How ML evlolved

How ML Evolved

Robin Milner

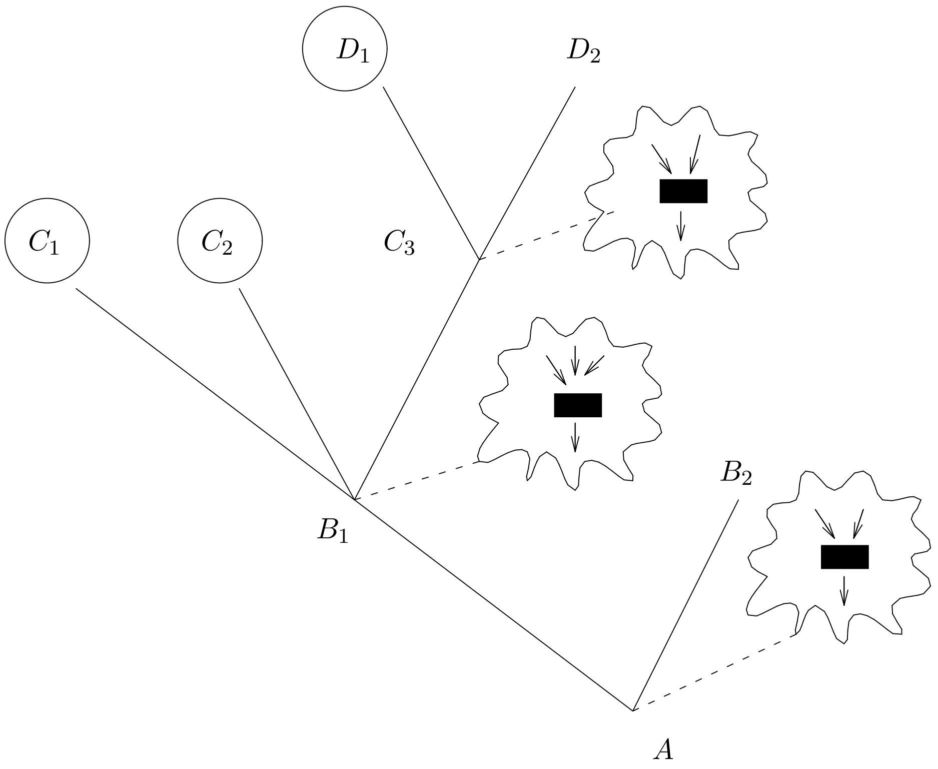
1982

ML is one among many functional programming languages. But not many were designed, as ML was, for a more-or-less specific task. The point of this note is to summarise the process by which we were guided to ML, as it now is, by the demands of the task. We (at least I) feel that to find a good metalanguage for machine assisted proof, which was the task, we could hardly have gone in an essentially different direction; the task seemed to determine the language—and even made it turn out to be a general purpose language.

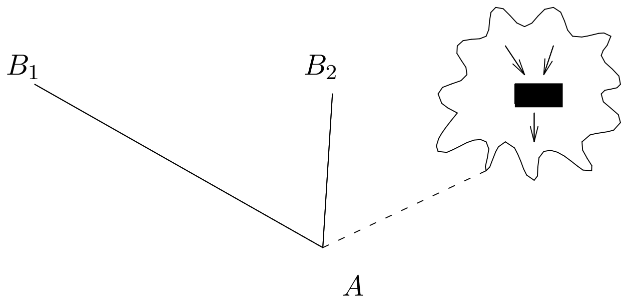
The context in 1974 (when the Edinburgh LCF project began) was our experience with the Stanford LCF proof assistant, developed there in 1971–72 by Richard Weyrauch, Malcolm Newey and myself. The development of ML as a metalanguage for interactive proof was the work of several people; in chronological order they were (besides me) Malcolm Newey, Lockwood Morris, Michael Gordon, and Christopher Wadsworth. Other people, who joined the LCF project after the language was more-or-less fixed, and used it for proofs, are Avra Cohn, Jacek Leszczylowski, David Schmidt, Larry Paulson and Brian Monahan. In the following summary of the development process, the actual logic involved (PPLAMBDA) is rather irrelevant, and it now seems that the same principles apply to any formal deductive system.

Consider the activity of goal-directed proof. If we have a goal A, a logical formula to be proved, then it is sound to replace it by subgoals $B_1, \ldots, B_n$, provided we know how to construct, from achievements of all the B_i, an achievement of A. We may call any subgoaling method a *tactic*; it is a *valid* tactic, then, if it also provides—for each set $\{ B_i \}$ of subgoals produced from a goal A—a way of extending achievements of the B_i into an achievement of A, and this "way" we call a *validation.* Of course, only valid tactics are useful!

Now, given a fixed repertoire of tactics, a natural proof assistant would be one which maintains a *goal tree*, with the user always working at a leaf. The Stanford LCF system was like this. After some subgoaling, the tree might look thus:

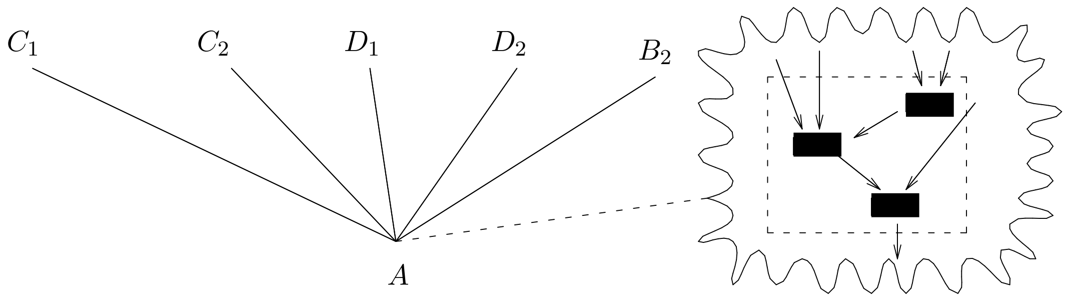


Here, we suppose that the ringed subgoals have (somehow) been achieved; the little black boxes sitting at the non-leaf nodes are the validations waiting to be applied to the (achieved) sons to achieve their father. So, after achieving D_2 the proof assistant would collapse the tree to:



Clearly, under this rigid discipline of tactical tree walking, the only way of proving the main goal is to return to the root, and the only doubt that the theorem has been correctly proved is in the correctness of the built-in programs for the basic repertoire of tactics and for treewalking.

But in a more flexible system, a user should at least be allowed to compose tactics into more powerful ones. For example, he should be allowed to compose the three tactics which were applied to goals A, B_1 and C_3 to produce the tree of our first diagram; this composite tactic applied to goal A would yield a flatter tree



Note that the validation produced by the composite tactic is a particular combination of those produced by the separate tactics. (Of course, as in our first diagrams, some of the subgoals may then be somehow achieved; we have omitted the rings here).

Well, to program this composition the user must be allowed to hold in his hand as objects (or: be allowed to bind as values to metavariables) both *goals* and validations; moreover, to put them together properly already suggests the need for structure-processing power of the kind found in LISP and other functional programming languages.

What kind of object is a validation? It is a "way of extending the achievements of subgoals into an achievement of the goal". But an achievement is (in our case) a *proof*, or the *theorem* which is the last step of a proof; so the validation for a tactic which produces subgoals $\{\ B_i\ \}$ from goal A could perhaps be represented by the theorem $\vdash B_1 \supset \cdots \supset B_n \supset A$, and to apply the validation is perhaps just to apply Modus Ponens repeatedly to this theorem and the achievements $B_1, \ldots, B_n$ to produce A. So perhaps validations are just theorems, proved somehow at the time that the tactic is applied?

To see that this is wrong, consider the tactic which converts a goal formula $\forall x.\, B(x)$ into a single subgoal formula, $B(x)$ (this is the common method of proving that something holds for *all* x by proving that it holds for *arbitrary* x). According to the above suggestions then, the validation should be the theorem $\vdash B(x) \supset \forall x.\, B(x)$, and there is *no such theorem*! In fact the validation should not be a *theorem*, but a *function from theorems to theorems*, i.e. a (primitive or derived) inference rule; in our case, it is the rule of generalisation

$$\text{GEN} \qquad \frac{\vdash B(x)}{\vdash \forall x.\, B(x)}$$

(This is why we called the tactic GENTAC).

We immediately see that a tactic is a *function producing* function; when applied to a goal it produces, as well as a subgoal list, a function which is the validation. So our metalanguage must express second order functions.

Further:

(1) when the validation function depends (as it may in general) upon the properties of the goal attacked, these properties will be bound into the validation at the time of tactic application, and the natural way of doing this critically requires the static binding convention, now normally accepted in preference to the dynamic binding of LISP. (In Landin's terms, the validation is a closure, i.e. an expression paired with an environment).

(2) Since the user is to be allowed to compose tactics (second order functions), his compositions will be third order functions; clearly a metalanguage which expresses functions of arbitrarily high order is the only natural choice.

(3) Since the user is to be allowed to hold validations (and, in general, any primitive or derived inference rule) in his hand, it is critically important that he is only allowed to apply them to *theorems*, not to other objects (such as formulae) which look so like theorems that in a moment of misguided inspiration he may mistake one for the other! So the metalanguage must be rigorously *typed*, in a way which at least distinguishes theorems from other things.

(4) For a given tactic T, there are usually goals for which it makes no sense to apply T (Example: it makes no sense to apply GENTAC to a goal formula which is not universally quantified). It would be vastly inconvenient to test a goal by some separate predicate before applying a tactic, so the tactic *itself* must assume the task of detecting inapplicability, and respond in some suitable way. But in a typed language, every *result* of applying a tactic must be a goal list paired with a validation, and it is irksome to have to construct a correctly typed but spurious 'result' when the application makes no sense; so the only alternative in this case is to have *no result at all.* Hence it is natural to have an *escape* or *failure* mechanism, under which senseless applications may avoid producing a result, but instead be detected for alternative action. In LISP this response could be to return the result NIL, for example.

(5) In a conversational typed functional language, it soon appeared intolerable to have to declare—for example—a new maplist function for mapping a function over a list, every time a new type of list is to be treated. Even if the maplist function could possess what Strachey

> called "parametric polymorphism" in an early paper, it also appeared intolerable to have to supply an appropriate type explicitly as a parameter, for each use of this function. (Perhaps this latter is rendered more acceptable if types can be suitably abbreviated by names, but note that even simple list constructors and destructors—CONS, CAR, CDR or whatever—would need explicit type parameters) so a polymorphic type discipline, with rigorous type checking, emerges as the most natural solution. Note that it emerges as such on purely *practical* considerations; it is a gift from the Gods that this discipline happens to have a simple semantic theory, and that the type checking has an elegant implementation based upon unification (Robinson). We only discovered afterwards that the proper lineage for this type checking is from Curry's functionality, through Roger Hindley's principal type schemes.

This discussion was somewhat simplified w.r.t. LCF (for example, LCF goals are not simply logical formulae to be proved). But it is, in essence, the process by which we arrived almost unavoidably at the metalanguage ML as it now exists. It shows why ML is a higher order functional programming language with rigorous polymorphic type discipline and an escape mechanism (and, of course, static binding).

Perhaps the development of ML has been made to seem too clean and trouble-free, from the above discussion. There still remain, however, some big problems and question marks. It is important to mention some of them; in doing so, we also place ML in context with other languages—both applicative and imperative.

(1) It is natural to recover, within ML, the simple treewalking proof methodology with which we started. But this global tree is a *changing* structure; as such, it cannot really be implemented without strain in a purely applicative language. (Or so it appears to me; this is a challenge to applicative language devotees who wish to rule out state change completely). So ML has an assignment statement!

    But this does not sit so easily beside the polymorphic type discipline. Recent work by Luis Damas (forthcoming Ph.D.) shows that the somewhat over-rigid treatment of the types of assignable variable in ML (viz. that they may not be polymorphic) can be relaxed; but the purity and obviousness of the discipline is inevitably lost.

(2) Even without assignment, the type discipline forced us to adopt a restrained form of escape mechanism in ML; it is not allowed to escape (or fail) with a value of arbitrary type, but only with a token. This problem has to do with the dynamic nature of the escape-trapping mechanism; the trap for each escape is not textually determined, and

it appeared to us most useful that it should not be so. A clean solution to this problem—probably easier than that for assignable variables—would be an important development.

(3) ML does not adopt the clausal form of function definition, which is found so convenient by users of HOPE and PROLOG. How can we get a semantically rigorous form of this clausal definition, in which the constructor-patterns in formal parameters can involve not only primitive constructors, but also the constructors of user-defined abstract types? The problem is to know that these constructors are constructors, in the sense of being uniquely decomposable (or else to admit non-determinism into the language). If I understand Rod Burstall right, this is partly why HOPE is called HOPE; the answer may eventually become CLEAR.

(4) ML does not use lazy evaluation; it calls by value. This was decided for no other reason than our inability to see the consequences of lazy evaluation for debugging (remember that we wanted a language which we could use rather than research into), and the interaction with the assignment statement, which we kept in the language for reasons already mentioned. In fact, this sharpens the challenge mentioned in (1) above; is there a good language in which lazy evaluation and controllable state-change sit well side-by-side? John Reynolds has for a long time worried about such possible incompatabilities between applicative and imperative languages (cf. his “Syntactic Control of Interference”, which exposes the problem with great honesty).

Conclusion

I hope this short essay has shown that machine-assisted proof provided a beautifully appropriate focus for developing functional programming and demonstrating its importance. It would be very useful for others to use this newsletter as a medium for reporting other real exercises in functional programming, so that a balance is kept between the seductive purity of functional languages and the methodology of their use. The remarks above tried to point out the considerable tension that exists between these two aspects of programming, and to show that it is not at all trivial to resolve the tension.