) and the *quantifiers*: 'for all' ($\forall$) and 'there exists' ($\exists$). A formal calculus includes strict rules for deducing conclusions from assumptions. *First-order logic* (also known as *predicate calculus*) is the simplest such system. It presupposes a fixed, non-empty universe of mathematical values (which could be numbers, sets, polygons, etc.).

There have always been those who felt that formal logic somehow captured human reasoning. During the 1970s, many practitioners of artificial intelligence felt that if one could only automate reasoning in the predicate calculus, one could automate thought itself. (Yes, it sounds ridiculous now.) McCarthy, a leading AI pioneer, held this view strongly. Mike's first meeting with McCarthy went like this:

> He went to McCarthy's office. With no preliminary, John said: 'I believe everything can be done in first-order predicate calculus.' Mike said nothing. John got up and walked out of his office. Soon he returned, though, and said 'with suitable extensions', and he left again.[3]

So when (in 1974) Mike took up a postdoctoral position at the Stanford AI Lab, he was again working on semantics alongside people focused on formal logic. He organized a discussion group on reasoning about programs, attracting researchers from Stanford and nearby research institutes. After work, he would go home to Richard Waldinger's shared house in Palo Alto. Waldinger also worked on logic and theorem proving, at the Stanford Research Institute's Artificial Intelligence Center.

One project at the Stanford AI Lab was Stanford LCF (Milner 1972), led by Robin Milner (FRS 1988). It has an amusing backstory. In 1969, Scott wrote a manuscript (Scott 1993) introducing a logical calculus with a rule called fixed-point induction, superseding a number of earlier techniques. (Scott's logic was quite different from Hoare's, which was concerned with program code.) Scott was concerned with pure recursive functions written in the typed λ-calculus, for which he proposed a domain-theoretic semantics. He began his paper boldly:

> No matter how much wishful thinking we do, the theory of types is here to stay. There is *no other way* to make sense of the foundations of mathematics.[4] (Scott 1993, p. 413)

Scott was firmly committing himself to the typed λ-calculus. But one month later, Scott made the astonishing discovery of a model for the *untyped* λ-calculus. So he withheld this work from publication, and it became known to researchers only through faded Xerox copies. Working at Stanford, Milner, along with Whitfield Diffie (ForMemRS 2017), Richard Weyhrauch and Malcolm Newey, wrote a computer program to implement Scott's logic, which Milner named the Logic for Computable Functions, or LCF. Milner had already left Stanford by the time Mike arrived. By 1975 they were both in Edinburgh and working together on a new version of LCF, along with Chris Wadsworth.

Stanford LCF had two major limitations. Stored proofs used too much memory, and its fixed command repertoire required lengthy, repetitive sequences of steps even for elementary proofs. Milner realized that he could address both problems by providing a programmable *metalanguage*, which he called ML. Making the prover programmable allowed users to automate any repetitive steps. Moreover, through a language concept known as *abstract types*, no proofs would have to be stored. An abstract type enforces the use of a fixed set of operations; by making those operations coincide precisely with a logic's rules of inference, we could define the type of theorems. The abstraction barrier would ensure that theorems were constructed strictly according to the rules. This technique works for essentially any logic (8).

Edinburgh LCF was finished by 1979 (3). It introduced a simple and effective architecture for *interactive*—as opposed to fully automatic—theorem proving. And far from being a mere metalanguage, ML (2) was seen as a general programming language with a highly innovative design. Mike had been fully involved in these great achievements (32), but was already preparing to strike out on his own. He had already written what would become the standard textbook on denotational semantics (4). With software verification apparently becoming a reality, Mike was the first to think seriously about verifying hardware.

[3] According to Richard Waldinger, as relayed by Bruce Anderson, personal communication, 4 April 2018.

[4] Italics in original.

By 1979, Edinburgh's Department of Computer Science had been transformed by a crowd of new arrivals. These included Rod Burstall and Gordon Plotkin (FRS 1992), who had moved from the Department of Artificial Intelligence, as well as Robin Milner, who had arrived earlier. Hardware and systems people found themselves cheek by jowl with a great many theoreticians. Mike's friendly and modest personality allowed him to overcome resentful tribal divisions. He wanted to investigate the semantics of hardware, and that required talking to the hardware specialists. By 1981, Mike had elaborated an approach to hardware verification—including theoretical development and fully worked out examples—that could scale to large devices (6, 7). He also had an invitation to join the rapidly expanding Computer Laboratory at Cambridge.

## Cambridge and the emergence of hardware verification

The first user of Edinburgh LCF was Avra Cohn. A PhD student of Milner's, she had used it to prove the correctness of an abstract compiler (Cohn 1979, 1983). She was also Mike's wife (figure 1). They had first met at Richard Waldinger's house during Mike's postdoctoral year at Stanford. Now, years later, they were sharing an office at Edinburgh. As the first LCF user, Avra influenced its design by pointing out bugs and suggesting improvements. She and Mike were already working together, a collaboration that would continue for many years. They got married in 1979, and together they brought LCF to Cambridge.

As a new university lecturer, Mike had much to occupy him. By October 1983, he was teaching an advanced course entitled *Topics in Programming Language Theory* (11), with an ambitious syllabus: the predicate calculus, Hoare logic, the λ-calculus, automatic theorem proving using the resolution method, and logic programming. Some of the material from his course notes later found its way into his second textbook (17), covering programming language theory and including LISP code to implement some of the techniques.

He also held a Science Research Council grant (jointly with Milner at Edinburgh) to continue the LCF project. Here I entered the picture, having been hired as a postdoc under this grant. I still remember Mike's kindness in meeting me at the airport and helping me take all my stuff to Cambridge. Avra helped me to get started with LCF. She gave me her code, a bundle of utilities written in ML to help carry out LCF proofs. These included sophisticated heuristic tools based on pattern matching. It is remarkable that this code had not already been incorporated into Edinburgh LCF, which was truly a bare-bones environment. Modified and extended by myself and others, Avra's code lives on in today's systems, for I had decided to take Edinburgh LCF apart, and, aided by Gérard Huet of the Inria[5] lab near Paris, put it back together again. The point was to make LCF more usable and much, much faster.

Meanwhile, Mike was continuing to develop his ideas. We can trace their evolution from his 1981 Edinburgh technical report (7). At 75 pages, this was a substantial document, not to be confused with the short conference version (6). Already he was treating both *combinational* devices, such as adders, and *sequential* devices, such as storage registers. Some examples were at the gate level and others were at the transistor level.

[5] *Institut national de recherche en informatique et en automatique*, the French national research institute for computer science.

Figure 1. Avra Cohn and Mike Gordon in 1980. This is an ASCII art image typical of that era.

From the beginning, Mike had the ambition of scalability. He presented a simple microcoded computer (figure 2) complete with a specification of the machine instructions and microinstructions, including a microprogram. The detailed design took up 21 pages.

While combinational devices can easily be modelled as functions from inputs to outputs, sequential devices are trickier to formalize, as they have internal state. Mike's initial idea was to use the power of domain theory. First he defined the domain of signals $Sig[X]$ (where $X$ is a set of wires) to denote the set of functions from $X$ to some fixed set of values. Sequential devices were also modelled as functions, incorporating the internal state as part of the result (7 p. 8):

> The domain $Seq[X;Y]$ of sequential behaviours from $X$ to $Y$ is defined to be the least solution of the domain equation:
>
> $$Seq[X;\,Y] = Sig[X] \rightarrow (Sig[Y] \times Seq[X;\,Y])$$

Such a behaviour maps the input $X$ to the output $Y$ paired with a new $Seq[X;Y]$ (which models the possibility of an internal state change). A precursor to this technique can already be seen in his brief note on the semantics of sequential machines (5). For the sake of uniformity, he proposed regarding combinational devices as the degenerate case of sequential devices (with an empty internal state), so everything would involve recursive domain equations. But

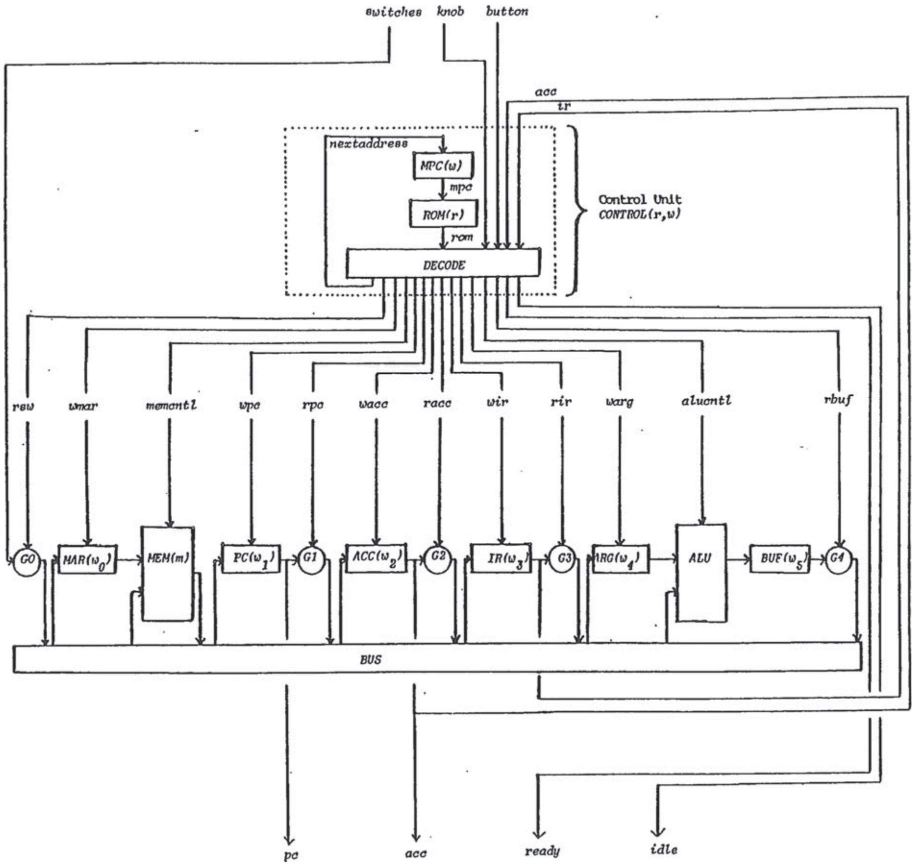


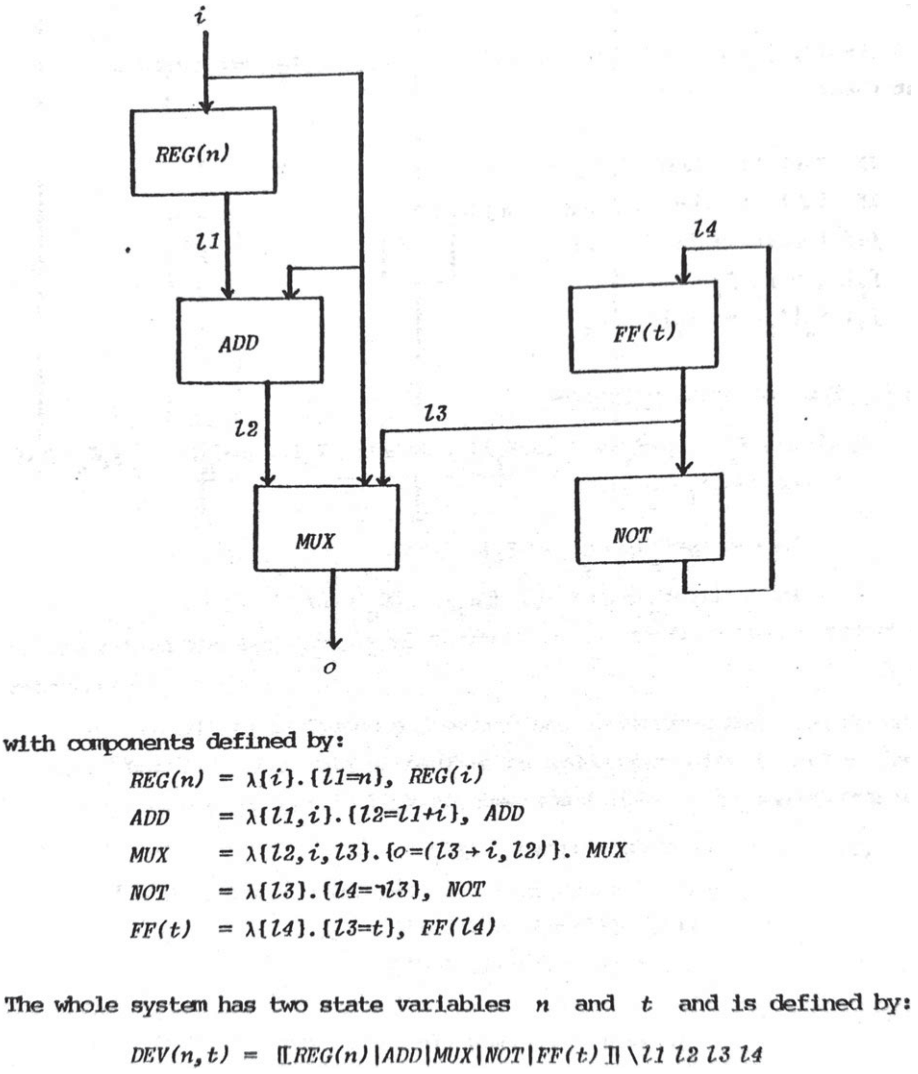
Figure 2. The Gordon computer. Reproduced from (7).

he was unhappy with this high-powered approach (7, p. 9) and was apparently trying to use operational semantics:

> The reader might wonder why we use sequential behaviours at all—why not just work with machines? . . . In fact, at various times during the development of our model, we have tried to eliminate behaviours in favour of machines, in order to avoid having to use the recursive domain equation which defines $Seq[X;Y]$. We have never succeeded.

But eventually, he did succeed, finding something even simpler than ‘machines’: pure logic.

His ambition reflected the broad scope of denotational semantics and the power of domain theory. The components of a computer, including families of input lines carrying time-indexed signals, could be modelled by mathematical functions, possibly nested. It is striking to see diagrams in this early report typical of his much later work (figures 2 and 3). Its main ingredients were already evident, including the notational devices for connecting devices together and hiding internal wires. The mathematical underpinnings would change drastically, but the conception remained the same.


i
REG(n)
l1
ADD
l2
MUX
o
l3
l4
FF(t)
NOT

with components defined by:

```
         REG(n) = λ{i}.{l1=n}, REG(i)
         ADD    = λ{l1,i}.{l2=l1+i}, ADD
         MUX    = λ{l2,i,l3}.{o=(l3→i,l2)}. MUX
         NOT    = λ{l3}.{l4=¬l3}, NOT
         FF(t)  = λ{l4}.{l3=t}, FF(l4)
```

The whole system has two state variables *n* and *t* and is defined by:

```
         DEV(n,t) = ⟦REG(n)|ADD|MUX|NOT|FF(t)⟧\l1 l2 l3 l4
```

Figure 3. Extract from Mike’s 1981 report on hardware verification.

By 1983, Mike had put his ideas into practice with his Logic for Sequential Machines (LSM). He implemented this formalism on top of the Cambridge LCF code base, calling the resulting system LCF_LSM (9). Two major changes are evident from his former work. One was the abandonment of domain theory, with its requirement that every domain had to be a partial ordering. The need to deal with the associated ‘bottom’ value (⊥) tended to clutter proofs. Mike thought it could go away temporarily. (It never came back.)

That led to the other major change: the replacement of functions by machines. Previously (7) he had used *Seq*$[X;Y]$ to denote a domain of functions, including the possibility of a state change. Now Mike had figured out how to model sequential devices without using functions,

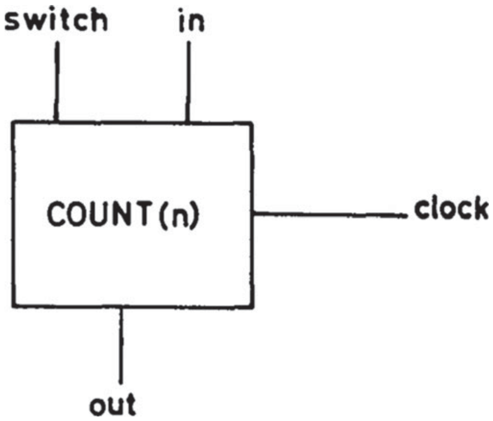


Figure 4. A counter. Reproduced from (13).

while continuing to regard a combinational device as simply a sequential device with an empty state.

LCF_LSM was inspired by Milner's Calculus of Communicating Systems (CCS), a mathematical model of concurrent computing (Milner 1980). CCS is concerned with systems composed of a fixed number of processes that can send messages to each other synchronously (where the sender and receiver act at the same time) and change state. CCS includes principles for demonstrating that two apparently different systems exhibit identical behaviour. Similarly, LCF_LSM concerns components with labelled wires that can be connected together. Wires can also be renamed or hidden. In LCF_LSM, we can write both specifications of desired behaviour and implementations built from smaller components. We can prove that two components have the same behaviour and prove that implementations satisfy a specification.

To illustrate the notation, the following formula specifies the behaviour of the counter in figure 4.

```
COUNT(n) == dev{switch,in,out}.{out = n};
    COUNT(switch->in|n+1)
```

The device has the three wires shown (the system clock is implicit and never appears in specifications). The output line equals the counter's stored value. At each clock tick, the counter loads the value of the input line if switch is true and otherwise increments itself.

Great things were achieved with LCF_LSM. John Herbert (13) used it to verify a bespoke chip design for the Cambridge Fast Ring, an early local area network. Mike used it to verify his computer (10, p. 1):

> The entire specification and verification described here took several months, but this includes some extending and debugging of LCF_LSM (necessary, as this was our first big example). I estimate that it would take me two to four weeks to do another similar exercise now. The complete

> proof requires several hours CPU time on a 2 megabyte Vax750. I found it necessary to prove some of the bigger lemmas ... in batch mode overnight.

This tremendous achievement demonstrated that hardware verification was becoming a reality. Nevertheless, Mike was not satisfied (9, p. 22):

> The selection of rules currently included in LSM is rather ad hoc — I have just implemented what seemed needed for the examples I have done. ... Further experimental work is needed.

Later in the report (pp. 37–8), he mentions the possibility of replacing LSM by some form of predicate logic.

## Higher-order logic, the HOL system and the VIPER Microprocessor

Today Mike's wish to use ordinary logic may seem natural, but in the 1980s many people were introducing specialized formalisms. I had given myself the research goal of providing support for multiple formalisms, only to see Mike's choice of higher-order logic (HOL) gradually take over the verification world. Few people favoured his choice at the time. I certainly didn't, sharing the views of most logicians (Van Benthem & Doets 1983, p. 241):

> Unlike first-order logic and some of its less baroque extensions, second and higher-order logic have no coherent well-established theory; the existent material consisting merely of scattered remarks quite diverse with respect to character and origin.

First-order logic was also strongly preferred by many researchers in artificial intelligence, such as McCarthy at Stanford, as we have seen. And yet, higher-order logic could be seen as a return to tradition (Moore 1988, p. 127):

> The logics considered from 1879 to 1923 ... were generally richer than first-order logic [and] ... at least as rich as second-order logic ... It was in Skolem's work on set theory (1923) that first-order logic was first proposed as all of logic and that set theory was first formulated within first-order logic.

The difference between these 'orders' of logic concerns their treatments of sets and functions. Recall that the symbol ∀ (the universal quantifier) means 'for all' and we can write statements like $\forall xy.\, x+y=y+x$ to assert the commutativity of addition. Here, $x$ and $y$ presumably range over numbers of some sort. But consider the following logical formula:

$$\forall P.\,[P(\mathit{True}) \land P(\mathit{False}) \to \forall x.\, P(x)]. \qquad [1]$$

The universally quantified variable, $P$, is a predicate, and $P(x)$ is a formula. But quantification over predicates is forbidden in first-order logic. First-order logic allows quantification only over some fixed domain of individuals; second-order logic also allows quantification over functions and predicates defined on individuals; higher-order logic allows quantification over arbitrary functions and predicates whose arguments may themselves be other functions and predicates.

Higher-order logic includes a type system to govern all this. For first-order logic there is no need, as all variables range over individuals and it is not essential to introduce different

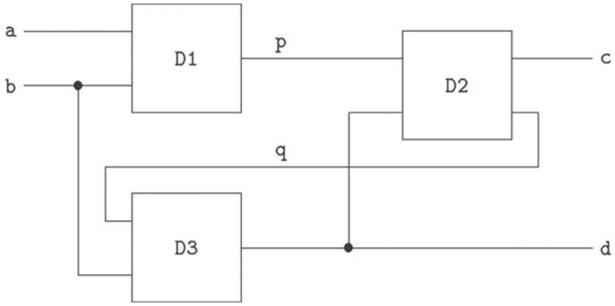


Figure 5. Representing circuit structure with predicates. Reproduced from (14, p. 157).

sorts of individuals, although this is sometimes done anyway. With higher-order logic, Church (1940) used the following types:[6]

- $\iota$, the type of individuals
- $o$, the type of the truth values *True* and *False*
- $\sigma \to \tau$, the type of functions from $\sigma$ to $\tau$

These include as a special case $\sigma \to o$, the type of predicates on type $\sigma$. For formula [1] to make sense, the variable $P$ must have type $o \to o$ and $x$ must have type $o$. Higher-order logic is an extension of Church's typed $\lambda$-calculus.

Mike introduced higher-order logic to the verification world in 1986 (14), sketching its syntax and semantics. He presented examples including an inverter, a full adder (implemented in terms of transistors) and a sequential multiplier. The state in a sequential device is modelled by taking the values on wires to be functions of time, indexed by integers. Then the output of a device at time $t+1$ can be related to its input at time $t$. Mike credited Ben Moszkowski with ideas for reasoning about timing properties. Credit for the suggestion of higher-order logic went to Keith Hanna (Hanna & Daeche 1986), who later decided to try his luck with more advanced type theories. But Mike's simple choice was the right one.

Mike's paper contains the definitive enunciation of the approach of representing hardware devices by relations or predicates. Recall that device behaviours were given first by recursive domain equations (7) and then by dedicated terms (9). But with higher-order logic, the behaviour of a device $D$ is simply a relation over $D$'s external lines, with no distinction between inputs and outputs. Devices are connected together by equating the corresponding lines. Wires are hidden from the outside by existential quantification: mathematically, this is the composition of relations. For example, the formula

$$\exists p\, q.\, D_1(a, b, p) \wedge D_2(p, d, c, q) \wedge D_3(q, b, d)$$

represents the device shown in figure 5. Two standard logical symbols, $\exists$ and $\wedge$, have replaced the special notation we saw in the last line of figure 3.

[6] Church used a different syntax, nearly incomprehensible to modern eyes.

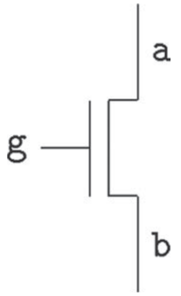


Figure 6. An n-type transistor. Reproduced from (14, p. 159).

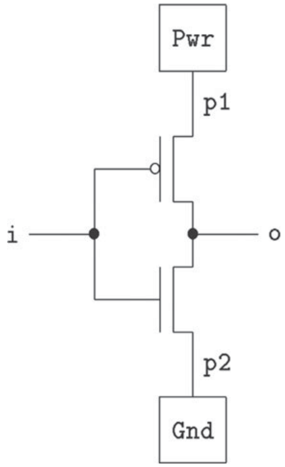


Figure 7. A CMOS inverter. Reproduced from (14, p. 158).

The relational approach is the right way to model individual transistors. Terminals *a* and *b* are neither inputs nor outputs, but are the terminals of a switch, controlled by *g*, the gate (figure 6).

Mike treated an inverter containing two transistors (figure 7). Note that the power and ground are viewed as explicit components, connected to the transistors by internal wires, *p*1 and *p*2. Later in the paper, Mike treats a full adder consisting of 24 transistors. He credits this example to Inder Dhingra and comments, ‘Such a proof would be difficult with the usual representation of combinational circuits as boolean functions. Relations rather than functions are needed to model bidirectionality’ (14, p. 162).

The methodology for verifying such a device is simplicity itself and scales all the way from this inverter to a full-sized computer. You define two predicates, say `INVERTER` (describing the desired behaviour of the inverter) and `INVERTER_IMP` (describing an implementation in terms of smaller components, as in figure 5). Those smaller components will typically be regarded abstractly; there is no need to go all the way down to the transistor level. Then you prove that `INVERTER_IMP`$(i, o)$ implies `INVERTER`$(i, o)$ for all $i$ and $o$. This states that every configuration of values on the wires permitted by the implementation is also permitted by the specification.

Some weaknesses of the methodology are also clear. One is that electronic issues such as gate delays and voltage levels are abstracted away. This approach will not tell you that one output is trying to drive too many inputs or that a combinational circuit is too slow. It is a general limitation of mathematical models that they can never capture the real world in full.

A specific limitation of this approach is that there exists one implementation that satisfies all specifications. Simply connect power to ground; that is formalized as $1 = 0$, which can prove anything. Nobody would do this on purpose, but a design could accidentally short circuit for certain combinations of inputs. The specification would be satisfied, but the implementation would burn. One solution to this difficulty is to prove the converse of the implication above (every behaviour allowed by the specification is satisfied by the implementation), but this is not always possible: most specifications allow some diversity of behaviours. Other measures can be used to check the sanity of the implementation.

Once again, Mike had the task of building a theorem prover, starting with the Cambridge LCF base and creating the world's first interactive implementation of higher-order logic. Avra Cohn was again the first user and, along with Mike, verified a counter circuit (19, 12).[7] This was a pilot study towards the first landmark HOL proof: the VIPER 32-bit microprocessor (Cullyer 1988) (figure 8). The counter, which originated with the UK's Royal Signals and Radar Establishment (RSRE), comprises nine flip-flops and a couple of dozen gates including the counter logic. A complication of the design is that one can request either a single or a double count; the latter is implemented by calling the increment logic twice, so the machine has a two-bit control state and its timing is not uniform. The verification requires reasoning about temporal properties of the circuit.

The verification of the VIPER microprocessor was the first proof of its kind, establishing HOL as a verification platform for realistic hardware. Yet again, this was the work of Avra Cohn. VIPER was designed by RSRE for military purposes, hence the interest in verification; it was specified in a series of levels, from abstract to concrete. Cohn verified the equivalence of the first two levels (Cohn 1988) and, later, the second pair of levels (Cohn 1989a).

Overshadowing these achievements was a controversy over what Cohn had actually accomplished (MacKenzie 1991). Exasperated by exaggerations of her work in marketing material, she wrote a paper (Cohn 1989b) pointing out the inherent limitations of her work in particular and hardware verification in general. She had indeed verified a major part of the VIPER design but not down to the gate level, and the specification omitted some important operating modes. More fundamentally, 'verification involves a pair of models that bear an uncheckable and possibly imperfect relation to the intended design and to the actual device' (1989b, 131–132). In other words, both the designer's objectives and the device's physical manifestation lie beyond the scope of formal verification.

[7] The technical report (12) contains the full HOL proof, some 30 pages of code.

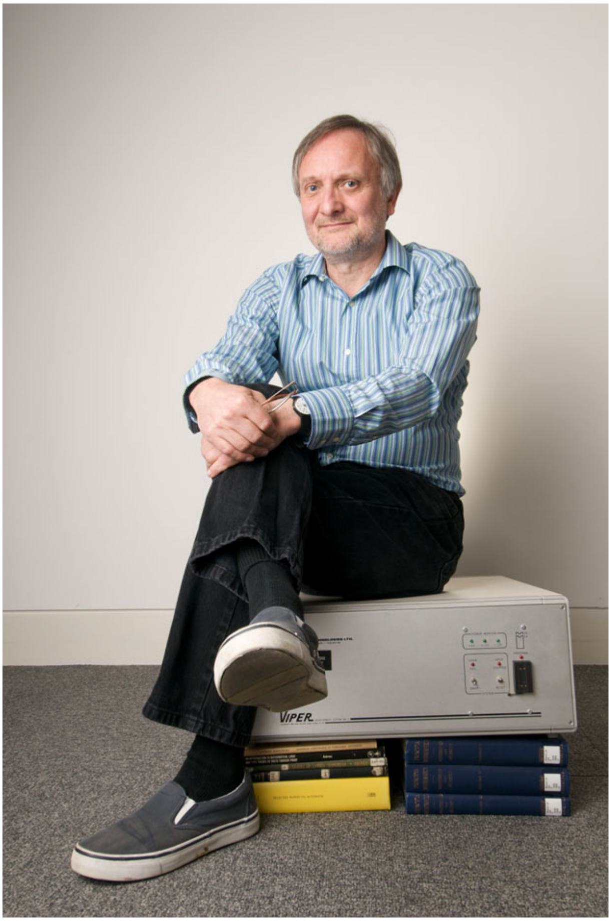


Figure 8. Mike and the Viper processor, which was verified by Avra Cohn using HOL. (Online version in colour.)

## THE GOLDEN AGE OF HOL

The name of Mike's new prover, HOL88, marks 1988 as the official start of the higher-order logic era (16). The achievements reported above had already been attracting a steady stream of PhD students. Graham Birtwistle and Jeff Joyce (15) used HOL88 to verify a simplified version of the Gordon Computer, which they called Tamarack.[8]

Tom Melham developed a comprehensive package for defining recursive data structures (Melham 1989), such as lists and trees; with Mike, he wrote the first HOL manual (20). And there was much more. International meetings on hardware verification were dominated by work done using HOL88 (Birtwistle & Subrahmanyam 1988, 1989). In 1991, Sara Kalvala

[8] Recently, Thomas Türk got a version of this old proof working on the latest version of HOL. It now runs in a couple of seconds.

compiled a snapshot of HOL activity around the world, listing over 80 diverse projects (Kalvala 1991).

By this time, HOL88 was being supplanted by Konrad Slind's faster HOL90, which eventually became today's HOL4 (Slind & Norrish 2008). Other systems inspired by HOL88 include John Harrison's HOL Light (Harrison 1996). In the USA, researchers chose an extended form of higher-order logic as the basis for their Prototype Verification System (PVS) (Owre *et al.* 1992). With my own verification tool (Isabelle), I would continue to push first-order logic and set theory as a basis for verification until the late 90s, when the dominance of higher-order logic became overwhelming. The other major formalism for verification is dependent type theory, exemplified by Coq (Dowek *et al.* 1991), which is a powerful extension of higher-order logic.

Mike was elected to the Royal Society in 1994, the year when the risk posed by hardware defects burst into public view. A floating-point division error in the Pentium processor forced Intel to recall millions of chips at a cost of $475 million (Nicely 2011). Until that date, many theorem provers did not even support negative numbers; it was suddenly urgent to deal with floating-point arithmetic and numerical algorithms. Harrison tackled this (Harrison 1994), and went on to accomplish great things in formalized mathematics, including verifying a floating-point exponential function (Harrison 2000) and (much later) playing a major role in verifying the celebrated Kepler conjecture (Hales *et al.* 2015).

Another landmark was the verification of probabilistic algorithms, which exploit randomness. They can achieve great efficiency, but their result is only guaranteed to be correct with a certain probability, e.g. of the form $1-2^{-n}$. To verify such an algorithm means to show that the probability of an error is no worse than the specification. Joe Hurd formalized enough measure theory to verify a variety of probabilistic algorithms (Hurd 2002). Harrison and Hurd's work led to the substantial libraries of analysis found in many of today's verification systems. They are just two of Mike's many students who did great things in HOL's golden age.

## Software verification, ARM6 and verified compilers

Mike's most far-reaching project was his collaboration with Graham Birtwistle to verify a modern processor. By the year 2000, several processors had been formally verified, but none were full-scale commercial designs containing advanced features such as instruction pipelining. The project involved working with ARM, whose processors are found in billions of mobile phones around the world. Anthony Fox, working at Cambridge, verified the ARM6 processor (figure 9). This work yielded a complete specification of the ARM6's instruction set architecture. Other researchers built projects upon that, aimed at verifying machine language code (30). However, to tell this story properly, we need to go back to the 1980s.

With HOL, Mike introduced a strict treatment of definitions: a new constant $c$ could be introduced only by asserting $c=t$, where $t$ is a λ-term not mentioning $c$ and without free variables. While axioms can lead to contradictions, definitions are conservative. Mike also introduced a principle for declaring new types as non-empty subsets of other types (22). *Recursive* definitions would require explicit fixed-point constructions, though these would soon be automated using ML (Melham 1989). The HOL group may have had Bertrand Russell

Figure 9. The Programming, Logic and Semantics Group at Cambridge, with Mike Gordon at the centre. Also in the photograph are several of Mike's colleagues and students, including Anthony Fox, Magnus Myreen, Scott Owens and Thomas Türk. (Online version in colour.)

in mind (Russell 2007, p. 71):

> The method of 'postulating' what we want has many advantages; they are the same as the advantages of theft over honest toil.

Russell was referring to the tedious construction of the real numbers from the rationals using Dedekind cuts, which was formalized by Harrison (Harrison 1994). While other verification groups preferred theft, Mike and his students were firmly committed to rigour.

In the 1970s, Mike had chosen hardware verification because software verification seemed likely to be solved soon. But that clearly was not happening (it still hasn't), and already in 1988, Mike was thinking about using HOL to verify software (18):

> The work described here is part of a long term project on verifying combined hardware/software systems by mechanized formal proof. (18, p. 3)

This eventually led to intensive research into techniques of verifying software, in ML-like languages and machine language, right down to the bit level.

The dominant approach to software verification, Hoare logic (Hoare 1969), concerned triples of the form

$$\{P\}\, S\, \{Q\}$$

where $S$ was an executable statement, $P$ was the precondition and $Q$ was the postcondition. This Hoare triple asserted that $Q$ would hold after the execution of $S$ provided $P$ held beforehand and the execution terminated. Hoare logic allowed clear, natural proofs, but many difficulties soon manifested themselves. It assumed that the Boolean expressions of the programming language could be identified with the quantifier-free formulas of the assertion language in which $P$ and $Q$ were written. But Boolean expressions are executable and subject to all the ambiguities and complexities that make semantics necessary in the first place. Many verification systems based on Hoare logic were of doubtful correctness or required users to assume many axioms.

Mike decided to implement Hoare logic upon the sound and expressive platform of HOL. His innovation (18) was to define a simple programming language by a formal operational semantics; the Hoare-style rules would then be derived, not simply asserted. Following his definitional approach, there would be no axioms. Through the power of ML—a modified pretty-printer disguising all the machinery—users would be given the illusion that they were working in Hoare logic.

This was the first example of what is now called a *shallow embedding*: a formalism (here Hoare logic) is not defined in HOL, but merely simulated, yielding a convenient proof environment for that formalism. If instead we define the formalism inductively as a mathematical object, then we have a *deep embedding*. The formalism's metatheory can easily be developed, but conducting derivations within the formalism will be painful. Over the years, many assertion languages would be implemented in HOL and other systems using one or the other approach (21). Hoare-style precondition/postcondition calculi remained a favourite. These techniques were well understood by the year 2000, when the ARM6 verification project commenced.

This landmark project, jointly between the universities of Cambridge and Leeds, was funded by the EPSRC. Birtwistle at Leeds would specify the instruction set architecture (ISA) and the processor implementation;[9] Mike at Cambridge would formalize and verify these specifications using HOL4. Anthony Fox, a post-doc of Mike's, undertook the Cambridge task and took about a year to prove that a model of the ARM6 processor correctly implemented the corresponding ISA. Fox went on to specify other ARM instruction sets and, independently, other researchers formalized the x86 and PowerPC. These exceptionally detailed ISA specifications (and associated tools) formed a resource that would be widely used.

With Magnus Myreen, a new PhD student, Mike decided to verify machine code programs. Prior work on verifying machine code was frustrated by the *frame problem*: the need to state explicitly which parts of the machine state were left unchanged. (When you flush the toilet, you don't wonder whether your car doors will unlock.)

A formalism known as separation logic (Reynolds 2002) had been proposed to deal with the frame problem, and Mike suggested adapting those ideas to higher-order logic. Myreen developed techniques to generate Hoare-style assertions for each machine instruction while specifying only which parts of the state changed (26, 27). He was then able to make a *decompiler*: to translate a string of machine instructions into a mathematical function expressing the state transformation, the equivalence automatically verified in HOL4 (28, 31). To crown it all, verified decompilation provided a means of verifying the result of *compilation*: the translation of source code to machine code. Myreen's technology allowed him to create verified LISP interpreters in three different machine languages (29). Myreen's PhD thesis won the British Computer Society's Distinguished Dissertation Award in 2010. His choice of LISP echoes Mike's own PhD thesis (1).

[9] The ISA describes the computer as a machine language programmer sees it. The implementation is in terms of memory, registers and an arithmetic/logic unit (ALU).

These outstanding results attracted substantial follow-up funding. One of the most striking outcomes is CakeML, a version of the ML language implemented as a mathematical function in HOL (Kumar *et al.* 2014). Ramama Kumar and colleagues followed a 'bootstrapping' procedure, initially using HOL itself, to translate fragments of CakeML into binary code; they thus obtained a usable compiler that has been proven to generate correct binary code. This solves the chicken and egg problem of compiler correctness: if you verify a compiler that is written in a high-level language, what compiler do you use to translate it correctly into binary? Mike's students and colleagues could not resist the temptation to apply these techniques to HOL itself (Kumar *et al.* 2016). And so another of Mike's students was honoured: Kumar won the ACM SIGPLAN Doctoral Dissertation Award for 2017.

## Legacy

The verification world of today is substantially shaped by Mike's work. Conferences for HOL users have been held annually since 1988, now broadened to related systems under the name Interactive Theorem Proving (ITP). The leading interactive theorem provers follow the LCF approach, are implemented in some version of ML and support higher-order logic or something stronger. Hardware verification is widely used in industry, while academic research continues apace.

Mike was always keenly interested in all these developments. He worked on many projects connected with hardware description languages, interoperability of verification tools and other technologies. He was fully aware of rival methods, including model checking (to verify system properties by enumeration of finite but large state spaces) and binary decision diagrams (BDDs: graph-based data structures capable of manipulating extremely large propositional formulas efficiently). He found an ingenious way of combining BDDs with HOL (23, 24). He admired the hardware verification research of the University of Texas at Austin using ACL2—a theorem prover based on an utterly different design from HOL's—and worked to link up that prover with HOL (25), combining their complementary strengths.

Although Mike rejected engineering as a degree course, it is clear that he wanted to make an impact on the world. By talking to real hardware designers, he learnt about their practices and problems. He devoted his career to finding realistic solutions. Ironically, although his decision to tackle hardware may have been prompted by a feeling that software was being solved, software developers have generally been uninterested in verification: software can always be patched, and the industry is protected by sweeping warranty disclaimers. However, hardware is not fully solved: the complexity of modern processor designs still makes complete verification unaffordable. Only a few critical components get formal scrutiny.

Much more could be written. Many of Mike's other students accomplished great things and found prominent positions in academia or industry. Mike had a keen interest in computational linguistics: he obtained a Masters degree in linguistics from Cambridge in 1974, and engaged in sponsored research along with Stephen Pulman on applications of higher-order logic to the semantics of natural language. Mike had many teaching and administrative responsibilities, including his role in the planning of the William Gates Building, which now houses the Department and opened in 2001, and his many duties as Deputy Head of Department.

Then there is his personal life. Avra, his wife, eventually retired from active research to bring up their two sons, Katriel and Reuben. She and Mike continued to discuss verification at home. Both of their sons went on to do PhDs in computing: Katriel in cybersecurity at Oxford, Reuben in computational linguistics at Stanford. Somehow this completes the circle.

Mike will be remembered for his kindness and modesty—always eager to confess his failings while concealing his triumphs—and his gentle sense of humour.

Additional information on the history of this period has been written by Mike himself (22, 32) and by his colleagues (Harrison *et al.* 2014; Paulson 2018).

## Acknowledgements

Avra Cohn and Katriel Cohn-Gordon answered many questions and made unique manuscripts available. Mike's former colleagues, students and others supplied valuable tidbits of information and insightful comments. These include Bruce Anderson, Jasmin Blanchette, Jon Crowcroft, Warren Hunt, Sara Kalvala, Simon Laughlin, Joe Leslie-Hurd, Stephen Levinson, Magnus Myreen, Michael Norrish, Gordon Plotkin, Lee Smith, Terence Moore, Richard Waldinger and Tjark Weber. The portrait was taken in 1994 by Prudence Cuming Associates and is © The Royal Society.

## About the author

Lawrence Paulson FRS is Professor of Computational Logic at the University of Cambridge, where he has held established positions since 1983. He has written over 100 refereed conference and journal papers as well as four books. In the 1980s, he worked with Mike Gordon on further development of the LCF proof assistant, which became the foundation of Gordon's LCF_LSM and HOL systems. He introduced the popular Isabelle theorem proving environment in 1986, and made contributions to the verification of cryptographic protocols, the formalization of mathematics, automated theorem proving technology, and other fields. He achieved a formal analysis of the ubiquitous TLS protocol, which is used to secure online shopping, and the formal verification of Gödel's second incompleteness theorem. In 2008, he introduced MetiTarski, an automatic theorem prover for real-valued functions such as logarithms and exponentials. He has the honorary title of Distinguished Affiliated Professor from the Technical University of Munich and is a Fellow of ACM as well as the Royal Society. He holds a PhD in Computer Science from Stanford University, and a BS in Mathematics from the California Institute of Technology.

## References to other authors

Barendregt, H. P. 1984 *The lambda calculus: its syntax and semantics*. North-Holland.

Birtwistle, G. & Subrahmanyam, P. A. (eds) 1988 *VLSI specification, verification and synthesis*. Kluwer Academic Publishers.

Birtwistle, G. & Subrahmanyam, P. A. (eds) 1989 *Current trends in hardware verification and automated theorem proving*. Springer.

Church, A. 1940 A formulation of the simple theory of types. *J. Symbol. Log.* **5**, 56–68. (doi:10.2307/2266170)

Cohn, A. 1979 Machine assisted proofs of recursion implementation. PhD thesis, University of Edinburgh.

Cohn, A. 1983 The equivalence of two semantic definitions: a case study in LCF. *SIAM J. Comput.* **12**, 267–285. (doi:10.1137/0212016)

Cohn, A. J. 1988 A proof of correctness of the VIPER microprocessor: the first level. In Birtwistle and Subrahmanyam 1988, pp. 27–71.

Cohn, A. 1989a Correctness properties of the Viper block model: the second level. In Birtwistle and Subrahmanyam 1989, pp. 1–91.

Cohn, A. 1989b The notion of proof in hardware verification. *J. Automat. Reason.* **5**, 127–139.

Cullyer, W. J. 1988 Implementing safety critical systems: the VIPER microprocessor. In Birtwistle & Subrahmanyam 1988, pp. 1–25.

Dowek, G. *et al.* 1991 The Coq proof assistant user's guide, technical report 134, version 5.6. INRIA-Rocquencourt.

Floyd, R. W. 1967 Assigning meanings to programs. *Proc. Sympos. App. Math.* **19**, 19–32.

Hales, T. C. *et al.* 2015 A formal proof of the Kepler conjecture. https://arxiv.org/abs/1501.02155.

Hanna, F. K. & Daeche, N. 1986 Specification and verification of digital systems using higher-order predicate logic. *IEE Proc. E: Comp. Dig. Tech.* **133**, 242–254. (doi:10.1049/ip-e.1986.0031)

Harrison, J. 1994 Constructing the real numbers in HOL. *Formal Meth. Sys. Des.* **5**, 35–59. (doi:10.1007/BF01384233)

Harrison, J. 1996 HOL Light: a tutorial introduction. In *Formal methods in computer-aided design: FMCAD '96* (ed. M. K. Srivas & A. J. Camilleri), LNCS 1166, pp. 265–269. Springer.

Harrison, J. 2000 Floating point verification in HOL Light: the exponential function. *Formal Meth. Sys. Des.* **16**, 271–305. (doi:10.1023/A:1008712907154)

Harrison, J., Urban, J. & Wiedijk, F. 2014 History of interactive theorem proving. In *Handbook of the history of logic (computational logic)* (ed. J. Siekmann), vol. 9, pp. 135–214. Elsevier.

Hoare, C. A. R. 1989 An axiomatic basis for computer programming. In *Essays in computing science* (ed. C. A. R. Hoare & C. B. Jones), pp. 45–58. Prentice-Hall. (Originally published in 1969.)

Hurd, J. 2002 Verification of the Miller–Rabin probabilistic primality test. *J. Logic Algebr. Program.* **56**, 3–21.

Kalvala, S. 1991 HOL around the world. In *International workshop on the HOL theorem proving system and its applications* (ed. M. Archer, J. J. Joyce, K. N. Levitt & P. J. Windley), pp. 4–12. IEEE Computer Society.

Kumar, R., Arthan, R., Myreen, M. O. & Owens, S. 2016 Self-formalisation of higher-order logic: semantics, soundness and a verified implementation. *J. Autom. Reasoning* **56**, 221–259. (doi:10.1007/s10817-015-9357-x)

Kumar, R., Myreen, M. O., Norrish, M. & Owens, S. 2014 CakeML: a verified implementation of ML. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '14*, pp. 179–191. ACM.

MacKenzie, D. 1991 The fangs of the VIPER. *Nature* **352**, 467–468. (doi:10.1038/352467a0)

Melham, T. F. 1989 Automating recursive type definitions in higher order logic. In *Birtwistle and Subrahmanyam* 1989, pp. 341–386.

Milner, R. 1972 Implementation and applications of Scott's logic for computable functions. *ACM SIGPLAN Not.* **7**, 1–6.

Milner, R. 1980 *A calculus of communicating systems*. LNCS 92. Springer.

Moore, G. H. 1988 The emergence of first-order logic. In *History and philosophy of modern mathematics* (ed. W. Aspray & P. Kitcher), pp. 95–135. University of Minnesota Press, http://hdl.handle.net/11299/185662 (accessed 14 August 2018).

Nicely, T. R. 2011 Pentium FDIV flaw, FAQ page at http://www.trnicely.net/pentbug/pentbug.html (accessed 14 August 2018).

Owre, S., Rushby, J. M. & Shankar, N. 1992 PVS: a prototype verification system. In *Automated deduction: CADE-11 international conference* (ed. D. Kapur), vol. 607 of *LNAI 607*, pp. 748–752. Springer.

Paulson, L. C. 2018 Computational logic: its origins and applications. *Proc. R. Soc. A: Math., Phys. Eng. Sci.* **474**(2210): 20170872. (doi:10.1098/rspa.2017.0872)

Plotkin, G. D. 2004 The origins of structural operational semantics. *J. Logic Algebra. Prog.* **60–61**, 3–15. (doi:10.1016/j.jlap.2004.03.009)

Reynolds, J. C. 2002 Separation logic: a logic for shared mutable data structures. In *17th annual IEEE symposium on logic in computer science*, pp. 55–74. IEEE Computer Society.

Russell, B. 2007 *Introduction to mathematical philosophy*. Cosimo. (First published in 1919.).

Scott, D. S. 1970 Outline of a mathematical theory of computation. Technical report PRG-2, University of Oxford.

Scott, D. S. 1993 A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoret. Comput. Sci.* **121**, 411–440. (Annotated version of the 1969 manuscript.) (doi:10.1016/0304-3975(93)90095-B)

Slind, K. & Norrish, M. 2008 A brief overview of HOL4. In *Theorem proving in higher order logics, TPHOLs 2008* (ed. O. A. Mohamed, C. Muñoz & S. Tahar), LNCS 5170, pp. 28–32. Springer.

VanBenthem, J. & Doets, K. 1983 Higher-order logic. In *Handbook of philosophical logic vol. I: elements of classical logic* (ed. D. Gabbay & F. Guenthner), pp. 275–329. Springer.

## Bibliography

The following publications are those referred to directly in the text. A full bibliography is available at https://doi.org/10.6084/m9.figshare.c.4237541.

(1) 1973 Evaluation and denotation of pure LISP programs: a worked example in semantics. PhD thesis, University of Edinburgh.

(2) 1978 (With R. Milner, L. Morris, M. Newey & C. Wadsworth) A metalanguage for interactive proof in LCF. In *5th ACM symposium on principles of programming languages, POPL '78*, New York, pp. 119–130. ACM.

(3) 1979 (With Robin Milner & Christopher P. Wadsworth) *Edinburgh LCF: a mechanised logic of computation*, LNCS 78. Springer.

(4) *The denotational description of programming languages: an introduction*. Springer.

(5) 1980 The denotational semantics of sequential machines. *Inf. Process. Lett.* **10**(1), 1–3.

(6) 1981 Register transfer systems and their behaviour. In *Computer hardware description languages and their applications* (ed. M. Breuer & R. Hartenstein), pp. 23–36. North-Holland.

(7) A model of register transfer systems with applications to microcode and VLSI correctness, technical report CSR-82-81. University of Edinburgh, https://doi.org/10.17863/CAM.22684 (accessed 14 August 2018).

(8) 1982 Representing a logic in the LCF metalanguage. In *Tools and notions for program construction: an advanced course* (ed. D. Néel), pp. 163–185. Cambridge University Press.

(9) 1983 LCF_LSM, a system for specifying and verifying hardware, technical report UCAM-CL-TR-41. University of Cambridge Computer Laboratory.

(10) Proving a computer correct with the LCF_LSM hardware verification system, technical report UCAM-CL-TR-42. University of Cambridge Computer Laboratory.

(11) Topics in programming language theory, handwritten lecture notes, Cambridge, October.

(12) 1986 (With Avra Cohn) A mechanized proof of correctness of a simple counter, technical report UCAM-CL-TR-94. University of Cambridge Computer Laboratory.

(13) (With J. Herbert) Formal hardware verification methodology and its application to a network interface chip. *IEE Proc. E: Comp. Digi. Tech.* **133**(5), 255–270.

(14) Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal aspects of VLSI design* (ed. G. Milne & P. A. Subrahmanyam), pp. 153–177. North-Holland.

(15) (With Jeff Joyce & Graham Birtwistle) Proving a computer correct in higher order logic, technical report UCAM-CL-TR-100. University of Cambridge Computer Laboratory.

(16) 1988 HOL: a proof generating system for higher-order logic. In *VLSI Specification, verification and synthesis* (ed. Graham Birtwistle & P. A. Subrahmanyam), pp. 73–128. Kluwer Academic Publishers.

(17) *Programming language theory and its implementation*. Prentice-Hall.

(18) 1989 Mechanizing programming logics in higher order logic. In *Current trends in hardware verification and automated theorem proving* (ed. Graham Birtwistle & P. A. Subrahmanyam), pp. 387–439. Springer.

(19) 1991 (With Avra Cohn) A mechanized proof of correctness of a simple counter. In *Theoretical foundations of VLSI design* (ed. Ken McEvoy & J. V. Tucker), pp. 65–96. Cambridge University Press.

(20) 1993 (With Thomas F. Melham) *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

(21) 1995 (With Jonathan Bowen) A shallow embedding of Z in HOL. *Info. Softw. Techn.* **37**(5), 269–276.

(22) 2000 From LCF to HOL: a short history. In *Proof, language and interaction: essays in honor of Robin Milner* (ed. Gordon Plotkin, Colin Stirling & Mads Tofte), pp. 169–185. MIT Press.

(23) 2002 Programming combinations of deduction and BDD-based symbolic calculation. *LMS J. Comput. Math.* **5**, 56–76.

(24) Puzzletool: An example of programming computation and deduction. In *Theorem proving in higher order logics: TPHOLs 2002* (ed. Victor A. Carreño, César A. Muñoz & Sofiène Tahar), LNCS 2410, pp. 214–229. Springer, http://link.springer.de/link/service/series/0558/tocs/t2410.htm.

(25) 2006 (With Warren A. Hunt Jr, Matt Kaufmann & James Reynolds) An embedding of the ACL2 logic in HOL. In *Sixth international workshop on the ACL2 theorem prover and its applications* (ed. Panagiotis Manolios & Matthew Wilding), pp. 40–46. ACM.

(26) 2007 (With Magnus O. Myreen & Anthony C. J. Fox) Hoare logic for ARM machine code. In *Fundamentals of software engineering* (ed. Farhad Arbab & Marjan Sirjani), pp. 272–286. Springer.

(27) (With Magnus O. Myreen) Hoare logic for realistically modelled machine code. In *Tools and algorithms for the construction and analysis of systems* (ed. Orna Grumberg & Michael Huth), pp. 568–582. Springer.

(28) 2008 (With Magnus O. Myreen & Konrad Slind) Machine-code verification for multiple architectures: an application of decompilation into logic. In *Formal Meth. Comp.-Aid. Des. FMCAD '08*, pp. 20:1–20:8. IEEE Press.

(29) 2009 (With Magnus O. Myreen) Verified LISP implementations on ARM, x86 and PowerPC. In *Theorem proving in higher order logics* (ed. Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel), LNCS 5674, pp. 359–374. Springer.

(30) 2010 (With Anthony C. J. Fox & Magnus O. Myreen) Specification and verification of ARM hardware and software. In *Design and verification of microprocessor systems for high-assurance applications* (ed. David S. Hardin), pp. 221–247. Springer.

(31) 2012 (With Magnus O. Myreen) Function extraction. *Sci. Comput. Program.* **77**(4), 505–517.

(32) 2015 Tactics for mechanized reasoning: a commentary on Milner (1984) The use of machines to assist in rigorous proof. *Phil. Trans. R. Soc. A: Math., Phys. Eng. Sci.* **373**(2039).

(33) 2017 Fifty year reunion of the class of 1966, http://www.cl.cam.ac.uk/archive/mjcg/plans/ClassReunion.html (accessed 14 August 2018).

(34) Management trainee at the North Thames Gas Board, http://www.cl.cam.ac.uk/archive/mjcg/plans/NorthThamesGasBoard.html (accessed 14 August 2018).

(35) Struggling with mathematics at Cambridge, http://www.cl.cam.ac.uk/archive/mjcg/plans/CambridgeUndergraduate.html (accessed 14 August 2018).

(36) 2018 Summer job at the National Physical Laboratory, my Part II essay on perceptrons and meeting David Marr, http://www.cl.cam.ac.uk/archive/mjcg/plans/NPL.html (accessed 14 August 2018).