Technical Report

UCAM-CL-TR-34
ISSN 1476-2986

Number 34

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Recent developments in LCF: examples of structural induction

Larry Paulson

January 1983

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

Recent Developments in LCF: Examples of Structural Induction

by Larry Paulson

University of Cambridge

January 1983

Abstract

Manna and Waldinger have outlined a large proof that probably exceeds the power of current automatic theorem-provers. The proof establishes the unification algorithm for terms composed of variables, constants, and other terms. Two theorems from this proof, involving structural induction, are performed in the LCF proof assistant. These theorems concern a function that searches for an occurrence of one term inside another, and a function that lists the variables in a term.

Formally, terms are regarded as abstract syntax trees. LCF automatically builds the first-order theory, with equality, of this recursive data structure.

The first theorem has a simple proof: induction followed by rewriting. The second theorem requires a cases split and substitution throughout the goal. Each theorem is proved by reducing the initial goal to simpler and simpler subgoals. LCF provides many standard proof strategies for attacking goals; the user can program additional ones in LCF's meta-language, ML. This flexibility allows the user to take ideas from such diverse fields as denotational semantics and logic programming.

Recent Developments in LCF:
Examples of Structural Induction

by Larry Paulson

University of Cambridge

January 1983

1. Introduction

An interactive proof assistant should be able to reason about a variety of data structures and algorithms, and automatically perform simple proofs. It should be flexible, allowing experimentation with different ways of expressing and performing proofs. These are the goals of the proof assistant Edinburgh LCF (Gordon, Milner, Wadsworth [1979]).

In view of the success of the Boyer and Moore [1979] theorem-prover, why should a proof assistant excite any interest? Manna and Waldinger [1981, page 47] manually prove the Unification Algorithm and remark, "Although the above proof may be beyond the power of current automatic systems, a partially interactive system could be used to produce it with known techniques. This approach requires more human effort, but it still would convey many of the benefits of automatic synthesis." This paper presents two theorems from Manna and Waldinger's theory.

2. Essential Background

LCF relies on one fundamental principle: proofs are conducted in a meta-language, ML. ML is a general-purpose functional programming language whose data values include terms, formulas, and theorems of the logic. Inference rules are ML functions that map theorems to theorems. The only way an ML program can prove a theorem is by applying inference rules to axioms or previously proved theorems.

LCF's logic, PPLAMBDA, uses Scott's theory of continuous partial orders (Stoy [1977]). PPLAMBDA has the usual introduction and elimination rules for each connective. Please note its ASCII representation of logical formulas (Figure 1).

<u>PPLAMBDA</u> <u>Terms</u>

```
c       constant, where c is a constant symbol
x       variable
\x.t    lambda-abstraction over a term
t u     combination (application of function to argument)

UU      "bottom" or "undefined" element
TT      truth-value "true"
FF      truth-value "false"
```

<u>PPLAMBDA</u> <u>Formulas</u>

```
t == u      equality of t and u
!x.A        universal quantifier
?x.A        existential quantifier
A /\ B      conjunction
A \/ B      disjunction
A ==> B     implication
A <=> B     if-and-only-if
~A          negation
etc.
```

Figure 1. Syntax of the logic PPLAMBDA

---

3. Axiomatising a Structure: Combinator Terms

Let us formalise Manna and Waldinger's [1981] theory of the Unification Algorithm, which concerns substitution over combinator terms. These are like PPLAMBDA terms without lambda-abstraction. For proofs we are only concerned with their abstract syntax:

```
term =  CONST const       |
        VAR var           |
        COMB term term
```

To axiomatise this structure in LCF, we introduce the abstract types "const", "var", and "term", then axiomatise them using LCF's structure package. LCF stores the types and axioms on a theory file, which can become part of a theory hierarchy.

```
%percent signs enclose comments%

new_type 0 `var`;;        %declare the types var, const, term%
new_type 0 `const`;;
new_type 0 `term`;;

struct_axm (":term",      %build the theory of terms%
      `strict`,
      [`CONST`, ["c:const"];
       `VAR`,   ["v:var"];
       `COMB`,  ["t1:term"; "t2:term"]] );;
```

The resulting theory includes the constructor functions CONST, VAR, and COMB, and axioms stating that these are distinct, one-to-one, etc. The constructor functions are all strict: for instance, CONST UU == UU. To build a theory that includes infinite and partially defined structures, call struct_axm with argument `lazy` instead of `strict`.

4. The Occurrence Relation

Our proofs concern the ordering relation "t OCCS u", an infix function that tests whether t occurs in u as a sub-structure. This requires a theory of the infix equality function, =. The structure package can prove theorems describing the outcome of the equality test[1] for various arguments; for instance, if v, t, u, t', u' are all defined, then

```
(COMB t u)=(COMB t' u')  ==  (t=t') AND (u=u')
(VAR v)=(COMB t u)  ==  FF
```

Figure 2 shows how to axiomatise the OCCS and OCCS_EQ functions, binding the axioms to ML names. The function OCCS_EQ tests "equals or occurs in." The function OCCS is defined as a set of clauses, one for each possible input:[2] a term cannot occur in a constant or variable, and occurs in COMB t1 t2 exactly if it equals or occurs in t1 or t2. This style of defining functions, reminiscent of Prolog (Clocksin and Mellish [1981]) or HOPE (Burstall et al. [1981]), eliminates the need for destructor and discriminator functions.

---

[1] Do not confuse the function = with the predicate ==. The formula "x==y", which may be proved using inference rules, asserts that x and y are equal. The term "x=y" represents a computable equality test applied to x and y. Likewise, do not confuse the truth-valued functions AND, OR, and NOT, with the logical connectives /\, \/, and ~.

[2] The first clause asserts that OCCS is strict; the other uses of UU confine the clauses to defined values only. Our theorems contain similar definedness hypotheses, a reflection that they were originally formulated for a first-order logic, not for PPLAMBDA.

```
let OCCS_EQ =
new_axiom (`OCCS_EQ`,
    "!t t2. t OCCS_EQ t2  ==  (t=t2) OR (t OCCS t2)");;


let OCCS_CLAUSES =
new_axiom (`OCCS_CLAUSES`,
"!t.  t OCCS UU == UU
   /\
   (!c. ~ c==UU  ==>
      t OCCS (CONST c) == FF)
   /\
   (!v. ~ v==UU  ==>
      t OCCS (VAR v) == FF)
   /\
   (!t1 t2. ~ t1==UU  ==>  ~ t2==UU  ==>
      t OCCS (COMB t1 t2) == (t OCCS_EQ t1) OR (t OCCS_EQ t2))");;
```

Figure 2. Axioms for the infix functions OCCS_EQ and OCCS.

5. The Variables Proposition

Our first theorem concerns a function VARS_OF, which computes a list of all the variables in a term. (We use a theory of lists, with constructors NIL and CONS, and infix operators APP for append, MEM for membership test.)

```
let VARS_OF_CLAUSES =
new_axiom (`VARS_OF_CLAUSES`,
    "   VARS_OF UU == UU
      /\
      (!c. ~ c == UU  ==>
         VARS_OF(CONST c) == NIL)
      /\
      (!v. ~ v == UU  ==>
         VARS_OF(VAR v) == CONS v NIL)
      /\
      (!t u. ~ t == UU  ==>  ~ u == UU  ==>
         VARS_OF(COMB t u) == (VARS_OF t) APP (VARS_OF u))");;
```

Let us prove that a variable v occurs in a term t exactly when v is a member of the list VARS_OF(t). We give LCF the <u>goal</u>:

```
set_goal ([],
    "!v. ~ v==UU  ==>
          !t. v MEM (VARS_OF t) == (VAR v) OCCS_EQ t"  );;
```

We will work backwards from the goal, by applying subgoaling functions, called tactics, to it. A tactic returns a list of subgoals, paired with a proof function that maps proofs of the subgoals to a proof of the original goal. By applying further tactics we reduce all the subgoals to trivial ones. Then we assemble the complete proof from the proof functions.

One simple tactic is GEN_TAC, which reasons that to prove !x.A(x) it suffices to choose a new variable x' and prove A(x'), since this theorem can then be generalised over x'. Another tactic is DISCH_TAC, which reasons that to prove A==>B, it suffices to prove B under the assumption A, since this assumption can then be discharged. Notation: the double bar means "suffices to prove"; assumptions are enclosed in [square brackets]; other assumptions of the goal are implicitly passed to the subgoals.

```
    !x.A(x)
=============        GEN_TAC
    A(x')


    A==>B
=============        DISCH_TAC
    [A] B
```

For interactive proof, LCF's subgoal package is convenient: it stacks pending and solved subgoals, displays the current goal, and applies the proof functions in the correct order. You can apply tactics and back up from faulty steps.

5.1. The structural induction tactic

The structure package provides a tactic, TERM_INDUCT_TAC, to perform structural induction on a goal !t.A(t). This produces four subgoals: t may be a COMB (the step case); t may be a VAR or CONST (the base cases); t may be UU (the undefined case). The subgoals include induction hypotheses and assumptions that the sub-structures are defined.

```
                                !t. A(t)                 TERM_INDUCT_TAC
==========================================================
[ A(t1) ; A(t2) ; ~ t1==UU ; ~ t2 ==UU ]    A(COMB t1 t2)
                            [ ~ v == UU ]    A(VAR v)
                            [ ~ c == UU ]    A(CONST c)
                                             A(UU)
```

Let us ask the subgoal package to expand the current goal using TERM_INDUCT_TAC. The induction variable t is submerged inside the goal, so the tactic calls GEN_TAC and DISCH_TAC before applying induction.

```
expand (TERM_INDUCT_TAC "t");;
```

5.2. The rewriting tactic

LCF prints the four resulting subgoals and their assumptions (Figure 3). We can prove each one by rewriting: if we have a theorem t==u, change the goal by replacing every instance t' of t by the corresponding instance u'. For implicative rewrites, a theorem A==>(t==u) may be used to rewrite an instance t' by u' if the antecedent A' can be proved.

```
"v MEM (VARS_OF(COMB t1 t2)) == (VAR v) OCCS_EQ (COMB t1 t2)"
     [ "~ v == UU" ]
     [ "v MEM (VARS_OF t1) == (VAR v) OCCS_EQ t1" ]
     [ "v MEM (VARS_OF t2) == (VAR v) OCCS_EQ t2" ]
     [ "~ t1 == UU" ]
     [ "~ t2 == UU" ]

"v MEM (VARS_OF(VAR v')) == (VAR v) OCCS_EQ (VAR v')"
     [ "~ v == UU" ]
     [ "~ v' == UU" ]

"v MEM (VARS_OF(CONST c)) == (VAR v) OCCS_EQ (CONST c)"
     [ "~ v == UU" ]
     [ "~ c == UU" ]

"v MEM (VARS_OF UU) == (VAR v) OCCS_EQ UU"
     [ "~ v == UU" ]
```

Figure 3. Subgoals after applying induction.

The most interesting case involves terms of the form (COMB t1 t2), with induction hypotheses for t1 and t2. The left and right sides converge:

```
v MEM (VARS_OF (COMB t1 t2))
        unfolding the definition of VARS_OF   --->
v MEM ((VARS_OF t1) APP (VARS_OF t2))
        by a theorem about MEM, APP, and OR   --->
(v MEM (VARS_OF t1)) OR (v MEM (VARS_OF t2))


(VAR v) OCCS_EQ (COMB t1 t2)
        unfolding the definition of OCCS_EQ   --->
(VAR v)=(COMB t1 t2) OR
(((VAR v) OCCS_EQ t1) OR ((VAR v) OCCS_EQ t2))
        since any VAR is distinct from any COMB   --->
(((VAR v) OCCS_EQ t1) OR ((VAR v) OCCS_EQ t2))
        by the induction hypotheses   --->
(v MEM (VARS_OF t1)) OR (v MEM (VARS_OF t2))
```

The other goals converge similarly. To perform such reasoning, LCF provides the tactic ASM_REWRITE_TAC. This rewrites the goal using its assumptions and a list of theorems furnished by the user. The symbols AND_CLAUSES, OR_CLAUSES, etc., denote axioms and theorems from parent theories.

```
expand (ASM_REWRITE_TAC
          [AND_CLAUSES; OR_CLAUSES;
           TERM_EQUAL_ALL;
           MEM_CLAUSES; MEM_SINGLE; MEM_APP;
           VARS_OF_CLAUSES; VARS_OF_TOTAL;
           OCCS_EQ_CLAUSES] );;
```

LCF reports that the subgoal is solved and prints those that remain. The above call of ASM_REWRITE_TAC includes enough theorems to solve any of the four subgoals.

5.3. Summarising the proof

Now that the interactive proof is complete, let us combine the tactics we used into a composite one that performs the entire proof. For combining tactics, LCF provides functions called *tacticals*. The basic ones are THEN, ORELSE, and REPEAT.

```
TAC1  THEN  TAC2
        calls TAC1, then applies TAC2 to all resulting subgoals

TAC1 ORELSE TAC2
        calls TAC1, if it fails then calls TAC2

REPEAT TAC
        calls TAC repeatedly on the goal and its subgoals
```

The tactic that proves the Variables theorem is

```
TERM_INDUCT_TAC  "t"   THEN
ASM_REWRITE_TAC [AND_CLAUSES; OR_CLAUSES; etc.]
```

In words, the proof is induction followed by rewriting. Many proofs have this simple form -- for instance, properties of list utilities (append, map, membership), and totality of recursively defined functions.

Given a set of theorems, ASM_REWRITE_TAC strips off universal quantifiers, splits apart conjunctions, and decides which of the resulting pieces are useful for rewriting or for solving implicative rewrites. It accepts not only term rewrites, t==u, but also <u>formula rewrites</u>, A<=>B. After rewriting, it removes tautologies from the goal -- perhaps solving it completely, returning an empty subgoal list.

Edinburgh LCF provided a similar tactic, SIMPTAC, consisting of seven inscrutable pages of ML. Since SIMPTAC was impossible to modify, Avra Cohn [1982] spent considerable effort adapting her proofs to its limitations. In contrast, ASM_REWRITE_TAC has a modular construction. Its apparently baroque strategy is controlled by a twelve-line ML function that calls tactics, pattern matchers, canonical form translators, rewriting functions, and tautology checkers. These components can easily be changed to suit individual needs or correct shortcomings.

6. Transitivity of the Occurrence Relation

Let us prove a more difficult theorem, that the ordering relation OCCS is transitive:

```
!ta. ~ ta==UU  ==>
!tb. ta OCCS tb == TT  ==>
!tc. tb OCCS tc == TT  ==>  ta OCCS tc == TT
```

If we induct on the variable tc, rewriting solves only three of its four subgoals.[3] The COMB case remains:

```
"((tb = t1) OR (tb OCCS t1)) OR
 ((tb = t2) OR (tb OCCS t2))        == TT
 ==>
 ((ta = t1) OR (ta OCCS t1)) OR
 ((ta = t2) OR (ta OCCS t2))        == TT"
     [ "~ ta == UU" ]
     [ "ta OCCS tb == TT" ]
     [ "tb OCCS t1 == TT  ==>  ta OCCS t1 == TT" ]
     [ "tb OCCS t2 == TT  ==>  ta OCCS t2 == TT" ]
     [ "~ t1 == UU" ]
     [ "~ t2 == UU" ]
```

The goal has the form A==>B, so we can use DISCH_TAC to attempt proving B by assuming the antecedent A. Since B has the form b1 OR b2 OR b3 OR b4 == TT, it suffices to prove that one of b1, b2, b3, or b4 equals TT. The antecedent A has the form a1 OR a2 OR a3 OR a4 == TT, and, by studying the assumptions, we realise that if any ai equals TT, then the corresponding bi must equal TT.

If A were a disjunction A1 \/ A2 \/ A3 \/ A4, then we could split into four subgoals, proving B in each case of whether A1, A2, A3, or A4 held. Now A uses the truth-valued functions OR and =, rather than the logical connectives \/ and ==, but we have theorems to correct this, using a weaker kind of formula rewriting:

```
OR_EQ_TT    !p q. p OR q == TT    ==>    p==TT  \/  q==TT
EQUAL_TT    !x y.    x=y == TT    ==>    x==y
```

[3] These three hold vacuously, by contradicting the antecedent tb OCCS tc == TT.

The inference rule MP_CHAIN, given a list of implications, recursively modifies a theorem using Modus Ponens on it and its parts. With the above theorems, it can change our antecedent to a disjunction of equalities.

```
before:
((tb = t1) OR (tb OCCS t1)) OR
((tb = t2) OR (tb OCCS t2))        == TT

after:
tb==t1  \/  tb OCCS t1 == TT  \/
tb==t2  \/  tb OCCS t2 == TT
```

This theorem has the proper form for the tactic SUBST_CASES_TAC. This splits the goal into four cases, and substitutes each equality through the corresponding goal and its assumptions. Figure 4 shows the first two cases; the other two are similar. An asterisk (*) marks those assumptions which, altered by the substitution, match part of the goal. Now ASM_REWRITE_TAC can finish each case.

```
"((ta = t1) OR (ta OCCS t1)) OR ((ta = t2) OR (ta OCCS t2)) == TT"
      [ "~ ta == UU" ]
      [ "ta OCCS tb == TT" ]
 *    [ "TT == TT  ==>  ta OCCS t1 == TT" ]
      [ "tb OCCS t2 == TT  ==>  ta OCCS t2 == TT" ]
      [ "~ t1 == UU" ]
      [ "~ t2 == UU" ]

"((ta = t1) OR (ta OCCS t1)) OR ((ta = t2) OR (ta OCCS t2)) == TT"
      [ "~ ta == UU" ]
 *    [ "ta OCCS t1 == TT" ]
      [ "t1 OCCS t1 == TT  ==>  ta OCCS t1 == TT" ]
      [ "t1 OCCS t2 == TT  ==>  ta OCCS t2 == TT" ]
      [ "~ t1 == UU" ]
      [ "~ t2 == UU" ]
```

Figure 4. Two cases after substitution in the assumptions.

In the goal A==>B, how do we grab hold of the antecedent A, to put it through MP_CHAIN and SUBST_CASES_TAC? During the interactive search for a proof, we might use DISCH_TAC to put A on the assumption list, then use one of LCF's tacticals for manipulating assumptions. But we have a slick way of expressing the completed proof. Instead of the tactic DISCH_TAC, we can use the tactical DISCH_THEN, which binds the antecedent A to a variable for further use.

```
TERM_TAC "tc"   THEN
ASM_REWRITE_TAC [OCCS_CLAUSES; OCCS_EQ]
THEN
%solves all base cases, but the COMB case remains%
DISCH_THEN      %binds antecedent to a variable%
  (\ante. SUBST_CASES_TAC  (MP_CHAIN [OR_EQ_TT; EQUAL_TT] ante))
THEN
%splits into four cases%
ASM_REWRITE_TAC [OR_CLAUSES; OR_R_TT; OR_TOTAL;
                 TERM_EQUAL_TOTAL; OCCS_TOTAL]
```

In the jargon of denotational semantics (Stoy [1977]), the argument to DISCH_THEN is a continuation that tells what to do with the antecedent. Only time will tell whether such a high-powered approach can be justified; flexibility to try different styles is the hallmark of LCF.

7. Postscript

We can prove many similar theorems. The ordering relation OCCS is anti-reflexive; it is also monotonic with respect to substitution.

```
!t. ~ t OCCS t == TT

!sl. ~ sl==UU  ==>
!t. ~ t==UU  ==>
!u. t OCCS u == TT  ==>
    (t SUBST sl) OCCS (u SUBST sl) == TT
```

Most of these proofs are straight-forward inductions, but some reveal weaknesses in LCF. For instance, we plan to extend the backwards-chaining primitives to handle existential implications such as (?x.A)==>B. This would be executing PPLAMBDA theorems as a Prolog program (Clocksin and Mellish [1981]).

LCF's methods apply to any logic. The logic PPLAMBDA can complicate first-order problems, adding cases about undefined elements. However, the extra cases are usually trivial. PPLAMBDA is essential for proofs about denotational semantics, compiler correctness, lazy evaluation, and higher-order functional programs.

In such a short paper it is impossible to document, motivate, or even mention all the techniques -- particularly experimental ones. You may not see how LCF helped to discover the proofs shown here, since I have omitted the fruitless first attempts. Look again at the subgoals in Figures 3 and 4, which LCF printed. Imagine writing them out by hand. With computer assistance, we can hope to prove theorems involving increasingly complex data structures.

Acknowledgments. I would like to thank Gerard Huet for porting the Lisp sources, and Mike Gordon for daily discussions. Robin Milner wrote the first structure package.

References

R. Boyer, J. Moore. A Computational Logic. Academic Press, 1979.

R. Burstall, D. MacQueen, D. Sannella. "HOPE: An Experimental Applicative Language." Technical Report CSR-62-80, University of Edinburgh, 1981.

W. Clocksin, C. Mellish. Programming in Prolog, Springer-Verlag, 1981.

A. Cohn, R. Milner. "On using Edinburgh LCF to prove the correctness of a parsing algorithm." Technical Report CSR-113-82, University of Edinburgh, 1982.

A. Cohn. "The correctness of a predecence parsing algorithm in LCF." Technical Report No. 21, University of Cambridge, 1982.

M. Gordon, R. Milner, C. Wadsworth. Edinburgh LCF. Springer-Verlag, 1979.

Z. Manna, R. Waldinger. "Deductive Synthesis of the Unification Algorithm." Science of Computer Programming, 1981, pages 5-48.

J. Stoy. Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977.