Dialogue with a Proof System

Robin Milner
Laboratory for Foundations of Computer Science
Computer Science Dept, University of Edinburgh
Edinburgh EH9 3JZ
UNITED KINGDOM

This short essay attempts to identify the ontology of the dialogue between a user and a machine in building or performing a formal proof. Our starting point is an assumption: that however non-rigorous, friendly and flexible is the interface between user and machine, the objects discussed are themselves precise - and that however loosely and summarily a proof is presented by either party to the other, the intended object is indeed a proof in the most formal sense. Under this assumption the interactive proof system may - and should - amplify the user's power to discover proofs and new results, and at the same time it must by its very construction be unable to assent to a theorem without also being able, if requested, to present its formal proof.

The view of computer-assisted reasoning expressed here arises partly from the author's own experience with proof systems, but has been greatly modified and enriched by the experience and ideas of many logicians and computer scientists, in particular Constable, Martin-Löf, Paulson, Plotkin and Schroeder-Heister. In fact many of the ideas informally expressed here have already been realised in the work on a unified Logical Framework by the current Edinburgh project on computer-assisted formal reasoning; for some of the ideas, however, it is not clear how best to incorporate them in the Logical Framework, so the ideas must be regarded as tentative to this extent.

Formalism is deliberately avoided in this essay. This appears to be strongly justified. If future proof systems are to be understood at all, then it must be possible to discuss them clearly but informally; the essay is therefore literally a 'trial' of how such a discussion should go.

It has become increasingly clear that the interface between user and machine must not only consist of powerful graphical and textual aids, but must - even more importantly - employ an ontology which is both large and well defined. This ontology, the objects of discussion, must include obvious things like formulae and proofs; it must also include more complex objects like strategies and logics. We take for granted that the graphical and textual medium must be very highly developed if the machine is to be of any assistance in difficult work, such as proving the correctness of large programs (indeed if the work is not difficult then a machine is probably redundant!). Our main focus is therefore upon the objects of discussion, and as we proceed we hope to show that they can be classified, by a type discipline, in

such a way as to form an intellectually coherent whole. This classification is essential for a user to conduct his proof work incrementally, in an environment (stored on files) containing the fruits of earlier work by himself and others.

The user wishes to build proofs in some logic; it is natural to suppose that he wishes to see his proof displayed on the screen as it grows. (Certainly there are other possibilities for what should be displayed, but the present supposition helps to focus our ideas and appears to yield a unifying discussion.) For the present paper, for simplicity, we shall consider a *proof* to be a tree in which each node is labelled with an *assertion*, and with the instance of the *inference rule* by which the assertion is inferred from the assertions labelling its sons. The root of the tree is labelled by the assertion proved; each leaf is labelled by an axiom instance, where an *axiom* is just a rule with no hypotheses. For Hilbert proof systems, the content of an assertion would be just a *formula*; for Gentzen systems it would be a *sequent*; what follows is not dependent upon the precise content of an assertion. We use "assertion" in the sense that Martin-Löf uses "judgment".

Thus proofs (i.e. proof trees) are objects of discussion. They are passed from machine to user (via screen or hard medium), and also from user to machine. In a simple proof building program the user's elementary activity is to construct a new root from given subtrees by invoking an inference rule, i.e. supplying a parameter for each schematic variable (e.g. formula variable) in the inference rule schema.

But we need something more general than a proof - that is, we need something which may contain uninstantiated schematic variables (e.g. formula variables) and thereby stands for a family of proofs; it is then better to talk of a *proof schema*. (The "proof" of an existentially quantified assertion can be considered as a proof schema containing an uninstantiated variable over terms - or perhaps over individuals.) A "proof" may also be partial; a leaf may be labelled by an assumed assertion. Partial proofs may be brought under the heading of proof schemata by labelling each assumption leaf by a *proof variable*, i.e. a variable which ranges over all possible proofs of the assumed assertion. Further, this idea allows proofs to be presented at a coarse level of detail, by labelling a node not with a rule instance but with (the name of) a partial proof; then one can "zoom in" on the node by replacing the node by the partial proof thus indicated.

By taking proof schemata, not just proofs, as objects of discussion the dialogue is greatly enriched. For a proof schema may not only be extended at the root, by inference; it may also be extended at a leaf by instantiation of the proof variable labelling the leaf. By progressive instantiation of schematic variables in a proof schema, the user can proceed from schema to proof in an orderly manner. The manner will be all the more orderly if all schematic variables are typed; this will guarantee that the ultimate proof, and all proof schemata used to achieve it, are well-formed. In the case of proof variables, the key idea is to take an assertion to be

the type of all its proofs, so that the type of a proof variable is the type of the proofs over which it ranges.

Proof schemata also induce a classification of the objects of discussion. An *assumption* is a proof schema with just a root, labelled by an assertion and by a proof variable whose type is the assertion. An *inference rule* is a proof schema consisting just of a root and zero or more assumption leaves; its other schematic variables include, for example, the formula variables appearing in the rule schema as normally presented in logic text-books.

A proof schema may contain uninstantiated variables, ranging over a variety of objects such as individuals, terms, formulae, assertions and proofs; thus it may be regarded as a function of such arguments. We are therefore led to consider *proof functions*, that is functions which produce proofs. A proof schema is rather a special kind of proof function just because it can be depicted as a tree containing variables which remain to be instantiated. An interesting case is the use of variables over partial proofs. We saw above that a node in a proof schema can be labelled by (the name of) a partial proof, to allow proofs to be presented at a coarse level of detail; if instead we label the node with a *variable* over partial proofs, then we represent a proof which is incomplete or partial at this node, and which can be rendered more complete by instantiating the variable, thus supplying the "missing" inference for which the variable stands. Note that this instantiation, together with "zooming in", still preserves the gross form of the main proof schema.

For the wider class of proof functions, tree-like depiction may be impossible, or at least unhelpful. An "automatic theorem prover" for example, is nothing more than a proof function (also called proof procedure) which takes an assertion A as an argument and yields a proof of A. (Since not all assertions have proofs, it is a partial proof function in general.) But the depiction of this proof function as a tree is impossible in general; the tree-like form of the resulting proof will be heavily dependent upon A, the argument. How do we fit such proof functions into a methodology in which displayed proof schemata play a significant role? Early theorem proving programs, though they were often powerful and were an indispensable step towards usable methodology, largely ignored this question. We speculate that a usable methodology must take a form which allows a user incrementally to develop the shape of a proof, and that the most promising way of achieving this is via progressive refinement of proof schemata.

At this point, it appears that the notion of *proof tactic*, or *partial proof strategy*, plays the vital mediating role. The original rather crude notion of proof tactic, which also provided as part of its result a "way of proving" the argument assertion from the list of result assertions as assumptions. Even with this crude notion of tactic, considerable methodological power was attained by the composition of tactics in a variety of ways to yield more powerful tactics, and by successive

application of tactics to the subgoals generated by previous tactics until the total list of unproved subgoals was empty; at this point, the "way of proving" could be invoked to yield a proof. The defect of this methodology is now clear; at no point until the very end was a proof schema visible - and then it was indeed a fully instantiated schema, i.e. a proof.

Within the present framework, it is also clear how to remove the defect. All that is necessary is to take a tactic to be a function from assertions to proof schemata; this includes proof procedure, or "automatic theorem power", as a special case in which the resulting schema is fully instantiated. (Note that this does make a tactic a proof function, since its result proof schema is a function which given further arguments yields a proof.) More precisely, a tactic takes as argument an assertion A and produces as result a schema whose root is labelled A. The strong typing of schemata ensures that the "way of proving" A from the leaf assumptions (the subgoals) is manifest in the schema - it exists by virtue of the schema's existence. By this means, then, the invocation of tactics is precisely the progressive refinement of proof schemata which we sought above.

It is most important to emphasize that the tactic itself is not a proof schema; we have introduced a distinction between proof schemata as those concrete objects which a user may treat as data objects - analysing them, editing them, displaying them and "zooming in" on them - and more general proof functions which he cannot treat in this way but which are none the less objects of discussion with a machine. Further, objects of still higher functionality arise naturally; the *tacticals* used in LCF, which were ways of composing tactics, are again definable as proof functions of higher type.

In all of the preceding discussion we have implicitly assumed a fixed object-language of assertions. (Here the term "object-language" is intended in the logician's sense of the syntactic material from which a logic is built.) We have also implied that the user and machine, in manipulating the object-language, require to express a wide range of entities such as formulae, assertions, trees, lists, and functions of rather sophisticated types. For this, a functional programming language seems to be ideal, and it should also provide a type discipline which ensures that the objects which can be built, particularly proof schemata, are well-formed even though they may be incomplete.

Once such a functional meta-language is employed, it is a short step to allowing it to handle a variety of logics. Indeed, a *logic* is just an object of a particular type in this meta-language. It consists first of an object language of assertions, and then of a collection of basic proof schemata over these formulae, i.e. the logical inference rules. To introduce a new logic to the machine, a user must therefore exploit the power of the meta-language to introduce new data types and functions over these types. Whatever rich class of proof functions may be definable in the meta-

language, a truly useful proof methodology will not emerge until the user is enabled to work in different logics at different times; further, he must even by allowed to compose logics via inference rules whose assumptions and conclusions are assertions of different logics, i.e. assertions of different types.

To meet this requirement, an interactive proof system must not only provide the user with such a meta-language; it must also provide a fixed set of elementary operations in this language by which inference rules are invoked. These operations exactly represent the uniform notion of inference which obtains in any logic which can be presented. At the present level of discussion it is not possible to say what these operations should be; the question is not simple, since it is crucially affected by the uniform way in which logics are presented. Another difficult question is whether tactics and strategies should indeed be expressed in the object language (as implied by our discussion), or whether they should be expressed in the functional meta-language as for example in the LCF proof system.

These questions constitute the frontier of research into proof systems. Once a good answer is obtained, it will be possible to integrate all computer-assisted reasoning within a single framework; thereafter, increasing power and convenience will be a matter of designing particular tools - graphic aids and proof strategies - for particular domains of reasoning within the framework. This specialised research will be lengthy and difficult, but will not suffer the same fate as most present proof systems which - because they have inherent limitations of formalism - reach a point where further extension requires total re-design.