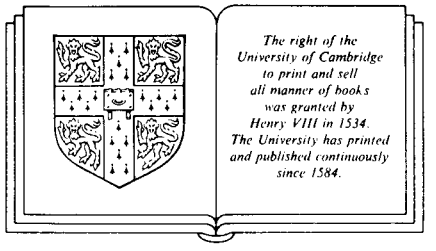
LOGIC AND COMPUTATION

Interactive Proof with Cambridge LCF

LAWRENCE C. PAULSON
Computer Laboratory, University of Cambridge



CAMBRIDGE UNIVERSITY PRESS
Cambridge
New York Port Chester
Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1987
First paperback edition 1990

British Library cataloguing in publication data
Paulson, Lawrence C.
Logic and computation: interactive proof with Cambridge LCF — (Cambridge tracts in theoretical computer science).
1. Cambridge LCF (Computer system)
2. Computable functions – Data processing
I. Title
004.1'25 QA9.59
Library of Congress cataloguing in publication data available

ISBN 0 521 34632 0 hardback
ISBN 0 521 39560 7 paperback

Transferred to digital printing 2003

Contents

Chapter 1

Survey and History of LCF

Cambridge LCF is an interactive theorem prover for reasoning about computable functions. The terms of its logic, PPλ, constitute a tiny functional programming language. Cambridge LCF can be used for experimenting with first order proof; studying abstract constructions in domain theory; comparing the denotational semantics of programming languages; and verifying functional programs. It can reason about both strict and lazy evaluation.

There are many theorem provers in the LCF family. Each is descended from Edinburgh LCF and retains its fundamental ideas:

- The user interacts with the prover through a programmable meta language, ML.
- Logical formulae, theorems, rules, and proof strategies are ML data.
- The prover guarantees soundness: it checks each inference and records each assumption.

Edinburgh LCF was developed in order to experiment with Scott's Logic of Computable Functions [41]. It performed many proofs involving denotational semantics and functional programming.

Cambridge LCF extended the logic of Edinburgh LCF with $\vee$, $\exists$, $\Longleftrightarrow$, and predicates, improved the efficiency, and added several inference mechanisms. It has been used in proofs about functional programming and several other theorem provers have been built from it.

LCF_LSM was developed for reasoning about digital circuits in a formalism related to CCS [32]. It verified some realistic devices [39], but is now obsolete.

HOL supports Church's Higher Order Logic [37], a general mathematical formalism. It is mainly used to prove digital circuits correct.

Another system supports a higher order *Calculus of Constructions* that can represent many other calculi. Coquand and Huet's examples include definitions of logical connectives and data types [27]. Proofs include Tarski's fixed point theorem and Newman's lemma about confluence of Noetherian relations.

The *Gothenburg Type Theory* system [81] supports Martin-Löf's Constructive Type Theory [65,74], a logic and specification language for computation.

Nuprl supports a similar constructive type theory [26]. It is the most sophisticated of all these systems. Proofs are edited using windows and mouse. From a proof of 'for all x there exists some y' it can extract a program to compute this function.

1.1 The structure of LCF

A typical version of LCF includes the meta language ML, a logic such as PPλ, subgoaling functions (tactics and tacticals), a simplifier, and commands for maintaining logical theories.

1.1.1 The meta language ML

LCF has a *programmable* meta language, ML. Every command is an ML function. Writing ML code adds commands and functions to LCF. ML and a few critical functions are implemented in Lisp. The standard rules and tactics consist of 5000 lines of ML.

ML is an important spin-off from LCF. Designed specifically for theorem proving, it applies to the whole area of symbolic computation. Its data structures allow tree and list processing rather like Lisp. ML supports functional programming; it includes higher order functions. ML supports imperative programming; it includes references and assignments. Exceptions (or *failures*) can be raised and handled. Exceptions allow a function to signal that it cannot compute a result, so that an alternative function can be tried.

ML was the first language to use Milner's polymorphic type system [67]. It prevents run-time type errors while retaining much of the flexibility of untyped languages. The programmer may state type restrictions or omit them. A *polymorphic* type contains type variables; a value of that type has also every instance of that type, obtained by substituting types for type variables. For instance, the function *length*, for taking the length of a list, has type $(\alpha\ list) \to int$. This function can be applied to a list of elements of any type, and returns the type *int*.

After many dialects started to appear, Milner convened the ML community to develop a Standard [45]. Standard ML is larger and more powerful than its predecessor. The main extensions are recursive data structures, function definitions by patterns, exceptions of arbitrary type, reference types, and modules. Cambridge LCF was converted to Standard ML in 1987. Although all interaction with LCF takes place via ML, only advanced users need to write serious programs.

1.1.2 Inference in the logic, PPλ

Most LCF proofs are conducted in PPλ, a first order logic for domain theory. Theorems are proved with respect to assumptions, the *natural deduction* formalization. If $A_1, \ldots, A_n, B$ are formulae, then $A_1, \ldots, A_n \vdash B$ means B is true if the assumptions $A_1, \cdots, A_n$ are all true.

The logic is embedded in ML as an ML data type *form* of formulae, with axioms and rules for proving theorems. Logical formulae are ML values: functions can take them apart and put them together. Theorems are values of the abstract data type *thm*. Type-checking guarantees that a theorem is constructed by axioms and inference rules, not by manipulation of its representation. Axioms are predeclared ML identifiers; rules are functions from theorems to theorems.

1.1.3 Tactics and tacticals

Applying inference rules to theorems produces other theorems. This is *forwards* proof. Most people work in the *backwards* direction. Start with a *goal*, the theorem to be proved. Reduce goals to simpler subgoals until all have been solved. Functions called *tactics* reduce goals to subgoals. A complete tactical proof may be imagined as a tree whose nodes are goals and whose leaves are known theorems.

The tactic *CONJ_TAC* reduces any goal of the form $A_1, \ldots, A_n \vdash A \wedge B$ to the two goals $A_1, \ldots, A_n \vdash A$ and $A_1, \ldots, A_n \vdash B$. It *fails* (raises an exception) if the goal is not a conjunction.

The tactic *DISCH_TAC* reduces the goal $A_1, \ldots, A_n \vdash A \Longrightarrow B$ to the subgoal $A_1, \ldots, A_n, A \vdash B$, which is the goal of proving B assuming A plus the previous assumptions.

The tactic *ACCEPT_ASM_TAC* reduces the goal $A_1, \ldots, A_n \vdash A$ to an empty subgoal list if A is A_i for some i, and otherwise fails. A goal is solved when it is reduced to an empty subgoal list.

These three tactics can prove any goal of the form $\vdash A \Longrightarrow (B \Longrightarrow A \wedge B)$. Calling *DISCH_TAC* gives the goal $A \vdash B \Longrightarrow A \wedge B$. Calling *DISCH_TAC* again gives $A, B \vdash A \wedge B$. Calling *CONJ_TAC* gives the two goals $A, B \vdash A$ and $A, B \vdash B$. Both are solved by *ACCEPT_ASM_TAC*.

Operators called *tacticals* combine tactics into larger ones. *THEN* combines two tactics *sequentially*; *ORELSE* combines two *alternative* tactics; *REPEAT* makes a tactic *repetitive*. Tacticals can express the above proof in many ways. A literal rendering of the four steps is

DISCH_TAC THEN DISCH_TAC THEN CONJ_TAC THEN ACCEPT_ASM_TAC

A tactic that can perform many similar proofs is

REPEAT(*DISCH_TAC ORELSE CONJ_TAC ORELSE ACCEPT_ASM_TAC*)

Tactics and tacticals form a level of abstraction, in ML, above theorems and rules. A tactic that reduces the goal $\vdash A$ to the subgoals $\vdash B_1, \cdots, \vdash B_n$ also returns a function that takes the list of theorems $\vdash B_1, \cdots, \vdash B_n$ to the theorem $\vdash A$. Once every subgoal has been solved, these functions are put together to produce the desired theorem as value of type *thm*. Tacticals and interactive proof commands do this bookkeeping.

1.1.4 Theories

An LCF *theory* contains a signature: the names of constants, types, and predicates. It also contains a set of axioms. A logical theory contains all consequences of its axioms; an LCF theory contains a finite number of theorems, proved and stored by user command. An LCF theory is stored on a *theory file*, which can be loaded in a later session for proving additional theorems.

A new theory can be built above other theories, its *parents*. It can itself become the parent of later ones, forming a hierarchy of theories. Consider a theory *nat* of the natural numbers, and a theory *list* of lists. A theory defining the length of a list would have parents *nat* and *list*. A long proof involves months of interactive sessions and dozens of theories.

1.1.5 Rewriting

Most proofs rely heavily on the *simplifier*, which applies rewrite rules and eliminates tautologies in terms, formulae, or theorems. It is most commonly used, via a standard tactic, to simplify goals.

A theorem like[1]

$$\vdash f(x,y) \equiv g(x,h(y))$$

is a *rewrite rule*. The simplifier instantiates the variables x and y by pattern matching. Searching in the goal, it replaces every occurrence of $f(t,u)$ by $g(t,h(u))$. A theorem like

$$\vdash P(x,y) \Longrightarrow f(h(x),y) \equiv h(y)$$

is an *implicative* rewrite rule. The simplifier replaces $f(h(t),u)$ by $h\,u$ whenever it can prove $P(t,u)$ by recursively invoking simplification.

The simplifier makes use of local assumptions. When simplifying a formula like $f(x,y) \equiv g(x,h(y)) \Longrightarrow B$, the simplifier assumes the rewrite rule $f(x,y) \equiv g(x,h(y))$ while simplifying B.

Simplification is not arbitrary symbol crunching. Every step is justified by a theorem.

[1] In LCF, $f(x,y) \equiv g(x,h(y))$ means $f(x,y)$ equals $g(x,h(y))$ for all x and y.

1.2 A brief history

In 1969 Dana Scott developed a Logic for Computable Functions, formalizing a new mathematical model of computability. Through its careful treatment of nontermination the model could handle not only numbers and lists, but also unbounded lists and similar infinite structures. Functions were also data: a function could operate on other functions.

Robin Milner, to perform proofs in this logic, developed the theorem prover Stanford LCF [66]. Stanford LCF performed interesting proofs but required tedious repetition of commands. Milner decided to provide a meta language in which a theorem prover could be programmed; the result was Edinburgh LCF. Edinburgh LCF performed many proofs and was adopted as the basis of several other theorem provers. Many of these take radical new directions; Cambridge LCF closely follows Edinburgh LCF.

The following history is incomplete. It concentrates on proofs performed by my colleagues at Edinburgh and Cambridge. Much other work is in progress, and I apologize to everyone whose work is neglected.

1.2.1 Proofs involving denotational semantics

For her dissertation [20], A. J. Cohn verified three program schemes for recursion removal; a compiler from an **if-while** language into a **goto** language; and a compiler for an abstract language with recursive procedures.

The compiler proofs involve denotational definitions of direct and continuation semantics, and also operational definitions. The second compiler involves four semantic definitions, descending from an abstract to a machine orientation. For the source language Cohn gives a standard denotational semantics, a closure semantics, and a stack semantics. The target machine has an operational semantics. The equivalence between the highest and the lowest level is proved as three equivalences between adjacent levels. Due to the proof's size and complexity, Cohn performed it only on paper. She later proved in LCF that the standard and closure semantics are equivalent [22].

Sokołowski has studied a simple programming language, expressing the **while** command as an infinite nest of **if** commands. He proves the equivalence of a denotational semantics and a Hoare-style axiomatic semantics [95]. The proof that a Hoare rule is sound requires routine but tedious processing: expanding definitions and following chains of implications. Sokołowski gives a search tactic that verifies every rule except the **while** rule, which requires fixed point induction.

Mulmuley has developed theories and tactics for proving the existence of inclusive (recursively defined) predicates [72]. These occur in compiler proofs as the *simulation relation* $x \sim y$ between the denotational semantics of the source

language and the operational semantics of the target machine. Although domain theory handles recursively defined *functions*, recursive *predicates* may cause inconsistency. Justifying an inclusive predicate involves a morass of technical detail.

Inclusive predicates were a major concern in Cohn's compiler proof. Her simple language and machine, and intermediate semantic levels, allow simulation relations of the form $x \sim y$ if and only if $f(x) \equiv g(y)$, for functions f and g. Such relations need no justification but are rarely useful. Mulmuley's techniques pave the way for more ambitious compiler proofs.

Mulmuley has LCF theories of the universal domain **U** and the domain **V** of finitary projections of **U**. The correspondence between domains and elements of **V** gives quantification over domains in PPλ. Asked to prove the existence of a predicate, Mulmuley's system generates goals and submits them to appropriate tactics, based on rewriting and resolution. The system, which totals sixty pages of ML, handles several predicates in the literature. It verifies Stoy's predicate automatically [96]; in another example, only one goal out of sixteen requires human assistance. Mulmuley relies on a machine-verified predicate in his construction of fully abstract models of the lambda-calculus [73].

1.2.2 Verification of functional programs

Leszczyłowski verified the 'algebraic laws' of Backus's functional language FP by defining the FP operations [60]. He also proved [59] the termination of the function NORM, which puts conditional expressions into normal form by repeatedly replacing

$$\mathrm{IF}(\mathrm{IF}(u,v,w),y,z) \qquad \text{by} \qquad \mathrm{IF}(u,\mathrm{IF}(v,y,z),\mathrm{IF}(w,y,z)) \ .$$

That NORM always terminates is far from obvious. Leszczyłowski proved by structural induction on x that $\mathrm{NORM}(\mathrm{IF}(x,y,z))$ terminates for all x,y,z such that x is defined and $\mathrm{NORM}(y)$ and $\mathrm{NORM}(z)$ both terminate. The termination of $\mathrm{NORM}(x)$ for all defined x follows by induction.

Boyer and Moore devised this termination example [12]. Their theorem-prover only accepts recursive functions that it can prove total.[2] They prove that NORM is total by considering two numerical measures on conditional expressions. I have found a proof, with the same structure as the LCF one, in a logic of total functions. Termination is expressed via well-founded relations instead of partial functions. My paper includes a termination proof in Cambridge LCF [79].

Cohn and Milner proved the correctness of a parser for expressions composed of atoms, unary operators, and binary operators within parentheses [25,68]. The proof is by structural induction on expressions, followed by rewriting and simple

[2]The function f is *total* if $f(x)$ is defined for all values of x; otherwise f is *partial*. A nonterminating program gives rise to a partial function.

resolution. A more interesting parser uses information about operator precedence [21]. Both parsers are verified against a function for printing an expression. Correctness is stated thus: printing an expression then parsing it yields the original expression. Chisholm has derived a similar parsing algorithm with the help of the Gothenburg Type Theory system [18].

S. Holmström verified several sorting algorithms in Edinburgh LCF. His paper compares LCF with the Boyer/Moore and Affirm theorem provers [49].

I recently verified a function for unification [76]. As an example, the paper presents the proof of a theorem about substitution, describing the workings of Cambridge LCF in detail. Manna and Waldinger developed a theory of unification in order to study the synthesis of a unification algorithm [63]. The theory concerns lists, finite sets, expressions, substitutions, and unifiers, comprising two dozen propositions. Its translation into PPλ is straightforward: quantifiers must be restricted to defined values, and every function proved total. The LCF proof contains nearly three hundred stored theorems, mostly trivial termination proofs and basic properties of lists, sets, truth values, and numbers. Manna and Waldinger prove the final theorem by well-founded induction. PPλ does not provide this general induction principle; correctness is proved by structural induction on the natural numbers followed by structural induction on expressions.

The termination of the unification function relies on the correctness of the results of the nested recursive calls it makes, so termination and correctness must be proved simultaneously [79]. PPλ has the flexibility required for difficult termination proofs. The price is that termination must be explicitly considered at all times.

My first experiments with the unification proof used Edinburgh LCF. That experience influenced the design of Cambridge LCF.

1.2.3 Verification of digital circuits

M. J. C. Gordon's LCF_LSM is a theorem prover, built from Cambridge LCF, for verifying hardware [32]. His Logic for Sequential Machines includes terms that denote synchronous devices, with a binding mechanism to indicate their connections. The next state of a device depends on its current state and inputs. Gordon used LCF_LSM to verify a simple sixteen-bit computer [33]. Its eight instructions were implemented using an ALU, memory, various registers, a thirty-bit microcode controller, and ROM holding twenty-six microinstructions. Gordon defined bit vector operations such as field extraction and addition. Concise axioms specified each component and the circuitry implementing the computer (host machine), and also the intended behavior of the computer (target machine).

J. M. J. Herbert used LCF_LSM to verify an ECL chip designed for the Cambridge Fast Ring [39]. The chip, an interface between the Ring and the slower

logic, consisted of NOR gates, inverters, and flipflops, equivalent to about 360 gates. It was developed using the Cambridge Design Automation system for gate arrays, which Herbert modified to produce a file of LCF_LSM axioms describing the chip. Herbert verified the chip with respect to its functional specification. Error messages from LCF_LSM helped to locate flaws in the specification and wiring.

Verification does not guarantee perfection. When fabricated and put into service, the ECL chip displayed minor problems! The error was traced to a discrepancy between the specification of the chip and the way it was actually used. A system is never correct: at best, it is consistent with its specification.

Melham verified an associative memory unit in LCF_LSM, uncovering errors in the gating and microcode [7]. The unit was not designed as a verification example but for an application: to store message identifiers in a network interface device. It contains an AM2910 microprogram controller, memories, counters, busses, and drivers: a total of 37 TTL chips. Correctness of the initialized device was proved; correctness of the initialization sequence was not attempted. The proof, developed over thirty months of part time effort, comprises 4800 lines of ML and requires ten and a half hours of computer time.

As a successor to LCF_LSM, Gordon has implemented a *higher order logic* (HOL) on top of Cambridge LCF [34,35,37,38]. He uses it to represent hardware: each device is a predicate, time is an integer, and each wire is a function over time. A circuit is a conjunction of the predicates representing its component devices; the arguments of a predicate represent the wires connected to the device. To illustrate the power of higher order logic, Gordon specifies devices from a CMOS inverter to a sequential multiplier [36]. Herbert compares LCF_LSM with HOL in specifying the Cambridge Fast Ring chip [46]. Another example is the counter verification by Cohn and Gordon [24].

Recent work includes Cohn's verification of the Viper microprocessor [23]. This microprocessor, designed at the Royal Signals and Radar Establishment (RSRE) for safety-critical applications, may become the first real computer to be formally verified. In the notation of LCF_LSM, RSRE staff wrote Viper's functional specification and informally proved the correctness of its implementation in a state-transition machine. Cohn formalized this work in HOL, uncovering minor errors that fortunately were not reflected in the hardware. Typical of hardware verification are gigantic formulae and proofs: the Viper proof involves more than a million primitive inferences. The next stage of Cohn's work will be to verify the implementation at a level closer to that of circuits.

1.3 Further reading

Other books describing particular theorem provers include Boyer and Moore [12] and Constable et al. [26]. Bledsoe and Loveland's survey [9] is interesting but

hardly mentions LCF or other European work.

Several new theorem provers borrow ideas from LCF. Hanna and Daeche's *Veritas* [43] uses a purely functional meta language, Miranda.[3] My *Isabelle* uses Standard ML, represents a rule as a construction instead of a function, and emphasizes unification during inference [80].

You will not get far in this book unless you know Standard ML. The definition [45] is not easy to read; Harper [44] and Wikström [100] have written gentle introductions.

Milner gives a readable introduction to rules, tactics, and induction in Edinburgh LCF [68]. Gordon demonstrates how to implement a simple logic in ML [31]. The Nuprl book by Constable et al. gives a sometimes different interpretation of these ideas [26].

Of the proofs mentioned above, several are described in easily available papers. These include Cohn's proof of semantic equivalence [22], my proof of a unification algorithm [76], and Melham's hardware proof [7]. The earlier theorem provers are still of historical interest. Milner and Weyhrauch describe a simple compiler correctness proof in Stanford LCF [70]. Gordon and Herbert's article [39] includes a readable introduction to LCF_LSM.

[3] 'Miranda' is a trademark of Research Software Ltd.