Technical Report

UCAM-CL-TR-20
ISSN 1476-2986

Number 20

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

On using Edinburgh LCF to prove the correctness of a parsing algorithm

Avra Cohn, Robin Milner

February 1982

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

On Using Edinburgh LCF to Prove the Correctness of a Parsing Algorithm

by

Avra Cohn
Computer Laboratory
University of Cambridge

and

Robin Milner
Computer Science Department
University of Edinburgh

February 1982*

Abstract

The methodology of Edinburgh LCF, a mechanized interactive proof system, is illustrated through a problem suggested by Gloess - the proof of a simple parsing algorithm. The paper is self-contained, giving only the relevant details of the LCF proof system. It is shown how tactics may be composed in LCF to yield a strategy which is appropriate for the parser problem but which is also of a generally useful form. Also illustrated is a general mechanized method of deriving structural induction rules within the system.

---

* The second author acknowledges the support of a grant from the Venture Research Unit of the British Petroleum Company.

1. INTRODUCTION

In this paper we give an account of an exercise in proving the correctness of a simple parsing algorithm in the LCF proof system [GMW]. The problem was suggested by a paper of Gloess [Glo], which describes his proof conducted in the Boyer-Moore proof system [BM], and some comparisons with his approach are made in the final section of the paper.

The main aim of the paper is to present the LCF methodology through a problem which is well-suited to this purpose. The paper is self-contained, and gives details of the LCF system when and as required. In sections 2-4 we introduce and solve the problem informally, but following the sequence which is later formalised; first the domains of the problem are introduced, then the necessary theory of these domains is developed, and finally the problem is formulated and the proof presented. Section 5 is concerned first with the necessary syntactic details, and then outlines the necessary interaction with LCF which leads to the formal statement of the problem. In Section 6 the proof methodology is described, and the section is mainly concerned with building - in the LCF metalanguage - a strategy which generates a proof of the correctness of the parser. Section 7 outlines the use of a standard LCF package for deriving induction rules, using as an example the rule of tree induction required in the parser proof. The whole methodology is discussed in the final section, where we also allude to a proof of a more complex parser to be presented in another paper.

The main emphasis of the paper is first on the natural expression in LCF of algorithms and of the statement of their properties, and second on the rôle of LCF's metalanguage (ML) in extending the basic logic to discuss new domains, in deriving new inference rules and in expressing strategies for generating proof. No formal proof in the logicians' sense appears in the paper, nor is it necessary; our point of view is that a strategy - or recipe - for proof should be presented to a machine much as it is communicated between mathematicians, and the machine can safely be left to perform it without error (complaining, of course, if it fails to work).

ACKNOWLEDGEMENTS

We would like to thank Paul Gloess for stimulating us to study this natural problem, and Jacek Leszczylowski for his help in designing the heart of our proof strategy.

2. WORDS AND PARSE-TREES

The parsing algorithm takes words, over an alphabet of symbols, to parse-trees. The class of well-formed (parsable) words can be specified by the BNF syntax

$$w ::= x \mid u\ w \mid \text{"("}\ w\ b\ w\ \text{")"}$$

where x, u and b range over identifiers, unary operators and binary operators respectively.

We wish to regard words as lists of symbols; this we represent as the domain (or type) definition

WORD = SYMB LIST

where SYMB is the domain of symbols, and LIST is a postfixed domain operator.

To set up the domain of symbols, we presuppose two domains ID of identifiers and OP of operators, specific to our problem, and a standard domain ONE containing one proper element. (In our model, every domain contains an improper element, the undefined element, so ONE really has two members; for the present the improper element can be ignored.) Since a symbol can be given by the syntax

$$s ::= \text{"("} \mid \text{")"} \mid x \mid u \mid b$$

we make the definition

SYMB = ONE + ONE + ID + OP + OP

- a disjoint sum. Two constants and three injections, with their types, are defined as follows:

LB: SYMB

RB: SYMB

IDEN: ID $\rightarrow$ SYMB

UNARY: OP $\rightarrow$ SYMB

BINARY: OP $\rightarrow$ SYMB

They specify how parentheses, identifiers and operators are embedded in the symbol domain, through its five respective summand domains.

The standard domain operator LIST is taken to satisfy the domain isomorphism

D LIST $\simeq$ ONE +(D $\times$ D LIST)

A constant and an injection

NIL: D LIST

CONS: D $\rightarrow$ D LIST $\rightarrow$ D LIST specify how lists are constructed from elements of an arbitrary domain D. Thus these two operations are polymorphic. For concatenating lists, the function

APP: D LIST $\longrightarrow$ D LIST $\longrightarrow$ D LIST

can be defined so that the following equations hold for all X in D and all L, L', L" in D LIST:

A1. APP NIL L = L

A2. APP(CONS X L)L' = CONS X (APP L L')

A3. APP(APP L L')L" = APP L (APP L' L")

A parse-tree, for our problem, is taken to be <u>either</u> a tip consisting of an identifier, <u>or</u> a unary node consisting of an operator and a parse-tree, <u>or</u> a binary node consisting of an operator and two parse-trees. Thus the domain PTREE satisfies

PTREE $\simeq$ ID + OP $\times$ PTREE + OP $\times$ PTREE $\times$ PTREE

with three injections

mkTIP: ID $\longrightarrow$ PTREE

mkUN: OP $\longrightarrow$ PTREE $\longrightarrow$ PTREE

mkBIN: OP $\longrightarrow$ PTREE $\longrightarrow$ PTREE $\longrightarrow$ PTREE

These domains WORD and PTREE, and the associated constants and injections, provide all the necessary basis for defining the parsing algorithm, and for formulating and proving its correctness.

3. THE PARSER AND UNPARSER

The parsing algorithm accepts an arbitrary word w, and produces a pair consisting of

(i) a parse-tree corresponding to some initial segment of w which is a well-formed word;

(ii) a word which is the remainder of w.

Thus its type is given by

parse: WORD $\longrightarrow$ (PTREE × WORD)

It uses two auxiliary functions. The first detects a binary operator at the head of its WORD argument, parses the ensuing string, and combines the resulting tree with its PTREE argument (which represents the already parsed first operand of the detected operator):

parse2: PTREE $\longrightarrow$ (WORD $\longrightarrow$ (PTREE × WORD))

The second merely detects a right bracket and discards it:

choprb: WORD $\longrightarrow$ WORD

We choose to present the algorithm as a set of clauses, one for each acceptable form of argument. (These clauses can easily be proved from the presentation of a recursive algorithm, which performs explicit case analysis on the argument. The present formulation is not only easier to read - a point recognised by PROLOG programmers for example - but also conveniently omits the error action for ill-formed input words which must be specified by a complete algorithm.)

P1. parse (CONS(IDEN I)w) = (mkTIP I, w)

P2. parse (CONS(UNARY op)w) = <u>let</u> t',w' = parse w
<u>in</u> (mkUN op t', w')

P3. parse (CONS LB w) = <u>let</u> t',w' = parse w
<u>in</u> parse2 t' w'

P4. parse2 t (CONS(BINARY op)w) = <u>let</u> t',w' = parse w
<u>in</u> (mkBIN op t t', choprb w')

P5. choprb(CONS RB w) = w

To formulate the correctnss of the parser, we shall need a function yielding for each parse-tree the unique word which it represents. This function is

unparse: PTREE $\longrightarrow$ WORD

and has the clauses

U1. unparse (mkTIP I) = CONS(IDEN I)NIL

U2. unparse (mkUN op t) = CONS(UNARY op)(unparse t)

U3. unparse (mkBIN op t1 t2) = APP(CONS LB(unparse t1))
(APP(CONS(BINARY op)(unparse t2))
(CONS RB NIL)))

4. STATEMENT AND PROOF OF CORRECTNESS

We follow Gloess essentially in formulating what should be proved about the parser. We wish to say that it treats correctly any word with a "parsable" initial segment, that is to say an initial segment which represents some parse-tree. Such a word must have the form APP(unparse t)w for some tree t and word w, so we naturally require

parse (APP(unparse t)w) = (t,w)

for all suitable t and w. We cannot require it for all trees t, since the domain PTREE contains trees which are infinite or partially defined or both. But if some formula WD[t] (with free variable t) characterises finite well-defined trees, then we can formulate correctness by

$\forall$t. $\forall$w. WD[t] $\supset$ parse (APP(unparse t)w) = (t,w)

Later we shall formulate WD[t] explicitly; we state now the properties of it which the proof requires. The first property concerns the completely undefined tree UU; every domain in the interpretation of our logic contains such an object as its minimum element, denoted by the polymorphic constant UU. Then the following must hold for all identifiers I, operators op and trees t, t1, t2:

WD1. WD[UU] is a contradiction (i.e. WD[UU] $\supset$ f is a theorem for any formula f).

WD2. WD[mkTIP I]

WD3. WD[mkUN op t] $\supset$ WD[t]

WD4. WD[mkBIN op t1 t2] $\supset$ WD[t1]

WD5. WD[mkBIN op t1 t2] $\supset$ WD[t2]

Now the proof of correctness proceeds by structural induction on trees t. The structural induction rule for trees, like all structural induction rules in our logic, is derivable from the basic induction rule (computation induction) once PTREE has been axiomatized; see Section 7 for the derivation. The rule is as follows, with hypotheses and conclusion written above and below a horizontal line:

$$\frac{\begin{array}{l}\mathcal{P}[UU] \\ \forall I.\ \mathcal{P}[mkTIP\ I] \\ \forall op\ t.\ \mathcal{P}[t] \supset \mathcal{P}[mkUN\ op\ t] \\ \forall op\ t1\ t2.\ \mathcal{P}[t1]\ \&\ P[t2] \supset \mathcal{P}[mkBIN\ op\ t1\ t2]\end{array}}{\forall t.\ \mathcal{P}[t]}$$

Here $\mathcal{P}[t]$ is any suitable property of trees. The proof is not complex, and uses only facts which have been already mentioned, but we outline it in order to refer to its structure later when we show how this structure can be presented as a composite proof strategy in our metalanguage.

Let $\mathcal{P}[t] \equiv \forall w.\ \mathcal{Q}[w,t]$,

where $\mathcal{Q}[w,t] \equiv WD[t] \supset parse(APP(unparse\ t)w) = (t,w)$

Note that the induction requires that Q[w,t], as inductive hypothesis, must be assumed for arbitrary w; the universal quantifier w is necessary in the induction formula $\mathcal{P}[t]$.

<u>To prove</u>: $\forall t.\ \mathcal{P}[t]$

<u>Undefined Case</u>

$\mathcal{P}[UU]$ holds since WD[UU] is a contradiction.

<u>Tip Case</u>

We must prove Q[w, mkTIP I] for arbitrary w,I.

<u>Assume</u> WD[mk tip I]

<u>Prove</u> parse(APP(unparse(mk TIP I))w) = (mk TIP I, w)

This follows directly by using U1, A2, A1 and P1 as rewriting rules.

<u>Unary Case</u>

<u>Assume</u> $\mathcal{P}[t]$ (IH)

We must prove $\mathcal{Q}$[w, mkUN op t] for arbitrary w and op.

<u>Assume</u> WD[mkUN op t] (ASS)

<u>Prove</u> parse(APP(unparse(mkUN op t))w) = (mkUN op t, w)

By using U2, A2 and P2 as rewriting rules, we reduce the left side to

LHS = <u>let</u> t',w' = parse (APP(unparse t)w)

<u>in</u> (mkUN op t',w')

But from ASS, WD3 and IH we obtain

parse(APP(unparse t)w) = (t,w)

and the result follows.

Binary Case

Assume $\mathcal{P}$[t1], $\mathcal{P}$[t2] (IH1, IH2)

We must prove $\mathcal{Q}$[w, mkBIN op t1 t2] for arbitrary w and op.

Assume WD[mkBIN op t1 t2] (ASS)

Prove parse(APP(unparse(mkBIN op t1 t2)w) = (mkBIN op t1 t2, w)

By using U3, A2, A3 and P3 as rewriting rules, we reduce the left side to

```
LHS = let t',w' = parse(APP(unparse t1)
                                (CONS(BINARY op)
                                    (APP(unparse t2)
                                        (CONS RB w))))
        in parse2 t' w'
```

Now from ASS, WD4 and IH1 (with appropriate instantiation of its universally quantified w) the right-hand side of the let clause reduces to a pair, and we obtain

```
LHS = parse2 t1(CONS(BINARY op)(APP(unparse t2)
                                    (CONS RB w)))
```

and by P4

```
    = let t',w' = parse(APP(unparse t2)(CONS RB w))
        in (mkBIN op t1 t', choprb w')
```

Again, from ASS, WD5 and an instantiation of IH2 the right-hand side of the let clause reduces to the pair (t2, CONS RB w), and finally by P5 we get

```
LHS = (mkBIN op t1 t2, w)
```

as required.

5. FORMALISATION

In this section, we show how the parser and its statement of correctness are formalised in LCF, by the construction of simple applied theories.

The LCF system consists of a logical calculus PPLAMBDA (Polymorphic Predicate LAMBDA calculus), together with a programming meta-language ML in which logical entities are manipulated. The latter term includes both performing inference and programming inference strategies.

The terms t of PPLAMBDA are, as in the lambda calculus

t::= c | x | (t t') | λx.t

where c ranges over constants, x over variables. Each term has a type corresponding to a domain, e.g. the type tr of truth values. Types may be built from type constants (e.g. tr) and type variables (e.g. *, **) by normally used type operators, and may be used in terms, with a prefixed colon, to qualify terms. Standard constants, with their types, include

TT:tr, FF:tr	Truth values
UU:*	undefined (the improper element)
DEF:* → tr	yields UU:tr on UU, TT:tr otherwise
, :* → ** → * x **	the pairing function

The last is infixed, allowing the syntax (t, t') for pairs.

The formulae f of PPLAMBDA are, as in the predicate calculus,

f::= TRUTH | t ==t' | t << t' | f & f' | f IMP f' | !x.f

TRUTH (distinct from the term TT) is a constant formula; "==" and "<<" are predicate constants standing for equality and partial order in domains; the remaining clauses are conjunction, implication and universal quantification (we shall henceforth use this non-standard notation, imposed by limitations of machine character-sets).

Applied calculi, or theories, may be built hierarchically upon PPLAMBDA by meta-linguistic operations for creating types, constants and axioms. We illustrate the process by building the PARSE theory from three sub-theories LIST, SYMB and TREE (each of which may serve as a sub-theory for many other theories).

To construct LIST, we first create the unary type operator LIST. Then we create the constants

NIL: * LIST

CONS: * → * LIST → * LIST

Two kinds of axiom are needed. First, certain axioms ensure the isomorphism

* LIST ≃ ONE +(* x * LIST)

In fact, two further constants - representing the isomorphism and its inverse - are needed to express these axioms. Second, NIL and CONS are defined as injections into * LIST via the isomorphism and with the help of standard constants associated with sum and product domains. We shall not give further details of these constructions; the structure package outlined in Section 7 can be used to automate the construction of the LIST theory, and to provide the appropriate induction rule.

At this point, the LIST theory can be extended at will by further constants and axiomatic definitions; in particular the function APP, defined recursively in the usual way, can be proved to satisfy the three properties A1-A3 listed in section 2. These theorems may be recorded permanently as part of the LIST theory.

To construct SYMB, an entirely analogous process begins with the introduction of the nullary type operators - or type constants - ID, OP and SYMB. The only differences is that the isomorphism

$$\mathrm{SYMB} \simeq \mathrm{ONE} + \mathrm{ONE} + \mathrm{ID} + \mathrm{OP} + \mathrm{OP}$$

corresponds this time to a non-recursive domain definition.

To construct PTREE, there are two possibilities. In one method, we build it upon the theory SYMB by introducing PTREE as a type constant, axiomatizing the isomorphism

$$\mathrm{PTREE} \simeq \mathrm{ID} + \mathrm{OP} \times \mathrm{PTREE} + \mathrm{OP} \times \mathrm{PTREE} \times \mathrm{PTREE}$$

and defining mkTIP, mkUN and mkBIN as injections. The meta-program mentioned above then provides the induction rule which we used in our informal proof. In the other method, we may proceed more generally by creating a theory TREE, with ternary type operator TREE, so that polymorphic type (*, **, ***)TREE - abbreviated to T - satisfies the isomorphism

$$T \simeq * + ** \times T + *** \times T \times T$$

The injections

mkTIP: $* \longrightarrow T$

mkUN: $** \longrightarrow T \longrightarrow T$

mkBIN: $*** \longrightarrow T \longrightarrow T \longrightarrow T$

are then defined polymorphically; they are available at all instances of the polymorphic type T. Furthermore, the induction rule for these general trees is also available at all instance types.

Adopting the second alternative we now wish to build the theory PARSE on top of three independent theories; the hierarchy can be pictured

$$\begin{tikzcd}
\mathrm{LIST} \arrow[dr] & \mathrm{SYMB} \arrow[d] & \mathrm{TREE} \arrow[dl] \\
& \mathrm{PARSE} &
\end{tikzcd}$$

The first step is to introduce the type definitions

WORD = SYMB LIST

PTREE = (ID, OP, OP)TREE

PW = PTREE × WORD

and the constants

parse: WORD ⟶ PW

parse2: PTREE ⟶ WORD ⟶ PW

choprb: WORD ⟶ WORD

unparse: PTREE ⟶ WORD

Next, in order to present the clauses P1-P5, U1-U3 as axioms in a graceful way, we add a new infixed operator

INTO: PW × (PTREE ⟶ WORD ⟶ PW) ⟶ PW

to represent the informal let - in construct. We define it by the axiom

⊢ (t,w) INTO f == f t w

(Any axiom containing variables is universally quantified over these variables on introduction.) Now the clauses P1-P5 and U1-U3 are introduced as axioms. For P2, for example, we write

⊢ parse (CONS(UNARY op) w) == parse w INTO λt'.λw'.(mkUN op t', w')

Note that if, for some particular w, parse w reduces to a pair (t', w'), then the above two axioms and lambda conversion allow us to prove

⊢ parse (CONS(UNARY op)w) == (mkUN op t', w')

as expected.

All that remains, in order to formulate the parser correctness, is to find a formula WD[t] for which the properties WD1-WD5 may be proved. For this purpose, we introduce a final constant

wd: PTREE ⟶ tr

with the defining axioms

⊢ wd(UU) == UU

⊢ wd(mkTIP I) == TT

⊢ wd(mkUN op t) == wd(t)

⊢ wd(mkBIN op t1 t2) == wd(t1) ⇒ wd(t2) | UU

where the conditional construct -- ⇒ -- | -- is standard syntax for the standard ternary conditional operator of PPLAMBDA. We then take WD[t] to be the formula

wd(t) == TT

and indeed the properties WD1-WD5 are easily proved by structural induction. The proof is preparatory to the main proof; it can be argued that these properties would be required for many problems , so need not be considered as part of our particular problem.

6. THE FORMAL PROOF

In this section we describe how LCF, with guidance, can be led to perform the correctness proof which we presented informally in Section 4. The relevant LCF concepts will be explained when and where necessary. Before attending to detail, however, it is worth examining the form which the informal proof takes, and which is common in most mathematical proof. It is predominantly goal-directed ; repeatedly, a goal or a subgoal is replaced by subgoals, generated by a variety of methods. Often, these methods are validated by appeal to a single (basic or derived) inference rule. In particular, the main goal is immediately analysed into separate subgoals by appeal to structural induction; a quantified goal is replaced by one without the quantifier ("prove... for arbitrary x") by appeal to the rule of generalization; an implicative goal is replaced by its consequent (the antecedent being assumed) by appeal to the rule of discharge of implication. The entire proof uses a mixture of such subgoaling methods - we call them tactics - with direct inference and rewriting. Such a mixture, as distinct from its application to a particular main goal, may be called a recipe for proof, or a strategy. We aim to extract from our informal proof a strategy which succeeds for our particular problem, and which deserves the name "strategy" because it would also succeed (with perhaps a change of parameters) for other problems. The strategy will be expressed in ML. We argue that such a strategy expression, because of its structure and the extent to which it suppresses detail, is a helpful answer to the question "How do you prove X?" ; in this respect it compares favourably with a fully formal proof, i.e. a sequence of steps each following by basic inference from previous steps. LCF could indeed print out the latter (else the strategy would not have succeeded), since it does indeed perform it; in fact it executes a procedure corresponding to each basic inference. But we certainly do not always want to watch the performance since we rely upon its correctness.

The two parts of the LCF system, ML and PPLAMBDA, are connected through the abstract types (or metatypes) of ML. That is, the language of PPLAMBDA is represented by the metatypes term, form and type. Also, a theorem of PPLAMBDA is an object of metatype thm, whose only constructors are the rules of inference of the logic - which in turn are examples of ML procedures. A theorem consisting of a set A of assumption formulae and a conclusion formula f is written

$$A \vdash f$$

(PPLAMBDA is a sequent calculus). Occasionally an assumption will be represented by a period, when the intended formula is clear from the context, so that a theorem with conclusion f and two assumptions may be written ". . $\vdash$ f" .

LCF can accommodate both <u>forward</u> proof (successive application of inference rules) and <u>goal-oriented</u> proof. In the latter method one sets out a <u>goal</u> to be achieved and applies to it <u>tactics</u>, which generate subgoals as well as a means of mapping theorems achieving the subgoals to a theorem achieving to original goal (i.e. a means of generating the intermediate chain of theorems). Often, a mixture of forward and goal-oriented proof is successful; this paper, indeed, is about one such mixture.

A goal is a composite ML object. It includes of course the goal formula, such as

```
!t w.  wd t == TT   IMP parse(APP(unparse t)w)==(t,w)
```

in our example; it also includes a set of assumption formulae. (So far, then, a goal is a sequent.) For example, midway in the Unary case of our informal proof is a subgoal with formula

```
parse(APP(unparse(mkUN op t))w)==(mkUN op t, w)
```

under two assumptions

```
!w. wd t==TT  IMP  parse(APP(unparse t)w)==(t,w)
wd(mkUN op t)== TT
```

The third and last component of a goal reflects the observation that most steps in a proof are just left-to-right applications of proved equations, such as the facts A1-A3 concerning APP. One wishes to apply such an equation, which is universally quantified over its variables, whenever a match can be found between its left-hand side and some subterm of the goal formula, by an instantiation of variables. Our informal proof contains many instances of such rewriting.

Facts to be used thus as simplification rules are included in the simpset component of a goal. simpset is another abstract metatype; each member of a simpset - usually an equational theorem - is called a simprule. But a simprule is also allowed to be conditional; an example is the induction hypotheses in the Unary case of our informal proof, namely

!w. wd(t)==TT IMP parse(APP(unparse t)w)==(t,w)

The consequent of such an implication is only used in simplification when the appropriately instantiated antecedent can first be reduced to a tantology, also by simplification. This process is applied recursively.

In summary, the metatype goal is defined as

goal = form × simpset × form list

(metatypes in ML are built analogously to types in PPLAMBDA). A goal-oriented proof is advanced by applying tactics to goals. A tactic is an ML procedure which, given a goal, returns both a list of subgoals and a justification, so we have the metatype definition

tactic = goal → (goal list × proof)

where

proof = thm list → thm

That is, a proof maps an achievement of the subgoals (a list of theorems) to an achievement of the goal (a theorem).

As we mentioned, a simple tactic is often justified by a single rule of inference. For example, the tactic "prove ... for arbitrary x" may be pictured as:

$$\text{GENTAC:}\quad \frac{\text{"!x.f", S, A}}{\text{"f[x'/x]", S, A}} \qquad \text{where x' is new}$$

and means: to prove "!x.f", prove "f[x'/x]" for an arbitrary new variable x'. The proof function returned by this tactic appeals to the PPLAMBDA inference rule

$$\text{GEN:}\quad \frac{A \vdash f}{A \vdash \text{!x.f}} \qquad \text{(x not free in A)}$$

Because the tactic inverts GEN, it is called GENTAC. It is pre-programmed (very simply) in ML. Another simple tactic, which inverts the rule of implification discharge, is

$$\text{DISCHTAC:}\quad \frac{\text{"f1 IMP f2", S, A}}{\text{"f2", \{"f1} \vdash \text{f1"\}} \cup \text{S, "f1".A}}$$

Note that the antecedent f1 is added to the assumption list (by an infixed period, which means "cons" in ML), and is also included in the simpset as the tantology "f1 $\vdash$ f1", which is generated by the ML rule of assumption. Not all formulae are suitable as simprules, and one may wish to use a version of DISCHTAC which merely adds the antecedent to the assumption list, leaving the simpset unaltered.

By contrast, simprules are <u>engaged</u> by a standard tactic called SIMPTAC. When applied to a goal (f,S,A), SIMPTAC uses all the simprules in S to rewrite f as often as possible until either no more simprules apply, or a tautologous subgoal is produced (SIMPTAC recognises certain tautologies) in which case an empty subgoal list is returned. In fact, SIMPTAC is the principal means by which a goal may be reduced to an empty subgoal list; when this occurs, all that remains for the user is to apply the generated proof function (whose complex structure he need never see) to the empty theorem list, in order to achieve his original goal as a theorem.

Our proof also calls for TREEINDUCTAC, the tactic which inverts the tree induction rule described in Section 4 above; thus, it takes the form

$$\frac{\text{"!t. } \wp\text{[t]", S, A}}{\begin{array}{l}\text{"}\wp\text{[UU]", S, A} \\ \text{"!I. } \wp\text{[mkTIP I]", S, A} \\ \text{"!op t.(}\wp\text{[t] IMP } \wp\text{[mkUN op t])", S, A} \\ \text{"!op t1 t2. (}\wp\text{[t1] \& } \wp\text{[t2] IMP } \wp\text{[mkBIN op t1 t2])", S, A.}\end{array}}$$

This tactic is derived automatically by the package described in Section 7.

The assembly of four tactics, so far described, would be enough to generate the parser correctness proof were it not for the small bit of reasoning which enables induction hypotheses to be used as conditional simprules. In fact, to establish the antecedents of these hypotheses requires the use of theorems WD3 - WD5, which cannot themselves be used as simprules. There are two distinct reasons why, for example, WD3

```
!op t. wd(mkUN op t)==TT  IMP  wd(t)==TT
```

is unsuitable as a simprule. First, any match to the left side, wd(t), of its consequent fails to determine an instance of the variable op which occurs in the antecedent; thus the simplifier cannot know which instance of the antecedent it should try to reduce to a tautology. Second, even if such an instance is determined, its left side will again match the left side of the consequent of any of WD3 - WD5; thus the attempt to reduce the instantiated antecedent to a tautology will induce an infinite regress in conditional simplification.

How can such lemmas be tactically engaged in a proof? Our solution is to factor their engagement into two parts, introduction and application, each represented by a tactic. First, to <u>introduce</u> them, we design a tactic parameterised on a list of theorems:

USELEMMASTAC[" ⊢ f1"; ... ; " ⊢ fn"] :

$$\frac{\text{f, S, A}}{\text{f, S, ["f1"; ... ;"fn"] @ A}}$$

(In ML, [x1;...;xn] denotes an explicit list and @ concatenates lists.) The proof function returned by USELEMMASTAC simply discharges any of the assumptions "fi" used in achieving the subgoal, and appeals to the lemmas and to the Modens Ponens rule to eliminate them.

Second, to <u>apply</u> such lemmas we design a tactic which, more generally, endeavours to deduce useful facts from <u>any</u> available assumptions. Its elementary action is to "resolve" any suitable pair of assumptions, for example

```
wd(mkUN op' t')==TT
!op t. wd(mkUN op t)==TT  IMP  wd(t)==TT
```

That is, since the first matches the antecedent of an instance of the second, the theorem

. . ⊢ wd(t')==TT

(where the periods stand for the two assumptions) is proved by Modens Ponens.

This theorem is then added to the simpset - it is quite acceptable - and thus allows the appropriate induction hypothesis to be successfully used as a conditional simprule. The formula "wd(t')==TT" is also added to the assumptions, where it may partake in further "resolutions".

We call the tactic RESTAC, because it is a primitive version, in LCF terms, of the classical resolution method [Rob]. In general, RESTAC searches the assumption list for any pair

```
!y1 .. ym. h
!x1 .. xn. (f IMP g)
```

where h is not an implication, and in which h and f are unifiable [8] to produce a common instance f[ti/xi]. Then the theorem

```
.. |— g[ti/xi]
```

is proved, and generalised on all variables not free in the assumptions. If f is a conjunction of form "f' & ..." , then "f IMP g" is treated as "f' IMP(...IMP g)". RESTAC puts this new theorem in the simpset, and its conclusion formula in the assumption list, subject to certain constraints (in particular to avoid adding simprules which are unsuitable, as described above).

The tactics required for our proof, but also of a general nature applicable in many proofs, are now ready; they need only be put together to form a strategy. To do this, one uses combinators which we call <u>tacticals</u> (by analogy with functionals). They may be programmed in ML, and there are a few standard ones. For example, for any tactics T and T':

- The sequencing tactic (T THEN T') applies T to obtain subgoals, and to each subgoal applies T';
- The iterating tactic (REPEAT T) applies T to obtain subgoals, to which T is again applied, repeatedly until T fails to apply;
- The alternating tactic (T ORELSE T') applies T if possible, otherwise T'.

By combining small tactics in this way into sophisticated structures one can generate whole proofs or large parts of proofs by a single tactic (or strategy) application. Much of the interest in LCF lies in the search for useful general strategies. One is assured that, behind the scenes, every necessary inference step is evaluated when the proof function (put together by the tacticals from the simple proof functions for each basic tactic) is at last invoked; but, to the extent that one's strategy is successful in reducing goals to trivial subgoals, one is not made aware of the proof details.

For the parser proof, the main goal includes as simprules all the defining axioms P1-P5 and U1-U3 of the parser and unparser, the theorems A1-A4 concerning APP, and the defining axiom of INTO, together with the non-implicative properties WD1,WD2 of the predicate WD.

The proof is generated by the following strategy, expressed in ML, where L stands for the implicative properties WD3-WD5 of WD:

```
     USELEMMASTAC L
THEN TREEINDUCTAC THEN SIMPTAC
THEN REPEAT (GENTAC ORELSE DISCHTAC)
THEN RESTAC THEN SIMPTAC
```

This strategy in turn

- Adds the properties WD3-WD5 as assumptions, later to be discharged (during the justification, or proof, generated when the strategy is applied);
- Produces the four subgoals of tree induction, then uses simplification in each case, thereby solving the Undefined and Tip cases;
- For the remaining Unary and Binary cases, proves for arbitrary values and repeatedly assumes antecedents;
- Resolves the assumptions introduced by USELEMMASTAC and added by DISCHTAC, producing as new simprules the conditions needed for using induction hypotheses as conditional simprules;
- Engages old and new simprules to solve both remaining goals.

This conceptual division of the necessary steps is rather natural, and corresponds to the order and style of reasoning in the informal proof.

A more compact strategy is adequate for the present proof, if we observe that the only necessary resolutions are between an antecedent of a subgoal formula and a lemma not suitable as a simprule. For then we can combine the function of USELEMMASTAC, DISCHTAC and RESTAC into a tactic, called DRESTAC say, carrying the lemmas L as a parameter.

When applied to an implicative goal it both assumes the antecedent and resolves it with the antecedent of any suitable lemma in L, generating possibly new assumptions and simprules; then it returns the consequent as a subgoal. The new strategy has the compact form

```
          TREEINDUCTAC THEN SIMPTAC
     THEN REPEAT (GENTAC ORELSE DRESTAC L)
     THEN SIMPTAC
```

This strategy does not result in a shorter proof; in fact the proof contains the same inferences but in a different order. In each case the number of basic PPLAMBDA inferences performed is about 800.

This completes our treatment of the parser problem; we hope that it illustrates a useful and natural methodology of proof. But it inevitably raises questions about the generality of the method, some of which we address in the final section. The next section shows how recursively defined data types like LIST and TREE can be automatically axiomatized and equipped with induction rules.

7. DERIVING INDUCTION

In this section we illustrate the use of the structure package, programmed in ML, which automates the axiomatization of a recursively defined data type and provides the associated induction tactic. We detail the interactions needed to do this for the polymorphic trees of which our parse-trees are an instance.

Suppose, then, that we are building the theory TREE. In LCF, one is always either building a theory - which we call drafting a theory - or working in an established theory to prove theorems. Any theory T (draft or established) is represented by two files: T.DFT or T.THY which contains its type operators, constants, and axioms, and T.FCT which contains its theorems. In drafting TREE, we first wish to create the ternary type operator TREE (the name need not be identical to the theory name), so we invoke the ML procedure "newtype" by

```
newtype 3 'TREE' ;;
```

The argument 'TREE' is of a metatype token; tokens are used also as theory names, and to build many other objects.

We now wish to set up sufficient data to allow the structure package to work. Two data items, respectively sty (the structure type) and shape (the constructions of the type) are all that is needed. For the first, we declare in this case

```
let sty = ":(*,**,***)TREE";;
```

The quotation ": .." is the means of mentioning PPLAMBDA types explicitly in ML; here we are merely establishing the use of particular type variables *,** and *** to stand for the argument types.

The shape of the type sty is an expression of the following informal domain description:

> A TREE is <u>either</u> a TIP consisting of a *.
> <u>or</u> a UN consisting of a ** and a TREE,
> <u>or</u> a BIN consisting of a *** and two TREEs.

The shape, in this case, is a list of three pairs; the first element of each pair is a constructor name presented as a token, and the second a PPLAMBDA term consisting of a tuple of variables of appropriate type:

```
let shape =
    ['mkTIP', "I:*";
     'mkUN', "(op:**), (t:↑sty)";
     'mkBIN', "(op:***), (t1:↑sty), (t2:↑sty)"];;
```

the quotation "..." is the means of mentioning PPLAMBDA terms and formulae in ML, and its inverse ↑ (antiquotation) allows appropriately typed meta-terms to appear within quotation. Note that as well as providing constructor names (which the package will use to create PPLAMBDA constants), suitable variables are presented to allow the package to formulate axioms in a form which the user can recognize.

At this point the first part of the structure package, a metaprogram, is evaluated; its effect is to set up appropriate constants and axioms on the file TREE.DFT; in particular, axioms expressing the domain isomorphism stated in Section 5.

Later, when working in the theory TREE, or any of its descendent theories, one only has to evaluate the second metaprogram of the package, whose only effect is to declare a parameterised tactic

```
STRUCTAC: token → token → tactic
```

The first token argument is the name of the theory in which a structure was axiomatized, and the second is the name of its type operator, so to set up the induction tactic we declare

```
let TREEINDUCTAC = STRUCTAC 'TREE' 'TREE' ;;
```

Thus this single command is all we need, when working in the theory PARSE, before performing inductive proofs. Indeed if our problem also required induction on lists then, since LIST is an ancestor theory of PARSE, we could also declare

let LISTINDUCTAC = STRUCTAC 'LIST' 'LIST' ;;

This illustrates the generality of the paremetric tactic STRUCTAC. Furthermore, since the nonrecursive domain SYMB was also set up by the package, the declaration

let SYMBCASESTAC = STRUCTAC 'SYMB' 'SYMB';;

would yield a tactic for case analysis in the symbol domain.

We shall not describe here the class of type definitions which can be handled by the package. It does not handle, for example, domain isomorphisms involving the function type operator $\rightarrow$; but it is open to extension, and at least some function domains can be treated by a natural extension. A simple case which is presently handled is the natural numbers

$$\text{INT} \simeq \text{ONE} + \text{INT}$$

(where the single proper element of the summand ONE stands for zero); one then obtains mathematical induction.

The method by which all these inductions are derived is due to Dana Scott. The derivation of LIST induction is described in Appendix 1 of [GMW], and the example of induction on the structure of programs in a simple imperative programming language is treated in [Mil].

8. CONCLUSION

Since our proof and the present paper were prompted by Gloess' [Glo] treatment of the same problem, using the Boyer-Moore theorem-prover, it is necessary to make some comparison with his proof. First, our motivations were somewhat different. Gloess expressed the aims of (i) formulating his parser and his problem without an eye to ease of proof, and (ii) avoiding modifying the formulation in the course of the exercise. He wished to find out how tractable the Boyer-Moore method is for a newcomer; his success in completing the proof is therefore evidence in favour of that method. We, on the other hand, wished to find as concise and tractable a presentation as we could, for the same problem, since we believe that the formulation of algorithms should be influenced by ease of proof.

Second, the difference between the Boyer-Moore system and LCF causes a striking difference in proof method. This is partly due to a different treatment of proof strategy: a sophisticated intelligent built-in strategy in Boyer-Moore, as opposed to a language for presenting strategies in LCF. More basically, it is due to a difference in the underlying logics. Since the Boyer-Moore system is concerned with total recursive functions, the user must first convince the system of the totality of any particular function (e.g. the parser), and this task represented a considerable proportion of the work for Gloess. In contrast, the interpretation of PPLAMBDA is in domains which are partially ordered with minimum element (undefined); this mathematical framework due to D. Scott was provided precisely to allow expression of general recursive functions, including partial functions. The effect for LCF is two-fold. First, a wide class of induction rules (including structural induction) is derivable from a single induction rule concerning continuous functions over complete partial orders. Second, to state and prove the totality of an algorithm (e.g. the parser in our example) is just part of stating and proving its correctness. In our example, since the domains of words and trees include partially and totally undefined elements, it was necessary to qualify the statement of correctness with the predicate WD[t] expressing the definedness of the parse-tree t. Thus the question of totality also requires careful treatment in LCF, though we believe our proof shows that it is naturally handled. Perhaps more importantly, the wider framework of general recursive functions allows treatment of useful algorithms which may only terminate on a subclass of the well-defined arguments.

Turning to the question of strategies, we must be careful in making any claim that our parser was proved correct "automatically". We are confident of one thing; a strategy with similar structure to the one used here is capable of achieving proofs of a wide variety of theorems, in various domains. The case studies by Leszczylowski [Les 1,2] and Cohn [Coh] provide evidence for this. But the strategy is parametric, and the user must exercise some thought in supplying the parameters. They are of three kinds, (i) <u>What induction is to be done?</u> If the induction is to be on lists say, rather than on trees, then LISTINDUCTAC should replace TREEINDUCTAC. More generally, some questions which were carefully considered by Boyer and Moore in devising their built-in strategy are left by us to the user; in particular, which variable should be the subject of induction, and whether some generalisation of the goal is needed before attempting induction. We believe that these elements of the Boyer-Moore strategy can be naturally incorporated in ML-expressible strategies, and thus easily varied; this is an interesting topic for future work. (ii) What simplification rules are appropriate? This parameter is supplied as a component of the main goal; in the present problem and many others it appears that a large set of simprules can be settled upon without doubt, but that a few need to be considered carefully before inclusion or omission. Often a rule can be admitted or excluded on syntactic grounds, as is indeed done by RESTAC. (iii) What lemmas should be supplied for resolution? Here some problem-specific analysis is needed; further experiment will determine how easy this analysis is. But the way is open to include powerful resolution proof methods, about which much is known; then the inclusion of a large battery of possibly useful lemmas should still not cause embarrassing inefficiency.

At this point we should recall that LCF is meant to be a proof assistant. Although we have focussed attention upon a strategy which happens to be completely successful for a particular problem, in general one may proceed by applying a strategy which is only partly successful, and which leaves some subgoals to be achieved. These can then be tackled by adhoc methods or by applying another strategy. The present simple problem was first proved by one of us (Milner) in just this way; after establishing the supporting theories and formulating the problem a simple strategy without resolution was attempted, and the two subgoals which remained were solved easily by direct inference. This first proof was completed in less than two days; it took somewhat longer to discover that a simple and general strategy incorporating resolution could handle the whole problem.

Our simple strategy certainly requires refinement in order to handle other problems for which induction is the central tool. For example, many forms of case analysis occur again and again in natural mathematical proof, and it is by no means obvious exactly when to engage them. One of us (Cohn) has studied a more sophisticated parser - a precedence parser - and proved an analogous correctness result for it; this work will be presented in a forthcoming paper. The proof follows the same general lines; after establishing a (considerably larger) set of lemmas, the strategy required for the main result is not much more complex than the one used here, but does rely on case analysis - in particular whether the precedence of one operator is less than, equal to, or greater than that of another.

The experience of this more complex proof suggests that in future we may be able to identify several possibly useful tactics, associated with the domains (e.g. the domains of lists, of symbols, of precedences, etc.) involved in a problem, and assemble them in a heuristic strategy which explores different tactical combinations. To make progress in this direction, we must persist in analysing a carefully graded sequence of problems; there appears to be no other approach.

REFERENCES [LNCS stands for Lecture Notes in Computer Science, Springer-Verlag].

[BM] R. Boyer and J. Moore (1979), A Computational Logic, Academic Press Inc., New York.

[Coh] A. Cohn (1981), "The Equivalence of Two Semantic Definitions; a Case Study in LCF", CSR-76-81, Computer Science Department, Edinburgh University.

[Glo] P. Gloess (1980), "An Experiment with the Boyer-Moore Theorem Prover; a Proof of the Correctness of a Simple Parser of Expressions", Proc. 5th Conference on Automated Deduction, LNCS 87, pp. 154-169.

[GMW] M. Gordon, R. Milner and C. Wadsworth (1979), Edinburgh LCF, LNCS 78.

[Les 1] J. Leszczylowski (1979), "An Experiment with Edinburgh LCF", CSR-50-79, Computer Science Department, Edinburgh University.

[Les 2] J. Leszczylowski (1980), "Theory of FP Systems in Edinburgh LCF", CSR-61-80, Computer Science Department, Edinburgh University.

[Mil] R. Milner (1976), "Program Semantics and Mechanized Proof", Foundations of Computer Science II, Part 2, Mathematical Centre Tracts 82, Amsterdam, pp 3-44.

[Rob] J.A. Robinson (1965), "A Machine-oriented Logic based on the Resolution Principle", J.ACM 12, 1.