Technical Report

UCAM-CL-TR-54
ISSN 1476-2986

Number 54

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Lessons learned from LCF

Lawrence Paulson

August 1984

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Lessons Learned from LCF

Lawrence Paulson
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG
England

August 1984

Abstract

The history and future prospects of LCF are discussed. The introduction sketches basic concepts such as the language ML, the logic PPLAMBDA, and backwards proof. The history discusses LCF proofs about denotational semantics, functional programs, and digital circuits, and describes the evolution of ideas about structural induction, tactics, logics of computation, and the use of ML. The bibliography contains thirty-five references.

CONTENTS

Lessons Learned from LCF [1]

1. Introduction

The acronym LCF stands for Scott's Logic for Computable Functions, but has come to denote a family of interactive proof assistants for this and other logics. This paper surveys several years' experience with LCF, examining its abilities and limitations. After an overview of LCF and ML, the paper describes the proofs that have been performed and the ideas which have grown and developed, and discusses future directions for the LCF approach. Below, "LCF" without qualification refers to the family; sometimes a particular version of LCF is mentioned.

2. LCF and its Metalanguage

This section outlines the principles of LCF. The main components of any LCF system are the metalanguage ML, a logic such as PPLAMBDA, subgoaling primitives (tactics and tacticals), a simplifier for using rewrite rules, and primitives 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 text, you can incrementally 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, but the bulk of the theorem-proving software is written in 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 text 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:

[1] To appear in D. Bjørner, editor, *Workshop on Formal Software Development: Combining Specification Methods*, Springer LNCS, 1984.

- Functions are first-class objects. They 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. ML gives each expression the most general type possible. You can easily define functions on lists, and use them on lists of integers, lists of booleans, etc.
- Exceptions (known as *failures*) can be raised and handled. A common way of combining LCF proof strategies is to try them one at a time until one terminates successfully. It is impractical to determine in advance whether a strategy is appropriate in a particular situation; instead, it proceeds as far as possible, perhaps signaling failure after a long computation.

2.2. The logic PPLAMBDA

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

I have mentioned that all the commands of LCF are available as ML functions. More importantly, the logic is available via ML. A logic consists of a 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 predeclared 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. Type-checking guarantees soundness, that 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.)

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

ML provides this rule as the function DISCH of type *form* $\rightarrow$ *thm* $\rightarrow$ *thm*.

The elimination rule is called Modus Ponens:

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

ML provides this rule as the function MP of type *thm* → *thm* → *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* → *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:

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

Many derived inference rules can be implemented as ML functions.

PPLAMBDA differs from other logics in its polymorphic type system, which resembles ML's. Furthermore, a type stands for a domain (complete partial order) rather than a set. You can construct types for higher-order functions or infinite streams. Every type includes the bottom element $\perp$; you can write formulas involving the partial order, such as $x \sqsubseteq y$. Logical terms include lambda expressions. There is a fixedpoint operator and the rule of fixedpoint induction. Stoy describes the theory of domains and its use in denotational semantics [35].

PPLAMBDA is best suited for difficult termination proofs and studies of denotational semantics and unbounded computation. A different problem area may demand a different logic: an LCF system for hardware verification is mentioned below. Versions of LCF have been written for several logics, but share most of the same ideas (and code). My report describes the version of PPLAMBDA used in Cambridge LCF [25]. Gordon introduces ML and illustrates how to embed a logic in it [10].

2.3. Tactics and tacticals

In formal logic, and in the lowest level of LCF, inference consists of producing theorems by applying rules to other theorems. This process, called *forwards* proof, suffices to enumerate all theorems, but you will have to wait a long time before it produces a theorem that interests you.

Most theorem-provers, whether human or mechanical, work in the *backwards* direction. You start with a *goal*, the theorem to be proved, and proceed by reducing goals to simpler subgoals. Even resolution theorem-provers use backwards

proof, though each resolution step is a forwards inference. Resolution consists of negating the goal and adding it as an axiom, then deducing new theorems until a contradiction occurs. The entire deduction is determined by the goal.

Functions called *tactics* reduce goals to subgoals. A complete tactical proof consists of a tree of goals, whose leaves are known theorems. In Cambridge LCF, the tactic CONJ_TAC reduces any goal of the form $A \wedge B$ to the two goals A and B, and fails if its input is not a conjuction. The tactic DISCH_TAC reduces the goal $A \Rightarrow B$ to $[A]B$, 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 A to the empty list of subgoals (and fails on other goals). If you reduce a goal to zero 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 $A \Rightarrow (B \Rightarrow A \wedge B)$. Calling DISCH_TAC gives the goal $[A]B \Rightarrow A \wedge B$. Calling DISCH_TAC again gives $[A; B]A \wedge B$. Calling CONJ_TAC gives the two goals $[A; B]A$ and $[A; B]B$. The first can be solved by the tactic ACCEPT_TAC(ASSUME "A"), and the second by ACCEPT_TAC(ASSUME "B").

LCF provides operators called *tacticals* for combining 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, for use 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 which, when applied to the goal $[A; B]B$, acts much like the tactic

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

searching in the goal for the assumption B.

Using these tacticals, we can express the proof of $A \Rightarrow (B \Rightarrow A \wedge B)$ in many ways:

```
DISCH_TAC THEN DISCH_TAC THEN CONJ_TAC THEN
(FIRST_ASSUM ACCEPT_TAC)
```

or

```
REPEAT (DISCH_TAC ORELSE CONJ_TAC ORELSE
    (FIRST_ASSUM ACCEPT_TAC))
```

or, using a standard tactic for breaking down goals,

```
REPEAT (STRIP_TAC ORELSE (FIRST_ASSUM ACCEPT_TAC)) .
```

The rewriting tactics described in the following section can solve many similar tautologies in one invocation.

Tactics are implemented in ML on top of the abstract data type for theorems. Along with the list of subgoals, each tactic returns a *validation* function for constructing the forwards proof. A tactic that reduces the goal A to the subgoals $B_1, \cdots, B_n$ should return a validation which, given the list of theorems $\vdash B_1, \cdots, \vdash B_n$, yields the theorem $\vdash A$.

Once every subgoal has been solved, these validation functions can be put together to produce the desired theorem as an ML value. A compound tactic such as tac_1 THEN tac_2 not only keeps track of the subgoals produced by tac_1 and tac_2, but also combines the validations to make a proper validation for the whole. The implementation of THEN requires tricky list manipulation. Cambridge LCF provides simple commands for interactively applying tactics to goals, and automatically keeps track of the validations.

My report describes the tactics of Cambridge LCF [26]. Schmidt gives a lucid and thorough discussion of the interplay between forwards and backwards reasoning in natural deduction proofs [32].

2.4. Rewriting

Every implementation of LCF includes a simplifier for applying rewrite rules and solving tautologies. A theorem of the form

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

is a rewrite rule that allows the simplifier to replace any term $t[a_1, \cdots, a_n]$ by $u[a_1, \cdots, a_n]$. A theorem of the form

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

is a *conditional* rewrite rule. The simplifier replaces $t[a_1, \cdots, a_n]$ by $u[a_1, \cdots, a_n]$ whenever it can prove $A[a_1, \cdots, a_n]$ by (recursive) simplification. When simplifying a formula $A \Rightarrow B$, where A contains syntactically acceptable rewrite rules, the simplifier assumes these while simplifying B. Rewriting can be used to simplify terms, formulas, or theorems. It is most commonly used, via a standard tactic, to simplify goals. Most proofs rely heavily on the simplifier.

The above discussion applies to the simplifiers in both Edinburgh and Cambridge LCF, despite their many differences. The Cambridge LCF simplifier uses operators for combining primitive rewriting functions into powerful ones, just as tacticals combine tactics. This makes it easy to build simplifiers that use a particular rewriting strategy [27].

2.5. Building Theories

Any serious proof requires extensions to the standard theory provided by the logic. LCF provides commands for declaring new constants, infix operators, and types. To endow these new objects with properties, you can introduce axioms. When introducing an axiom A, the command (an ML function) returns the theorem $\vdash A$ as its value.

A long proof may require weeks or months of interactive sessions. LCF provides a simple database for recording theories. A *theory file* contains all the constants, operators, types, and axioms of a theory. You can also store away theorems on the file. You can retrieve the theorems in a later session, after asking LCF to restore the logical context (constants, axioms, etc.) in which the theorem was proved.

You can define a theory to be an extension of several other theories, called *parents*. This theory can become the parent of later ones, forming a hierarchy. A theory hierarchy should be a sensible decomposition of the concepts involved in the proof. For example, you might create a theory *nat* of the natural numbers, and a theory *list* of lists. A theory defining the length of a list would have *nat* and *list* and parents.

The theory primitives are simple, but powerful enough to handle substantial proofs. My unification proof, discussed below, involves a hierarchy of several dozen theories.

3. A Brief History

If we include Stanford LCF, the LCF project is over ten years old; only a few

of the many proofs can be mentioned. History really began with Edinburgh LCF, the first version to include ML. Let us see what people have accomplished using all this machinery.

3.1. Proofs in denotational semantics

For her dissertation, 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 [4].

Cohn's proofs rely on PPLAMBDA's domain theory. The proof that two functions are equal, $f \equiv g$, often uses fixedpoint induction to prove that $f \sqsubseteq g$ and $g \sqsubseteq f$. The compiler proofs involve denotational definitions of direct and continuation semantics, and also operational definitions. Due to the complexity of comparing a high-level semantic definition with a low-level one, the second compiler was only verified on paper. Cohn later formalized part of this in LCF [7].

A related problem is the equivalence between denotational and axiomatic definitions of semantics. Sokołowski proved the soundness of Hoare rules for an **if-while** language, relative to a denotational definition [34]. He took the unusual step of allowing infinite programs in the language, and defined the **while** command as an infinite nesting of **if** commands. (In the logic PPLAMBDA, infinite data structures are easier to handle than finite ones!) He verified each Hoare rule, which, by induction on Hoare proofs, demonstrates the soundness of the logic. He could not formalize induction on Hoare 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 predicates [23]. These proofs, which are extremely tedious to do by hand, are important in compiler verification. 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 [35]; 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 [24]. The system should allow the verification of more realistic compilers than Cohn's.

3.2. Verification of functional programs

Leszczyłowski verified the algebraic laws of Backus's functional language FP [17]. He also proved the termination of a normalization function for conditional expressions [16]. Because the logic PPLAMBDA allows reasoning about non-terminating functions, he could prove lemmas about the function before proving termination. The termination proof may be contrasted with Boyer and Moore's [1]. Their theorem-prover only allows total functions, requiring that every recursive call decrease some numeric measure of the argument. They introduce the normalization function along with the lexicographic combination of two measures on conditional expressions.

Cohn and Milner proved the correctness of a simple parser for expressions composed of atoms, unary operators, and binary operators within parentheses [5]. They give a readable account of the LCF approach and structural induction. Cohn later proved a much more difficult theorem for a parser where operators have precedence and associativity [6]. Both proofs introduce 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.

I recently verified a function for unification, formalizing a theory due to Manna and Waldinger [18]. This required developing theories of lists, finite sets, expressions, substitutions, and unifiers [28]. Before proceeding deeply into the unification project, I decided to remedy some defects that Edinburgh LCF users had encountered, by providing a faster ML compiler, the connectives $\vee$, $\exists$, and $\leftrightarrow$, a new simplifier, and new tactics and tacticals. The resulting prover was called Cambridge LCF.

The LCF proof is less elegant than Manna and Waldinger's. They prove the final theorem by well-founded induction, which is not available in LCF. I simulate their induction by nested structural induction on the natural numbers and on expressions. PPLAMBDA is inconvenient for reasoning about functions that terminate; this cannot be left implicit, but requires frequent appeals to termination theorems. But PPLAMBDA is vital for proving that unification terminates. Its termination relies on the correctness of the results of the nested recursive calls it makes, so termination and correctness must be proved simultaneously. The proof requires extensive reasoning about the unification function before its termination

is known.

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 input lines. The domain theory has been removed, for it is of little relevance to hardware. The prover for this logic, built on top of Cambridge LCF, is called LCF_LSM [11].

As a case study in LCF_LSM, Gordon has verified a simple sixteen-bit computer [12]. Its eight instructions are implemented using an ALU, memory, various registers, and thirty-bit microcode controller. These components are specified, and the configuration proved equivalent to a specification of the machine's intended behavior. Unusually, LCF_LSM proofs use mainly forwards proof rather than tactics.

John Herbert used LCF_LSM to verify a chip used in the Fast Cambridge Ring [14]. The chip, an eight-bit modulator-demodulator, 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 the implementation with respect to its functional specification, another set of LCF_LSM axioms.

Melham used LCF_LSM to verify most of an associative memory unit, uncovering a flaw in the original design [21]. The device includes memories, counters, busses, and drivers. John Moxon verified a number of adders, including a carry-lookahead adder, in Cambridge LCF.

4. The Development of Ideas

The most useful outcome of an LCF proof is not the theorem, but a better understanding of proof techniques. Interaction with LCF makes it easy to experiment with different approaches.

4.1. Structural induction

Consider the theory of trees and other recursive data structures. The deriva-

tion of 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]. Problems she encountered persuaded the implementors to change the treatment of union types, though she did not benefit from having the system shift under her feet.

Some time later, Milner wrote an ML program for structural induction. When Cohn and Milner verified a simple parser for expressions, Milner's program generated the theory of syntax trees [5]. Cohn and Milner still had the problem of handling infinite data structures, which often creep into LCF types. My unification work required types with strict constructors, which construct only finite objects. I also studied mutually recursive types and types where the constructors satisfy equational constraints [29].

4.2. Tactics

Our understanding of tactics has also grown. The Edinburgh LCF manual lists only a handful of tactics, not even the basic CONJ_TAC and DISCH_TAC [9]. The early LCF papers, especially by Cohn, describe additional tactics to introduce or eliminate connectives, make use of assumptions, perform special substitutions, and resolve implications against other theorems. These ideas are implemented in Cambridge LCF.

Schmidt has studied tactics from an abstract point of view [32]. 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 [33]. Regular naming conventions emphasize the contrast between introduction and elimination rules. 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 [34].

4.3. Logics

The proofs mentioned in the previous section were conducted in PPLAMBDA or in Gordon's Logic for Sequential Machines [11]. Gordon has also implemented a *Higher-Order Logic* on top of Cambridge LCF [13]. He plans to use it in hardware proofs, as a successor to LCF_LSM. It should be applicable to proofs in classical mathematics. Since the logic allows quantified predicate variables, it can directly express inference rules such as induction.

I have been studying Martin-Löf's *Intuitionistic Type Theory* [20], a logic concerned with the computation of total recursive functions. Its type system is rich enough to express logical propositions: you prove a proposition by constructing a functional program that can be executed. 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 [30]. Constable and Bates have built a substantial system, implementing a related logic on top of ML [8].

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 useful in other areas, but people want to use ML for symbolic processing, artificial intelligence, and even systems programming. Milner's Standard ML effort is a response to the demand for improvements to the language and its implementations. The new Standard takes account of experience with ML and related languages [22]. (This proposal is not final; input/output and modules are still being considered.) The main improvements are pattern-directed function calls as in HOPE [2], a more flexible exception mechanism, and reference types. Implementations are under way, some of which are more efficient than Pascal or compiled Lisp [3].

5. Future Directions

All indications are that LCF research will continue to progress in a variety of approaches. Sannella and Burstall have implemented new operators for building theories [31]. A theory can be abstracted, producing a theory whose details of construction are hidden. A parametrized theory is be a function yielding theories; it can be applied to any theory satisfying stated logical properties.

LCF's theory package needs improvement. A theory contains declarations of constants, types, axioms, and ancestor theories. LCF stores this information, plus any theorems that have been proved, on a theory file. To work in a theory you must load its theory file into LCF. You are then stuck in the theory for the rest of the session. The current theory is part of LCF's state, when perhaps theories should be ML objects. LCF also needs a way of saving tactics and other ML objects related to a theory.

More theorem-proving tasks should be automated: current LCF practice relies excessively on rewriting. The resolution tactics are weak, but Sokołowski's

unification techniques may allow someone to implement proper resolution. The Knuth-Bendix completion procedure would be helpful in detecting conflicts between different rewrite rules [15]. The LCF philosophy is cool towards artificial intelligence ideas because they might make LCF hard to control, but heuristics related to those of Boyer and Moore could be valuable [1].

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 [19]. These could be formalized into a library of LCF theories, to be included when needed in any proof.

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. Type Theory is defined on top of a theory of expressions: an abstract syntax for typed abstraction and application. An implementation of these expressions could provide syntactic functions such as occurrence testing, substitution, pattern matching, and unification, which could be used to implement nearly any logic. This would be a step towards a reconfigurable proof assistant.

Acknowledgements

R. Milner conceived most of the ideas behind LCF. M. J. C. Gordon has been involved for many years, and worked closely with me at Cambridge. Gérard Huet and Guy Cousineau contributed a great deal to the Lisp implementation of ML. N. Shankar offered several comments on this paper. I am also grateful to the organizers of this most productive and enjoyable workshop.

6. Bibliography

[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, Dept. of Computer Science, University of Edinburgh, 1981.

[3] L. Cardelli, ML under Unix, Report (in preparation), 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, Dept. of Computer Science, University of Edinburgh, 1982.

[6] A. J. Cohn, The correctness of a precedence parsing algorithm in LCF, Report 21, Computer Lab., University of Cambridge, 1982.

[7] A. J. Cohn, The equivalence of two semantic definitions: a case study in LCF, *SIAM Journal of Computing* **12**, pages 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. Wadsworth, *Edinburgh* LCF, Springer LNCS 78, 1979.

[10] M. J. C. Gordon, Representing a logic in the LCF metalanguage, in: D. Néel, editor, *Tools and Notions for Program Construction*, Cambridge University Press, pages 163–185, 1982.

[11] M. J. C. Gordon, LCF_LSM: A system for specifying and verifying hardware, Report 41, Computer Lab., 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, Higher Order Logic: description of the HOL proof generating system, Report (in preparation), Computer Lab., University of Cambridge, 1984.

[14] J. M. J. Herbert, Verification by formal proof of the Cambridge Fast Ring modulator and demodulator chip, Report (in preparation), Computer Lab., University of Cambridge, 1984.

[15] G. Huet and D. Oppen, Equations and rewrite rules: a survey, in: R. Book, Editor, *Formal Language Theory: Perspectives and Open Problems*, Academic Press, pages 349–406, 1980.

[16] J. Leszczyłowski, An experiment with 'Edinburgh LCF,' in: W. Bibel and R. Kowalski, editors, *Fifth Conference on Automated Deduction*, Springer LNCS 87, pages 170–181, 1980.

[17] J. Leszczyłowski, Theory of FP systems in Edinburgh LCF, Report CSR-61-80, Dept. of Computer Science, University of Edinburgh, 1980.

[18] Z. Manna, R. Waldinger, Deductive synthesis of the unification algorithm, *Science of Computer Programming* 1, pages 5–48, 1981.

[19] Z. Manna, R. Waldinger, *The Logical Basis for Computer Programming: Volume I: Informal Reasoning*, In preparation, 1984.

[20] P. Martin-Löf, Constructive mathematics and computer programming, in: L. J. Cohen, J. Los, H. Pfeiffer and K.-P. Podewski (editors), *Logic, Methodology, and Science VI*, North Holland, pages 153–175, 1979.

[21] T. Melham, Proof of correctness of the flooding sink memory chip, Report (in preparation), Dept. of Computer Science, University of Calgary, 1984.

[22] R. Milner, A proposal for Standard ML, Report CSR-157-83, Dept. of Computer Science, University of Edinburgh, 1983.

[23] K. Mulmuley, The mechanization of existence proofs of recursive predicates, in: R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, Springer LNCS 170, pages 460–475, 1984.

[24] K. Mulmuley, *Full Abstraction and Semantic Equivalence*, PhD thesis (in preparation), Carnegie-Mellon University, 1984.

[25] L. Paulson, The revised logic PPLAMBDA: a reference manual, Report 36, Computer Lab., University of Cambridge, 1983.

[26] L. Paulson, Tactics and tacticals in Cambridge LCF, Report 39, Computer Lab., University of Cambridge, 1983.

[27] L. Paulson, A higher-order implementation of rewriting, *Science of Computer Programming* 3, pages 119–149, 1983.

[28] L. Paulson, Verifying the unification algorithm in LCF, Report 50, Computer Lab., University of Cambridge, 1984. (To appear in *Science of Computer Programming.*)

[29] L. Paulson, Deriving structural induction in LCF, in: G. Kahn, D. B. MacQueen, G. Plotkin, editors, *International Symposium on Semantics of Data Types*, Springer LNCS 173, pages 197–214, 1984.

[30] K. Petersson, A programming system for type theory, Report 21, Department of Computer Sciences, Chalmers University, Göteborg, Sweden, 1982.

[31] D. Sannella, R. Burstall, Structured theories in LCF, Report CSR-129-83, Dept. of Computer Science, University of Edinburgh, 1983.

[32] D. Schmidt, A programming notation for tactical reasoning, in: R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, Springer LNCS 170, pages 445–459, 1984.

[33] S. Sokołowski, A note on tactics in LCF, Report CSR-140-83, Dept. of Computer Science, University of Edinburgh, 1983.

[34] S. Sokołowski, An LCF proof of the soundness of Hoare's logic, Report CSR-146-83, Dept. of Computer Science, University of Edinburgh, 1983.

[35] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M.I.T. Press, 1977.