Lessons learned from LCF: A Survey of Natural Deduction Proofs*

LAWRENCE C. PAULSON
Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG

The LCF project has produced a family of interactive, programmable theorem-provers, particularly intended for verifying computer hardware and software. The introduction sketches basic concepts: the metalanguage ML, the logic PPLAMBDA, backwards proof, rewriting, and theory construction. A historical section surveys some LCF proofs. Several proofs involve denotational semantics, notably for compiler correctness. Functional programs for parsing and unification have been verified. Digital circuits have been proved correct, and some subsequently fabricated.

There is an extensive bibliography of work related to LCF. The most dynamic issues at present are data types, subgoaling techniques, logics of computation, and the development of ML.

1. INTRODUCTION

'Natural deduction' is a technical term referring to formal logics where theorems are proved with respect to a changing set of assumptions. Many people question whether any formal proof can really be natural. With computer assistance, we can prove useful theorems in complete rigour. This paper describes the LCF family of interactive proof assistants. After an overview, it surveys the proofs that have been performed and the resulting insights.

There are many other interactive theorem provers. Boyer and Moore[1] and the AUTOMATH project[16] have conducted major studies.

2. LCF AND ITS METALANGUAGE

The main components of any LCF system are the metalanguage ML, a logic such as PPLAMBDA, subgoaling functions (tactics and tacticals), a simplifier for using rewrite rules, and commands for maintaining hierarchies of theories.

The Edinburgh LCF Manual[9] contains a full description. Though later versions of LCF vary in details, the fundamental concepts remain the same.

2.1. The language ML

An important aspect of LCF is that it can be programmed in its metalanguage ML. All commands are provided as ML functions. By writing ML code, you extend LCF with new commands and functions: this is how LCF grew over the years. Some data structures and pattern-matching primitives are implemented in Lisp, as is ML itself. The bulk of LCF is implemented as 5000 lines of ML.

ML is a functional programming language that provides typical data structures such as numbers, booleans, tuples and lists. The most important control structure is function call. Almost all ML code consists of function definitions and invocations, though ML provides assignment and iteration statements. ML has a few unusual features that were intended to support theorem proving, and have turned out to be generally useful.

Functions can be arguments or results of other functions. All of LCF's tacticals and rewriting operators are functions that operate on other functions.

Types are polymorphic, giving the security of rigid type systems such as Pascal's with almost all the flexibility of typeless languages. For example, you can define list operations that work on lists of integers, lists of lists of booleans, etc.

Exceptions (known as *failures*) can be raised and handled. A common way to combine theorem-proving functions is to try them one at a time until one terminates successfully.

2.2. The logic PPLAMBDA

Most LCF proofs are conducted in PPLAMBDA, a logic for domain theory.[9,27] Formulas are built up from the connectives $\forall$, $\wedge$, $\Rightarrow$, etc. Theorems are proved with respect to a changing set of assumptions, the *natural deduction* approach. If $A_1, \ldots, A_n, B$ are formulas, then a theorem of the form $[A_1; \ldots; A_n] \vdash B$ states that the conclusion B can be proved from the assumptions $A_1, \ldots, A_n$. The theorem $\vdash B$ has no assumptions. Inference rules are typical of natural deduction: *introduction* and *elimination* rules for each connective.

The logic is embedded in ML.[10] A logic consists of an ML data-type *form* of formulas, together with axioms and inference rules for proving theorems. Formulas of the logic are ML values, with functions for taking formulas apart and putting them together. Theorems are values of type *thm*. Axioms are pre-declared ML identifiers, while inference rules are functions mapping theorems to theorems. Theorems form an abstract data type, for a theorem can be constructed only via axioms and inference rules, not by arbitrary manipulation of its representation. ML's type-checking guarantees *soundness*: every theorem is true.

For example, consider the PPLAMBDA rules for implication. The introduction rule is called the 'discharge rule' because it discharges (cancels) the assumption A in the premiss. Any assumptions other than A are passed along to the conclusion.

$$\frac{[A] \vdash B}{\vdash A \Rightarrow B}$$

* An early version of this paper will appear in *Workshop on Formal Software Development: Combining Specification Methods*, edited by D. Bjørner, Springer (1985).

ML provides this rule as the function DISCH of type $form \rightarrow (thm \rightarrow thm)$. Its arguments are the formula A and the theorem $[A] \vdash B$.

The elimination rule is called Modus Ponens:

$$\frac{\vdash A \Rightarrow B \quad \vdash A}{\vdash B}$$

ML provides this rule as the function MP of type $thm \rightarrow (thm \rightarrow thm)$. Like many LCF inference rules, MP uses the failure mechanism to reject inappropriate premisses. It fails unless its arguments have the form $\vdash A \Rightarrow B$ and $\vdash A$.

The function *concl*, of type $thm \rightarrow form$, maps any theorem $\vdash A$ to its conclusion A. Now the ML function definition

let CUT *bth ath* = MP(DISCH(*concl ath*) *bth*) *ath*;;

implements the cut rule:

$$\frac{[A] \vdash B \quad \vdash A}{\vdash B}$$

Many derived inference rules can be implemented as ML functions.

PPLAMBDA differs from other logics in its polymorphic type system, which resembles ML's. A type is not a set but a domain, as in denotational semantics.[37] Types of higher-order functions and infinite streams can be constructed. Functions can be defined using lambda expressions or recursion equations. The rule of fixedpoint induction allows reasoning about the computation of a recursive function, and is the basis for other induction rules.

Every type includes the bottom element $\perp$. This represents the 'undefined' result of a nonterminating computation, to reason about partial functions and continuously operating processes. The formula $x \neq \perp$ means 'x is defined', and $\forall x.\, x \neq \perp \Rightarrow f(x) \neq \perp$ means 'f is a total function'. There is a partial order: $x \subseteq y$ means 'x is less defined than y'.

Versions of LCF exist for several logics other than PPLAMBDA.[11, 13, 32]

2.3. 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.[34] (Usually this notional tree is not actually stored in computer memory.) The goal of proving a formula A is written $\overset{?}{\vdash} A$. In Cambridge LCF,[28] the tactic CONJ_TAC reduces any goal of the form $\overset{?}{\vdash} A \wedge B$ to the two goals $\overset{?}{\vdash} A$ and $\overset{?}{\vdash} B$, and fails if its input is not a conjunction. The tactic DISCH_TAC reduces the goal $\overset{?}{\vdash} A \Rightarrow B$ to $[A] \overset{?}{\vdash} B$, the goal of proving B assuming A plus any previous assumptions.

The tactic function ACCEPT_TAC, applied to a theorem $\vdash A$, reduces a goal $\overset{?}{\vdash} A$ to the empty list of subgoals. It fails on other goals. If you reduce a goal to the empty list of subgoals, you have solved it and can turn your attention to some other goal in the tree.

These three tactics suffice to prove the goal $\overset{?}{\vdash} A \Rightarrow (B \Rightarrow A \wedge B)$. Calling DISCH_TAC gives the goal $[A] \overset{?}{\vdash} B \Rightarrow A \wedge B$. Calling DISCH_TAC again gives $[A; B] \overset{?}{\vdash} A \wedge B$. Calling CONJ_TAC gives the two goals $[A; B] \overset{?}{\vdash} A$ and $[A; B] \overset{?}{\vdash} B$. The first can be solved by the tactic ACCEPT_TAC(ASSUME 'A'), and the second by ACCEPT_TAC(ASSUME 'B').

Operators called *tacticals* combine tactics into larger ones. The basic ones are

THEN combines two tactics *sequentially*: the tactic (tac_1 THEN tac_2) applies tac_1 to the goal, gives the subgoals to tac_2, and returns all resulting subgoals.

ORELSE combines two *alternative* tactics: the tactic (tac_1 ORELSE tac_2) applies tac_1 to the goal. If tac_1 fails, then it applies tac_2.

REPEAT makes a tactic *repetitive*: the tactic (REPEAT *tac*) applies *tac* to the goal, its subgoals, etc. It returns a list of the goals on which *tac* fails.

Cambridge LCF provides additional tacticals for iteration down lists.[28] These work with tactic functions like ACCEPT_TAC. The tactical FIRST_ASSUM applies a tactic function to the first assumption; if the resulting tactic fails, it tries the second, third, ... assumption. So

FIRST_ASSUM ACCEPT_TAC

is a tactic. Applied to the goal $[A; B] \overset{?}{\vdash} B$, it acts like the tactic

(ACCEPT_TAC(ASSUME 'A')) ORELSE
(ACCEPT_TAC(ASSUME 'B')),

searching in the goal for the assumption B.

These tacticals can express the proof of $\overset{?}{\vdash} A \Rightarrow (B \Rightarrow A \wedge B)$ in many ways:

DISCH_TACTHENDISCH_TACTHENCONJ_TAC
THEN (FIRST_ASSUM ACCEPT_TAC)

or

REPEAT (DISCH_TAC ORELSE CONJ_TAC
ORELSE (FIRST_ASSUM ACCEPT_TAC))

or, using the standard tactic STRIP_TAC for breaking down goals,

REPEAT (STRIP_TAC ORELSE (FIRST_ASSUM
ACCEPT_TAC)).

The rewriting tactics described below can solve many similar tautologies in one step.

Tactics are implemented in ML on top of the abstract data type for theorems. A tactic that reduces the goal $\overset{?}{\vdash} A$ to the subgoals $\overset{?}{\vdash} B_1, \ldots, \overset{?}{\vdash} B_n$ also returns a function that takes the list of theorems $\vdash B_1, \ldots, \vdash B_n$ to the theorem $\vdash A$. Once every subgoal has been solved, these functions can be put together to produce the desired theorem as an

ML value. LCF's tacticals and interactive proof commands do this bookkeeping.

2.4. Rewriting

Every implementation of LCF includes a simplifier for applying rewrite rules and solving tautologies. Rewriting can simplify terms, formulas, or theorems. It is most commonly used, via a standard tactic, to simplify goals. Most proofs rely heavily on the simplifier.

A theorem of the form

$$\vdash t[x_1, \ldots, x_n] \equiv u[x_1, \ldots, x_n]$$

is a rewrite rule. The simplifier instantiates the variables $x_1, \ldots, x_n$ by pattern-matching: searching in the goal, it replaces every occurrence of $t[a_1, \ldots, a_n]$ by $u[a_1, \ldots, a_n]$. A theorem of the form

$$\vdash A[x_1, \ldots, x_n] \Rightarrow t[x_1, \ldots, x_n] \equiv u[x_1, \ldots, x_n]$$

is a *conditional* rewrite rule. The simplifier replaces $t[a_1, \ldots, a_n]$ by $u[a_1, \ldots, a_n]$ whenever it can prove $A[a_1, \ldots, a_n]$ by recursively invoking simplification. When simplifying a formula $A \Rightarrow B$, where A contains syntactically acceptable rewrite rules, the simplifier assumes these while simplifying B.

Cambridge LCF uses operators for combining primitive rewriting functions into powerful ones, as tacticals combine tactics. This makes it easy to implement a desired rewriting strategy.[29]

2.5. Building Theories

You can extend the logic with new constants, infix operators, types, and axioms. The ML function for declaring an axiom A returns the theorem $\vdash A$. LCF stores this information on a *theory file*, along with theorems proved within the theory. In a later session you can load in the theory file for proving additional theorems.

A theory may be an extension of other theories, called *parents*. This theory can itself become the parent of later ones, forming a directed acyclic graph. Suppose we have a theory *nat* of the natural numbers, and theory *list* of lists. A theory defining the length of a list would have parents *nat* and *list*. Long proofs involve months of interactive sessions and dozens of theories.

Sannella and Burstall have implemented new operators for building theories.[33] A theory can be abstracted, producing a theory whose details of construction are hidden. A parametrized theory is a function yielding theories; it can be applied to any theory satisfying stated logical properties.

3. A BRIEF HISTORY

3.1. Proofs in denotational semantics

For her dissertation,[4] A. J. Cohn verified

three schemes for recursion removal;

a compiler from an **if-while** language into a **goto** language;

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.[7]

A related problem is the equivalence between denotational and axiomatic definitions of semantics. Sokołowski has proved the soundness of Hoare rules for an **if-while** language, relative to a denotational definition.[36] He allows infinite programs, defining the **while** command as an infinite nest of **if** commands. (In PPLAMBDA, infinite data structures are easier to handle than finite ones!) Verification of Hoare rules is a largely routine but tedious process of expanding definitions and searching down chains of implications. Sokołowski's paper illustrates the search in action. The same tactic verifies every rule except **while**, which requires fixed-point induction. Verifying each Hoare rule demonstrates the soundness of the Hoare logic, by induction on proofs. He could not formalize induction on proofs in LCF; one attempt violated PPLAMBDA's requirement that all functions be continuous.

Mulmuley has implemented theories and tactics for proving the existence of inclusive (recursively defined) predicates.[25] He 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** allows quantification over domains to be expressed in PPLAMBDA. Asked to prove the existence of a predicate, Mulmuley's system generates goals and gives each one to an appropriate tactic. The tactics use rewriting and resolution. The system, which totals sixty pages of ML, handles several predicates in the literature. It verifies Stoy's predicate automatically;[37] 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.[26]

Inclusive predicates typically 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 makes it easy to introduce recursive *functions*, recursive predicates may cause inconsistency. Establishing them by hand is extremely technical and tedious. This was the major concern in Cohn's compiler proof. Her simple language and machine, and intermediate semantic levels, allow simple simulation relations. Each has the form $x \sim y$ if and only if $f(x) \equiv g(y)$, for particular functions f and g. Mulmuley's techniques pave the way for more ambitious compiler proofs.

3.2. Verification of functional programs

Leszczyłowski verified the algebraic laws of Backus's functional language FP.[18] He also proved[17] the termination of the function NORM, which puts conditional expressions into 'normal form' by repeatedly replacing

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

Leszczyłowski proved (by structural induction on x) that NORM(IF(x, y, z)) terminates for all x, y, z such that x is defined and NORM(y) and NORM(z) both terminate. The termination of NORM(x) for all defined x follows (by induction) from this peculiar lemma. Boyer and Moore devised this termination example [1]. Their theorem-prover only accepts recursive functions that it can prove total. They prove that NORM is total by considering two numerical measures on conditional expressions.

Cohn and Milner proved the correctness of a parser for expressions composed of atoms, unary operators, and binary operators within parentheses.[5] The proof is by structural induction on expressions, followed by rewriting and simple resolution. Cohn later verified a parser where operators have precedence and associativity.[6] Both parsers are verified with respect to a function for printing an expression as a list of terminal symbols. Correctness is stated as: printing an expression, then parsing it, gives back the same expression. The precedence parser proof unfortunately involves a large number of technical lemmas.

I recently verified a function for unification,[30] formalizing a theory due to Manna and Waldinger.[19] To begin I provided Edinburgh LCF with a faster ML compiler, the connectives $\vee$, $\exists$, and $\leftrightarrow$, a new simplifier, and new tactics and tacticals. This produced Cambridge LCF.

Manna and Waldinger's theory involves lists, finite sets, expressions, substitutions, and unifiers. They state about two dozen propositions. The translation into PPLAMBDA 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 ones such as termination proofs and basic properties of lists, sets, truth values, and numbers. Manna and Waldinger prove the final theorem by well-founded induction. PPLAMBDA does not provide this general induction principle, but the induction in the correctness proof can be achieved by nested structural induction on the natural numbers and 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. PPLAMBDA has the flexibility required for difficult termination proofs. The price is that termination must be explicitly considered at all times.

3.3. Verification of digital circuits

M. J. C. Gordon has been verifying hardware. His Logic for Sequential Machines (LSM) extends PPLAMBDA with bit strings and communication lines. A term can represent a device with inputs and outputs, with a binding mechanism for indicating how devices are wired together. Only synchronous devices can be specified: the next state depends on the current state and the values on the input lines. The domain theory has been removed. The prover for this logic, built on top of Cambridge LCF, is called LCF_LSM.[11]

Gordon used LCF_LSM to verify a simple sixteen-bit computer.[12] 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 behaviour of the computer (target machine). Gordon used forwards proof rather than tactics. Host and target behaviour descriptions were expanded out, producing enormous formulas that required hours of processor time to simplify.

J. M. J. Herbert used LCF_LSM to verify an ECL chip designed for the Cambridge Fast Ring.[14] The chip, an interface between the ring and the slower logic, was developed using the Cambridge Design Automation system for gate arrays. Herbert modified the design system to produce a file containing LCF_LSM axioms describing the implementation of the chip. He verified an implementation consisting of NOR gates, inverters, and flipflops (equivalent to about 360 gates) with respect to its functional specification. Error messages from LCF_LSM helped to locate flaws in both the specification and wiring. The chip was fabricated and found to work correctly.

Moxon verified a number of adders, including a carry-lookahead adder, in Cambridge LCF. Melham has been using LCF_LSM to verify an associative memory unit, uncovering errors in the gating and microcode.[22] The unit is intended for a real application. It contains a microcode controller, memories, counters, buses and drivers. The verification may never be completed: its latter stages are putting tremendous demands on LCF_LSM.

As a successor to LCF_LSM, Gordon has implemented a *higher-order logic* (HOL) on top of Cambridge LCF.[13] 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. HOL also supports proofs in classical mathematics. It allows quantification over predicate variables, directly expressing inference rules such as induction.

4. THE DEVELOPMENT OF IDEAS

4.1. Data types

Consider recursive data structures such as lists and trees. Deriving structural induction in PPLAMBDA is far from trivial. In her compiler proofs, Cohn spent months developing theories of syntax trees for the source languages.[4] Then Milner wrote an ML program to derive structural induction automatically. It handled recursive data types defined as sums of products. Later proofs used his program to generate theories of syntax trees.[5, 6, 36]

Cohn and Milner[5] still had the problem of excluding infinite data structures, which exist unless the constructor functions are *strict*. (Example: for lists, CONS is strict if CONS $\perp l \equiv$ CONS $x \perp \equiv \perp$.) I rewrote Milner's program to let the user specify whether the constructors should be strict or not. I also studied mutually recursive types and types where the constructors satisfy equational constraints.[31]

Monahan's thesis is an extensive and detailed study of LCF types using category theory.[24] Monahan defines concatenation for lists and shows it to form a free

monoid, taking the empty list for the identity element. He defines multisets as equivalence classes of lists, using a computable function to test whether two lists represent the same multiset (contain the same elements, regardless of order). For most theorems he presents an informal proof and one or more LCF proofs, discussing each step and considering alternatives. He describes resolution tactics, unification functions, and other tools developed within Edinburgh LCF. He has extended Milner's program to accept a much larger class of recursive type definitions. The bibliography lists a vast range of works about theorem proving and data types.

Past LCF proofs have begun with a long phase of laying the groundwork: theories of lists have been derived again and again. Manna and Waldinger have put together fundamental theories of data structures: numbers, strings, trees, lists, sets, bags, and tuples.[20] These could be formalised into a library of LCF theories, to be included when needed in any proof.

4.2. Tactics

The Edinburgh LCF manual lists only a handful of tactics, not even the basic CONJ_TAC and DISCH_TAC.[9] Early LCF users, especially Cohn, implemented additional tactics to introduce or eliminate connectives, make use of assumptions, perform special substitutions, and resolve implications against other theorems. Cambridge LCF provides many of these.

In an abstract study of tactics, Schmidt discusses the interplay between forwards and backwards reasoning in natural deduction proofs.[34] Sokołowski's new set of tactics implements these ideas, providing systematic rewriting and decomposition of the goal and assumptions, and detection that the goal has been reduced to tautologies.[35] Sokołowski's major innovation is allowing goals to contain pattern variables that can be unified against assumptions. His proofs are remarkably clean: the new tactics should be useful for most LCF applications.[36]

More theorem-proving tasks should be automated. The resolution tactics should use unification rather than one-way matching. The Knuth–Bendix completion precedure could help to eliminate ordering conflicts among rewrite rules.[15] The LCF philosophy insists that heuristics be easy to understand and independent of other heuristics. Otherwise only experts could hope to predict the course of a proof.

4.3. Logics

Martin-Löf's *Intuitionistic Type Theory*[21] is a formal logic for constructive mathematics. It can be seen as a programming and specification language for total recursive functions. Its types include the familiar data structures, and are rich enough to express logical propositions. A proposition is proved by constructing a computable function. You can specify the type of all sorting functions, and satisfy the specification by exhibiting a function of that type. Petersson has embedded Type Theory in ML.[32] Constable and Bates have built a substantial theorem prover, implementing a related type theory on top of ML.[8]

Variants of LCF differ primarily in the logic used for proofs: PPLAMBDA, LSM, higher-order logic, Intuitionistic Type Theory, etc. It requires great effort to embed a new logic into LCF. I am implementing a data structure for terms built by abstraction and application, with a library of basic functions such as occurrence testing, substitution, pattern-matching and unification. This will be used to re-implement Type Theory, and could support other logics.

4.4. Standard ML

Most of these developments have been implemented in ML. It is remarkable that a language developed for proving theorems should prove widely useful; people in several universities and companies want to use ML to implement specification tools, compilers, etc. Milner is coordinating improvements to the language and its implementations. He has organized a series of meetings of the ML community, and distilled the many suggestions into the new Standard ML.[23]

The main extensions are recursive data structures with pattern-directed function calls as in HOPE,[2] exceptions of arbitrary type, and reference types. Cardelli[3] and D. C. J. Matthews have written compilers giving faster execution than Pascal or compiled Lisp. The University of Edinburgh have developed a portable compiler written in its own language. A small kernel written in C provides garbage collection and an interpreter for ML's abstract machine. Standard ML is already proving itself in early experiments.

Acknowledgements

R. Milner conceived most of the ideas behind ML and LCF. He led the development of Edinburgh LCF; other implementors included M. J. C. Gordon, L. Morris, M. Newey and C. P. Wadsworth. Cambridge LCF was implemented in collaboration with INRIA (Rocquencourt, France), particularly G. Huet and G. Cousineau. Most LCF research has been funded by the SERC. J. M. J. Herbert and N. Shankar offered several comments on this paper.

REFERENCES

1. R. Boyer and J. Moore, *A Computational Logic*, Academic Press, (1979).
2. R. M. Burstall, D. B. MacQueen, D. T. Sannella, HOPE: an experimental applicative language, Report CSR-62-80, Department of Computer Science, University of Edinburgh (1981).
3. L. Cardelli, ML under Unix, Report (in preparation), ATT Bell Labs, Murray Hill, NJ (1984).
4. A. J. Cohn, Machine Assisted Proofs of Recursion Implementation, report CST-6-79, *PhD Thesis*, University of Edinburgh (1980).
5. A. J. Cohn and R. Milner, On using Edinburgh LCF to prove the correctness of a parsing algorithm, report CSR-113-82, Department of Computer Science, University of Edinburgh (1982).
6. A. J. Cohn, The correctness of a precedence parsing

algorithm in LCF, report 21, Computer Laboratory, University of Cambridge (1982).
7. A. J. Cohn, The equivalence of two semantic definitions: a case study in LCF, *SIAM Journal of Computing* **12**, 267–285 (1983).
8. R. L. Constable and J. L. Bates, The nearly ultimate PEARL, Report TR 83–551, Cornell University (1984).
9. M. J. C. Gordon, R. Milner and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Springer LNCS 78 (1979).
10. M. J. C. Gordon, Representing a logic in the LCF metalanguage. In *Tools and Notions for Program Construction*, edited D. Néel, Cambridge University Press, pp. 163–185 (1982).
11. M. J. C. Gordon, LCF_LSM: A system for specifying and verifying hardware, Report 41, Computer Laboratory, University of Cambridge (1983).
12. M. J. C. Gordon, Proving a computer correct with the LCF_LSM hardware verification system, Report 42, Computer Lab., University of Cambridge (1983).
13. M. J. C. Gordon, HOL: A machine oriented formulation of Higher Order Logic, Report 68, Computer Laboratory, University of Cambridge (1985).
14. J. M. J. Herbert, The application of formal specification and verification to a hardware design, Proceedings of IFIP 7th International Symposium on Computer Hardware Description Languages and their Applications, 1985 (to appear).
15. G. Huet and D. Oppen, Equations and rewrite rules: a survey. In: *Formal Language Theory: Perspectives and Open Problems*, edited R. Book, Academic Press pp. 349–406.
16. L. S. van Benthem Jutting, Checking Landau's 'Grundlagen' in the AUTOMATH system, *PhD Thesis*, Technische Hogeschool, Eindhoven (1977).
17. J. Leszczyłowski, An experiment with 'Edinburgh LCF', In *Fifth Conference on Automated Deduction*, edited W. Bibel and R. Kowalski pp. 170–181 Springer (1980).
18. J. Leszczyłowski, Theory of FP systems in Edinburgh LCF, Report CSR-61-80, Dept. of Computer Science, University of Edinburgh (1980).
19. Z. Manna and R. Waldinger, Deductive synthesis of the unification algorithm, *Science of Computer Programming* **1**, 5–48 (1981).
20. Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*: Volume I: *Deductive Reasoning* (Addison-Wesley, 1985).
21. P. Martin-Löf, Constructive mathematics and computer programming. In *Logic, Methodology, and Science VI*, edited by L. J. Cohen, J. Los, H. Pfeiffer and K.-P. Podewski. pp. 153–175, 1982. Amsterdam: North-Holland.
22. T. Melham, R. Schediwy and G. Birtwistle, An experience with using LCF_LSM to verify the flooding sink memory design, Report (in preparation), Dept. of Computer Science, University of Calgary (1985).
23. R. Milner, The Standard ML core language, Report CSR-168-84, Dept. of Computer Science, University of Edinburgh (1984).
24. B. Q. Monahan, Data Type Proofs using Edinburgh LCF, *PhD Thesis*, University of Edinburgh, 1985.
25. K. Mulmuley, The mechanization of existence proofs of recursive predicates. In *Seventh Conference on Automated Deduction*, edited, R. E. Shostak pp. 460–475. Springer LNCS 170 (1984).
26. K. Mulmuley, Full Abstraction and Semantic Equivalence, *PhD Thesis* (in preparation), Carnegie-Mellon University (1985).
27. L. Paulson, The revised logic PPLAMBDA: a reference manual, Report 36, Computer Laboratory, University of Cambridge (1983).
28. L. Paulson, Tactics and tacticals in Cambridge LCF, Report 39, Computer Laboratory, University of Cambridge (1983).
29. L. Paulson, A higher-order implementation of rewriting, *Science of Computer Programming* **3**, 119–149 (1983).
30. L. Paulson, Verifying the unification algorithm in LCF, *Science of Computer Programming* **5**, pp. 143–170, 1985.
31. L. Paulson, Deriving structural induction in LCF. In *International Symposium on Semantics of Data Types*, edited G. Kahn, D. B. MacQueen, G. Plotkin, pp. 197–214. Springer LNCS 173 (1984).
32. K. Petersson, A programming system for type theory, Report 21, Department of Computer Sciences, Chalmers University, Göteborg, Sweden (1982).
33. D. Sannella, R. M. Burstall, Structured theories in LCF, *Eighth Colloquium on Trees in Algebra and Programming*, pp. 377–391. Springer (1983).
34. D. Schmidt, A programming notation for tactical reasoning. In *Seventh Conference on Automated Deduction*, edited R. E. Shostak, Springer LNCS 170, pp. 445–459. (1984).
35. S. Sokołowski, A note on tactics in LCF, Report CSR-140-83, Department of Computer Science, University of Edinburgh (1983).
36. S. Sokołowski, An LCF proof of the soundness of Hoare's logic, Report CSR-146–83, Department of Computer Science, University of Edinburgh (1983). (To appear in *ACM Transactions on Programming Languages and Systems*.)
37. J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, M.I.T. Press (1977).