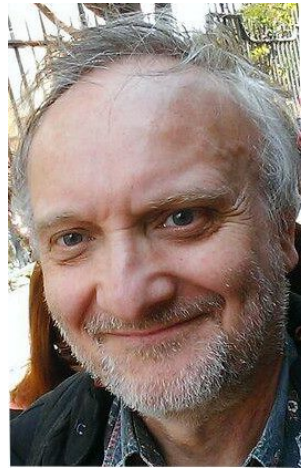
Mike Gordon: Tribute to a Pioneer in Theorem Proving and Formal Verification

John Harrison
Amazon Web Services

ITP, Monday 9th July 2018 (11:00-12:00)



From HUG to HOL to TPHOLs to ITP

- HOL User Group (HUG): Cambridge 1988, Cambridge 1989, Aarhus 1990.
- Workshop on the HOL theorem proving system and its applications (HOL): Davis 1991, Leuven 1992, Vancouver 1993.
- Higher Order Logic Theorem Proving and its Applications (HOL): Valletta 1994, Aspen Grove 1995.
- Theorem Proving in Higher Order Logics (TPHOLs): Turku 1996, Murray Hill 1997, Canberra 1998, Nice 1999, Portland 2000, Edinburgh 2001, Hampton 2002, Rome 2003, Park City 2004, Oxford 2005, Seattle 2006, Kaiserslautern 2007, Montreal 2008, Munich 2009.
- Interactive Theorem Proving (ITP): Edinburgh 2010, Nijmegen 2011, Princeton 2012, Rennes 2013, Vienna 2014, Nanjing 2015, Nancy 2016, Brasilia 2017, Oxford 2018.

The 1988 HUG meeting

The 1988 HUG meeting (detail)

The 1992 HOL meeting

The 1992 HOL meeting (detail)

Early Life

http://www.cl.cam.ac.uk/archive/mjcg/plans/ClassReunion.html

Born in Ripon, Yorkshire, 28 February 1948

Family

"My father, John Gordon, was a part-time lecturer on philosophy and psychology for the WEA [Workers' Educational Association]. He suffered from depression which he tried to cure using psychoanalysis. It didn't work and he committed suicide in 1956, when I was eight."

Family

"My father, John Gordon, was a part-time lecturer on philosophy and psychology for the WEA [Workers' Educational Association]. He suffered from depression which he tried to cure using psychoanalysis. It didn't work and he committed suicide in 1956, when I was eight."

"My mother, Daphne Gordon, was born in India and spent her childhood there (according to a DNA test by Ancestry.co.uk my genetic ancestry is 21% Asian)."

Family

"My father, John Gordon, was a part-time lecturer on philosophy and psychology for the WEA [Workers' Educational Association]. He suffered from depression which he tried to cure using psychoanalysis. It didn't work and he committed suicide in 1956, when I was eight."

"My mother, Daphne Gordon, was born in India and spent her childhood there (according to a DNA test by Ancestry.co.uk my genetic ancestry is 21% Asian)."

"Up until my teens I'd spend the holidays with my mother and her unmarried sister in a rented flat in Ripon, Yorkshire, where their mother, my grandmother Molly, lived."

A year at Dartington Hall School

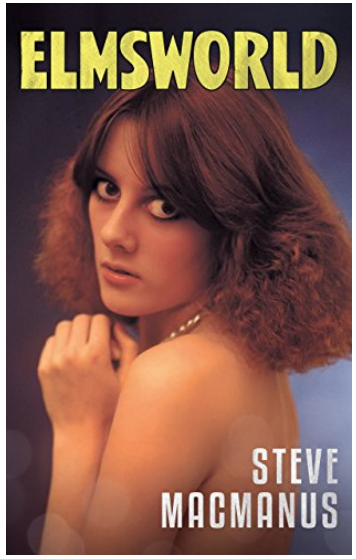
"I spent a year as a border at the junior school of the notorious Dartington Hall. My memories of this are dim, but to the extent that I remember anything, I remember it as being a lot of fun. My mother told me that she was unhappy with Dartington because I forgot how to read and write when there . . ."

A year at Dartington Hall School

"I spent a year as a border at the junior school of the notorious Dartington Hall. My memories of this are dim, but to the extent that I remember anything, I remember it as being a lot of fun. My mother told me that she was unhappy with Dartington because I forgot how to read and write when there . . ."

SCHOOL FOR SCANDAL

Roy Kerridge on the last days of Dartington Hall



STEVE MACMANUS

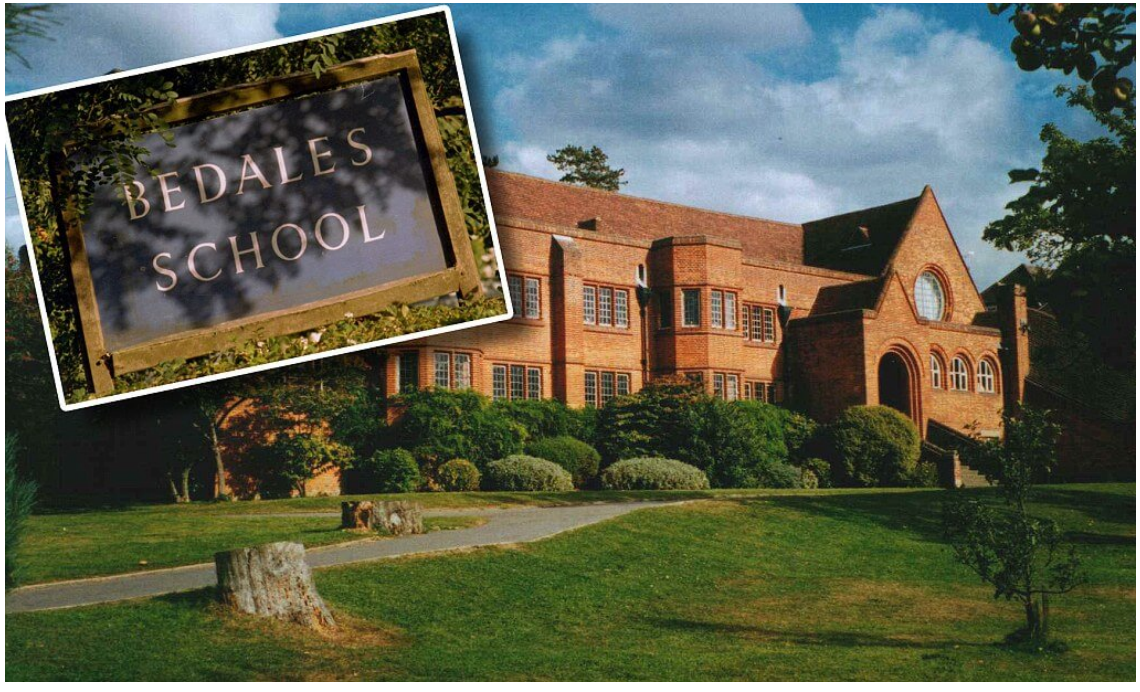
ELMSWORLD

MY LIFE AT DARTINGTON HALL SCHOOL

1963 – 1971

Bedales School

"I regard Bedales as being my home between the ages of eight and eighteen: first the junior school, called Dunhurst, and then the senior school. Both Dunhurst and Bedales are boarding schools and I think of them as my childhood home because they were the places I lived most of the year until I went to university."



Life at school

"The only friends I had when growing up were the children I knew at school. I had no friends outside school. As an only child, my classmates were the closest I had to siblings."

Life at school

"The only friends I had when growing up were the children I knew at school. I had no friends outside school. As an only child, my classmates were the closest I had to siblings."

"I was pretty happy at school . . . I also remember realising after my O-levels that I had only two more years at Bedales and worrying about leaving — but I was happy to leave when the time actually came."

Academic interests

"Music was the only subject for which I remember receiving punishment for bad behaviour (maybe I also did for sport, which I mostly hated). I was also terrible at the humanities and dropped everything I could as soon as possible. I dreaded writing essays. My academic life at school centred around science."

Reflections 50 years later

"I'm grateful to Bedales for turning me from being a mixed-up child who might have gone off the rails into someone I think of as reasonably stable and normal . . . "

"That I can't remember what my classmates were like at school is reassuring — it gives me hope that they now can't remember what I was like!"

Cambridge

http://www.cl.cam.ac.uk/archive/mjcg/plans/NorthThamesGasBoard.html
http://www.cl.cam.ac.uk/archive/mjcg/plans/CambridgeUndergraduate.html

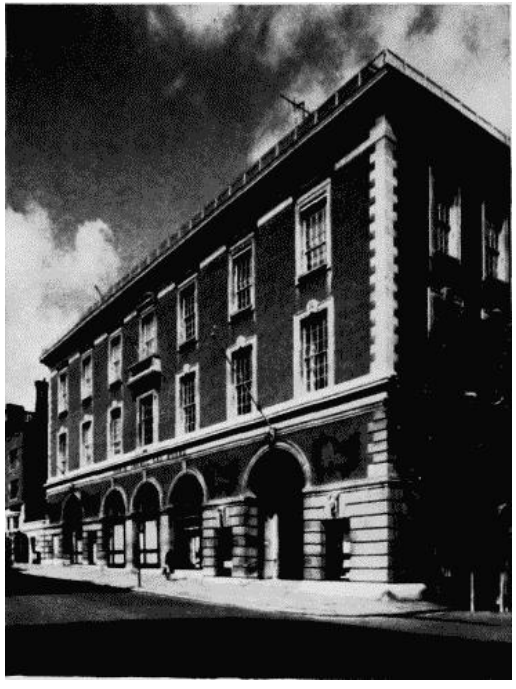
http://www.cl.cam.ac.uk/archive/mjcg/plans/NPL.html

Accepted at Cambridge

"In 1966 I applied to Cambridge to study engineering. I was accepted at Cambridge and the Engineering Department wrote to me recommending that I use the gap year to get appropriate work experience.

Accepted at Cambridge

"In 1966 I applied to Cambridge to study engineering. I was accepted at Cambridge and the Engineering Department wrote to me recommending that I use the gap year to get appropriate work experience. . . . I somehow ended up as a management trainee at the now-defunct North Thames Gas Board."



“How the gasworks changed my life”

“In my teens I made model aircraft both gliders and ones powered by tiny petrol engines; some of which I attempted to make radio-controlled . . . I also built transistor radios from kits . . . Going to university to study engineering was an obvious choice.”

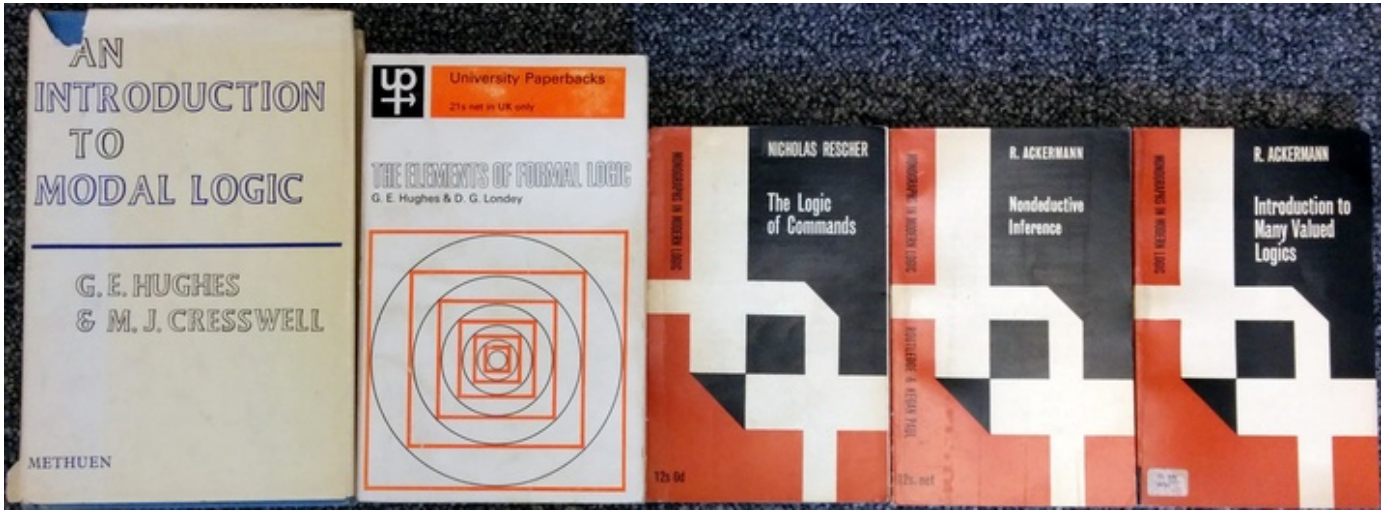
"How the gasworks changed my life"

"In my teens I made model aircraft both gliders and ones powered by tiny petrol engines; some of which I attempted to make radio-controlled . . . I also built transistor radios from kits . . . Going to university to study engineering was an obvious choice."

"My time at the North Thames Gas Board, particularly Becton, destroyed my enthusiasm for engineering. Furthermore, I found the preparatory reading that I'd been sent by the engineering department very boring. I decided to try to switch to a different subject as far away from engineering as possible."

Towards mathematics and logic

"I used to like to hang out in the Gower Street Dillons bookshop and browse through books on exotic subjects. I somehow stumbled upon symbolic logic and bought books on it, some of which I read on the train when commuting between Becton Gasworks and Crouch End, where I was living with my mother. I still have these books. Some examples are below."



Switching to mathematics

"Despite my relative weakness in the subject, I decided to try to switch from engineering to mathematics. I was incredibly lucky in that my tutor John Casey supported my wish to switch subjects and managed to make it happen. I think he may have thought engineering to be of dubious academic value and that he should do his best to save me from it."

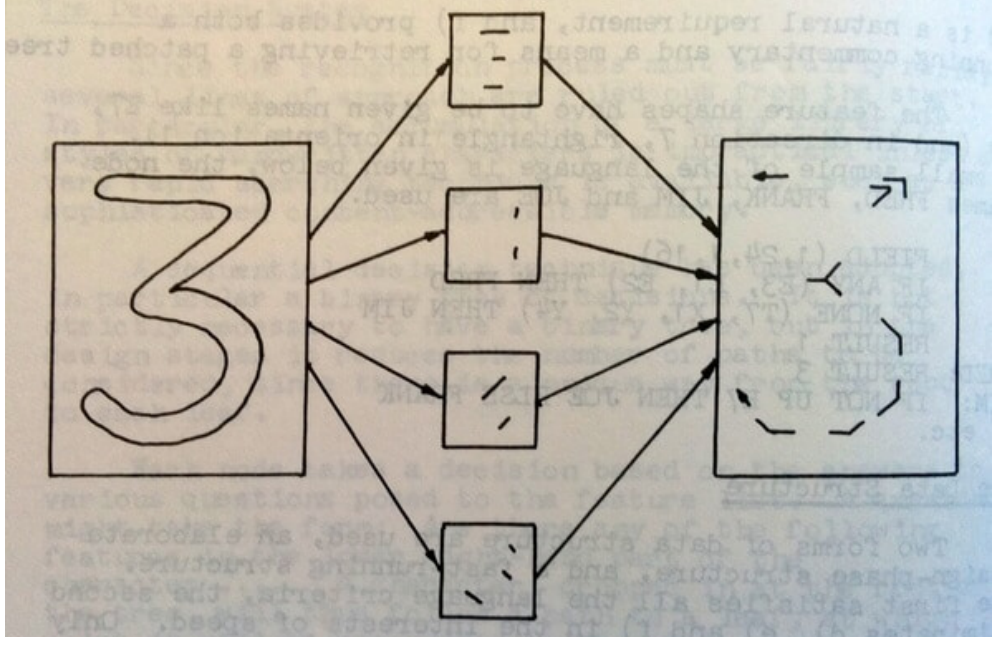
Switching to mathematics

"Despite my relative weakness in the subject, I decided to try to switch from engineering to mathematics. I was incredibly lucky in that my tutor John Casey supported my wish to switch subjects and managed to make it happen. I think he may have thought engineering to be of dubious academic value and that he should do his best to save me from it."

"Although I struggled with the Mathematical Tripos, and it turned out to involve writing an essay, I ended up having a career that I don't think I would have had if I'd studied engineering."

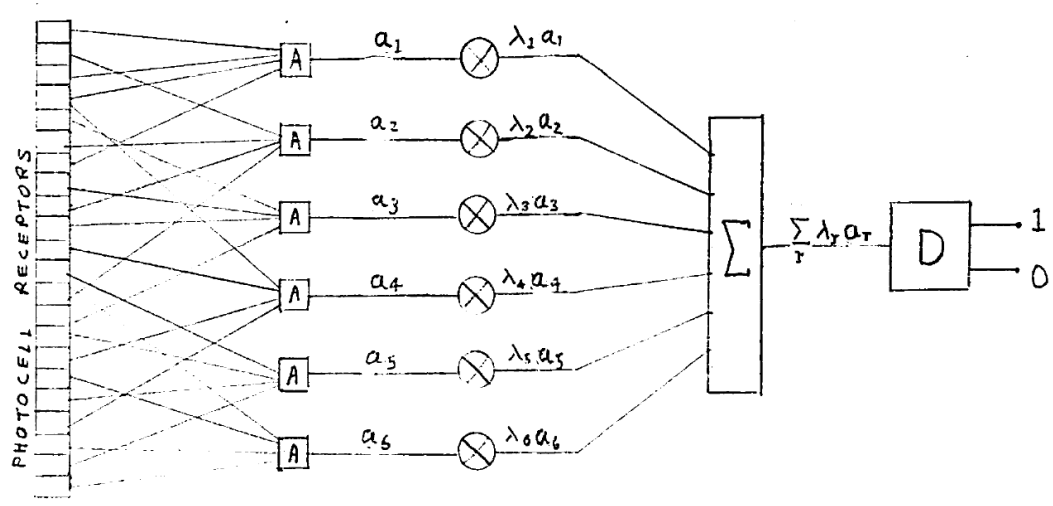
Summer job at the National Physical Laboratory

" In the summer vacation following my second undergraduate year I got a job at the NPL. I was assigned to Julian Ullman, in whose office I was put. . . . I chatted a lot with Ullman about pattern recognition theory in general. I was also given a concrete project to use NPL software to write a program to recognise hand written characters."



Perceptrons essay

"During my third year at Cambridge there was an experimental scheme allowing Part II Mathematical Tripos students to submit an essay. Inspired by my time at NPL, I decided to submit an essay on Perceptrons."



"Perceptrons were an early form of machine learning. Had I pursued my interest in this area perhaps my career would have taken a very different direction!"

The right choice?

"I'm pretty sure that switching from engineering to mathematics was a good move . . . Although I found the course very tough, it gave me the tools and confidence to feel that with sufficient effort — often more than I was willing to deploy — I could master any mathematical material I needed. This laid a solid foundation for my subsequent academic career. I've been more successful than I initially hoped and expected — there's a good chance that the Mathematical Tripos should be thanked for this."

From Edinburgh to Stanford and back

http://www.cl.cam.ac.uk/archive/mjcg/plans/NPL.html
https://arxiv.org/abs/1806.04002

https:

//pdfs.semanticscholar.org/f1a1/fa6f7edcef40cd386b347d00430e711fce9f.pdf

Diploma in Machine Intelligence and Perception

"Writing character recognition programs at the NPL was my first experience of using a computer for symbolic processing and it was the first time I enjoyed programming. My time at NPL got me interested in artificial intelligence, with the result that after I graduated I enrolled in the postgraduate Diploma in Machine Intelligence and Perception at Edinburgh and then stayed on in Edinburgh for my PhD (though I ended up working on programming language semantics, not AI)."

1973 Edinburgh PhD

Evaluation and denotation of pure LISP programs: a worked example in semantics (supervisor Rod Burstall)

ABSTRACT

A Scott/Strachey style denotational semantics intended to describe pure LISP is examined. I present evidence that it is an accurate rendering of the language described in chapter 1 of the LISP 1.5 Programmer's Manual, in particular I show that call-by-value and fluid variables are correctly handled. To do this I have:

(1) Written an operational 'semantics' of pure LISP and shown it equivalent to the denotational one

(2) Proved that, relative to the denotational semantics, the LISP functions apply,eval,...,etc. correctly compute meanings.

The proof techniques used are derived from the work of Wadsworth; roughly one first proves the results for a class of 'finite' programs and then extends them to all programs by a limiting argument. Conceptually these arguments are inductions on length of computation and to bring this out I've formulated a rule of inference which enables such operational reasoning to be applied to the denotational semantics.

1974 Stanford Postdoc

Worked as a post-doc with John McCarthy at the Stanford AI Lab. According to Richard Waldinger:

> *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.*

1974 Stanford Postdoc

Worked as a post-doc with John McCarthy at the Stanford AI Lab. According to Richard Waldinger:

> *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, said "with suitable extensions" and he left again.*

1974 Stanford Postdoc

Worked as a post-doc with John McCarthy at the Stanford AI Lab. According to Richard Waldinger:

> *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, said "with suitable extensions" and he left again.*

Among other things, Mike met his wife-to-be, Avra Cohn, at Richard Waldinger's house here!

Logical relations

Mike continued his study of program semantics, in particular the connection between loops and fixed points. This led to the the idea of what Plotkin called *logical relations*:

G.D.P. -12- SAI-RM-4
October 1973

But the only λ-definable functionals of type $(\iota,\iota,0)$ are $\lambda\alpha^{\iota}_{0}\lambda\alpha^{\iota}_{1}\lambda\alpha^{\iota}_{2}\lambda\alpha^{\iota}_{3}\ \alpha^{\iota}_{j}$ for $0\leq j\leq 3$ none of which are $=_{\iota}$ if $|D_{\iota}|>1$.

M. Gordon proposed, as a possible remedy, that relations $R_{\iota}\subseteq D^{2}_{\iota}$ should be extended - not just permutations. Starting with such an R_{ι}, the R_{σ}'s are defined by:

$$R_{(\sigma\to\tau)}(f,g)\equiv \forall x,y\in D_{\sigma}.(R_{\sigma}(x,y)\to R_{\tau}(fx,gy)).$$

The idea was independently used by Reynolds to relate direct and continuation semantics.

Edinburgh LCF

http://www.cl.cam.ac.uk/archive/mjcg/plans/NPL.html
https://arxiv.org/abs/1806.04002

http://rsta.royalsocietypublishing.org/content/roypta/373/2039/20140234.full.pdf

Stanford LCF

A bit earlier (starting in 1970), Robin Milner had also spent time with McCarthy at the Stanford AI Lab. Together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey he developed *Stanford LCF*, a proof assistant for Scott's Logic of Computable Functions.

Stanford LCF

A bit earlier (starting in 1970), Robin Milner had also spent time with McCarthy at the Stanford AI Lab. Together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey he developed *Stanford LCF*, a proof assistant for Scott's Logic of Computable Functions.

- Support for backward, goal-directed proof

Stanford LCF

A bit earlier (starting in 1970), Robin Milner had also spent time with McCarthy at the Stanford AI Lab. Together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey he developed *Stanford LCF*, a proof assistant for Scott's Logic of Computable Functions.

- Support for backward, goal-directed proof
- A powerful simplification mechanism

Stanford LCF

A bit earlier (starting in 1970), Robin Milner had also spent time with McCarthy at the Stanford AI Lab. Together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey he developed *Stanford LCF*, a proof assistant for Scott's Logic of Computable Functions.

- Support for backward, goal-directed proof
- A powerful simplification mechanism

Just one of many significant proof assistants being developed at around the same time (AUTOMATH, Mizar, SAM), and had significant drawbacks:

Stanford LCF

A bit earlier (starting in 1970), Robin Milner had also spent time with McCarthy at the Stanford AI Lab. Together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey he developed *Stanford LCF*, a proof assistant for Scott's Logic of Computable Functions.

- Support for backward, goal-directed proof
- A powerful simplification mechanism

Just one of many significant proof assistants being developed at around the same time (AUTOMATH, Mizar, SAM), and had significant drawbacks:

- Memory limitations made it hard to store large proofs

Stanford LCF

A bit earlier (starting in 1970), Robin Milner had also spent time with McCarthy at the Stanford AI Lab. Together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey he developed *Stanford LCF*, a proof assistant for Scott's Logic of Computable Functions.

- Support for backward, goal-directed proof
- A powerful simplification mechanism

Just one of many significant proof assistants being developed at around the same time (AUTOMATH, Mizar, SAM), and had significant drawbacks:

- Memory limitations made it hard to store large proofs
- The set of proof commands could not easily be extended.

Back to Edinburgh

When Mike arrived at Stanford, Milner had already returned to Edinburgh and was working on the successor to Stanford LCF.

- Initially worked with Malcolm Newey and Lockwood Morris
- In 1975 they took up faculty positions and were replaced by Mike Gordon and Chris Wadsworth.

The LCF project and the ML language itself were finalized, and Mike wrote down the first semantics of ML.

Edinburgh LCF and its architecture

The design of LCF tackled the two shortcomings of Stanford LCF:

Edinburgh LCF and its architecture

The design of LCF tackled the two shortcomings of Stanford LCF:

- Did not store complete proofs, just remembering the *conclusions* of proofs

Edinburgh LCF and its architecture

The design of LCF tackled the two shortcomings of Stanford LCF:

- Did not store complete proofs, just remembering the *conclusions* of proofs
- Provided a full programming 'meta-language' (ML) so that the user could extend the set of proof commands

Edinburgh LCF and its architecture

The design of LCF tackled the two shortcomings of Stanford LCF:

- Did not store complete proofs, just remembering the *conclusions* of proofs
- Provided a full programming 'meta-language' (ML) so that the user could extend the set of proof commands

But how to ensure that theorems were proved correctly, not just arbitrarily asserted or created by buggy user proof commands?

Edinburgh LCF and its architecture

The design of LCF tackled the two shortcomings of Stanford LCF:

- Did not store complete proofs, just remembering the *conclusions* of proofs
- Provided a full programming 'meta-language' (ML) so that the user could extend the set of proof commands

But how to ensure that theorems were proved correctly, not just arbitrarily asserted or created by buggy user proof commands?

- Make theorems an abstract type in the metalanguage ('`thm`') with its only constructors being primitive inference rules of the logic.

Edinburgh LCF and its architecture

The design of LCF tackled the two shortcomings of Stanford LCF:

- Did not store complete proofs, just remembering the *conclusions* of proofs
- Provided a full programming 'meta-language' (ML) so that the user could extend the set of proof commands

But how to ensure that theorems were proved correctly, not just arbitrarily asserted or created by buggy user proof commands?

- Make theorems an abstract type in the metalanguage ('`thm`') with its only constructors being primitive inference rules of the logic.

The requirements of the LCF system directly motivated many features of ML.

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage.

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage.
For example, if `th1` is the theorem $\vdash p \Rightarrow (q \Rightarrow p)$ and `th2` is $p \vdash p$, then `MP th1 th2` gives $p \vdash q \Rightarrow p$.

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

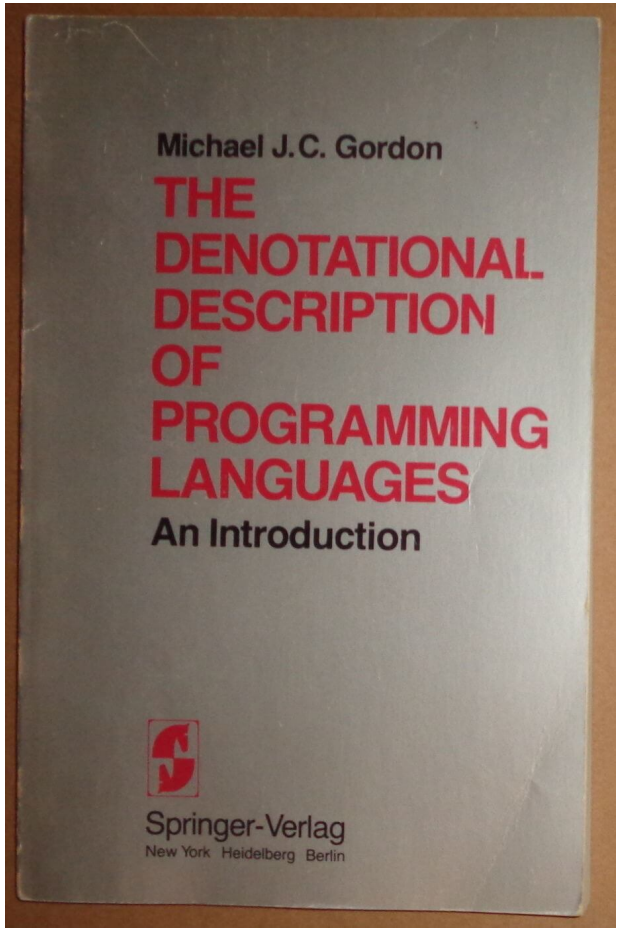
$$\frac{\Gamma \vdash p \Rightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage.
For example, if `th1` is the theorem $\vdash p \Rightarrow (q \Rightarrow p)$ and `th2` is $p \vdash p$, then `MP th1 th2` gives $p \vdash q \Rightarrow p$.
Highly automated or convenient *derived* inference rules can be programmed using these as the basic building-blocks, including support for backward proof via 'tactics'.

The Edinburgh LCF book

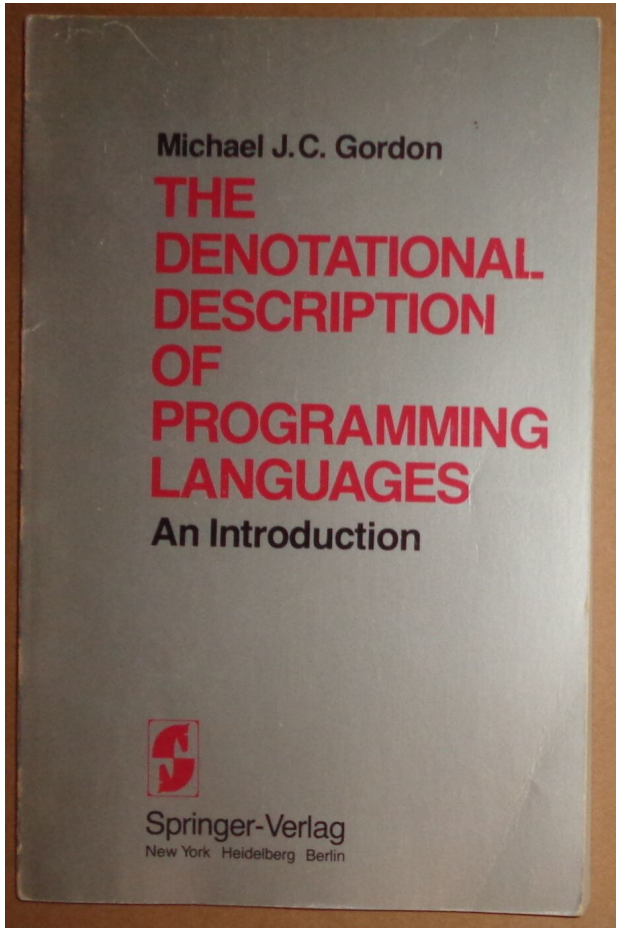
Denotational semantics textbook

As well as wrapping up the LCF project and developing an interest in hardware verification, Mike found time to write one of the most popular textbooks on denotational semantics.



Denotational semantics textbook

As well as wrapping up the LCF project and developing an interest in hardware verification, Mike found time to write one of the most popular textbooks on denotational semantics.



And he and Avra got married!

Cambridge, hardware verification and HOL

https://www.cl.cam.ac.uk/archive/mjcg/papers/HolHistory.pdf

https://arxiv.org/abs/1806.04002

Move to Cambridge

In 1981 Mike returned to Cambridge to take up a lectureship at the University Computer Laboratory, bringing Avra and Edinburgh LCF with him

Move to Cambridge

In 1981 Mike returned to Cambridge to take up a lectureship at the University Computer Laboratory, bringing Avra and Edinburgh LCF with him

- A joint LCF project was funded by the Science and Engineering Research Council run by Mike Gordon in Cambridge and Robin Milner in Edinburgh

Move to Cambridge

In 1981 Mike returned to Cambridge to take up a lectureship at the University Computer Laboratory, bringing Avra and Edinburgh LCF with him

- A joint LCF project was funded by the Science and Engineering Research Council run by Mike Gordon in Cambridge and Robin Milner in Edinburgh
- David Schmidt and later Lincoln Wallen hired as RAs in Edinburgh

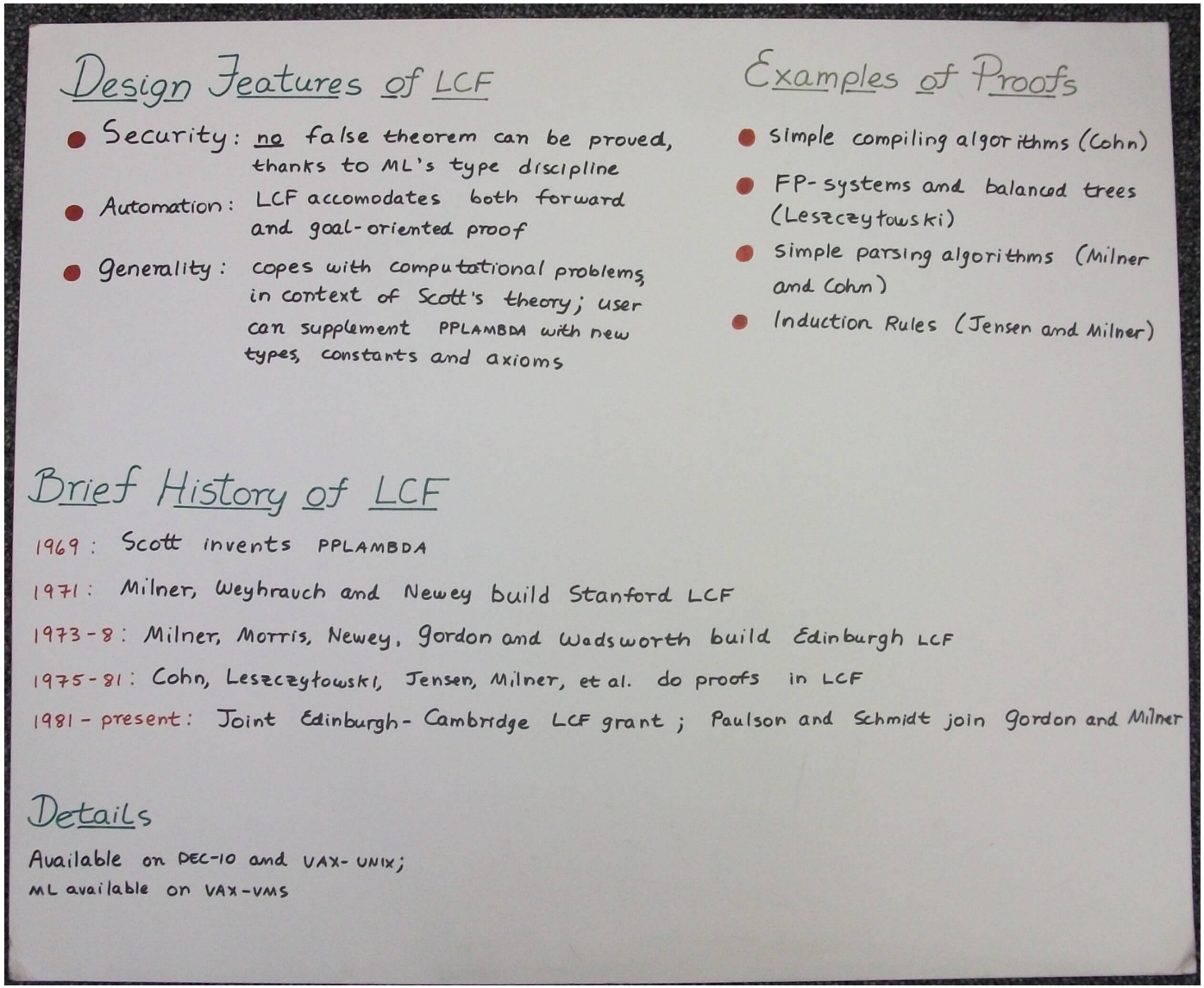
Move to Cambridge

In 1981 Mike returned to Cambridge to take up a lectureship at the University Computer Laboratory, bringing Avra and Edinburgh LCF with him

- A joint LCF project was funded by the Science and Engineering Research Council run by Mike Gordon in Cambridge and Robin Milner in Edinburgh
- David Schmidt and later Lincoln Wallen hired as RAs in Edinburgh
- Larry Paulson hired as RA in Cambridge

Avra Cohn's LCF Poster

Avra was the first real LCF user, doing her PhD on verifying programming language implementations with LCF.



Design Features of LCF

- Security: no false theorem can be proved, thanks to ML's type discipline
- Automation: LCF accomodates both forward and goal-oriented proof
- Generality: copes with computational problems in context of Scott's theory; user can supplement PPLAMBDA with new types, constants and axioms

Examples of Proofs

- simple compiling algorithms (Cohn)
- FP-systems and balanced trees (Leszczytowski)
- simple parsing algorithms (Milner and Cohn)
- Induction Rules (Jensen and Milner)

Brief History of LCF

1969: Scott invents PPLAMBDA

1971: Milner, Weyhrauch and Newey build Stanford LCF

1973-8: Milner, Morris, Newey, Gordon and Wadsworth build Edinburgh LCF

1975-81: Cohn, Leszczytowski, Jensen, Milner, et al. do proofs in LCF

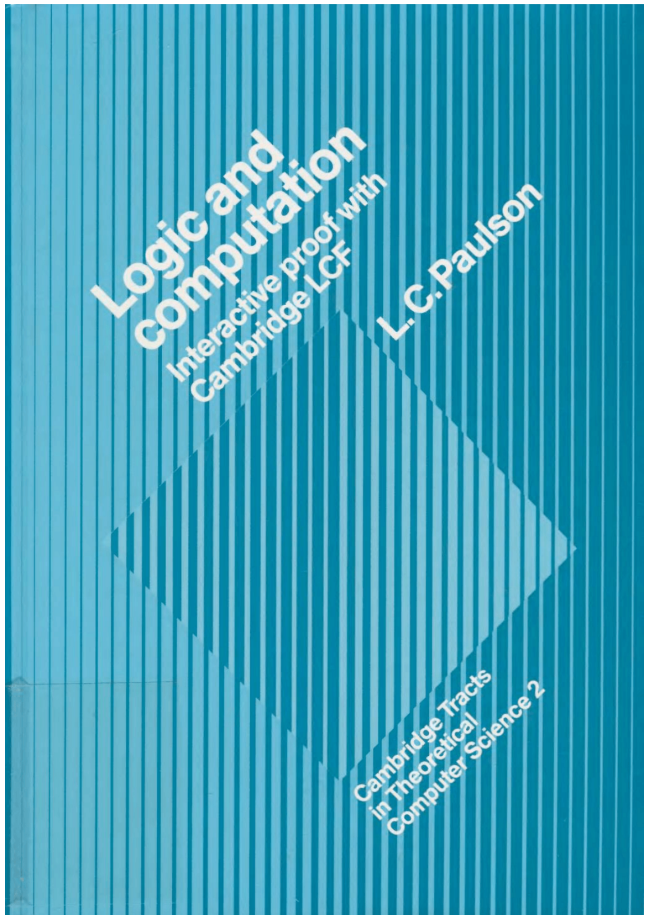
1981-present: Joint Edinburgh-Cambridge LCF grant; Paulson and Schmidt join Gordon and Milner

Details

Available on DEC-10 and VAX-UNIX;
ML available on VAX-VMS

Cambridge LCF

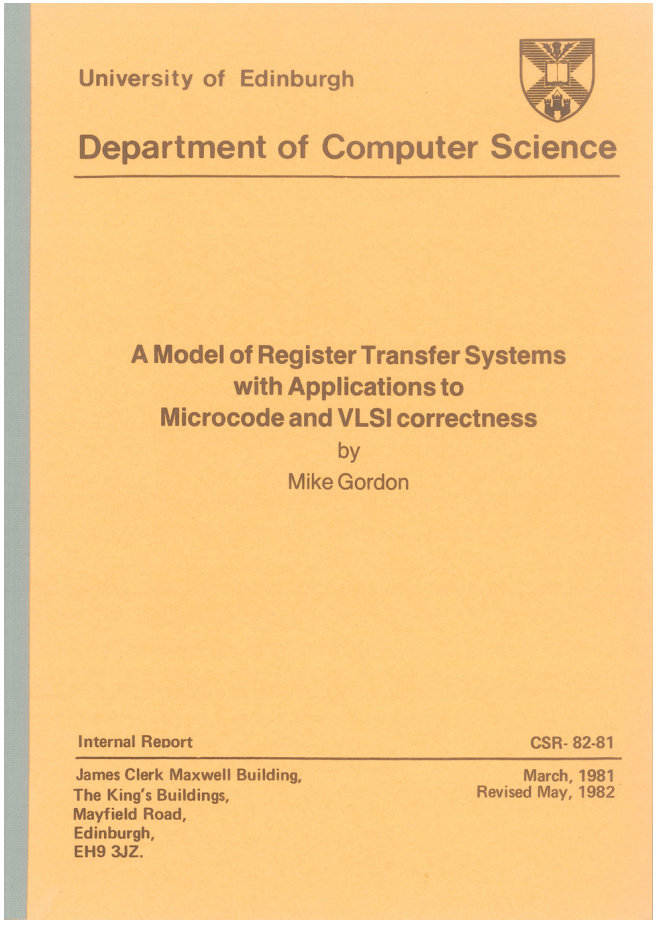
Larry Paulson worked with Gérard Huet on substantial rewrite of LCF and contributed many architectural ideas like *conversions*, leading to a much improved system *Cambridge LCF*:



He later went on to develop Isabelle. . .

Hardware verification

Mike was already working on formal models of hardware RTL around the time of his move from Edinburgh to Cambridge



University of Edinburgh

Department of Computer Science

A Model of Register Transfer Systems with Applications to Microcode and VLSI correctness

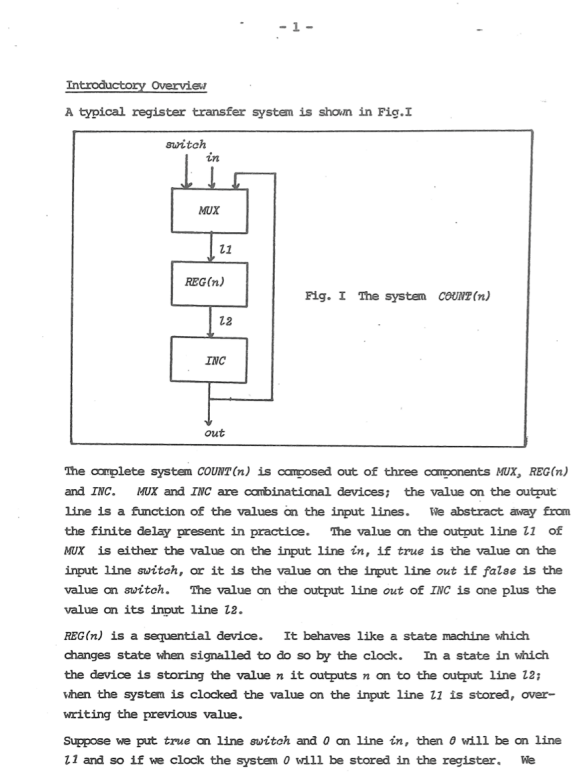
by

Mike Gordon

Internal Report CSR- 82-81

James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

March, 1981
Revised May, 1982



Introductory Overview

A typical register transfer system is shown in Fig.I

Fig. I The system *COUNT(n)*

The complete system *COUNT(n)* is composed out of three components *MUX*, *REG(n)* and *INC*. *MUX* and *INC* are combinational devices; the value on the output line is a function of the values on the input lines. We abstract away from the finite delay present in practice. The value on the output line *l1* of *MUX* is either the value on the input line *in*, if *true* is the value on the input line *switch*, or it is the value on the input line *out* if *false* is the value on *switch*. The value on the output line *out* of *INC* is one plus the value on its input line *l2*.

REG(n) is a sequential device. It behaves like a state machine which changes state when signalled to do so by the clock. In a state in which the device is storing the value *n* it outputs *n* on to the output line *l2*; when the system is clocked the value on the input line *l1* is stored, overwriting the previous value.

Suppose we put *true* on line *switch* and *0* on line *in*, then *0* will be on line *l1* and so if we clock the system *0* will be stored in the register. We

LSM

"I invented a rather ad hoc *notation (called "LSM" for Logic of Sequential Machines) for denoting sequential machine behaviour, together with a law manipulatively similar to CCS's Expansion Theorem. To provide a proof assistant for LSM, I lashed up a version of Cambridge LCF (called LCF_LSM) that added parsing and pretty-printer support for LSM and provided the expansion-law as an additional axiom scheme."*

LSM

"I invented a rather ad hoc *notation (called "LSM" for Logic of Sequential Machines) for denoting sequential machine behaviour, together with a law manipulatively similar to CCS's Expansion Theorem. To provide a proof assistant for LSM, I lashed up a version of Cambridge LCF (called LCF_LSM) that added parsing and pretty-printer support for LSM and provided the expansion-law as an additional axiom scheme."*

"This lash-up worked quite well and even got used outside Cambridge. I used it to verify a toy microprocessor subsequently called Tamarack, and the LSM notation was used by a group in the Royal Signals and Radar Establishment (RSRE) to specify the ill-fated Viper processor."

From LCF_LSM to HOL

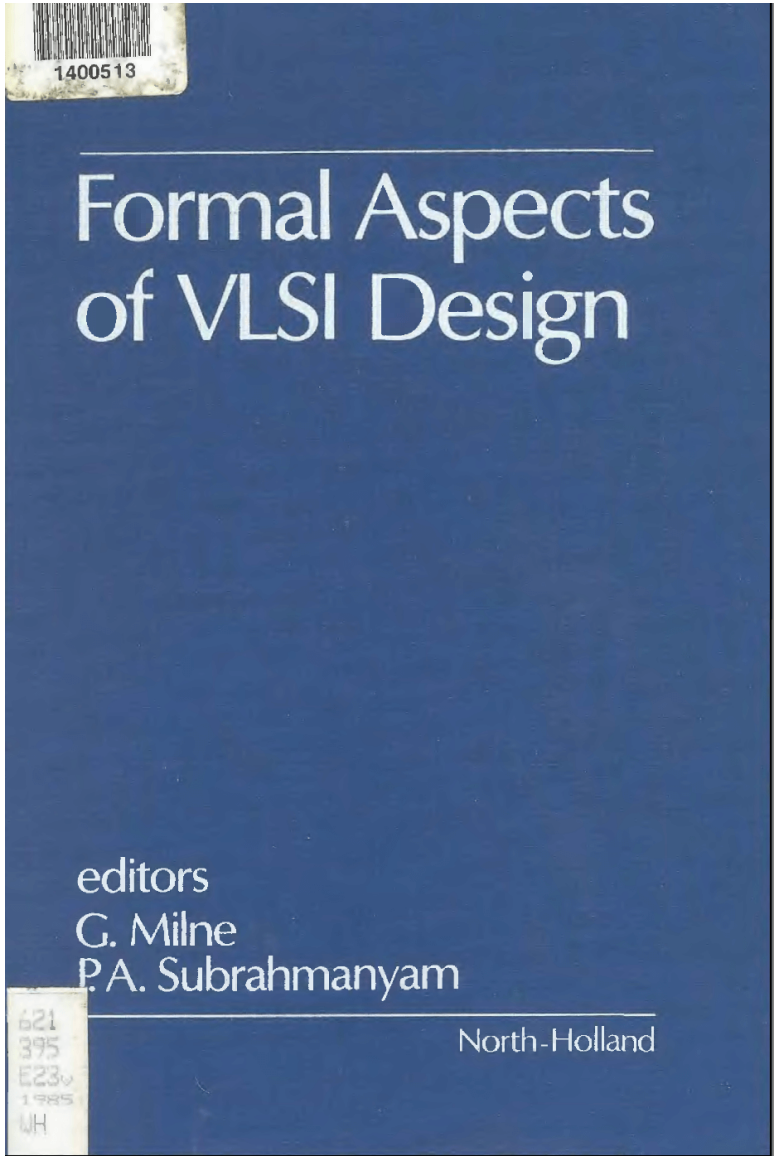
> *"During this time Ben Moskowski, who had recently graduated from Stanford, was doing a postdoc at Cambridge. He showed me how the terms of LSM could be encoded in predicate calculus in such a way that the LSM expansion-law just becomes a derived rule (i.e. corresponds to a sequence of standard predicate calculus inferences). This approach is both more elegant and rests on a firmer logical foundation, so I switched to it and HOL was born."*

Mike credited Keith Hanna with the idea of using higher-order logic (HOL) as a natural way of formalizing sequential circuits via 'functions from time'

The genesis of HOL

"The design of HOL was largely taken 'off the shelf' the theory being classical higher order logic and the implementation being LCF . . . HOL terms were encoded as LCF constructs in a way designed to support maximum reuse of LCF code (the encoding did not represent any coherent domain-theoretic semantics)."

The Higher Order Logic Manifesto



Formal Aspects of VLSI Design
G.J. Milne and P.A. Subrahmanyam (editors)

Why higher-order logic is a good formalism for specifying and verifying hardware

Mike Gordon
Computer Laboratory
Corn Exchange Street, Cambridge CB2 3QG

Higher-order logic was originally developed as a foundation for mathematics. We show how it can be used both as a hardware description language, and as a deductive system for proving that designs meet their specifications. Examples used to illustrate various specification and verification techniques include a CMOS inverter, a CMOS full adder, an n-bit ripple-carry adder, a sequential multiplier and an edge-triggered Dtype register.

1. Introduction

The purpose of this paper is to show, via examples, that many kinds of digital systems can be formally specified using the notation of formal logic and, furthermore, that the inference rules of logic provide a practical means of proving system designs correct. We claim that there is no need for specialized hardware description languages or specialized deductive systems; 'pure logic' suffices.

The particular logical system used here is called higher-order logic. It is hoped that Section 2 below will enable readers who are not familiar with predicate calculus to understand what follows. Thorough introductions to higher-order logic can be found in textbooks on mathematical logic [8], in Church's original paper [1], or in the report on the HOL logic [4].

2. Introduction to higher-order logic

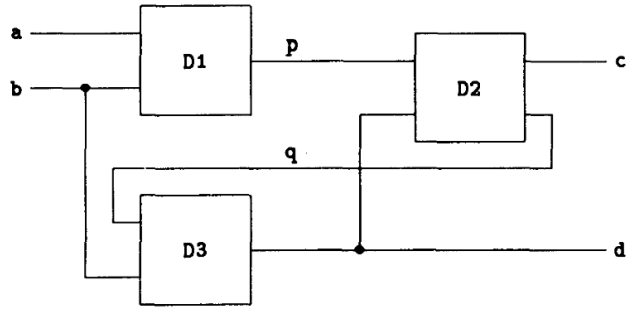
Higher-order logic uses standard predicate logic notation:

- "$P(x)$" means "x has property P",
- "$\neg t$" means "not t",
- "$t_1 \vee t_2$" means "t_1 or t_2",
- "$t_1 \wedge t_2$" means "t_1 and t_2",
- "$t_1 \supset t_2$" means "t_1 implies t_2",

Predicate representation of circuits

4. Representing circuit structure with predicates

Consider the following structure (called D):



This device is built by connecting together three component devices D1, D2 and D3. The external lines of D are a, b, c and d. The lines p and q are internal and are not connected to the 'outside world'. (External lines might correspond to the pins of an integrated circuit, and internal lines to tracks.)

Suppose the behaviours of D1, D2 and D3 are specified by predicates D_1, D_2 and D_3 respectively. How can we derive the behaviour of the system D shown above? Each device constrains the values on its lines. If a, b and p denote the values on the lines a, b and p, then D1 constrains these values so that $D_1(a, b, p)$ holds. To get the constraint imposed by the whole device D we just conjoin (*i.e.* $\wedge$-together) the constraints imposed by D1, D2 and D3; the combined constraint is thus:

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

This expression constrains the values on both the external lines a, b, c and d and the internal lines p and q. If we regard D as a 'black box' with the internal lines invisible, then we are really only interested in what constraints are imposed on its external lines. The variables a, b, c and d will denote possible values at the external lines a, b, c and d if and only if the conjunction above holds *for some* values p and q. We can therefore define a predicate D representing the behaviour of D by:

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

Sequential circuits via functions

8. Sequential Devices

All of the examples so far have been combinational; *i.e.* the values on the outputs have only depended on the current input values, not on input values at past times. Sequential devices can be modelled by taking the values on lines to be functions of time. For example, a unit-delay element `Del`, with input line `i` and output line `o`, is modelled by specifying that the value output at time $t+1$ equals the value input at time t. This is expressed in higher-order logic by:

$$\mathsf{Del}(i, o) \equiv \forall t.\ o(t+1) = i(t)$$

Combinational devices can be modelled as sequential devices having no delay. To illustrate this, recall the specification of the adder:

$$\begin{aligned}&\mathsf{Adder}(n)(a, b, cin, sum, cout) \equiv\\ &(2^{n+1} \times \mathsf{Bit\_Val}(cout) + \mathsf{Val}(n, sum) =\\ &\ \mathsf{Val}(n, a) + \mathsf{Val}(n, b) + \mathsf{Bit\_Val}(cin))\end{aligned}$$

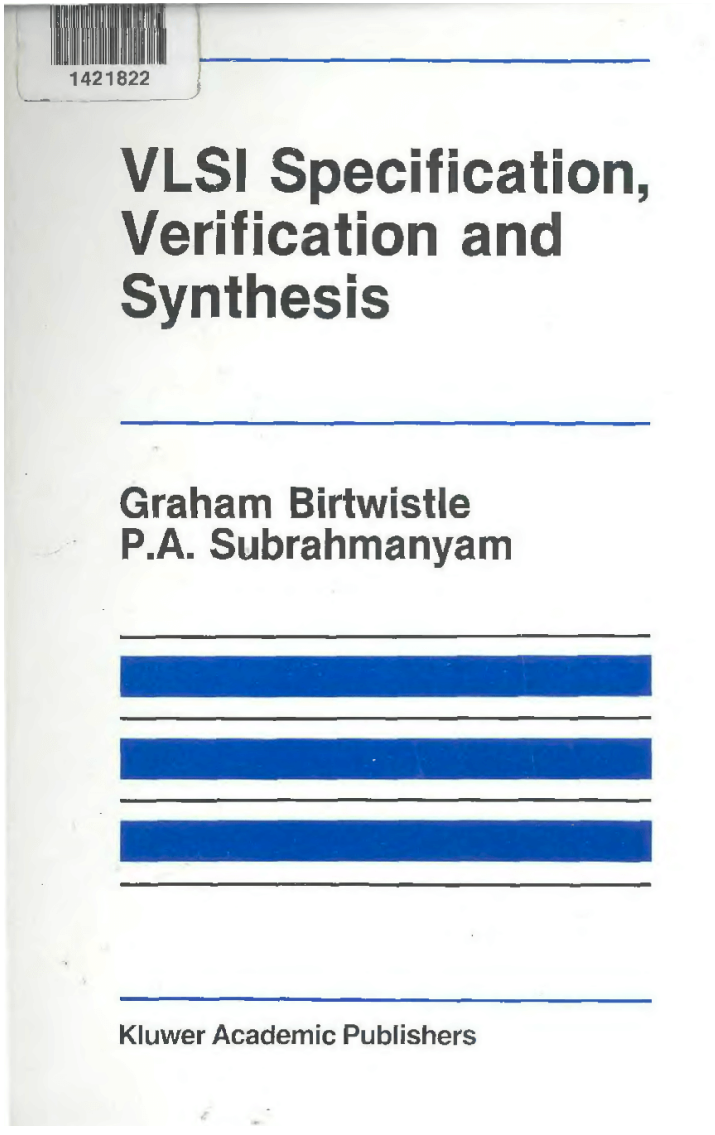
The variables a, b and sum range over words (modelled as functions) and the variables cin and $cout$ range over truth-values. To model the adder as a zero-delay sequential device we must represent its behaviour with a predicate whose arguments are functions of time.

$$\begin{aligned}&\mathsf{Combinational\_Adder}(n)(a', b', cin', sum', cout') \equiv\\ &\ \forall t.\ \mathsf{Adder}(n)(a'(t), b'(t), cin'(t), sum'(t), cout'(t))\end{aligned}$$

The variables a', b' and sum' range over functions from time to words, and the variables cin' and $cout'$ range over functions from time to truth-values. Thus, for example, $out'(7)(5)$ is bit 5 of the word output at time 7. If we wanted to specify the adder as having a unit-delay then we could define:

$$\begin{aligned}&\mathsf{Unitdelay\_Adder}(n)(a', b', cin', sum', cout') \equiv\\ &\ \forall t.\ \mathsf{Adder}(n)(a'(t), b'(t), cin'(t), sum'(t+1), cout'(t+1))\end{aligned}$$

HOL introduced to the world



3

HOL: A Proof Generating System for Higher-Order Logic

Michael J.C. Gordon

University of Cambridge
Computer Laboratory
Corn Exchange Street
Cambridge, CB2 3QG
England.

Abstract: HOL is a version of Robin Milner's LCF theorem proving system for higher-order logic. It is currently being used to investigate (1) how various levels of hardware behaviour can be rigorously modelled and (2) how the resulting behavioral representations can be the basis for verification by mechanized formal proof. This paper starts with a tutorial introduction to the meta-language ML. The version of higher-order logic implemented in the HOL system is then described. This is followed by an introduction to goal-directed proof with *tactics* and *tacticals*. Finally, there is a little example of the system in action which illustrates how HOL can be used for hardware verification.

1 Introduction to HOL

Higher-order logic is a promising language for specifying all aspects of hardware behaviour [8], [1]. It was originally developed as a foundation for mathematics [2]. Its use for hardware specification and verification was first advocated by Keith Hanna [9].

The approach to mechanising logic described in this paper is due to Robin Milner [6]. He originally developed the approach for a system called LCF designed for reasoning about higher-order recursively defined functions. The HOL system is a direct descendant of this work. The original LCF system was implemented at Edinburgh and is called "Edinburgh LCF". The code for it was ported from Stanford Lisp to Franz Lisp by Gerard Huet at INRIA and formed the basis for a French research project called "Formel". Huet's Franz Lisp version of LCF was further developed at Cambridge by Larry Paulson and eventually became known as "Cambridge LCF" [18]. The HOL system is implemented on top of Cambridge LCF and consequently many (good and bad) features of LCF are found in it. In particular, the axiomatization of higher-order logic used is not the classical one due to Church, but an equivalent formulation strongly influenced by LCF.

Derived inference rules

It is sometimes claimed that 'LCF-style' systems can never be practical, because the efficiency needed to handle real examples can only be obtained with decision procedures coded as primitive rules. . . . the truth of such claims is not obvious. Research is currently in progress to see if a variety of practical decision algorithms can be implemented as efficient derived rules.

Derived inference rules

It is sometimes claimed that 'LCF-style' systems can never be practical, because the efficiency needed to handle real examples can only be obtained with decision procedures coded as primitive rules. . . . the truth of such claims is not obvious. Research is currently in progress to see if a variety of practical decision algorithms can be implemented as efficient derived rules.

- Linear arithmetic and combined decision procedures (Boulton)
- First-order automation (Kumar, Hurd)
- Knuth-Bendix completion (Slind)
- Boyer-Moore automated induction (Boulton, Papapanagiotou)
- Certified computer algebra (Harrison and Théry)
- Algebraic theories via Gröbner bases (Harrison)

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.
One of the innovations of Mike's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.
One of the innovations of Mike's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

This is the standard approach in mathematics, even if most of the time people don't bother about it (e.g. the construction of the real numbers as Dedekind cuts or whatever).

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.
One of the innovations of Mike's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

This is the standard approach in mathematics, even if most of the time people don't bother about it (e.g. the construction of the real numbers as Dedekind cuts or whatever).
Just using axioms was compared by Russell to theft in place of honest toil.

Definitions: object or meta?

It's often suggested that formalized mathematics is impractical because terms hidden inside definitions become ridiculously large. Bourbaki may even have underestimated:

A term of length 4,523,659,424,929

A. R. D. MATHIAS

Universidad de los Andes, Santa Fé de Bogotá and Humboldt Universität zu Berlin

Abstract Bourbaki suggest that their definition of the number 1 runs to some tens of thousands of symbols. We show that that is a considerable under-estimate, the true number of symbols being that in the title, not counting 1,179,618,517,981 disambiguatory links.

Definitions: object or meta?

It's often suggested that formalized mathematics is impractical because terms hidden inside definitions become ridiculously large. Bourbaki may even have underestimated:

A term of length 4,523,659,424,929

A. R. D. MATHIAS
Universidad de los Andes, Santa Fé de Bogotá and Humboldt Universität zu Berlin

Abstract Bourbaki suggest that their definition of the number 1 runs to some tens of thousands of symbols. We show that that is a considerable under-estimate, the true number of symbols being that in the title, not counting 1,179,618,517,981 disambiguatory links.

Instead of considering them as *metalogical* abbreviations, HOL considers definitions as *object-level* notions extending the logic signature in a conservative way

- A new constants can be defined corresponding to existing terms
- A new type can be defined in bijection with a nonempty subset of an existing type

Definitions: object or meta?

It's often suggested that formalized mathematics is impractical because terms hidden inside definitions become ridiculously large. Bourbaki may even have underestimated:

A term of length 4,523,659,424,929

A. R. D. MATHIAS
Universidad de los Andes, Santa Fé de Bogotá and Humboldt Universität zu Berlin

Abstract Bourbaki suggest that their definition of the number 1 runs to some tens of thousands of symbols. We show that that is a considerable under-estimate, the true number of symbols being that in the title, not counting 1,179,618,517,981 disambiguatory links.

Instead of considering them as *metalogical* abbreviations, HOL considers definitions as *object-level* notions extending the logic signature in a conservative way

- A new constants can be defined corresponding to existing terms
- A new type can be defined in bijection with a nonempty subset of an existing type

Even these simple rules have their subtleties as Roger Jones pointed out . . .

Derived definitional principles

Many theorem provers build in complex definitional principles as basic, and it's a common source of soundness bugs to get them slightly wrong.

In HOL systems these are all *derived* from the primitive ones

Derived definitional principles

Many theorem provers build in complex definitional principles as basic, and it's a common source of soundness bugs to get them slightly wrong.
In HOL systems these are all *derived* from the primitive ones

- Definition of inductive types like trees, lists etc. (Melham)
- Inductive definitions of sets and predicates (Andersen, Melham)
- Definition of general recursive functions using wellfounded orderings (Ploegerts, Slind)

Derived definitional principles

Many theorem provers build in complex definitional principles as basic, and it's a common source of soundness bugs to get them slightly wrong.
In HOL systems these are all *derived* from the primitive ones

- Definition of inductive types like trees, lists etc. (Melham)
- Inductive definitions of sets and predicates (Andersen, Melham)
- Definition of general recursive functions using wellfounded orderings (Ploegerts, Slind)

Instrumental in widening the range of applications and encompassing 'functional programming'.

Embedding programming logics

G. Birtwistle P.A. Subrahmanyam
Editors

Current Trends in Hardware Verification and Automated Theorem Proving

With 95 Figures

Springer-Verlag
New York Berlin Heidelberg
London Paris Tokyo

Mechanizing Programming Logics in Higher Order Logic

Michael J.C. Gordon

Computer Laboratory	SRI International
New Museums Site	Suite 23
Pembroke Street	Millers Yard
Cambridge CB2 3QG	Cambridge CB2 1RQ

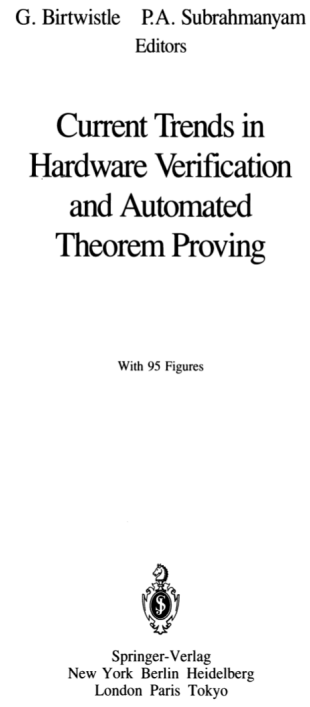
Abstract:

Formal reasoning about computer programs can be based directly on the semantics of the programming language, or done in a special purpose logic like Hoare logic. The advantage of the first approach is that it guarantees that the formal reasoning applies to the language being used (it is well known, for example, that Hoare's assignment axiom fails to hold for most programming languages). The advantage of the second approach is that the proofs can be more direct and natural.

In this paper, an attempt to get the advantages of both approaches is described. The rules of Hoare logic are *mechanically* derived from the semantics of a simple imperative programming language (using the HOL system). These rules form the basis for a simple program verifier in which verification conditions are generated by LCF-style tactics whose validations use the derived Hoare rules. Because Hoare logic is derived, rather than postulated, it is straightforward to mix semantic and axiomatic reasoning. It is also straightforward to combine the constructs of Hoare logic with other application-specific notations. This is briefly illustrated for various logical constructs, including termination statements, VDM-style 'relational' correctness specifications, weakest precondition statements and dynamic logic formulae .

The theory underlying the work presented here is well known. Our contribution is to propose a way of mechanizing this theory in a way that makes certain practical details work out smoothly.

Embedding programming logics

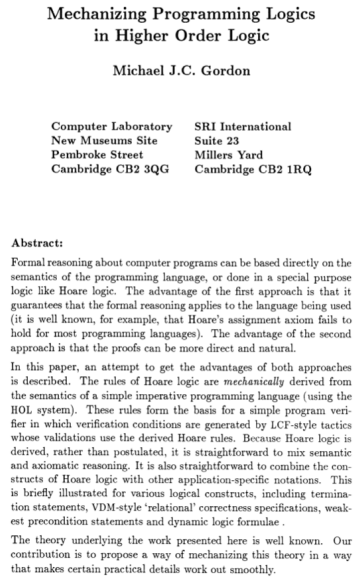


G. Birtwistle P.A. Subrahmanyam
Editors

Current Trends in
Hardware Verification
and Automated
Theorem Proving

With 95 Figures

Springer-Verlag
New York Berlin Heidelberg
London Paris Tokyo



Mechanizing Programming Logics
in Higher Order Logic

Michael J.C. Gordon

Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG

SRI International
Suite 23
Millers Yard
Cambridge CB2 1RQ

Abstract:

Formal reasoning about computer programs can be based directly on the semantics of the programming language, or done in a special purpose logic like Hoare logic. The advantage of the first approach is that it guarantees that the formal reasoning applies to the language being used (it is well known, for example, that Hoare's assignment axiom fails to hold for most programming languages). The advantage of the second approach is that the proofs can be more direct and natural.

In this paper, an attempt to get the advantages of both approaches is described. The rules of Hoare logic are *mechanically* derived from the semantics of a simple imperative programming language (using the HOL system). These rules form the basis for a simple program verifier in which verification conditions are generated by LCF-style tactics whose validations use the derived Hoare rules. Because Hoare logic is derived, rather than postulated, it is straightforward to mix semantic and axiomatic reasoning. It is also straightforward to combine the constructs of Hoare logic with other application-specific notations. This is briefly illustrated for various logical constructs, including termination statements, VDM-style 'relational' correctness specifications, weakest precondition statements and dynamic logic formulae .

The theory underlying the work presented here is well known. Our contribution is to propose a way of mechanizing this theory in a way that makes certain practical details work out smoothly.

Other embeddings include program refinement (von Wright), CCS (Nesi), CSP (Camilleri), TLA (von Wright), UNITY (Andersen), ELLA (Boulton), VHDL (van Tassel), Silage (Andy Gordon), π-calculus (Melham), Z (Bowen-Gordon, Arthan), Verilog (Stewart), PSL/Sugar (Gordon) ...

Deep and shallow embeddings

Experience with embedding hardware description languages in HOL

Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, John Van Tassel

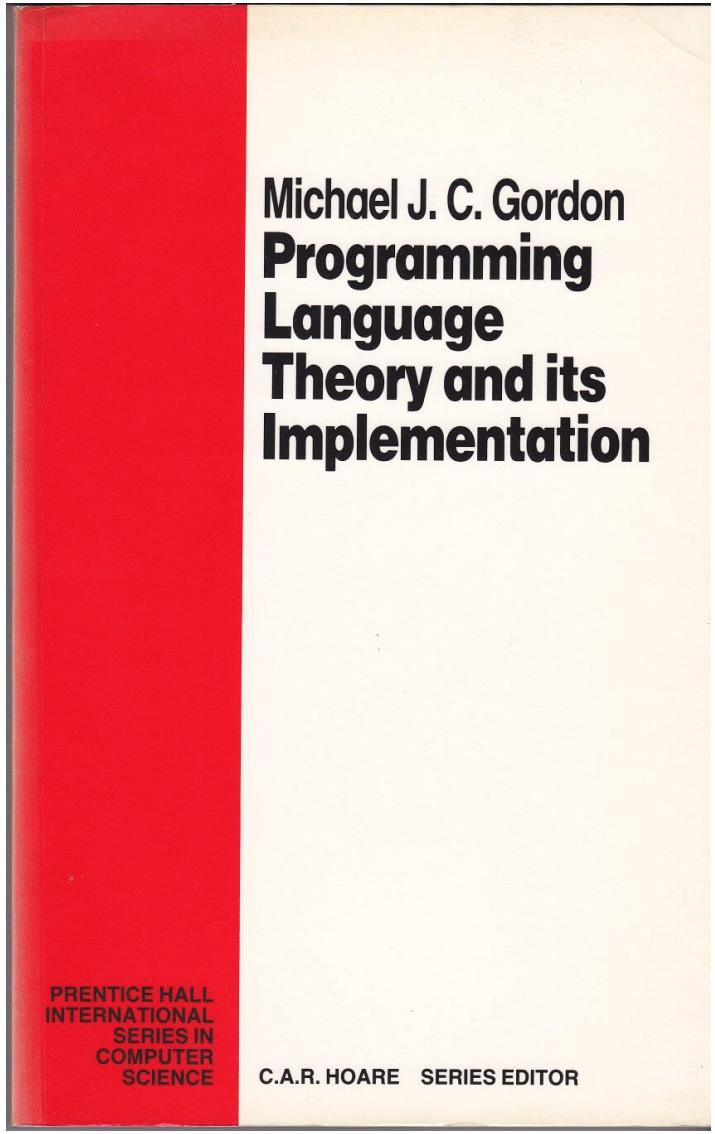
University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England.

Abstract

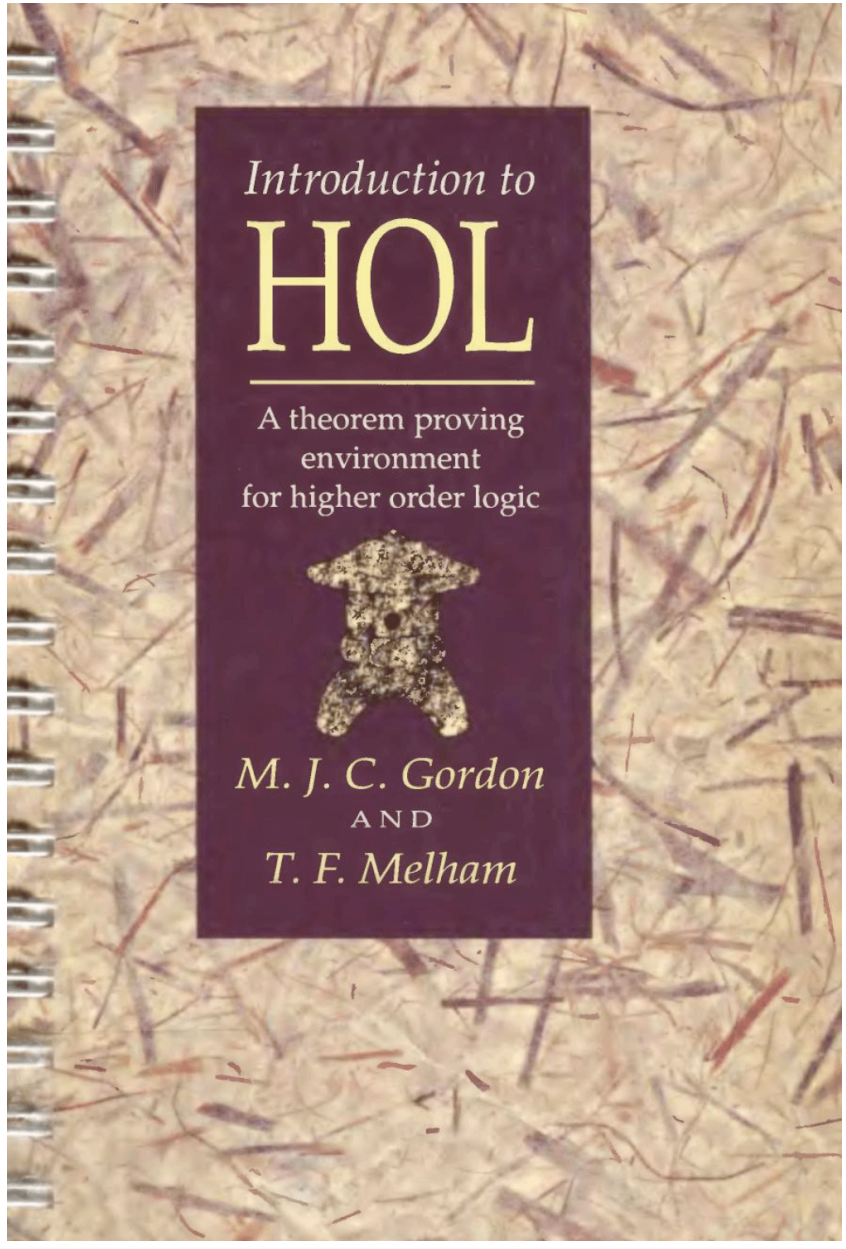
The semantics of hardware description languages can be represented in higher order logic. This provides a formal definition that is suitable for machine processing. Experiments are in progress at Cambridge to see whether this method can be the basis of practical tools based on the HOL theorem-proving assistant. Three languages are being investigated: ELLA, SILAGE and VHDL. The approaches taken for these languages are compared and current progress on building semantically-based theorem-proving tools is discussed.

Keyword Codes: F.4.1; B.7.2; I.2.3
Keywords: Mathematical Logic; Integrated Circuits, Design Aids; Deduction and Theorem Proving.

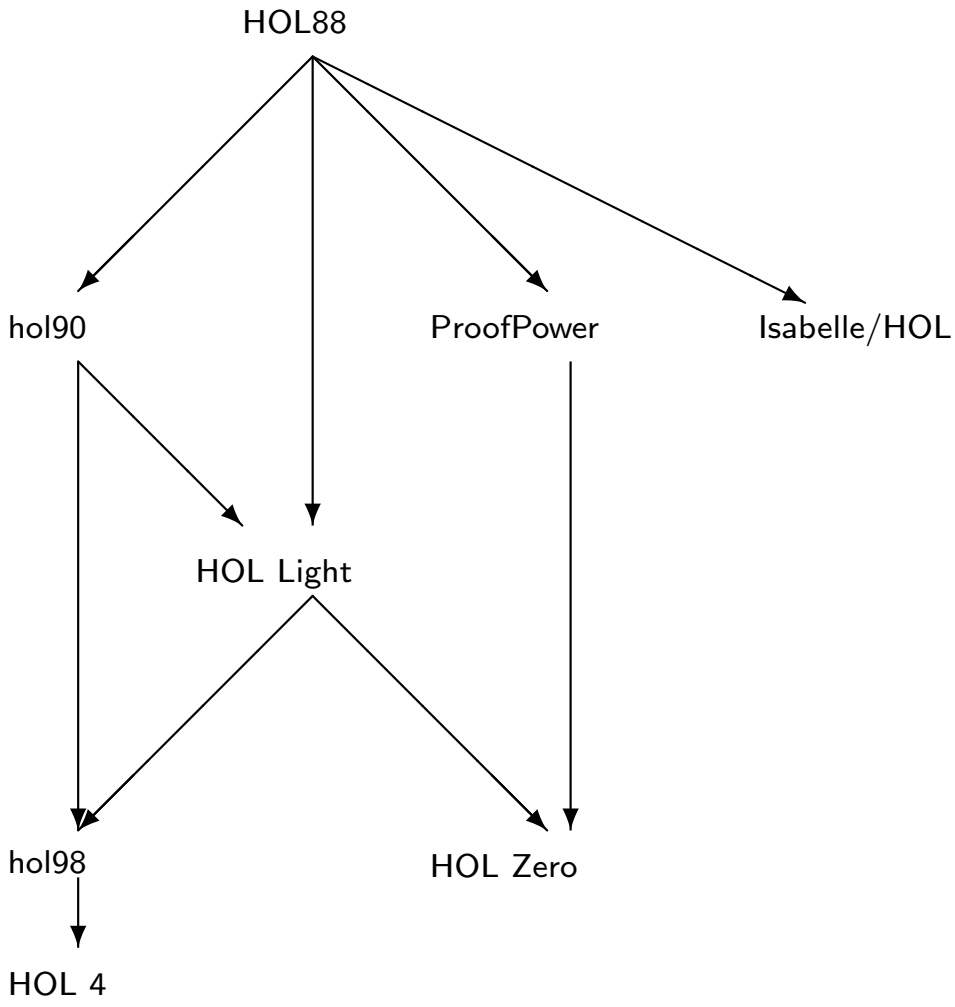
Programming language book



The seminal HOL88



Other versions of HOL



Applications of HOL

A small sample of verification applications:

- Formal verification of a 'functional' SECD machine (Graham)
- Formal model of ARM architecture and proofs of design correctness for ARM6 (Fox)
- Probabilistic algorithms (Hurd)
- Floating-point algorithms (Harrison)
- CakeML verified functional language compiler (Kumar, Myreen, Norrish, Owens, . . .)
- Machine code correctness proofs (Myreen)
- Self-verification of HOL itself (Kumar)

Mathematics in HOL

HOL has also been used to formalize pure mathematics, albeit often for verification applications, e.g. real analysis (Harrison), measure theory and probability (Hurd), and most impressively:

Mathematics in HOL

HOL has also been used to formalize pure mathematics, albeit often for verification applications, e.g. real analysis (Harrison), measure theory and probability (Hurd), and most impressively:

The Flyspeck project to verify the Hales-Ferguson proof of the Kepler sphere-packing conjecture.

Later Life

http://www.cl.cam.ac.uk/archive/mjcg/plans/ClassReunion.html

http://www.cl.cam.ac.uk/archive/mjcg/plans/Report0.html

http://www.cl.cam.ac.uk/archive/mjcg/plans/Report1.html

http://www.cl.cam.ac.uk/archive/mjcg/plans/MITB.html

Later Academic Career

While continuing to supervise students (including Distinguished Dissertation Award winners) Mike steadily accumulated academic honours

- Reader in Computer Science 1988
- Fellow of the Royal Society 1994
- Professor of Automated Reasoning 1996

He also continued to explore research ideas like linking HOL4 and ACL2 and the MITB ('MAC in the box') project.

60th birthday Festschrift

In March 2008, we celebrated Mike's 60th birthday at the Royal Society with talks on many topics in verification and formalization of mathematics:

Retirement

Mike eased his way into retirement from the Computer Laboratory in 2016

> *I feel pretty happy at the moment and I want to remain so as I age . . . I wrote in my retirement-planning web page that retirement feels like being on sabbatical without any worries that time is running out and academic chores will soon be back . . . it still does.*

Michael John Caldwell Gordon, 28 February 1948 – 22 August 2017

THE AUTOMATED REASONING GROUP IS LOOKING FOR PHD INTERNS!

Amazon Web Services 2019

Locations: seattle, new york, boston, cupertino, minneapolis
Timeline: 12-week internships with flexible start dates in 2019
Candidate requirements:

- enrolled in a programming languages or formal methods phd program at the time of application and while interning
- programming and software development experience
- interested in gaining hands-on experience with cloud computing and amazon web services

email **CV** to **nedic@amazon.com** **AND** apply via
https://www.amazonuniversity.jobs/?JobId=676519

aws