Proof Style

John Harrison

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England

Abstract. We are concerned with how computer theorem provers should expect users to communicate proofs to them. There are many stylistic choices that still allow the machine to generate a completely formal proof object. The most obvious choice is the amount of guidance required from the user, or from the machine perspective, the degree of automation provided. But another important consideration, which we consider particularly significant, is the bias towards a 'procedural' or 'declarative' proof style. We will explore this choice in depth, and discuss the strengths and weaknesses of declarative and procedural styles for proofs in pure mathematics and for verification applications. We conclude with a brief summary of our own experiments in trying to combine both approaches.

1 What is a proof?

The word 'proof' is used in several different ways, and it is worth making clear from the outset the way in which we employ the word. With mechanized reasoning in mind, a proof may be:

1. What is found in a mathematical textbook, typically a sketch given in a mixture of formal symbols and natural language, taking for granted certain knowledge on the part of the reader (Trybulec and Święczkowska 1992).
2. A script to be presented to a machine for checking. This might also be a sketch in some sense, but is itself written in a formal (or at least non-natural) language. It may even be a program telling the machine how to construct a formal proof (Constable, Knoblock, and Bates 1985).
3. A proof object in a particular formal system, e.g. one of the standard axiomatizations of first order set theory or intuitionistic type theory.
4. A 'canonical proof' used as a theoretical idea in the explication of the intuitionistic meaning of the logical connectives.

We will normally use 'proof' in the second sense, i.e. a proof is a formal script to be presented to a machine for checking. Our title should be understood in this sense: we will focus on the style of such a proof. A reasonable goal in choosing a style is that our 'type 2' proofs should be almost as easy to read and write as proofs of type 1, while allowing the machine to construct a proof of type 3 from them. The former objective is likely to be necessary if we are ever to entice a large body of pure mathematicians to use our proof assistants. The latter lends our proofs the reliability that a real programme of formalization demands.

While we want to make the gap between 1 and 2 small, it is not essential that it be made completely negligible. The success of TEX and LATEX shows that mathematicians are prepared to learn to use tricky computer-based systems if they can see clear benefits, and perhaps, get a certain aesthetic or creative satisfaction. TEX and LATEX turn out, after a modest investment of effort, to be cheaper and more efficient than the traditional reliance on specially skilled secretaries. It's doubtful whether mechanized reasoning will become, in the near future, the most efficient way to produce a proof, but it may turn out to be the most efficient way to produce a *correct* proof.

2 Classification of proof styles

We can classify proof styles according to several criteria.

1. The level of automation the style admits. Provers differ greatly in how much they can prove, and how much they will try to prove, without user intervention.
2. The degree of user control afforded. A high level of automation is often felt to be in conflict with controllability, but there is no reason why this should be so. The SAM project (Guard, Oglesby, Bennett, and Settle 1969) was one of the earliest attempts to combine automation and interaction judiciously.
3. The potential for extensibility, i.e. whether and to what extent ordinary users can modify the proof style to suite their particular needs.
4. The emphasis on a declarative style, where the user merely states the facts to be proved and some general information about the method of solution, or on a procedural style where the user instructs the machine more or less explicitly.
5. Proof direction. Some systems insist that proofs must proceed forward from premisses to conclusion. Others insist that on the contrary they must proceed by refinement, from the conclusion to the premisses. Still others allow the two styles to be intermixed, or even allow a proof to be attacked from different places in a more or less arbitrary order, while retaining a clear idea about when all the extant proof obligations have been satisfied.
6. Whether proofs are processed interactively or via a script submitted to a batch process. The pioneering proof checking systems such as AUTOMATH (de Bruijn 1970) and Mizar (Trybulec 1978) were all batch-oriented, whereas most recent systems support an interactive style of working. Partly this is a matter of general fashion in computing, but we believe there are a few objective qualities of batch mode that deserve consideration.

The divide between 'declarative' and 'procedural' in computer science is a traditional and long-established one, arguably traceable back to the contrast between Gödel's and Turing's approaches to computability (Robinson 1994). However it is also notoriously hard to make sharp. We will not attempt to do so in our context, but we think the division is useful and thought-provoking, even if understood in a fairly weak and impressionistic way. Prolog, for example, is usually considered a declarative language, but it does nevertheless give full control information. The only real difference is that the control information is implicit, or perhaps more honestly, less obvious to a human reader. By the way, TEX and LATEX illustrate the distinction, with the latter being avowedly more declarative and less procedural.

In programming languages, there is a strong association between the phrases 'declarative' and 'high level'. Indeed, one might argue that a declarative style of proof, if it is to be successful, should feature a high level of automation. On this view, rather than the level of automation and the proof style being independent properties, automation is an essential 'enabling technology' for a good declarative style.

3 Some existing systems

Various positions on the automatic-manual and procedural-declarative axes are exemplified by some existing systems.

AUTOMATH (low automation, procedural)

AUTOMATH (de Bruijn 1970), perhaps the earliest 'proof checker', was used in some pioneering experiments by van Bentham Jutting (1977) in the formalization of mathematics. It provides low automation and the style of proof is procedural. The user does not have to provide a completely explicit formal proof, though the proof commands provided have limited power, and require the user to be fairly thorough.

Mizar (low automation, declarative)

Mizar (Trybulec 1978), of all present-day systems, has been used for the greatest amount of formalized mathematics. It provides a rather low degree of automation, and no user extensibility, but the automation has been chosen judiciously and supports a declarative proof style rather well. Proofs are structured according to a 'skeleton' which lays out the basic patterns of natural deduction steps. The intermediate parts are written in a linear sequence, indicating at all times from which hypotheses the current line follows, but giving no information about *how* it follows. Mizar is, in a sense, the only system supporting a proof style that is declarative and that really provides user control over the proof.

PVS (high automation, procedural)

PVS (Owre, Rushby, and Shankar 1992) is designed to provide cost-effective support for proofs in computer system verification. It is distinguished from its predecessor EHDM particularly in providing a high level of interactive control of the proof. At the same time, it provides a powerful suite of decision procedures, e.g. for linear arithmetic, which can make it easy to perform proofs in certain theories. The proof style is highly procedural, based on fairly high-level tactics for performing backward proof in sequent calculus.

NQTHM (high automation, declarative)

The NQTHM prover (Boyer and Moore 1979) represents an interesting extreme in two respects. First, it offers a high degree of automation, including powerful rewriting and linear arithmetic procedures together with the automation of induction proofs. It includes very few interactive features to guide the proof (the PC-NQTHM extension tries to remedy this problem). Many traditional provers for first order logic show similar characteristics in their respective domain.

However, the state of the art in automation falls well short (and will probably do so for the foreseeable future) of being able to prove many theorems of real mathematical substance without a great deal of help. As already stated, NQTHM offers no interactive features to provide user control. However it is possible to expand the knowledge base available to the prover by proving useful lemmas. In practice, then, one approaches a difficult theorem step by step via a series of carefully graded lemmas, each of which can be proved automatically given the previous lemmas.

Therefore NQTHM's proof style is almost exclusively declarative: one states what one wants proved, and the only procedural information about how to prove it consists in the series of hints given by the choice of previous lemmas, as well as some associated tags indicating how each one is to be used (e.g. as a rewrite, or an induction lemma). All the same, this proof style is only concerned with 'proof' in a fairly weak sense. Such a series of lemmas is unlike a conventional proof; perhaps it could be said to correspond to a textbook consisting largely of a series of graded exercises. (This would fit naturally with certain pedagogical approaches such as the Socratic or 'Moore' method, which emphasize student participation.)

LP (the middle way?)

The Larch Prover (Garland and Guttag 1991) occupies an interesting intermediate position. The level of automation provided is high in the area of equational reasoning (the system is descended from Reve, a pure term rewriting system), but low elsewhere. The proof style is mainly procedural, but in comparison with say HOL or PVS, contains some nods towards a declarative style. Proofs often proceed by explicitly quoting the current 'proof state' (i.e. the facts that have been established and any corresponding context), though these are usually interspersed with explicit proof methods. In fact it is possible for the proof method in a `prove` command to be omitted; Larch will then use one of the currently selected proof methods. However none of these methods are really as general as in Mizar; the only really powerful ones are concerned with equational reasoning, while, as with Mizar, there are no facilities for adding others.

4 Declarative and procedural proofs

To illustrate the distinction between declarative and procedural proofs, here are a couple of examples. Both are from HOL, but the first is done in a 'Mizar style' that we discuss later, and the second in a traditional procedural style using HOL tactics. The first proof is of the Knaster-Tarski fixpoint theorem that every monotone function on the complete lattice of subsets has a fixed point. The details of the syntax are not important for our present purposes, but let us note that `/\` represents conjunction ('and') and `==>` implication ('if …then …'), while `!` is the universal quantifier ('for all') and `?` the existential ('there exists').

```
 !f. (!x y. x <= y /\ y <= x ==> (x = y)) /\
     (!x y z. x <= y /\ y <= z ==> x <= z) /\
     (!x y. x <= y ==> f x <= f y) /\
     (!X. ?s:A. (!x. x IN X ==> s <= x) /\
                (!s'. (!x. x IN X ==> s' <= x) ==> s' <= s))
     ==> ?x. f x = x
proof
  let f be A->A;
  assume antisymmetry: (!x y. x <= y /\ y <= x ==> (x = y)) by L;
     and transitivity: (!x y z. x <= y /\ y <= z ==> x <= z) by L;
     and monotonicity: (!x y. x <= y ==> f x <= f y) by L;
     and greatest_lower_bound:
        (!X. ?s:A. (!x. x IN X ==> s <= x) /\
                   (!s'. (!x. x IN X ==> s' <= x) ==> s' <= s));
  set Y_def: Y = {b | f b <= b};
  Y_thm: !b. b IN Y = f b <= b by Y_def,IN_ELIM_THM,BETA_THM;
  consider a such that
      glb: (!x. x IN Y ==> a <= x) /\
           (!a'. (!x. x IN Y ==> a' <= x) ==> a' <= a)
      by greatest_lower_bound;
  take a;
  now let b be A;
      assume b_in_Y: b IN Y;
      then L0: f b <= b by Y_thm;
      a <= b by b_in_Y, glb;
      so f a <= f b by monotonicity;
      hence f a <= b by L0, transitivity;
      end;
  so Part1: f(a) <= a by glb;
  so f(f(a)) <= f(a) by monotonicity;
  so f(a) IN Y by Y_thm;
  so a <= f(a) by glb;
  hence thesis by Part1, antisymmetry;
end
```

Observe that the proof is structured to look like a textbook proof. Even though certain lines do have a procedural reading (e.g. '`let f be A->A`' as 'introduce a universal quantifier'), this is nicely kept implicit. Moreover each of the main assertions is not tagged with any proof method, merely with the other assertions from which it is deemed to follow (e.g. ‘`by monotonicity`’). Contrast the following proof, from the HOL real analysis library, that the composition of continuous functions is continuous:

```
let CONT_COMPOSE = prove
 (`!f g x. f contl x /\ g contl (f x) ==> (\x. g(f x)) contl x`,
  REPEAT GEN_TAC THEN REWRITE_TAC[contl; LIM; REAL_SUB_RZERO] THEN
  BETA_TAC THEN DISCH_TAC THEN X_GEN_TAC `e:real` THEN DISCH_TAC THEN
  FIRST_ASSUM(UNDISCH_TAC o assert is_conj o concl) THEN
  DISCH_THEN(CONJUNCTS_THEN MP_TAC) THEN
  DISCH_THEN(fun th -> FIRST_ASSUM(MP_TAC o MATCH_MP th)) THEN
  DISCH_THEN(X_CHOOSE_THEN `d:real` STRIP_ASSUME_TAC) THEN
  DISCH_THEN(MP_TAC o SPEC `d:real`) THEN ASM_REWRITE_TAC[] THEN
  DISCH_THEN(X_CHOOSE_THEN `c:real` STRIP_ASSUME_TAC) THEN
  EXISTS_TAC `c:real` THEN ASM_REWRITE_TAC[] THEN
  X_GEN_TAC `h:real` THEN DISCH_THEN(ANTE_RES_THEN MP_TAC) THEN
  ASM_CASES_TAC `&0 < abs(f(x + h) - f(x))` THENL
   [UNDISCH_TAC `&0 < abs(f(x + h) - f(x))` THEN
    DISCH_THEN(fun th -> DISCH_THEN(MP_TAC o CONJ th)) THEN
    DISCH_THEN(ANTE_RES_THEN MP_TAC) THEN REWRITE_TAC[REAL_SUB_ADD2];
    UNDISCH_TAC `~(&0 < abs(f(x + h) - f(x)))` THEN
    REWRITE_TAC[GSYM ABS_NZ; REAL_SUB_0] THEN DISCH_THEN SUBST1_TAC THEN
    ASM_REWRITE_TAC[REAL_SUB_REFL; ABS_0]]);;
```

Here almost all the steps are direct instructions to HOL as to how to proceed: repeatedly strip off universal quantifiers, rewrite with the following theorems, perform beta reduction, and so on. There is little declarative content.

5 Extensibility and full programmability

Present-day systems descended from LCF (Gordon, Milner, and Wadsworth 1979) attain an extreme of extensibility, in that a full Turing-complete programming language is available to write special proof procedures. The lack of such a language is felt to be a significant defect by many users of PVS, for example. Even this level of power is felt by some to be insufficient. Pollack (1995) has suggested using, instead of a conventional programming language like ML, a dependently typed metalanguage of total functions. This allows one to ensure that tactics always terminate and, in some sense, produce a correct result, merely by virtue of their type.

The use of a full programming language brings the ‘Java problem’[1] — one wants programming power without allowing the user to do anything dangerous, in this case produce false ‘theorems’. The LCF approach (Gordon, Milner, and

---

[1] Many standard browsers for the World Wide Web allow programs written in Java to be downloaded automatically and executed on the user’s machine, e.g. to perform sophisticated animation.

Wadsworth 1979) solves this difficulty by the use of an abstract datatype of theorems. However it brings other problems. The availability of a Turing-complete language requires, in the context of proof, many activities normally associated with programming. For example, designing an interface to a prover is harder, as it needs, *in extremis*, to support arbitrary programming. Finding errors in proof scripts may require the use of a full debugger.

Given that there are, therefore, disadvantages as well as advantages in having a full programming language, let us consider just what the uses of LCF-style programmability are.

1. First it is used to make substantial, difficult enhancements to the proof system. These are written only once and with a considerable expenditure of time and effort, but then become available to other users.
2. It is also used to write small nonce programs to automate currently tiresome problems. These tend to be ad hoc programs, thrown together quickly and seldom re-used even by their author.

Since the programs of the first kind are so widely used, they can be considered almost as a standard extension of the proof system. As such, they are not in such sharp conflict with the idea of a declarative proof style, especially when some of the most useful proof procedures allow one to replace an explicit proof with a casual indication such as 'by arithmetic'. However the second kind of program is more problematical. We speculate that the need for such programs results partly from a bad choice of existing primitives. (For example in HOL there are poor facilities for manipulating assumptions.) Were such problems remedied, we imagine there would be much less use of small ad hoc programs. They may be useful, though, in some verification proofs, e.g. performing what is effectively symbolic execution by proof.

The Coq system is a good illustration of an intermediate position consistent with the above remarks. It provides a fixed proof language for everyday use, whereas many LCF-style systems present the user with ML, the actual implementation language. Coq does however permit new proof commands to be written in ML à la LCF, and linked to names for new proof commands. Since the procedure is not completely trivial, this tends to discourage the development of small nonce programs, which as we have said, may be a good thing. However more experience is necessary to find out whether this is still the case for verification proofs. We are not sure whether most of those unhappy with PVS's primitive proof language are concerned about large extensions or small nonce programs.

6 Assessment of proof styles

How are we to assess the strengths and weaknesses of different proof styles? We will discuss several important considerations.

Writability

How easy is it to make a machine accept a valid proof? A typical problem with machine-checkable proofs is that they are long compared with their informal counterparts. de Bruijn (1970) suggests that the ratio tends to be more or less constant, but this was based on work translating a single, highly meticulous textbook (Landau 1930) into AUTOMATH. More recently Paulson and Grąbczewski (1996) report much more variable results, even within a single textbook. Significant though proof size is, it is not a direct indicator of how difficult a proof is to construct. Probably at least as important is how the structure of the proof script compares with (or aptly formalizes) that of an informal proof.

Since there often occur in proofs repeated patterns and 'clichés', it seems that extensibility is an important issue for writability (Pollack 1995). Just how important this is depends on the domain of application.

Readability

Actually, once a proof has been accepted by a machine, then there is a tendency, unless the proof has a particular wit or charm, rare in verification applications, to accept the theorem as true and forget about the proof. (At least if the proof checker is accepted as trustworthy.) Similarly, when a program appears to work, one is apt to treat it as a black box and forget its internal structure. But these tendencies are dangerous, for the same reason, and we shall discuss this in the next section.

Whatever the form of the proof, it is perfectly possible for the machine to produce a transcript of it in some form considered more suitable for human consumption. NQTHM includes facilities for producing a readable summary of the proof it finds. An early investigation of providing a readable form of HOL proofs is given by Cohn (1990); a prototype system for producing text from Coq proofs is described by Coscoy, Kahn, and Théry (1995), while the PVS prover is capable of generating summaries of important parts of its proofs.

Alternatively, users can intersperse formal proofs with a series of informal comments designed for humans, just as programmers can for their programs. One of the standard systems for 'literate programming' can be used to separate out the threads in a single document. However there are obvious attractions in making the formal proof submitted to the machine reasonably pleasant for people to read. Then the same document can serve as input to the computer and as a record of the proof for people; that is, we have 'self documenting' proofs. This document-centred view of computing is standard in, for example, WYSIWYG word processing. It also provides a link to the low-tech analogue of writing a proof with pen and paper.

What makes a proof pleasant to read is of course highly subjective. An obvious target is to make the proofs look similar to those found in mathematics texts. However this may not be suitable for all applications, e.g. in verification proofs involving special proof algorithms and very large terms. It is sometimes claimed that proofs are better presented using new techniques such as hierarchical structuring (Lamport 1993) or structured calculational proof (Back, Grundy, and von Wright 1996).[2] Moreover, restricting ourselves to a traditional textual display might be considered anachronistic, with new possibilities opened up by hypertext (Grundy 1996). We will not enter into these questions here, but content ourselves with noting that there are differences in readability on which all can agree.

Maintainability

Writing a machine-checkable proof is generally a lot of work, and one wants to be able to re-use as much as possible of this work in a related situation. For example, in verification work, the system specification or implementation may change slightly — how easy is it to modify the proof for the new situation? This has been considered by Curzon (1995). Again, suppose the infrastructure of the prover is enhanced, or that the foundational system supported is changed; similar questions arise. These problems are quite similar to those that come up when modifying existing software for a different situation, and give rise to many of the same pitfalls.

Readability is always helpful for maintainability, if only in providing orientation when a proof does break. Moreover, note that in verification, maintainability can be an essential component even of writing the proof in the first place. Commonly, one finds errors in one's intended specification or implementation, or at least in some lemmas or invariants, necessitating a rewrite of significant parts of the proof. This means that improvements in maintainability often have consequential benefits for writability. This might apply to proofs in pure mathematics too if the theorem prover were being used as an aid to proof discovery. However that is seldom the case at present; one usually starts with a clear and correct informal proof.

Efficiency of processing

We have focused on the difficulty of reading and writing the proof for humans. But what of the difficulty for the machine in checking a proof script? This can also cause inconvenience for people. One measure is the computational demand made, which translates into user waiting time. Another is implementation difficulty.

In the final analysis, this factor is decisive, since if a proof cannot feasibly be processed in a reasonable time, or a proof checker cannot reasonably be written, the whole business of mechanized reasoning grinds to a halt. However within reason, one can always accept a certain inefficiency, given that computers are still getting faster all the time. A moderate emphasis on human considerations is likely to be vindicated by technological progress.

---

[2] Actually, Lamport even suggests that hierarchical structuring gives a better way to write proofs in the first place.

Other factors

There are other properties that certain people may consider important. For example, it is often desirable to support the basic theorem prover with a suite of additional tools, including a convenient interface. The Mizar system has tools to find unnecessary assumptions and unused proof steps. The style of proof can have a significant impact on how easy it is to provide such tools.

For constructivists interested in extracting programs or answers from proofs, the level of user control afforded over the formal proof object eventually produced (or implicitly produced) may be highly significant. We will have little to say about this, since our own experience is in classical logic. However we should note that typically it is only a tiny part of a proof that is really interesting, even to a constructivist. For example, it has often been pointed out, e.g. by Kreisel (1985), that there is normally little point in constructivizing the proofs of universal lemmas at all, let alone in optimizing the proof structure.

7 Procedural versus declarative

What can we conclude about the merits of a procedural or declarative style from the experience of present-day systems? First let us say that, since our experience is necessarily limited, much of the following is anecdotal. In particular, we have more experience with procedural proof styles than with declarative ones, which gives us a sharpened awareness of the potential and the defects of the procedural. Moreover, it's worth noting that other factors can influence the suitability of a particular proof style — in particular the logical system supported. For example, type theory is often felt to be easier to mechanize than set theory, in that many proof obligations are automatable. However it may be that in a declarative style, or with a higher level of automation, this difference is much less significant. For example, Paulson (1990) was fairly negative about the suitability of set theory for mechanization, but in recent publications he is much more positive — perhaps the change can be attributed to the superior automation now available in the Isabelle system.

Writability

Tastes differ on what is easier to write, and once one has got used to a particular style, the taste tends to stick. We can at least say that declarative styles are easier for the beginner, simply because they tend to require a smaller 'vocabulary'. Even in highly automated theorem provers such as PVS, one needs to master a fairly large number of proof commands in order to avoid getting stuck. In HOL, with a panoply of low-level proof procedures together with a full (and probably unfamiliar) programming language, the situation is worse. In a declarative system, things are much easier; one just needs a little background knowledge about the general format of proofs, and thereafter merely has to make the proof script simple enough to be understood by the machine.

In the case of NQTHM, one can imagine that it is very easy to get started, since one just needs to type in the theorems to be proved. Of course, one needs to develop strong intuitions about the power of the prover in order to make effective use of the 'series of graded lemmas' approach. Sometimes it will be frustrating that there are no ways to direct the proof procedurally, and additions to NQTHM have been developed to meet these problems. Nevertheless it seems that with such highly declarative systems the learning curve is not as steep at the very beginning as it is with, say, HOL. At worst, one can proceed in such tiny steps that the automated prover is certain to plug the gaps.

We believe that in general, declarative proofs are closer to those found in mathematics texts. These normally have very little procedural content, and the reader is expected to plug 'obvious' gaps. (Most readers will have had the experience of finding certain gaps, thought by the author to be obvious, difficult or even impossible to fill.) A notable exception is the realm of classical geometry, where proofs often proceed via a series of constructions (drop a perpendicular, produce this line, . . .).

For verification proofs, a procedural style is often more appealing. In particular one can develop customized proof commands to deal with various situations, such as the symbolic execution of the system being modelled. These can also be directly parametrized, e.g. by the size of a machine word, whereas in a declarative style, this is difficult, usually necessitating manual editing.

Readability

We claim as the most striking advantage of declarative proofs their greater readability. In order to construct the proof state partway through a procedural script, one needs in general to *execute* all the previous steps. This is analogous to replaying a chess game in one's head given just the series of moves. Few enough people can even do that — how much more difficult to anticipate the result of running what are, in the final analysis, arbitrary programs on a fast computer. Certainly, each step is deterministic, and many of them are fairly straightforward. However proof commands often make apparently arbitrary choices, e.g. over the order in which to execute non-confluent rewrites, or which conditional expression to case-split over. (HOL's `RES_TAC` which performs undirected forward chaining, and PVS's `inst?` which instantiates quantified variables by finding a match in the rest of the sequent, are well known examples in the respective user communities.) And in general they can do so much computation that it is impossible in practice to visualize the result without actually running the proof script.

One possibility is to annotate the proof with intermediate steps, just as, to continue our analogy, one typically includes a few diagrams in the text of a chess game. However this gives a rather artificial separation between the parts of the proof intended for human consumption and those parts for the machine. Really, this is just a form of commenting, and we have already discussed this in general terms.

The above assessment was biased towards 'clean' proofs with a fairly abstract structure, e.g. those typically found in pure mathematics. The assessment may need to be reversed, however, for many verification proofs, where the terms involved are very large. Quoting such terms explicitly may be out of the question, and spotting the modest differences between nearby terms may be quite impractical, whereas a procedural style might help to focus the mind on which parts are being manipulated.

Even if one still wants a separate document giving a (more) palatable transcript of the proof, it seems that a declarative base document is a better starting point. One simple idea is simply to have the machine fill in some of the gaps that it can justify as 'obvious', merely making the user's proof outline more explicit while preserving its overall structure.

Maintainability

This is a thorny problem, and we are not yet in a position to make a clear assessment of whether procedural or declarative proofs are more maintainable, though anecdotal evidence tends to support declarative proofs. There are two aspects to maintainability: how likely proofs are to break (under various perturbations of the system and/or the problem) and how easy they are to fix once broken. For the latter, it seems clear that declarative proofs are better, because of the better readability we have already drawn attention to.

We believe that declarative proofs are certainly stabler under changes to the prover. In the extreme case of full automation, then provided the prover's power increases monotonically, proofs will never break. By contrast, in procedural scripts, quite small changes to a single proof command can be very troublesome, since these changes can propagate to the proof state following each instance of it, causing the rest of the script to become inappropriate. Fixing this then becomes a tedious debugging exercise.

For stability under changes to the problem, the situation is not quite so clear cut. In some sense, declarative proofs are more robust, since very often the proof is insensitive to small changes, such as reversing the order of disjuncts in a theorem. By contrast, this may dramatically alter the procedural script required. Against that, declarative proofs rely more heavily on explicit quotation of terms, and so can require alteration if the underlying terms change. However the fact that this information is explicit rather than (in the case of procedural proofs) implicit is arguably a strength rather than a weakness, since the changes can be effected by standard editing operations on the text. Most existing experience (Chen 1992; Gonthier 1996) supports the declarative style as yielding proofs that are easier to maintain.

Efficiency of processing

Here procedural proofs are definitely better. They instruct the computer precisely how to construct the proof, and cut out most search. Of course the basic procedural components may themselves indulge in a substantial amount of search, but at least they do not have to construct the basic structure of the given proof. By contract, search procedures often come to the fore in a declarative style.

One possibility to improve the efficiency of automatic proof procedures, whatever the proof style, is to process the proof into some intermediate form (perhaps not human-readable) containing more explicit information about the proof. This can then cut out a large part of the search when the proof is re-run. Of course it needs to be 'recompiled' if the original proof changes, but nevertheless it seems quite promising. Such an idea has been tried in PVS.

Tool support

Many of the properties making declarative proofs more readable also make them more suitable for support with other tools. For example, one often wants an interface to allow one to navigate about the proof freely. This requires the establishment of context at any given point in the script. Though not quite as difficult for the computer as for the user, this is still in general tedious and time-consuming, since it requires the intermediate proof steps to be executed. Moreover this has some negative consequences for modularity, because these support tools then become tied to the prover itself as an indispensable subroutine. Similar problems would apply if one tried to implement in a procedural system some of the supporting tools provided with Mizar. For example, to detect redundant hypotheses, one would need to re-execute steps in a variety of slightly modified situations. With a declarative style, one can even support computer-aided proof construction, where the computer makes explicit some of its automatic reasoning, or helps to lay out the proof like an advanced structure editor (Syme 1997a).

Style of working

Declarative proof scripts give a wider choice over how work is processed, again because of the relative ease of establishing context. It becomes a real possibility to process small parts of a large proof independently, just as one typically compiles small parts of a large program separately. This is more difficult in a procedural system, which therefore tends to impose a rather rigid structure on the order in which proofs are tackled. For example, to get round this problem in HOL, one often temporarily asserts lemmas using the loophole `mk_thm` in order to tackle the interesting parts first. Inevitably, mistakes creep in and may only be detected in a final 'sanity check'. With declarative proof scripts, the work can be dealt with in any order in a freer fashion.

Declarative styles also make a batch style of working more convenient (and this is the style supported by Mizar). Error recovery is easier, since even if a given proof step fails, the proof context can be recovered and useful errors given for the remainder of the script. Though interactive proof development is often attractive, particularly when feeling one's way towards a proof in an exploratory fashion, it is sometimes more convenient to write a proof sketch, submit it for checking, then refine it according to the error messages received. In particular, this is the standard model for programming language compilation, as well as TEX-style word processing, and so has a certain familiarity for many people. This could serve to reduce the 'culture shock' felt by many people when tackling mechanized reasoning systems for the first time.

What about automation?

It could be argued that the level of automation itself has as much importance as a declarative style for each of the above factors. For example, a high level of automation often has direct benefits for maintainability. This is true at least when the automated steps actually *solve* goals, but perhaps not when the highly automated parts are intermediate steps. Certainly, anecdotal evidence from users of Isabelle and PVS supports the view that, as users rely more heavily on powerful automation, their proofs become more robust. Moreover, proofs certainly become easier to write. However they can become hard to read if the prover is capable of making jumps of baffling complexity. For example, a tendency to insert extra rewrites into the background automatically may have considerable benefits for those writing a proof, at great detriment to its eventual readability. A balance needs to be struck between too much power and too little, the difficulty being that the notion of 'obviousness' differs fundamentally for people and for machines (Rudnicki 1987).

Conclusions

We have pointed out many advantages of a declarative style of working. Indeed, we believe this idea has been unduly neglected by the main stream of research in mechanized reasoning. (We exclude NQTHM, which belongs solidly to this main stream, because it hardly considers *proofs* in the conventional sense.) At the same time, the declarative style is not without its disadvantages. Intuitively one feels that many large verification proofs would be much easier in PVS or HOL than in a more declarative system such as Mizar. Roughly speaking, a declarative proof style is good for pure, abstract proofs, whereas a procedural one is good for big, ugly, concrete proofs. Occasionally this assessment may need reversing. For example Gonthier (1996) presents Larch proofs of a distributed garbage collection algorithm, and the proof scripts have a strongly declarative flavour. More recently, Syme (1997b) has proved type soundness for a subset of Java using a declarative system called `DECLARE` (Syme 1997a). On the other side of the coin, nowadays a number of mathematical results are established with the aid of computer checking (Lam 1990); it seems that a procedural proof style offers more possibilities of absorbing these results into mechanized reasoning.

8 The best of both worlds?

Since the merits of the two styles are not clear cut, the ideal is perhaps to have a free choice of both. We have experimented with supporting Mizar-style proofs in the HOL system (Harrison 1996). The objective is to combine the strengths of HOL (reliability, extensibility and interactivity) with those of Mizar (readability and declarative style).

True to our remarks on the subjectivity of the declarative/procedural divide, the method is to find a procedural reading for each Mizar construct within the HOL tactic mechanism. To bridge the 'obvious' gaps, the user can install arbitrary automated provers; a default prover is provided which is capable of basic first order reasoning and a few other limited functions. At the same time, a degree of procedural content can be injected into the Mizar/HOL proof scripts by explicitly indicating a particular automated prover, e.g. '`by rewriting with X`' or '`by arithmetic with Y`'. At the same time, conventional HOL tactics can be interspersed arbitrarily with Mizar constructs, even within a single proof.

This mixture of styles seems especially suitable for our own current research interest of verifying floating point algorithms, mainly for the transcendental functions. Such proofs often involve a mix of abstract pure mathematics and specialized procedures used to extract verification conditions, e.g. proving that loop invariants are maintained. To arrive at the right proof style, more experience of real proofs in a range of areas is the most valuable guide, together with a real willingness for system designers to learn from each other.

Related work

Chen (1992) contrasts a declarative and procedural style of proof (exemplified for him by Ontic and Nuprl respectively) and the desirability of combining the best features of each. Prasetya (1993) points out how the standard HOL proof styles differ from those in textbooks and discusses a prototype system to improve matters.

Acknowledgements

The importance of the declarative style of proof was brought home to me by Andrzej Trybulec and the success of his Mizar system. Many of the issues discussed in this paper were inspired by conversations with Donald Syme. In particular I owe to him many observations about the difficulty of establishing context in procedural proof scripts, and the issues connected with having a full programming language available. I'm also grateful to David Basin and Paul Jackson for some pointers to relevant literature. Richard Boulton, Mike Gordon, Michael Norrish and Mark Staples offered some valuable comments on this work, and in particular on the slippery distinction between procedural and declarative, while Konrad Slind's comments on the paper have improved several key parts, as have the comments of anonymous referees. Thanks also to Natarajan Shankar who pointed out that I originally wrote 'least upper bound' where I meant 'greatest lower bound' in the Mizar example. Many of those present at the TYPES'96 workshop provided valuable suggestions and advice; thanks to Christine Paulin-Mohring and the other organizers for giving me the opportunity to present these ideas. The work described was funded by the European Commission under the HCM scheme, and by the UK Engineering and Physical Sciences Research Council.

Glossary of systems

Here we will give some starting points for finding out more about the theorem proving systems mentioned in the text.

AUTOMATH The system is no longer used, but there is an extensive collection of papers (Nederpelt, Geuvers, and de Vrijer 1994) describing the system, its applications, and the underlying philosophy.

Coq The Coq system and its documentation can be found via the Web page `http://pauillac.inria.fr/coq/systeme_coq-eng.html`.

HOL HOL is described in a book by Gordon and Melham (1993), and the two versions of the system, as well as documentation and numerous papers, are available from `http://www.cl.cam.ac.uk/Research/HVG/HOL/index.html`. Of particular note is an extensive bibliography of HOL-related papers available as `http://www.dcs.glasgow.ac.uk/~tfm/hol-bib.html`.

Isabelle Isabelle is described in a book by Paulson (1994), and there is a Web page for the system: `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

Larch The Larch Prover is described by Garland and Guttag (1991), and the page `http://larch-www.lcs.mit.edu:8001/larch/LP/overview.html` has additional information.

LCF The original Edinburgh LCF project is described in a book by Gordon, Milner, and Wadsworth (1979), and the later Cambridge version by Paulson (1987). As far as we know, the system is no longer used, but Coq, HOL, Isabelle and Nuprl are all descended from it and follow the same general approach to proof.

Mizar A good overview of Mizar is given by Rudnicki (1992). There are some older papers describing the system by Trybulec (1978) and by Trybulec and Blair (1985). The system manuals are a good guide, but are mostly out of print. However the entire set of the *Journal of Formalized Mathematics* is devoted to Mizar formalizations, and this is now online at `http://mizar.uw.bialystok.pl/JFM/`. The Web page `http://web.cs.ualberta.ca:80/~piotr/Mizar/` is a good starting point.

Nuprl An older version of Nuprl is described in a book (Constable 1986); the current version and supporting documentation and bibliographic information is online at `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

Ontic Ontic seems not to be widely used now, but is described by McAllester (1989).

NQTHM NQTHM is described in the classic book by Boyer and Moore (1979); `http://www.cs.utexas.edu/users/moore/best-ideas/nqthm/index.html` is a short description of the system's history including a link to the FTP distribution. A newer prover called ACL2 supersedes NQTHM in most important respects, and `http://www.cs.utexas.edu/users/moore/acl2/index.html` includes the system, documentation and examples.

PVS PVS is described by Owre, Rushby, and Shankar (1992). The system and associated documentation and bibliographic data is available via the Web page `http://www.csl.sri.com/pvs.html`.

References

Back, R., Grundy, J., and von Wright, J. (1996) Structured calculational proof. Technical Report 65, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland. Also available as Technical Report TR-CS-96-09 from the Australian National University.

Boyer, R. S. and Moore, J S. (1979) *A Computational Logic*. ACM Monograph Series. Academic Press.

de Bruijn, N. G. (1970) The mathematical language AUTOMATH, its usage and some of its extensions. In Laudet, M., Lacombe, D., Nolin, L., and Schützenberger, M. (eds.), *Symposium on Automatic Demonstration*, Volume 125 of *Lecture Notes in Mathematics*, pp. 29–61. Springer-Verlag.

Chen, W. (1992) Tactic-based theorem proving and knowledge-based forward chaining. See Kapur (1992), pp. 552–566.

Cohn, A. (1990) Proof accounts in HOL (incomplete draft). Available on the Web as `http://www.cl.cam.ac.uk/users/mjcg/AccountsPaper.ps.gz`.

Constable, R. (1986) *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall.

Constable, R. L., Knoblock, T. B., and Bates, J. L. (1985) Writing programs that construct proofs. *Journal of Automated Reasoning*, **1**, 285–326.

Coscoy, Y., Kahn, G., and Théry, L. (1995) Extracting text from proofs. In Dezani-Ciancaglini, M. and Plotkin, G. (eds.), *Second International Conference on Typed Lambda Calculi and Applications, TLCA'95*, Volume 902 of *Lecture Notes in Computer Science*, Edinburgh, pp. 109–123. Springer-Verlag.

Curzon, P. (1995) Tracking design changes with formal machine-checked proof. *The Computer Journal*, **38**, 91–100.

Garland, S. J. and Guttag, J. V. (1991) A guide to LP, the Larch Prover. Technical report, MIT Laboratory for Computer Science.

Gonthier, G. (1996) Verifying the safety of a practical concurrent garbage collector. In Alur, R. and Henzinger, T. A. (eds.), *Proceedings of the 8th international conference on computer aided verification (CAV'96)*, Volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, pp. 462–465. Springer-Verlag.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Grundy, J. (1996) A browsable format for proof presentation. In Gefwert, C., Orponen, P., and Seppänen, J. (eds.), *Proceedings of the Finnish Artificial Intelligence Society Symposium: Logic, Mathematics and the Computer*, Volume 14 of *Suomen Tekoälyseuran julkaisuja*, pp. 171–178. Finnish Artificial Intelligence Society.

Guard, J. R., Oglesby, F. C., Bennett, J. H., and Settle, L. G. (1969) Semi-automated mathematics. *Journal of the ACM*, **16**, 49–62.

Harrison, J. (1996) A Mizar mode for HOL. In von Wright, J., Grundy, J., and Harrison, J. (eds.), *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, Volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, pp. 203–220. Springer-Verlag.

van Bentham Jutting, L. S. (1977) *Checking Landau's "Grundlagen" in the AUTOMATH System.* Ph. D. thesis, Eindhoven University of Technology. Useful summary in Nederpelt, Geuvers, and de Vrijer (1994), pp. 701–732.

Kapur, D. (ed.) (1992) *11th International Conference on Automated Deduction*, Volume 607 of *Lecture Notes in Computer Science*, Saratoga, NY. Springer-Verlag.

Kreisel, G. (1985) Proof theory and the synthesis of programs: Potential and limitations. In Buchberger, B. (ed.), *EUROCAL '85: European Conference on Computer Algebra*, Volume 203 of *Lecture Notes in Computer Science*, pp. 136–150. Springer-Verlag.

Lam, C. W. H. (1990) How reliable is a computer-based proof? *The Mathematical Intelligencer*, **12**, 8–12.

Lamport, L. (1993) How to write a proof. Research Report 94, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA.

Landau, E. (1930) *Grundlagen der Analysis.* Leipzig. English translation by F. Steinhardt: 'Foundations of analysis: the arithmetic of whole, rational, irrational, and complex numbers. A supplement to textbooks on the differential and integral calculus', published by Chelsea; 3rd edition 1966.

McAllester, D. A. (1989) *ONTIC: A Knowledge Representation System for Mathematics.* MIT Press.

Nederpelt, R. P., Geuvers, J. H., and de Vrijer, R. C. (eds.) (1994) *Selected Papers on Automath*, Volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

Owre, S., Rushby, J. M., and Shankar, N. (1992) PVS: A prototype verification system. See Kapur (1992), pp. 748–752.

Paulson, L. C. (1987) *Logic and computation: interactive proof with Cambridge LCF*, Volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Paulson, L. C. (1990) Isabelle: The next 700 theorem provers. In Odifreddi, P. G. (ed.), *Logic and Computer Science*, Volume 31 of *APIC Studies in Data Processing*, pp. 361–386. Academic Press.

Paulson, L. C. (1994) *Isabelle: a generic theorem prover*, Volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag. With contributions by Tobias Nipkow.

Paulson, L. C. and Grąbczewski, K. (1996) Mechanizing set theory: Cardinal arithmetic and the axiom of choice. *Journal of Automated Reasoning*, **17**, 291–323.

Pollack, R. (1995) On extensibility of proof checkers. In Dybjer, P., Nordström, B., and Smith, J. (eds.), *Types for Proofs and Programs: selected papers from TYPES'94*, Volume 996 of *Lecture Notes in Computer Science*, Båstad, pp. 140–161. Springer-Verlag.

Prasetya, I. S. W. B. (1993) On the style of mechanical proving. In Joyce, J. J. and Seger, C. (eds.), *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, Volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, pp. 475–488. Springer-Verlag.

Robinson, J. A. (1994) Logic, computers, Turing and von Neumann. In Furukawa, K., Michie, D., and Muggleton, S. (eds.), *Machine Intelligence 13*, pp. 1–35. Clarendon Press.

Rudnicki, P. (1987) Obvious inferences. *Journal of Automated Reasoning*, **3**, 383–393.

Rudnicki, P. (1992) An overview of the MIZAR project. Available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

Syme, D. (1997a) DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Syme, D. (1997b) Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Trybulec, A. (1978) The Mizar-QC/6000 logic information language. *ALLC Bulletin (Association for Literary and Linguistic Computing)*, **6**, 136–140.

Trybulec, A. and Blair, H. A. (1985) Computer aided reasoning. In Parikh, R. (ed.), *Logics of Programs*, Volume 193 of *Lecture Notes in Computer Science*, Brooklyn, pp. 406–412. Springer-Verlag.

Trybulec, Z. and Święczkowska, H. (1991-1992) The language of mathematical texts. *Studies in Logic, Grammar and Rhetoric, Białystok*, **10/11**, 103–124.

This article was typeset using the LaTeX macro package with the LLNCS2E class.