A Metalanguage for Interactive Proof in LCF*

M. Gordon, R. Milner
University of Edinburgh

L. Morris
Syracuse University

M. Newey
Australian National University

C. Wadsworth
University of Edinburgh

Introduction

LCF (Logic for Computable Functions) is a proof generating system consisting of an interactive programming language ML (MetaLanguage) for conducting proofs in PPλ (Polymorphic Predicate λ-calculus), a deductive calculus suitable for the formalisation of reasoning about recursively defined functions, in particular about the syntax, semantics and implementations of many programming languages. PPλ is an enrichment (in respect of type structure and expressive power) of an extended λ-calculus due to Dana Scott and is fully discussed elsewhere [22] . The purposes of this paper are

(a) to illustrate the features of ML which make it of general interest in language design quite independently of its use for machine assisted formal proof,
(b) to illustrate ML applied to PPλ, in encoding interesting proof-finding-and-performing procedures, and
(c) to convey a methodology for controlled semi-automatic proof.

We avoid formal description; we hope that our examples and discussion will achieve these purposes more clearly. A complete description of ML, and its use with PPλ, exists as a technical report [9].

The implementation (using LISP on a DEC 10 computing system) of ML and PPλ began over three years ago at Edinburgh; for about two years the system has been usable, and its development is now virtually complete. Recently it has been used in various studies concerning formal semantics: theorems about data structures, recursion removal, direct versus continuation semantics, and other topics.

The need for and design of ML is based on experience with an earlier system at Stanford [17, 18] . In that system, beyond the ability to direct it to execute a basic inference (e.g. beta conversion, or transitivity of equivalence), the user could

(a) invoke simplification with respect to a set of equivalences specified by him,
(b) adopt a goal-directed proof style, generating subgoals by built-in tactics based upon the inference rules and simplification, and
(c) use theorems previously proved.

These facilities were enough to enable several non-trivial case studies to be tackled [1,23,24,34] but further use of the system became increasingly limited by the fixed, and rather primitive, nature of its repertoire of commands (rather like using an interactive assembly language - and one without a proper subroutine feature at that! - in which one is working all the time at top-level). Proofs often contained many instances of a few patterns of inference which one would like to express as

*This work was supported by the Science Research Council of Great Britain under grant number B/RG/48175.

derived inference rules or - in the goal-directed style - as derived tactics or strategies.

Our present point of view is that neither a straightforward proof-checker (laborious and repetitive to use) nor an automatic theorem-prover (inefficient because of general search) is satisfactory. What is required is a framework in which a user can both design his own partial proof strategies (where he can find them) and execute single steps of proof (where he needs to). We believe also that, although formal proofs are important and should be retrievable, it is pragmatically more convincing to achieve clear expression of proof strategy; the latter entails that the way in which the strategy is built from sub-strategies should be evident in its expression.

In other words, we're not so concerned with checking or generating proofs as with performing proofs. Thus, we don't normally store or display proofs but only the results of them - i.e. theorems. These form an abstract type on which the only allowed operations are the inference rules of PPλ ; this ensures that a well-typed program cannot perform faulty proofs (it may not prove the theorem expected but the result will be a theorem!). If extra security or formal proof-checking is desired, full proofs are easily generated - only minor changes in the implementation of the abstract type for theorems would be required.

The principal aims then in designing ML were to make it impossible to prove non-theorems yet easy to program strategies for performing proofs. A strategy - or recipe for proof - could be something like "induction on f and g , followed by assuming antecedents and doing case analysis, all interleaved with simplification". This is imprecise - analysis of what cases? - what kind of induction, etc, etc. - but these in turn may well be given by further recipes, still in the same style. The point is that such strategies appear to be built from simpler ones (which we call tactics rather than strategies) by a number of general operations in fairly regular ways; we call these operations tacticals by analogy with functionals.

For programming tactics and tacticals, and more generally for manipulation of PPλ in finding proofs, the following ingredients in ML were soon found to be expedient (almost necessary) : the ability to handle higher order functions, a rigorous but flexible type structure, a mechanism for generating and trapping failures, and an abstract syntactic representation of the object language PPλ .

Acknowledgments

We are indebted to Dana Scott for providing a large part of the theoretical basis for our work: to John McCarthy for encouraging the forerunner of this project at Stanford: to Richard Weyhrauch who contributed greatly to that project: to Tony Hoare and Jerry Schwarz for help with the notion of abstract types in ML : to Avra Cohn for help in the final design stages through her experiments: to Rod Burstall and many colleagues in Edinburgh for illuminating discussions.

Outline of ML.

ML is a higher-order functional programming language in the tradition of ISWIM [15], PAL [8], POP2 [6] and GEDANKEN [26], but differs principally in its handling of failure and, more so, of types. It is an expression-based language, though expressions may have side-effects because of the presence of assignment (the expression "x:=e" has as value the value of e , and also gives x this value). An important expression construct is "let x = e in e'", which binds x to the value of e throughout e' ; alternative forms of declaration are "let f(x,y,...) = e" for defining functions, "letrec f(x,y,...) = e" for defining functions recursively, "letref x = e" for declaring and initializing assignable variables, and generalizations of these forms for simultaneous declarations.

Another important expression construct is "e ? e'" (read "e or else e'"), whose value is the value of e unless e generates a failure, in which case it is the value of e'. The system generates certain failures automatically, and the user may generate his own with the expression "fail", or the expression "failwith e" where the value of e

is a token which identifies the kind of failure; a generalization of the form "e ? e'" can be used to trap only certain failure tokens (kinds of failure). The type token is one of ML's basic types; tokens are just symbol strings. In our current application of ML, the use of failure as a dynamic escape and escape-trapping mechanism facilitates a natural programming style for composing tactics and strategies which are usually inapplicable to certain goals.

As hinted above, if d is a declaration and e an expression, then "d in e" is an expression. In interactive programming (which is how proofs are conducted), one evaluates a mixed sequence of declarations and expressions, separated by ";;" . ML is a "static binding" language, like ISWIM, PAL and GEDANKEN (but unlike LISP and POP2); a free variable z in the declaration "let f(x) = ..." refers to the textually enclosing declaration of z , not to any subsequent declaration.

An example which illustrates most of the features of ML is a generalised scalar product (sum the products of two vectors) which is parameterised on its product and summation functions and on a zero (for null vectors). Two ways - the first recursive and the second iterative - of writing this in ML, with vectors represented as lists and failure for vectors of unequal length, are as follows:

```
letrec scalarprod ($*,$+,zero)(ℓ1,ℓ2)=                    (1,2)
 ( let  x1.ℓ1' = ℓ1 and x2.ℓ2' = ℓ2                        (3)
   in (x1*x2)+scalarprod($*,$+,zero)(ℓ1',ℓ2')
 ) ?                                                       (4)
 (if  null(ℓ1) & null(ℓ2) then zero
                          else fail)

let itscalarprod ($*,$+,zero) (ℓ1,ℓ2) =                   (1)
  letref  acc, ℓ1, ℓ2 = zero, ℓ1, ℓ2
  in
    ( loop  acc, ℓ1, ℓ2 :=                                (5)
        ( (x1*x2)+acc, ℓ1', ℓ2'
          where  x1.ℓ1' = ℓ1 and x2.ℓ2' = ℓ2)
    ) ? (if  null(ℓ1) & null(ℓ2)  then acc
                                  else fail)
```

Notes: (1) "letrec f(x,y,z)(u,v) = - - -" is equivalent to "letrec f = λ(x,y,z).λ(u,v). - - -". (Similarly, let ...). That is, scalarprod and itscalarprod are defined here as (partially) curried functions, as is the style in functional programming. The separation of arguments into two groups allows scalarprod to be "partially applied" to three arguments to obtain particular scalar product functions; it also suggests a more efficient recursive definition in which we replace

```
letrec scalarprod($*,$+,zero)(ℓ1,ℓ2) =
  - - -scalarprod($*,$+,zero)(ℓ1',ℓ2')- -
```

by a form which recurses on only two arguments, namely

```
let scalarprod($*,$+,zero) = scalp
whererec scalp(ℓ1,ℓ2) =
  - - -  scalp(ℓ1',ℓ2') - - -
```

(2) Prefixing $ to a token enables its use as a binary infix without the $.

(3) Infixed "." is the cons function. Use on the left of a declaration, as here, binds x1 and ℓ1' to the head and tail, respectively, of ℓ1, with failure when ℓ1 is null.

(4) The failure trapped by "?" here is that of the declaration when one of ℓ1,ℓ2 is null.

(5) "loop e" repeats e until failure.

The functions scalarprod and itscalarprod make sense on a wide variety of objects. Applications of either are well-typed provided their arguments have types which are instances of

$* : $(\alpha \times \beta) \to \gamma$

$+ : $(\gamma \times \delta) \to \delta$

zero : δ

where $\alpha,\beta,\gamma,\delta$ are type variables, and then the result is a function which has the corresponding instance of (α list $\times$ β list) $\to$ δ as its type. We say that the type

$$((\alpha\times\beta\to\gamma)\times(\gamma\times\delta\to\delta)\times\delta) \to (\alpha \text{ list} \times \beta \text{ list} \to \delta)$$

is generic for scalarprod (or itscalarprod); this means that these functions may be used at any type which is a substitution instance of the generic type, in which $\alpha,\beta,\gamma,\delta$ are type variables. Thus, since $*,$+ : int × int → int are arithmetic functions predeclared in ML, and using the ML notation "[el;...;en]" for lists, we have

```
scalarprod($*,$+,0)([1;2;3],[4;5;6])
          = 1×4 + 2×5 + 3×6
```

using scalarprod at the instance $\alpha=\beta=\gamma=\delta=$ int of its generic type.

To define a function which, given two vectors [bl;...;bn] , [cl;...;cn] of truth values, will count the number of times that both bi and ci are true, we may define

```
let bothtruecount = scalarprod(bothtrue, $+, 0)
   where bothtrue(b1,b2) = if b1 & b2 then 1
                                      else 0
```

using scalarprod at its type instance $\alpha=\beta=$bool and $\gamma=\delta=$ int. We may even define a function of type α list × (α list)list → α list such that

[xl;...;xn],[lisl;...lisn]↦(xl.lisl)@...@(xn.lisn)

using the predeclared append function $@ ; the definition is

```
let mapconsappend = scalarprod ($.,$@,nil)
```

using scalarprod at its type instance $\beta=\gamma=\delta=\alpha$ list. Notice then that both scalarprod and mapconsappend possess a type which contains type variables; they are polymorphic functions. The polymorphism of ML should not be confused with the polymorphism present in the object language PPλ ; we will allude briefly to the latter in a later section.

Such polymorphism with respect to program types is possible, to a greater or lesser extent, in several languages (e.g. PASQUAL [30]) which allow procedures to have explicit type parameters. ML relies instead on a type-checker which not only checks that polymorphic functions are used consistently at instances of their generic type but can, in nearly all practical cases, infer the types of all variables without these being supplied explicitly (e.g. it will infer the type given above for scalarprod and itscalarprod). Thus we come close to the discipline which a good programmer will impose upon himself in using a typeless language such as LISP. It is remarkably convenient in interactive programming to be relieved of the need to specify types, with assurance that badly-typed phrases will be caught, reported, and not evaluated. Of course, for off-line programming it is often advisable to specify the types in declarations - including the types of formal parameters, and we are aware that many prefer to adopt such a discipline for intelligibility and for documentation purposes. To this end, ML always allows the user to specify his types explicitly. If he cares to write

```
letrec  scalarprod ($*: (α×β → γ),
                    $+: (γ×δ → δ),
                    zero: δ)
                   (ℓ1: α list ,
                    ℓ2: β list) : δ = ....
```

he can do so, and the typechecker will check these types for him. It turns out that, in the presence of polymorphism, essentially the same typechecking algorithm is necessary even if type specification is made compulsory (unless indeed the type of every expression were required, which would be intolerable).

The typechecking method may be illustrated by a simple example. Consider the following function for mapping a function over a list:

```
letrec map(f,lis) = if null(lis) then nil
                                 else f(hd(lis)). map(f,tl(lis))
```

The generic type of map should be $(\gamma\to\delta)\times\gamma$ list → δ list. How may we infer this type from the bare declaration? First, the generic types of the identifiers occurring free in the declaration are

```
null : α list → bool
nil  : α list
hd   : α list → α
tl   : α list → α list
$.   : α × α list → α list
```

and we require that every occurrence of such an identifier is given, as type, a substitution instance of its generic type (different occurrences may be assigned different type instances). Second, every occurrence of a formal parameter must be given the same type, and every occurrence

(in its declaration) of a recursively defined identifier must be given the same type. Third, if we denote by σ_{id} the type to be given to each identifier in the declaration, then besides the above-mentioned constraints on σ_{null} etc, the following equations must hold for some types $\tau_1,\tau_2,\ldots$:

$\sigma_{map} = \sigma_f \times \sigma_{lis} \to \tau_1$ $\quad$ $\sigma_f = \tau_2 \to \tau_4$

$\sigma_{null} = \sigma_{lis} \to \underline{bool}$ $\quad$ $\sigma_{map} = \sigma_f \times \tau_3 \to \tau_5$

$\sigma_{hd} = \sigma_{lis} \to \tau_2$ $\quad$ $\sigma_{\$.} = \tau_4 \times \tau_5 \to \tau_6$

$\sigma_{tl} = \sigma_{lis} \to \tau_3$ $\quad$ $\tau_1 = \tau_{nil} = \tau_6$

Each of these equations except the last arises from some sub-expression which is a function application; the last arises because a conditional expression is given the same type as its two arms, and because the definiens and definiendum of a declaration are given the same type.

Now if we choose distinct type variables $\alpha_1,\ldots,\alpha_5$ and set $\sigma_{null} = \alpha_1$ $\underline{list} \to \underline{bool}$, $\sigma_{nil} = \alpha_2$ $\underline{list}$, etc, the equations may be solved for the variables $\alpha_1,\ldots\alpha_5$, $\tau_1,\ldots,\tau_6$ and σ_{map}, σ_f, σ_{lis} using Robinsons unification algorithm [27]. It turns out, as the reader may like to check, that for some distinct pair of variables γ,δ we obtain

$$\sigma_{map} = (\gamma \to \delta) \times \gamma\ \underline{list} \to \delta\ \underline{list}$$

as expected. This then is the generic type of map, which may be instantiated (differently) for each later occurrence of the identifier.

This example does not illustrate all the typing constraints. There are further rules concerning the instantiation of generic types of variables declared within a function body (for example, the type of the variable xl in the declaration of scalarprod is fully determined by the type of the formal parameter ℓ1, and the rules will demand in this case that xl is given the same type at each occurrence), and concerning the types of variables declared by letref (in particular, like formal parameters, they must be given the same type at each occurrence). For the complete algorithm, and a proof of its semantic correctness for a simpler language, see [21] ; the algorithm is in fact rather straightforward.

ML also includes a facility for defining abstract types, including simultaneous and/or recursive and/or parametric ones. In ML these are not "really abstract" in the sense of the algebraic abstract types of, e.g., Guttag [10] or Zilles [36] but rather are analogous to SIMULA classes [7], CLU clusters [16], and ALPHARD forms [35]. As the latter are by now well-known, it will be enough to describe briefly our syntax for abstract type declarations and give a simple example. The declaration form is

abstype <tyargs><id>=<ty> and...and <tyargs><id>=<ty>
with ...

where the identifiers <id> are the new (parameterized) types being declared, each <tyargs> is a sequence (possibly empty) of type variables - the formal parameters - and each <ty> is a type expression. The part "with ..." has the syntax of a normal declaration (but with let replaced by with), and defines the operations or other objects available at the new types; the essence of type abstraction is that one may get at the representation of the new types only in the with-part, and this representation is provided by two halves of the isomorphism (denoted by "absid" and "repid" for each type identifier "id") between each type and its representation. For (mutually) recursive types one must use absrectype in place of abstype. We give, as an example, the redefinition of the ML type operator list for lists; note that the isomorphism functions are polymorphic - they are

abslist : (. + (α × α list)) → α list
replist : α list → (. + (α × α list))

where the basic type "." is that with just one element denoted by the expression (), and "+" between types is disjoint sum. The declaration is:

```
absrectype α list = . + (α × α list)
      with  nil = abslist(inl())
       and  $.(x,ℓ) = abslist(inr(x,ℓ))
       and  null(ℓ) = isl(replist(ℓ))
       and  hd(ℓ) = fst(outr(replist(ℓ)))
       and  tl(ℓ) = snd(outr(replist(ℓ)))
```

The polymorphic functions inl and inr, outl and outr, and isl and isr are left and right injections, projections (with failure for arguments in wrong summands) and predicates for disjoint sum types.

(Note: we have underlined types and reserved words in this paper, for clarity, but our implementation requires no underlining; we have now abandoned it for newly declared types).

Functional types are allowed in abstract type declarations, and this yields some interesting possibilities. A simple example is streams - a notion of infinite implicit lists due to Landin. Here is a definition which provides two stream operations; one for splitting a stream into its first member and remainder, and one for building a stream from a function of the natural numbers:

```
absrectype α stream = . → (α × α stream)
 with next(s : α stream) = repstream(s)()
 and  streamof = str : (int → α) → α stream
      whererec str(f) =
        absstream(λ().(f(1),str(λx.f(x+1))))
```

As an aside, we can show that the recursiveness of types also gives us the power of normal recursion, so that in the presence of absrectype, the letrec construct is theoretically redundant! In fact, a fixed-point function

FIX : ((β → γ) → (β → γ)) → (β → γ)

can be defined so that "let f = FIX(λf.e)" is exactly equivalent to "letrec f = e" ; the reader may like to puzzle out how the following does the trick:

```
absrectype α fixty = α fixty → α
 with FIX(f) = F(absfixty F)
          where F y = f(λx.repfixty(y)(y)(x))
```

PPλ in ML

PPλ is discussed in ML via pre-defined abstract types, one for each of its principal syntactic classes. This method could be adopted for the discussion of any syntactic system within ML, but we have also built in the special ability to discuss PPλ in terms of a concrete representation of its syntax. This is a necessary convenience; to provide concrete syntax for other syntactic systems the user would need to write, in ML, a parser and an "unparser" to map from concrete to abstract syntax and vice-versa.

The formulae of PPλ are those of a first-order predicate calculus built by conjunction, implication and universal quantification from atomic ones; atomic formulae are equivalences and inequivalences (i.e. partial ordering ⊑) between terms of a typed λ-calculus with a fixed-point operator, a conditional operator and other constants. Many of these constants - including the two mentioned - are polymorphic; as with ML the polymorphism involves the use of type variables, and type instantiation is one of the inference rules of PPλ . Thus PPλ is represented in ML by the three abstract types form, term and type (objects of type type are syntactic - they are PPλ type expressions) ; primitive operations provided at these (ML) types are constructors and destructors. Examples of constructors are:

```
mkvar     : token × type → term
                (a variable consists of a token
                 with a type)

mkcomb    : term × term → term
                (a combination, or function
                 application)

mkinequiv : term × term → form
                (to build an atomic formula)

mkquant   : term × form → form
                (the term must be a variable)
```

and to each constructor corresponds a destructor (destvar, destcomb etc) of inverse functional type. Destructors fail if their argument is not a term or form of the right sort - e.g. destvar(mkcomb(..)) will fail. Concrete syntax is provided via quotation ⌜...⌝ ; this syntax is what one would expect, and allows types to be mentioned explicitly, although the system will often deduce types using a method similar to that in ML. Here then are two equivalent ML expressions of type form , assuming that the user has introduced "integer" as a PPλ type (see the later section on Theories):

```
⌜∀X:integer. X ⊑ X⌝
 let x = mkvar(`X`,mkconsttype`integer`)
       in mkquant(x,mkinequiv(x,x))
```

Further, a device which we call antiquotation ⌞...⌟ allows ML expressions (of appropriate ML type!) to appear within quotations, so that the following is also equivalent to the above:

```
let x = ⌜X:integer⌝ in ⌜∀⌞x⌟.⌞mkinequiv(x,x)⌟⌝
```

Now a sequent of PPλ is an object (Γ,w) of type form list × form. PPλ is a sequent calculus, so a PPλ theorem is a sequent which follows from the axioms by the inference rules. A theorem is represented as "Γ ⊢ w" on output; theorems of course may not be input, but only deduced, and for this purpose the axioms and inference rules are provided as primitive objects at the abstract type thm . We give part of the definition of this type; the types of the inference rules mentioned are ASSUME: form → thm, GEN: term → (thm → thm) , TRANS: thm × thm → thm , BETACONV: term → thm :

```
abstype  thm = form list × form
with  ASSUME w = absthm([w],w)          (Infer w ⊢ w)
and  GEN x th =
   let Γ,w = repthm th in               (From Γ ⊢ w
   if ¬ isvar x or x ε freevars(Γ)       infer Γ ⊢ ∀X.w
      then  failwith `GEN`               when X is not
      else  absthm (Γ,mkquant(x,w))      free in Γ)

and  TRANS(th1,th2) = . . .             (transitivity
                                         of equivalence
                                         and inequival-
                                         ence)
and  BETACONV  t    = . . .             ( β reduction)

and  . . . .

. . . .

and  destthm = repthm
```

Notice that repthm is provided to the user (under the name destthm) to allow him to analyse his theorems syntactically, but he is deprived of absthm - and thus assured that all objects of type thm are indeed theorems, since he can only make (prove!) them with the inference rules.

The PPλ calculus was discussed in detail in [22]; our present implementation provides essentially that calculus, but for convenience - and some efficiency - many of the axioms are presented instead as inference rules, of which there are about thirty (more than strictly necessary, again for convenience).

Goals tactics and tacticals

As a simple example to illustrate our methodology, consider an obvious fact about conditionals (for "T => X|Y" read "if T then X else Y"), namely

```
⊢ ∀T1 T2 X Y Z W.(T1 => (T2 => X|Y)|(T2 => Z|W) ≡
                  T2 => (T1 => X|Z)|(T1 => Y|W))
```

The natural way one would prove this informally is: strip off the quantifiers ("Consider any T1,T2,...,W"), then do case analysis on any truth-valued term in sight - and do any simplifications that are possible. So superficially, as a tactic for this (and many other similar goals) we would like to write

```
(REPEAT GENTAC) THEN REPEAT(ANYCASESTAC THEN SIMPTAC).
```

As a first approximation, a tactic should take as argument a goal and produce as result a list of subgoals. We shall here assume that a goal is a sequent, that is

```
goal = form list × form
```

though we in fact use a slight ramification of this type. The idea is that, by repeated subgoaling (i.e. tactic application) we shall reach subgoals which may be achieved by theorems, until no subgoals are left; a theorem Γ' ⊢ w' is said to achieve the goal (Γ,w) if (up to α-conversion) w' = w and Γ' ⊆ Γ - the formulae of Γ in the goal are to be thought of as assumptions, some or all of which may be used in proving w.

But now we can see a deficiency in our first approximation to a tactic. Suppose that a tactic T, applied to goal g , has generated the subgoal list $[g_1;\ldots;g_n]$ and that somehow theorems th_i achieving g_i $(1 \le i \le n)$ have been found. Who is to deduce from $[th_1;\ldots;th_n]$ a theorem th achieving g ? Our answer is that it is the job of T to provide a way of performing this deduction; to this end we define

```
proof  =  thm list → thm
tactic =  goal → (goal list × proof)
```

and we call the proof component of a tactic's result a validation. When a composite tactic has somehow generated an empty goal list, the validations of the components can be composed to yield a theorem, and this composite validation (a function) can be generated automatically as part of the business of composing tactics.

However, not all tactics will be very useful. We shall call the useful ones valid (related but not identical with logicians' use of the word): T is a valid tactic if, whenever $T(g) = ([g_1;\ldots;g_n],p)$ and whenever th_i achieves g_i $(1 \le i \le n)$, then $p[th_1;\ldots;th_n]$ evaluates successfully to a theorem which achieves g. In particular, when $n = 0$ - i.e. T reduces g to an empty subgoal list - then p[] achieves g and we say that T solves g (e.g. the simplification tactic manages this when the goal simplifies to an obvious tautology).

To illustrate, here is a tactic for quantifier stripping (yielding one subgoal) which is "inverse" to the basic inference rule GEN, where we write in some types explicitly as an aid to the reader:

```
let GENTAC ( (Γ,w): goal) =
    let x,wl = destquant w ? failwith `GENTAC`
    in
    let x' = variant (x,freevars(w.Γ))
    in
      [(Γ,substinform(x',x)wl)]: goal list ,
       (GEN x'  ∘  hd) : proof
```

where ∘ is function composition, and variant(x,vars) primes x, if necessary, to obtain a variable not in the list vars. The call of variant in GENTAC is needed to ensure that it is a valid tactic, in this case to prevent possible variable clashes causing the validation function to fail unexpectedly (when applied to a singleton theorem list [th] where th achieves the one subgoal produced by GENTAC). Note that at worst an invalid tactic can fail or prove the wrong theorem; it can never produce a "false theorem" !

Other elementary tactics are also easily defined, e.g. for case analysis on a truth-valued term (producing three subgoals - the true, false and undefined cases), and for computation induction (producing two subgoals - the induction basis and the induction step).

We are mainly interested in valid tactics, though one may demand even more of a tactic. We say that a (valid) tactic is strongly valid if, whenever the argument goal is achievable (by some theorem) so is each of the generated subgoals. One may not always be able to use strongly valid tactics. Consider for example the heuristic tactic for proving $\forall n.f(n) < g(n)$ by finding some function h for which $\forall n.f(n) < h(n) \le g(n)$; in particular, we find that $\forall n.\ n < 2n + 1$ is true (and achievable !) over the non-negative integers, but $\forall n.\ n < 2n \le 2n+1$ is false (not achievable). More relevant is that various induction rules, used in reverse, yield tactics which are not strongly valid. It is good practice to use strongly valid tactics when possible, and always to use valid tactics.

Tacticals are functions on tactics, building simple tactics into composite ones. Three obvious examples are: binary tacticals THEN (apply a second tactic to all subgoals produced by a first) and ORELSE (try one tactic, or if it fails try another), and a unary one REPEAT (iterate a tactic until failure). Defining these, and many others, in ML is a straightforward exercise in functional programming with lists. Moreover, it is easy to show that THEN, ORELSE and REPEAT preserve the validity - even strong validity - of tactics.

(It is worth noting that this tactics-and tacticals style could be adopted in general for problem solving; all that is involved is a type goal, a type for proposed solutions - which might be called shot - and an achievement relation between shots and goals.)

Isolating useful tactics and tacticals, and encoding them in ML, is a subject of ongoing study in the various applications to do with formal semantics which we mentioned in the introduction.

Theories

In our discussion of PPλ we did not suggest the variety of its application; in fact, just as first-order predicate calculus (or any pure calculus) may be extended to an applied calculus by the introduction of non-logical constants and non-logical axioms, so may PPλ be extended. An important part of this extension is the introduction of new types (the only non-trivial primitive type available is that of truth-values). New types may be either introduced independently or defined - perhaps recursively - in terms of existing types. A set of types, together with some new constants

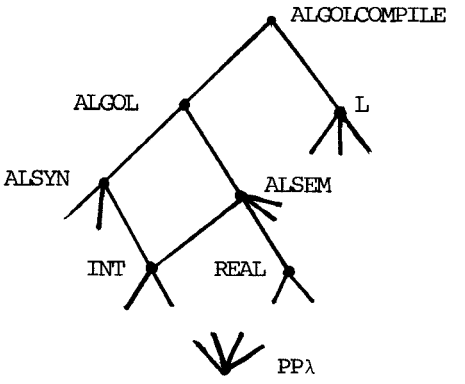
and axioms, is called a theory, and all work with LCF consists in setting up theories, extending them or joining them to form larger theories, or (most important) adding new useful theorems to the list of those proved in an existing theory. Theories are preserved permanently on files, to allow incremental working; for each theory T there is a fixed file T.THY with the types, constants and axioms, and a growing file T.FCT of useful theorems(facts).

Suppose for example one wishes to prove the correctness of a compiling algorithm from ALGOL to some target language L . One will develop first a theory of ALGOL and a theory of L (we give the theories these names), then join them and extend the result by adding a constant "ALcompile" of type "ALprog → Lprog" say, and an axiom defining this compiling function. The resulting theory might be called ALGOLCOMPILE; its parent theories are ALGOL and L , and one of its theorems will assert the compiler's correctness.

But ALGOL itself will be a composite theory. One of its parent theories will be ALSYN - the theory of ALGOL syntax; the types in this theory will be such as ALprog, ALblock, ALstatmt, ALexpn etc, the constants will be the abstract syntactic operations such as
mkassignment : ALvar → ALexpn → ALstatmt, and the axioms will characterise these operations. Further constants and axioms will be concerned with auxiliary syntactic operations - e.g. a predicate for determining whether an identifier occurs free (i.e. non-local) in a block - and the theorems of ALSYN will be about these purely syntactic matters. ALSYN may have a parent INT - the theory of integers - if for example one of the syntactic operations counts the number of free occurrences of an identifiers in a block.

Another parent theory of ALGOL will be ALSEM; this is the theory of domains used to specify the semantics of ALGOL, and may in turn have parent theories INT, REAL, etc corresponding to various data types. ALGOL itself will be the join of ALSYN and ALSEM extended by the definition of the semantic function - call it "ALmeaning" - whose type will be perhaps "ALprog → (ALstate → ALstate)" where "ALstate" is the type of ALGOL machine states introduced in ALSEM. Theorems of ALGOL will have nothing to do with compilation; for example, there are many interesting results to be proved concerning meaning-preserving transformations of programs.

So far, we have outlined an ancestry graph for the theory ALGOLCOMPILE; bearing in mind that the basic theory PPλ is an ancestor of every theory, the graph looks like this



and of course the subgraph for L (in particular) has not been discussed. We refer to [9] for the details of LCF theories; here we will conclude by remarking that it appears impossible to exploit the full power of an interactive proof system without some disciplined framework, such as theories provide, within which to work incrementally.

Relations with other systems

There are several dimensions along which LCF can be compared with other proof systems;

Checking vs. Proving

At one extreme a system just accepts complete proofs and then simply checks their correctness; a sophisticated example of this is the AUTOMATH system of de Bruijn et al [5] . At the other extreme goals are submitted and an attempt is made to automatically achieve them; examples are early resolution theorem proving [27] and the work on mechanising structural induction [2,3] . Between these is a continuum. One can reduce the tedium of using a pure checker by increasing the 'gap' between proof steps (these gaps being bridged by e.g. a simple theorem prover); for example the Stanford LCF system [18] did some simplifications automatically but otherwise the proof had to be provided by the user. Conversely a pure theorem prover can be made more flexible by allowing a user to provide information (perhaps interactively)

to guide the search for a proof (e.g. this is one of the aims of the GOLUX project [11]). Our aim is to construct a system which can be used at any point on this continuum - for some problem areas there already exists useful proof strategies (e.g. Aubin [2], Boyer & Moore [3] for induction on Lists, Brown [4] for integer arithmetic) and we would like to be able to code them up straightforwardly. In other less explored areas we want to experiment with manual proofs to isolate common patterns of inference. Once these are found they can be programmed as ML tactics.

Security

In systems based on general problem solving languages like MICROPLANNER [29] or QLISP [25] there is a danger that in performing a proof wrong inferences may be done. This danger is greatest for systems which are not based on any explicit logic (e.g. [3,32]) ; for these it is not even always clear what the valid inferences are. However even when an explicit logic is used (e.g. Von Henke and Luckham [12] which is based on Hoare's inference system [14]) there may still be a risk that invalid manipulations of theorems might accidently occur - this is especially so if inexperienced users are allowed to program strategies. In LCF we give the user the freedom to write his own tactics (in ML) but the type-checker ensures that these cannot perform faulty proofs - at worst a tactic can lead to an unwanted theorem (for example which does not achieve the desired goal).

Generality

A number of program-proving systems are tailored to particular programming languages thereby enabling efficient special purpose heuristics to be used (e.g. [12,31]). Such systems are good for reasoning about algorithms encoded in their particular language but cannot perform proofs of theorems connecting different languages - e.g. proofs of compilers. We have tried to get the best of both worlds - special purpose heuristics and generality - by specializing our system not to any programming language (e.g. ALGOL, PASCAL etc.) but to the deductive system PPλ , and then providing facilities to enable various particular languages to be axiomatized as PPλ theories; efficient special purpose tactics can then be programmed in ML for these theories. Note however that PPλ itself is oriented towards reasoning about universes of recursively defined objects of various types (viz. Domains of Scotts Theory of Computation [28]) and so reasoning about other objects may be indirect. The Stanford FOL system of Wehyrauch [33] is based on a more general logic, and it remains to be established whether this extra generality is needed (e.g. for reasoning about applications of programs to the 'real world'); there is a delicate trade off between generality and specialization - PPλ is just general enough to handle reasoning about the syntax, semantics and implementations of programs, but is fairly specialized to these.

REFERENCES

[1] Aiello, L., Aiello, M. & Wehyrauch, R., The semantics of PASCAL in LCF, AI Memo 221 Computer Science Dept., Stanford, 1974.

[2] Aubin, R., Mechanising structural induction, Ph.D. thesis, University of Edinburgh, 1976.

[3] Boyer, R.S. & Moore, J.S., Proving theorems about LISP function, JACM 22,1 (Jan.1975), 129-144.

[4] Brown, F.M., A deductive System for Elementary Arithmetic. AISB Summer Conference, Edinburgh, 1976.

[5] de Bruijn, N.G., AUTOMATH, a language for mathematics, T.H.- Report 68-WSK-05, Dept. of Mathematics, Technological University, Eindhoven, Netherlands 1968.

[6] Burstall, R.M. & Popplestone, R., POP2 reference manual, in Machine Intelligence 2, eds. E. Dale & D. Michie, American Elsevier, New York, 1968, 205-246.

[7] Dahl, O.-J. et al, The SIMULA 67 Common base language, Norwegian Computing Centre, Oslo, 1968.

[8] Evans, A., PAL: a language designed for teaching programming linguistics, Proc. ACM 23rd Nat. Conf., 1968, Brandin Systems Press, Princeton, N.J., 395-403.

[9] Gordon, M., Milner, R. & Wadsworth, C. Edinburgh LCF, Department of Computer Science Internal Report CSR-11-77, University of Edinburgh, 1977.

[10] Guttag, J.V., The specification and application to programming of abstract data types, Ph.D. thesis, University of Toronto, 1975.

[11] Hayes, P.J., The language GOLUX. University of Essex, 1974.

[12] von Henke, F.N.& Luckham, D.C., A methodology for verifying programs, Proceedings of the International Conference on Reliable Software, Los Angeles, California 1975.

[13] Hewitt, C., PLANNER: a language for manipulating models and proving theorems in a robot, AI Memo 168, Project MAC, M.I.T., 1970.

[14] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM Vol.12, No.10, 1969.

[15] Landin, P.J., The next 700 programming languages, Comm. ACM 9,3 (March 1966), 157-166.

[16] Liskov, B.H. & Zilles, S., Programming with abstract data types, Proc. of a Symposium on Very High-Level Languages, SIGPLAN Notices 9,4 (April 1974), 50-59.

[17] Milner, R., Implementation and application of Scott's logic for computable functions, Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices 7,1 (Jan.1972), 1-6.

[18] Milner, R., Logic for computable functions: description of a machine implementation, AI Memo 169, Computer Science Department, Stanford, 1972.

[19] Milner, R., Program semantics and mechanised proof, Proc. 2nd Advanced Course in Foundations of Computer Science, Mathematical Centre, Amsterdam, 1976.

[20] Milner, R., LCF: a methodology for performing rigorous proofs about programs, Proc. 1st IBM Symposium on Mathematical Foundations of Computer Science, Amagi, Japan, 1976.

[21] Milner, R., A Theory of Type Polymorphism in Programming, Department of Computer Science Internal Report CSR-9-77, University of Edinburgh, 1977.

[22] Milner, R., Morris, F.L. & Newey, M., A logic for computable functions with reflexive and polymorphic types, Proc. Conf. on Proving and Improving Programs, Arc-et-Senans, 1975.

[23] Milner, R. & Weyhrauch, R., Proving compiler correctness in a mechanised logic, in Machine Intelligence 7, ed. D. Michie, Edinburgh University Press, 1972.

[24] Newey. M., Formal semantics of LISP with applications to program correctness, Ph.D. thesis, Stanford, 1975.

[25] Reboh, R., Sacerdoti, E., A Preliminary QLISP Manual, Technical note 81, Artificial Intelligence Centre, SRI., Mento Park, California 1973.

[26] Reynolds, J.C., GEDANKEN: a simple typeless language based on the principle of completeness and the reference concept, Comm. ACM 13,5 (May 1970), 308-319.

[27] Robinson, J.A., A machine-oriented logic based on the resolution principle, JACM 12.1 (Jan. 1965) 23-41.

[28] Scott, D.S. & Strachey, C., Toward a Mathematical Semantics for Computer Languages, Proceedings of the Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Vol.21, Polytechnic Institute of Brooklyn, 1972.

[29] Sussman, G., Winograd, T. & Charniak, E. Microplanner Reference Manual, AI Memo 203, Project MAC, M.I.T. 1970.

[30] Tennent, R.D., PASQUAL: a proposed generalisation of PASCAL, Tech. Report 75-32, Queens University, Kingston, Ontario, 1975.

[31] Topor, R. Interactive Program Verification using Virtual Programs, Ph.D. Thesis, Edinburgh, 1975.

[32] Waldinger, R.J. & Levitt, C., Reasoning about Programs, Artificial Intelligence, Vol.5 No.3.

[33] Weyhrauch, R.W. A User's Manual for FOL, Stanford Artificial Intelligence memo AIM 235.1, 1977.

[34] Weyhrauch, R. & Milner, R., Program semantics and correctness in a mechanised logic, Proc. USA - Japan Computer Conference, Tokyo, 1972.

[35] Wulf, R.A., London, R.L. & Shaw, M., Abstraction and verification in ALPHARD: introduction to language and methodology, Carnegie-Mellon University, 1976.

[36] Zilles, S., Algebraic specification of data types, Computation Structures Group Memo 119, M.I.T., 1974.