Phil. Trans. R. Soc. Lond. A **312**, 411–422 (1984) [411]
Printed in Great Britain

The use of machines to assist in rigorous proof

By R. Milner

Computer Science Department, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K.

A methodology for computer assisted proof is presented with an example. A central ingredient in the method is the presentation of tactics (or strategies) in an algorithmic metalanguage. Further, the same language is also used to express combinators, by which simple elementary tactics – which often correspond to the inference rules of the logic employed – are combined into more complex tactics, which may even be strategies complete for a class of problems. However, the emphasis is not upon completeness but upon providing a metalogical framework within which a user may express his insight into proof methods and may delegate routine (but error-prone) work to the computer. This method of tactic composition is presented at the start of the paper in the form of an elementary theory of goal-seeking. A second ingredient of the methodology is the stratification of machine-assisted proof by an ancestry graph of applied theories, and the example illustrates this stratification. In the final section, some recent developments and applications of the method are cited.

1. A theory of goal-seeking

The search for a proof of a conjecture expressed as a formula in some formal language is strikingly similar to many goal-seeking activities. These activitives are as widely different as seeking to win at chess and seeking to meet a friend before noon on Saturday. But the similarity can be articulated in terms of a little theory of goal-seeking; a theory that has nothing to do with finding the best strategy, or with minimizing the prospect of failure (important though these things are), but which tries to make precise how concepts like goal, strategy, achievement, failure, etc. relate logically to each other. Before applying this theory to the business of machine-assisted proof, we shall exhibit some of its generality by means of an every-day example.

We may discern two prime entity classes in any sphere of goal-seeking activity: the goals and the events. A *goal* may sometimes be thought of as a description that may be satisfied by one or more (or no) occurrences, and an *event* is simply a particular occurrence. For example:

a goal, G_1: A and B to meet before noon on Saturday;
an event, E_1: A meets B under the clock at Waterloo Station at 11h53 on Saturday.

It is clear that event E_1 satisfies the description that has been designated as goal G_1, i.e. it achieves goal G_1. In general, whatever the sphere of activity, we must postulate or define a relation of *achievement* between events and goals:

$$\text{achieves} \subseteq \text{event} \times \text{goal}$$

(where we use the nouns event, goal in the singular as type symbols, standing for the classes of all possible events and all possible goals).

Now, in planning how to achieve G_1, we might justifiably replace it by two rather more specific subgoals:

G_{11}: A to arrive under the clock at Waterloo Station before noon on Saturday;
G_{12}: B to arrive under the clock at Waterloo Station before noon on Saturday.

In fact we can isolate the method by which G_{11} and G_{12} are gained from G_1, and call it a *tactic*. In this case we might call it the clock-at-Waterloo-Station tactic, and it could be applied to many different goals (differing from G_1 in time, date and persons, for example) to yield in each case a different pair of subgoals. In general then, we may express tactics as partial functions from goals to lists of goals:

$$\text{tactic} = \text{goal} \rightharpoonup \text{goal list}$$

and we say that a tactic *fails* upon, or is inapplicable to, a goal outside its domain. In our example, the clock-at-Waterloo-Station tactic would fail on the goal 'A must never meet B again', but may not fail on 'Ronald Reagan to meet Napoleon Bonaparte before noon on Saturday' (though some later tactic, applied to a refined subgoal, will fail in the attempt to get a dead man to move).

It is important to note that a tactic may be invalid. In our example, a variant of the chosen tactic would be invalid which, when applied to G_1, yielded variants of G_{11} and G_{12} in which 'noon' was replaced by '13h00'. We now make precise the property of *validity* of tactics; to do so we need a new entity class. For each sphere of activity, we postulate a collection of *procedures*; each procedure represents how, given a list of events, some new event may be realized; that is:

$$\text{procedure} \subseteq \text{event list} \rightharpoonup \text{event}$$

(procedures are partial functions; they too may fail). Now we can see the clock-at-Waterloo-Station tactic is justified just because we are assuming the existence of a waiting procedure,

W: A and B each waits until he sees the other.

For W, when applied to the following pair of events that achieve G_{11} and G_{12} respectively,

E_{11}: A arrives under the clock at Waterloo Station at 11h47 on Saturday;
E_{12}: B arrives under the clock at Waterloo Station at 11h53 on Saturday;

will yield an event that achieves G_1, namely the event E_1 given above.

In fact, we can see that the procedure W is stronger still; it has the property that it will, from *any* pair of events that achieve G_{11} and G_{12} (differing perhaps in time from E_{11}, E_{12}), produce an event that achieves G_1. This is what truly justifies the clock-at-Waterloo-Station tactic when applied to G_1.

But we can conceive a tactic whose application to one goal G_1 may be justifiable, but whose application to another goal G_1' (even though it succeeds in producing some subgoals, G_{11}' and G_{12}' say) is unjustifiable because there is no justifying procedure that, given achievements (achieving events) of G_{11}' and G_{12}', will always produce an achievement of G_1'. Such dishonest tactics – ones that promise more than can be performed – are to be avoided; an honest tactic will be called *valid*, and to define validity we first wish to refine the notion of tactic.

A sensible tactic, when it resolves a goal G into subgoals, should make explicit a procedure

(which may depend upon G) that it claims will always lead from achievements of the respective subgoals to an achievement of G. We therefore redefine

$$\text{tactic} = \text{goal} \rightharpoonup \text{goal list} \times \text{procedure}$$

and now we may define a tactic to be valid just when its claim, in the present sense, is always justified.

Definition. A tactic T is *valid* if, whenever

$$T(G) = [G_1; \ldots; G_n], P$$

is defined for a goal G, and whenever events $[E_1; \ldots; E_n]$ respectively achieve the goals $[G_1; \ldots; G_n]$, then the event $P[E_1; \ldots; E_n]$ in turn achieves G.

Now it is essential for the achievement of distant or difficult goals that simple tactics be composed into more complex ones, which are objects of the same type but may be called *strategies*. Even in our example, if A starts from Andover and B from Birmingham then the complete strategy will involve many component tactics. (The procedures that validate these will be, for example, 'B walks from the Underground station to the clock at Waterloo'.) We shall illustrate below, in the context of machine-assisted proof, how tactics can be composed in distinct ways, each represented by a certain combinator; we call such combinators *tacticals*, and we say that a tactical is valid just when it preserves validity of tactics. Of course, where possible only valid tacticals should be used.

Let us suppose that a strategy S for our toy problem has been composed in this way from valid tactics, and is therefore itself valid. In the best case, we have

$$S(G_0) = [\,], P_0,$$

which is to say that no subgoals remain to be achieved (we used [ ] to mean the empty list). P_0 is then a plan for achieving G_0, and need only be executed, i.e. it is applied to the empty list of events, giving the event

$$P_0[\,],$$

which will achieve G_0. In the toy problem, this execution consists of real travel (and other action) by A and B; in the context of machine-assisted proof, it consists of the performance of a formal proof. Here, it is a matter of taste whether the human prover wishes to see this performance done by the machine, in all its frequently repulsive detail, or wishes only to see the highlights, or is merely content to let the machine announce the result (a theorem!). We indicate in the sequel that the last alternative is often appropriate, since the human user has often exercised (or gained) his insight into the problem in the process of composing his strategy. During this composition he may well have interacted with the machine, in applying partial strategies and then deciding whether or not he can make progress from the subgoals generated at each stage.

2. The use of a computer to make proofs

The formal logic of LCF (Gordon *et al.* 1979) is a blend of the predicate calculus (with equality) and the lambda calculus. It was based upon Dana Scott's theory of domains of continuous functions, and is particularly suited to the formulation and proof of properties of algorithms and algorithmic languages. We need not be more precise here about its formulae,

since we are mainly concerned with aspects of proof methodology that apply to arbitrary logics. For the present it is enough to state that its sentences are sequents (Γ,F), where F is a formula and Γ a finite set (or list) of formulae, and its theorems (proved sequents) are written $\Gamma \vdash F$.

Within the goal-seeking jargon of the previous section we have the following interpretations

(1) A *goal* is a sequent (Γ,F). (Prove F from assumptions Γ.)

(2) An *event* is a theorem $\Delta \vdash G$.

(3) An event $\Delta \vdash G$ *achieves* a goal (Γ,F) if for some subset Γ' of Γ the sequents (Δ,G) and (Γ',F) are identical, up to renaming of bound variables.

(4) A *procedure* is therefore a partial function that takes a list of theorems and yields a theorem. The primitive (given) procedures are just the inference rules of the logic; further, derived procedures (derived inference rules, or proof procedures) are derivable naturally by composition, controlled algorithmically.

To manipulate these entities, and others relating to them, an algorithmic metalanguage called ML is used. It is a general purpose functional programming language, whose power of handling higher-order functions is particularly important; this is because tactics (as explained earlier) yield proof procedures, which are functions, as results, and even more so because tacticals are functions over tactics.

Basically, the elementary tactics are just 'inverses' of the given inference rules. The rule of universal generalization provides a good example. The rule

$$\text{GEN}\quad \frac{\Gamma \vdash F}{\Delta \vdash \forall x\,.\,F}\quad (x \text{ not free in } \Gamma)$$

is provided in ML by the function GEN, such that

$$\text{GEN } \text{“x”}\colon (\Gamma \vdash F) \mapsto (\Gamma \vdash \forall x\,.\,F).$$

Note that GEN takes an extra parameter, which is a variable of the object language. Object language constructions are quoted within the metalanguage, and we shall here use lower case letters for object language variables, and upper case for metavariables over object language constructs. The use of antiquotation ↑ allows metavariables to occur within quotation, so that if X = “x” and F = “x = y”, then we may write “∀↑X . ↑F” in the metalanguage to mean “∀x . x = y”. Now the tactic GENTAC, which inverts GEN, must take any goal $(\Gamma,$“∀↑X . ↑F”$)$ and return the subgoal (Γ,F), together with the procedure GEN X, which justifies the application of the tactic. In fact more care is needed, since the rule GEN fails when X is free in Γ; the following declaration of the tactic takes care of this, and is almost exactly as written in ML:

```
val GENTAC (Γ,"∀↑X . ↑F") =
  let val X' = variant X Γ      {X' is a variable not free in Γ}
    val F' = subst [X',X]F      {replace X by X' in F}
  in [(Γ,F')], GEN X'           {subgoal list and justification)
  end
```

(the exact ML definition takes care to require the input formula to be of the form “∀x. ...” and to fail appropriately otherwise).

At a slightly higher level, we already need tacticals to build simple composite tactics that

a mathematician would apply without thinking. A good example is a tactic that repeatedly strips off leading universal quantifiers, and 'assumes' the antecedent of any implication in the goal formula, so that a goal like

$$(\Gamma, \text{“}\forall x . \uparrow F_1 \supset \forall y . \forall z . \uparrow F_2 \supset \uparrow F_3\text{”})$$

is transformed (assuming that x, y and z only occur bound) to the goal

$$(\Gamma \cup \{F_1, F_2\}, F_3).$$

If, besides GENTAC, we have a primitive tactic DISCHTAC, which assumes a single antecedent if there is one (i.e. it inverts the rule of assumption-discharge), then with two tacticals, ORELSE and REPEAT, our composite tactic is just

REPEAT (GENTAC ORELSE DISCHTAC).

The tactic T_1 ORELSE T_2, when applied to G, acts like $T_1(G)$ unless this fails, in which case it acts like $T_2(G)$; the tactic REPEAT T, when applied to G, applies T to G, then to all the subgoals of T(G) and so on, until failure occurs or no subgoals remain. These tacticals have one line definitions in ML, and are easily seen to preserve validity.

At a higher level still, we may express full automatic proof methods as tactics. One such method is the resolution method due to Robinson. This is a complete proof method, at least for the pure first-order predicate calculus, but in the context of interactive proof it is wise to apply it in a controlled manner, just at those points in a strategy at which progress may be expected by routine logical methods such as instantiation of assumptions, modus ponens and the like. We shall not describe the tactic in detail, but we shall call it RESTAC when we later employ it in an example.

At a similar level is a simplification tactic, called SIMPTAC, based upon a collection of equational theorems of form

$$\Gamma \vdash t_1 = t_2,$$

where t_1 and t_2 are terms, possibly containing variables that may be instantiated to match t_1 to some subterm of a goal, which may then be simplified by substituting the corresponding instance of t_2.

All of these tactics, together with others based upon appropriate forms of induction and case analysis, are indubitably part of the standard repertoire of a mathematician. But towards the end of our list (certainly with simplification, induction and case analysis) we begin to meet tactics that take a different specialized form in each problem domain. To take an introverted example, proofs *about* a formal language typically employ a case analysis upon the leading connective of a formula, and the number and nature of the cases is clearly domain-dependent. Again, in some problem domains it is almost routine to use a combination of induction and case analysis, while in others (consider elementary topology) induction is of no use; a relevant induction rule may even not exist.

These observations suggest that in any realistic work with the machine as a proof assistant we expect to be working in a particular problem domain, or *theory* as we shall call it, and that when we specify the theory that concerns us the machine makes available to us all the data types, non-logical constants and axioms of that theory, together with theorems that have

already been proved in that theory; it should also allow us access to those tactics that we have previously defined either of general use or pertaining to the theory in question. Futhermore, almost every interesting applied theory is founded upon more primitive theories (called its ancestor theories), and while working in any theory we expect to have access to all material pertaining to its ancestors.

An important function of the proof assistant is therefore to keep our tower of theories properly organized, allowing us both to introduce new theories – by specifying their parents and their own new types, constants and axioms – and to work in any existing theory (not only those at the top of the tower) by proving new theorems in them.

In this work, *polymorphic* theories play a vital role. An example of such a theory is a theory of trees whose nodes are labelled by objects of arbitrary type, where theorems (which are also polymorphic) are proved without any assumption of the nature, i.e. the type, of the node-labelling objects. Such theorems may be instantiated (by an inference rule called *type instantiation*) later, during work on a daughter theory that involves trees with node labels of some particular type; for example, this type may be *integer* in a theory of tree-sorting.

In the next section we illustrate several of these points by giving an outline of an approach to a particular problem. Before we do so, we have to admit that the preceding exposition has oversimplified one detail of our methodology (this was deliberate, since the methodology is not clearly the best, and others are possible). In place of sequents (Γ,F) as goals, we adopt the slightly more complex form (Γ,F,S) for goals; here, S is a set of assumptions that are equational, and suitable for use as simplification rules by the tactic SIMPTAC. Other tactics are embellished by letting them add suitable equations to S; for example, the tactic DISCHTAC, which assumes an antecedent, is understood to add this antecedent to S as well as to Γ, in cases where it is equational (and satisfies other criteria).

Finally, we draw attention to a point of great pragmatic significance. Our metalanguage allows a user to write tactics – even invalid ones – with great freedom. But with this freedom he can by no means generate an event that which is not a theorem. This follows from the fact that events are objects of type *theorem*, and that the only operations for generating them are the basic inference rules (like GEN) and rules derived from them. This is a fine illustration of the security provided by a type discipline; indeed, without it we could not claim to present a viable methodology.

3. An example: a theorem about parsing

We suppose that we wish to investigate the performance of a particular parsing algorithm, which operates upon a list of symbols and produces a particular kind of tree known as a parse tree.

Since there are two polymorphic theories – the theory of lists and the theory of trees – relevant to our problem, we first discuss the creation of these theories, each as a daughter theory of the NULL or *pure* theory, which contains no non-logical types, constants or axioms. We suppose that the theory NULL possesses a type ONE, whose only proper member object is denoted by the constant symbol ( ) (which may be pronounced 'nothing').

To construct the theory called LIST, we first create a unary *type constructor*, also called LIST. This will allow us to discuss objects of type α LIST; greek letters are used for type variables,

and type constructors are postfixed to their argument(s). We then introduce two *constants*, with their polymorphic types:

$$\begin{aligned}&\text{NIL} &&: \alpha\ \text{LIST}\\ &\text{CONS} &&: \alpha \to \alpha\ \text{LIST} \to \alpha\ \text{LIST}\\ &\text{APPEND} &&: \alpha\ \text{LIST} \to \alpha\ \text{LIST} \to \alpha\ \text{LIST}\end{aligned}$$

(more constants may be useful, but we only need these for now). Finally, to establish the basis of the theory, we introduce axioms that characterize the LIST constructor by establishing the isomorphism

$$\alpha\ \text{LIST} \simeq \text{ONE} + \alpha \times \alpha\ \text{LIST}$$

and which characterize the constants, by stating (or entailing as simple corollaries) such equations as

$$\text{APPEND (CONS a x) y} = \text{CONS a (APPEND x y)}.$$

From this slender basis, many polymorphic theorems will follow while working in the theory LIST; other list-processing functions may be introduced as constants to enrich the theory.

Entirely analogously, the theory TREE can be created. We first introduce a binary type constructor TREE, characterized by the isomorphism

$$(\alpha,\beta)\text{TREE} \simeq \alpha + \beta \times (\alpha,\beta)\text{TREE} + \beta \times (\alpha,\beta)\text{TREE} \times (\alpha,\beta)\text{TREE},$$

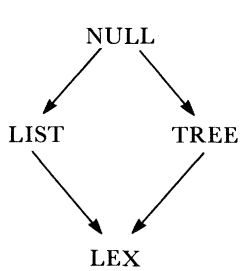
which states, in effect, that a tree is either an α object (a TIP labelled with an object of type α) or a node labelled with an object of type β and possessing either one or two son-trees. Then we naturally introduce as constants the three tree constructors

$$\begin{aligned}&\text{TIP} &&: \alpha \to (\alpha,\beta)\text{TREE}\\ &\text{NODE1} &&: \beta \to (\alpha,\beta)\text{TREE} \to (\alpha,\beta)\text{TREE}\\ &\text{NODE2} &&: \beta \to (\alpha,\beta)\text{TREE} \to (\alpha,\beta)\text{TREE} \to (\alpha,\beta)\ \text{TREE},\end{aligned}$$

which construct, from appropriate arguments, trees of the three respective kinds (just as NIL and CONS are the list constructors).

It is a particular strength of the underlying logic of LCF, which is due to pioneering work by Dana Scott, that from its induction rule (called computation induction) can be derived the rules of structural induction for both lists and trees. Each of these may then be inverted in the standard manner, described in the previous section, to form (for trees) the induction tactic TREEINDUCTAC, which we shall have occasion to use in our example.

With the theories LIST and TREE as parents, we are now ready to introduce the lexical theory, which we shall call LEX, in which we can state and prove our theorem about parsing. The ancestry graph of theories is shown here.



For LEX, we first postulate two arbitrary types ID and OP, for the variables and operators that occur in the symbol strings to be parsed. Since brackets '(' and ')' also occur as symbols, we then define the type isomorphism

$$\text{SYMB} \simeq \text{ONE} + \text{ONE} + \text{ID} + \text{OP} + \text{OP}.$$

The first two summands represent the left and right brackets, respectively, and the last two represent the distinguished occurrences of an operator as a unary and a binary function symbol respectively. We therefore introduce and axiomatize the five constructors

$$\begin{aligned}
&\text{LB} && : \text{SYMB} \\
&\text{RB} && : \text{SYMB} \\
&\text{VAR} && : \text{ID} \to \text{SYMB} \\
&\text{UNARY} && : \text{OP} \to \text{SYMB} \\
&\text{BINARY} && : \text{OP} \to \text{SYMB}.
\end{aligned}$$

We are now ready to give the parsing algorithm, expressed as a set of axioms about a function PARSE, whose type is

$$\text{PARSE} : \text{SYMB LIST} \to (\text{ID,OP})\text{TREE} \times \text{SYMB LIST}.$$

Thus, PARSE takes an arbitrary symbol list, which begins with a well formed formula, and produces from it a parse tree, representing the shortest initial segment of the argument that is parsable, together with the remainder of the argument string. Notice that our theory will instantiate polymorphic theorems about lists and trees, since, for example, a parse tree is a tree whose tips are labelled with identifiers in ID, and whose nodes are labelled with operators in OP.

Now the axioms for PARSE can be written (with a little syntactic sugar) as follows, where we use logical variables $x \in \text{ID}$, $f \in \text{OP}$, $s \in \text{SYMB LIST}$ and $t \in (\text{ID,OP})\text{TREE}$:

$$\begin{aligned}
&\text{PARSE (CONS(VAR x)s)} = \text{(TIP x,s)} \\
&\text{PARSE (CONS(UNARY f)s)} = \text{(NODE1 f t', s')} \\
&\qquad\qquad \textbf{where } \text{(t',s')} = \text{PARSE s} \\
&\text{PARSE (CONS LB s)} = \text{PARSETWO t' s'} \\
&\qquad\qquad \textbf{where } \text{(t',s')} = \text{PARSE s} \\
&\text{PARSETWO t (CONS(BINARY f)s)} = \text{(NODE2 f t t',CHECKRB s')} \\
&\qquad\qquad \textbf{where } \text{(t',s')} = \text{PARSE s} \\
&\text{CHECKRB (CONS RB s)} = \text{s}.
\end{aligned}$$

The sugaring here is the use of **where**, which is expressible by a simple logical combinator. Note that the action of PARSE depends upon the nature of the leading symbol of its argument. The detailed working of the parser need only concern the most assiduous reader; we intend mainly to illustrate the style in which an algorithm may be presented as a set of equational axioms.

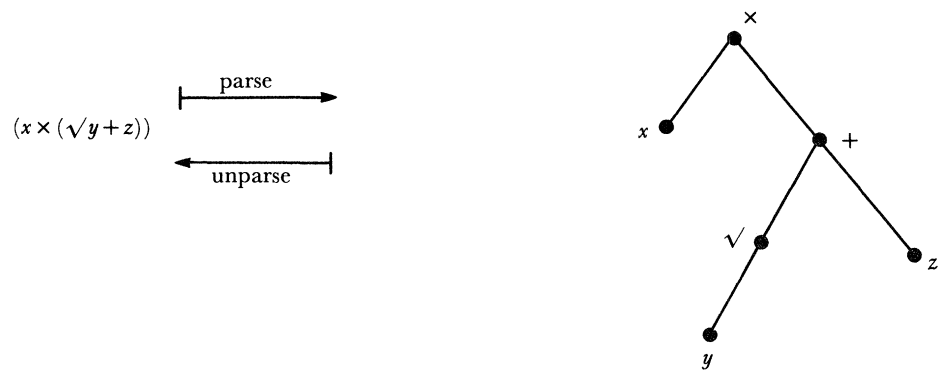
To express the property that we wish to prove, we also need a simpler function

$$\text{UNPARSE} : (\text{ID,OP})\text{TREE} \to \text{SYMB LIST},$$

which linearizes any parse tree into the symbol string that it represents. UNPARSE is axiomatized thus:

```
UNPARSE(TIP x) = CONS(VAR x)NIL
UNPARSE(NODE1 f t) = CONS(UNARY f) (UNPARSE t)
UNPARSE(NODE2 f t t') =
    APPEND(CONS LB(UNPARSE t))
           (APPEND(CONS(BINARY f) (UNPARSE t'))
                  (CONS RB NIL))).
```

The correspondence that we intend, between strings and parse trees, can be illustrated by a little example, in which $+, \times$ appear as binary operators and $\surd$ as a unary operator:



where we have ignored the 'tail' string also produced by PARSE.

Indeed, this correspondence is precisely the goal that we wish to prove; it may be expressed more exactly, and in general, as

$$\forall t\,.\,\forall s.\ \mathrm{WD}[t] \supset \mathrm{PARSE}(\mathrm{APPEND}(\mathrm{UNPARSE}\ t)s) = (t,s).$$

Here, the antecedent WD[t] is a formula (which we do not detail) with a free variable t, and expresses that the tree t is appropriately well defined. Let us call this formula F. Then the first step of tactical proof is to set up a goal

$$G = (\Gamma,F,S),$$

where Γ is empty and S is a set of simplification rules that includes simple properties of the basic functions over lists and trees, and all the axioms of PARSE and UNPARSE.

The intuition that the user must provide, for this problem, is that it is natural to attack it by structural induction upon trees, and that thereafter it should yield to a mixture of simplification (SIMPTAC) and routine logical manipulation (RESTAC). But other ingredients are needed in the complete strategy. First, because quantification and implication both occur in the goal formula, it is reasonable to expect some use of GENTAC and DISCHTAC. Second, the formula WD[t] (which we did not detail) is the subject of some simple lemmas, which would be applicable in many problems about trees and may reasonably be expected to have been proved in advance. It is worth noting that these lemmas, and many of the simplification rules in S, are theorems not of the theory LEX but of its parent theories LIST and TREE. This

illustrates how a proof in an interesting applied theory will often be stratified; it will rarely be conducted entirely within the theory in which it is stated.

In view of the above remarks, a reasonable interactive approach to the goal G would be to attack it with TREEINDUCTAC, followed perhaps by SIMPTAC, and then to inspect the resulting goals to see what further tactics are appropriate. This was the approach adopted when we first tackled the problem with LCF. When the theorem was successfully achieved by such means, it was observed that the combination of tactics used was remarkably similar to that which succeeded on other problems, in totally different problem domains. The difference was mainly in the particular induction rule used, in the particular auxiliary lemmas employed, and in the particular simplification rules embodied in the main goal G. This suggested that a widely applicable strategy could be expressed parametrically (the above particulars being supplied as parameters in each different problem). While we have not yet fully separated the general from the parametric ingredients in this strategy, it is informative to present the instance of it that works with no interaction whatever for our present problem, and exactly as it is written in ML. In doing so, we employ one further tactical THEN, which is of general use; the tactic T1 THEN T2 first applies T1 to a goal, then T2 to all subgoals produced. Note that THEN is clearly associative.

If we represent by L the set of auxiliary lemmas (pertaining to our antecedent formula WD[t]), then the complete strategy is expressed in ML as

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

All ingredients of this strategy have been discussed previously, except the first line, which merely has the effect of introducing the lemmas, so that when (eventually) RESTAC is applied to descendent goals it may employ these lemmas.

This discussion cannot, for lack of space, explore many finer points of strategy construction (for example, why has SIMPTAC been used here in exactly two places?), but it has illustrated that powerful strategies can find a pleasantly simple and legible form of expression. For a more detailed exposition of the parsing problem, see Cohn & Milner (1982). A point that deserves the strongest emphasis is that this proof methodology does not pursue the 'will-o-the-wisp' of magically all-powerful strategies; it merely represents a step towards the ability to capture whatever expertise we develop in particular problem domains, by providing a suitably expressive metalogical framework.

4. Some recent developments

The nature of the metalanguage ML is independent of the logic that is to be used. The richness of (meta)types in ML is such that any logic can be presented within it. Since it is proposed to use ML with a variety of logics, and since the language has evolved somewhat since its inception, it has become important to establish a standard design for it. A step towards this standard is presented in Milner 1983; though written by the present author, it represents the work of many interested researchers.

Concerning application, several studies have been made. We cite two recent examples of

proofs made in LCF with its original logic, which was also used for the example in the present paper. Michael Gordon has shown how to validate computer hardware design; he verifies that a simple but non-trivial complete computer meets its behavioural specification (Gordon 1983*a*, *b*). Stefan Sokolowski has formally demonstrated the soundness of a Hoare logic for reasoning about imperative programs (Sokolowski 1983). These two very different applications show that the original logic is quite appropriate for problems in computer science itself. Larry Paulson has enriched the logic in his work with Gordon at Cambridge; in particular, he has shown how the simplification mechanism can be rendered much more flexible and powerful by the use of tactical methods to compose simplification primitives in a variety of ways (Paulson 1983).

The ML framework, with its method of tactical engineering, has also been applied to other logics. At Göteborg, Sweden, a group have adapted LCF to implement the logic of intuitionistic type theory (Peterssen 1983). At Cornell University, Robert Constable and his colleagues are building PRL, an environment for computer assisted proof particularly in constructive mathematics; they employ tactics and tactic combination in an adventurous way, in particular to combine large tactics – such as a partial decision procedure for arithmetic – into still larger ones (Constable & Bates 1983).

Recently David Schmidt, at Edinburgh, has been studying the fundamental relation between inference rules and tactics (Schmidt 1984). Specifically, he addresses the question whether an inference rule may be represented directly by a tactic, rather than, as here, by a function (from theorems to theorems) from which in turn a tactic is derived.

It emerges from these various studies that the method of composing proof tactics, which is illustrated in this paper on a rather simple example, not only provides a means of communicating proof methods to a machine, and of tuning them to particular needs, but also presents to mathematicians and engineers a lucid way of communicating such methods among themselves.

References

Cohn, A. & Milner, R. 1982 On using Edinburgh LCF to prove the correctness of a parsing algorithm. *Internal rep.* no. CSR-113-82, Computer Science Dept, Edinburgh University.

Constable, R. L. & Bates, J. L. 1983 The nearly ultimate pearl. *Tech. rep.* no. TR 83–551, Computer Science Dept, Cornell University.

Gordon, M. 1983*a* LCF–LSM: a system for specifying and verifying hardward. *Tech. rep.* no. 41, The Computer Laboratory, Cambridge University.

Gordon, M. 1983*b* Proving a computer correct with the LCF–LSM hardware verification system. *Tech. rep.* no. 42, The Computer Laboratory, Cambridge University.

Gordon, M., Milner, R. & Wadsworth, C. 1979 Edinburgh LCF. *Lecture Notes in Computer Science*, vol. 78. Berlin: Springer-Verlag.

Milner, R. 1983 A proposal for standard ML. *Internal Rep.* no. CSR-157-83, Computer Science Dept, Edinburgh University.

Paulson, L. 1983 Higher order implementation of rewriting. *Sci. computer Programming* **3**, 119–149.

Peterssen, K. 1983 A programming system for type theory. *LPM Memo.* no. 21, Dept of Computer Science, Chalmers University of Technology, Göteborg, Sweden.

Schmidt, D. 1984 A programming notation for tactical reasoning. In *Proceedings of Seventh International Conference on Automated Deduction, Lecture Notes in Computer Science*, vol. 170, pp. 445–459.

Sokolowski, S. 1983 An LCF proof of soundness of Hoare's logic. *Internal rep.* no. CSR-146-83, Computer Science Dept, Edinburgh University.

Discussion

R. S. BIRD (*Programming Research Group, Oxford University, U.K.*). I have a question about induction. The major problem about inductive proofs is surely that of finding the right generalization of the induction hypothesis to enable the proof to go through. I have a suspicion that the parse–unparse problem Dr Milner presented was formulated in such a way as to incorporate the necessary generalization from the start. Is this suspicion correct and, if so, what can be done to systematize the search for generalizations of hypotheses?

R. MILNER. I agree that a major problem in inductive proof is to find a sufficiently general induction hypothesis, and confirm that the induction formula for the parse–unparse formula was found by us and not by the machine.

LCF was not designed to incorporate the kind of intelligence needed to find such induction formulae, which is an arbitrarily hard task. But it *was* designed to allow a user to communicate strategies for such tasks to the machine. Indeed, the pioneering work by Boyer and Moore, on finding induction formulae, consists of a composite strategy that can be written in ML quite conveniently.